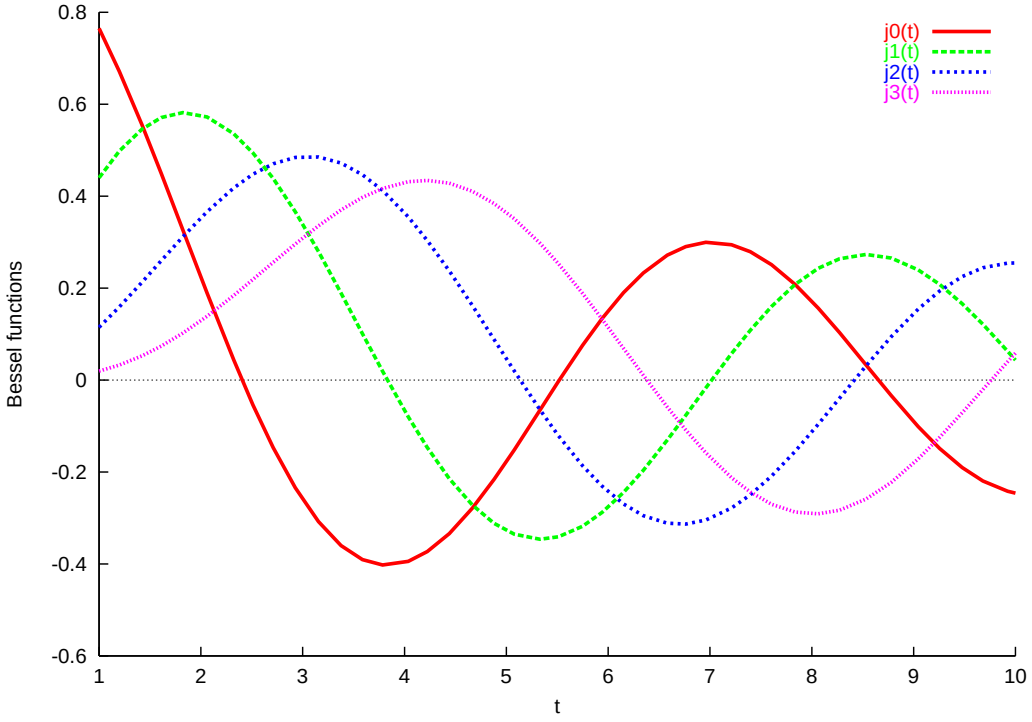
SoftIntegration®

The Ch Language Environment

Version 5.1

Reference Guide



How to Contact SoftIntegration

Mail	SoftIntegration, Inc. 216 F Street, #68 Davis, CA 95616
Phone	+ 1 530 297 7398
Fax	+ 1 530 297 7392
Web	http://www.softintegration.com
Email	info@softintegration.com

Revision 5.1.0, March 2006

Preface

Ch is a C compatible cross-platform scripting language environment. Ch is also a C virtual machine and a superset of C interpreter with salient features from C++, other languages, and software packages. Descriptions of functions and classes of Ch are given in this reference guide. The header files in the distribution of the Ch language environment are listed in Table 1. Functions not explained in this reference manual follow the interpretation of ISO and de facto standards such as ISO C and POSIX standards.

Typographical Conventions

The following list defines and illustrates typographical conventions used as visual cues for specific elements of the text throughout this document.

- Interface components are window titles, button and icon names, menu names and selections, and other options that appear on the monitor screen or display. They are presented in boldface. A sequence of pointing and clicking with the mouse is presented by a sequence of boldface words.

  Example: Click **OK**

  Example: The sequence **Start->Programs->Ch5.0->Ch** indicates that you first select **Start**. Then select submenu **Programs** by pointing the mouse on **Programs**, followed by **Ch5.0**. Finally, select **Ch**.

- Keycaps, the labeling that appears on the keys of a keyboard, are enclosed in angle brackets. The label of a keycap is presented in typewriter-like typeface.

  Example: Press `<Enter>`

- Key combination is a series of keys to be pressed simultaneously (unless otherwise indicated) to perform a single function. The label of the keycaps is presented in typewriter-like typeface.

  Example: Press `<Ctrl><Alt><Enter>`

- Commands presented in lowercase boldface are for reference only and are not intended to be typed at that particular point in the discussion.

  Example: "Use the **install** command to install..."

  In contrast, commands presented in the typewriter-like typeface are intended to be typed as part of an instruction.

  Example: "Type `install` to install the software in the current directory."

- Command Syntax lines consist of a command and all its possible parameters. Commands are displayed in lowercase bold; variable parameters (those for which you substitute a value) are displayed

Table 1: Library summary.

Header file	Description	C	POSIX	Ch
aio.h	Asynchronous input and output		X	X
arpa/inet.h	Internet operations			X
array.h	Computational arrays			X
assert.h	Diagnostics	X	X	X
chplot.h	2D and 3D plotting			X
chshell.h	Ch Shell functions			X
complex.h	Complex numbers	X		X
cpio.h	Cpio archive values			X
crypt.h	Encryption functions			X
ctype.h	Character handling	X	X	X
dirent.h	Format of directory entries		X	X
dlfcn.h	Dynamically loaded functions			X
errno.h	Error numbers	X	X	X
fcntl.h	Interprocess communication functions		X	X
fenv.h	Floating-point environment	X		X
float.h	platform-dependent floating-point limits	X	X	X
glob.h	Pathname pattern-matching types			X
grp.h	Group structure		X	X
inttypes.h	Fixed size integral types	X		X
iostream.h	input and output stream in C++ style			X
iso646.h	Alternative spellings	X		X
libintl.h	Message catalogs for internationalization			X
limits.h	platform-dependent integral limits	X	X	X
locale.h	Locale functions	X	X	X
malloc.h	Dynamic memory management functions			X
math.h	Mathematical functions	X	X	X
mqueue.h	Message queues		X	X
netconfig.h	Network configuration database			X
netdb.h	Network database operations			X
netdir.h	Name-to-address mapping for transport protocols			X
netinet/in.h	Internet Protocol family			X
new.h	Memory allocation error handling in C++ style			X
numeric.h	Numerical analysis			X
poll.h	Definitions for the poll() function			X
pthread.h	Threads		X	
pwd.h	Password structure		X	X
re_comp.h	regular-expression-matching functions for re_comp()			X
readline.h	Readline function			X
regex.h	regular-expression-matching types			X
sched.h	execution scheduling		X	X
semaphore.h	Semaphore functions		X	X
setjmp.h	Non-local jumps	X	X	X

Table 1: Library summary (continued).

Header file	Description	C	POSIX	Ch
signal.h	Signal handling	X	X	X
stdarg.h	Variable argument lists	X	X	X
stdbool.h	Boolean numbers	X		X
stddef.h	Miscellaneous functions and macros	X	X	X
stdint.h	Integer types	X	X	X
stdio.h	Input and output	X	X	X
stdlib.h	Utility functions	X	X	X
string.h	String functions	X	X	X
stropts.h	streams interface			X
sys/acct.h	Process accounting			X
sys/fcntl.h	Control file			X
sys/file.h	Accessing the file struct array			X
sys/ioctl.h	Control device			X
sys/ipc.h	Interprocess communication access structure			X
sys/lock.h	Locking processes			X
sys/mman.h	Memory management declarations		X	X
sys/msg.h	Message queue structures			X
sys/procset.h	Set processes			X
sys/resource.h	XSI resource operations			X
sys/sem.h	Semaphore facility			X
sys/shm.h	Shared memory facility			X
sys/socket.h	Internet Protocol family			X
sys/stat.h	File structure function		X	X
sys/time.h	Time types			X
sys/times.h	File access and modification times structure		X	X
sys/types.h	Data types		X	X
sys/uio.h	Vector I/O operations			X
sys/un.h	Unix-domain sockets			X
sys/utsname.h	System name structure		X	X
sys/wait.h	Evaluating exit statuses		X	X
syslog.h	System error logging			X
tar.h	Extended tar definitions			X
termios.h	Define values for termios		X	X
tgmath.h	Type-generic mathematical functions	X		X
time.h	Time and date functions	X	X	X
tiuser.h	Transport layer interface			X
unistd.h	System and process functions		X	X
utime.h	Access and modification times structure		X	X
wait.h	Wait for child process to stop or terminate			X
wchar.h	Multibyte I/O and string functions	X		X
wctype.h	Multibyte character class tests	X		X

Note: the symbol ‘X’ indicates that the library is supported by the standard.

in lowercase italics; constant parameters are displayed in lowercase bold. The brackets indicate items that are optional.

Example: **ls** [**-aAbcCdfFgilLmnopqrRstux1**] [*file ...*]

- Command lines consist of a command and may include one or more of the command's possible parameters. Command lines are presented in the typewriter-like typeface.

  Example: `ls /home/username`

- Screen text is a text that appears on the screen of your display or external monitor. It can be a system message, for example, or it can be a text that you are instructed to type as part of a command (referred to as a command line). Screen text is presented in the typewriter-like typeface.

  Example: The following message appears on your screen

```
usage:  rm [-fiRr] file ...


ls [-aAbcCdfFgilLmnopqrRstux1] [file ... ]
```

- Function prototype consists of return type, function name, and arguments with data type and parameters. Keywords of the Ch language, typedefed names, and function names are presented in boldface. Parameters of the function arguments are presented in italic. The brackets indicate items that are optional.

  Example: **double derivative**(**double** (**func*)(**double**), **double** *x*, ... [**double** **err*, **double** *h*]);

- Source code of programs is presented in the typewriter-like typeface.

  Example: The program **hello.ch** with code

```
int main() {
    printf("Hello, world!\n");
}
```

  will produce the output `Hello, world!` on the screen.

- Variables are symbols for which you substitute a value. They are presented in italics.

  Example: module *n* (where *n* represents the memory module number)

- System Variables and System Filenames are presented in boldface.

  Example: startup file **/home/username/.chrc** or **.chrc** in directory `/home/username` in Unix and **C:\ >_chrc** or **_chrc** in directory C:\ > in Windows.

- Identifiers declared in a program are presented in typewriter-like typeface when they are used inside a text.

  Example: variable `var` is declared in the program.

- Directories are presented in typewriter-like typeface when they are used inside a text.

  Example: Ch is installed in the directory `/usr/local/ch` in Unix and `C:/Ch` in Windows.

- Environment Variables are the system level variables. They are presented in boldface.

  Example: Environment variable **PATH** contains the directory /usr/ch.

Other Relevant Documentations

The core Ch documentation set consists of the following titles. These documentation come with the CD and are installed in CHHOME/docs, where CHHOME is the Ch home directory.

- *The Ch Language Environment — Installation and System Administration Guide*, version 5.0, SoftIntegration, Inc., 2005.

  This document covers system installation and configuration, as well as setup of Ch for Web servers.

- *The Ch Language Environment, — User's Guide*, version 5.0, SoftIntegration, Inc., 2005.

  This document presents language features of Ch for various applications.

- *The Ch Language Environment, — Reference Guide*, version 5.0, SoftIntegration, Inc., 2005.

  This document gives detailed references of functions, classes and commands along with sample source code.

- *The Ch Language Environment CGI Toolkit User's Guide*, version 3.5, SoftIntegration, Inc., 2003.

  This document describes Common Gateway Interface in CGI classes with detailed references for each member function of the classes.

Table of Contents

Chapter 1

Diagnostic Test — <assert.h>

The header **assert.h** defines the **assert** macro and refers to another macro,

NDEBUG

which is *not* defined by **assert.h**. If **NDEBUG** is defined as a macro at the point in the source file where **assert.h** is included, the **assert** macro is defined simply as

 # define assert (ignore) ((void)0)

The **assert** macro is redefined according to the current state of **NDEBUG** each time that **assert.h** is included.

Macros

The following macros are defined by the **asserth** header file.

Macro	Description
assert	Inserts diagnostic tests into program.

Portability

This header has no known portability problem.

assert

Synopsis
#include <**assert.h**>
void assert(**scalar** *expression*);

Purpose
Insert diagnostic tests into program.

Return Value
The **assert** macro returns no value.

Parameters
expression argument.

Description
The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if *expression* (which shall have a scalar type) is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the arguement, the name of the source file, the source line number, and the name of the enclosing function – the latter are respectively the values of the preprocessing macro **__FILE__** and **__LINE__** and of the identifier **__func__**) on the standard output. It then calls the **abort** function.

Example

```
/* a example for assert(). The program will
abort while i < 0 and display error message.*/
#include <assert.h>

int main() {
    int i;
    for(i = 5; ; i--)
    {
       printf("i= %d\n", i);
       assert(i >= 0);
    }
}
```

Output

```
i= 5
i= 4
i= 3
i= 2
i= 1
i= 0
i= -1
Assertion failed: i >= 0, file assert_func.c, line 10, function main()
```

See Also
abort()

Chapter 2

2D and 3D plotting <chplot.h>

The high-level plotting features described in this chapter is available only in Ch Professional Edition. The header file **chplot.h** contains the definition of the plotting **CPlot** class, defined macros used with **CPlot**, and definitions of high-level plotting functions that are based on **CPlot** class .

The plotting class **CPlot** allows for high-level creation and manipulation of plots within the Ch language environment. **CPlot** can be used directly within many types of Ch programs, including applications, function files, and CGI programs. Plots can be generated from data arrays or files, and can be displayed on a screen, saved in a large number of different file formats, or generated as a stdout stream in png or gif file format on the Web.

The high-level plotting functions **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyf**(), **plotxyz**(), and **plotxyzf**() are designed to be easy to use and allow plots to be created quickly. These functions can be used in conjunction with the **CPlot** member functions to create more sophisticated plots.

CPlot Class

The **CPlot** class can be used to produce two dimensional (2D) and three dimensional (3D) plots through a Ch program. Member functions of the **CPlot** class generate actual plots using a plotting engine. Gnuplot is used internally as the plotting engine for display in this release of the Ch language environment.

Public Data

None.

Public Member Functions

Function	Description
CPlot()	Class constructor. Creates and initializes a new instance of the class.
~**CPlot**()	Class destructor. Frees memory associated with a instance of the class.
arrow()	Add an arrow to a plot.
autoScale()	Enable or disable autoscaling of plot axes.
axis()	Enable or disable drawing of x-y axis on 2D plots.
axisRange()	Set the range for a plot axis.

border()	Enable or disable drawing of a border around the plot.
borderOffsets()	Set plot offsets of the plot border.
boundingBoxOrigin()	Set the location of the origin for the bounding box of the plot.
changeViewAngle()	Change the view angles for a 3D plot.
circle()	Add a circle to a 2D plot.
contourLabel()	Enable or disable contour labels for 3D surface plots.
contourLevels()	Set contour levels for 3D plot to be displayed at specific locations.
contourMode()	Set the contour display mode for 3D surface plots.
coordSystem()	Set the coordinate system for a 3D plot.
data2D()	Add one or more 2D data sets to an instance of the **CPlot** class.
data2DCurve()	Add a set of data for 2D curve to an instance of the **CPlot** class.
data3D()	Add one or more 3D data sets to an instance of the **CPlot** class.
data3DCurve()	Add a set of data for 3D curve to an instance of the **CPlot** class.
data3DSurface()	Add a set of data for 3D surface to an instance of the **CPlot** class.
dataFile()	Add a data file to an instance of the **CPlot** class.
deletePlots()	Remove any data from a previously used instance of the **CPlot** class and reinitialize option to default values.
dimension()	Set plot dimension to 2D or 3D.
displayTime()	Display the current time and date on the plot.
getLabel()	Get the label of an axis.
getOutputType()	Get the output type of a plot.
getSubplot()	Get a pointer to an element of a subplot.
getTitle()	Get the title of the plot.
grid()	Enable or disable display of a grid.
isUsed()	Test if an instance of the **CPlot** class has been used.
label()	Set axis labels.
legend()	Add a legend for a data set.
legendLocation()	Specify the plot legend location.
line()	Add a line to a plot.
margins()	Set plot margins.
outputType()	Set the plot output type.
plotType()	Set the plot type.
plotting()	Produce a plot from an instance of the **CPlot** class.
point()	Add a point to a plot.
polarPlot()	Set a 2D plot to use the polar coordinate system.
polygon()	Add a polygon to a plot.
rectangle()	Add a rectangle to a 2D plot.
removeHiddenLine()	Enable or disable hidden line removal for 3D plots.
scaleType()	Set the axis scale type for a plot.
showMesh()	Enable or disable display of mesh of a 3D plot.
size()	Change the size of a plot.
size3D()	Change the size of a 3D plot.
sizeRatio()	Change the aspect ratio of a plot.
subplot()	Create a group of subplots.
text()	Add a text string to a plot.
tics()	Enable or disable display of axis tics.
ticsDay()	Set axis tic-mark labels to days of the week.
ticsDirection()	Set the direction in which axis tic-marks are drawn.

ticsFormat()	Set the number format for tic labels.
ticsLabel()	Set location and text label for arbitrary axis labels.
ticsLevel()	Set the z-axis offset for drawing of tics in 3D plots.
ticsLocation()	Specify the location of axis tic marks to be on the border or the axis.
ticsMirror()	Enable or disable display of axis tics on the opposite axis.
ticsMonth()	Set axis tic-mark labels to months.
title()	Set the plot title.

Macros

The following macros are defined for the **CPlot** class.

Macro	Description
PLOT_ANGLE_DEG	Select units in degree for angular values.
PLOT_ANGLE_RAD	Select units in radian for angular values.
PLOT_AXIS_X	Select the x axis only.
PLOT_AXIS_XY	Select the x and y axes.
PLOT_AXIS_XYZ	Select the x, y, and z axes.
PLOT_AXIS_Y	Select the y axis only.
PLOT_AXIS_Z	Select the z axis only.
PLOT_BORDER_BOTTOM	The bottom of the plot.
PLOT_BORDER_LEFT	The left side of the plot.
PLOT_BORDER_TOP	The top of the plot.
PLOT_BORDER_RIGHT	The right side of the plot.
PLOT_BORDER_ALL	All sides of the plot.
PLOT_CONTOUR_BASE	Draw contour lines for a surface plot on the x-y plane.
PLOT_CONTOUR_SURFACE	Draw contour lines for a surface plot on the surface.
PLOT_COORD_CARTESIAN	Use a Cartesian coordinate system for a 3D plot.
PLOT_COORD_CYLINDRICAL	Use a cylindrical coordinate system in a 3D plot.
PLOT_COORD_SPHERICAL	Use a spherical coordinate system in a 3D plot.
PLOT_GRID_POLAR	Draw a polar grid in a 2D plot.
PLOT_GRID_RECTANGULAR	Draw a rectangular grid in a 2D plot.
PLOT_OFF	Flag to disable an option.
PLOT_ON	Flag to enable an option.
PLOT_OUTPUTTYPE_DISPLAY	Display the plot on a screen.
PLOT_OUTPUTTYPE_FILE	Output the plot to a file.
PLOT_OUTPUTTYPE_STREAM	Output the plot as a stdout stream.
PLOT_PLOTTYPE_DOTS	Use dots to mark each data point.
PLOT_PLOTTYPE_FSTEPS	Adjacent points are connected with two line segments, one from (x1,y1) to (x1,y2), and a second from (x1,y2) to (x2,y2).
PLOT_PLOTTYPE_HISTEPS	The point x1 is represented by a horizontal line from ((x0+x1)/2,y1) to ((x1+x2)/2,y1). Adjacent lines are connected with a vertical line from ((x1+x2)/2,y1) to ((x1+x2)/2,y2).
PLOT_PLOTTYPE_IMPULSES	Display vertical lines from the x-axis (for 2D plots) or

	the x-y plane (for 3D plots) to the data points.
PLOT_PLOTTYPE_LINES	Data points are connected with a line.
PLOT_PLOTTYPE_LINESPOINTS	Markers are displayed at each data point and connected with a line.
PLOT_PLOTTYPE_POINTS	Markers are displayed at each data point.
PLOT_PLOTTYPE_STEPS	Adjacent points are connected with two line segments, one from (x1,y1) to (x2,y1), and a second from (x2,y1) to (x2,y2).
PLOT_SCALETYPE_LINEAR	Use a linear scale for a specified axis.
PLOT_SCALETYPE_LOG	Use a logarithmic scale for a specified axis.
PLOT_TEXT_CENTER	Center text at a specified point.
PLOT_TEXT_LEFT	Left justify text at a specified point.
PLOT_TEXT_RIGHT	Right justify text at a specified point.
PLOT_TICS_IN	Draw axis tic-marks inward.
PLOT_TICS_OUT	Draw axis tic-marks outward.

Functions

The following functions are implemented using the **CPlot** class.

Function	Description
fplotxy()	Plot a 2D function of *x* in a specified range or initialize an instance of the **CPlot** class.
fplotxyz()	Plot a 3D function of *x* and *y* in a specified range or initialize an instance of the **CPlot** class.
plotxy()	Plot a 2D data set or initialize an instance of the **CPlot** class.
plotxyf()	Plot 2D data from a file or initialize an instance of the **CPlot** class.
plotxyz()	Plot a 3D data set or initialize an instance of the **CPlot** class.
plotxyzf()	Plot 3D data from a file or initialize an instance of the **CPlot** class.

References

T. Williams, C. Kelley, D. Denholm, D. Crawford, et al., *Gnuplot — An Interactive plotting Program,* Version 3.7, December 3, 1998, ftp://ftp.gnuplot.vt.edu/.

Copyright Notice of Gnuplot

```
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.
 *
 * Permission to modify the software is granted, but not the right to
 * distribute the complete modified source code.  Modifications are to
 * be distributed as patches to the released version.  Permission to
 * distribute binaries produced by compiling modified sources is granted,
 * provided you
 *   1. distribute the corresponding source modifications from the
 * released version in the form of a patch file along with the binaries,
 *   2. add special version identification to distinguish your version
 * in addition to the base release version number,
 *   3. provide your name and address as the primary contact for the
 * support of your modified version, and
 *   4. retain our contact information in regard to use of the base
 * software.
 * Permission to distribute the released version of the source code
 * along with corresponding source modifications in the form of a patch
 * file is granted with same provisions 2 through 4 for binary
 * distributions.
 *
 * This software is provided "as is" without express or implied warranty
 * to the extent permitted by applicable law.
]*/
```

CPlot::arrow

Synopsis
#include <**chplot.h**>
void arrow(**double** *x_head*, **double** *y_head*, **double** z_*head*, **double** *x_tail*, **double** *y_tail*, **double** z_*tail*, ...
/* [**int** *line_type*, **int** *line_width*] */);

Syntax
arrow(*x_head*, *y_head*, z_*head*, *x_tail*, *y_tail*, z_*tail*)
arrow(*x_head*, *y_head*, z_*head*, *x_tail*, *y_tail*, z_*tail*, *line_type*)
arrow(*x_head*, *y_head*, z_*head*, *x_tail*, *y_tail*, z_*tail*, *line_type*, *line_width*)

Purpose
Add an arrow to a plot.

Return Value
None.

Parameters
x_head The x coordinate of the head of the arrow.

y_head The y coordinate of the head of the arrow.

z_head For 2D plots this is ignored. For 3D plots, the z coordinate of the head of the arrow.

x_tail For x coordinate of the tail of the arrow.

y_tail The y coordinate of the tail of the arrow.

z_tail For 2D plots this is ignored. For 3D plots, the z coordinate of the tail of the arrow.

line_type An integer index representing the line type for drawing.

line_width A scaling factor for the arrow width. The arrow width is *line_width* multiplied by the default width.

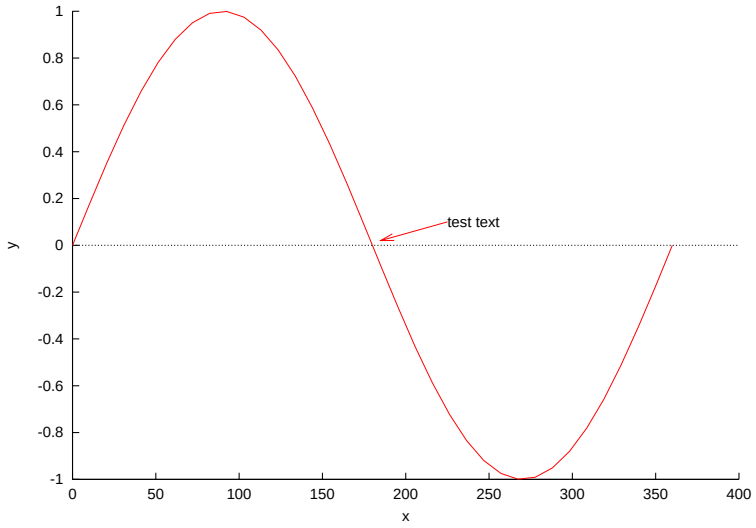
Description
This function adds an arrow to a plot. The arrow points from (*x_tail*, *y_tail*, *z_tail*) to (*x_head*, *y_head*, *z_head*). These coordinates are specified using the same coordinate system as the curves of the plot. The *line_type* is an optional argument specifying an index for the line type used for drawing the arrow. The line type varies depending on the terminal type used (see **CPlot**::**outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types. The *line_width* is an optional argument used to specify the arrow width. The line width is *line_width* multiplied by the default width. Typically the default width is one pixel.

Example 1
Compare with output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>
```

```
int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.arrow(185, 0.02, 0, 225, 0.1, 0);
    plot.text("test text", PLOT_TEXT_LEFT, 225, 0.1, 0);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

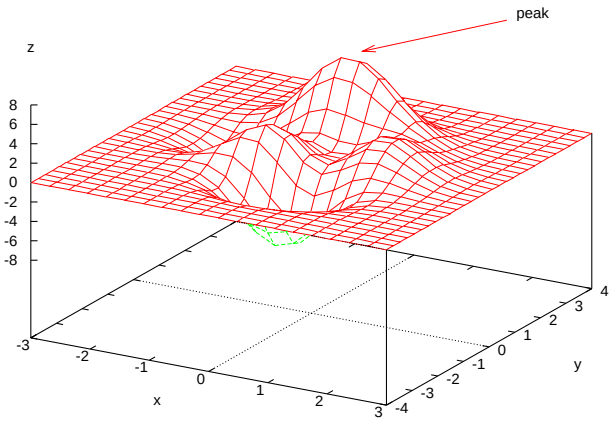


Example 2
Compare with the output for examples in **CPlot**::**data3D**() and **CPlot**::**data3DSurface**().

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.arrow(0, 2, 8, 2, 3, 12);
    plot.text("peak", PLOT_TEXT_LEFT, 2.1, 3.15, 12.6);
    plot.plotting();
```

```
}
```

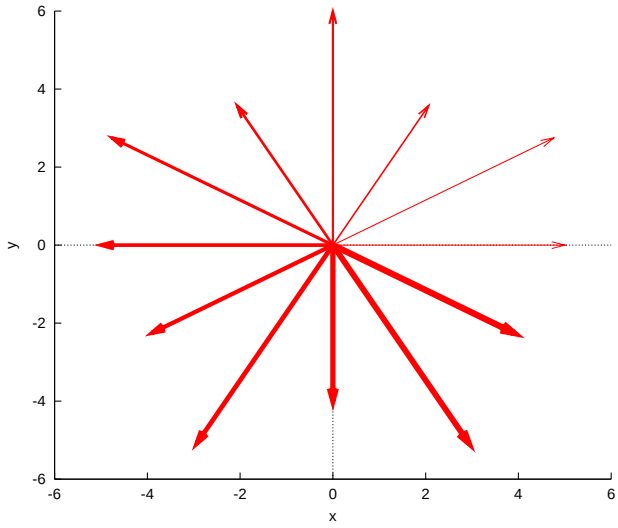
Output



Example 3

```
#include <math.h>
#include <chplot.h>

int main() {
    double complex z[12];
    int i;
    class CPlot plot;

    for (i=0; i<12; i++) {
        real(z[i]) = (5+sin(150*i*M_PI/180))*cos(30*i*M_PI/180);
        imag(z[i]) = (5+sin(150*i*M_PI/180))*sin(30*i*M_PI/180);
        plot.arrow(real(z[i]), imag(z[i]), 0, 0, 0, 0, 0, i+1);
    }
    plot.axisRange(PLOT_AXIS_XY, -6, 6);   /* one point cannot do autorange */
    plot.point(real(z[0]), imag(z[0]), 0); /* CPlot::arrow() itself is not a data set */
    plot.plotType(PLOT_PLOTTYPE_DOTS, 0);
    plot.sizeRatio(-1);
    plot.plotting();
}
```

Output



See Also
CPlot::**circle**(), **CPlot**::**data2D**(), **CPlot**::**outputType**(), **CPlot**::**plotType**(), **CPlot**::**point**(), **CPlot**::**polygon**(), **CPlot**::**rectangle**().

CPlot::autoScale

Synopsis
#include <chplot.h>
void autoScale(**int** *axis*, **int** *flag*);

Purpose
Set autoscaling of axes on or off.

Return Value
None.

Parameters
axis The axis to be set. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

flag Enable or disable auto scaling.

PLOT_ON The option is enabled.

PLOT_OFF The option is disabled.

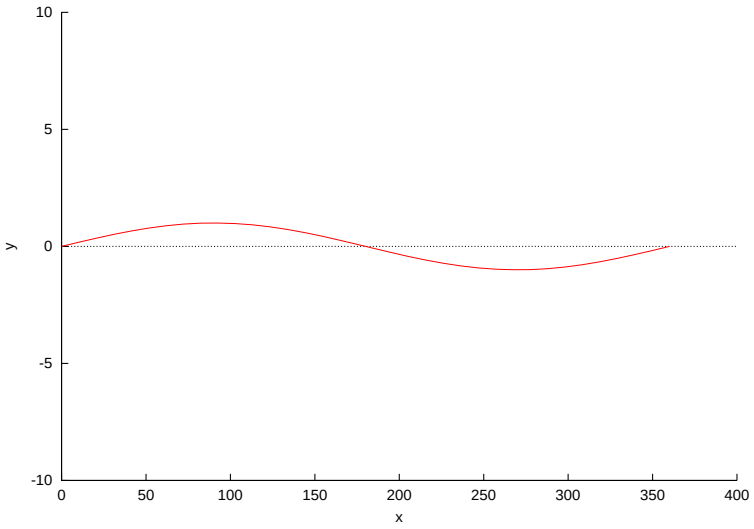
Description
Autoscaling of the axes can be **PLOT_ON** or **PLOT_OFF**. Default is **PLOT_ON**. If autoscaling for an axis

is disabled, an axis range of [-10:10] is used.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.autoScale(PLOT_AXIS_Y, PLOT_OFF);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



CPlot::axis

Synopsis
#include <**chplot.h**>
void axis(**int** *axis*, **int** *flag*);

Purpose
Enable or disable drawing of x-y axis on 2D plots.

Return Value
None.

Parameters

axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_XY Select the x and y axes.

flag This parameter can be set to:

PLOT_ON Enable drawing of the specified axis.

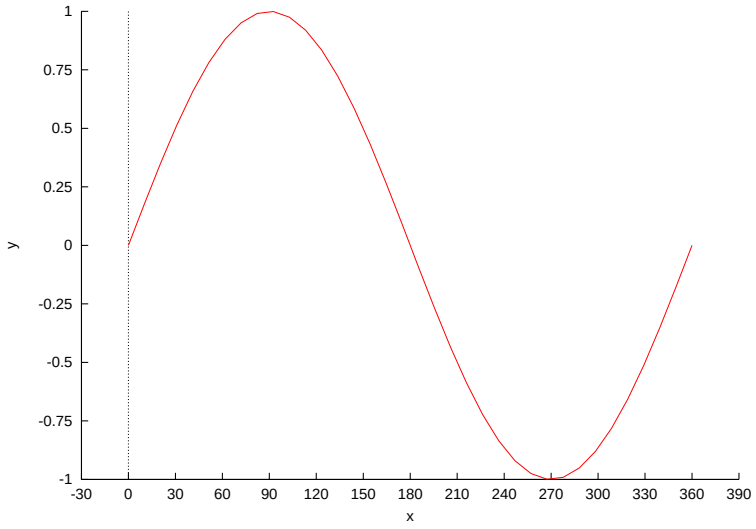
PLOT_OFF Disable drawing of the specified axis.

Description
Enable or disable the drawing of the x-y axes on 2D plots. By default, the x and y axes are drawn.

Example
Compare with the output for the example in **CPlot**::**axisRange**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    int i;
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);                        // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, -30, 390, 30);
    plot.axisRange(PLOT_AXIS_Y, -1, 1, .25);
    plot.axis(PLOT_AXIS_X, PLOT_OFF);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



CPlot::axisRange

Synopsis
#include <**chplot.h**>
void axisRange(**int** *axis*, **double** *minimum*, **double** *maximum*, ... /* [**double** *incr*] */);

Syntax
axisRange(*axis*, *minimum*, *maximum*)
axisRange(*axis*, *minimum*, *maximum*, *incr*)

Purpose
Specify the range for an axis.

Return Value
None.

Parameters
axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

minimum The axis minimum.

maximum The axis maximum.

incr The increment between tic marks. By default or when *incr* is 0, the increment between tic marks is calculated internally.

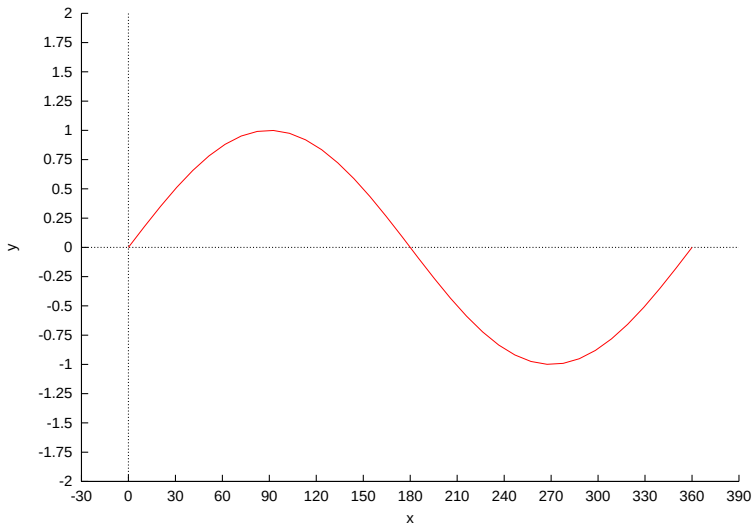
Description
The range for an axis can be explicitly specified with this function. Autoscaling for the specified axis is disabled and any previously specified labeled tic-marks are overridden.

Example 1
Compare with the output for the examples in **CPlot**::**axis**() and **CPlot**::**grid**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);                        // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, -30, 390, 30);
```

```
    plot.axisRange(PLOT_AXIS_Y, -2, 2, .25);
    plot.data2D(x, y);
    plot.plotting();
}
```

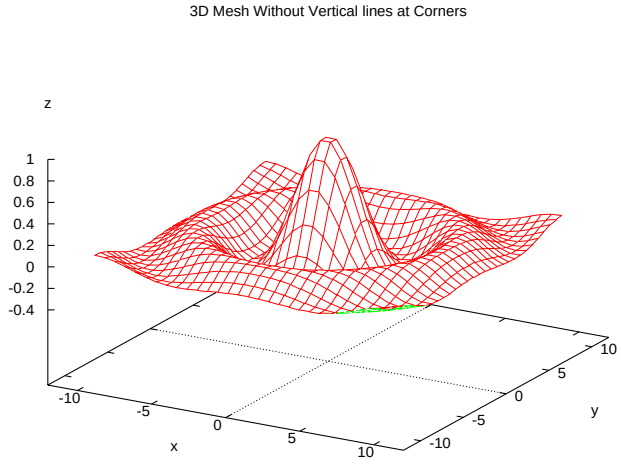
Output



Example 2
3D mesh plot without vertical lines at the corners. Compare with the output for examples in **CPlot**::**data3D**() and **CPlot**::**data3DSurface**().

```
#include <chplot.h>
#include <math.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.axisRange(PLOT_AXIS_XY, -12, 12);
    plot.title("3D Mesh Without Vertical lines at Corners");
    plot.plotting();
}
```

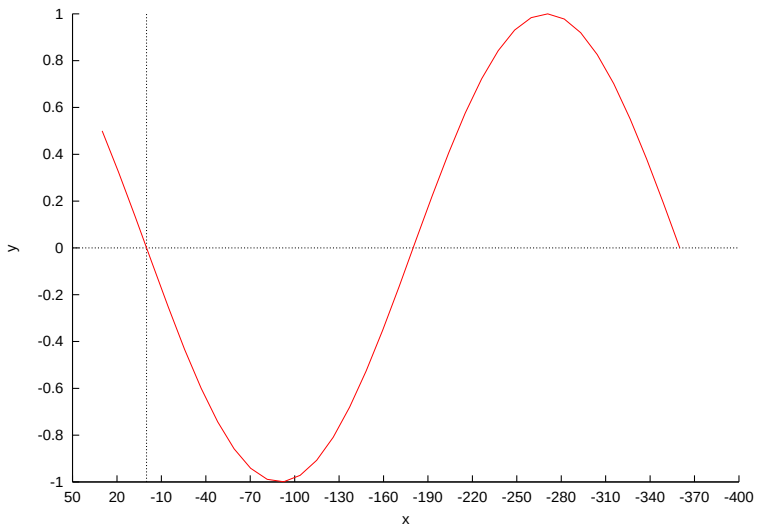
Output



Example 3
X axis range is reversed.

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, -360, 30);
    y = sin(x*M_PI/180);                        // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, 50, -400, -30);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also

CPlot::**autoScale**(), **CPlot**::**ticsLabel**().

CPlot::border()

Synopsis
#include <**chplot.h**>
void border(**int** *location*, **int** *flag*);

Purpose
Enable or disable display of a bounding box around the plot.

Return Value
None.

Parameter
location The location of the effected border segment.

PLOT_BORDER_BOTTOM The bottom of the plot.

PLOT_BORDER_LEFT The left side of the plot.

PLOT_BORDER_TOP The top of the plot.

PLOT_BORDER_RIGHT The right side of the plot.

PLOT_BORDER_ALL All sides of the plot.

flag Enable or disable display of a box around the boundary or the plot.

PLOT_ON Enable drawing of the box.

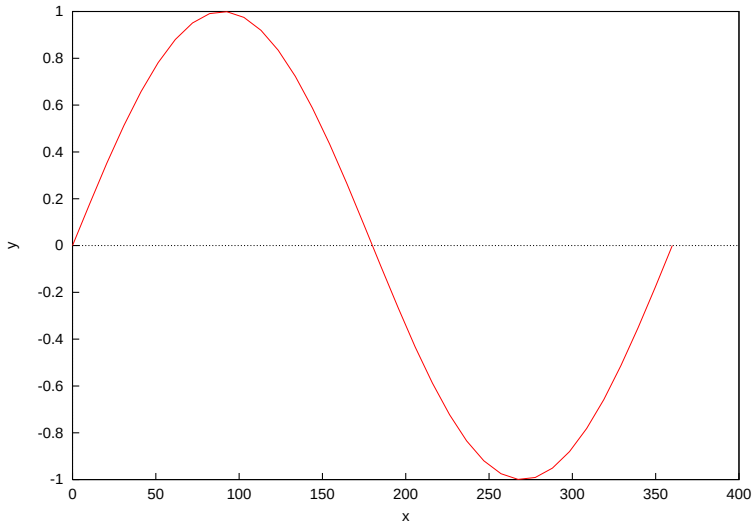
PLOT_OFF Disable drawing of the box.

Description
This function turns the display of a border around the plot on or off. By default, the border is drawn on the left and bottom sides for 2D plots, and on all sides for 3D plots.

Example 1
Compare with the example output in **CPlot**::**ticsMirror**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}
```

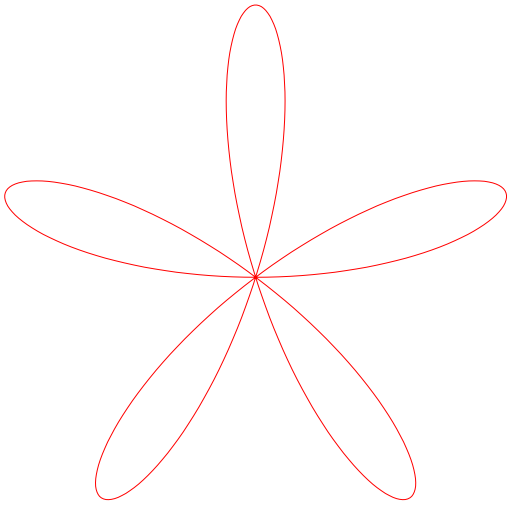
Output



Example 2
Compare with the example output in **CPlot**::**polarPlot**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    linspace(theta, 0, M_PI);
    r = sin(5*theta);                  // Y-axis data.
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.label(PLOT_AXIS_XY, NULL);
    plot.sizeRatio(1);
    plot.border(PLOT_BORDER_ALL, PLOT_OFF);
    plot.tics(PLOT_AXIS_XY, PLOT_OFF);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}
```

Output



CPlot::borderOffsets

Synopsis
#include <chplot.h>
void borderOffsets(**double** *left*, **double** *right*, **double** *top*, **double** *bottom*);

Purpose
Specify the size of offsets around the data points of a 2D plot in the same units as the plot axis.

Return Value
None.

Parameters
left The offset on the left side of the plot.

right The offset on the right side of the plot.

top The offset on the top of the plot.

bottom The offset on the bottom of the plot.

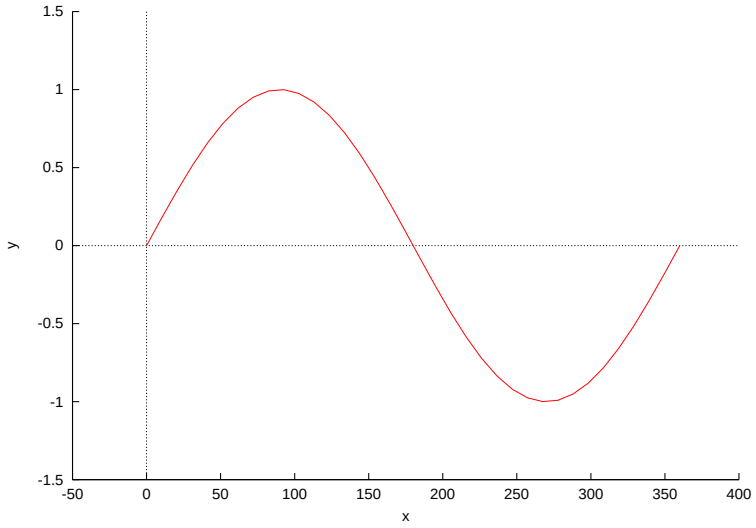
Description
For 2D plots, this function specifies the size of offsets around the data points of an autoscaled plot. This function can be used to create empty space around the data. The *left* and *right* arguments are specified in the units of the x-axis. The *top* and *bottom* arguments are specified in the units of the y-axis.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
```

```
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.borderOffsets(30, 30, .5, .25);
    plot.plotting();
}
```

Output



See Also
CPlot::**autoScale**(), **CPlot**::**axisRange**(), **CPlot**::**ticsLabel**(), **CPlot**::**margins**().

CPlot::boundingBoxOrigin

Synopsis
#include <**chplot.h**>
void boundingBoxOrigin(**double** *x_orig*, **double** *y_orig*);

Purpose
Set the location of the origin for the drawing of the plot.

Return Value
None.

Parameters
x_orig The x coordinate of the origin for the drawing of the plot.

y_orig The y coordinate of the origin for the drawing of the plot.

Description
This function specifies the location of the origin for the drawing of the plot. This is the location of the bottom

left corner of the bounding box of the plot, not the location of the origin of the plot coordinate system. The values *x_org* and *y_org* are specified as numbers between 0 and 1, where (0,0) is the bottom left of the plot area (the plot window) and (1,1) is the top right. This function is used internally by method **CPlot**::**subplot**() in conjunction with method **CPlot**::**size**().

See Also
CPlot::**getSubplot**(), **CPlot**::**size**(), **CPlot**::**subplot**().

CPlot::changeViewAngle

Synopsis
#include <chplot.h>
void changeViewAngle(**double** *x_rot*, **double** *z_rot*);

Purpose
Change the viewpoint for a 3D plot.

Return Value
None.

Parameters
x_rot The angle of rotation for the plot (in degrees) along an axis horizontal axis of the screen.

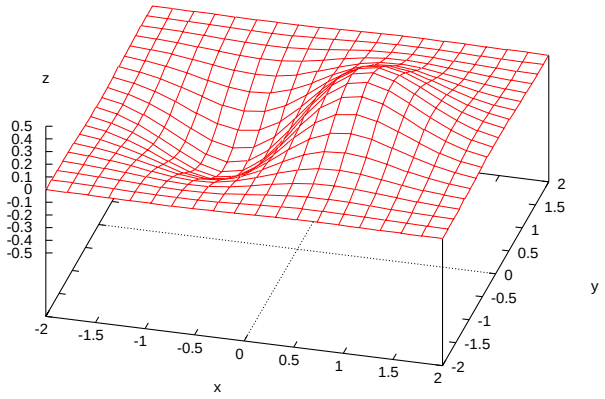
z_rot The angle of rotation for the plot (in degrees) along an axis perpendicular to the screen.

Description
This function provides rotation of a 3D plot. *x_rot* and z_*rot* are bounded by the range [0:180] and the range [0:360] respectively. By default, 3D plots are displayed with *x_rot* $= 60^{\circ}$ and *z_rot* $= 30^{\circ}$.

Example
Compare with the output for examples in **CPlot**::**data3D**() and **CPlot**::**data3DSurface**().

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[20], y[20], z[400];
    int i, j;
    class CPlot plot;

    linspace(x, -2, 2);
    linspace(y, -2, 2);
    for (i=0; i<20; i++) {
        for(j=0; j<20; j++) {
            z[i*20+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.changeViewAngle(45, 15);
    plot.plotting();
}
```

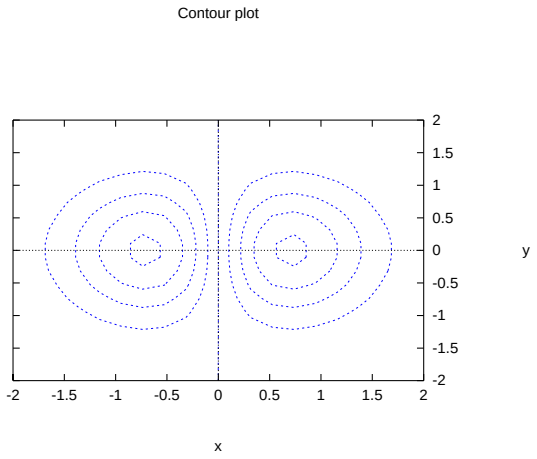
Output



Example

```
#include <chplot.h>
#include <math.h>

int main() {
    array double x[20], y[20], z[400];
    int i, j;
    class CPlot plot;

    linspace(x, -2, 2);
    linspace(y, -2, 2);
    for (i=0; i<20; i++) {
        for(j=0; j<20; j++) {
            z[i*20+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.changeViewAngle(0,0);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.showMesh(PLOT_OFF);
    plot.title("Contour plot");
    plot.plotting();
}
```

Output



See Also
CPlot::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**().

CPlot::circle

Synopsis
#include <**chplot.h**>
int circle(**double** *x_center*, **double** *y_center*, **double** *r*);

Purpose
Add a circle to a 2D plot.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x_center The x coordinate of the center of the circle.

y_center The y coordinate of the center of the circle.

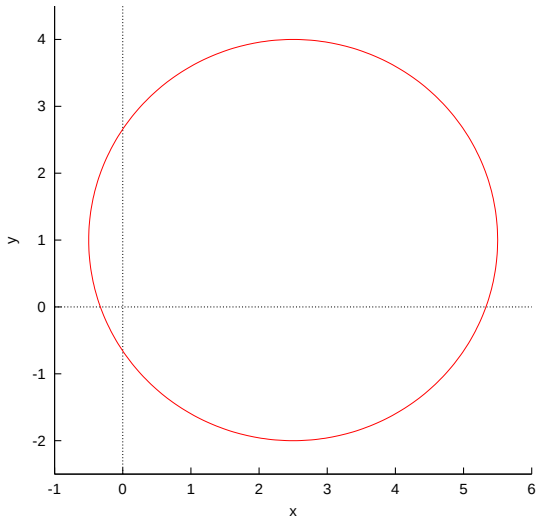
r The radius of the circle.

Description
This function adds a circle to a 2D plot. It is a convenience function for creation of geometric primitives. A circle added with this function is counted as a data set for later calls to **CPlot**::**legend**() and **CPlot**::**plotType**(). For rectangular plots, *x* and *y* are the coordinates of the center of the circle and *r* is the radius of the circle, all specified in units of the x and y axes. For polar plots, the location of the center of the circle is specified in polar coordinates where *x* is θ and *y* is r.
Example 1

```
#include <chplot.h>

int main() {
    double x_center = 2.5, y_center = 1.0, r = 3;
```

```
    class CPlot plot;

    plot.circle(x_center, y_center, r);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_Y, -2.5, 4.5);
    plot.plotting();
}
```

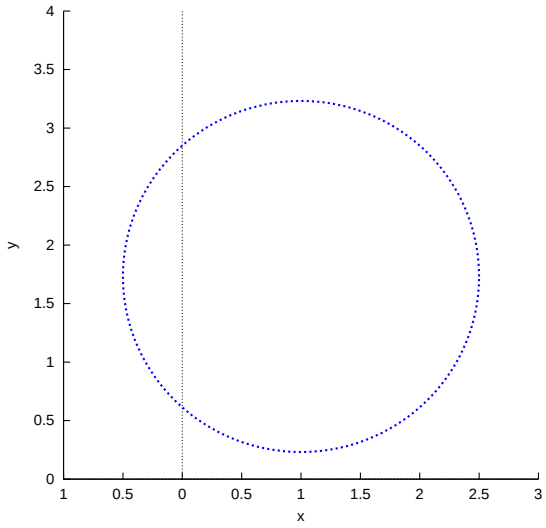
Output



Example 2

```
#include <chplot.h>
#include <math.h>

int main() {
    double x_center = M_PI/3, y_center = 2.0, r = 1.5;
    class CPlot plot;

    plot.polarPlot(PLOT_ANGLE_RAD); /* Polar coordinate system */
    plot.circle(x_center, y_center, r);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0, 3, 4);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_X, -1, 3);
    plot.axisRange(PLOT_AXIS_Y, 0, 4);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**line**(), **CPlot**::**outputType**(), **CPlot**::**plotType**(), **CPlot**::**point**(), **CPlot**::**polygon**(), **CPlot**::**rectangle**().

CPlot::contourLabel

Synopsis
#include <**chplot.h**>
void contourLabel(**int** *flag*);

Purpose
Set display of contour line elevation labels for 3D plots on or off.

Return Value
None.

Parameter
flag Enable or disable drawing of contour line labels.

PLOT_ON Enable contour labels.

PLOT_OFF Disable contour labels.

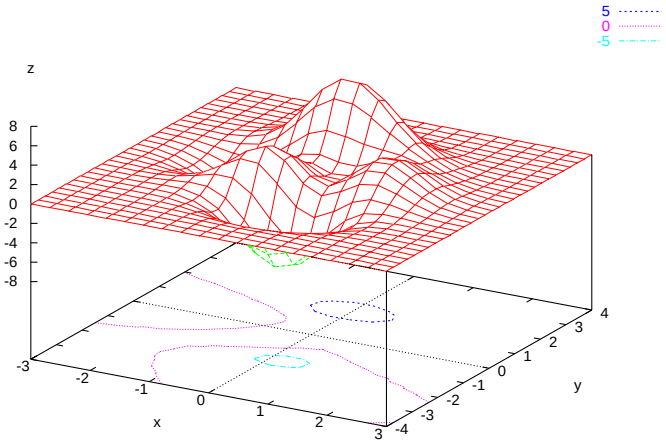
Description
Enable or disable contour line labels for 3D surface plots. labels appear with the plot legend. By default, labels for contours are not displayed.

Example 1
Compare with the output for examples in **CPlot**::**data3D**(), **CPlot**::**data3DSurface**(). **CPlot**::**contourLevels**(), and **CPlot**::**showMesh**().

```
#include <math.h>
#include <chplot.h>

int main() {
```

```
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.plotting();
}
```

Output

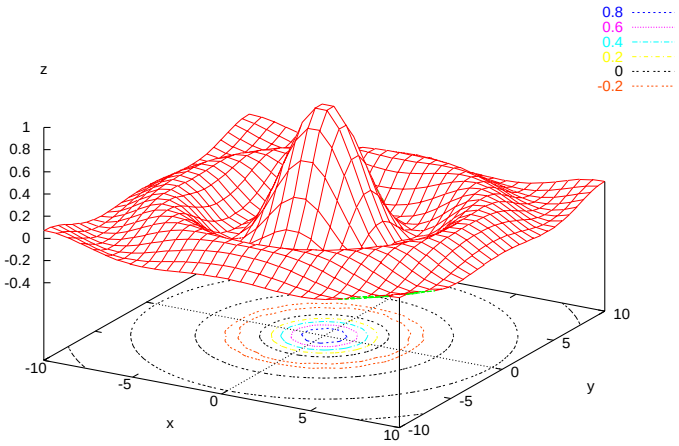


Example 2

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
```

```
    plot.data3D(x, y, z);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.plotting();
}
```

Output



See Also
CPlot::**data3D**(), **CPlot**::**contourLevels**(), **CPlot**::**contourMode**(), **CPlot**::**legend**(), **CPlot**::**showMesh**().

CPlot::contourLevels

Synopsis
#include <**chplot.h**>
void contourLevels(**double** *levels*[:], ... /* [**int** *num*] */);

Syntax
contourLevels(*level*)
contourLevels(*level*, *num*)

Purpose
Set contour levels for 3D plot to be displayed at specific locations.

Return Value
None.

Parameter
levels An array of z-axis levels for contours to be drawn.

num The number of elements of array *levels*.

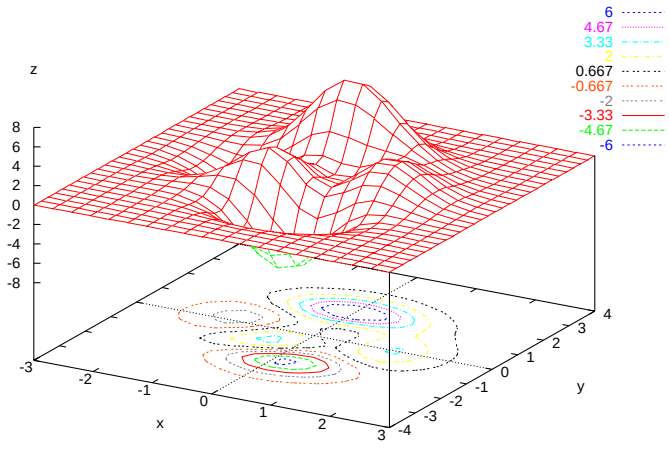
Description
This function allows contour levels for 3D grid data to be displayed at any desired z-axis position. The

contour levels are stored in an array where the lowest contour is in the first array element.

Example 1
Compare with the output from the example in **CPlot**::**contourLabel**().

```
#include <math.h>
#include <chplot.h>

int main() {
    float x[20], y[30], z[600];
    double levels[10];
    int i,j;
    class CPlot plot;

    linspace(levels, -6, 6);
    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.contourLevels(levels);
    plot.plotting();
}
```

Output

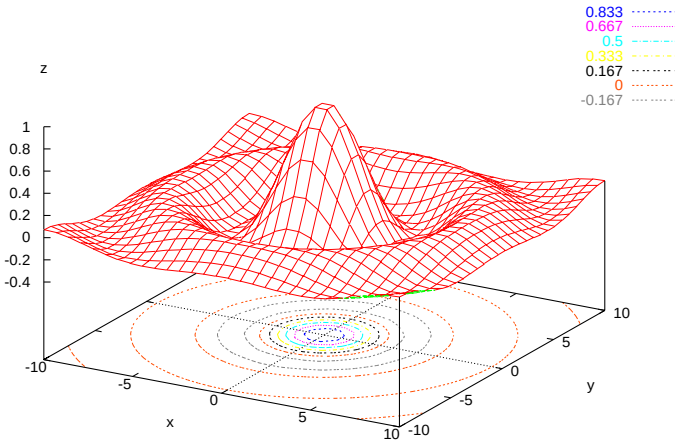


Example 2

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
```

```
    double levels[6];
    double r;
    int i, j;
    class CPlot plot;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    linspace(levels, -0.2, 0.8);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.contourLabel(PLOT_ON);
    plot.contourMode(PLOT_CONTOUR_BASE);
    plot.contourLevels(levels);
    plot.plotting();
}
```

Output



See Also
CPlot::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**contourLabel**(), **CPlot**::**contourMode**(), **CPlot**::**showMesh**().

CPlot::contourMode

Synopsis
#include <**chplot.h**>
void contourMode(**int** *mode*);

Purpose
Selects options for the drawing of contour lines in 3D plots.

Return Value
None.

Parameter
mode The following contour modes are available and can be combined using the logical or (|) operator:

PLOT_CONTOUR_BASE Contour lines drawn on the xy plane.

PLOT_CONTOUR_SURFACE Contour lines drawn on the surface of the plot.

Description
This option is available for display of 3D grid data. The positions of the contour levels are determined internally unless explicitly set using **CPlot**::**contourLevels**(). The **PLOT_CONTOUR_SURFACE** option does not work with hidden line removal. The hidden lines are removed by default. It can be disabled by member function **CPlot**::**removeHiddenLine**().

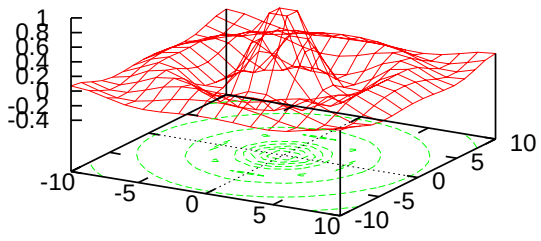
Example

```
#include <chplot.h>
#include <math.h>

int main() {
    double x[16], y[16], z[256];
    double r;
    int i, j;
    class CPlot subplot, *spl;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<16; i++) {
        for(j=0; j<16; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[16*i+j] = sin(r)/r;
        }
    }
    subplot.subplot(2,2);
    spl = subplot.getSubplot(0,0);
    spl->data3D(x, y, z);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->contourMode(PLOT_CONTOUR_BASE);
    spl->removeHiddenLine(PLOT_OFF);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_CONTOUR_BASE");
    spl = subplot.getSubplot(0,1);
    spl->data3D(x, y, z);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->contourMode(PLOT_CONTOUR_SURFACE);
    spl->removeHiddenLine(PLOT_OFF);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_CONTOUR_SURFACE");
    spl = subplot.getSubplot(1,0);
    spl->data3D(x, y, z);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->contourMode(PLOT_CONTOUR_BASE|PLOT_CONTOUR_SURFACE);
    spl->removeHiddenLine(PLOT_OFF);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_CONTOUR_BASE|PLOT_CONTOUR_SURFACE");
```

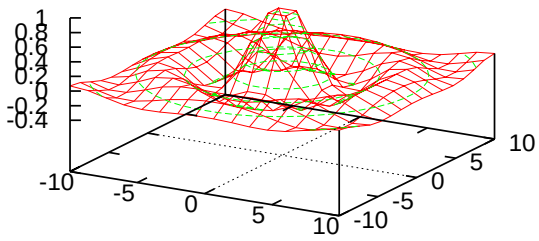
```
    subplot.plotting();
}
```

Output

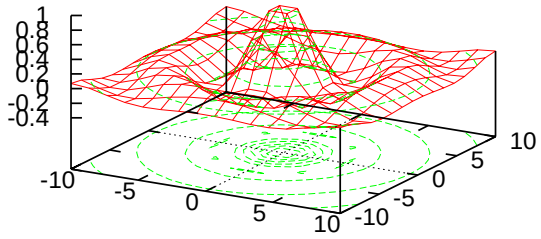
PLOT_CONTOUR_BASE



PLOT_CONTOUR_SURFACE



PLOT_CONTOUR_BASE|PLOT_CONTOUR_SURFACE



See Also
CPlot::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**contourLevels**(), **CPlot**::**removeHiddenLine**(), **CPlot**::**showMesh**().

CPlot::coordSystem

Synopsis
#include <**chplot.h**>
void coordSystem(**int** *coord_system*, ... /* [**int** *angle_unit*] */);

Syntax
coordSystem(*coord_system*)
coordSystem(*coord_system*, *angle_unit*)

Purpose
Select coordinate system for 3D plots.

Return Value

None.

Parameters

coord_system The coordinate system can be set to any of the following:

PLOT_COORD_CARTESIAN Cartesian coordinate system. Each data point consists of three values (x,y,z).

PLOT_COORD_SPHERICAL Spherical coordinate system. Each data point consists of three values (θ,ϕ,r).

PLOT_COORD_CYLINDRICAL Cylindrical coordinates. Each data point consists of three values (θ,z,r).

angle_unit an optional parameter to specify the unit for mesurement of an angular position in **PLOT_COORD_SPHERICAL** and **PLOT_COORD_CYLINDRICAL** coordinate systems. The following options are available:

PLOT_ANGLE_DEG Angles measured in degree.

PLOT_ANGLE_RAD Angles measured in radian.

Description

This function selects the coordinate system for 3D plots. For a spherical coordinate system, points are mapped to Cartesian space by:

$$\begin{aligned} x &= r\cos(\theta)\cos(\phi) \\ y &= r\sin(\theta)\cos(\phi) \\ z &= r\sin(\phi) \end{aligned}$$

For a cylindrical coordinate system, points are mapped to Cartesian space by:

$$\begin{aligned} x &= r\cos(\theta) \\ y &= r\sin(\theta) \\ z &= z \end{aligned}$$

The default coordinate system is **PLOT_COORD_CARTESIAN**. For **PLOT_COORD_SPHERICAL** and **PLOT_COORD_CYLINDRICAL**, the default unit for *angle_unit* is **PLOT_ANGLE_RAD**.

Example 1

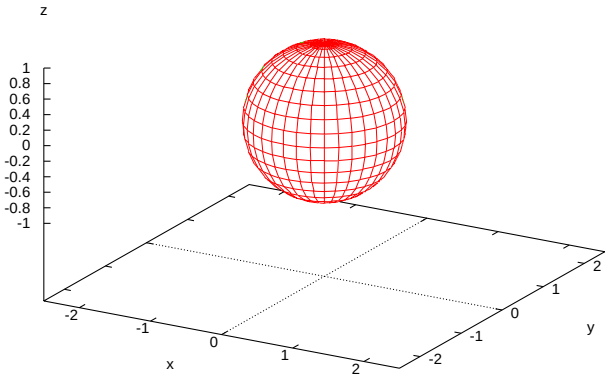
See **CPlot**::**data3D**().

Example 2

```
#include <chplot.h>
#include <math.h>

int main() {
    array double theta[37], phi[19], r[703];
    class CPlot plot;

    linspace(theta, 0, 2*M_PI);
    linspace(phi, -M_PI/2, M_PI/2);
```

```
    r = (array double [703])1;
    plot.data3D(theta, phi, r);
    plot.coordSystem(PLOT_COORD_SPHERICAL);
    plot.axisRange(PLOT_AXIS_XY, -2.5, 2.5);
    plot.plotting();
}
```

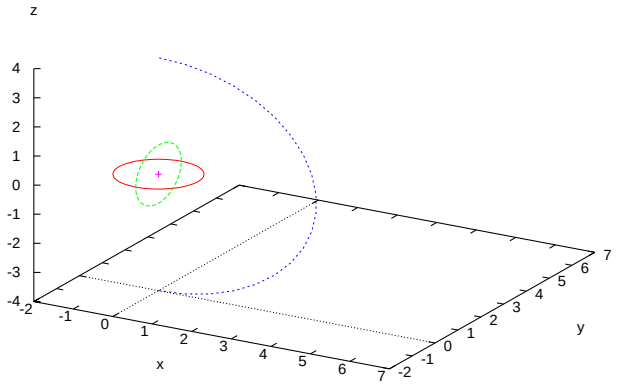
Output



Example 3

```
#include <chplot.h>
#include <math.h>

int main() {
    array double theta1[37], phi1[37], r1[37];
    array double theta2[37], phi2[37], r2[37];
    array double theta3[20], phi3[20], r3[20];
    class CPlot plot;

    linspace(theta1, 0, 2*M_PI);
    phi1 = (array double [37])0;
    r1 = (array double [37])1;
    theta2 = (array double [37])M_PI/2;
    linspace(phi2, 0, 2*M_PI);
    r2 = (array double [37])1;
    theta3 =  (array double [20])0;
    linspace(phi3, -M_PI/2, M_PI/2);
    r3 = (array double [20])4;
    plot.data3D(theta1, phi1, r1);
    plot.data3D(theta2, phi2, r2);
    plot.data3D(theta3, phi3, r3);
    plot.point(0, 0, 0);
    plot.coordSystem(PLOT_COORD_SPHERICAL);
    plot.axisRange(PLOT_AXIS_XY, -2, 7);
    plot.ticsLevel(0);
    plot.removeHiddenLine(PLOT_OFF);
    plot.plotting();
}
```

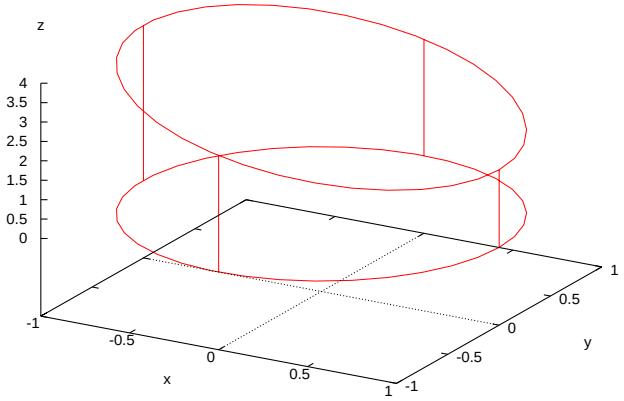
Output



Example 4

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double r1[numpoints], theta1[numpoints], z1[numpoints];
    array double r2[numpoints], theta2[numpoints], z2[numpoints];
    class CPlot plot;

    linspace(theta1, 0, 360);
    z1=(array double [numpoints])3+
       sin((theta1-(array double [numpoints])90)*M_PI/180);
    r1=(array double [numpoints])1;
    linspace(theta2, 0, 360);
    z2=(array double [numpoints])0;
    r2=(array double [numpoints])1;
    plot.data3D(theta1, z1, r1);
    plot.data3D(theta2, z2, r2);
    plot.coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
    plot.line(0, 0, 1, 0, 2, 1);
    plot.line(90, 0, 1, 90, 3, 1);
    plot.line(180, 0, 1, 180, 4, 1);
    plot.line(270, 0, 1, 270, 3, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 2, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 3, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 4, 1, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 5, 1, 1);
    plot.plotting();
}
```

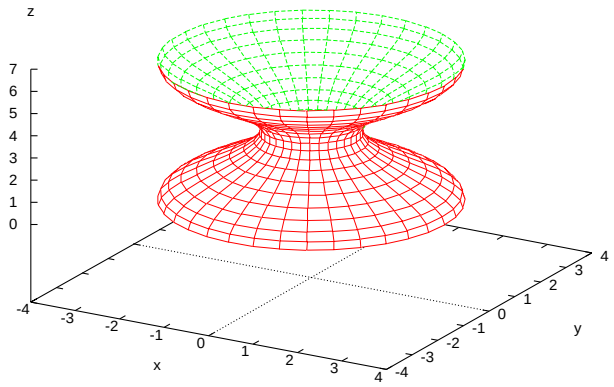
Output



Example 5

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double theta[36], z[20], r[720];
    int i, j;
    class CPlot plot;

    linspace(theta, 0, 360);
    linspace(z, 0, 2*M_PI);
    for(i=0; i<36; i++) {
        for(j=0; j<20; j++) {
            r[i*20+j] = 2+cos(z[j]);
        }
    }
    plot.data3D(theta, z, r);
    plot.coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
    plot.axisRange(PLOT_AXIS_XY, -4, 4);
    plot.plotting();
}
```

Output



See Also
CPlot::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**().

CPlot::data2D

Synopsis
#include <**chplot.h**>
int data2D(**array double** *x*[&], **array double** &*y*);

Purpose
Add one or more 2D data sets to an instance of **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x a one-dimensional array of reference type used for the x axis of the plot.

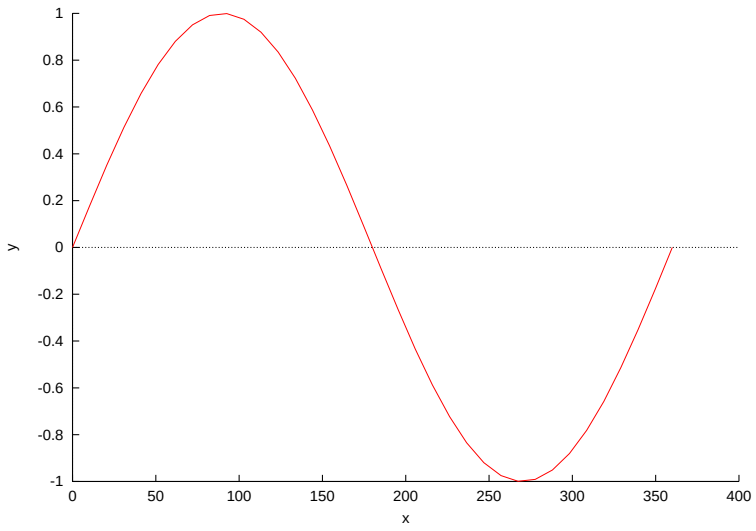
y an array of reference type. If the size of array x is n, array y is a one-dimensional array of size n or a two-dimensional array of size m x n containing m curves, each of which is plotted against *x*.

Description
This function adds the specified x-y data to a previously declared instance of the **CPlot** class. The parameter *x* is a one-dimensional array of size n. The parameter *y* is a one-dimensional array of size n or a two-dimensional array of size m x n. In the case that *y* is m x n, each of the m rows of *y* is plotted against *x*. Each of the rows of y is a separate data set. The *x* and *y* arrays can be of any supported data type. Conversion of the data to **double** type is performed internally. Data points with a y value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *y* to this value. The plot of the data is generated using the **CPlot**::**plotting** member function.

Example 1
Compare with the output for the examples in **CPlot**::**arrow**(), **CPlot**::**autoScale**(), **CPlot**::**borderOffsets**(), **CPlot**::**displayTime**(), **CPlot**::**label**(), **CPlot**::**ticsLabel**(), **CPlot**::**margins**(), **CPlot**::**ticsDirection**(), **CPlot**::**ticsFormat**(), **CPlot**::**ticsLocation**(), and **CPlot**::**title**().

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

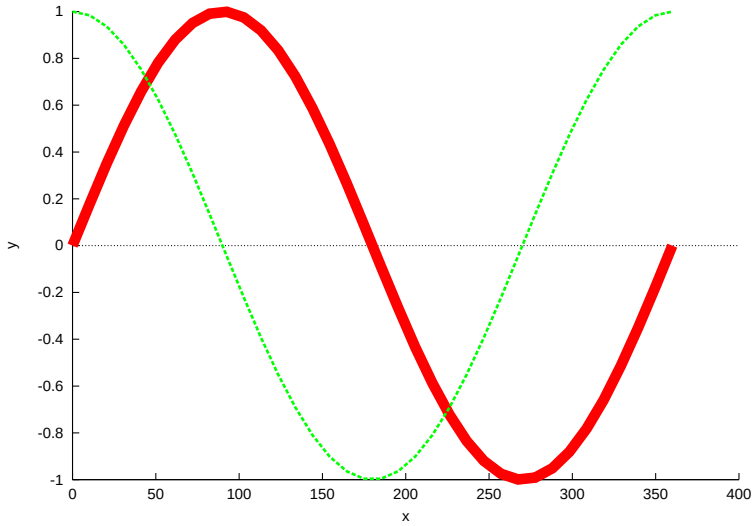


Example 2

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>
#define CURVE1 0
#define CURVE2 1

int main() {
    int numpoints = 36;
    array double x[numpoints], y[2][numpoints];
    int i;
    class CPlot plot;

    linspace(x, 0, 360);
    for(i=0; i<numpoints; i++) {
        y[0][i] = sin(x[i]*M_PI/180);
        y[1][i] = cos(x[i]*M_PI/180);
    }
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, CURVE1, 0, 20); /* set first data set to
                                                          use the default line
                                                          type and a width of
```

```
                                                        twenty  times the
                                                        default width */
    plot.plotType(PLOT_PLOTTYPE_LINES, CURVE2, 0, 5);  /* set second data set to
                                                        use the default line
                                                        type and a width of
                                                        five times the
                                                        default */
    plot.plotting();
}
```

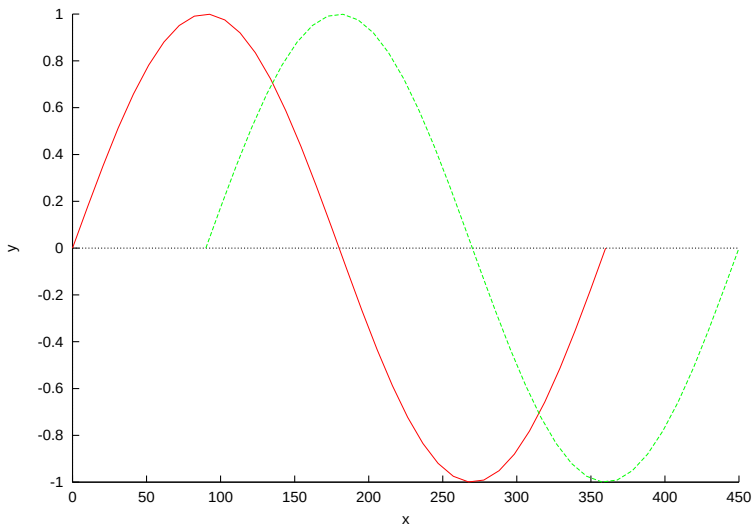
Output



Example 3

```
/* Two curves have a phase shift */
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    int numpoints = 36;
    array double x1[numpoints], y1[numpoints];
    array double x2[numpoints];
    class CPlot plot;

    linspace(x1, 0, 360);
    y1 = sin(x1*M_PI/180);
    linspace(x2, 90, 450);
    plot.data2D(x1, y1);
    plot.data2D(x2, y1);
    plot.plotting();
}
```

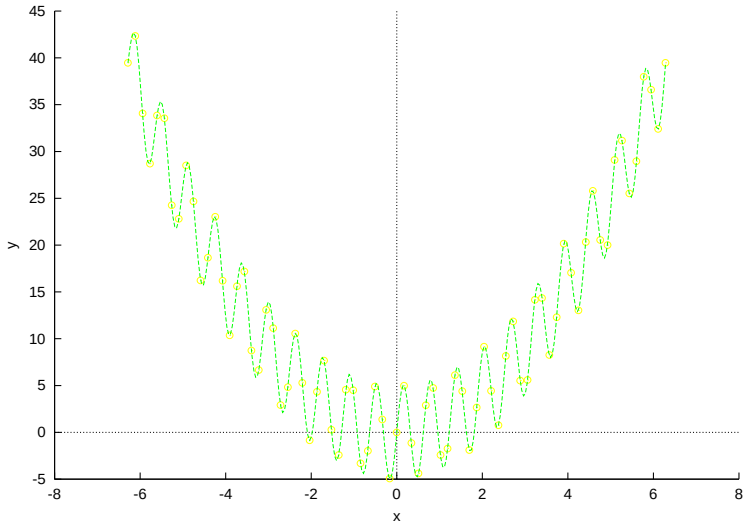
Output



Example 4

```
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    array float x1[75], y1[75];
    array float x2[300], y2[300];
    class CPlot plot;

    linspace(x1, -2*M_PI, 2*M_PI);
    linspace(x2, -2*M_PI, 2*M_PI);
    y1 = x1.*x1+5*sin(10*x1);
    y2 = x2.*x2+5*sin(10*x2);
    plot.data2D(x1, y1);
    plot.data2D(x2, y2);
    plot.plotType(PLOT_PLOTTYPE_POINTS, 0, 6, 1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1, 2, 1);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**plotting**(), **plotxy**().

CPlot::data2DCurve

Synopsis
#include <chplot.h>
int data2DCurve(double *x*[], **double** *y*[], int n);

Purpose
Add a set of data for 2D curve to an instance of **CPlot** class.
Return Value
This function returns 0 on success and -1 on failure.

Parameters
x a one-dimensional array of double type used for the x axis of the plot.

y a one-dimentaional array of double type for the y axis.

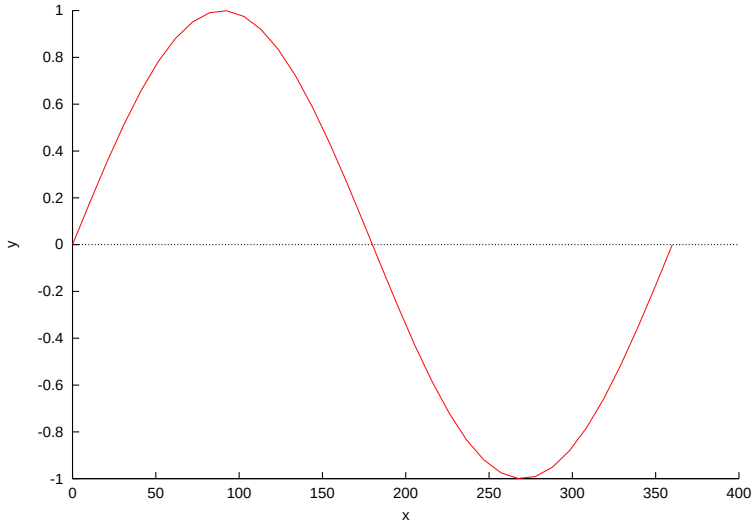
n number of elements of arrays *x* and *y*.

Description
This function adds the specified x-y data to a previously declared instance of the **CPlot** class. The parameter *x* is a one-dimensional array of size n. The parameter *y* is a one-dimensional array of size n. Data points with a y value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *y* to this value. The plot of the data is generated using the **CPlot**::**plotting** member function.

Example 1
Compare with the output for examples in **CPlot**::**arrow**(), **CPlot**::**autoScale**(), **CPlot**::**borderOffsets**(), **CPlot**::**data2D**(), **CPlot**::**displayTime**(), **CPlot**::**label**(), **CPlot**::**ticsLabel**(), **CPlot**::**margins**(), **CPlot**::**ticsDirection**(), **CPlot**::**ticsFormat**(), **CPlot**::**ticsLocation**(), and **CPlot**::**title**().

```
#include <math.h>
#include <chplot.h>
#ifndef M_PI
#define M_PI           3.14159265358979323846
#endif

#define NUM 36
int main() {
    int i;
    double x[NUM], y[NUM];
    class CPlot plot;

    for(i=0; i< NUM; i++) {
      x[i] = i*10;
      y[i] = sin(x[i]*M_PI/180);
    }
    plot.data2DCurve(x, y, NUM);
    plot.plotting();
    return 0;
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**plotting**(), **plotxy**().

CPlot::data3D

Synopsis
#include <**chplot.h**>
int data3D(**array double** *x*[&], **array double** *y*[&], array **double** &*z*);

Purpose
Add one or more 3D data sets to an instance of **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x a one-dimensional array of size n_x used for the x axis of the plot.

y a one-dimensional array of size n_y used for the y axis of the plot.

z a one-dimensional array of size n_z, or a two dimensional array of size m x n_z, containing m 3D data sets.

Description
This function adds one or more 3D data sets to an existing plot variable. For Cartesian data, *x* is a one-dimensional array of size n_x and *y* is a one-dimensional array of size n_y. *z* can be of two different sizes depending on what type of data is to be plotted. If the data are for a 3D curve, *z* is a one-dimensional array of size n_z or a two-dimensional array of size m x n_z, with $n_x = n_y = n_z$. If the data are for a 3D surface or grid, *z* is m x n_z, with $n_z = n_x \cdot n_y$. For cylindrical or spherical data *x* is a one dimensional array of size n_x (representing θ), *y* is a one dimensional array of size n_y (representing z or ϕ), and *z* is of the size m x n_z(representing r) where $n_x = n_y = n_z$. Each of the m rows of *z* are plotted against x and y, and correspond to a separate data set. In all cases these data arrays can be of any supported data type. Conversion of the data to **double** type is performed internally. For grid data, hidden line removal is enabled automatically (see **CPlot**::**removeHiddenLine**()). If it is desired to plot both grid data and non-grid data on the same plot, hidden line removal should be disabled manually after all data are added. Data points with a z value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *z* to this value.

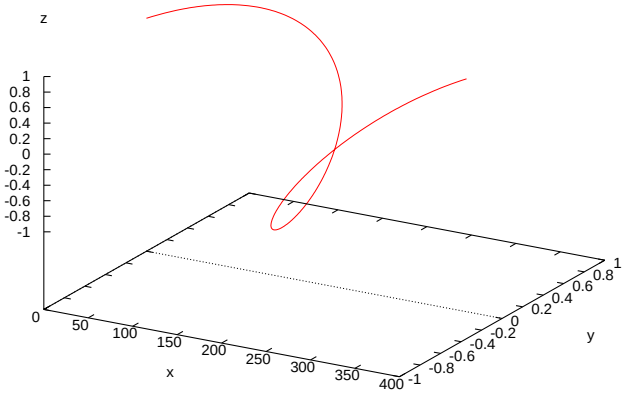
It is important to note that for a 3D grid, the ordering of the z data is important. For calculation of the z values, the *x* value is held constant while *y* is cycled through its range of values. The *x* value is then incremented and *y* is cycled again. This is repeated until all the data are calculated. So, for a 10x20 grid the data shall be ordered as follows:

x1 y1 z1
x1 y2 z2
x1 y3 z3
.
.
.
x1 y18 z18
x1 y19 z19
x1 y20 z20
x2 y1 z21
x2 y2 z22
x2 y3 z23
.
.
.
x10 y18 z198
x10 y19 z199
x10 y20 z200

Example 1

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    array double x[360], y[360], z[360];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    z = cos(x*M_PI/180);
    plot.data3D(x, y, z);
    plot.plotting();
}
```

Output

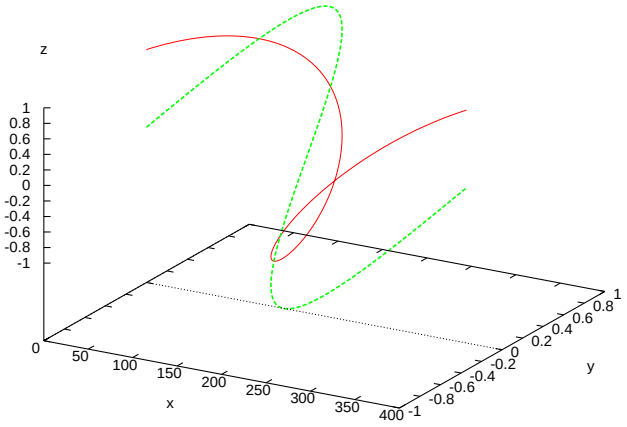


Example 2

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    array double x[360], y[360], z[2][360];
    int i;
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    for(i=0; i<360; i++) {
        z[0][i] = cos(x[i]*M_PI/180);
        z[1][i] = y[i];
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1, 0, 3); /* set the second data set to
                                                    use the default line type
                                                    and a line width three
                                                    times the default */
    plot.plotting();
```

```
}
```

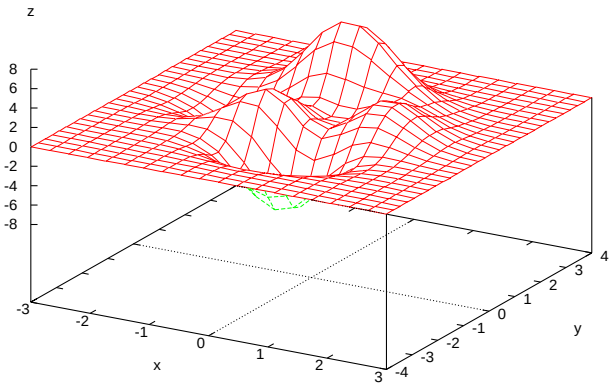
Output



Example 3
Compare with output for examples in **CPlot**::**arrow**(), **CPlot**::**contourLabel**(), **CPlot**::**grid**(), **CPlot**::**removeHiddenLine**(), **CPlot**:**size3D**(), **CPlot**::**changeViewAngle**(), and **CPlot**::**ticsLevel**().

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();

}
```

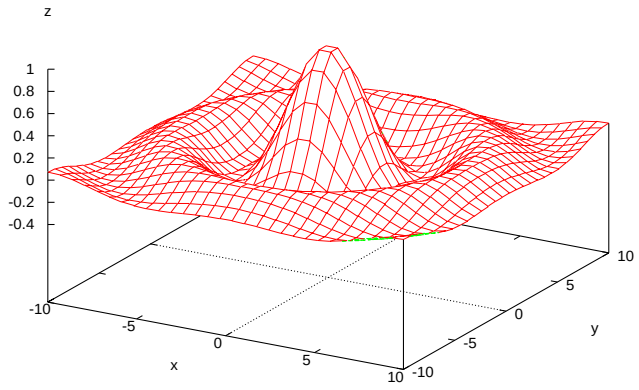
Output



Example 4
Compare with output for example **CPlot**::**axisRange**().

```
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}
```

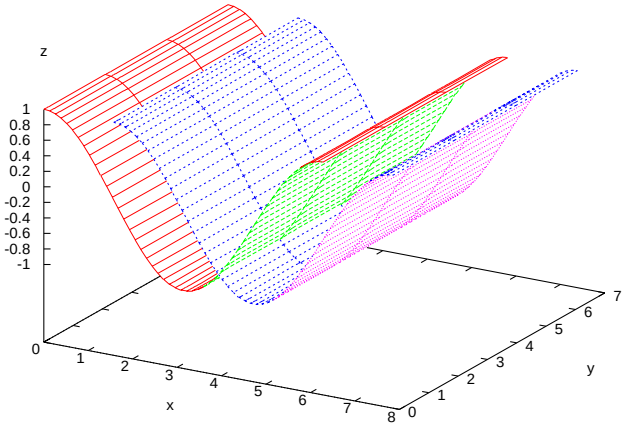
Output



Example 5

```
#include <chplot.h>
#include <math.h>
#include <numeric.h>

int main() {
    array double x1[50], x2[50], y[4], z[200];
    int i,j,angle;
    class CPlot plot;

    linspace(x1, 0, 2*M_PI);
    linspace(x2, M_PI/2, 2*M_PI+M_PI/2);
    linspace(y, 0, 2*M_PI);
    for (i=0;i<50;i++) {
        for (j=0;j<4;j++) {
            z[j+4*i]=cos(x1[i]);         // Z-axis data.
        }
    }
    plot.data3D(x1, y, z);
    plot.data3D(x2, y, z);
    plot.plotting();
}
```

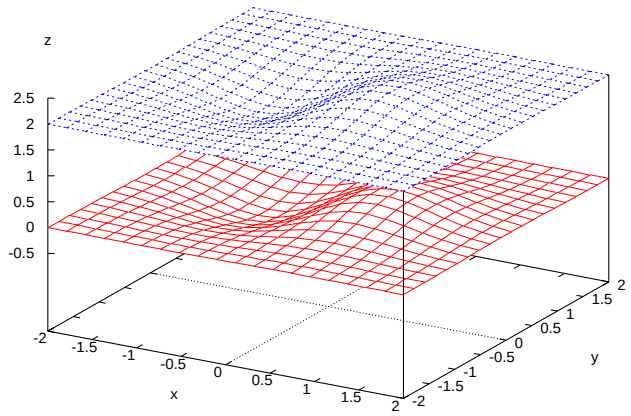
Output



Example 6
Compare with the output for the example **CPlot**::**changeViewAngle**().

```
#include <chplot.h>
#include <math.h>
#include <numeric.h>

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
int main() {
    array double x[NUMX], y[NUMY], z[NUMCURVE][NUMX*NUMY];
    int i, j;
    class CPlot plot;

    linspace(x, -2, 2);
    linspace(y, -2, 2);
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z[0][i*NUMX+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z[1][i*NUMX+j] = z[0][i*NUMX+j] +2;
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**plotting**(), **plotxyz**().

CPlot::data3DCurve

Synopsis
#include <chplot.h>
int data3DCurve(double *x*[], **double** *y*[], **double** *z*[], **int** n);

Purpose
Add a set of data for 3D curve to an instance of **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x a one-dimensional array of size n used for the x axis of the plot.

y a one-dimensional array of size n used for the y axis of the plot.

z a one-dimensional array of size n used for the z axis of the plot.

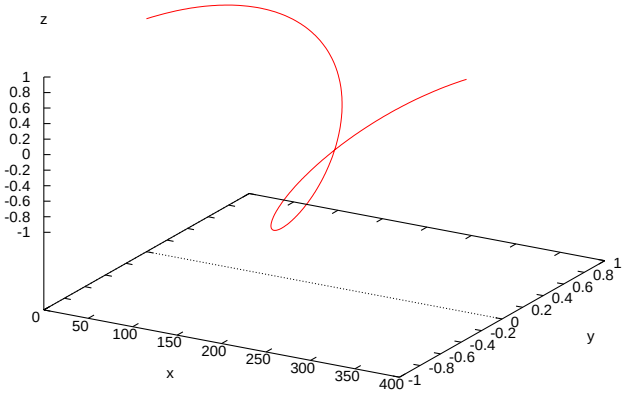
n the number of elements for arrays *x, y*, and *z*.

Description
Add a set of data for 3D curve to an instance of **CPlot** class. Arrays x, *y*, and *z* have the same number of elements of size *n*. In a Cartesian coordinate system, these arrays represent data in X-Y-Z coordinates. In a cylindrical coordinate system, *x* represents θ, *y* for z, and *z* for r. In a spherical coordinate system, *x* represents θ, *y* for ϕ, and *z* for r. Data points with a z value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *z* to this value.

Example 1
Compare with output for examples in **CPlot**::**data3D**().

```
#include <math.h>
#include <chplot.h>
#ifndef M_PI
#define M_PI            3.14159265358979323846
#endif

#define NUM 36
int main() {
    int i;
    double x[NUM], y[NUM], z[NUM];
    class CPlot plot;

    for(i=0; i< NUM; i++) {
      x[i] = i*10;
      y[i] = sin(x[i]*M_PI/180);
      z[i] = cos(x[i]*M_PI/180);
    }
    plot.data3DCurve(x, y, z, NUM);
    plot.plotting();
    return 0;
}
```

Output

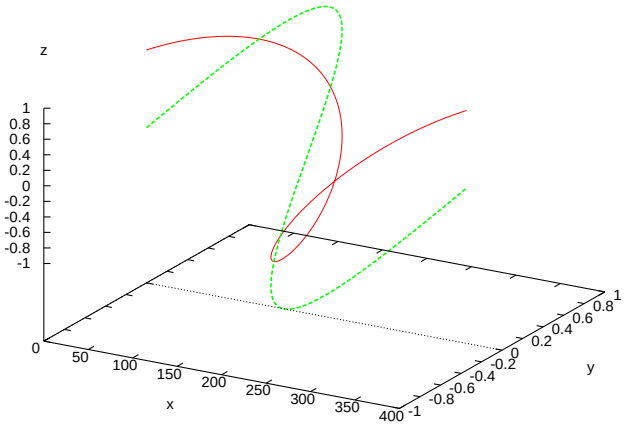


Example 2
Compare with output for examples in **CPlot**::**data3D**().

```
#include <math.h>
#include <chplot.h>
#ifndef M_PI
#define M_PI            3.14159265358979323846
#endif

#define NUM 360
int main() {
    double x[NUM], y[NUM], z1[NUM], z2[NUM];
    int i;
    class CPlot plot;

    for(i=0; i<360; i++) {
```

```
        x[i]  = i;
        y[i]  = sin(x[i]*M_PI/180);
        z1[i] = cos(x[i]*M_PI/180);
        z2[i] = y[i];
    }
    plot.data3DCurve(x, y, z1, NUM);
    plot.data3DCurve(x, y, z2, NUM);
    plot.plotType(PLOT_PLOTTYPE_LINES, 1, 0, 3); /* set the second data set to
                                                    use the default line type
                                                    and a line width three
                                                    times the default */
    plot.plotting();
    return 0;
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**plotting**(), **plotxyz**().

CPlot::data3DSurface

Synopsis
#include <chplot.h>
int data3DSurface(**double** *x*[], **double** *y*[], **double** *z*[], **int** n, **int** m);

Purpose
Add a set of data for 3D surface to an instance of **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x a one-dimensional array of size n used for the x axis of the plot.

y a one-dimensional array of size m used for the y axis of the plot.

z a one-dimensional array of size nXm

n the number of elements for array *x*.

m the number of elements for array *y*.

Description
Add a set of data for 3D surface plot to an instance of **CPlot** class. If one-dimensional array x has the number of elements of size n, and *y* has size m, z shall be a one-dimensional array of size $n_z = n \cdot m$. In a Cartesian coordinate system, arrays *x*, *y*, and z represent values in X-Y-Z coordinates, respectively. In a cylindrical coordinate system, arrays *x*, *y*, and *z* represent θ), z, and r coordinates, respectively. In a spherical coordinate system, arrays *x*, *y*, and z represent θ), ϕ, and r coordinates, respectively. Hidden line removal is enabled automatically (see **CPlot**::**removeHiddenLine**()). If it is desired to plot both grid data and non-grid data on the same plot, hidden line removal should be disabled manually after all data are added. Data points with a z value of NaN are internally removed before plotting occurs. "Holes" in a data set can be constructed by manually setting elements of *z* to this value.

It is important to note that for a 3D grid, the ordering of the z data is important. For calculation of the z values, the *x* value is held constant while *y* is cycled through its range of values. The *x* value is then incremented and *y* is cycled again. This is repeated until all the data are calculated. So, for a 10x20 grid the data shall be ordered as follows:

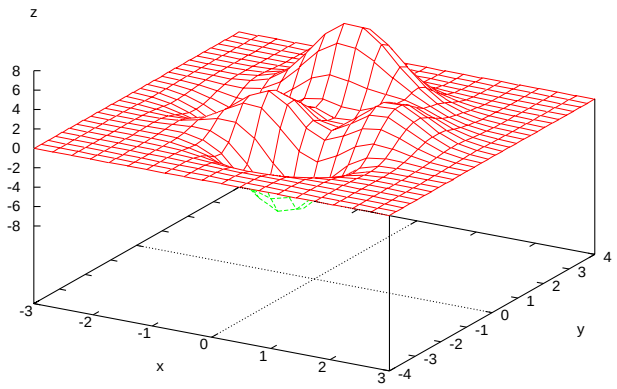
x1 y1 z1
x1 y2 z2
x1 y3 z3
.
.
.
x1 y18 z18
x1 y19 z19
x1 y20 z20
x2 y1 z21
x2 y2 z22
x2 y3 z23
.
.
.
x10 y18 z198
x10 y19 z199
x10 y20 z200

Example 1
Compare with output for examples in **CPlot**:**data3D**(), **CPlot**::**arrow**(), **CPlot**::**contourLabel**(), **CPlot**::**grid**(), **CPlot**::**removeHiddenLine**(), **CPlot**:**size3D**(), **CPlot**::**changeViewAngle**(), and **CPlot**::**ticsLevel**().

```
#include <math.h>
#include <chplot.h>

#define N 20
```

```
#define M 30
int main() {
    double x[N], y[M], z[N*M];
    int i,j;
    class CPlot plot;

    for(i=0; i<N; i++) {
      x[i] = -3 + i*6/19.0;  // linspace(x, -3, 3)
    }
    for(i=0; i<M; i++) {
      y[i] = -4 + i*8/29.0;  // linspace(y, -4, 4)
    }
    for(i=0; i<N; i++) {
        for(j=0; j<M; j++) {
            z[M*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3DSurface(x, y, z, N, M);
    plot.plotting();
    return 0;
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**dataFile**(), **CPlot**::**plotting**(), **plotxyz**().

CPlot::dataFile

Synopsis
#include <chplot.h>
int dataFile(**string_t** *file*);

Purpose
Add a data file to an existing instance of the **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameter
file name of the file to be plotted.

Description
Add a data file to an existing plot variable. Each data file corresponds to a single data set. The data file should be formatted with each data point on a separate line. 2D data are specified by two values per point. An empty line in a 2D data file causes a break of the curve in the plot. Multiple curves can be plotted in this manner, however the plot style will be the same for all curves. 3D data are specified by three values per data point. For a 3D grid or surface data, each row is separated in the data file by a blank line. The symbol # will comment out a subsequent text terminated at the end of a line in a data file. For example, a 3 x 3 grid would be represented as follows:

```
# This is a comment line
x1   y1   z1
x1   y2   z2
x1   y3   z3

x2   y1   z4
x2   y2   z5
x2   y3   z6

x3   y1   z7
x3   y2   z8
x3   y3   z9
```

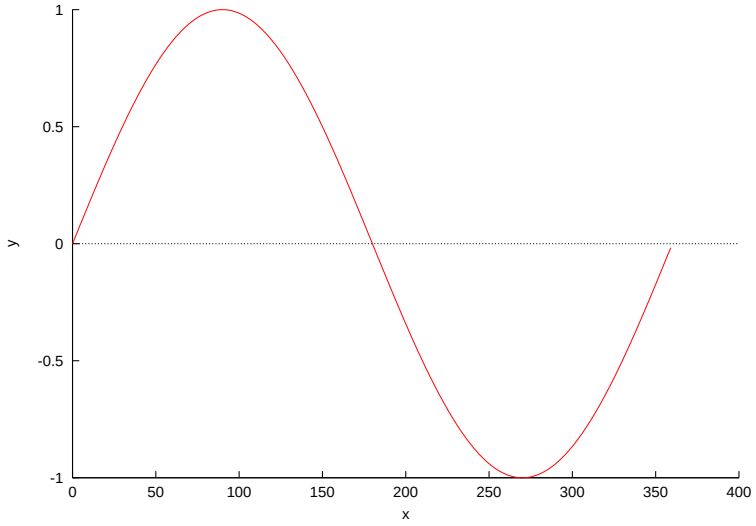
Two empty lines in the data file will cause a break in the plot. Multiple curves or surfaces can be plotted in this manner, however, the plot style will be the same for all curves or surfaces. Member function **CPlot**::**dimension**(3) must be called before 3D data file can be added.

Example

```
#include <stdio.h>
#include <chplot.h>
#include <math.h>

int main() {
    string_t file;
    int i;
    class CPlot plot;
    FILE *out;

    file = tmpnam(NULL);              //Create temporary file.
    out=fopen (file,"w");             //Write data to file.
    for(i=0;i<=359;i++) fprintf(out,"%i %f \n",i,sin(i*M_PI/180));
    fclose(out);
    plot.dataFile(file);
    plot.plotting();
    remove(file);
```

```
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**, **CPlot**::**data3DCurve**, **CPlot**::**data3DSurface**, **CPlot**::**outputType**(), **CPlot**::**plotting**(), **plotxyf**(), **plotxyzf**().

CPlot::deletePlots

Synopsis
#include <chplot.h>
void deletePlots();

Purpose
Delete all plot data and reinitialize an instance of the class to default values.

Return Value
None.

Parameters
None.

Description
This function frees all memory associated with previously allocated plot data, text strings, arrows, points, lines, polygons, rectangles, circles, and labeled tic-marks. This function also resets all plotting options to their default values. This function allows for the reuse of a single instance of the **CPlot** class to create multiple plots. This function is used internally by **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyz**(), **plotxyf**(), **plotxyzf**().

See Also
CPlot::**arrow**(), **CPlot**::**circle**(), **CPlot**::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**ticsLabel**(), **CPlot**::**line**(),

CPlot::**point**(), **CPlot**::**polygon**, **CPlot**::**rectangle**(), **CPlot**::**text**(), **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyz**(), **plotxyf**(), **plotxyzf**().

CPlot::dimension

Synopsis
#include <**chplot.h**>
void dimension(**int** *dim*);

Purpose
Set plot dimension to 2D or 3D.

Return Value
None.

Parameter
dim 2 for 2D and 3 for 3D. Default is 2.

Description
Set the dimension of the plot. The plot dimension should be set before data are added to the plot if member functions **CPlot**::dataThreeD(), **CPlot**::dataThreeDCurve(), or **CPlot**::dataThreeDSurface() are not called before. This member function must be used when 3D plotting data are added by **CPlot**::dataFile() and **CPlot**::**polygon**().

Example
See **CPlot**::**polygon**().

See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**polygon**().

CPlot::displayTime

Synopsis
#include <**chplot.h**>
void displayTime(**double** *x_offset*, **double** *y_offset*);

Purpose
Display the current time and date.

Return Value
None.

Parameters
x_offset Offset of the time-stamp in the x direction from its default location.

y_offset Offset of the time-stamp in the y direction from its default location.

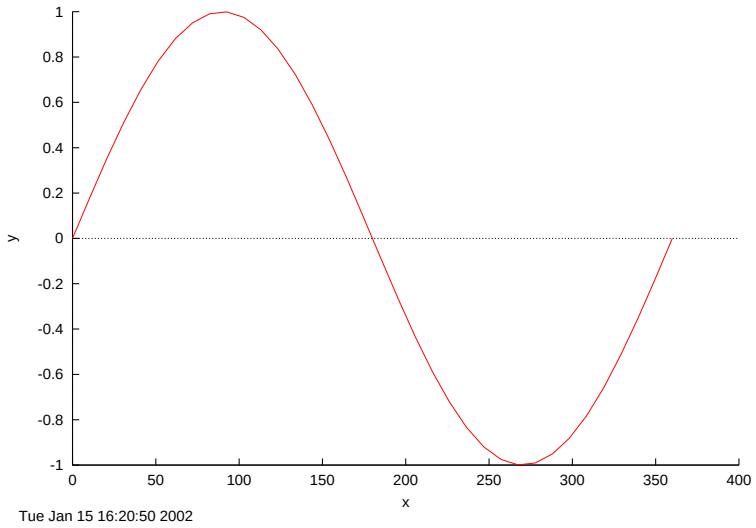
Description
This function places the current time and date near the left margin. The exact location is device dependent.

The offset values, *x_offset* and *y_offset*, are in screen coordinates and are measured in characters. For example, if both *x_offset* and *y_offset* are 2, the time will be moved approximately two characters to the right and two characters above the default location.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.displayTime(10,0);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



CPlot::getLabel

Synopsis
#include <**chplot.h**>
char * **getLabel**(**int** `axis`);

Purpose
Get the label for a plot axis.

Return Value
The label of the axis.

Parameters
axis The axis with its label to be obtained. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

Description
Get the label of a plot axis.

Example

```
#include <math.h>
#include <chplot.h>

int main() {
    class CPlot plot;
    string_t xlabel, ylabel, zlabel, title;

    plot.label(PLOT_AXIS_X, "xlabel");
    xlabel = plot.getLabel(PLOT_AXIS_X);
    ylabel = plot.getLabel(PLOT_AXIS_Y);
    zlabel = plot.getLabel(PLOT_AXIS_Z);
    printf("xlabel = %s\n", xlabel);
    printf("ylabel = %s\n", ylabel);
    printf("zlabel = %s\n", zlabel);
    title = plot.getTitle();
    printf("title = %s\n", title);
    plot.title("New Title");
    title = plot.getTitle();
    printf("title = %s\n", title);
}
```

Output

```
xlabel = xlabel
ylabel = y
zlabel = z
title = (null)
title = New Title
```

See Also
CPlot::**label**(), **CPlot**::**title**(), **CPlot**::**getTitle**().

CPlot::getOutputType

Synopsis
#include <chplot.h>
plotinfo_t getOutputType();

Purpose
Get the output type for a plot.

Return Value
The output type in one of the following forms:

PLOT_OUTPUTTYPE_DISPLAY Display the plot on the screen.

PLOT_OUTPUTTYPE_STREAM Output the plot as a standard output stream.

PLOT_OUTPUTTYPE_FILE Output the plot to a file in one of a variety of formats.

Parameters None
Description
Get the output type of a plot.

Example

```
#include <math.h>
#include <chplot.h>

int main() {
    class CPlot plot;
    plotinfo_t outputtype;

    outputtype = plot.getOutputType();
    if(outputtype == PLOT_OUTPUTTYPE_DISPLAY)
      printf("output is displayed\n");
    else if(outputtype == PLOT_OUTPUTTYPE_STREAM)
      printf("output is stdout stream\n");
    else if(outputtype == PLOT_OUTPUTTYPE_FILE)
      printf("output is in a file\n");
    return 0;
}
```

Output

```
output is displayed
```

See Also
CPlot::**outputType**().

CPlot::getSubplot

Synopsis
#include <**chplot.h**>
class CPlot* **getSubplot**(**int** *row*, **int** *col*);

Purpose
Get a pointer to an element of a subplot.

Return Value
Returns a pointer to the specified element of the subplot.

Parameters
row The row number of the desired subplot element. Numbering starts with zero.

col The column number of the desired subplot element. Numbering starts with zero.

Description
After the creation of a subplot using **CPlot**::**subplot**(), this function can be used to get a pointer to an element of the subplot. This pointer can be used as a **CPlot** pointer normally would be. Addition of data sets or selection of plotting options are done normally. However, each option only effects the subplot element currently pointed to.

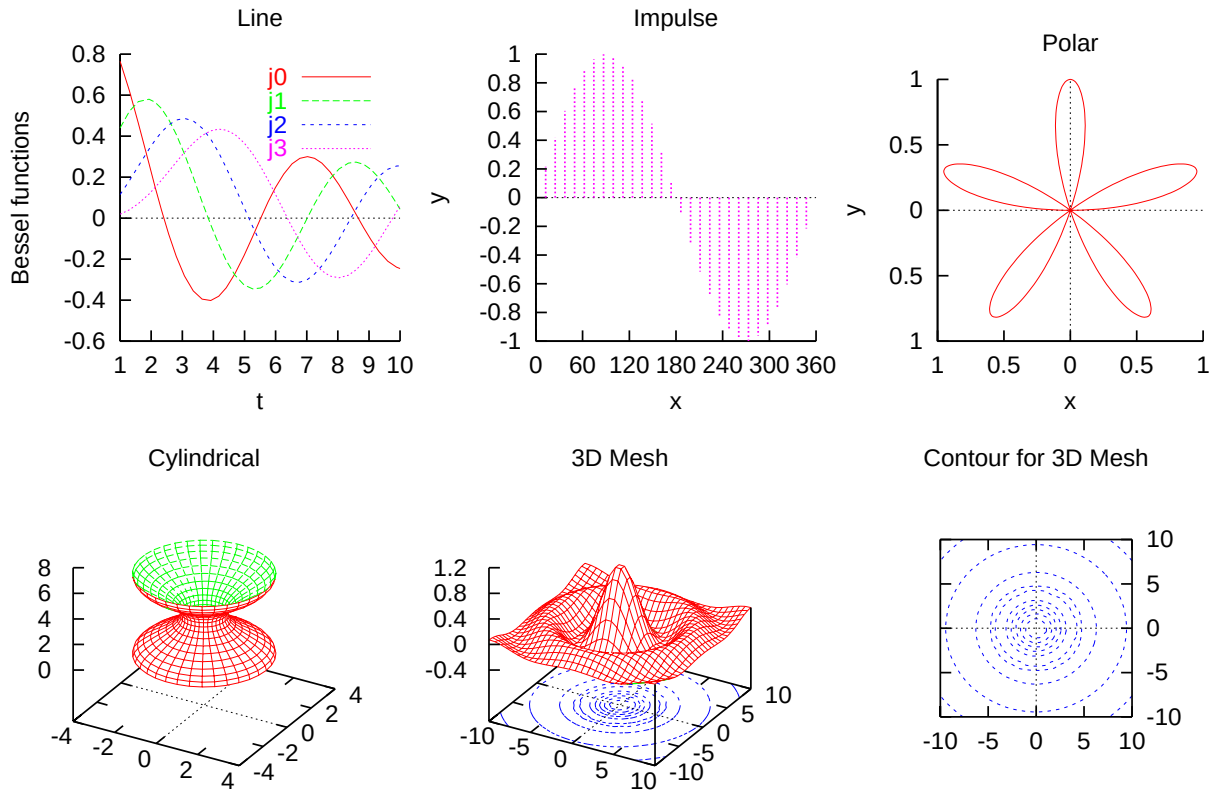
Example

```
#include <chplot.h>
#include <math.h>

#define NVAR 4
#define POINTS 50
int main() {
    void derivs(double t, double y[], double dydt[]) {
      dydt[0] = -y[1];
      dydt[1]=y[0]-(1.0/t)*y[1];
      dydt[2]=y[1]-(2.0/t)*y[2];
      dydt[3]=y[2]-(3.0/t)*y[3];
    }

    double t0=1, tf=10, y0[NVAR];
    double t[POINTS], y1[NVAR][POINTS];
    int  points;
    int  datasetnum=0, line_type=4, line_width = 2;
    array double x2[30], y2[30];
    array double theta3[360], r3[360];
    double r;
    double theta4[36], z4[20], r4[720];
    double x5[30], y5[30], z5[900];

    int i, j;
    class CPlot subplot, *spl;

    /* plot 1 */
    y0[0]=j0(t0);
    y0[1]=j1(t0);
    y0[2]=jn(2,t0);
    y0[3]=jn(3,t0);
    points = odesolve(t, y1, derivs, t0, tf, y0);

    /* plot 2 */
    linspace(x2, 0, 360);
    y2 = sin(x2*M_PI/180);

    /* plot 3 */
    linspace(theta3, 0, M_PI);
    r3 = sin(5*theta3);
```

```
/* plot 4 */
linspace(theta4, 0, 360);
linspace(z4, 0, 2*M_PI);
for(i=0; i<36; i++) {
    for(j=0; j<20; j++) {
        r4[i*20+j] = 2+cos(z4[j]);
    }
}

/* plot 5 */
linspace(x5, -10, 10);
linspace(y5, -10, 10);

for(i=0; i<30; i++) {
    for(j=0; j<30; j++) {
        r = sqrt(x5[i]*x5[i]+y5[j]*y5[j]);
        z5[30*i+j] = sin(r)/r;
    }
}

subplot.subplot(2,3); /* create 2 x 3 subplot */
spl = subplot.getSubplot(0,0); /* get subplot (0,0) */
spl->title("Line");
spl->label(PLOT_AXIS_X,"t");
spl->label(PLOT_AXIS_Y,"Bessel functions");
spl->data2D(t, y1);
spl->legend("j0", 0);
spl->legend("j1", 1);
spl->legend("j2", 2);
spl->legend("j3", 3);
spl = subplot.getSubplot(0,1); /* get subplot (0,1) */
spl->title("Impulse");
spl->axisRange(PLOT_AXIS_X, 0, 360, 60);
spl->data2D(x2, y2);
spl->plotType(PLOT_PLOTTYPE_IMPULSES, datasetnum, line_type, line_width);
spl = subplot.getSubplot(0,2); /* get subplot (0,2) */
spl->title("Polar");
spl->axisRange(PLOT_AXIS_XY, -1, 1, .5);
spl->data2D(theta3, r3);
spl->polarPlot(PLOT_ANGLE_RAD);
spl->sizeRatio(-1);
spl = subplot.getSubplot(1,0); /* get subplot (1,0) */
spl->title("Cylindrical");
spl->data3D(theta4, z4, r4);
spl->coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
spl->axisRange(PLOT_AXIS_Z, 0, 8, 2);
spl->axisRange(PLOT_AXIS_XY, -4, 4, 2);
spl->label(PLOT_AXIS_XYZ, NULL);
spl = subplot.getSubplot(1,1); /* get subplot (1,1) */
spl->title("3D Mesh");
spl->axisRange(PLOT_AXIS_X, -10, 10, 5);
spl->axisRange(PLOT_AXIS_Y, -10, 10, 5);
spl->axisRange(PLOT_AXIS_Z, -.4, 1.2, .4);
spl->data3D(x5, y5, z5);
spl->label(PLOT_AXIS_XYZ, NULL);
spl->contourMode(PLOT_CONTOUR_BASE);
spl = subplot.getSubplot(1,2); /* get subplot (1,2) */
spl->data3D(x5, y5, z5);
spl->title("Contour for 3D Mesh");
```

```
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->changeViewAngle(0, 0);
    spl->contourMode(PLOT_CONTOUR_BASE);
    spl->axisRange(PLOT_AXIS_XY, -10, 10, 5);
    spl->showMesh(PLOT_OFF);
    subplot.plotting();
}
```

Output



See Also
CPlot::**subplot**().

CPlot::getTitle

Synopsis
#include <**chplot.h**>
string_t getTitle(void);

Purpose
Get the plot title.

Return Value
The plot title.

Parameters
None.

Description
Get the title of the plot. If no title, `NULL` will be returned.

Example
see **CPlot**::**getLabel**().

See Also
CPlot::**label**(), **CPlot**::**getLabel**(), **CPlot**::**title**().

CPlot::grid

Synopsis
#include <**chplot.h**>
void grid(**int** *flag*, ... /* [**int** *grid_type*] */);

Syntax
grid(*flag*)
grid(*flag*, *grid_type*)

Purpose
Enable or disable the display of a grid on the xy plane.

Return Value
None.

Parameters
flag This parameter can be set to:

PLOT_ON Enable the display of the grid.

PLOT_OFF Disable the display of the grid.

grid_type The grid type can be set to:

PLOT_GRID_RECTANGULAR Draw a rectangular grid in a 2D/3D plot.

PLOT_GRID_POLAR Draw a polar grid in a 2D plot.

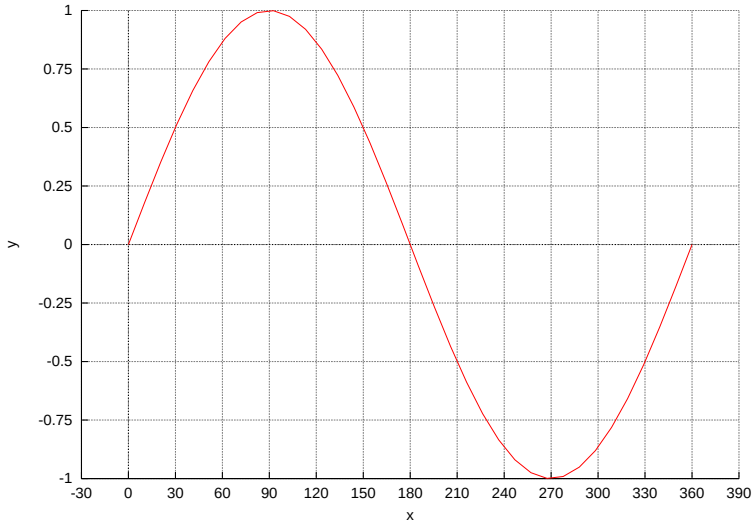
Description
Enable or disable the display of a grid on the xy plane. By default, the grid is off and the grid type is rectangular. **PLOT_GRID_POLAR** is only valid for 2D plots.

Example 1
Compare with the output for the example in **CPlot**::**axisRange**().

```
#include <math.h>
#include <chplot.h>
```

```
int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    int i;
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);                          // Y-axis data.
    plot.axisRange(PLOT_AXIS_X, -30, 390, 30);
    plot.axisRange(PLOT_AXIS_Y, -1, 1, .25);
    plot.grid(PLOT_ON);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output

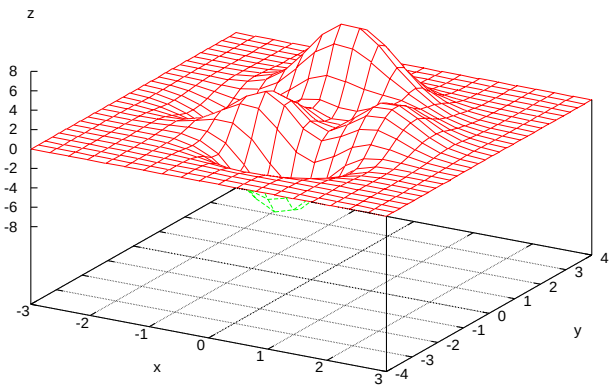


Example 2
Compare with the output for examples in **CPlot**::**data3D**() and **CPlot**::**data3DSurface**().

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.grid(PLOT_ON);
```

```
    plot.plotting();
}
```

Output



CPlot::isUsed

Synopsis
#include <chplot.h>
int isUsed();

Purpose
Test if an instance of the **CPlot** class has been used.

Return Value
Returns `true` if the **CPlot** instance has been previously used, returns `false` otherwise.

Parameters
None.

Description
This function determines if an instance of the **CPlot** class has previously been used. The function actually tests if data have previously been added to the instance, by checking if an internal pointer in the CPlot class is NULL. If the pointer is not NULL, then the instance has been used. This function is used internally by **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyz**(), **plotxyf**(), and **plotxyzf**().

See Also
fplotxy(), **fplotxyz**(), **plotxy**(), **plotxyz**(), **plotxyf**(), and **plotxyzf**().

CPlot::label

Synopsis
#include <**chplot.h**>
void label(**int** `axis`, **string_t** *label*);

Purpose
Set the labels for the plot axes.

Return Value
None.

Parameters
axis The axis to be set. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

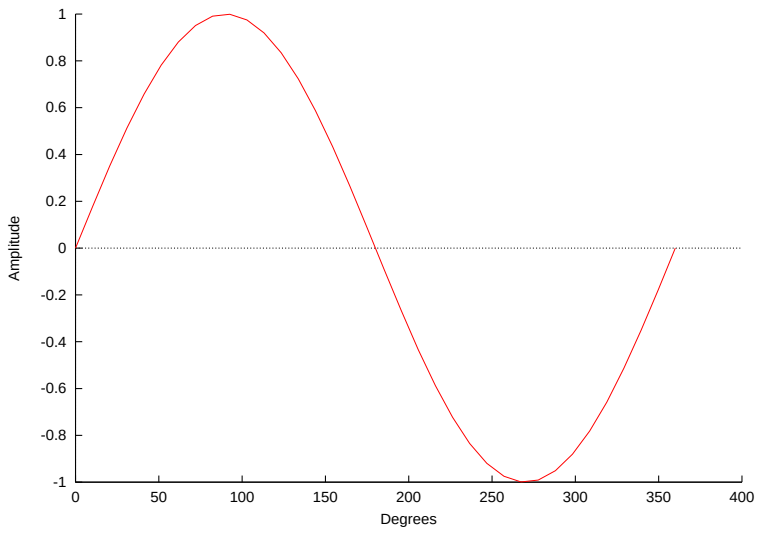
label label of the axis.

Description
Set the plot axis labels. The label for the z-axis can be set only for a 3D plot. If no label is desired, the value of `NULL` can be used as an argument. For example, function call `plot.label(PLOT_AXIS_XY, NULL)` will suppress the labels for both x and y axes of the plot. By default, labels are "x", "y", and "z" for the first x, y, and z axes, respectively.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t xlabel="Degrees",
             ylabel="Amplitude";
    class CPlot plot;
    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also
CPlot::**getLabel**(), **CPlot**::**title**().
CPlot::**getTitle**().

CPlot::legend

Synopsis
#include <**chplot.h**>
void legend(**string_t** *legend*, **int** *num*);

Purpose
Specify a legend string for a previously added data set.

Return Value
None.

Parameters
legend The legend string.

num The data set the legend is added to.

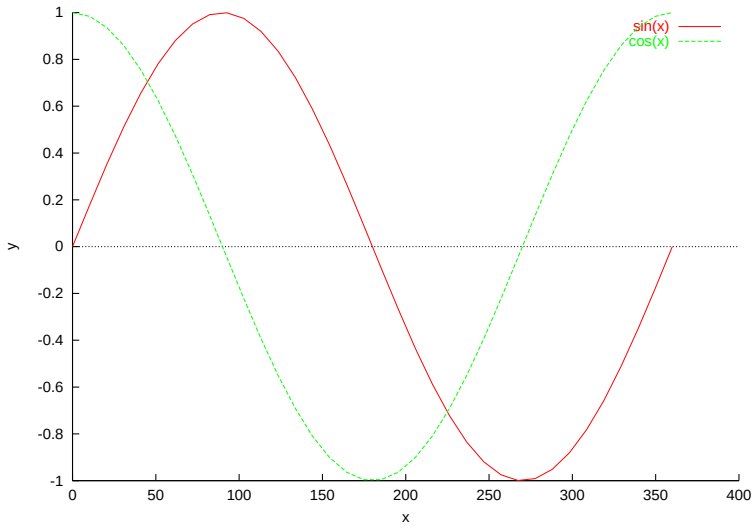
Description
The legend string is added to a plot legend located in the upper-right corner of the plot by default. Numbering of the data sets starts with zero. New legends will replace previously specified legends.
This member function shall be called after plotting data have been added by member functions **CPlot**::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**().

Bugs
The legend string may not be displayed for a data set with a single point. To suppress the legend string completely, passs value of `" "` to the argument `legend`. Use **CPlot**::**text**() to add a legend for the data set.

Example
Compare with the output for the example in **CPlot**::**legendLocation**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints], y2[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    y2= cos(x*M_PI/180);
    plot.data2D(x, y);
    plot.data2D(x, y2);
    plot.legend("sin(x)", 0);
    plot.legend("cos(x)", 1);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**dataFile**(), **CPlot**::**legendLocation**().

CPlot::legendLocation

Synopsis
#include <**chplot.h**>
void legendLocation(**double** *x*, **double** *y*, ... /* [**double** *z*] */);

Syntax
legendLocation(*x*, *y*)
legendLocation(*x*, *y*, *z*)

Purpose
Specify the plot legend (if any) location

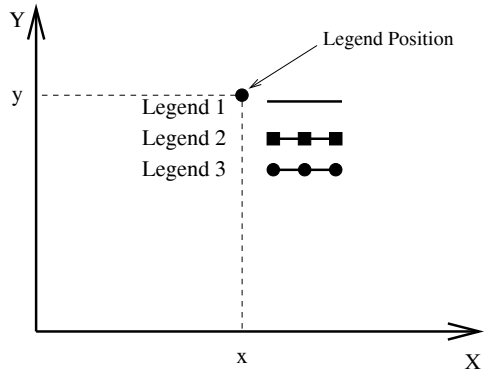
Return Value
None.

Parameters
x The x coordinate of the legend.

y The y coordinate of the legend.

z The z coordinate of the legend.

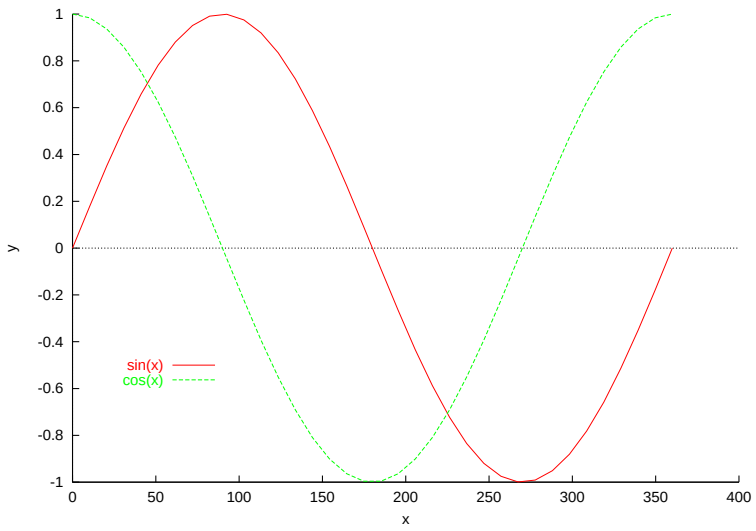
Description
This function specifies the position of the plot legend using plot coordinates. The position specified is the location of the top part of the space separating the markers and labels of the legend, as shown below. By default, the location of the legend is near the upper-right corner of the plot.



Example
Compare with the output for the example in **CPlot**::**legend**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints], y2[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    y2= cos(x*M_PI/180);
    plot.data2D(x, y);
    plot.data2D(x, y2);
    plot.legend("sin(x)", 0);
    plot.legend("cos(x)", 1);
    plot.legendLocation(60, -.5);
    plot.plotting();
}
```

Output



See Also
CPlot::**legend**().

CPlot::line

Synopsis
#include <**chplot.h**>
int line(**double** *x1*, **double** *y1*, **double** *z1*, **double** *x2*, **double** *y2*, **double** *z2*);

Purpose
Add a line to a plot.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x1 The x coordinate of the first endpoint of the line.

y1 The y coordinate of the first endpoint of the line.

z1 The z coordinate of the first endpoint of the line. This parameter is ignored for 2D plots.

x2 The x coordinate of the second endpoint of the line.

y2 The y coordinate of the second endpoint of the line.

z2 The z coordinate of the second endpoint of the line. This parameter is ignored for 2D plots.

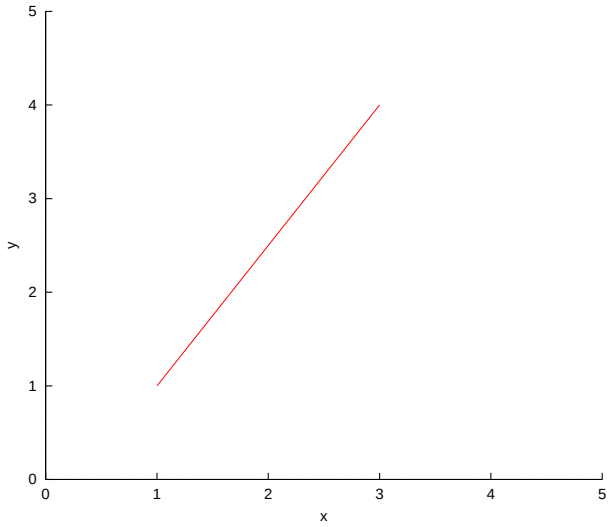
Description
This function adds a line to a plot. It is a convenience function for creation of geometric primitives. A line added with this function is counted as a data set for later calls to **CPlot**::**legend**() and **CPlot**::**plotType**(). For 2D rectangular and 3D cartesian plots, (*x1*, *y1*, *z1*) and (*x2*, *y2*, *z2*) are the coordinates of the endpoints of the line, specified in units of the x, y, and z axes. However, for 2D plots, *z1* and *z2* are ignored. For 2D polar and 3D cylindrical plots, the endpoints are specified in polar coordinates where *x* is θ, *y* is r, and *z* is

unchanged. Again, for 2D plots, *z1* and *z2* are ignored. For 3D plots with spherical coordinates *x* is θ, *y* is ϕ and *z* is r.

Example 1

```
#include <chplot.h>

int main(){
    double x1 = 1, y1 = 1, x2 = 3, y2 = 4;
    class CPlot plot;

    plot.line(x1, y1, 0, x2, y2, 0);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, 0, 5);
    plot.plotting();
}
```

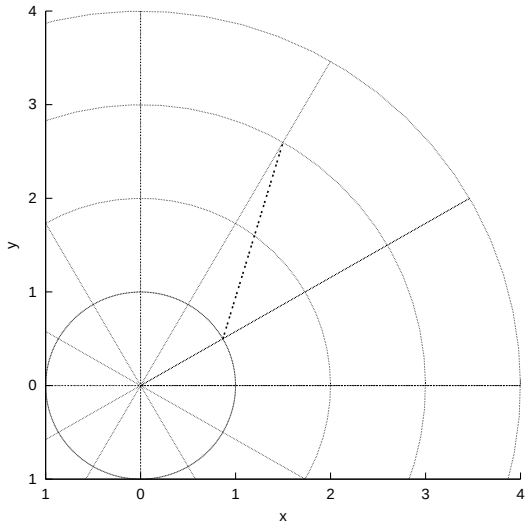
Output



Example 2

```
#include <chplot.h>
#include <math.h>

int main(){
    double theta1 = 30, theta2 = 60, r1 = 1, r2 = 3;
    class CPlot plot;

    plot.grid(PLOT_ON, PLOT_GRID_POLAR);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.line(theta1, r1, 0, theta2, r2, 0);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0, 7, 3);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, -1, 4);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**circle**(), **CPlot**::**outputType**(), **CPlot**::**plotType**(), **CPlot**::**point**(), **CPlot**::**polygon**(), **CPlot**::**rectangle**().

CPlot::margins

Synopsis
#include <**chplot.h**>
void margins(**double** *left*, **double** *right*, **double** *top*, **double** *bottom*);

Purpose
Set the size of the margins around the edge of the plot.

Return Value
None.

Parameters
left The size of the left margin in character width.

right The size of the right margin in character width.

top The size of the top margin in character height.

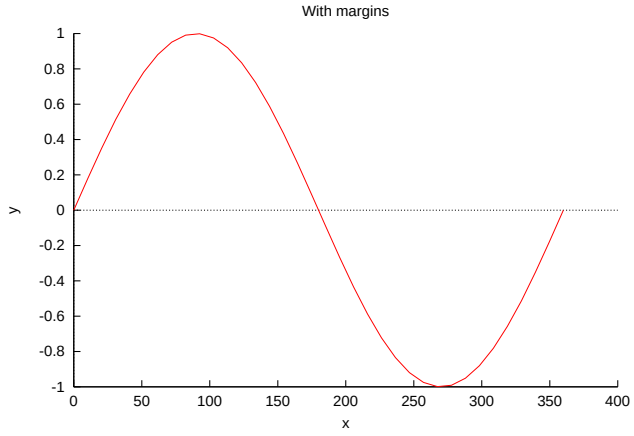
bottom The size of the bottom margin in character height.

Description
By default, the plot margins are calculated automatically. They can be set manually with this function. Specifying a negative value for a margin causes the default value to be used.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.margins(15, 10, 5, 7);
    plot.title("With margins");
    plot.plotting();
}
```

Output



See Also
CPlot::**borderOffsets**().

CPlot::outputType

Synopsis
#include <**chplot.h**>
void outputType(**int** *outputtype*, ... /* [**string_t** *terminal*, **string_t** *filename*] */);

Syntax
outputType(*outputtype*)
outputType(**PLOT_OUTPUTTYPE_DISPLAY**)
outputType(**PLOT_OUTPUTTYPE_STREAM**, *terminal*)
outputType(**PLOT_OUTPUTTYPE_FILE**, *terminal*, *filename*)

Purpose
Set the output type for a plot.

Return Value
None.

Parameters
outputtype This can have any of the following values:

PLOT_OUTPUTTYPE_DISPLAY Display the plot on the screen. The plot is displayed in its own separate window. A plot window can be closed by pressing the 'q' key in the X-Windows system.

PLOT_OUTPUTTYPE_STREAM Output the plot as a standard output stream. This output type is useful for CGI (Common Gateway Interface) when a Ch program is used as CGI script in a Web server.

PLOT_OUTPUTTYPE_FILE Output the plot to a file in one of a variety of formats. If this output option is selected two additional arguments are necessary: the *terminal* type and *filename*.

terminal Supported terminal types when gnuplot is used as a plotting engine are as follow:

Terminal	Description
aifm	Adobe Illustrator 3.0.
corel	EPS format for CorelDRAW.
dxf	AutoCAD DXF.
dxy800a	Roland DXY800A plotter.
eepic	Extended LaTeXpicture.
emtex	LaTeXpicture with emTeX specials.
epson-180dpi	Epson LQ-style 24-pin printer with 180dpi.
epson-60dpi	Epson LQ-style 24-pin printers with 60dpi.
epson-lx800	Epson LX-800, Star NL-10 and NX-100.
excl	Talaris printers.
fig	Xfig 3.1.
gif	GIF file format.
gpic	gpic/groff package.
hp2648	Hewlett Packard HP2647 an HP2648.
hp500c	Hewlett Packard DeskJet 500c.
hpdj	Hewlett Packard DeskJet 500.
hpgl	HPGL output.
hpljii	HP LaserJet II.
hppj	HP PaintJet and HP3630 printers.
latex	LaTeXpicture.
mf	MetaFont.
mif	Frame Maker MIF 3.00.
nec-cp6	NEC CP6 and Epson LQ-800.
okidata	9-pin OKIDATA 320/321 printers.
pcl5	Hewlett Packard LaserJet III.
pbm	Portable BitMap.

png	Portable Network Graphics.
postscript	Postscript.
pslatex	LaTeXpicture with postscript specials.
pstricks	LaTeXpicture with PSTricks macros.
starc	Star Color Printer.
tandy-60dpi	Tandy DMP-130 series printers.
texdraw	LaTeXtexdraw format.
tgif	TGIF X-Window drawing format.
tpic	LaTeXpicture with tpic specials.

`aifm` Output an Adobe Illustrator 3.0 file. The format for the *terminal* string is:

```
"aifm [colormode] [\"fontname\"] [fontsize]"
```

`colormode` can be `color` or `monochrome`.
`fontname` is the name of a valid PostScript font.
`fontsize` is the size of the font in points.

Defaults are `monochrome`, `"Helvetica"` and 14pt.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "aifm color", "plot.aif");`

`corel` Output EPS format for CorelDRAW.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "corel", "plot.eps");`

`dxf` Output AutoCAD DXF file.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "dxf", "plot.dxf");`

`dxy800a` Output file for Roland DXY800A plotter.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "dxy800a", "plot.dxy");`

`eepic` Output extended LaTeXpicture.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "eepic", "plot.tex");`

`emtex` Output LaTeXpicture with emTeX specials.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "emtex", "plot.tex");`

`epson-180dpi` Epson LQ-style 24-pin printers with 180dpi.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "epson-180dpi", "plot");`

`epson-60dpi` Epson LQ-style 24-pin printers with 60dpi.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "epson-60dpi", "plot");`

`epson-lx800` Epson LX-800, Star NL-10 and NX-100.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "epson-lx800", "plot");`

`excl` Talaris printers such as EXCL Laser printer and 1590.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "excl", "plot");`

`fig` Xfig 3.1 file. The format for the *terminal* string is:

```
"fig [colormode] [size] [pointsmax num_points] [orientation]
[units] [fontsize fntsize] [size xsize ysize] [thickness width]
[depth layer]"
```

`colormode` can be `monochrome` or `color`
`size` can be `small` or `big`
`num_points` is the maximum number of points per polyline.
`orientation` can be `landscape` or `portrait`. `units` can be `metric` or `inches`.
`fontsize` is the size of the text font in points. Must be preceded by the `fontsize` keyword.
`xsize` and `ysize` set the size of the drawing area. Must be preceded by the `size` keyword.
`width` is the line thickness. Must be preceded by the `thickness` keyword.
`layer` is the line depth. Must be preceded by the `depth` keyword.

Default values are: `monochrome small pointsmax 1000 landscape inches fontsize 10 thickness 1 depth 10`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "fig color big", "plot.fig");`

`gif` GIF file format. The format for the *terminal* string is:

```
"gif [transparent] [interlace] [font_size] [size x,y]
[color0 color1 color2 ...]
```

Specifying the `transparent` keyword will generate a transparent GIF. By default, white is the transparent color.
Specifying the `interlace` key word will generate an interlaced GIF.
`font_size` is `small` (6x12 pixels), `medium` (7x13 pixels), or `large` (8x16 pixels).
`x` and `y` are the image size in pixels. Must be preceded by the `size` keyword.
`color`s are specified in the format ”xrrggbb” where x is the character ”x” and ”rrggbb” are the

RGB components of the color in hexadecimal. A maximum of 256 colors can be set. If the GIF is transparent, the first color is used as the transparent color.

The default values are: `small size 640,480`

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "gif size 1024,768", "plot.gif");`

`gpic` Output for use with the Free Software Foundation gpic/groff p package. The format for the *terminal* string is:

`"gpic [x] [y]"`

where `x` and `y` are the location of the origin. Default is (0,0).

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "gpic 5 5", "plot.gpic");`

`hp2633a` Hewlett Packard HP2623A.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hp2633a", "plot");`

`hp2648` Hewlett Packard HP2647 an HP2648.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hp2648", "plot");`

`hp500c` Hewlett Packard DeskJet 500c. The format for the *terminal* string is:

`"hp500c [resolution] [compression]"`

`resolution` can be `75`, `100`, `150`, or `300` dpi.
`compression` can be `rle` or `tiff`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hp500c 100 tiff", "plot");`

`"hpdj`: Hewlett Packard DeskJet 500. The format for the *terminal* string is:

`"hp500c [resolution]"`

`resolution` can be `75`, `100`, `150`, or `300` dpi. Default is `75`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hpdj 100", "plot");`

`hpgl` Produces HPGL output for devices such as the HP7475A plotter. The format for the *terminal* string is:

```
"hpgl [num_of_pens] [eject]"
```

`num_of_pens` is the number of available pens. The default is 6.
`eject` is a keyword that tells the plotter to eject a page when done.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hpgl 4", "plot");`

`hpljii` HP LaserJet II

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hpljii", "plot");`

item[`hppj`] HP PaintJet and HP3630 printers. The format for the *terminal* string is:

```
"hppj [font]"
```

`font` can be `FNT5X9`, `FNT9x17`, or `FNT13X25`. Default is `FNT9x17`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "hppj FNT5X9", "plot");`

`latex` Output a LaTeXpicture. The format of the *terminal* string is:

```
"latex [font] [size]"
```

where `font` can be `courier` or `roman` and `size` can be any point size. Default is `roman` 10pt.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "latex", "plot.tex");`

`mf` Output file for the MetaFont program.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "mf", "plot.mf");`

`mif` Output Frame Maker MIF file format version 3.00. The format of the *terminal* string is:

```
"mif pentype curvetype"
```

`pentype` can be: `colour` or `monochrome`.

`curvetype` can be:

`polyline` curves drawn as continuous lines
`vectors` curves drawn as a collection of vectors

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "mif colour vectors", "plot.mif");`

`nec-cp6` Generic 24-pin printer such as NEC CP6 and Epson LQ-800.

The format for the *terminal* string is:

```
"nec-cp6a [option]
```

`option` can be `monochrome`, `colour`, or `draft`.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "nec-cp6a draft", "plot");`

`okidata` 9-pin OKIDATA 320/321 printers.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "okidata", "plot");`

`pcl5` Hewlett Packard LaserJet III. This actually produces HPGL-2. The format of the *terminal* string is:

```
"pcl5 [mode] [font] [fontsize]"
```

`mode` is `landscape` or `portrait`.
`font` is `stick`, `univers`, or `cg_times`.
`fontsize` is the font size in points.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "pcl5 landscape", "plot");`

`pbm` Output a Portable BitMap. The format for the *terminal* string is:

```
"pbm [fontsize] [colormode]"
```

`fontsize` is `small`, `medium`, or `large`.
`colormode` is `monochrome`, `gray`, or `color`.

Example:

```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "pbm medium gray",
"plot.pbm");
```

`png` Portable Network Graphics format. The format of the *terminal* string is:

```
"png [fontsize] [colormode]"
```

`fontsize` can be `small`, `medium`, or `large`.
`colormode` can be `monochrome`, `gray`, or `color`.

Default is `small` and `monochrome` with the output size of 640x480 pixel. Use member function **CPlot**::**size**() to change the size of the plot.

Example:
```
plot.outputType(PLOT_OUTPUTTYPE_FILE, "png large color",
"plot.png");
```

`postscript` This produces a postscript file. The format for the *terminal* string is:

```
"postscript [mode] [colormode] [dash] [\"fontname\"] [fontsize]"
```

`mode` can be `landscape`, `portrait`, `eps`, or `default`
`colormode` can be `color` or `monochrome`.
`dash` can be `solid` or `dashed`.
`fontname` is the name of a valid PostScript font.
`fontsize` is the size of the font in points.

The `default` mode is `landscape`, `monochrome`, `dashed`, `"Helvetica"`, 14pt.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE,"postscript eps \"Times\" 11", "plot.eps");`

`pslatex` Output LaTeXpicture with postscript specials.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "pslatex", "plot.tex");`

`pstricks` Output LaTeXpicture with PSTricks macros.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "pstricks", "plot.tex");`

`starc` Star Color Printer.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "starc", "plot");`

`tandy-60dpi` Tandy DMP-130 series printers.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "tandy-60dpi", "plot");`

`texdraw` Output LaTeXtexdraw format.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "texdraw", "plot.tex");`

`tgif` Output TGIF X-Window drawing format.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "tgif", "plot.tgif");`

`tpic` Output LaTeXpicture with tpic specials.

Example: `plot.outputType(PLOT_OUTPUTTYPE_FILE, "tpic", "plot.tex");`

filename The filename the plot is saved to. On machines that support `popen()` functions, the output can also be piped to another program by placing the '`|`' character in front of the command name and using it as the *filename*. For example, on Unix systems, setting *terminal* to "`postscript`" and *filename* to "`|lp`" could be used to send a plot directly to a postscript printer.

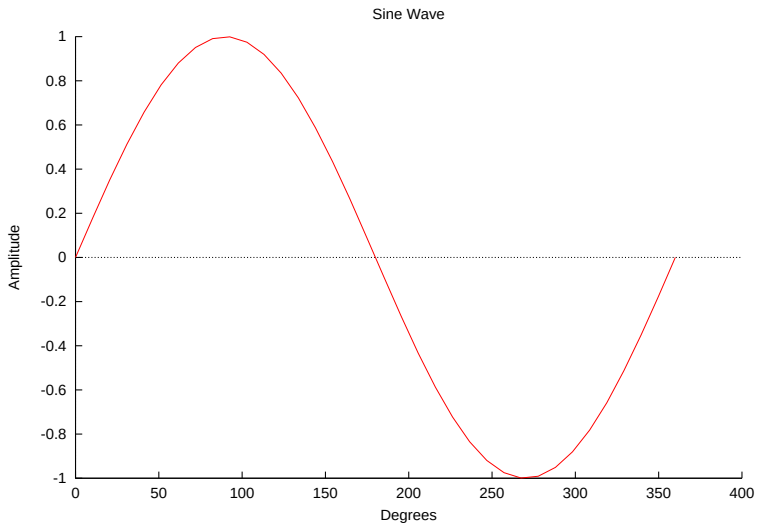
Description

This function is used to display a plot on the screen, save a plot to a file, or generate a plot to the stdout stream in GIF format for use on the World Wide Web. By default, the output type is **PLOT_OUTPUTTYPE_DISPLAY**.

Example 1

```
/* a plot is created in postscript file plot.eps */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t title="Sine Wave",                // Define labels.
             xlabel="Degrees",
             ylabel="Amplitude";
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);                       // Y-axis data.
    plot.title(title);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "plot.eps");
    plot.plotting();
}
```

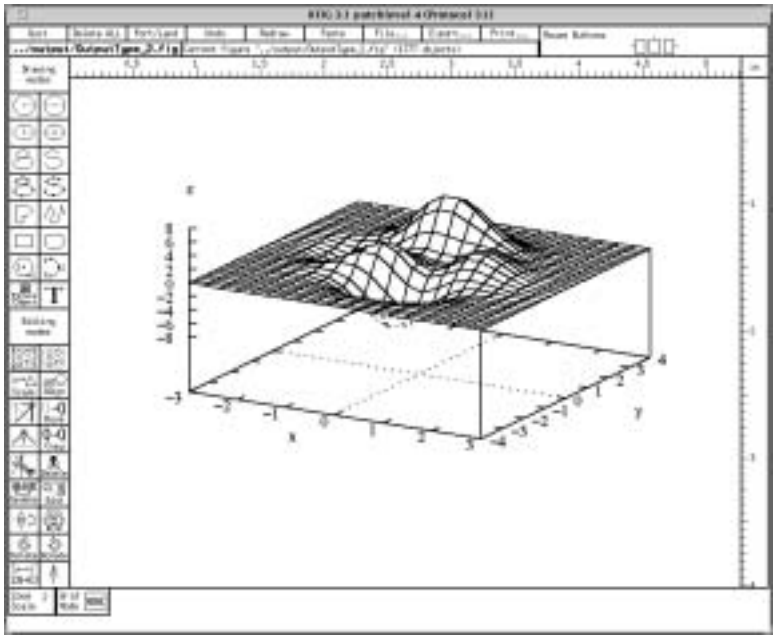
Output



Example 2

```
/* In this example, the plot is saved as an xfig file first.
   Next the xfig program is invoked. The plot can then be edited using xfig in Unix */
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.outputType(PLOT_OUTPUTTYPE_FILE, "fig", "../output/outputType_2.fig");
    plot.plotting();

    xfig ../output/outputType_2.fig&

}
```

Output



Example 3
To run this code as a CGI program in a Web server, place outputType.ch in your `cgi-bin` directory. If you place outputType.ch in a different directory, change `/cgi-bin` to specify the correct location. In your Web browser, open the html file.

```
#!/bin/ch
```

```
/* This flle is called outputType.ch, a plot is created in png format  */
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t title="Sine Wave",
             xlabel="Degrees",
             ylabel="Amplitude";
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    setbuf(stdout, NULL);
    printf("Content-type: image/png\n\n");
    plot.title(title);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.outputType(PLOT_OUTPUTTYPE_STREAM, "png color");
    plot.plotting();
}
```

outputType.html

```
<! this file is called outputType.html >
<html>
<head>
<title>
Example Plot
</title>
</head>

<img src="/cgi-bin/outputType.ch">

</html>
```

CPlot::plotType

Synopsis
#include <**chplot.h**>
void plotType(**int** *plot_type*, **int** *num*, ... /* [[**int** *line_type*, **int** *line_width*],
[**int** *point_type*, **int** *point_size*]] */);

Syntax
plotType(**PLOT_PLOTTYPE_LINES**, *num*)
plotType(**PLOT_PLOTTYPE_LINES**, *num*, *line_type*, *line_width*)
plotType(**PLOT_PLOTTYPE_IMPULSES**, *num*)
plotType(**PLOT_PLOTTYPE_IMPULSES**, *num*, *line_type*, *line_width*)
plotType([**PLOT_PLOTTYPE_STEPS**, *num*)
plotType([**PLOT_PLOTTYPE_STEPS**, *num*, *line_type*, *line_width*)
plotType(**PLOT_PLOTTYPE_FSTEPS**, *num*)

plotType(**PLOT_PLOTTYPE_FSTEPS**, *num*, *line_type*, *line_width*)
plotType(**PLOT_PLOTTYPE_HISTEPS**, *num*)
plotType(**PLOT_PLOTTYPE_HISTEPS**, *num*, *line_type*, *line_width*)
plotType(**PLOT_PLOTTYPE_POINTS**, *num*)
plotType(**PLOT_PLOTTYPE_POINTS**, *num*, *point_type*, *point_size*)
plotType(**PLOT_PLOTTYPE_LINESPOINTS**, *num*)
plotType(**PLOT_PLOTTYPE_LINESPOINTS**, *num*, *line_type*, *line_width*, *point_type*, *point_size*)
plotType(**PLOT_PLOTTYPE_DOTS**, *num*)
plotType(**PLOT_PLOTTYPE_DOTS**, *num*, *point_type*)

Purpose
Set the plot type for a data set.

Return Value
None.

Parameters
plot_type The plot type. Valid values are:

PLOT_PLOTTYPE_LINES Data points are connected with a line. This is the default for both 2D and 3D plots.

PLOT_PLOTTYPE_IMPULSES Display vertical lines from the x-axis (for 2D plots) or the xy plane (for 3D plots) to the data points.

PLOT_PLOTTYPE_STEPS Adjacent points are connected with two line segments, one from (x1,y1) to (x2,y1), and a second from (x2,y1) to (x2,y2). This type is available only for 2D plots.

PLOT_PLOTTYPE_FSTEPS Adjacent points are connected with two line segments, one from (x1,y1) to (x1,y2), and a second from (x1,y2) to (x2,y2). This type is available only for 2D plots.

PLOT_PLOTTYPE_HISTEPS This type is intended for plotting histograms. The point x1 is represented by a horizontal line from ((x0+x1)/2,y1) to ((x1+x2)/2,y1). Adjacent lines are connected with a vertical line from ((x1+x2)/2,y1) to ((x1+x2)/2,y2). This type is available only for 2D plots.

PLOT_PLOTTYPE_POINTS Markers are displayed at each data point.

PLOT_PLOTTYPE_LINESPOINTS Markers are displayed at each data point and connected with a line.

PLOT_PLOTTYPE_DOTS A small dot is displayed at each data point. The optional *point_type* argument effects only the color of the dot.

num The data set to which the plot type applies.

line_type An integer index representing the line type for drawing.

line_width A scaling factor for the line width. The line width is *line_width* multiplied by the default width.

point_type An integer index representing the desired point type.

point_size A scaling factor for the size of the point used. The point size is *point_size* multiplied by the default size.

Description
Set the desired plot type for a previously added data set. For 3D plots, only **PLOT_PLOTTYPE_LINES**, **PLOT_PLOTTYPE_IMPULSES**, **PLOT_PLOTTYPE_POINTS**, and **PLOT_PLOTTYPE_LINESPOINTS** are valid. If other types are specified, **PLOT_PLOTTYPE_POINTS** is used. Numbering of the data sets starts with zero. New plot types replace previously specified types. The line style and/or marker type for the plot are selected automatically, unless the appropriate combination of *line_type*, *line_width*, *point_type*, and *point_size* are specified. The *line_type* is an optional argument specifying an index for the line type used for drawing the line. The line type varies depending on the terminal type used (see **CPlot**::**outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types. The *line_width* is an optional argument used to specify the line width. The line width is *line_width* multiplied by the default width. Typically the default width is one pixel. *point_type* is an optional argument used to change the appearance (color and/or marker type) of a point. It is specified with an integer representing the index of the desired point type. All terminals support at least six different point types. *point_size* is an optional argument used to change the size of the point. The point size is *point_size* multiplied by the default size. If *point_type* and *point_size* are set to zero or a negative number, a default value is used.

Portability
The *line_width* and *point_size* options is not supported by all terminal types.

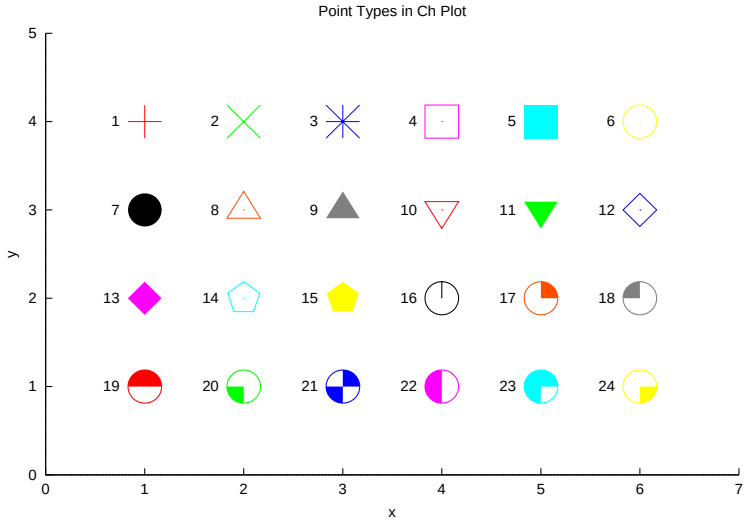
For 3D plots on some systems with output type set to `postscript` (see **CPlot**::**outputType**()), data may not be displayed for **PLOT_PLOTTYPE_DOTS**.

Example 1
This example shows some of the different point types for the default X-window and the `postscript` terminal types (see **CPlot**::**outputType**). In this example the points have a point size of five times the default. The appearance of points for different terminal types may be different.

```
#include <chplot.h>

int main() {
    double x, y;
    string_t text;
    int datasetnum=0, point_type = 1, point_size = 5;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 7);
    plot.axisRange(PLOT_AXIS_Y, 0, 5);
    plot.title("Point Types in Ch Plot");
    for (y = 4; y >= 1; y--) {
        for (x = 1; x <= 6; x++) {
            sprintf(text, "%d", point_type);
            plot.point(x, y, 0);
            plot.plotType(PLOT_PLOTTYPE_POINTS, datasetnum,  point_type, point_size);
            plot.text(text, PLOT_TEXT_RIGHT, x-.25, y, 0);
            datasetnum++; point_type++;
        }
    }
    plot.plotting();
}
```

Output
Output displayed in X-window. Identical shape markers have different colors on screen.

Output
Output as a postscript file.

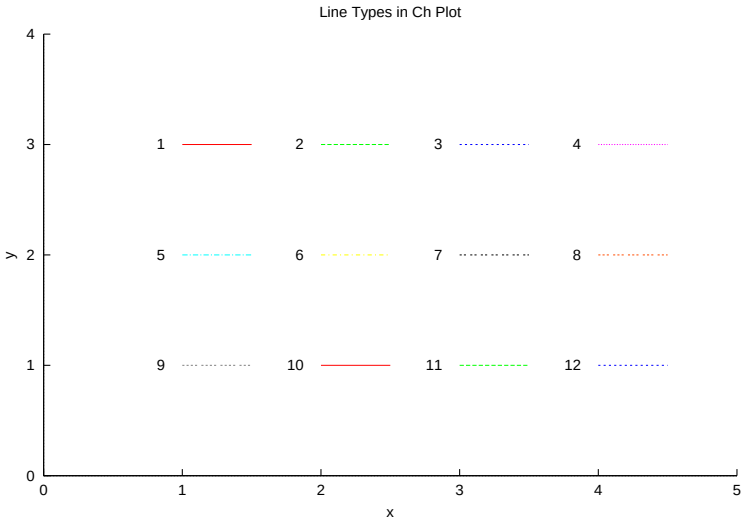


Example 2
This example shows some of the different line types for the `postscript` terminal type (see **CPlot**::**outputType**). The appearance of lines for different terminal types may be different.

```
#include <chplot.h>

int main() {
    double x, y, xx[2], yy[2];
    string_t text;
    int line_type = 1, line_width = 1, datasetnum = 0;
    class CPlot plot;
```

```
    plot.axisRange(PLOT_AXIS_X, 0, 5);
    plot.axisRange(PLOT_AXIS_Y, 0, 4, 1);
    plot.title("Line Types in Ch Plot");
    for (y = 3; y >= 1; y--) {
        for (x = 1; x <= 4; x++) {
            sprintf(text, "%d", line_type);
            linspace(xx, x, x+.5);
            linspace(yy, y, y);
            plot.data2D(xx, yy);
            plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum, line_type,
                          line_width);
            plot.text(text, PLOT_TEXT_RIGHT, x-.125, y, 0);
            datasetnum++;
            line_type++;
         }
    }
    plot.plotting();
}
```

Output

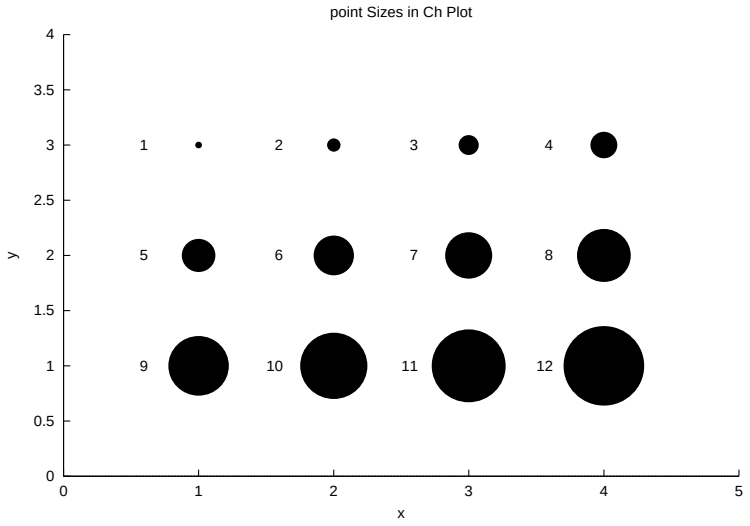


Example 3
This example shows some of the different point sizes for the `postscript` terminal type (see **CPlot**::**outputType**). The appearance of points for different terminal types may be different.

```
#include <chplot.h>

int main() {
    double x, y;
    string_t text;
    int datasetnum=0, point_type = 7, point_size = 1;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 5);
    plot.axisRange(PLOT_AXIS_Y, 0, 4);
    plot.title("point Sizes in Ch Plot");
    for (y = 3; y >= 1; y--) {
        for (x = 1; x <= 4; x++) {
```

```
            sprintf(text, "%d", point_size);
            plot.point(x, y, 0);
            plot.plotType(PLOT_PLOTTYPE_POINTS, datasetnum,  point_type, point_size);
            plot.text(text, PLOT_TEXT_RIGHT, x-.375, y, 0);
            datasetnum++; point_size++;
        }
    }
    plot.plotting();
}
```

Output

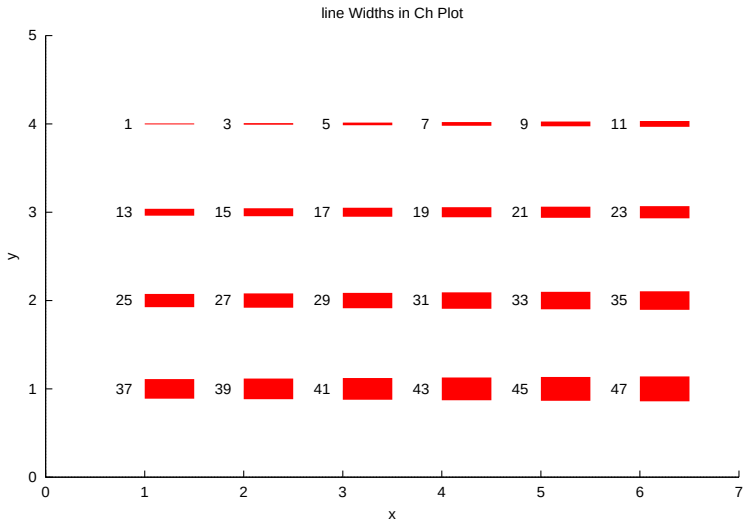


Example 4
This example shows some of the different line sizes for the `postscript` terminal type (see **CPlot**::**outputType**). The appearance of lines for different terminal types may be different.

```
#include <chplot.h>

int main() {
    double x, y, xx[2], yy[2];
    string_t text;
    int line_type = 1, line_width = 1, datasetnum = 0;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 7);
    plot.axisRange(PLOT_AXIS_Y, 0, 5);
    plot.title("line Widths in Ch Plot");
    for (y = 4; y >= 1; y--) {
        for (x = 1; x <= 6; x++) {
            sprintf(text, "%d", line_width);
            linspace(xx, x, x+.5);
            linspace(yy, y, y);
            plot.data2D(xx, yy);
            plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum, line_type,
                          line_width);
            plot.text(text, PLOT_TEXT_RIGHT, x-.125, y, 0);
            datasetnum++;
            line_width+=2;
         }
```

```
    }
    plot.plotting();
}
```

Output

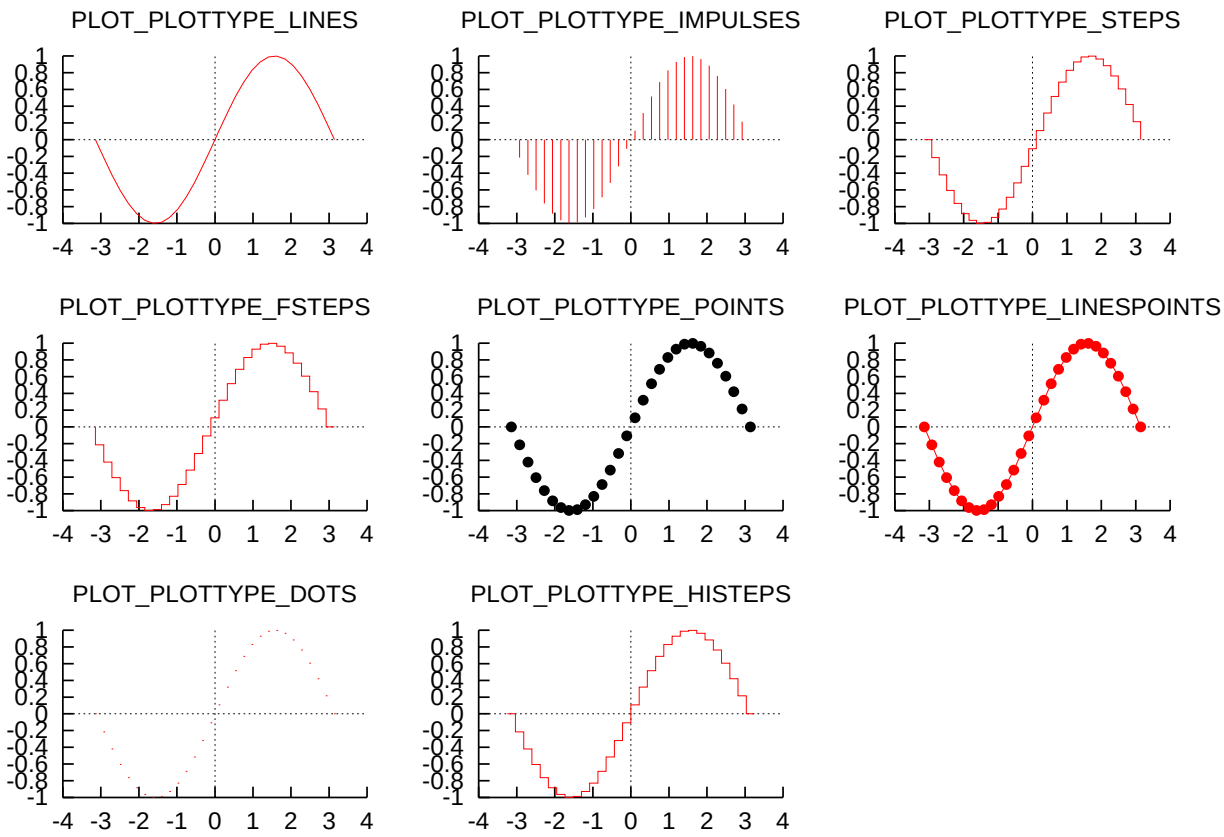


Example 5

```
#include <chplot.h>
#include <math.h>

int main() {
    class CPlot subplot, *spl;
    array double x1[30], y1[30];

    linspace(x1, -M_PI, M_PI);
    y1 = sin(x1);
    subplot.subplot(3,3);
    spl = subplot.getSubplot(0,0);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_LINES");
    spl = subplot.getSubplot(0,1);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_IMPULSES");
    spl = subplot.getSubplot(0,2);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_STEPS, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_STEPS");
    spl = subplot.getSubplot(1,0);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_FSTEPS, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_FSTEPS");
    spl = subplot.getSubplot(1,1);
    spl->data2D(x1, y1);
```

```
    spl->plotType(PLOT_PLOTTYPE_POINTS, 0, 7, 1);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_POINTS");
    spl = subplot.getSubplot(1,2);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_LINESPOINTS, 0, 0, 1, 7, 1);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_LINESPOINTS");
    spl = subplot.getSubplot(2,0);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_DOTS, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_DOTS");
    spl = subplot.getSubplot(2,1);
    spl->data2D(x1, y1);
    spl->plotType(PLOT_PLOTTYPE_HISTEPS, 0);
    spl->label(PLOT_AXIS_XY, NULL);
    spl->title("PLOT_PLOTTYPE_HISTEPS");
    subplot.plotting();
}
```

Output

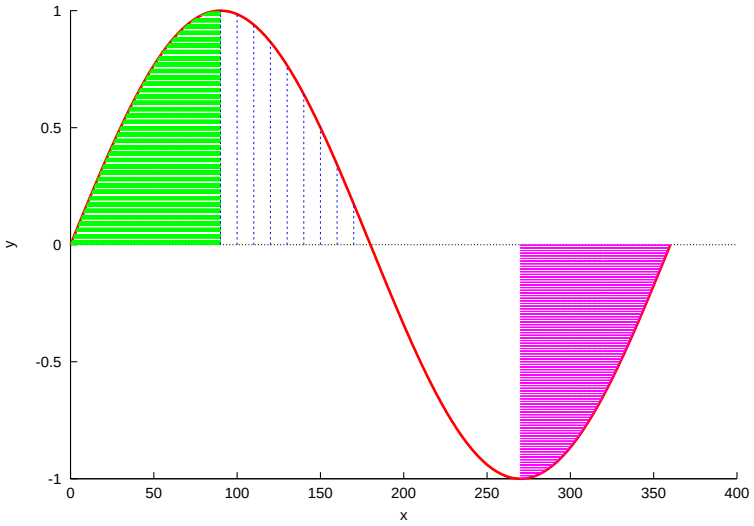


Example 6

```
#include <math.h>
#include <chplot.h>
```

```
int main() {
    int numpoints = 360,       // number of data points
        numpts=10;             // number of data points
    array double x0[numpoints], y0[numpoints];
    array double x1[numpoints], y1[numpoints];
    array double x2[numpts], y2[numpts];
    array double x3[numpoints], y3[numpoints];
    int line_type=0, line_width =5;
    class CPlot plot;

    linspace(x0, 0, 360);   // assign x0 with -10 <= x0 <= 360 linearly
    y0 = sin(x0*M_PI/180);  // array y0 is sine of array x0, element-wise
    linspace(x1, 0, 90);    // assign x1 with 0 <= x1 <= 90 linearly
    y1 = sin(x1*M_PI/180);  // array y1 is sine of array x1, element-wise
    linspace(x2, 90, 180);  // assign x2 with 90 <= x2 <= 180 linearly
    y2 = sin(x2*M_PI/180);  // array y2 is sine of array x2, element-wise
    linspace(x3, 270, 360); // assign x3 with 90 <= x3 <= 180 linearly
    y3 = sin(x3*M_PI/180);  // array y3 is sine of array x3, element-wise
    plot.data2D(x0, y0);
    plot.data2D(x1, y1);
    plot.data2D(x2, y2);
    plot.data2D(x3, y3);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0, line_type, line_width); // line for (x,y)
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 1);    // impulse plot for (x1, y1)
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 2);    // impulse plot for (x2, y2)
    plot.plotType(PLOT_PLOTTYPE_IMPULSES, 3);    // impulse plot for (x3, y3)
    plot.plotting();                             // get the plotting job done
}
```

Output



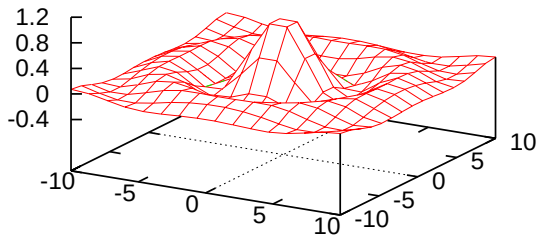
Example 7

```
#include <chplot.h>
#include <math.h>

int main() {
    double x[16], y[16], z[256];
    double r;
    int i, j;
```

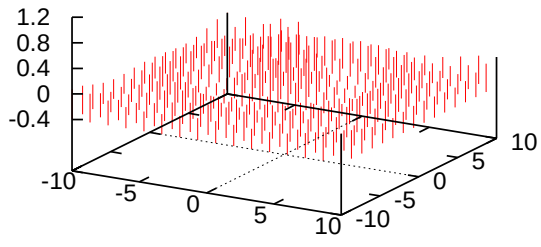
```
    class CPlot subplot, *spl;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<16; i++) {
        for(j=0; j<16; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[16*i+j] = sin(r)/r;
        }
    }
    subplot.subplot(2,2);
    spl = subplot.getSubplot(0,0);
    spl->data3D(x, y, z);
    spl->axisRange(PLOT_AXIS_Z, -.4, 1.2, .4);
    spl->axisRange(PLOT_AXIS_XY, -10, 10, 5);
    spl->plotType(PLOT_PLOTTYPE_LINES, 0);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_PLOTTYPE_LINES");
    spl = subplot.getSubplot(0,1);
    spl->data3D(x, y, z);
    spl->axisRange(PLOT_AXIS_Z, -.4, 1.2, .4);
    spl->axisRange(PLOT_AXIS_XY, -10, 10, 5);
    spl->plotType(PLOT_PLOTTYPE_IMPULSES, 0);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_PLOTTYPE_IMPULSES");
    spl = subplot.getSubplot(1,0);
    spl->data3D(x, y, z);
    spl->axisRange(PLOT_AXIS_Z, -.4, 1.2, .4);
    spl->axisRange(PLOT_AXIS_XY, -10, 10, 5);
    spl->plotType(PLOT_PLOTTYPE_POINTS, 0, 7, 1);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_PLOTTYPE_POINTS");
    spl = subplot.getSubplot(1,1);
    spl->data3D(x, y, z);
    spl->axisRange(PLOT_AXIS_Z, -.4, 1.2, .4);
    spl->axisRange(PLOT_AXIS_XY, -10, 10, 5);
    spl->plotType(PLOT_PLOTTYPE_LINESPOINTS, 0, 0, 1, 7, 1);
    spl->label(PLOT_AXIS_XYZ, NULL);
    spl->title("PLOT_PLOTTYPE_LINESPOINTS");
    subplot.plotting();
}
```

Output

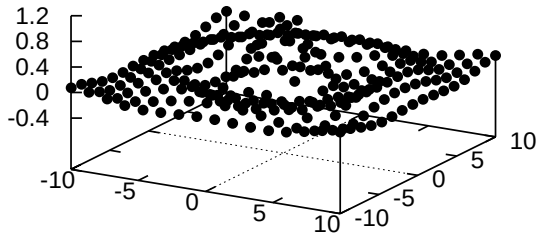
PLOT_PLOTTYPE_LINES



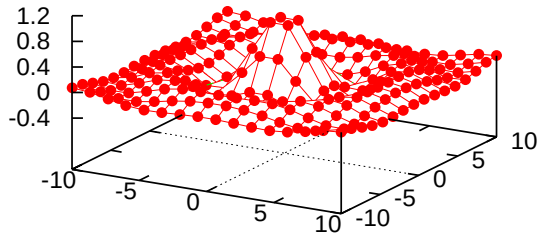
PLOT_PLOTTYPE_IMPULSES



PLOT_PLOTTYPE_POINTS



PLOT_PLOTTYPE_LINESPOINTS



CPlot::plotting

Synopsis
#include <chplot.h>
void plotting();

Purpose
Generate a plot file or display a plot.

Return Value
None.

Parameters
None.

Description
The plot is displayed or a file is generated containing the plot when this function is called. It shall be called after all the desired plot options are set.

Example
See **CPlot::data2D**().

CPlot::point

Synopsis
#include <chplot.h>
int point(**double** *x*, **double** *y*, **double** *z*);
Purpose
Add a point to a 2D plot.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x The x coordinate of the point.

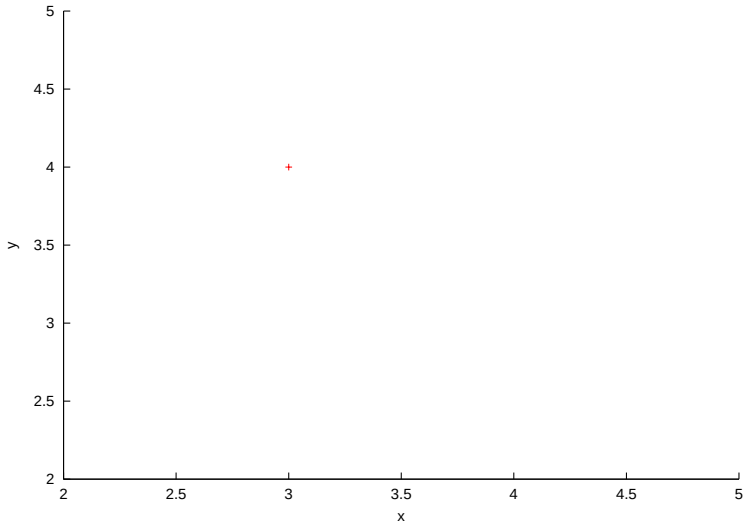
y The y coordinate of the point.

z The z coordinate of the point.

Description
This function adds a point to a plot. It is a convenience function for creation of geometric primitives. A point added with this function counts as a data set for later calls to **CPlot**::**legend**() and **CPlot**::**plotType**(). For 2D rectangular and 3D cartesian plots, *x*, *y*, and *z* are the coordinates of the point specified in units of the x, y and z axes. However, for 2D plots, *z* is ignored. For 2D polar and 3D cylindrical plots, the point is specified in polar coordinates where *x* is θ, *y* is r, and *z* is unchanged. Again, for 2D plots, *z* is ignored. For 3D plots with spherical coordinates *x* is θ, *y* is ϕ and *z* is r. For 3D plots with points, hidden line removal should be disabled (see **CPlot**::**removeHiddenLine**()) after all data are added.
Example 1

```
#include <chplot.h>

int main() {
    double x = 3, y = 4;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_XY, 2, 5); /* one point cannot do autorange */
    plot.point(x, y, 0);
    plot.plotting();
}
```

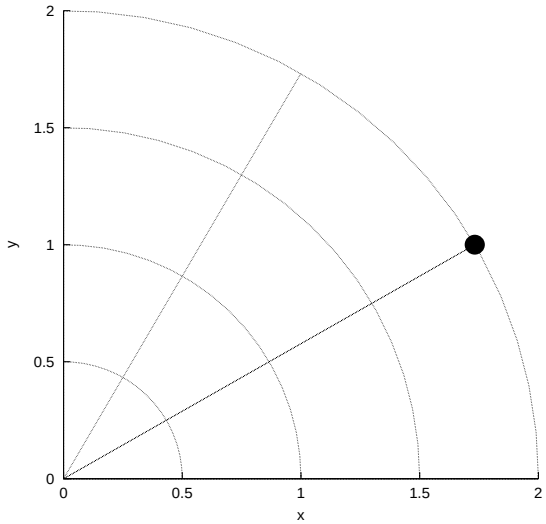
Output



Example 2

```
#include <chplot.h>

int main() {
    double theta1 = 30, r1 = 2;
    class CPlot plot;

    plot.grid(PLOT_ON, PLOT_GRID_POLAR);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.point(theta1, r1, 0);
    plot.plotType(PLOT_PLOTTYPE_POINTS, 0, 7, 3);
    plot.axisRange(PLOT_AXIS_XY, 0, 2);
    plot.sizeRatio(-1);
    plot.plotting();
}
```

Output



See Also

CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(),

CPlot::**data3DSurface**(), **CPlot**::**circle**(), **CPlot**::**line**(), **CPlot**::**outputType**(), **CPlot**::**plotType**(), **CPlot**::**polygon**(), **CPlot**::**rectangle**().

CPlot::polarPlot

Synopsis
#include <**chplot.h**>
void polarPlot(**int** *angle_unit*);

Purpose
Set a 2D plot to use polar coordinates.

Return Value
None.

Parameter
angle_unit Specify the unit for mesurement of an angular position. The following options are available:

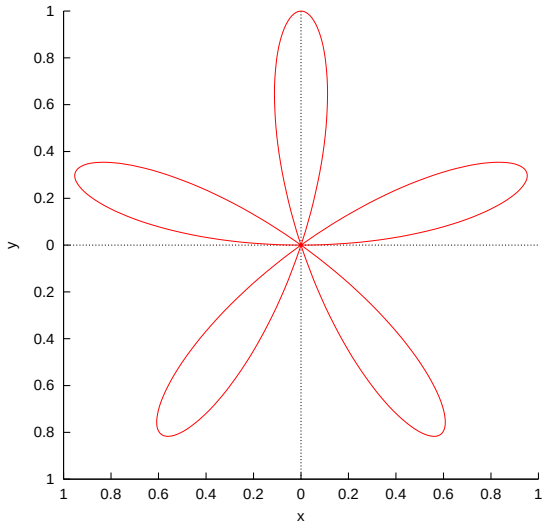
PLOT_ANGLE_DEG Angles measured in degree.

PLOT_ANGLE_RAD Angles measured in radian.

Description
Set a 2D plot to polar coordinates. In polar mode, two numbers, θ and r, are required for each point. First two arguments in member functions **CPlot**::dataTwoD() and **CPlot**::dataTwoDCurve() are the phase angle θ and magnitude r of points to be plotted.
Example 1
Compare with the example output in **CPlot**::**border**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    linspace(theta, 0, M_PI);
    r = sin(5*theta);                // Y-axis data.
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.sizeRatio(1);
    plot.plotting();
}
```

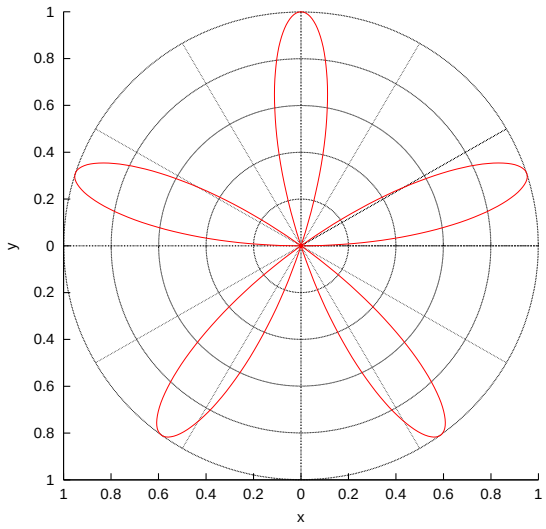
Output



Example 2

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    linspace(theta, 0, M_PI);
    r = sin(5*theta);                  // Y-axis data.
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.sizeRatio(1);
    plot.grid(PLOT_ON, PLOT_GRID_POLAR);
    plot.plotting();
}
```

Output



See Also
CPlot::**grid**().

CPlot::polygon

Synopsis
#include <**chplot.h**>
int polygon(**double** *x*[:], **double** *y*[:], **double** *z*[:], ... /* [**int** *num*] */);
Syntax
polygon(*x*, *y*, *z*)
polygon(*x*, *y*, *z*, *num*)

Purpose
Add a polygon to a plot.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x An array containing the x coordinates of the vertices of the polygon.

y An array containing the y coordinates of the vertices of the polygon.

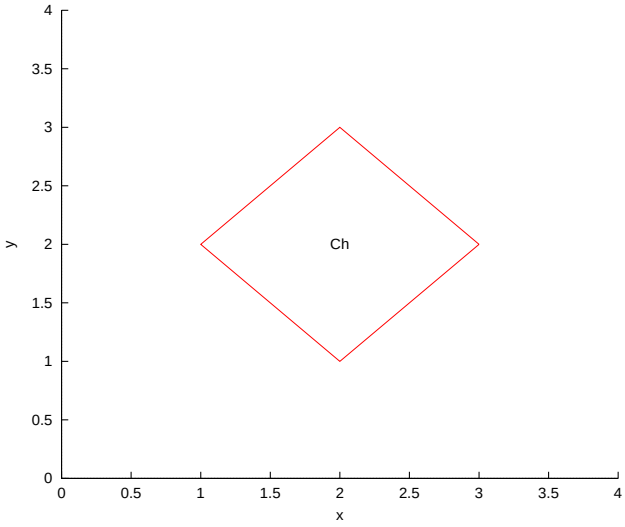
z An array containing the z coordinates of the vertices of the polygon.

num The number of elements of arrays *x*, *y*, and *z*.

Description
This function adds a polygon to a plot. It is a convenience function for creation of geometric primitives. A polygon added with this function is counted as a data set for later calls to **CPlot**::**legend**() and **CPlot**::**plotType**(). For 2D rectangular plots and 3D cartesian plots, *x*, *y*, and *z* contain the polygon vertices specified in units of the x, y, and z axes. However, for 2D plots, *z* is ignored. For 2D polar and 3D cylindrical plots, the locations of the vertices are specified in polar coordinates where *x* is θ, *y* is r, and *z* is unchanged. Again, for 2D plots, *z* is ignored. For 3D plots with spherical coordinates *x* is θ, *y* is ϕ and *z* is r. Each of the points is connected to the next in a closed chain. The line type and width vary depending on the terminal type used (see **CPlot**::**outputType**). Typically, changing the line type will change the color of the line or make it dashed or dotted. All terminals support at least six different line types.
Example 1

```
#include <chplot.h>

int main() {
    double x[5] = {3, 2, 1, 2, 3}, y[5] = {2, 3, 2, 1, 2};
    class CPlot plot;

    plot.polygon(x, y, NULL);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, 0, 4);
    plot.text("Ch", PLOT_TEXT_CENTER, 2, 2, 0);
    plot.plotting();
}
```

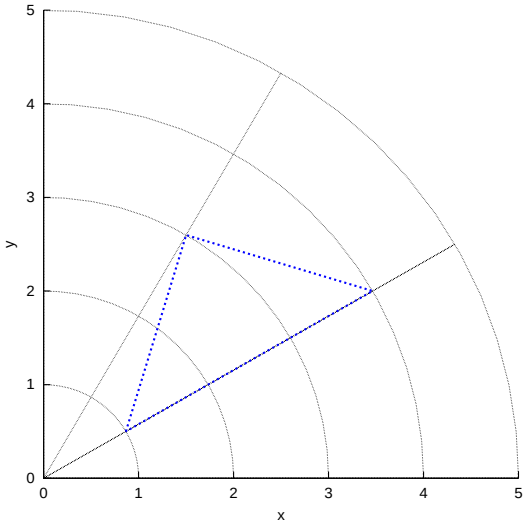
Output



Example 2

```
#include <chplot.h>
#include <math.h>

int main(){
    double theta[4] = {30, 60, 30, 30}, r[4] = {1, 3, 4, 1};
    class CPlot plot;

    plot.grid(PLOT_ON, PLOT_GRID_POLAR);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.polygon(theta, r, NULL);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0, 3, 4);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_XY, 0, 5);
    plot.plotting();
}
```

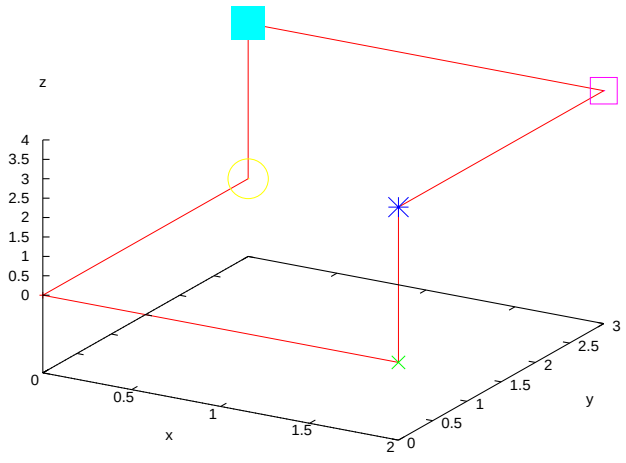
Output



Example 3

```
#include <chplot.h>

int main() {
    double x[7] = {0, 2, 2, 2, 0, 0, 0}, y[7] = {0, 0, 0, 3, 3, 3, 0},
           z[7] = {0, 0, 4, 4, 4, 0, 0};
    class CPlot plot;
    int datasetnum, pointtype, pointsize;

    plot.dimension(3);
    plot.polygon(x, y, z);
    plot.point(0, 0, 0);
    plot.point(2, 0, 0);
    plot.point(2, 0, 4);
    plot.point(2, 3, 4);
    plot.point(0, 3, 4);
    plot.point(0, 3, 0);
    for(datasetnum=1, pointtype=1, pointsize=1;
        datasetnum <= 6;
        datasetnum++, pointtype++, pointsize++)
       plot.plotType(PLOT_PLOTTYPE_POINTS,
                     datasetnum, pointtype, pointsize);
    plot.removeHiddenLine(PLOT_OFF);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**circle**(), **CPlot**::**line**(), **CPlot**::**outputType**(), **CPlot**::**plotType**(), **CPlot**::**point**(), **CPlot**::**rectangle**().

CPlot::rectangle

Synopsis

#include <chplot.h>
int rectangle(**double** *x*, **double** *y*, **double** *width*, **double** *height*);
Purpose
Add a rectangle to a 2D plot.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x The x coordinate of the lower-left corner of the rectangle.

y The y coordinate of the lower-left corner of the rectangle.

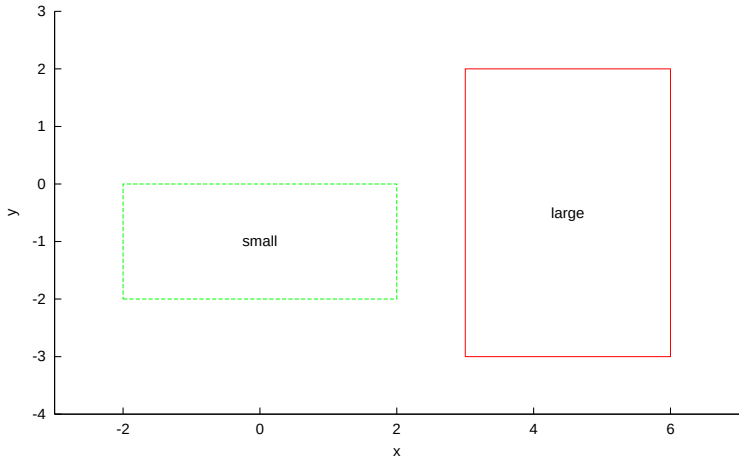
width The width of the rectangle.

height The height of the rectangle.

Description
This function adds a rectangle to a 2D plot. It is a convenience function for creation of geometric primitives. A rectangle added with this function is counted as a data set for later calls to **CPlot**::**legend**() and **CPlot**::**plotType**(). For rectangular plots, *x* and *y* are the coordinates of the lower-left corner of the rectangle. For polar plots, the coordinates of the lower-left corner are given in polar coordinates where *x* is theta and *y* is r. In both cases the *width* and *height* are the dimensions of the rectangle in rectangular coordinates.
Example 1

```
#include <chplot.h>

int main() {
    double x1 = 3, y1 = -3, width1 = 3, height1 = 5;
    double x2 = -2, y2 = -2, width2 = 4, height2 = 2;
    class CPlot plot;

    plot.rectangle(x1, y1, width1, height1);
    plot.rectangle(x2, y2, width2, height2);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_X, -3, 7);
    plot.axisRange(PLOT_AXIS_Y, -4, 3);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.text("large", PLOT_TEXT_CENTER, 4.5, -0.5, 0);
    plot.text("small", PLOT_TEXT_CENTER, 0, -1, 0);
    plot.plotting();
}
```

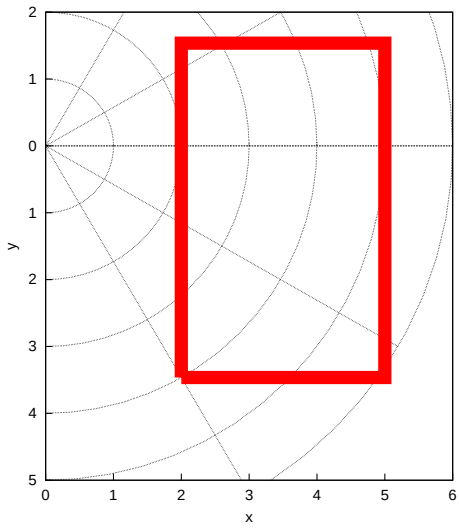
Output



Example 2

```
#include <chplot.h>

int main() {
    double theta1 = -60, r1 = 4, width1 = 3, height1 = 5;
    class CPlot plot;

    plot.grid(PLOT_ON, PLOT_GRID_POLAR);
    plot.polarPlot(PLOT_ANGLE_DEG);
    plot.rectangle(theta1, r1, width1, height1);
    plot.plotType(PLOT_PLOTTYPE_LINES, 0, 0, 25);
    plot.sizeRatio(-1);
    plot.axisRange(PLOT_AXIS_X, 0, 6);
    plot.axisRange(PLOT_AXIS_Y, -5, 2);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.plotting();
}
```

Output



See Also
CPlot::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**circle**(), **CPlot**::**line**(), **CPlot**::**outputType**(), **CPlot**::**plotType**(), **CPlot**::**point**(), **CPlot**::**polygon**().

CPlot::removeHiddenLine

Synopsis
#include <**chplot.h**>
void removeHiddenLine(**int** *flag*);

Purpose
Enable or disable hidden line removal for 3D surface plots.

Return Value
None.

Parameter
flag This parameter can be set to:

PLOT_ON Enable hidden line removal.

PLOT_OFF Disable hidden line removal.

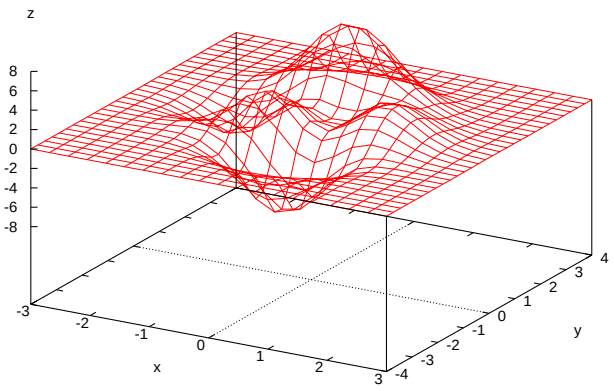
Description
Enable or disable hidden line removal for 3D surface plots. This option is only valid for 3D plots with a plot type set to **PLOT_PLOTTYPE_LINES** or **PLOT_PLOTTYPE_LINESPOINTS** with surface display enabled. By default hidden line removal is enabled. This function should be called after data set are added to the plot. The **PLOT_CONTOUR_SURFACE** option for **CPlot**::**contourMode**() does not work when hidden line removal is enabled. **CPlot**::**point**() cannot be used when hidden line removal is enabled. By default, the hidden lines are removed.

Example 1
Compare with the output for the example in **CPlot**::**data3D**().

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
```

```
    plot.removeHiddenLine(PLOT_OFF);
    plot.plotting();
}
```

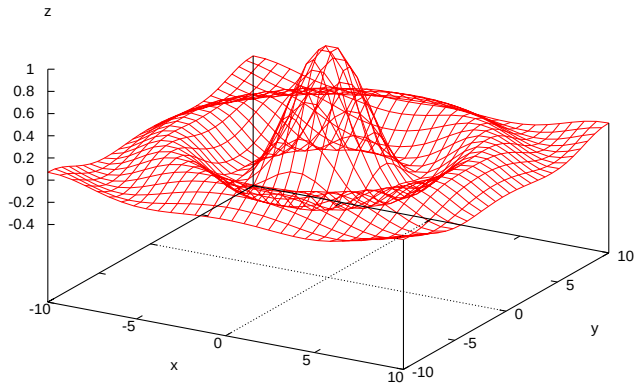
Output



Example 2

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.removeHiddenLine(PLOT_OFF);
    plot.plotting();
}
```

Output



See Also
CPlot::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**contourMode**(), **CPlot**::**plotType**(), **CPlot**::**point**(), **CPlot**::**showMesh**().

CPlot::scaleType

Synopsis
#include <**chplot.h**>
void scaleType(**int** *axis*, **int** *scale_type*, ... /* [**double** *base*] */);

Syntax
scaleType(*axis*, *scale_type*)
scaleType(*axis*, *scale_type*, *base*)

Purpose
Set the axis scale type for a plot.

Return Value
None.

Parameters
axis The axis to be modified. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

scale_type The scale type for the specified axis. Valid values are:

PLOT_SCALETYPE_LINEAR Use a linear scale for a specified axis.

PLOT_SCALETYPE_LOG Use a logarithmic scale for a specified axis.

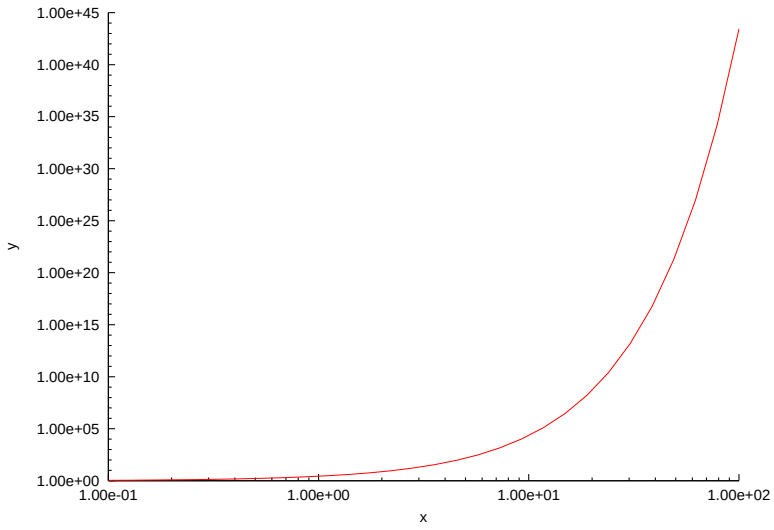
base The base for a log scale. For log scales the default base is 10.

Description
Using this function an axis can be modified to have a logarithmic scale. By default, axes are in linear scale.

Example

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 30;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    logspace(x, -1, 2);
    y = exp(x);                                 // Y-axis data.
    plot.scaleType(PLOT_AXIS_XY, PLOT_SCALETYPE_LOG);
    plot.ticsFormat(PLOT_AXIS_XY, "%.2e");
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



CPlot::showMesh

Synopsis
#include <chplot.h>
void showMesh(**int** *flag*);

Purpose
Display 3D data.

Return Value
None.

Parameters
flag This parameter can be set to:

PLOT_ON Enable the display of 3D data.

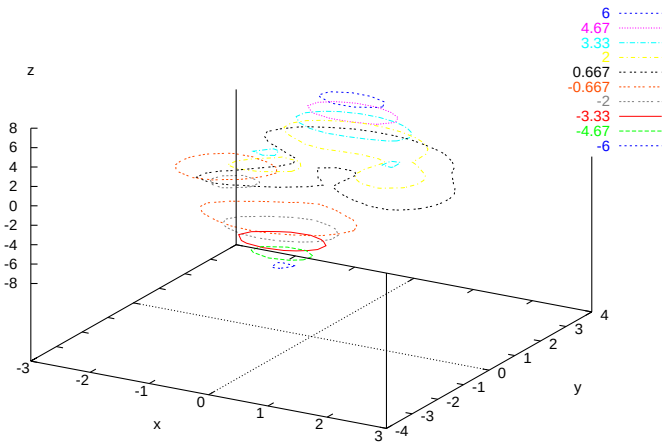
PLOT_OFF Disable the display of 3D data.

Description
Enable or disable the display of 3D data. If this option is disabled, data points or lines will not be drawn. This option is useful with **CPlot**::**contourMode**() to display contour lines without the display of a surface grid.

Example 1
Compare with the output for the example in **CPlot**::**contourLabel**().

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    double levels[10];
    int i,j;
    class CPlot plot;

    linspace(levels, -6, 6);
    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.contourLabel(PLOT_ON);
    plot.showMesh(PLOT_OFF);
    plot.contourMode(PLOT_CONTOUR_SURFACE);
    plot.contourLevels(levels);
    plot.plotting();
}
```

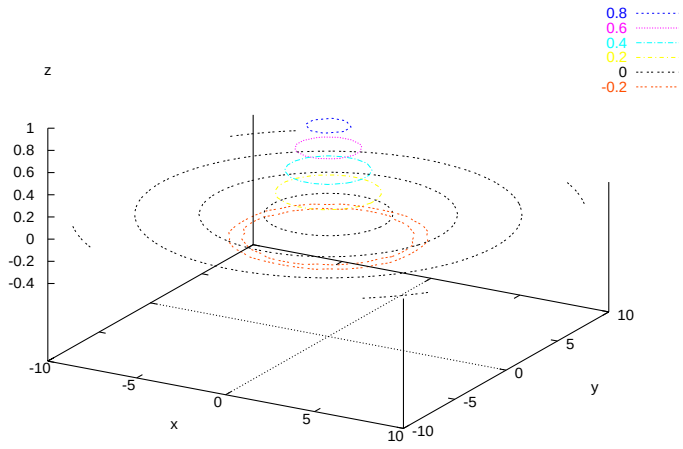
Output



Example 2

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[30], y[30], z[900];
    double r;
    int i, j;
    class CPlot plot;

    linspace(x, -10, 10);
    linspace(y, -10, 10);
    for(i=0; i<30; i++) {
        for(j=0; j<30; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.contourLabel(PLOT_ON);
    plot.showMesh(PLOT_OFF);
    plot.contourMode(PLOT_CONTOUR_SURFACE);
    plot.plotting();
}
```

Output



See Also
CPlot::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **CPlot**::**contourMode**(), **CPlot**::**plotType**().

CPlot::size

Synopsis
#include <**chplot.h**>
void size(**double** *x_scale*, **double** *y_scale*);

Purpose
Change the size of a 2D plot.

Return Value
None.

Parameters
x_scale A positive multiplicative scaling factor for the x direction.

y_scale A positive multiplicative scaling factor for the y direction.

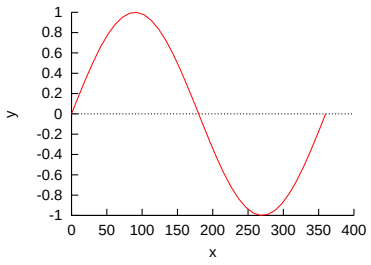
Description
This function can be used to scale a plot to the desired size. If the plot is displayed on a screen, the plot is scaled within the plot window. If the plot is saved to a file, or output to the stdout stream, the size of the plot image is scaled. For a plot with subplots, this function should be called before **CPlot**::**subplot**() is called.

Example

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
```

```
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.size(.5, .5);
    plot.plotting();
}
```

Output



See Also
CPlot::**size3D**() and **CPlot**::**sizeRatio**().

CPlot::size3D

Synopsis
#include <**chplot.h**>
void size3D(**float** *scale*, **float** *z_scale*);

Purpose
Scale a 3D plot.

Return Value
None.

Parameters
scale A positive multiplicative scaling factor that is applied to the entire plot.

z_scale A positive multiplicative scaling factor that is applied to the z-axis only.

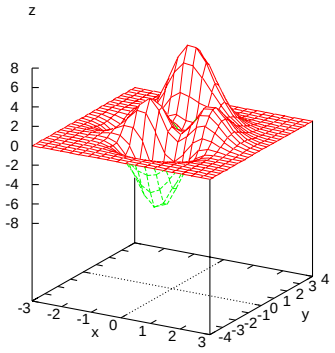
Description
This function can be used to scale a 3D plot to the desired size. By default, *scale* and z_*scale* are both 1.

Example
Compare with the output for examples in **CPlot**::**data3D**() and **CPlot**::**data3DSurface**().

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;
```

```
    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.size3D(0.5, 2);
    plot.plotting();
}
```

Output



See Also
CPlot::**size**().

CPlot::sizeRatio

Synopsis
#include <**chplot.h**>
void sizeRatio(**float** *ratio*);

Purpose
Change the aspect ratio for a plot.

Return Value
None.

Parameter
ratio The aspect ratio for the plot.

Description

The function sets the aspect ratio for the plot. The meaning of *ratio* changes depending on its value. For a positive *ratio*, it is the ratio of the length of the y-axis to the length of the x-axis. So, if *ratio* is 2, the y-axis will be twice as long as the x-axis. If *ratio* is zero, the default aspect ratio for the terminal type is used. If it is negative, *ratio* is the ratio of the y-axis units to the x-axis units. So, if *ratio* is -2, one unit on the y-axis will be twice as long as one unit on the x-axis.

Portability

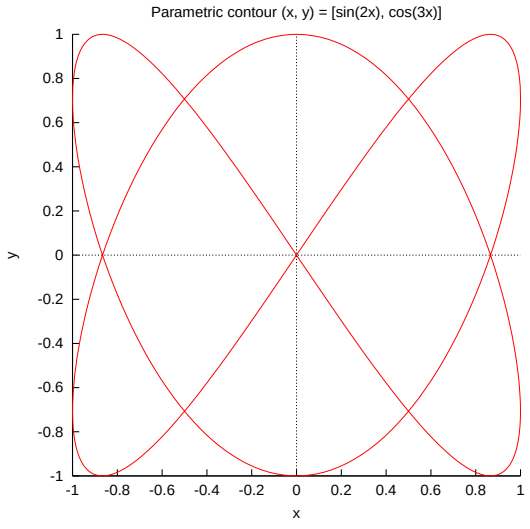
The aspect ratio specified is not exact on some terminals. To get a true ratio of 1, for example, some adjustment may be necessary.

Example

Compare with output for example in **plotxy**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double t[numpoints], x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(t, 0, 2*M_PI);
    x = sin(2*t);
    y = cos(3*t);
    plot.data2D(x, y);
    plot.title("Parametric contour (x, y) = [sin(2x), cos(3x)]");
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");

    /* both x and y axes the same length */
    plot.sizeRatio(1);
    plot.plotting();
}
```

Output



See Also
CPlot::**size**(), **CPlot**::**size3D**().

CPlot::subplot

Synopsis
#include <**chplot.h**>
int subplot(**int** *row*, **int** *col*);

Purpose
Create a group of subplots.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
row The number of rows in the subplot.

col The number of columns in the subplot.

Description
This function allocates memory for (*row***col*)-1 instances of the **CPlot** class to be used in a subplot. The location of the drawing origin and the scale for each element of the subplot is set automatically. The first element of the subplot (an instance of the **CPlot** class) is created normally by the user before this function is called. The remaining elements of the subplot are created and stored internally when this function is called. These elements are accessible through a pointer returned by the **CPlot**::**getSubplot**() function. Calling **CPlot**::**subplot**() with *row* = *col* = 1 has no effect.

Example
See **CPlot**::**getSubplot**().

See Also
CPlot::**boundingBoxOrigin**(),**CPlot**::**getSubplot**(), **CPlot**::**size**().

CPlot::text

Synopsis
#include <**chplot.h**>
void text(**string_t** string, **int** *just*, **double** *x*, **double** *y*, **double** *z*);

Purpose
Add a text string to a plot.

Return Value
None.

Parameters

string The string to be added at location (*x*,*y*) for 2D plots or (*x*,*y*,*z*) for 3D plots. The location of the text is in plot coordinates.

just The justification of the text. Valid values are:

PLOT_TEXT_LEFT The specified location is the left side of the text string.

PLOT_TEXT_RIGHT The specified location is the right side of the text string.

PLOT_TEXT_CENTER The specified location is the center of the text string.

x The x position of the text.

y The y position of the text.

z The z position of the text. This argument is ignored for a 2D plot.

Description
This function can be used to add text to a plot at an arbitrary location.

Example
See **CPlot**::**arrow**(), **CPlot**::**polygon**(), and **CPlot**::**rectangle**().

CPlot::tics

Synopsis
#include <chplot.h>
void tics(**int** *axis*, **int** *flag*)

Purpose
Enable or disable display of axis tics.

Return Value
None.

Parameters
axis The axis which labels are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

flag This parameter can be set to:

PLOT_ON Enable drawing of tics for the specified axis.

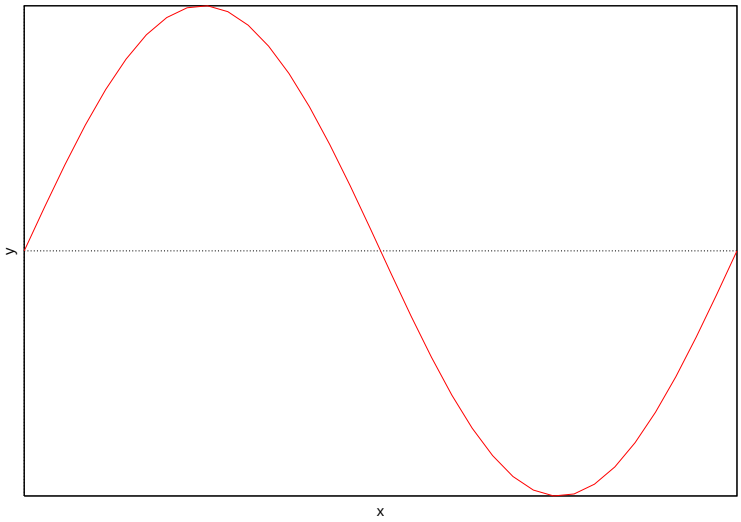
PLOT_OFF Disable drawing of tics for the specified axis.

Description
Enable or disable the display of tics and numerical labels for an axis. By default, tics are displayed for the x and y axes on 2D plots and for the x, y and z axes on 3D plots.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.tics(PLOT_AXIS_XY, PLOT_OFF);
    plot.plotting();
}
```

Output



See Also
CPlot::**ticsDirection**(), **CPlot**::**ticsFormat**(), **CPlot**::**ticsLabel**(), **CPlot**::**ticsLevel**(), **CPlot**::**ticsLocation**(), and **CPlot**::**ticsMirror**().

CPlot::ticsDay

Synopsis
#include <chplot.h>
void ticsDay(**int** *axis*);

Purpose
Set axis tic-marks to days of the week.

Return Value
None.

Parameter
axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

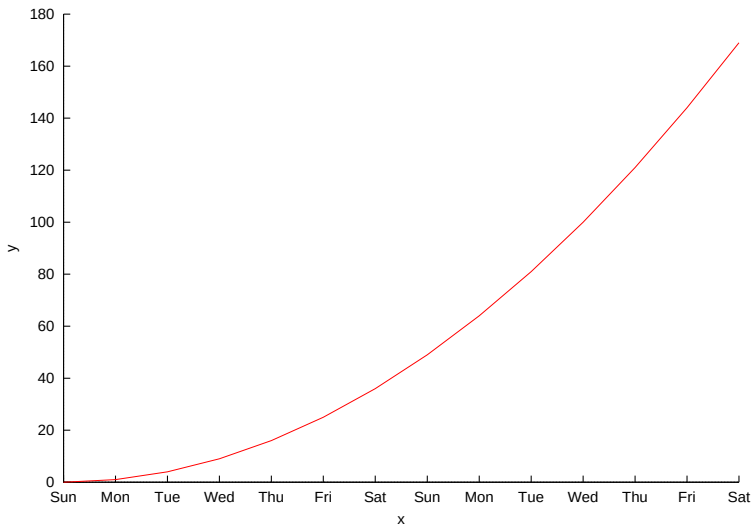
PLOT_AXIS_XYZ Select the x, y, and z axes.

Description
Sets axis tic marks to days of the week (0=Sunday, 6=Saturday). Values greater than 6 are converted into the value of modulo 7.

Example

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 14;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, numpoints-1);
    y = x.*x;
    plot.ticsDay(PLOT_ON);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also
CPlot::**ticsMonth**(), **CPlot**::**ticsLabel**().

CPlot::ticsDirection

Synopsis
#include <**chplot.h**>
void ticsDirection(**int** *direction*);

Purpose
Set the direction in which the tic-marks are drawn for an axis.

Return Value
None.

Parameter
direction Direction tic-marks are drawn. Can be set to:

PLOT_TICS_IN Draw axis tic-marks inward.

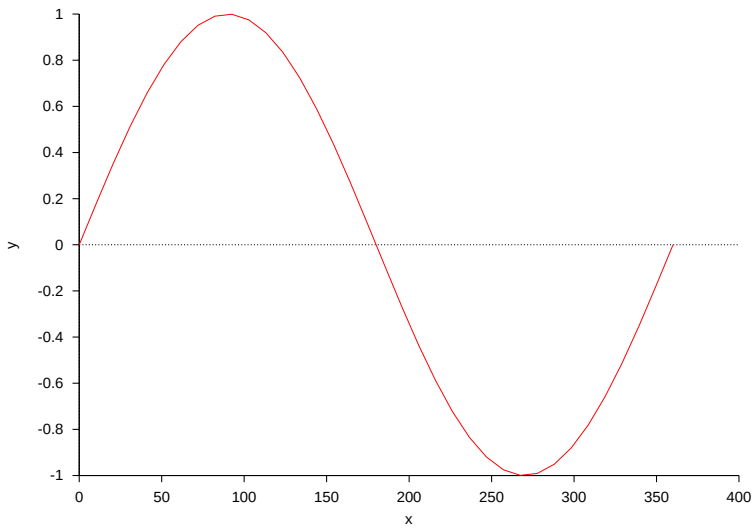
PLOT_TICS_OUT Draw axis tic-marks outward.

Description
Set the direction in which tic-marks are drawn in the inward or outward direction from the axes. The default is **PLOT_TICS_IN**.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.ticsDirection(PLOT_TICS_OUT);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also
CPlot::**ticsDay**(), **CPlot**::**ticsLabel**(), **CPlot**::**ticsLocation**(), and **CPlot**::**ticsMonth**().

CPlot::ticsFormat

Synopsis
#include <chplot.h>
void ticsFormat(**int** *axis*, **string_t** *format*);

Purpose
Set the number format for tic labels.

Return Value
None.

Parameters
axis The axis to be modified. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

format A C-style conversion specifier. Any conversion specifier suitable for double precision floats is acceptable, but other formats are available.

Description
This function allows for custom number formats for tic-mark labels. The default format is `"%g"`. The table below gives some of the available tics formats in C style.

Format	Effect
%f	Decimal notation for a floating point number. By default, 6 digits are displayed after the decimal point.
%e or %E	Scientific notation for a floating point value. There is alwayse one digit to the left of the decimal point. By default, 6 digits are displayed to the right of the decimal point.
%g or %G	A floating point number. If the value requires an exponent less than -4 or greater than the precision, e or E is used, otherwise f is used.

These format specifiers can be used with the standard C flag characters (-, +, #, etc.), minimum field width specifications and precision specifiers for function **fprintf**(). See **fprintf**() for details.

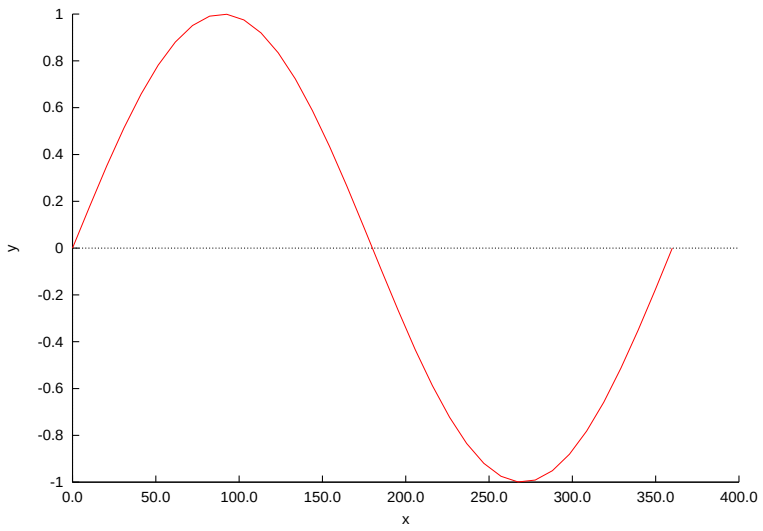
Examples:

format	number	output
”%f”	234.5678	234.567800
”%.2f”	234.5678	234.57
”%e”	123.456	0.123456e+03
”%E”	123.456	0.123456E+03
”%5.0f”	125.0	125
”%#5.0f”	125.0	125.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.ticsFormat(PLOT_AXIS_X, "%.1f");
    plot.plotting();
}
```

Output



See Also
CPlot::**ticsLabel**(), **fprintf**().

CPlot::ticsLabel

Synopsis
#include <**chplot.h**>
void ticsLabel(**int** *axis*, ... /* [[**string_t** *label*, **double** *location*], ...] */);

Syntax
ticsLabel(*axis*) /* does nothing */
ticsLabel(*axis*, *label*, *location*)
ticsLabel(*axis*, *label*, *location*, *label2*, *location2*)
etc.

Purpose
Add tic-marks with arbitrary labels to an axis.

Return Value
None.

Parameters
axis The axis which labels are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

label The tic-mark label.

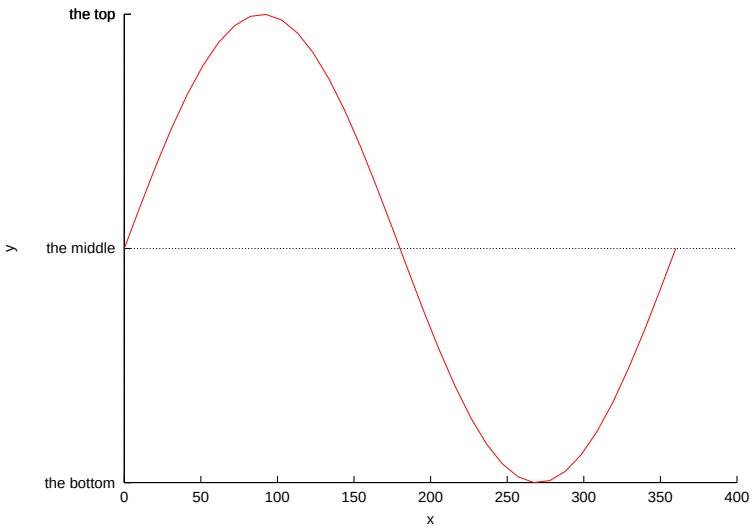
location The location of the tic-mark on the axis.

Description
Add tic marks with arbitrary labels to an axis. The axis specification is followed by one or more pairs of arguments. Each pair consists of a label string and a double precision floating point location. This function disables numerical labels for the specified axis.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.ticsLabel(PLOT_AXIS_Y, "the bottom", -1, "the middle", 0,
                    "the top", 1);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also
CPlot::**ticsDirection**(), **CPlot**::**ticsFormat**(), **CPlot**::**ticsLevel**(), **CPlot**::**ticsLocation**().

CPlot::ticsLevel

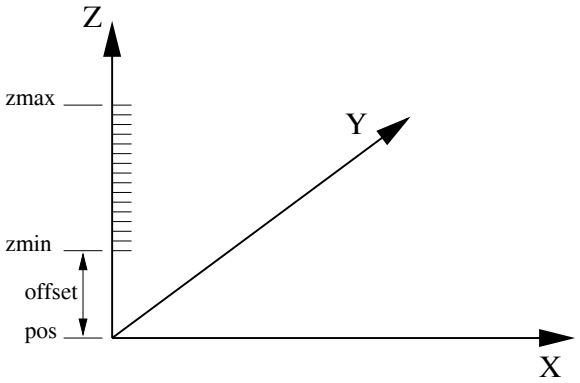
Synopsis
#include <chplot.h>
void ticsLevel(**double** *level*);

Purpose
Set the z-axis offset for drawing of tics in 3D plots.

Return Value
None.

Parameters
level The distance between the xy plane and the start of tic-marks on the z axis as a multiple of the full z range. This can be any non-negative number.

Description
This function specifies an offset between the xy plane and the start of z-axis tics-marks as a multiple of the full z range. By default the value for *level* is 0.5, so the z offset is a half of the z axis range. To place the xy-plane at the specified position `pos` on the z-axis, *level* shall equal (zmin-pos)/(zmax-zmin).



level = (zmin - pos)/(zmax - zmin)

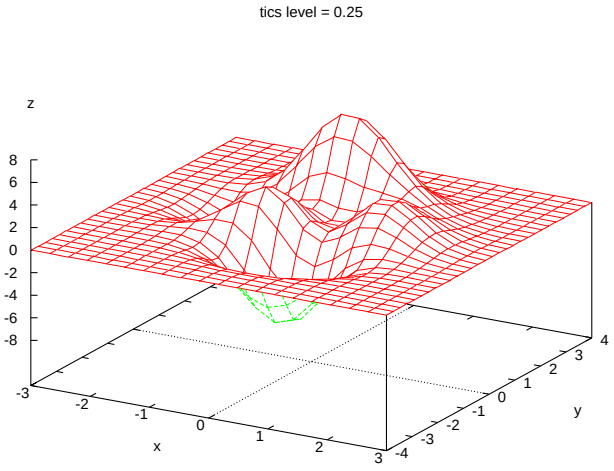
offset = level * (zmax - zmin)

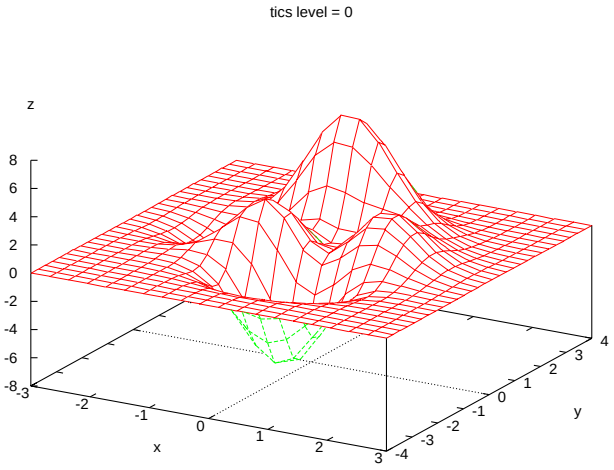
Example
Compare with the output for examples in **CPlot**::**data3D**() and **CPlot**::**data3DSurface**().

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;
    class CPlot plot;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plot.data3D(x, y, z);
    plot.ticsLevel(.25);
    plot.title("tics level = 0.25");
    plot.plotting();
```

```
    plot.ticsLevel(0);
    plot.title("tics level = 0");
    plot.plotting();
}
```

Output





CPlot::ticsLocation

Synopsis
#include <chplot.h>
void ticsLocation(int *axis*, **string_t** *location*)

Purpose
Specify the location of axis tic marks to be on the border or the axis.

Return Value
None.

Parameters
axis The *axis* parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_XY Select the x and y axes.

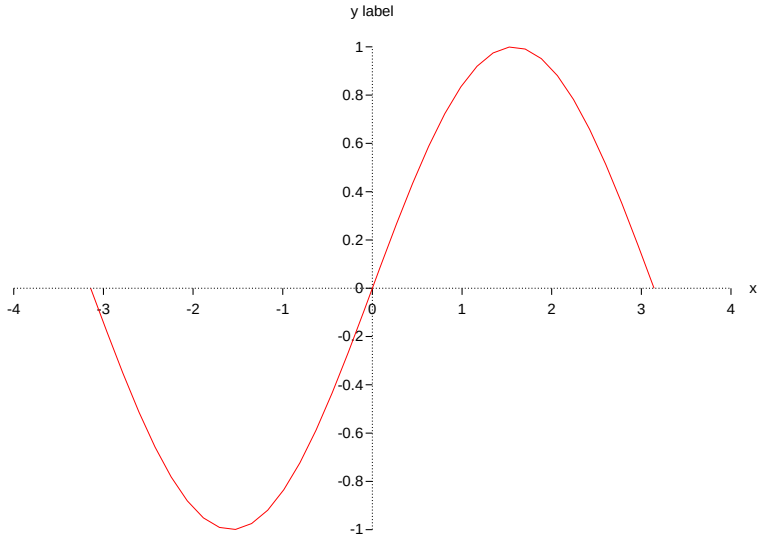
location Tic marks are placed on the plot border with `"border"` or on the axis itself with `"axis"`. By default, tic marks are on the border.

Description
Specify the location of axis tic marks to be on the plot border or the axis itself.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, -M_PI, M_PI);
    y = sin(x);
    plot.data2D(x, y);
    plot.ticsLocation(PLOT_AXIS_XY, "axis");
    plot.border(PLOT_BORDER_BOTTOM|PLOT_BORDER_LEFT, PLOT_OFF);
    plot.label(PLOT_AXIS_XY, NULL);
    plot.text("y label", PLOT_TEXT_CENTER, 0, 1.15, 0);
    plot.text("x", PLOT_TEXT_CENTER, 4.25, 0, 0);
    plot.margins(-1, -1, 2, -1); /* adjust top margin for y label */
    plot.plotting();
}
```

Output



See Also
CPlot::**tics**(), **CPlot**::**ticsDirection**(), **CPlot**::**ticsFormat**(), **CPlot**::**ticsLabel**, **CPlot**::**ticsLevel**(), **CPlot**::**ticsLocation**, and **CPlot**::**ticsMirror**().

CPlot::ticsMirror

Synopsis
#include <chplot.h>
void ticsMirror(**int** *axis*, **int** *flag*)

Purpose
Enable or disable the display of axis tics on the opposite axis.

Return Value
None.

Parameters
axis The axis which labels are added to. This parameter can take one of the following values:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

PLOT_AXIS_XYZ Select the x, y, and z axes.

flag This parameter can be set to:

PLOT_ON Enable drawing of tics for the specified axis.

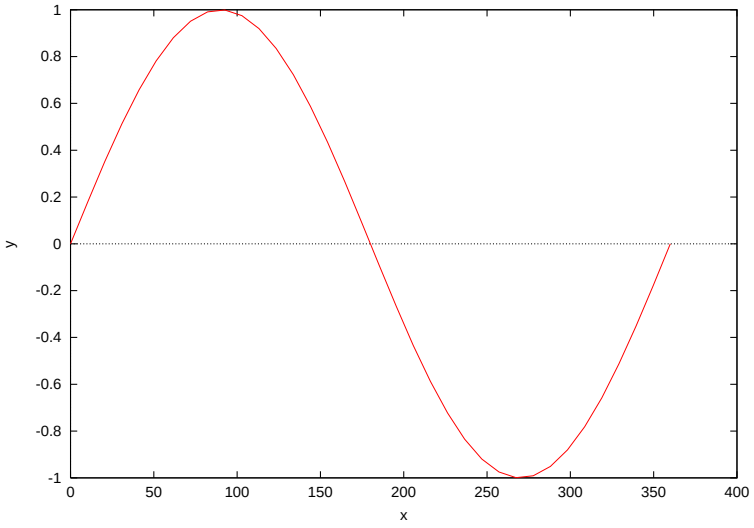
PLOT_OFF Disable drawing of tics for the specified axis.

Description
Enable or disable the display of tics on the opposite (mirror) axis. By default, on 2D plots tics on the opposite axis are not displayed. On 3D plots they are displayed

Example
Compare with output for example in **CPlot**::**border**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.ticsMirror(PLOT_AXIS_XY, PLOT_ON);
    plot.plotting();
}
```

Output



See Also
CPlot::**tics**(), **CPlot**::**ticsDirection**(), **CPlot**::**ticsFormat**(), **CPlot**::**ticsLabel**(), **CPlot**::**ticsLevel**(), and **CPlot**::**ticsLocation**().

CPlot::ticsMonth

Synopsis
#include <chplot.h>
void ticsMonth(**int** *axis*);

Purpose
Set axis tic-marks to months.

Return Value
None.

Parameter
axis The axis to be changed. Valid values are:

PLOT_AXIS_X Select the x axis only.

PLOT_AXIS_Y Select the y axis only.

PLOT_AXIS_Z Select the z axis only.

PLOT_AXIS_XY Select the x and y axes.

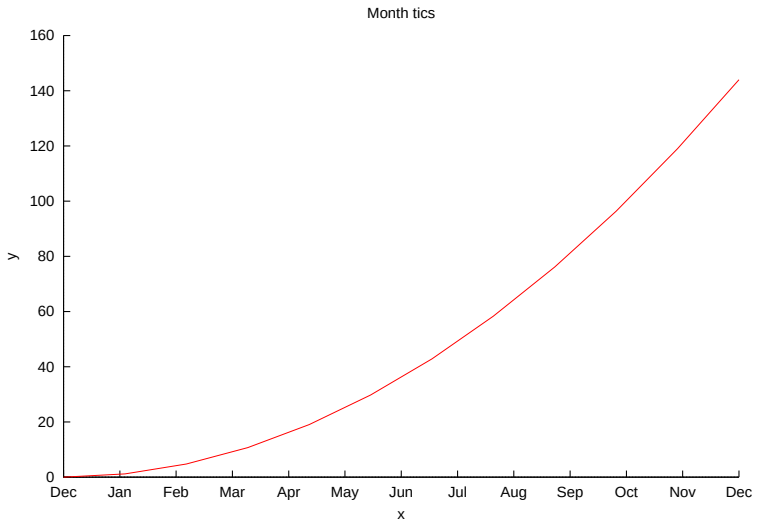
PLOT_AXIS_XYZ Select the x, y, and z axes.

Description
Sets axis tic marks to months of the year (1=January, 12=December). Values greater than 12 are converted into the value of modulo 12.

Example

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[12], y[12];
    string_t title="Month tics",                    // Define labels.
             xlabel="x",
             ylabel="y";
    class CPlot plot;

    linspace(x, 0, 12);
    y = x.*x;
    plot.ticsMonth(PLOT_AXIS_X);
    plot.title(title);
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also
CPlot::**ticsDay**(), **CPlot**::**ticsLabel**().

CPlot::title

Synopsis
#include <**chplot.h**>
void title(**string_t** *title*);

Purpose
Set the plot title.

Return Value
None.

Parameters
title The plot title.

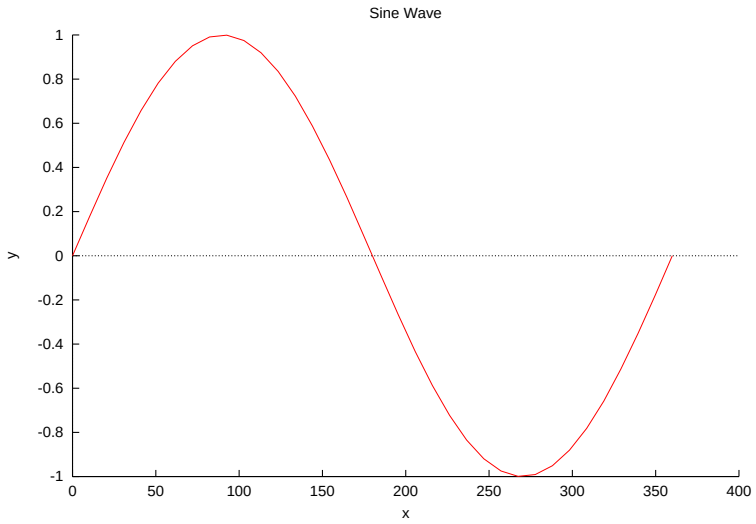
Description
Add a title string to an existing plot variable. For no title, `NULL` can be specified. By default, no title is specified.

Example
Compare with the output for examples in **CPlot**::**data2D**() and **CPlot**::**data2DCurve**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    string_t title="Sine Wave";                    // Define labels.
    class CPlot plot;

    linspace(x, 0, 360);
```

```
    y = sin(x*M_PI/180);
    plot.title(title);
    plot.data2D(x, y);
    plot.plotting();
}
```

Output



See Also
CPlot::**label**(), **CPlot**::**getLabel**(), **CPlot**::**getTitle**().

fplotxy

Synopsis
#include <chplot.h>
int fplotxy(**double** *(*func)(***double** *x)*, **double** *x0*, **double** *xf*, ...
/* [**int** *num*, [**string_t** *title*, **string_t** *xlabel*, **string_t** *ylabel*], [**class CPlot** **pl*]] */);

Syntax
fplotxy(*func*, *x0*, *xf*)
fplotxy(*func*, *x0*, *xf*, *num*)
fplotxy(*func*, *x0*, *xf*, *num*, *title*, *xlabel*, *ylabel*)
fplotxy(*func*, *x0*, *xf*, *num*, *&plot*)
fplotxy(*func*, *x0*, *xf*, *num*, *title*, *xlabel*, *ylabel*, *&plot*)

Purpose
Plot a 2D function of *x* in the range $x0 \leq x \leq xf$.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
func A pointer to a function that takes a `double` as an argument and returns a `double`.

x0 The lower bound of the range to be plotted.

xf The upper bound of the range to be plotted.

num The number of points to be plotted. The default is 100.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

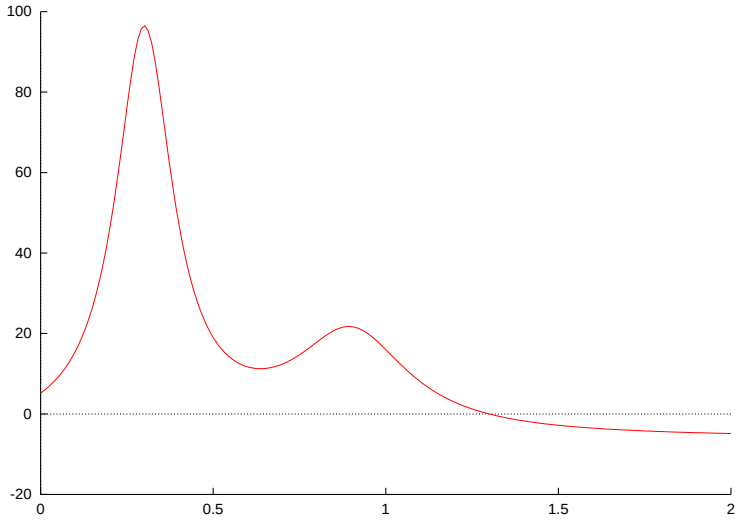
pl A pointer to an instance of the **CPlot** class.

Description
Plot a 2D function of *x* in the range $x0 \leq x \leq xf$. The function to be plotted, *func*, is specified as a pointer to a function that takes a *double* as an argument and returns a *double*. The arguments *x0* and *xf* are the end-points of the range to be plotted. The optional argument *num* specifies how many points in the range are to be plotted. The number of points plotted are evenly spaced in the range. By default, 100 points are plotted. The *title*, *xlabel*, and *ylabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot**::**plotting**() member function. If a previously initialized *plot* variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot**::**plotting**() member function.

Example

```
// Demonstrates the usage of the fplotxy() function.
// Print out a sine wave with appropriate labels and grid range.
#include <math.h>
#include <chplot.h>

int main() {
    double x0 = 0, xf = 2;
    int num = 200;

    /* local function to be plotted */
    double func2(double x) {
        return 1/((x-0.3)*(x-0.3)+0.01) + 1/((x-0.9)*(x-0.9)+0.04) - 6;
    }
    fplotxy(func2,x0,xf,num);
}
```

Output



See Also

CPlot, **fplotxyz**(), **plotxy**(), **plotxyf**(), **plotxyz**(), **plotxyzf**().

fplotxyz

Synopsis
#include <chplot.h>
int fplotxyz(**double** *(*func)(***double** *x,* **double** *y)*, **double** *x0*, **double** *xf*, **double** *y0*, **double** *yf*, ...
/* [**int** *x_num*, **int** *y_num*, **string_t** *title*, **string_t** *xlabel*, **string_t** *ylabel*, **string_t** *zlabel*],
[**class CPlot** **pl*]]*/);

Syntax
fplotxyz(*func*, *x0*, *xf*, *y0*, *yf*)
fplotxyz(*func*, *x0*, *xf*, *y0*, *yf*, *x_num*, *y_num*)
fplotxyz(*func*, *x0*, *xf*, *y0*, *yf*, *x_num*, *y_num*, *&plot*)
fplotxyz(*func*, *x0*, *xf*, *y0*, *yf*, *x_num*, *y_num*, *title*, *xlabel*, *ylabel*, *zlabel*)
fplotxyz(*func*, *x0*, *xf*, *y0*, *yf*, *x_num*, *y_num*, *title*, *xlabel*, *ylabel*, *zlabel*, *&plot*)

Purpose
Plot a 3D function of *x* and *y* in the range $x0 \leq x \leq xf$ and $y0 \leq y \leq yf$.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
func A pointer to a function that takes two **double** arguments and returns a **double**.

x0 The lower bound of the x range to be plotted.

xf The upper bound of the x range to be plotted.

y0 The lower bound of the y range to be plotted.

yf The upper bound of the y range to be plotted.

x_num The number of points to be plotted. The default is 25.

y_num The number of points to be plotted. The default is 25.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

zlabel The z-axis label.

pl A pointer to an instance of the **CPlot** class.

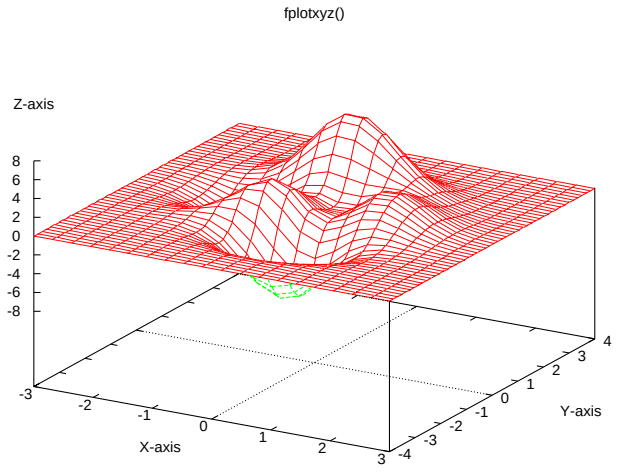
Description
Plot a 3D function of *x* and *y* in the range $x0 \leq x \leq xf$ and $y0 \leq y \leq yf$. The function to be plotted, *func*, is specified as a pointer to a function that takes two **double** arguments and returns a **double**. *x0* and *xf* are the end-points of the *x* range to be plotted. *y0* and *yf* are the end-points of the *y* range to be plotted. The optional arguments *x_num* and *y_num* specify how many points in the *x* and *y* ranges are to be plotted. The number of

points plotted are evenly spaced in the ranges. By default, *x_num* and *y_num* are 25. The *title*, *xlabel*, *ylabel*, and *zlabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot**::**plotting**() member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot**::**plotting**() member function. This function can only be used to plot 3D grid or scatter data, it cannot be used to plot 3D paths.

Example

```
// Demonstrates the usage of the fplotxyz() function.
// Print out a sine wave with appropriate labels and grid range.
#include <math.h>
#include <chplot.h>

int main() {
    string_t title="fplotxyz()",                    // Define labels.
             xlabel="X-axis",
             ylabel="Y-axis",
             zlabel="Z-axis";
    double x0 = -3, xf = 3, y0 = -4, yf = 4;
    int x_num = 20, y_num = 50;

    double func(double x, double y) {               // function to be plotted

        return 3*(1-x)*(1-x)*exp(-(x*x) - (y+1)*(y+1) )
               - 10*(x/5 - x*x*x - pow(y,5))*exp(-x*x-y*y)
               - 1/3*exp(-(x+1)*(x+1) - y*y);
    }
    fplotxyz(func, x0, xf, y0, yf, x_num, y_num, title, xlabel, ylabel, zlabel);
}
```

Output



See Also

CPlot, **fplotxy**(), **plotxy**(), **plotxyf**(), **plotxyz**(), **plotxyzf**().

plotxy

Synopsis
#include <**chplot.h**>
int plotxy(**double** *x*[&], **array double** &*y*, ... /* [**string_t** *title*, **string_t** *xlabel*, **string_t** *ylabel*],
[**class CPlot** **pl*] */);

Syntax
plotxy(*x*, *y*)
plotxy(*x*, *y*, *title*, *xlabel*, *ylabel*)
plotxy(*x*, *y*, *title*, *xlabel*, *ylabel*, &*plot*)

Purpose
Plot a 2D data set or initialize an instance of the **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x A one-dimensional array of size n. The value of each element is used for the x-axis of the plot.

y A m x n dimensional array containing m curves, each of which is plotted against x.

title The *title* of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description
The arrays *x* and *y* can be of any supported data type of real numbers. Conversion of the data to **double** type is performed internally. The *title*, *xlabel*, and *ylabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot**::**plotting**() member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot**::**plotting**() member function.

The following code segment

```
class CPlot plot;
plotxy(x, y, "title", "xlabel", "ylabel", &plot);
```

is equivalent to

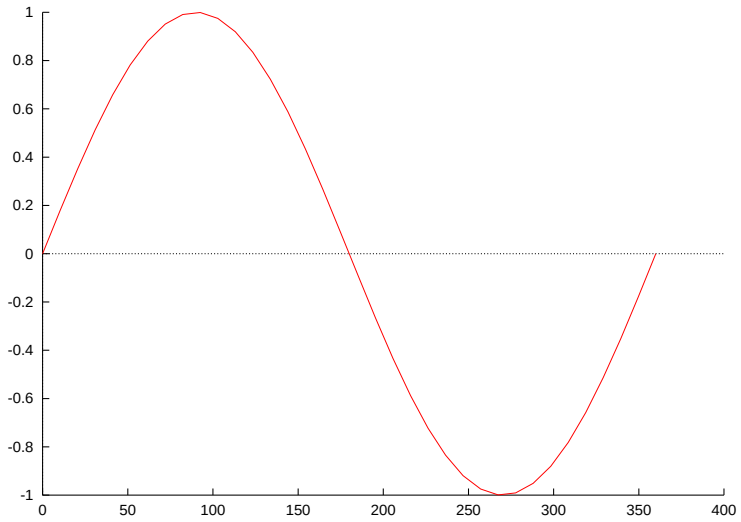
```
class CPlot plot;
plot.data2D(x, y);
```

```
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
```

Example 1

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[36];

    linspace(x, 0, 360);
    plotxy(x, sin(x*M_PI/180));
}
```

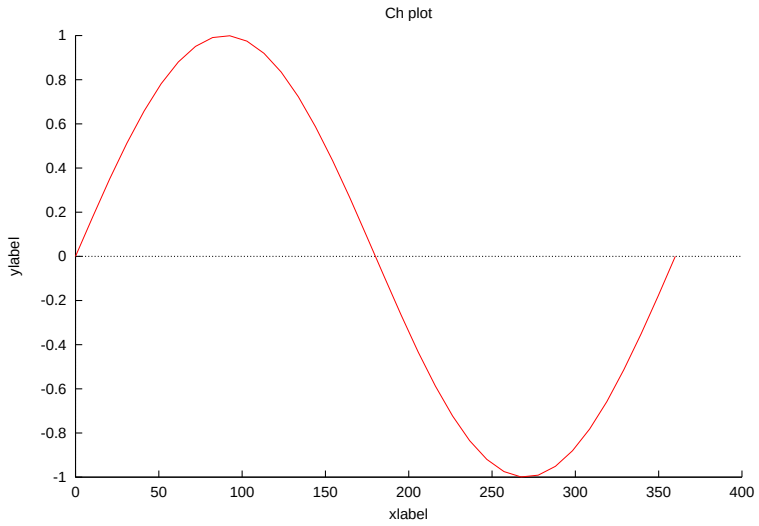
Output



Example 2

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    plotxy(x, y, "Ch plot", "xlabel", "ylabel");
}
```

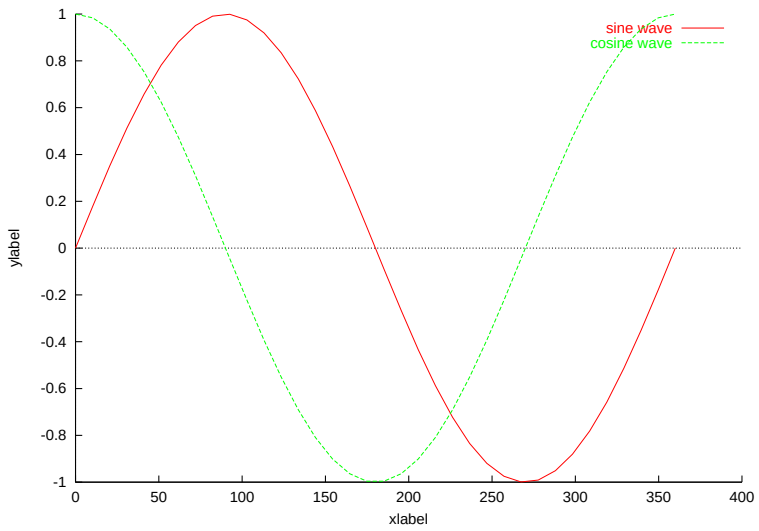
Output



Example 3

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[2][numpoints];
    int i;
    class CPlot plot;

    linspace(x, 0, 360);
    for(i=0; i<numpoints; i++) {
        y[0][i] = sin(x[i]*M_PI/180);
        y[1][i] = cos(x[i]*M_PI/180);
    }
    plotxy(x, y, NULL, "xlabel", "ylabel", &plot);
    plot.legend("sine wave", 0);
    plot.legend("cosine wave", 1);
    plot.plotting();
}
```

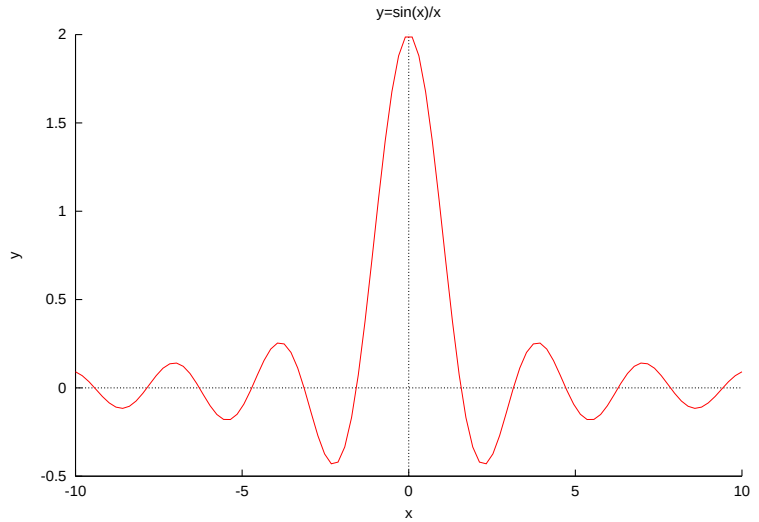
Output



Example 4

```
#include <math.h>
#include <float.h>
#include <chplot.h>

int main() {
    int numpoints = 100;
    array double x[numpoints], y[numpoints];

    linspace(x, -10, 10);
    x = x+(x==0)*FLT_EPSILON; /* if x==0, x becomes epsilon */
    y = sin(2*x)./x;
    plotxy(x, y, "y=sin(x)/x", "x", "y");
}
```

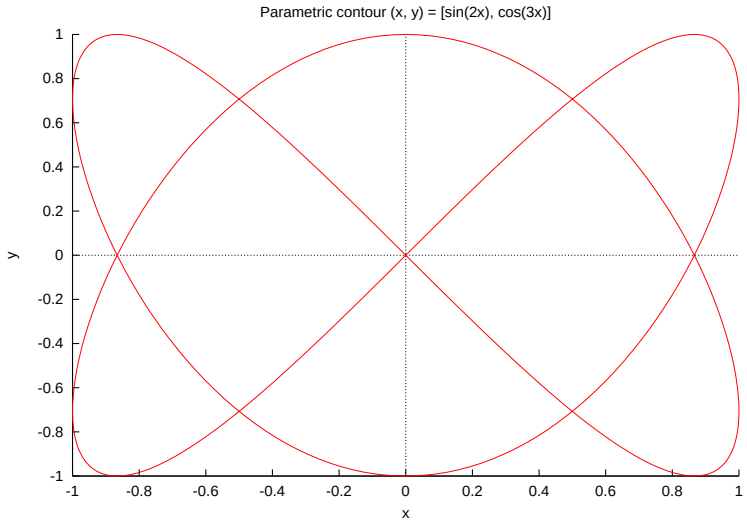
Output



Example 5
Compare with the output in the example in **CPlot**::**sizeRatio**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double t[numpoints], x[numpoints], y[numpoints];

    linspace(t, 0, 2*M_PI);
    x = sin(2*t);
    y = cos(3*t);
    plotxy(x, y, "Parametric contour (x, y) = [sin(2x), cos(3x)]", "x", "y");
}
```

Output



See Also

CPlot, **CPlot**::**data2D**(), **CPlot**::**data2DCurve**(), **fplotxy**(), **fplotxyz**(), **plotxyf**(), **plotxyz**(), **plotxyzf**().

plotxyf

Synopsis
#include <chplot.h>
int plotxyf(string_t *file*, ... /* [**string_t** *title*, **string_t** *xlabel*, **string_t** *ylabel*], [**class CPlot** **pl*] */);

Syntax
plotxyf(*file*)
plotxyf(*file*, *title*, *xlabel*, *ylabel*)
plotxyf(*file*, *title*, *xlabel*, *ylabel*, *&plot*)

Purpose
Plot 2D data from a file or initialize an instance of the **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
file The file containing the data to be plotted.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description
Plot 2D data from a file or initialize a **CPlot** variable. The data file should be formatted with each data point on a separate line. 2D data are specified by two values per point. The *title*, *xlabel*, and *ylabel*, for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot**::**plotting** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a `NULL` pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot**::**plotting**() member function. An empty line in the data file causes a break in the plot. Multiple curves can be plotted in this manner, however, the plot style will be the same for all curves.

The following code segment

```
class CPlot plot;
plotxyf("datafile", "title", "xlabel", "ylabel", &plot);
```

is equivalent to

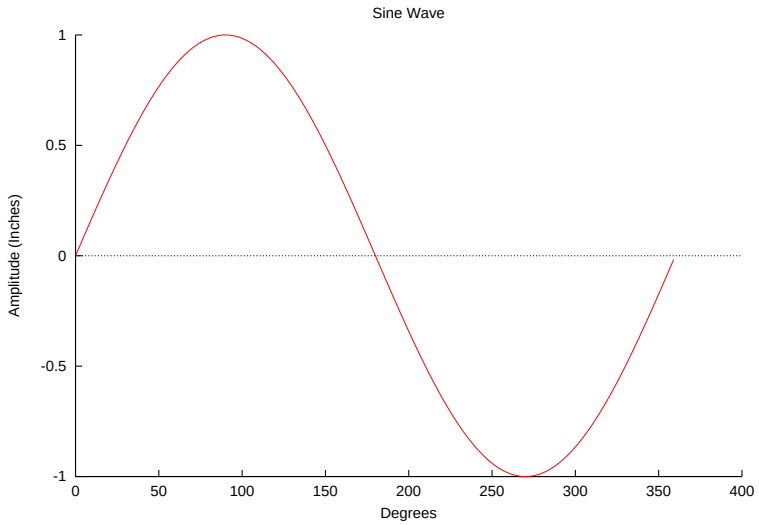
```
class CPlot plot;
plot.dataFile("datafile");
```

```
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
```

Example

```
// Demonstrates the usage of the plotxyf() function.
#include <stdio.h>
#include <chplot.h>
#include <math.h>

int main() {
    string_t title="Sine Wave",          //Define
           xlabel="Degrees",
           ylabel="Amplitude (Inches)";
    string_t file;
    file = tmpnam(NULL);                 //Create temporary file.
    int i;
    FILE *out;
    out=fopen (file,"w");              //Write data to file.
    for(i=0;i<=359;i++) fprintf(out,"%i %f \n",i,sin(i*M_PI/180));
    fclose(out);
    plotxyf(file,title,xlabel,ylabel); //Call plotting function.
    remove(file);
}
```

Output



See Also

CPlot, **CPlot**::**dataFile**, **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyz**(), **plotxyzf**().

plotxyz

Synopsis
#include <chplot.h>
int plotxyz(**double** *x*[&], **double** *y*[&], **array double** &*z*, ...
/* [**string_t** *title*, **string_t** *xlabel*, **string_t** *ylabel*, **string_t** *zlabel*], [**class CPlot** **pl*] */);

Syntax
plotxyz(*x*, *y*, z)
plotxyz(*x*, *y*, z, *title*, *xlabel*, *ylabel*, *zlabel*)
plotxyz(*x*, *y*, z, *title*, *xlabel*, *ylabel*, *zlabel*, &*plot*)

Purpose
Plot a 3D data set or initialize an instance of the **CPlot** class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x A one-dimensional array of size n_x. The value of each element is used for the x-axis of the plot.

y A one-dimensional array of size n_y. The value of each element is used for the y-axis of the plot.

z If the data are for a 3D curve, z is a m x n_z dimensional array, and $n_x = n_y = n_z$. If the data are for a 3D surface or grid, z is a m x n_z dimensional array, and $n_z = n_x \cdot n_y$.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

zlabel The y-axis label.

pl A pointer to an instance of the **CPlot** class.

Description
Plot a 3D data set or initialize a **CPlot** variable. For Cartesian data, *x* is a one-dimensional array of size n_x and *y* is a one-dimensional array of size n_y. *z* can be of two different dimensions depending on what type of data is to be plotted. If the data is for a 3D curve, *z* is a m x n_z dimensional array, and $n_x = n_y = n_z$. If the data is for a 3D surface or grid, z is a m x n_z dimensional array, and $n_z = n_x \cdot n_y$. For cylindrical or spherical data *x* is a one dimensional array of size n_x (representing θ), *y* is a one dimensional array of size n_y (representing z or ϕ), and z is a m x n_z dimensional array (representing r). In all cases these data arrays can be of any supported data type. Conversion of the data to **double** type is performed internally. The *title*, *xlabel*, *ylabel*, and *zlabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be initialized with the function parameters. The plot can then be displayed using the **CPlot**::**plotting** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without

calling the **CPlot**::**plotting**() member function.

The following code segment

```
class CPlot plot;
plotxyz(x, y, z, "title", "xlabel", "ylabel", "zlabel", &plot);
```

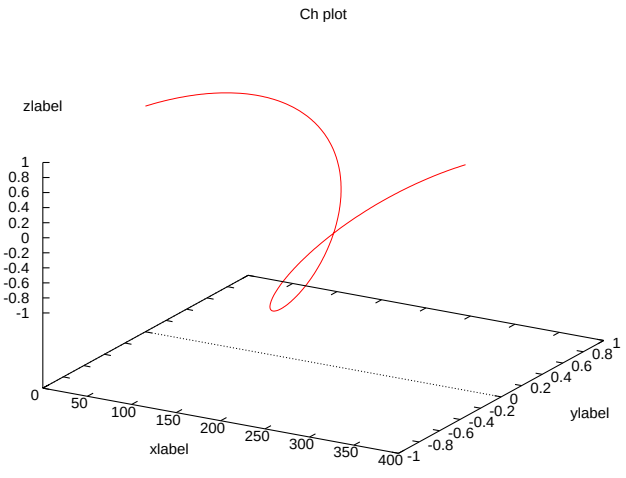
is equivalent to

```
class CPlot plot;
plot.data3D(x, y, z);
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
plot.label(PLOT_AXIS_Z, "zlabel");
```

Example 1

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[360], y[360], z[360];

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    z = cos(x*M_PI/180);
    plotxyz(x, y, z, "Ch plot", "xlabel", "ylabel", "zlabel");
}
```

Output

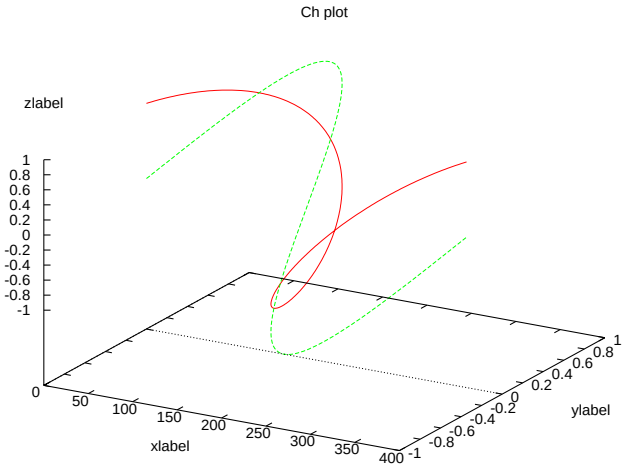


Example 2

```
#include <math.h>
#include <chplot.h>
```

```
int main() {
    array double x[360], y[360], z[2][360];
    int i;

    linspace(x, 0, 360);
    y = sin(x*M_PI/180);
    for(i=0; i<360; i++) {
        z[0][i] = cos(x[i]*M_PI/180);
        z[1][i] = y[i];
    }
    plotxyz(x, y, z, "Ch plot", "xlabel", "ylabel", "zlabel");
}
```

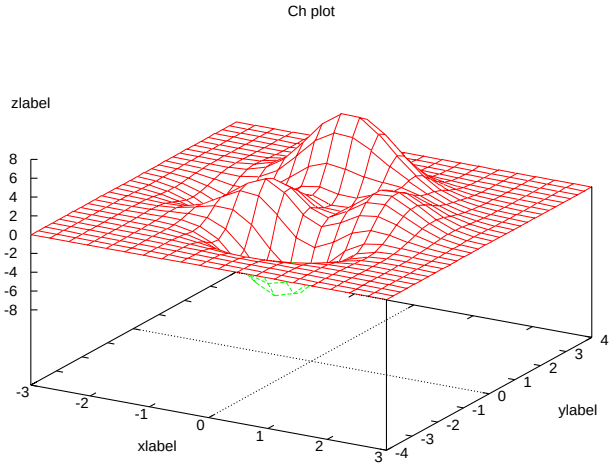
Output



Example 3

```
#include <math.h>
#include <chplot.h>

int main() {
    double x[20], y[30], z[600];
    int i,j;

    linspace(x, -3, 3);
    linspace(y, -4, 4);
    for(i=0; i<20; i++) {
        for(j=0; j<30; j++) {
            z[30*i+j] = 3*(1-x[i])*(1-x[i])*exp(-(x[i]*x[i])-(y[j]+1)*(y[j]+1))
            - 10*(x[i]/5 - x[i]*x[i]*x[i]-pow(y[j],5))*exp(-x[i]*x[i]-y[j]*y[j])
            - 1/3*exp(-(x[i]+1)*(x[i]+1)-y[j]*y[j]);
        }
    }
    plotxyz(x, y, z, "Ch plot", "xlabel", "ylabel", "zlabel");
}
```

Output



See Also

CPlot, **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyf**(), **plotxyzf**().

plotxyzf

Synopsis
#include <chplot.h>
int plotxyzf(string_t *file*, ... /* [**string_t** *title*, **string_t** *xlabel*, **string_t** *ylabel*, **string_t** *zlabel*],
[**class CPlot** **pl*] */);

Syntax
plotxyzf(*file*)
plotxyzf(*file*, *title*, *xlabel*, *ylabel*, *zlabel*)
plotxyzf(*file*, *title*, *xlabel*, *ylabel*, *zlabel*, *&plot*)

Purpose
Plot 3D data from a file or initialize an instance of the **CPlot**class.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
file The file containing the data to be plotted.

title The title of the plot.

xlabel The x-axis label.

ylabel The y-axis label.

zlabel The z-axis label.

pl A pointer to an instance of the **CPlot** class.

Description
Plot 3D data from a file or initialize a **CPlot** variable. The data file should be formatted with each data point on a separate line. 3D data is specified by three values per data point. For a 3D grid or surface data, each row is separated in the data file by a blank line. For example, a 3 x 3 grid would be represented as follows:

```
x1   y1   z1
x1   y2   z2
x1   y3   z3

x2   y1   z4
x2   y2   z5
x2   y3   z6

x3   y1   z7
x3   y2   z8
x3   y3   z9
```

This function can only be used to plot Cartesian grid data. The *title*, *xlabel*, *ylabel*, and *zlabel* for the plot can also optionally be specified. A pointer to a plot structure can also be passed to this function. If a non-NULL pointer is passed, it will be be initialized with the function parameters. The plot can then be displayed

using the **CPlot**::**plotting** member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed, an internal **CPlot** variable will be used and the plot will be displayed without calling the **CPlot**::**plotting**() member function. Two empty lines in the data file will cause a break in the plot. Multiple curves or surfaces can be plotted in this manner however, the plot style will be the same for all curves or surfaces.

The following code segment

```
class CPlot plot;
plotxyzf("datafile", "title", "xlabel", "ylabel", "zlabel", &plot);
```

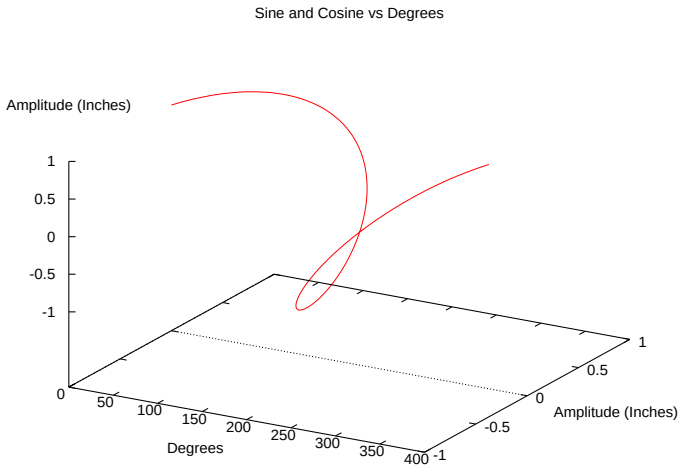
is equivalent to

```
class CPlot plot;
plot.dimension(3);
plot.dataFile("datafile");
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
plot.label(PLOT_AXIS_Z, "zlabel");
```

Example

```
#include <chplot.h>
#include <stdio.h>
#include <math.h>

int main() {
    string_t title="Sine and Cosine vs Degrees",
           xlabel="Degrees",
           ylabel="Amplitude (Inches)",
           zlabel="Amplitude (Inches)";
    string_t file;
    file = tmpnam(file);                                //temporary file
    int i;
    FILE *out;

    out=fopen (file,"w");                        // write data to file
    for(i=0;i<=359;i++) fprintf(out,"%i %f %f\n",i,sin(i*M_PI/180),cos(i*M_PI/180));
    fclose(out);
    plotxyzf(file,title,xlabel,ylabel,zlabel);  // call plotting function:
    remove(file);
}
```

Output



See Also
CPlot, **CPlot**::**dataFile**(), **fplotxy**(), **fplotxyz**(), **plotxy**(), **plotxyz**(), **plotxyz**().

Chapter 3

Shell Functions — <chshell.h>

Header file **chshell.h** contains the definitions of Ch shell functions. The following functions are designed for Ch shell programming.

Functions

Function	Description
chinfo()	Get information about the Ch language environment.
iskey()	Find out if a string is a keyword.
isstudent()	Find out if the Ch language environment is a student version.
isvar()	Find the property of a string.
sizeofelement()	Find the size of an element of an array.

Constants

The following macros are defined in **chshell.h**.

Constant	Description
CH_NOTVAR	Defined as 0, indicating the string is not a variable
CH_SYSTEMCONST	Indicating the string is a system constant.
CH_SYSTEMVAR	Indicating the string is a system variable.
CH_SYSTEMFUN	Indicating the string is a system function.
CH_EXTERNVAR	Indicating the string is a external variable.

Structure

Structure **chinfo_t** shall contain the following members

Struct chinfo_t {

```
  string_t edition;          // "Professional" or "Student"
  string_t releasedate;      // "October 1 1999"
  string_t version;          // "1.0"
  unsigned intvermajor;      // 1
  unsigned intverminor;      // 0
}
```

Datatype

The data type **chinfo_t** holds the parameters of the Ch language environment described above.

chinfo

Synopsis
#include <**chshell.h**>
int chinfo(**chinfo_t** **info*);

Purpose
Get information about the Ch language environment.

Return Value
This function returns 0 if it succeeds and -1 if it fails.

Parameters
info A pointer to a structure containing the edition, release date, version, major version number, minor version number, micro version number, and build number.

Description
This function gets the information about the Ch language environment including the edition, release date , version, version number.

Example

```
#include<chshell.h>

int main() {
  chinfo_t info;

  if(chinfo(&info)==-1)
     printf("Error: chinfo() error\n");
  printf("info.edition  = %s\n", info.edition);
  printf("info.releasedate = %s\n", info.releasedate);
  printf("info.version  = %s\n", info.version);
  printf("info.vermajor = %d\n", info.vermajor);
  printf("info.verminor = %d\n", info.verminor);
  printf("info.vermicro = %d\n", info.vermicro);
  printf("info.verbuild = %d\n", info.verbuild);
}
```

Output

```
info.edition  = Professional
info.releasedate = March 17, 2005
info.version  =
info.vermajor = 5
info.verminor = 0
info.vermicro = 1
info.verbuild = 12201
```

See Also
isstudent().

iskey

Synopsis
#include <chshell.h>
int iskey(**string_t** *name*);

Purpose
Find out if a string is a keyword.

Return Value
This function returns 1 if the string is a generic function, 2 if it is a keyword starting with aphabetical letter, 3 if it is a other keyword, 0 otherwise.

Parameters
name The input string.

Description
This function determines if the input string *name* is not a keyword.

Example

```
/* access element of a string */
#include <stdio.h>
#include <string.h>
#include <chshell.h>

int g;
int main() {
  int i;

  printf("iskey(\"sin\") = %d\n", iskey("sin"));
  printf("iskey(\"int\") = %d\n", iskey("int"));
  printf("iskey(\"=\") = %d\n", iskey("="));
  printf("iskey(\"unknown\") = %d\n", iskey("unknown"));

  printf("isstudent() = %d\n", isstudent());

  printf("isvar(\"i\") == CH_NOTVAR = %d\n", isvar("i") == CH_NOTVAR);
  printf("isvar(\"NaN\") == CH_SYSTEMCONST = %d\n", isvar("NaN") == CH_SYSTEMCONST);
  printf("isvar(\"_path\") == CH_SYSTEMVAR = %d\n", isvar("_path") == CH_SYSTEMVAR);
  printf("isvar(\"g\") == CH_EXTERNVAR = %d\n", isvar("g") == CH_EXTERNVAR);
  printf("isvar(\"isvar\") == CH_EXTERNFUN = %d\n", isvar("isvar") == CH_EXTERNFUN);
  printf("isvar(\"unknown\") = %d\n", isvar("unknown"));
}
```

Output

```
iskey("sin") = 1
iskey("int") = 2
iskey("=") = 3
iskey("unknown") = 0
isstudent() = 0
isvar("i") == CH_NOTVAR = 1
isvar("NaN") == CH_SYSTEMCONST = 1
```

```
isvar("_path") == CH_SYSTEMVAR = 1
isvar("g") == CH_EXTERNVAR = 1
isvar("isvar") == CH_EXTERNFUN = 1
isvar("unknown") = 0
```

See Also
isenv(), **isnum**(), **isstudent**(), **isvar**().

isstudent

Synopsis
#include <**chshell.h**>
int isstudent(**void**);

Purpose
Find out if the Ch language environment is a student version.

Return Value
This function returns 1 if it is a student version and 0 if it is a professional version.

Description
This function determines if the current Ch language environment is a student version.

Example
See **iskey**().

See Also
isenv(), **isnum**(), **iskey**(), **isvar**().

isvar

Synopsis
#include <**chshell.h**>
int isvar(**string_t** *name*);

Purpose
Find the property of a string.

Return Value
This function returns one of the following values:
CH_NOTVAR The string is not a variable.

CH_SYSTEMCONST The string is a system constant.

CH_SYSTEMVAR The string is a system variable.

CH_SYSTEMFUN The string is a system function.

CH_EXTERNVAR The string is a external variable.

Parameters
name The input string.

Description
This function returns an integer to indicate the characteristic of the input string *name*.

Example
See **iskey**()

See Also
isenv(), **iskey**(), **isnum**(), **isstudent**().

sizeofelement

Synopsis
#include <**chshell.h**>
int sizeofelement(**int** *etype*);

Purpose
Find the size of elements of an array.

Return Value
This function returns the number of characters of the data type represented in integral value.

Parameters
etype An integer indicating the data type of the element of array.

Description
This function finds the size of an element of an array. The argument *etype* is the data type of the array, and it can be obtained by calling built-in function **elementtype**().

Example

```
/* sizeofelement() returns the size of the data type of the
   its argument. If it is an array, the data type of the array element
   is used */

#include <stdio.h>

void func1(double a[&]) {
    int size;

    size =sizeofelement(elementtype(a));
    printf("sizeofelement(a) = %d\n", size);
}

void func2(array double &a) {
    int size;

    size =sizeofelement(elementtype(a));
    printf("sizeofelement(a) = %d\n", size);
}
int main() {
    int    a1[3][4], b1[3];
    double a2[3][4], b2[3];
    func1(b1);
    func1(b2);
    func2(a1);
    func2(b1);
    func2(a2);
    func2(b2);
}
```

Output

```
sizeofelement(a) = 4
sizeofelement(a) = 8
```

```
sizeofelement(a) = 4
sizeofelement(a) = 4
sizeofelement(a) = 8
sizeofelement(a) = 8
```

See Also
elementtype().

Chapter 4

Complex Functions — <complex.h>

The header **complex.h** defines macros and declares functions that support complex arithmetic in the latest ISO C standard. Each synopsis specifies a family of functions consisting of a principal function with one or more double complex parameters and a double complex, or double, return value; and other functions with the same name but with f and l suffixes which are corresponding functions with float and long double parameters and return values.

Functions

The following functions are declared in the **complex.h** header file.

Function	Description
cabs()	Calculate complex absoulte value.
cacos()	Calculate complex arc cosine.
cacosh()	Calculate complex arc hyperbolic cosine.
carg()	Calculate complex argument.
casin()	Calculate complex arc sine.
casinh()	Calculate complex arc hyperbolic sine.
catan()	Calculate complex arc tangent.
catanh()	Calculate complex arc hyperbolic tangent.
ccos()	Calculate complex cosine.
ccosh()	Calculate complex hyperbolic cosine.
cexp()	Calculate complex exponential.
cimag()	Obtain the imaginary part of a complex number.
clog()	Calculate complex logarithm.
conj()	Calculate complex conjugate.
cpow()	Calculate complex power of two variables.
creal()	Obtain the real part of a complex number.
csin()	Calculate complex sine.
csinh()	Calculate complex hyperbolic sine.
ctan()	Calculate complex tangent.
ctanh()	Calculate complex hyperbolic tangent.
iscnan()	Test if the argument is a complex Not-a-Number.

Macros

The following macro is defined by the **complex.h** header file.

Macro	Description
I	Imaginary number I expands to **complex(0.0, 1.0)**.

Portability

This header has no known portability problem.

Differences between C and Ch

Function **conj** is a built-in function in Ch.

cabs

Synopsis
#include <**complex.h**>
double cabs(**double complex** *z*);
float cabsf(**float complex** *z*);
long double cabsl(**long double complex** *z*);

Purpose
Calculates an absolute value of a complex number.

Return Value
The **cabs** functions return the complex absoulte value.

Parameters
z Complex argument.

Description
The **cabs** functions compute the complex absolute value (also called norm, modulus or magnitude) of z.

It is recommended that the polymorphic generic function **abs**(), instead of the **cabs**() function, be used to compute the complex absolute value of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("cabs(z) = %f\n", cabs(z));
}
```

Output

```
cabs(z) = 2.236068
```

See Also
abs().

cacos

Synopsis
#include <**complex.h**>
double complex cacos(**double complex** *z*);
float complex cacosf(**float complex** *z*);
long double complex cacosl(**long double complex** *z*);

Purpose
Calculates an arc cosine of a complex number.

Return Value
The **cacos** functions return the complex arc cosine value, in the range of a strip mathematically unbound along the imaginary axis, and in the interval [0, π] along the real axis.

Parameters
z Complex argument.

Description
The **cacos** functions compute the complex arc cosine of *z*, with branch cuts outside the interval [-1, 1] along the real axis.

It is recommended that the polymorphic generic function **acos()** instead of the **cacos**() function, be used to compute the complex arc cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("cacos(z) = %f\n", cacos(z));
}
```

Output

```
cacos(z) = complex(1.143718,-1.528571)
```

See Also
cos(), **ccos**(), **cacosh**(), **ccosh**().

cacosh

Synopsis
#include <**complex.h**>
double complex cacosh(**double complex** *z*);
float complex cacoshf(**float complex** *z*);
long double complex cacoshl(**long double complex** *z*);

Purpose
Calculates an arc hyperbolic cosine of a complex number.

Return Value
The **cacosh** functions return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis, and in the interval [-$i\pi$, $i\pi$] along the imaginary axis.

Parameters
z Complex argument.

Description
The **cacosh** functions compute the complex arc cosine of *z*, with branch a cut at values less than 1 along the real axis.

It is recommended that the polymorphic generic function **acosh()**, instead of the **cacosh**() function, be used to compute the complex arc hyperbolic cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("cacosh(z) = %f\n", cacosh(z));
}
```

Output

```
cacosh(z) = complex(1.528571,1.143718)
```

See Also
cos(), **cacos**(), **ccos**(), **ccosh**().

carg

Synopsis
#include <**complex.h**>
double carg(**double complex** *z*);
float cargf(**float complex** *z*);
long double cargl(**long double complex** *z*);

Purpose
Calculates an argument of a complex number.

Return Value
The **carg** functions return the value of the argument in the range [-π, π].

Parameters
z Complex argument.

Description
The **carg** functions compute the argument (also called phase angle) of z, with a branch cut along the negative real axis.

It is recommended that the polymorphic generic function **arg()**, instead of the **carg**() function, be used to compute the argumentof a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
  int i=7;
  double f = 30;
  double d = -10;
  complex z = complex(3, 4);
  printf("carg(%d) = %f radian \n", i, carg(i));
  printf("carg(%f) = %f radian \n", f, carg(f));
  printf("carg(%lf) = %f radian \n", d, carg(d));
  printf("carg(%f) = %f radian \n", z, carg(z));
}
```

Output

```
carg(7) = 0.000000 radian
carg(30.000000) = 0.000000 radian
carg(-10.000000) = 3.141593 radian
carg(complex(3.000000,4.000000)) = 0.927295 radian
```

See Also

casin

Synopsis
#include <**complex.h**>
double complex casin(**double complex** *z*);
float complex casinf(**float complex** *z*);
long double complex casinl(**long double complex** *z*);

Purpose
Calculates an arc sine of a complex number.

Return Value
The **casin** functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis, and in the interal [-π/2, π/2] along the real axis.

Parameters
z Complex argument.

Description
The **casin** functions compute the complex arc sine of *z*, with branch cuts outside the interval [-1, 1] along the real axis.

It is recommended that the polymorphic generic function **asin()**, instead of the **casin**() function, be used to compute the complex arc sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("casin(z) = %f\n", casin(z));
}
```

Output

```
casin(z) = complex(0.427079,1.528571)
```

See Also
sin(), **csin**(), **casinh**(), **csinh**().

casinh

Synopsis
#include <**complex.h**>
double complex casinh(**double complex** *z*);
float complex casinhf(**float complex** *z*);
long double complex casinhl(**long double complex** *z*);

Purpose
Calculates an arc hyperbolic sine of a complex number.

Return Value
The **casinh** functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis, and in the interval [-$i\pi$/2, $i\pi$/2] along the imaginary axis.

Parameters
z Complex argument.

Description
The **casinh** functions compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval $[-i, i]$ along the imaginary axis.

It is recommended that the polymorphic generic function **asinh()**, instead of the **casinh**() function, be used to compute the complex arc hyperbolic sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("casinh(z) = %f\n", casinh(z));
}
```

Output

```
casinh(z) = complex(1.469352,1.063440)
```

See Also
sin(), **casin**(), **csin**(), **csinh**().

catan

Synopsis
#include <**complex.h**>
double complex catan(**double complex** *z*);
float complex catanf(**float complex** *z*);
long double complex catanl(**long double complex** *z*);

Purpose
Calculates an arc tangent of a complex number.

Return Value
The **catan** functions return the complex are tangent value, in the range of a strip mathematically unbounded along the imaginary axis, and in the interval [-π/2, π/2].

Parameters
z Complex argument.

Description
The **catan** functions compute the complex arc tangent of *z*, with branch cuts outside the interval [-*i*, *i*] along the imaginary axis.

It is recommended that the polymorphic generic **atan()** function, instead of the **catan**() function, be used to compute the complex arc tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("catan(z) = %f\n", catan(z));
}
```

Output

```
catan(z) = complex(1.338973,0.402359)
```

See Also
tan(), **ctan**(), **catanh**(), **ctanh**().

catanh

Synopsis
#include <**complex.h**>
double complex catanh(**double complex** *z*);
float complex catanhf(**float complex** *z*);
long double complex catanhl(**long double complex** *z*);

Purpose
Calculates an arc hyperbolic tangent of a complex number.

Return Value
The **catanh** functions reutrn the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis, and in the interval [-*i*π/2, *i*π/2] along the imaginary axis.

Parameters
z Complex argument.

Description
The **catanh** functions compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval [-1, 1] along the real axis.

It is recommended that the polymorphic generic **atanh()** function, instead of the **catanh**() functions, be used to compute the complex arc hyperbolic tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("catanh(z) = %f\n", catanh(z));
}
```

Output

```
catanh(z) = complex(0.173287,1.178097)
```

See Also
tan(), **catan**(), **ctan**(), **ctanh**().

CCOS

Synopsis
#include <**complex.h**>
double complex ccos(**double complex** *z*);
float complex ccosf(**float complex** *z*);
long double complex ccosl(**long double complex** *z*);

Purpose
Calculates a cosine of a complex number.

Return Value
The **ccos** functions return the complex cosine.

Parameters
z Complex argument.

Description
The **cacos** functions compute the complex cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("ccos(z) = %f\n", ccos(z));
}
```

Output

```
ccos(z) = complex(2.032723,-3.051898)
```

See Also
cos(), **cacos**(), **cacosh**(), **ccosh**().

ccosh

Synopsis
#include <**complex.h**>
double complex ccosh(**double complex** *z*);
float complex ccoshf(**float complex** *z*);
long double complex ccoshl(**long double complex** *z*);

Purpose
Calculates a hyperbolic cosine of a complex number.

Return Value
The **ccosh** functions return the complex hyperbolic cosine.

Parameters
z Complex argument.

Description
The **cacosh** functions compute the complex hyperbolic cosine of z.

It is recommended that the polymorphic generic function **cosh()**, instead of the **ccosh**() function, be used to compute the complex hyperbolic cosine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("ccosh(z) = %f\n", ccosh(z));
}
```

Output

```
ccosh(z) = complex(-0.642148,1.068607)
```

See Also
cos(), **cacos**(), **ccos**(), **cacosh**().

cexp

Synopsis
#include <**complex.h**>
double complex cexp(**double complex** *z*);
float complex cexpf(**float complex** *z*);
long double complex cexpl(**long double complex** *z*);

Purpose
Calculates a base-*e* exponential of a complex number.

Return Value
The **cexp** functions return the complex base-*e* exponential value.

Parameters
z Complex argument.

Description
The **cexp** functions compute the complex base-*e* exponential of z.

It is recommended that the polymorphic generic function **exp()**, instead of the **cexp**() function, be used to compute the base-*e* exponential of complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("cexp(z) = %f\n", cexp(z));
}
```

Output

```
cexp(z) = complex(-1.131204,2.471727)
```

See Also
exp().

cimag

Synopsis
#include <**complex.h**>
double complex cimag(**double complex** *z*);
float complex cimagf(**float complex** *z*);
long double complex cimagl(**long double complex** *z*);

Purpose
Get the imaginary part of a complex number.

Return Value
The **cimag** functions return the imaginary part value (as a real).

Parameters
z Complex argument.

Description
The **cimag** functions compute the imaginary part of z.

It is recommended that the polymorphic generic function **imag**(), instead of the **cimag**() function, be used to obtain the imaginary part of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("cimag(z) = %f\n", cimag(z));
}
```

Output

```
cimag(z) = 2.000000
```

See Also
imag().

clog

Synopsis
#include <**complex.h**>
double complex clog(**double complex** *z*);
float complex clogf(**float complex** *z*);
long double complex clogl(**long double complex** *z*);

Purpose
Calculates a natural base-*e* logarithm of a complex number.

Return Value
The **clog** functions return the complex natural base-*e* logarithm value, in the range of a strip mathematically unbounded along the real axis, and in the interal [-iπ, iπ] along the imaginary axis.

Parameters
z Complex argument.

Description
The **clog** functions compute the complex natural base-*e* logarithm of z, with a branch cut along the negative real axis.

It is recommended that the polymorphic generic **log()** functions, instead of the **clog**() functions, be used to compute the natural base-*e* logarithm of the complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("clog(z) = %f\n", clog(z));
}
```

Output

```
clog(z) = complex(0.628609,1.107149)
```

See Also
log().

conj

Synopsis
#include <**complex.h**>
double complex conj(**double complex** *z*);
float complex conjf(**float complex** *z*);
long double complex conjl(**long double complex** *z*);

Purpose
Calculates the conjugate of a complex number.

Return Value
The **conj** functions return the complex conjugate value.

Parameters
z Complex argument.

Description
The **conj** functions compute the complex conjugate of z, by reversing the sign of its imaginary part.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("conj(z) = %f\n", conj(z));
}
```

Output

```
conj(z) = complex(1.000000,-2.000000)
```

See Also

cpow

Synopsis
#include <**complex.h**>
double complex cpow(**double complex** *z*, **double complex** *y*);
float complex cpowf(**float complex** *z*, **float complex** *y*);
long double complex cpowl(**long double complex** *z*,**long double** complex *y*);

Purpose
Calculates the power function of a complex number.

Return Value
The **cpow** functions return the complex power function value.

Parameters
z Complex argument.

y Complex argument.

Description
The **cpow** functions compute the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

It is recommended that the polymorphic generic function **pow()**, instead of the **cpow**() function, be used to compute the power function of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   double complex y = complex(3, 4);
   printf("cpow(z,y) = %f\n", cpow(z,y));
}
```

Output

```
cpow(z,y) = complex(0.129010,0.033924)
```

See Also
pow().

creal

Synopsis
#include <**complex.h**>
double creal(**double complex** *z*);
float crealf(**float complex** *z*);
long double creall(**long double complex** *z*);

Purpose
Get the real part of a complex number.

Return Value
The **creal** functions return the real part value.

Parameters
z Complex argument.

Description
The **creal** functions compute the real part of *z*.

It is recommended that the polymorphic generic function **real**(), instead of the **creal**() function, be used to compute the real part of a complex number *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("conj(z) = %f\n", conj(z));
}
```

Output

```
conj(z) = complex(1.000000,-2.000000)
```

See Also
real().

csin

Synopsis
#include <**complex.h**>
double complex csin(**double complex** *z*);
float complex csinf(**float complex** *z*);
long double complex csinl(**long double complex** *z*);

Purpose
Calculates a sine of a complex number.

Return Value
The **csin** functions return the complex sine.

Parameters
z Complex argument.

Description
The **csin** functions compute the complex sine of *z*.

It is recommended that the polymorphic generic **sin()** functions, instead of the **csin**() functions, be used to compute the complex sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("csin(z) = %f\n", csin(z));
}
```

Output

```
csin(z) = complex(3.165779,1.959601)
```

See Also
sin(), **casin**(), **casinh**(), **csinh**().

csinh

Synopsis
#include <**complex.h**>
double complex csinh(**double complex** *z*);
float complex csinhf(**float complex** *z*);
long double complex csinhl(**long double complex** *z*);

Purpose
Calculates a hyperbolic sine of a complex number.

Return Value
The **csinh** functions return the complex hyperbolic sine.

Parameters
z Complex argument.

Description
The **csinh** functions compute the complex hyperbolic sine of z.

It is recommended that the polymorphic generic **sinh()** functions, instead of the **csinh**() functions, be used to compute the complex hyperbolic sine of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("csinh(z) = %f\n", csinh(z));
}
```

Output

```
csinh(z) = complex(-0.489056,1.403119)
```

See Also
sin(), **casin**(), **csin**(), **casinh**().

csqrt

Synopsis
#include <**complex.h**>
double complex csqrt(**double complex** *z*);
float complex csqrtf(**float complex** *z*);
long double complex csqrtf(**long double complex** *z*);

Purpose
Calculates a square root of a complex number.

Return Value
The **csqrt** functions return the complex square root value, in the range of the right half plane (including the imaginary axis).

Parameters
z Complex argument.

Description
The **csqrt** functions compute the complex complex square root of z, with a branch cut clong the negative real axis.

It is recommended that the polymorphic generic **sqrt()** functions, instead of the **csqrt**() function, be used to compute the complex square root of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("csqrt(z) = %f\n", csqrt(z));
}
```

Output

```
csqrt(z) = complex(1.272020,0.786151)
```

See Also

ctan

Synopsis
#include <**complex.h**>
double complex ctan(**double complex** *z*);
float complex ctanf(**float complex** *z*);
long double complex ctanl(**long double complex** *z*);

Purpose
Calculates a tangent of a complex number.

Return Value
The **ctan** functions return the complex tangent.

Parameters
z Complex argument.

Description
The **ctan** functions compute the complex tangent of *z*.

It is recommended that the polymorphic generic **tan()** functions, instead of the **ctan**() functions, be used to compute the complex tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("ctan(z) = %f\n", ctan(z));
}
```

Output

```
ctan(z) = complex(0.033813,1.014794)
```

See Also
tan(), **catan**(), **catanh**(), **ctanh**().

ctanh

Synopsis
#include <**complex.h**>
double complex ctanh(**double complex** *z*);
float complex ctanhf(**float complex** *z*);
long double complex ctanhl(**long double complex** *z*);

Purpose
Calculates a hyperbolic tangent of a complex number.

Return Value
The **ctanh** functions return the complex hyperbolic tangent.

Parameters
z Complex argument.

Description
The **ctanh** functions compute the complex hyperbolic tangent of z.

It is recommended that the polymorphic generic **tanh()** functions, instead of the **ctanh**() functions, be used to compute the complex hyperbolic tangent of *z*.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z = complex(1, 2);
   printf("ctanh(z) = %f\n", ctanh(z));
}
```

Output

```
ctanh(z) = complex(1.166736,-0.243458)
```

See Also
tan(), **catan**(), **ctan**(), **catanh**().

iscnan

Synopsis
#include <**complex.h**>
int iscnan(**complex** *z*);

Purpose
Tests whether the complex number is a complex Not-a-Number, ComplexNaN.

Return Value
The **iscnan** function returns 1 value when *z* is a complex Not-a-Number.

Parameters
z complex number.

Description
The **iscnan** function tests whether the complex number is a complex Not-a-Number.

Example

```
#include <complex.h>
#include <stdio.h>

int main () {
   double complex z1 = complex(1, 2);
   double complex z2 = ComplexNaN;
   complex z3 = complex(1,2);
   complex z4 = ComplexNaN;
   printf("iscnan(z1) = %d\n", iscnan(z1));
   printf("iscnan(z2) = %d\n", iscnan(z2));
   printf("iscnan(z3) = %d\n", iscnan(z3));
   printf("iscnan(z4) = %d\n", iscnan(z4));
   printf("iscnan(ComplexNaN) = %d\n", iscnan(ComplexNaN));
   printf("iscnan(ComplexInf) = %d\n", iscnan(ComplexInf));
}
```

Output

```
iscnan(z1) = 0
iscnan(z2) = 1
iscnan(z3) = 0
iscnan(z4) = 1
iscnan(ComplexNaN) = 1
iscnan(ComplexInf) = 0
```

See Also

Chapter 5

Character Handling — <ctype.h>

The header **ctype.h** declares several functions useful for testing and mapping characters. In all cases the argument is of int type, the value of which shall be representable as an **unsigned character** or shall equal the value of the macro **EOF**, defined in header file **stdio.h** to represent end-of-file. If the argument has any other value, the behavior is undefined. These functions shall return a nonzero value for true or zero for false.

The behavior of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the "C" locale are noted below. The term *printing character* refers to a member of locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.

Functions

Function	Description
isalnum()	Tests if an argument is a digit or alphabetic character.
isalpha()	Tests if an argument is an alphabetic character.
iscntrl()	Tests if an argument is a control character.
isdigit()	Tests if an argument is a decimal-digit character.
isgraph()	Tests if an argument is a printing character.
islower()	Tests if an argument is a lowercase character.
isprint()	Tests if an argument is a printing character.
ispunct()	Tests if an argument is a punctuation character.
isspace()	Tests if an argument is a white-space character.
isupper()	Tests if an argument is an uppercase character.
isxdigit()	Tests if an argument is a hexadecimal-digit character.
tolower()	Converts an uppercase character to a lowercase character.
toupper()	Converts a lowercase character to an uppercase character.

Portability
This header has no known portability problem.

isalnum

Synopsis
#include <**ctype.h**>
int isalnum(**int** *c*);

Purpose
Tests whether the character is a digit or alphabetic character.

Return Value
The **isalnum** function returns a nonzero value when *c* is a digit or alphabetic character.

Parameters
c Character.

Description
The **isalnum** function tests for any character for which **isalpha** or **isdigit** is true.

Example

```
/* a sample program testing isalnum() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c='A';
    printf("isalnum('%c') returns %d\n", c,isalnum(c));
    if(isalnum(c) > 0)
        printf("'%c' is an alphanumeric character.\n",c);
    else
        printf("'%c' is not an alphanumeric character.\n", c);
    c = '4';
    printf("isalnum('%c') returns %d\n", c, isalnum(c));
    if(isalnum(c) > 0)
        printf("'%c' is an alphanumeric character.\n",c);
    else
        printf("'%c' is not an alphanumeric character.\n", c);

}
```

Output

```
isalnum('A') returns 1
'A' is an alphanumeric character.
isalnum('4') returns 1
'4' is an alphanumeric character.
```

See Also

isalpha

Synopsis
#include <**ctype.h**>
int isalpha(**int** *c*);

Purpose
Tests whether the character is an alphabetic character.

Return Value
The **isalpha** function returns a non-zero value when *c* is an alphabetic character.

Parameters
c Character.

Description
The **isalpha** function tests for any character for which **isupper** or **islower** is true, or any character that is one of a locale-specific set of alphabetic characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace**is true. In the "C" locale, **isalpha** returns true only when **isupper** or **islower** is true.

Example

```
/* a sample program testing isalpha() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c='A';
    printf("isalpha('%c') returns %d\n", c, isalpha(c));
    if(isalpha(c) > 0)
        printf("'%c' is an alphabetical character.\n", c);
    else
        printf("'%c' is not an alphabetical character.\n",c);
    c =' ';
    printf("isalpha('%c') returns %d\n", c,isalpha(c));
    if(isalpha(c) > 0)
        printf("'%c' is an alphabetical character.\n", c);
    else
        printf("'%c' is not an alphabetical character.\n",c);

}
```

Output

```
isalpha('A') returns 1
'A' is an alphabetical character.
isalpha(' ') returns 0
' ' is not an alphabetical character.
```

See Also

iscntrl

Synopsis
#include <**ctype.h**>
int iscntrl(**int** *c*);

Purpose
Tests whether the character is a control character.

Return Value
The **iscntrl** function returns a non-zero value when *c* is a control character.

Parameters
c Character.

Description
The **iscntrl** function test for any control character.

Example

```
/* a sample program testing iscntrl() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("iscntrl('%c') returns %d\n", c, iscntrl(c));
    if(iscntrl(c) > 0)
        printf("'%c' is a control character.\n", c);
    else
        printf("'%c' is not a control character.\n",c);
}
```

Output

```
iscntrl('A') returns 0
'A' is not a control character.
```

See Also

isdigit

Synopsis
#include <**ctype.h**>
int isdigit(**int** *c*);

Purpose
Tests whether the character is a decimal-digit character.

Return Value
The **isdigit** function returns a non-zero value when *c* is a decimal-digit character.

Parameters
c Character.

Description
The **isdigit** function tests for any decimal-digit character.

Example

```
/* a sample program testing isdigit() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isdigit('%c') returns %d\n", c, isdigit(c));
    if(isdigit(c) > 0)
        printf("'%c' is a digit.\n", c);
    else
        printf("'%c' is not a digit.\n",c);
    c = '1';
    printf("isdigit('%c') returns %d\n", c, isdigit(c));
    if(isdigit(c) > 0)
        printf("'%c' is a digit.\n",c);
    else
        printf("'%c' is not a digit.\n",c);

}
```

Output

```
isdigit('A') returns 0
'A' is not a digit.
isdigit('1') returns 1
'1' is a digit.
```

See Also

isgraph

Synopsis
#include <**ctype.h**>
int isgraph(**int** *c*);

Purpose
Tests whether the character is a printing character.

Return Value
The **isgraph** function returns a non-zero value when *c* is a printing character.

Parameters
c Character.

Description
The **isgraph** function tests for any printing character, except space (**' '**).

Example

```
/* a sample program testing isgraph() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isgraph('%c') returns %d\n", c, isgraph(c));
    if(isgraph(c) > 0)
        printf("'%c' is a printable character.\n", c);
    else
        printf("'%c' is not a printable character or it is a space.\n", c);
    c = ' ';
    printf("isgraph('%c') returns %d\n", c, isgraph(c));
    if(isgraph(c) > 0)
        printf("'%c' is a printable character.\n", c);
    else
        printf("'%c' is not a printable character or it is a space.\n", c);
}
```

Output

```
isgraph('A') returns 1
'A' is a printable character.
isgraph(' ') returns 0
' ' is not a printable character or it is a space.
```

See Also

islower

Synopsis
#include <**ctype.h**>
int islower(**int** *c*);

Purpose
Tests whether the character is a lowercase letter.

Return Value
The **islower** functions return a non-zero value when *c* is a lowercase character.

Parameters
c Character.

Description
The **islower** function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the "C" locale, **islower** returns true only for the characters defined as lowercase letters.

Example

```
/* a sample program testing islower() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("islower('%c') returns %d\n", c,islower(c));
    if(islower(c) > 0)
        printf("'%c' is a lower case character.\n", c);
    else
        printf("'%c' is not a lower case character.\n", c);
    c = 'a';
    printf("islower('%c') returns %d\n", c,islower(c));
    if(islower(c) > 0)
        printf("'%c' is a lower case character.\n", c);
    else
        printf("'%c' is not a lower case character.\n",c);

}
```

Output

```
islower('A') returns 0
'A' is not a lower case character.
islower('a') returns 1
'a' is a lower case character.
```

See Also

isprint

Synopsis
#include <**ctype.h**>
int isprint(**int** *c*);

Purpose
Tests whether the character is a printing character.

Return Value
The **isprint** function returns a non-zero value when *c* is a printing character.

Parameters
c Character.

Description
The **isprint** function test for any printing character, including space (**' '**).

Example

```
/* a sample program testing isprint() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isprint('%c') returns %d\n", c,isprint(c));
    if(isprint(c) > 0)
        printf("'%c' is a printable character.\n", c);
    else
        printf("'%c' is not a printable character.\n", c);
    c = ' ';
    printf("isprint('%c') returns %d\n", c,isprint(c));
    if(isprint(c) > 0)
        printf("'%c' is a printable character.\n", c);
    else
        printf("'%c' is not a printable character.\n",c);

}
```

Output

```
isprint('A') returns 1
'A' is a printable character.
isprint(' ') returns 1
' ' is a printable character.
```

See Also

ispunct

Synopsis
#include <**ctype.h**>
int ispunct(**int** *c*);

Purpose
Tests whether the character is a punctuation character.

Return Value
The **ispunct** functions return a non-zero value when *c* is a punctuation character.

Parameters
c Character.

Description
The **ispunct** function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither **isspace** nor **isalnum** is true. In the "C" locale, **ispunct** returns true for every printing character for which neither **isspace** nor **isalnum** is true.

Example

```
/* a sample program testing ispunct() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("ispunct('%c') returns %d\n", c,ispunct(c));
    if(ispunct(c) > 0)
        printf("'%c' is a punctuation mark.\n", c);
    else
        printf("'%c' is not a punctuation mark.\n", c);
    c = ';';
    printf("ispunct('%c') returns %d\n", c,ispunct(c));
    if(ispunct(c) > 0)
        printf("'%c' is a punctuation mark.\n", c);
    else
        printf("'%c' is not a punctuation mark.\n",c);
}
```

Output

```
ispunct('A') returns 0
'A' is not a punctuation mark.
ispunct(';') returns 1
';' is a punctuation mark.
```

See Also

isspace

Synopsis
#include <**ctype.h**>
int isspace(**int** *c*);

Purpose
Tests whether the character is a white-space character.

Return Value
The **isspace** functions return a non-zero value when *c* is a white-space character.

Parameters
c Character.

Description
The **isspace** function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which **isalnum** is false. The standard white-space characters are the following: space (**' '**), form feed (**'\f'**), new-line (**'\n'**), carriage return (**'\r'**), horizontal tab (**'\t'**), and vertical tab (**'\v'**). In the "C" locale, **isspace** returns true only for the standard white-space characters.

Example

```
/* a sample program testing isspace() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isspace('%c') returns %d\n", c,isspace(c));
    if(isspace(c) > 0)
        printf("'%c' is a white space character.\n", c);
    else
        printf("'%c' is not a white space character.\n", c);
    c = ' ';
    printf("isspace('%c') returns %d\n", c,isspace(c));
    if(isspace(c) > 0)
        printf("'%c' is a white space character.\n", c);
    else
        printf("'%c' is not a white space character.\n",c);
}
```

Output

```
isspace('A') returns 0
'A' is not a white space character.
isspace(' ') returns 1
' ' is a white space character.
```

See Also

isupper

Synopsis
#include <**ctype.h**>
int isupper(**int** *c*);

Purpose
Tests whether the character is an uppercase character.

Return Value
The **isupper** function returns a non-zero value when *c* is an uppercase character.

Parameters
c Character.

Description
The **isupper** function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the "C" locale, **isupper** returns true only for the characters defined as uppercase letters.

Example

```
/* a sample program testing isupper() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'A';
    printf("isupper('%c') returns %d\n", c,isupper(c));
    if(isupper(c) > 0)
        printf("'%c' is a upper case character.\n", c);
    else
        printf("'%c' is not a upper case character.\n", c);
    c = 'a';
    printf("isupper('%c') returns %d\n", c,isupper(c));
    if(isupper(c) > 0)
        printf("'%c' is a upper case character.\n", c);
    else
        printf("'%c' is not a upper case character.\n",c);
}
```

Output

```
isupper('A') returns 1
'A' is a upper case character.
isupper('a') returns 0
'a' is not a upper case character.
```

See Also

isxdigit

Synopsis
#include <**ctype.h**>
int isxdigit(**int** *c*);

Purpose
Tests whether the character is a hexadecimal-digit character.

Return Value
The **isxdigit** functions return a non-zero value when *c* is an hexadecimal-digit character.

Parameters
c Character.

Description
The **isxdigit** function tests for any hexadecimal-digit character.

Example

```
/* a sample program testing isxdigit() */
#include <ctype.h>
#include <stdio.h>

int main() {
    char c = 'T';
    printf("isxdigit('%c') returns %d\n", c,isxdigit(c));
    if(isxdigit(c) > 0)
        printf("'%c' is a hexadecimal-digit character.\n", c);
    else
        printf("'%c' is not a hexadecimal-digit character.\n", c);
    c = 'a';
    printf("isxdigit('%c') returns %d\n", c,isxdigit(c));
    if(isxdigit(c) > 0)
        printf("'%c' is a hexadecimal-digit character.\n", c);
    else
        printf("'%c' is not a hexadecimal-digit character.\n",c);
}
```

Output

```
isxdigit('T') returns 0
'T' is not a hexadecimal-digit character.
isxdigit('a') returns 1
'a' is a hexadecimal-digit character.
```

See Also

tolower

Synopsis
#include <**ctype.h**>
int tolower(**int** *c*);

Purpose
Converts an uppercase character to a lowercase character.

Return Value
If the arguement for which **isupper** is true, and there are one or more corresponding characters, as specified by the current locale, for which **islower** is true, the **tolower** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the arguement is returned unchanged.

Parameters
c Character.

Description
The **tolower** function converts an uppercase letter to a corresponding lowercase letter.

Example

```
/* a sample program that displays lower case of the letter. */
#include <stdio.h>
#include <ctype.h>

int main() {
    char c = 'Q';
    printf("tolower('%c') = %c\n", c,tolower('Q'));
}
```

Output

```
tolower('Q') = q
```

See Also

toupper

Synopsis
#include <**ctype.h**>
int toupper(**int** *c*);

Purpose
Converts a lowercase character to an uppercase character.

Return Value
If the arguement for which **islower** is true, and there are one or more corresponding characters, as specified by the current locale, for which **isupper** is true, the **toupper** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the arguement is returned unchanged.

Parameters
c Character.

Description
The **toupper** function converts a lowercase character to a corresponding uppercase character.

Example

```
/* a sample program that displays uppercase of the letter. */
#include <stdio.h>
#include <ctype.h>

int main() {
    char c = 'a';
    printf("toupper('%c') = %c\n", c, toupper(c));
}
```

Output

```
toupper('a') = A
```

See Also

Chapter 6

Errors — <errno.h>

The header **errno.h** defines several macros, all relating to the reporting of error conditions.

The macros are
EDOM
EILSEQ
ERANGE
which expand to integer constant expressions with type **int**, distinct positive values, and which are suitable for use in **#if** preprocessing directives; and
errno
which expands to a modifiable lvalue that has type **int**, the value of which is set to a positive error number by several library functions.
The value of **errno** is zero at program startup, but is never set to zero by any libary function. The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in this International Standard.

Additional macro definitions, beginning with **E** and a digit or **E** and an uppercase letter, are specified in the *errno.h* header file.

Macros
The following macros are defined bt the **errno.h** header file.

Macro	Description
EDOM	Math arg out of domain of func
EILSEQ	Illegal byte sequence.
ERANGE	Math result not representable

Portability
This header has no known portability problem.

Chapter 7

Characteristics of floating types <float.h>

The header **float.h** defines several macros that expand to various limits and parameters of the standard floating-point types.

The macros, their meanings , and the constraints (or restrictions) on their values are listed below.

Macros

The following macros are defined by the **float**.h header file.

Macro	Constraint on the Value	Description
FLT_RADIX	**2**	radix of exponent representation.

Macro	Constraint on the Value	Description
FLT_MANT_DIG **DBL_MANT_DIG** **LDBL_MANT_DIG**	**24** **53**	number of base-**FLT_RADIX**digits in the floating-point significant.

Macro	Constraint on the Value	Description
FLT_DIG **DBL_DIG** **LDBL_DIG**	**6** **15**	number of decimal digits.

Macro	Constraint on the Value	Description

FLT_MIN_EXP **DBL_MIN_EXP** **LDBL_MIN_EXP**	**(- 125)** **(- 1021)**	minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized

Macro	Constraint on the Value	Description
FLT_MIN_10_EXP **DBL_MIN_10_EXP** **LDBL_MIN_10_EXP**	**(- 37)** **(- 307)**	minimum negative integer such that 10 raised to that power is in the range of no number.

Macro	Constraint on the Value	Description
FLT_MAX_EXP **DBL_MAX_EXP** **LDBL_MAX_EXP**	**128** **1024**	maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number.

Macro	Constraint on the Value	Description
FLT_MAX_10_EXP **DBL_MAX_10_EXP** **LDBL_MAX_10_EXP**	**38** **308**	maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.

Macro	Constraint on the Value	Description
FLT_MAX **DBL_MAX** **LDBL_MAX**	**3.40282347E+38F** **1.797693134862315E+308**	maximum representable finite floating-point number.

Macro	Constraint on the Value	Description
FLT_EPSILON **DBL_EPSILON** **LDBL_EPSILON**	**1.19209290E-07F** **2.2204460492503131E-16**	the difference between 1 and the least value greater than 1 that is representable between 1 and the least.

Macro	Constraint on the Value	Description
FLT_MIN **DBL_MIN** **LDBL_MIN**	**1.17549435E-38F** **2.2250738585072014E-308**	minimum normalized positive floating-point number.

Portability
This header has no known portability problem.

Chapter 8

Sizes of integer types — <limits.h>

The header **limits.h** defines several macros that expand to various limits and parameters of the standard integer types.

The macros, their meanings, and the constraints (or restrictions) on their values are listed below.

Portability
This header has no known portability problem.

Macros
The following macros are defined by the **limits.h** header file.

Macro	Description Constraint on the Value
CHAR_BIT	number of bits for smallest object that is not a bit_filed(byte) **8**
SCHAR_MIN	minimum value for an object of type **signed char** **-128** // -2^7
SCHAR_MAX	maximum value for an object of type **signed char** **+127** // 2^7-1
UCHAR_MAX	maximum value for an object of type **unsigned char** **255** // 2^8-1
CHAR_MIN	minimum value for an object of type **char** **-128** // -2^7
CHAR_MAX	maximum value for an object of type **char** **+127** // 2^7-1
MB_LEN_MAX	maximum number of bytes in a multibyte character,for any supported locale **1** (This value is platform-dependent.)
SHRT_MIN	minimum value for an object of type **short int** **-32768** // -2^{15}
SHRT_MAX	maximum value for an object of type **short int** **+32767** // 2^{15}-1
USHRT_MAX	maximum value for an object of type **unsigned short int** **65535** // 2^{16}-1

INT_MIN	minimum value for an object of type **int** **-2147483648** // -2^{31}
INT_MAX	maximum value for an object of type **int** **+2147483647** // $2^{31}-1$
UINT_MAX	maximum value for an object of type **unsigned int** **4294967295** // $2^{32}-1$
LONG_MIN	minimum value for an object of type **long int** **-2147483648** // -2^{31}
LONG_MAX	maximum value for an object of type **long int** **2147483647** // $2^{31}-1$
ULONG_MAX	maximum value for an object of type **unsigned long int** **4294967295** // $2^{32}-1$
LLONG_MIN	minimum value for an object of type **long long int** **-9223372036854775808** // -2^{63}
LLONG_MAX	maximum value for an object of type **long long int** **+9223372036854775807** // $2^{63}-1$
ULLONG_MAX	maximum value for an object of type **unsigned long long int** **18446744073709551615** // $2^{64}-1$

Chapter 9

Localization <locale.h>

The header **locale.h** declares two functions, one type, and defines several macros.
The type is

struct lconv

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained later. In the Ch locale, the members shall have the values specified in the comments.

```
char   *decimal_point;       //  "."
char   *thousands_sep;       //  ""
char   *grouping;            //  ""
char   *mon_decimal_point    //  ""
char   *mon_thousands_sep;   //  ""
char   *mon_grouping;        //  ""
char   *positive_sign;       //  ""
char   *negative_sign;       //  ""
char   *currency_symbol;     //  ""
char   frac_digits;          //  CHAR_MAX
 char  p_cs_precedes;        //  CHAR_MAX
char   n_cs_precedes;        //  CHAR_MAX
char   p_sep_by_space;       //  CHAR_MAX
char   n_sep_by_space;       //  CHAR_MAX
char   p_sign_posn;          //  CHAR_MAX
char   n_sign_posn;          //  CHAR_MAX
char   *int_curr_symbol;     //  ""
char   int_frac_digits;      //  CHAR_MAX
char   int_p_cs_precedes;    //  CHAR_MAX
char   int_n_cs_precedes;    //  CHAR_MAX
char   int_p_sep_by_space;   //  CHAR_MAX
char   int_n_sep_by_space;   //  CHAR_MAX
char   int_p_sign_posn;      //  CHAR_MAX
char   int_n_sign_posn;      //  CHAR_MAX
```

The macros defined are

LC_ALL

LC_COLLATE

LC_CTYPE

LC_MONETARY

LC_NUMERIC

LC_TIME

which expand to integer constant expressions with distinct values, suitable for use as the first arguement to the **setlocale** function. Additional macro definitions, beginning with the characters **LC_** and an uppercase letter, may be added in the future version.

Functions

The following functions are defined by the **locale.h** header file.

Function	Description
localeconv()	Returns a pointer containing the values appropriate for the formatting of the numeric quantities according to the current locale.
setlocale()	Selects the appropriate piece of the program's locale as specified by the category, and may be used to change or query the program's international enviornment.

Macros

The following macros are defined by the **locale.h** header file.

Macro	Description
LC_ALL	Determines all localization categories.
LC_COLLATE	Affects the behavior of the string collation functions **strcoll()** and **strxfrm()**.
LC_CTYPE	Affects the behavior of the character classication and conversion functions.
LC_MONETARY	Affects the monetary unit format.
LC_NUMERIC	Determines the deciaml-point character for formatted input/output functions.
LC_TIME	Determines the behavior of the **strftime()** function.

Declared Types

The following types are defined by the **locale.h** header file.

Type	Description
lconv	struct - holds values related to the formatting of numerical values.

Portability

This header has no known portability problem.

localeconv

Synopsis
#include <**locale.h**>
structlconv ∗**localeconv**(**void**);

Purpose
Numeric formatting convention inquiry.

Return Value
The **localeconv** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **localeconv** function. In addition, calls to the **setlocale** function with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the structure.

Parameters
No argument.

Description
The **localeconv** function sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type **char** ∗ are pointers to strings, any of which (except **decimal_point**) can point to **""**, to indicate that the value is not available in the current locale or is of zero length. Apart from **grouping** and **mon_grouping**, the strings shall start and end in the initial shift state. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members include the following:

char ∗**decimal_point**
The decimal-point character used to format nonmonetary quantities.

char ∗**thousands_sep**
The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

char ∗**grouping**
A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char ∗**mon_decimal_point**
The decimal-point used to format monetary quantities.

char ∗**mon_thousands_sep**
The separator for groups of digits before the decimal-point in formatted monetary quantities.

char ∗**mon_grouping**
A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
The string used to indicate a nonnegative-valued formatted monetary quantity.

char *negative_sign
The string used to indicate a negative-valued formatted monetary quantity.

char *currency_symbol
The local currency symbol applicable to the current locale.

char frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

char n_cs_precedes
Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

char p_sep_by_space
Set to a value indicating the separation of the **currency_ symbol**, the sign string, and the value for a nonnegative locally formatted monetary quantity.

char n_sep_by_space
Set to a value indicating the separation of the **currency_ symbol**, the sign string, and the value for a negative locally formatted monetary quantity.

char p_sign_posn
Set to a value indicating the positioning of the **positive_ sign** for a nonnegative locally formatted monetary quantity.

char n_sign_posn
Set to a value indicating the positioning of the **negative_ sign** for a negative locally formatted monetary quantity.

char *int_curr_symbol
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217:1995. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char int_frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char int_p_cs_precedes
Set to 1 or 0 if the **int_currency_symbol** respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

char int_n_cs_precedes
Set to 1 or 0 if the **int_currency_symbol** respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

char int_p_sep_by_space
Set to a value indicating the separation of the **int_currency_symbol**, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

char int_n_sep_by_space
Set to a value indicating the separation of the **int_currency_symbol**, the sign string, and the value for a negative internationally formatted monetary quantity.

char int_p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a nonnegative internationally formatted monetary quantity.

char int_n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative internationally formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
other	The integer value is the number of digits that compose the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space** are interpreted according to the following:

0 No space separates the currency symbol and value.
1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.
2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

0 Parentheses surround the quantity and currency symbol.
1 The sign string precedes the quantity and currency symbol.
2 The sign string succeeds the quantity and currency symbol.
3 The sign string immediately precedes the currency symbol.
4 The sign string immediately succeeds the currency symbol.

The implementation shall behave as if no library function calls the **localeconv** function.

EXAMPLE 1 The following table illustrates the rules which may well by used by four countries to format monetry quantities.

	Local format		International format	
Country	Positive	Negative	Positive	Negative
Finland	**1.234,56 mk**	**-1.234,56 mk**	**FIM 1.234,56**	**FIM -1.234,56**
Italy	**L.1.234**	**-L.1.234**	**ITL 1.234**	**-ITL 1.234**
Netherlands	**f 1.234,56**	**f -1.234,56**	**NLG 1.234,56**	**NLG -1.234,56**
Switzerland	**SFrs.1,234.56**	**SFrs.1,234.56**	**CHF 1,234.56**	**CHF -1,234.56**

For those four countires, the repective values for the monetary members of the structure returned by **localeconv** are:

	Finland	Itlay	Netherlands	Switzerland
mon_decimal_point	**","**	**""**	**","**	**"."**
mon_thousands_sep	**"."**	**"."**	**"."**	**","**
mon_grouping	**"\3"**	**"\3"**	**"\3"**	**"\3"**
positive_sign	**""**	**""**	**""**	**""**
negative_sign	**"-"**	**"-"**	**"-"**	**"C"**
currency_symbol	**"mk"**	**"L."**	**"\u0192"**	**"SFrs."**
frac_digits	**2**	**0**	**2**	**0**
p_cs_precedes	**0**	**1**	**1**	**1**
n_cs_precedes	**0**	**1**	**1**	**1**
p_sep_by_space	**1**	**0**	**1**	**0**
n_sep_by_space	**1**	**0**	**1**	**0**
p_sign_posn	**1**	**1**	**1**	**1**
n_sign_posn	**1**	**1**	**4**	**2**
int_curr_symbol	**"FIM"**	**"ITL"**	**"NLG"**	**"CHF"**
int_facr_digits	**2**	**0**	**2**	**2**
int_p_cs_precedes	**1**	**1**	**1**	**1**
int_n_cs_precedes	**1**	**1**	**1**	**1**
int_p_sep_by_space	**0**	**0**	**0**	**0**
int_n_sep_by_space	**0**	**0**	**0**	**0**
int_p_sign_posn	**1**	**1**	**1**	**1**
int_n_sign_posn	**4**	**1**	**4**	**2**

EXAMPLE 2 The following table illustrates how the p_cs_precedes, p_sep_by_space, and p_sign_posn members affect the formatted value.

p_cs_precedes	p_sign_posn	p_sep_by_space		
		0	1	2
0	0	(1.25$)	(1.25 $)	(1.25$)
	1	+1.25$	+1.25 $	+ 1.25$
	2	1.25$+	1.25 $+	1.25 $ +
	3	1.25+$	1.25 +$	1.25 + $
	4	1.25$+	1.25 $+	1.25 $ +
1	0	(1.25)	($ 1.25)	($1.25)
	1	+$1.25	+$ 1.25	+ $1.25
	2	$1.25+	$ 1.25+	$1.25 +
	3	+$1.25	+$ 1.25	+ $1.25
	4	$+1.25	$+1.25	$ +1.25

Example

```
/* a sample program that displays the decimal point
character used by the current locale. */
#include <stdio.h>
#include <locale.h>

int main() {
    struct lconv *lc;
    lc = localeconv();
    printf("Decimal symbol is: %s\n", lc->decimal_point);
    printf("Thousands separator is: %s\n", lc->thousands_sep);
    printf("Currency symbol is: %s\n", lc->currency_symbol);
}
```

Output

```
Decimal symbol is: .
Thousands separator is:
Currency symbol is:
```

See Also

setlocale

Synopsis
#include <**locale.h**>
char ∗**setlocale**(**int** *category*, **const char**∗ *locale*);

Purpose
Locale control.

Return Value
If a pointer to a string is given for *locale* and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified *category* for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

A null pointer for *locale* causes the **setlocale** function to return a pointer to the string associated with the *category* for the program's current locale; the program's locale is not changed.

The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale** function.

Parameters
category Category of locales.

locale Locale value.

Description
The setlocale function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for category names the program's entire locale; the other values for category name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions and the multibyte and wide-character functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** function.

A value of "C" for locale specifies the minimal environment for $\mathbf{C}^{\mathbf{H}}$ translation; a value of **""** for locale specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

At program startup, the equivalent of

setlocale(**LC_ALL**, "C");

is executed.

The implementation shall behave as if no library function calls the **setlocale** function.

Example

```
/* a sample program that displays the current locale setting.*/
#include <locale.h>
#include <stdio.h>

int main() {
    printf("local locale is %s\n", setlocale(LC_ALL,""));
}
```

Output

```
local locale is C
```

See Also
strcoll(), **strftime()**, **strxfrm()**.

Chapter 10

Mathematics <math.h>

The header **math.h** declares two types, several mathematical functions, and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more double parameters, a double return value, or both; and other functions with the same name but with **f** and ł suffixes which are corresponding functions with float and long double parameters, return values, or both. Integer arithmetic functions and conversion functions are discussed later.

The macro
HUGE_VAL
expands to a positive double constant expression, not necessarily representable as a **float**.
The macro
INFINITY
expands to a constant expression of type float representing positive or unsigned infinity, if available; else to a positive constant of type float that overflows at translation time.
The macro
NAN
is defined if and only if the implementation supports quiet NaNs for the **float** type. It expands to a constant expression of type **float** representing a quiet **NaN**.

The macros
FP_ILOGB0
FP_ILOGBNAN
expand to integer constant expressions whose values are returned by **ilogb(x)** if **x** is zero or NaN, respectively. The value of **FP_ILOGB0** shall be either **INT_MIN** or **INT_MAX**. The value of **FP_ILOGBNAN** shall be either **INT_MAX** or **INT_MIN**.

Functions

The following functions are defined by the **math.h** header file.

Function	Description
acos()	Calculates the arc cosine of an argument.
acosh()	Calculates the arc hyperbolic cosine of an argument.
asin()	Calculates the arc sine of an argument.
asinh()	Calculates the arc hyperbolic sine of an argument.
atan()	Calculates the arc tangent of an argument in the range [-π/2, π/2].
atan2()	Calculates the arc tangent of an argument in the range [-π, pi].
atanh()	Calculates the arc hyperbolic tangent of an argument.
cbrt()	Calculates the cubed root of an argument.
ceil()	Calculates the smallest integer value not less than the argument.
copysign()	Switches signs of two arguments.
cos()	Calculates the cosine of an argument.
cosh()	Calculates the hyperbolic cosine of an argument.
erf()	Calculates the error function.
erfc()	Calculates a modified error function.
exp()	Calculates the base-*e* exponential of an argument.
exp2()	Calculates the base-2 exponential of an argument.
expm1()	Calculates the base-*e* -1 exponential of an argument.
fabs()	Calculates the absolute value of an argument.
fdim()	Calculates the difference between two arguments.
floor()	Calculates the largest integer value not greater than the argument.
fma()	Calculates (**x**+**y**)$\times$**z**.
fmax()	Calculates the maximum value of two arguments.
fmin()	Calculates the minimum value of two arguments.
fmod()	Calculates the remaindes of two arguments.
frexp()	Calculates the normalized fractions of an argument.
hypot()	Calculates $\sqrt{z^2 + y^2}$.
ilogb()	Calculates the exponent of the argument as a signed **int** value.
ldexp()	Calculates **x**$\times 2^{exp}$.
lgamma()	Calculates the absolute logrithmic gamma value of an argument.
log()	Calculates the logarithm of an argument.
log10()	Calculates the base-10 logarithm of an argument.
log1p()	Calculates the base-*e* logarithim of one plus the argument.
log2()	Calculates the base-2 logarithm of an argument.
logb()	Calculates the signed exponent of the argument.
lrint()	Calculates the rounded integer value of an argument according to the current rounding direction.
lround()	Calculates the rounded integer value of an argument, rounding halfway cases to zero, regardless of ounding direction.
modf()	Calculates the signed fractional part of the argument.
nan()	Returns a quiet NaN.
nearbyint()	Calculates the rounded integer value of an argument, using the current rounding direction.

nextafter()	Calculates the next representable value of an argument.
nexttoward()	Calculates the next representable value of an argument.
pow()	Calculates $\mathbf{x}^{\mathbf{y}}$.
remainder()	Calculates the remainder of two arguments.
remquo()	Calculates the reaminder of two arguments.
rint()	Calculates the rounded integer value of an argument.
round()	Calculates the rounded integer value of an argument, rounding halfway cases away from zero, regardless of current rounding direction.
scalbn()	Calculates $\mathbf{x} \times \mathbf{FLT\_RADIX}^{n}$.
sin()	Calculates the sine of an argument.
sinh()	Calculates the hyperbolic sine of an argument.
sqrt()	Calculates the square root of an argument.
tan()	Calculates the tangent of an argument.
tanh()	Calculates the hyperbolic tangent of an argument.
tgamma()	Calculates the gamma function of an argument.
trunc()	Calculates the truncated integer value of an argument.

Macros

The following macros are defined by the **math.h** header file.

Macro	Description
HUGE_VAL	double - Positive constant expression.
INFINITY	float - Positive infinity.
NAN	float - Not-a-Number.
isfinite	int - Returns a nonzero value if the argument is finite.
isgreater	double - Returns the value of $\mathbf{x}>\mathbf{y}$.
isgreaterequal	double - Returns the value of $\mathbf{x}\geq\mathbf{y}$.
isinf	int - Returns a nonzero value if the argument is infinite.
isless	double - Returns the value of $\mathbf{x}<\mathbf{y}$.
islessequal	double - Returns the value of $\mathbf{x}\leq\mathbf{y}$.
islessgreater	double - Determines if **x** is less than, greater than, or equal to **y**.
isnan	int - Returns a nonzero value if the argument is NaN.
isnormal	int - Returns a nonzero value if the argument is normal, (neither zero, subnormal, infinite, nor NaN).
isunordered	Returns 1 if the argument is unordered.
signbit	int - Returns a nonzero value if the argument has a negative value.

Portability

Functions **isnormal**(), fpclassify(), signbit() are not available in all platforms. Functions **isunordered**(), **lrint**(), **lround**(), **NAN**(), **nearbyint**(), **nexttoward**(), **remquo**(), **round**(), **tgamma**(), and **trunc**() are not supported in any platform.

acos

Synopsis
#include <math.h>
type acos(**type** *x*);
double acos(**double** *x*);
float acosf(**float** *x*);
long double acosl(**long double** *x*);

Purpose
Calculate arc cosine.

Return Value
The **acos** functions return the arc cosine in the range [0, π] radians.

Parameters
x Argument.

Description
The **acos** functions compute the arc cosine of *x*. A domain error occurs for arguments not in the range [-1, +1]. The function returns the following values related to special numbers:
— **acos**($\pm\infty$) returns NaN.
— **acos**(NaN) returns NaN.
— **acos**(± 0.0) returns $\pi/2$.
— **acos**(x) returns NaN for $|x| > 1$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("acos(x) = %f\n", acos(x));
}
```

Output

```
acos(x) = 1.266104
```

See Also
cos(), **acosh**(), **cosh**().

acosh

Synopsis
#include <math.h>
type acosh(**type** *x*);
double acosh(**double** *x*);
float acoshf(**float** *x*);
long double acoshl(**long double** *x*);

Purpose
Calculate arc hyperbolic cosine.

Return Value
The **acosh** functions return the arc hyperbolic cosine in the range [0, ∞].

Parameters
x Argument.

Description
The **acosh** functions compute the (nonnegative) arc hyperbolic cosine of *x*. A domain error occurs for arguments less than 1. The function returns the following values related to special numbers:
— **acosh**($-\infty$) returns NaN.
— **acosh**($+\infty$) returns $+\infty$.
— **acosh**(NaN) returns NaN.
— $\mathbf{acosh}(-x)$ returns NaN, for $x > 0$.
— $\mathbf{acosh}(x)$ returns NaN, for $x < 1.0$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("acosh(x) = %f\n", acosh(x));
}
```

Output

```
acosh(x) = NaN
```

See Also
cos(), **acos**(), **cosh**().

asin

Synopsis
#include <math.h>
type asin(**type** *x*);
double asin(**double** *x*);
float asinf(**float** *x*);
long double asinl(**long double** *x*);

Purpose
Calculate arc sine.

Return Value
The **asin** functions return the arc sine in the range [-π/2, π/2].

Parameters
x Argument.

Description
The **asin** functions compute the principal value of the arc sine of *x*. A domain error ocurs for arguments not in the range [-1, +1]. The function returns the following values related to special numbers:
— $\mathbf{asin}(\pm\infty)$ returns NaN.
— **asin**(NaN) returns NaN.
— $\mathbf{asin}(\pm 0.0)$ returns ± 0.0.
— $\mathbf{asin}(x)$ returns NaN for $|x| > 1$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("asin(x) = %f\n", asin(x));
}
```

Output

```
asin(x) = 0.304693
```

See Also
sin(), **asinh**(), **sinh**().

asinh

Synopsis
#include <**math.h**>
type asinh(**type** *x*);
double asinh(**double** *x*);
float asinhf(**float** *x*);
long double asinhl(**long double** *x*);

Purpose
Calculate arc sine.

Return Value
The **asinh** functions return the arc hyperbolic sine value. The function returns the following values related to special numbers:
— **asinh**($\pm\infty$) returns $\pm\infty$.
— **asinh**(NaN) returns NaN.
— **asinh**(± 0.0) returns ± 0.0.
— **asinh**($-x$) returns -**asin**(x), for $x > 0$.

Parameters
x Argument.

Description
The **asinh** functions compute the arc hyperbolic sine of *x*.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("asinh(x) = %f\n", asinh(x));
}
```

Output

```
asinh(x) = 0.295673
```

See Also
sin(), **asin**(), **sinh**().

atan

Synopsis
#include <math.h>
type atan(**type** *x*);
double atan(**double** *x*);
float atanf(**float** *x*);
long double atanl(**long double** *x*);

Purpose
Calculate arc tangent.

Return Value
The **atan** functions return the arc tangent in the range [-π/2, π/2] radians.

Parameters
x Argument.

Description
The **atan** functions compute the arc tangent of *x*. The function returns the following values related to special numbers:
— $\mathbf{atan}(\pm\infty)$ returns $\pm\pi/2$.
— $\mathbf{atan}(\pm 0.0)$ returns ± 0.0.
— **atan**(NaN) returns NaN.
— $\mathbf{atan}(-x)$ returns -**atan**(x), for $x > 0$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("atan(x) = %f\n", atan(x));
}
```

Output

```
atan(x) = 0.291457
```

See Also
tan(), **atanh**(), **atanh**().

atan2

Synopsis
#include <math.h>
type atan2(**type** *y*, **type** *x*);
double atan2(**double** *y*, **double** *x*);
float atanf(**float** *y*, **float** *x*);
long double atanl(**long double** *y*, **long double** *x*);

Purpose
Calculate arc tangent.

Return Value
The **atan2** functions return the arc tangent of **y/x**, in the range [-π, π] radians.

Parameters
x Argument.

y Argument.

Description
The **atan2** functions compute the principal value of the arc tangent of *y/x*, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero. The function returns the following values related to special numbers:
— **atan2**($\pm 0, x$) returns ± 0, for $x > 0$.
— **atan2**($\pm 0, +0$) returns ± 0.
— **atan2**($\pm 0, x$) returns $\pm \pi$, for $x < 0$.
— **atan2**($\pm 0, -0$) returns $\pm \pi$.
— **atan2**($y, \pm 0$) returns $\pi/2$ for $y > 0$.
— **atan2**($y, \pm 0$) returns $\pi/2$ for $y < 0$.
— **atan2**($\pm y, \infty$) returns ± 0 for $yr >< 0$.
— **atan2**($\pm \infty, x$) returns $\pm \pi/2$, for finite x.
— **atan2**($\pm y, -\infty$) returns $\pm \pi$, for $y > 0$.
— **atan2**($\pm \infty, \infty$) returns $\pm \pi/4$.
— **atan2**($\pm \infty, -\infty$) returns $\pm 3\pi/4$.
— **atan2**(NaN, NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    double y = 0.5;
    printf("atan2(x,y) = %f\n", atan2(x,y));
}
```

Output

```
atan2(x,y) = 0.540420
```

See Also
tan(), **atan2h**(), **tan2h**().

atanh

Synopsis
#include <**math.h**>
type atanh(**type** *x*);
double atanh(**double** *x*);
float atanhf(**float** *x*);
long double atanhl(**long double** *x*);

Purpose
Calculate arc hyperbolic tangent.

Return Value
The **atanh** functions return the arc hyperbolic tangent value.

Parameters
x Argument.

Description
The **atanh** functions compute the arc hyperbolic tangent of *x*. A domain error occurs for arguments not in the range [-1, 1]. A range error may occur if the argument equals -1 or +1. The function returns the following values related to special numbers:
— **atanh**(± 0) returns ± 0.
— **atanh**(± 1) returns $\pm\infty$.
— **atanh**(*x*) returns NaN if $|x| > 1$.
— **atanh**($\pm\infty$) returns NaN.
— **atanh**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("atanh(x) = %f\n", atanh(x));
}
```

Output

```
atanh(x) = 0.309520
```

See Also
tan(), **atan**(), **tanh**().

cbrt

Synopsis
#include <math.h>
double cbrt(**double** *x*);
float cbrtf(**float** *x*);
long double cbrtl(**long double** *x*);

Purpose
Calculate cubed root.

Return Value
The **cbrt** functions return the value of the cube root.

Parameters
x Argument.

Description
The **cbrt** functions compute the real cube root of *x*. The function returns the following values related to special numbers:
—**cbrt**($\pm\infty$) returns $\pm\infty$.
—**cbrt**(± 0) returns ± 0.
—**cbrt**(NaN) returns NaN.

Example

```
/* a sample program that solves the cube root */
#include <stdio.h>
#include <math.h>

int main() {
    double x= 8.0;
    printf( "the cube root of 8 is %f\n",cbrt(x));
}
```

Output

```
the cube root of 8 is 2.000000
```

See Also

ceil

Synopsis
#include <math.h>
type ceil(**type** *x*);
double ceil(**double** *x*);
float ceilbf(**float** *x*);
long double ceilbl(**long double** *x*);

Purpose
Calculate smallest integer.

Return Value
The **ceil** functions return the smallest integer value not less than *x*, expressed as a floating-point number. The function returns the following values related to special numbers:
— **ceil**($\pm\infty$) returns $\pm\infty$.
— **ceil**(± 0) returns ± 0.
— **ceil**(NaN) returns NaN.

Parameters
x Argument.

Description
The **ceil** functions compute the smallest integer value not less than *x*.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("ceil(x) = %f\n", ceil(x));
}
```

Output

```
ceil(x) = 1.000000
```

See Also

copysign

Synopsis
#include <math.h>
double copysign(**double** *x*, **double** *y*);
float copysignf(**float** *x*, **float** *y*);
long double copysignl(**long double** *x*, **long double** *y*);

Purpose
Switch signs of values.

Return Value
The **copysign** functions return a value with magnitude of *x* and the sign of *y*.

Parameters
x Argument.

y Argument.

Description
The **copysign** functions produce a value with the magnitude of *x* and the sign of *y*. They produce a NaN (with the sign of *y*) if *x* is a NaN. On implementations that represent a signed zero, but do not treat negative zero consistently in arithmetic operations, the **copysign** functions regard the sign of zero as positive.

Example

```
/* a sample program that changes the first number's
sign to the second number's */
#include <stdio.h>
#include <math.h>

int main() {
    double x,y,z;

    x = -1.563;
    y = 3.7664;
    z = copysign(x,y);
    printf("before copysign x = %f,  y = %f \n",x,y);
    printf("after copysign x = %f,  y = %f \n",x,y);
}
```

Output

```
before copysign x = -1.563000,  y = 3.766400
after copysign x = -1.563000,  y = 3.766400
```

See Also

cos

Synopsis
#include <**math.h**>
type cos(**type** *x*);
double cos(**double** *x*);
float cosf(**float** *x*);
long double cosl(**long double** *x*);

Purpose
Calculate cosine.

Return Value
The **cos** functions return the cosine value.

Parameters
x Argument.

Description
The **cos** functions compute the cosine of *x* (measured in radians). The function returns following values related to special numbers:
— **cos**(± 0) returns 1.
— **cos**($\pm \infty$) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("cos(x) = %f\n", cos(x));
}
```

Output

```
cos(x) = 0.955336
```

See Also
acos(), **acosh**(), **cosh**().

cosh

Synopsis
#include <math.h>
type cosh(**type** *x*);
double cosh(**double** *x*);
float coshf(**float** *x*);
long double coshl(**long double** *x*);

Purpose
Calculate hyperbolic cosine.

Return Value
The **cosh** functions return the hyperbolic cosine value.

Parameters
x Argument.

Description
The **cosh** functions compute the hyperbolic cosine of *x*. A range error occurs if the magnitude of *x* is too large. The function returns following values related to special numbers:
— cosh(± 0) returns 1.
— cosh($\pm\infty$) returns $\pm\infty$.
— cosh(Nan) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
   double x = 0.3;
   printf("cosh(x) = %f\n", cosh(x));
}
```

Output

```
cosh(x) = 1.045339
```

See Also
cos(), **acosh**(), **acos**().

erf

Synopsis
#include <math.h>
double erf(**double** *x*);
float erff(**float** *x*);
long double erfl(**long double** *x*);

Purpose
Calculate error function.

Return Value
The **erf** functions return the error function value.

Parameters
x Argument.

Description
The **erf** functions compute the error function of *x*: $\frac{2}{\sqrt{\pi}} \int_0^{\mathbf{x}} e^{-t^2} dt$. The function returns following values related to special numbers:
— **erf**(± 0) returns ± 0.
— **erf**($\pm \infty$) returns ± 1.

Example

```
/* a sample program that returns the error fuction of x */
#include <stdio.h>
#include <math.h>

int main() {
    double x,y;

    x = 1;
    y = erf(x);
    printf("the error fuction of x= %f is %f\n",x,y);
    y = erfc(x);
    printf("the complementary value of x= %f is %f\n",x,y);
}
```

Output

```
the error fuction of x= 1.000000 is 0.842701
the complementary value of x= 1.000000 is 0.157299
```

See Also

erfc

Synopsis
#include <math.h>
double erfc(**double** *x*);
float erfcf(**float** *x*);
long double erfcl(**long double** *x*);

Purpose
Calculate complementary error function.

Return Value
The **erfc** functions return the complementary error function value.

Parameters
x Argument.

Description
The **erfc** functions compute the complementary error function of *x*: $\frac{2}{\sqrt{\pi}} \int_{\mathbf{x}}^{\infty} e^{-t^2}\, dt$. A range error occurs if *x* is too large. The function returns following values related to special numbers:
— **erfc**($+\infty$) returns ± 0.
— **erfc**($-\infty$) returns ± 2.

Example
See **erf**().

See Also

exp

Synopsis
#include <**math.h**>
type exp(**type** *x*);
double exp(**double** *x*);
float expf(**float** *x*);
long double expl(**long double** *x*);

Purpose
Calculate base-*e* exponential.

Return Value
The **exp** functions return the exponential value.

Parameters
x Argument.

Description
The **exp** functions compute the base-*e* exponential of *x*: e^x. A range error occurs if the magnitude of *x* is too large. The function returns following values related to special numbers:
— **exp**(± 0) returns 1.
— **exp**($+\infty$) returns $+\infty$.
— **exp**($-\infty$) returns $+0$.
— **exp**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("exp(x) = %f\n", exp(x));
}
```

Output

```
exp(x) = 1.349859
```

See Also
exp2(), **expm1**(), **frexp**().

exp2

Synopsis
#include <**math.h**>
double exp2(**double** *x*);
float exp2f(**float** *x*);
long double exp2l(**long double** *x*);

Purpose
Calculate base-2 exponential.

Return Value
The **exp2** functions return the base-2 exponential value.

Parameters
x Argument.

Description
The **exp2** functions compute the base-2 exponential of *x*: 2^x. A range error occurs if the magnitude of *x* is too large. The function returns following values related to special numbers:
— **exp2**(± 0) returns 1.
— **exp2**($+\infty$) returns $+\infty$.
— **exp2**($-\infty$) returns $+0$.
— **exp2**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main() {
    double x = 3.0;
    printf("exp2(%f) = %f\n", x, exp2(x));
}
```

Output

```
exp2(3.000000) = 8.000000
```

See Also
exp(), **expm1**(), **frexp**().

expm1

Synopsis
#include <math.h>
double expm1(double *x*);
float expm1f(float *x*);
long double expm1l(long double *x*);

Purpose
Calculate base-*e*-1 exponential.

Return Value
The **expm1** functions return the value of $e^x - 1$.

Parameters
x Argument.

Description
The **expm1** functions compute the base-*e* exponential of the arguement, minus 1: $e^x - 1$. A range error occurs if *x* is too large. The function returns following values related to special numbers:
— **expm1**(± 0) returns 0.
— **expm1**($+\infty$) returns $+\infty$.
— **expm1**($-\infty$) returns -1.
— **expm1**(NaN) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("expm1(x) = %f\n", expm1(x));
}
```

Output

```
expm1(x) = 0.349859
```

See Also
exp(), **exp2**(), **frexp**().

fabs

Synopsis
#include <math.h>
double fabs(**double** *x*);
float fabsf(**float** *x*);
long double fabsl(**long double** *x*);

Purpose
Calculate absolute value.

Return Value
The **fabs** functions return the absolute value of *x*.

Parameters
x Argument.

Description
The **fabs** functions compute the absolute value of a floating-point number *x*.The function returns following values related to special numbers:
— **fabs**(± 0) returns $+0$.
— **fabs**($\pm\infty$) returns $+\infty$.
— **fabs**(NaN) returns NaN.

Example

```
/* a sample program that returns the absolute
value of x. */
#include <stdio.h>
#include <math.h>

int main() {
    float x;

    x = -12.5;
    printf("The absolute of %3.1f is %3.1f \n",x,fabs(x));
}
```

Output

```
The absolute of -12.5 is 12.5
```

See Also

fdim

Synopsis
#include <math.h>
double fdim(**double** *x*, **double** *y*);
float fdimf(**float** *x*, **float** *y*);
long double fdiml(**long double** *x*, **long double** *y*);

Purpose
Calculate difference value.

Return Value
The **fdim** functions return the positive difference value.

Parameters
x Argument.

y Argument.

Description
The **fdim** functions determine the *positive difference* between their arguments:
$x - y$ if $x \geq y$
+0 if $x \leq y$
A range error may occur.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x[2] = {1.0, 3.0};
    double y[2] = {3.0, 1.0};
    printf("fdim(x[0],y[0]) = %f\n", fdim(x[0], y[0]));
    printf("fdim(x[1],y[1]) = %f\n", fdim(x[1], y[1]));
}
```

Output

```
fdim(x[0],y[0]) = 0.000000
fdim(x[1],y[1]) = 2.000000
```

See Also

floor

Synopsis
#include <math.h>
type floor(**type** *x*);
double floor(**double** *x*);
float floorbf(**float** *x*);
long double floorbl(**long double** *x*);

Purpose
Calculate largest integer.

Return Value
The **floor** functions return the largest integer value not greater than *x*, expressed as a floating-point number.

Parameters
x Argument.

Description
The **floor** functions compute the largest integer not greater than *x*. The function returns following values related to special numbers:
— **floor**(*x*) returns x if x is $\pm\infty or \pm 0$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("floor(x) = %f\n", floor(x));
}
```

Output

```
floor(x) = 0.000000
```

See Also

fma

Synopsis
#include <math.h>
double fma(**double** *x*, **double** *y*, **double** *z*);
float fmaf(**float** *x*, **float** *y*, **float** *z*);
long double fmal(**long double** *x*, **long double** *y*, **long double** *z*);

Purpose
Calculate sum plus product.

Return Value
The **fma** functions return the sum *z* plus the product *x* times *y*, rounded as one ternary operation.

Parameters
x Argument.

y Argument.

z Argument.

Description
The **fma** functions compute the sum of *z* plus the product *x* times *y*, rounded as one ternary operation: they compute the sum *z* plus the product *x* times *y* (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value **FLT_ROUNDS**. The function returns following values related to special numbers:
— **fma**(x, y, z) returns NaN if one of x and y is infinite, the other is zero.
— **fma**(x, y, z) returns NaN if one of x and y is an exact infinite and z is also an infinity but with the opposite sign.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double x = 1.0, y = 2.0, z = 3.0;
    printf("fma(%f,%f,%f) = %f\n", x, y, z, fma(x, y, z));
}
```

Output

```
fma(1.000000,2.000000,3.000000) = 5.000000
```

See Also

fmax

Synopsis
#include <math.h>
double fmax(**double** *x*, **double** *y*);
float fmaxf(**float** *x*, **float** *y*);
long double fmaxl(**long double** *x*, **long double** *y*);

Purpose
Calculate maximum numeric value.

Return Value
The **fmax** functions return the maximum numeric value of their arguments.

Parameters
x Argument.

y Argument.

Description
The **fmax** functions determine the maximum numeric value of their arguments. The function returns the following values related to special numbers:
— If just one argument is NaN, the **fmax** returns the other argument. If both arguments are NaN, the function
returns a NaN.

Example

```
/* a sample program that decides which of the two values is a max. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 1.0;
    y = 2.0;
    z =fmax(x, y);
    printf("Value 1: %f\n", x);
    printf("Value 2: %f\n", y);
    printf("%f is the maximum numeric value.\n",z);
}
```

Output

```
Value 1: 1.000000
Value 2: 2.000000
2.000000 is the maximum numeric value.
```

See Also

fmin

Synopsis
#include <math.h>
double fmin(double *x*, **double** *y*);
float fminf(float *x*, **float** *y*);
long double fminl(long double *x*, **long double** *y*);

Purpose
Calculate minimum numeric value.

Return Value
The **fmin** functions return the minimum numeric value of their arguments.

Parameters
x Argument.

y Argument.

Description
The **fmin** functions determine the minimum numeric value of their arguments. The function returns the following values related to special numbers:
— If just one argument is NaN, the **fmin** returns the other argument. If both arguments are NaN, the function returns a NaN.

Example

```
/* a sample program that decides which of the two values is a min. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 1.0;
    y = 2.0;
    z =fmin(x, y);
    printf("Value 1: %f\n", x);
    printf("Value 2: %f\n", y);
    printf("%f is the minimum numeric value.\n",z);
}
```

Output

```
Value 1: 1.000000
Value 2: 2.000000
1.000000 is the minimum numeric value.
```

See Also

fmod

Synopsis
#include <math.h>
type fmod(**type** *x*, **double** *y*);
double fmod(**double** *x*, **double** *y*);
float fmodf(**float** *x*, **float** *y*);
long double fmodl(**long double** *x*, **long double** *y*);

Purpose
Calculate remainder.

Return Value
The **fmod** functions compute the floating-point remainder of *x*/*y*.

Parameters
x Argument.

y Argument.

Description
The **fmod** functions return the value *x*-*ny*, for some integer *n* such that, if *y* is nonzero, the result has the same sign as *x* and magnitude less than the magnitude of *y*. If *y* is zero, whether a domain error occurs or the **fmod** functions return zero is implementation defined. The function returns the following values related to special numbers:
— **fmod**($\pm 0, y$) returns ± 0 if y is not zero.
— **fmod**(x, y) returns a NaN if x is infinite y is zero.
— **fmod**($x, \pm\infty$) returns x if x is not infinite.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    double y = 1.7;
    printf("fmod(x,y) = %f\n", fmod(x,y));
}
```

Output

```
fmod(x,y) = 0.300000
```

See Also

fpclassify

Synopsis
#include <math.h>
int fpclassify(*real-floating x*);

Purpose
Classifies argument.

Return Value
The **fpclassify** macro returns the value of the number classification macro appropriate to the value of its argument.

Parameters
x Argument.

Description
The **fpclassify** macro classifies its arguement value as NaN, infinite, normal, subnormal, or zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

See Also

frexp

Synopsis
#include <math.h>
double frexp(**double value**, **int** **exp*);
float frexpf(**float value**, **int** **exp*);
long double frexpl(**long double value**, **int** **exp*);

Purpose
Converts number into normalized fractions.

Return Value
The **frexp** functions return the value *x*, such that *x* has a magnitude in the interval [1/2, 1] or zero, and *value* equal $x \times 2^{*\mathbf{exp}}$. If *value* is zero, both parts of the result are zero.

Parameters
value Argument.

exp Argument.

Description
The **frexp** functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the **int** object pointed to by *exp*. The function returns following values related to special numbers:
— **frexp**($\pm 0, exp$) returns ± 0.
— **frexp**($\pm\infty, exp$) returns $\pm\infty$.
— **frexp**(NaN, *exp*) returns NaN.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    double value = 3.1415926;
    double x;
    int expt;
    x = frexp(value, &expt);
    printf("x = %f\n", x);
    printf("*exp = %d\n", expt);
}
```

Output

```
x = 0.785398
*exp = 2
```

See Also
exp2(), **expm1**(), **exp**().

hypot

Synopsis
#include <math.h>
double hypot(**double** *x*, **double** *y*);
float hypotf(**float** *x*, **float** *y*);
long double hypotl(**long double** *x*, **long double** *y*);

Purpose
Calculate sum of squares.

Return Value
The **hypot** functions return the value of the square root of the sum of the squares.

Parameters
x Argument.

y Argument.

Description
The **hypot** functions compute the square root of the sum of the squares of *x* and *y*, without undue overflow or underflow. A range error may occur. The function returns the following values related to special numbers:
— **hypot**(x, y) returns $\pm\infty$ if x is infinite, even if y is a NaN.
— **hypot**$(x, \pm 0)$ is equivalent to **fabs**(x).

Example

```
/* a sample program that use hypot()*/
#include <stdlib.h>
#include <math.h>

int main() {
    double z,y;
    double x;

    x = 3;
    y = 4;
    z = hypot(x,y);
    printf(" the model of the complex %2.1f+%2.1fi is %2.1f\n",x,y,z);
}
```

Output

```
 the model of the complex 3.0+4.0i is 5.0
```

See Also

ilogb

Synopsis
#include <math.h>
int ilogb(**double** *x*);
int ilogbf(**float** *x*);
int double ilogbl(**long double** *x*);

Purpose
Calculate exponent.

Return Value
The **ilogb** functions return the exponent of *x* as a signed **int** value.

Parameters
x Argument.

Description
The **ilogb** functions extract the exponent of *x* as a signed **int** value. If *x* is zero they compute the value **FP_ILOGB0**; if *x* is infinite they compute the value **INT_MAX**; if *x* is a NaN they compute the value **FP_ILOGBNAN**; otherwise, they are equivalent to calling the corresponding **logb** function and casting the returned value to type **int**. A range error may occur if *x* is 0.

Example

```
/*  The ilogb functions return the signed exponent of x as a signed int value */
#include <stdlib.h>
#include <math.h>

int main() {
    double x = 10.0;
    printf("The result is %d\n",ilogb(x));
}
```

Output

```
The result is 3
```

See Also

isfinite

Synopsis
#include <math.h>
int isfinite(*real-floating x*);

Purpose
Determines if an argument has a finite value.

Return Value
The **isfinite** macro returns a nonzero value if and only if its argument has a finite value.

Parameters
x Argument.

Description
The **isfinite** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
/* a sample program that decides if x is finite. */
#include <math.h>
#include <stdio.h>

int main() {
    double x;

    x = 1.0;
    if(isfinite(x))
        printf("%f is a finite number\n",x);
    else
        printf("%f is not a finite value\n", x);
}
```

Output

```
1.000000 is a finite number
```

See Also

isgreater

Synopsis
#include <math.h>
double isgreater(*real-floating x*, *real-floating y*);

Purpose
Calculate greater value.

Return Value
The **isgreater** macro returns the value for **(***x***)** > **(***y***)**.

Parameters
x Argument.

y Argument.

Description
The **isgreater** macro determines whether its first argument is greater than its second argument. The value of **isgreater**(*x*,*y*) is always equal to **(***x***)**>**(***y***)**; however, unlike **(***x***)**>**(***y***)**, **isgreater(***x***,***y***)** does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
   double x = 2.0, y=1.0;
   printf("isgreater(%f %f)= %d\n", x, y, isgreater(x, y));
}
```

Output

```
isgreater(2.000000 1.000000)= 1
```

See Also

isgreaterequal

Synopsis
#include <math.h>
double isgreaterequal(*real-floating x*, *real-floating y*);

Purpose
Calculate greater or equal value.

Return Value
The **isgreaterequal** macro returns the value of **(***x***)** $\geq$ **(***y***)**.

Parameters
x Argument.

y Argument.

Description
The **isgreaterequal** macro determines whether its first argument is greater than or equal to its second argument. The value of **isgreaterequal(***x***,***y***)** is always equal to **(***x***)** $\geq$ **(***y***)**; however, unlike **(***x***)** $\geq$ **(***y***)**, **isgreaterequal(***x***,***y***)** does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
   double x = 2.0, y=1.0, z = 2.0;
   printf("isgreaterequal(%f %f)= %d\n", x, y, isgreaterequal(x, y));
   printf("isgreaterequal(%f %f)= %d\n", x, z, isgreaterequal(x, z));
}
```

Output

```
isgreaterequal(2.000000 1.000000)= 1
isgreaterequal(2.000000 2.000000)= 1
```

See Also

isinf

Synopsis
#include <math.h>
int isinf(*real-floating x*);

Purpose
Determines if argument has an infinite value.

Return Value
The **isinf** macro returns a nonzero value if and only if its argument has a infinite value.

Parameters
x Argument.

Description
The **isinf** macro determines whether its argument is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
/* a sample program that decides if x is infinite. */
#include <math.h>
#include <stdio.h>

int main() {
    double x;

    x = 1.0;
    if(isinf(x))
        printf("%f is infinite.\n",x);
    else
        printf("%f is not infinite. \n", x);
    x = +INFINITY;
    if(isinf(x))
        printf("%f is infinite.\n",x);
    else
        printf("%f is not infinite. \n", x);
}
```

Output

```
1.000000 is not infinite.
Inf is infinite.
```

See Also

isless

Synopsis
#include <math.h>
double isless(*real-floating x*, *real-floating y*);

Purpose
Calculate smaller value.

Return Value
The **isless** macro returns the value for **(***x***)** < **(***y***)**.

Parameters
x Argument.

y Argument.

Description
The **isless** macro determines whether its first argument is less than its second argument. The value of **isless**(*x*,*y*) is always equal to **(***x***)**<**(***y***)**; however, unlike **(***x***)**<**(***y***)**, **isless(***x***,***y***)** does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
   double x = 2.0, y=1.0;
   printf("isless(%f %f)= %d\n", x, y, isless(x, y));
}
```

Output

```
isless(2.000000 1.000000)= 0
```

See Also

islessequal

Synopsis
#include <math.h>
double islessequal(*real-floating x*, *real-floating y*);

Purpose
Calculate less or equal value.

Return Value
The **islessequal** macro returns the value of **(***x***)** $\leq$ **(***y***)**.

Parameters
x Argument.

y Argument.

Description
The **islessequal** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal(***x***,***y***)** is always equal to **(***x***)** $\leq$ **(***y***)**; however, unlike **(***x***)** $\leq$ **(***y***)**, **islessequal(***x***,***y***)** does not raise the *invalid* exception when *x* and *y* are unordered.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
   double x = 2.0, y=1.0, z = 2.0;
   printf("islessequal(%f %f)= %d\n", x, y, islessequal(x, y));
   printf("islessequal(%f %f)= %d\n", x, z, islessequal(x, z));
}
```

Output

```
islessequal(2.000000 1.000000)= 0
islessequal(2.000000 2.000000)= 1
```

See Also

islessgreater

Synopsis
#include <math.h>
double islessgreater(*real-floating x*, *real-floating y*);

Purpose
Determine which value is greater.

Return Value
The **islessgreater** macro returns the value of **(***x***)** $\leq$ **(***y***)** || **(***x***)** $\geq$ **(***y***)**.

Parameters
x Argument.

y Argument.

Description
The **islessgreater** macro determines whether its first argument is less than or greater than its second argument. The **islessgreater**macro is similar to**(***x***)** $\leq$ **(***y***)** || **(***x***)** $\geq$ **(***y***)**; however, **islessgreater(***x***,***y***)** does not raise the *invalid* exception when *x* and *y* are unordered (nor does it evaluate *x* and *y* twice).

Example

```
#include <stdio.h>
#include <math.h>

int main() {
   double x = 2.0, y=1.0, z = 2.0;
   printf("islessgreater(%f %f)= %d\n", x, y, islessgreater(x, y));
   printf("islessgreater(%f %f)= %d\n", y, z, islessgreater(y, z));
}
```

Output

```
islessgreater(2.000000 1.000000)= 1
islessgreater(1.000000 2.000000)= 1
```

See Also

isnan

Synopsis
#include <math.h>
int isnan(*real-floating x*);

Purpose
Determines if argument has an NaN value.

Return Value
The **isnan** macro returns a nonzero value if and only if its argument has a NaN value.

Parameters
x Argument.

Description
The **isnan** macro determines whether its argument is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
/* a sample program that assigns NaN (not a number) to x
and decide if it is NaN. */
#include <math.h>
#include <stdio.h>

int main() {
    double x;

    x = NaN;
    if(isnan(x)) printf("%f is not a number\n",x);
}
```

Output

```
NaN is not a number
```

See Also

isnormal

Synopsis
#include <math.h>
int isnormal(*real-floating x*);

Purpose
Determines if argument has a normal value.

Return Value
The **isnormal** macro returns a nonzero value if and only if its argument has a normal value.

Parameters
x Argument.

Description
The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    printf("isnormal(3) = %d\n",  isnormal(3));
    printf("isnormal(-3) = %d\n",  isnormal(-3));
    printf("isnormal(0) = %d\n",  isnormal(0));
    printf("isnormal(-0) = %d\n",  isnormal(-0));
    printf("isnormal(Inf) = %d\n",  isnormal(Inf));
    printf("isnormal(-Inf) = %d\n",  isnormal(-Inf));
    printf("isnormal(NaN) = %d\n",  isnormal(NaN));
}
```

Output

```
isnormal(3) = 1
isnormal(-3) = 1
isnormal(0) = 0
isnormal(-0) = 0
isnormal(Inf) = 0
isnormal(-Inf) = 0
isnormal(NaN) = 0
```

See Also

isunordered

Synopsis
#include <math.h>
double isunordered(*real-floating x*, *real-floating y*);

Purpose
Checks for order.

Return Value
The **isunordered** macro determines whether its arguments are unordered.

Parameters
x Argument.

y Argument.

Description
The **isunordered** macro returns 1 if its arguments are unordered and 0 otherwise.

Example
Output
See Also

ldexp

Synopsis
#include <math.h>
double ldexp(**double** *x*, **int** *exp*);
float ldexpf(**float** *x*, **int** *exp*);
long double ldexpl(**long double** *x*, **int** *exp*);

Purpose
Multiply floating-point number by power of 2.

Return Value
The **ldexp** functions return the value of $x \times 2^{\mathbf{exp}}$.

Parameters
x Argument.

exp Exponent.

Description
The **ldexp** functions multiply a floating-point number by an integral power of 2. A range error may occur.

Example
Output

```
ldexp(3.000000, 2) = 12.000000
```

See Also

lgamma

Synopsis
#include <math.h>
double lgamma(**double** *x*);
float lgammaf(**float** *x*);
long double lgammal(**long double** *x*);

Purpose
Calculate absolute logrithmic gamma value.

Return Value
The **lgamma** functions return the gamma function value.

Parameters
x Argument.

Description
The **lgamma** functions compute the natural logarithm of the absolute value of gamma of *x*: $\log_e |\Gamma(x)|$. A range error may occur if the magnitude of *x* is too large or too small.

Example

```
/* The lgamma functions compute the natural  logarithm  of
       the  absolute value of Gamma of x. */
#include <stdio.h>
#include <math.h>

int main() {
    double x = -12.5;

    printf("ln(|Gamma (%2.1f)| is %2.1f\n",x,lgamma(x));
}
```

Output

```
ln(|Gamma (-12.5)| is -20.1
```

See Also

log

Synopsis
#include <math.h>
double log(double *x*);
float logf(float *x*);
long double logl(long double *x*);

Purpose
Calculate base-*e* logarithm.

Return Value
The **log** functions return the base-*e* logarithm value.

Parameters
x Argument.

Description
The **log** functions compute the base-*e* (natural) logarithm of *x*. A domain error occurs if the argument is negative. A range error may occur if the argument is zero. The function returns the following values related to special numbers:
— **log**(± 0) returns $-\infty$.
— **log**(1) returns +0.
— **log**(*x*) returns NaN if $x < 0$.
— **log**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("log(x) = %f\n", log(x));
}
```

Output

```
log(x) = -1.203973
```

See Also

log10

Synopsis
#include <**math.h**>
double log10(**double** *x*);
float loggf(**float** *x*);
long double loggl(**long double** *x*);

Purpose
Calculate base-10 logarithm.

Return Value
The **log10** functions return the base-10 logarithm value.

Parameters
x Argument.

Description
The **log10** functions compute the base-10 (common) logarithm of *x*. A domain error occurs if the argument is negative. A range error may occur if the argument is zero. The function returns the following values related to special numbers:
— **log10**(± 0) returns $-\infty$.
— **log10**(1) returns +0.
— **log10**(*x*) returns NaN if $x < 0$.
— **log10**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("log10(x) = %f\n", log10(x));
}
```

Output

```
log10(x) = -0.522879
```

See Also

log1p

Synopsis
#include <math.h>
double log1p(**double** *x*);
float log1pf(**float** *x*);
long double log1pl(**long double** *x*);

Purpose
Calculate base-*e* logarithm.

Return Value
The **log1p** functions return the value of the base-*e* logarithm of 1 plus the argument.

Parameters
x Argument.

Description
The **log1p** functions compute the base-*e* (natural) logarithm of 1 plus the argument. A domain error occurs if the argument is less than -1. A range error may occur if the argument equals -1. The function returns the following values related to special numbers:
— **log**(± 0) returns ± 0.
— **log**(-1) returns $-\infty$.
— **log**(*x*) returns NaN if $x < -1$.
— **log**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("log(x) = %f\n", log(x));
}
```

Output

```
log(x) = -1.203973
```

See Also

log2

Synopsis
#include <**math.h**>
double log2(**double** *x*);
float log2f(**float** *x*);
long double log2l(**long double** *x*);

Purpose
Calculate base-*e* logtarithm.

Return Value
The **log2** functions return the base-2 logtarithm value.

Parameters
x Argument.

Description
The **log2** functions compute the base-2 logtarithm of *x*. A domain error occurs if the argument is less than zero. A range error may occur if the argument is zero. The function returns the following values related to special numbers:
— **log10**(± 0) returns $-\infty$.
— **log10**(*x*) returns NaN if $x < 0$.
— **log10**($+\infty$) returns $+\infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 2;
    printf("log2(x) = %f\n", log2(x));
}
```

Output

```
log2(x) = 1.000000
```

See Also

logb

Synopsis
#include <math.h>
double logb(**double** *x*);
float logbf(**float** *x*);
long double logbl(**long double** *x*);

Purpose
Returns signed exponent.

Return Value
The **logb** functions return the signed exponent of *x*.

Parameters
x Argument.

Description
The **logb** functions extract the exponent of *x*, as a signed integer value in floating-point format. If *x* is subnormal it is treated as though it were normalized; thus, for positive finite *x*,

$1 \leq x \times \mathbf{FLT\_RADIX}^{-\mathbf{log}(x)} \leq \mathbf{FLT\_RADIX}$.

A domain error may occur if the argument is zero. The function returns the following values related to special numbers:
— **log10**(± 0) returns $-\infty$.
— **log10**($\pm \infty$) returns $+\infty$.

Example

```
/*  The logb functions return the signed exponent of x. */
#include <stdlib.h>
#include <math.h>

int main() {
    double x = 10.0;
    printf("The result is %f\n",logb(x));
}
```

Output

```
The result is 3.000000
```

See Also

lrint

Synopsis
#include <math.h>
long int lrint(**double** *x*);
long int lrintf(**float** *x*);
long int lrintl(**long double** *x*);
long long int llrint(**double** *x*);
long long int llrintf(**float** *x*);
long long int llrintl(**long double** *x*);

Purpose
Calculate rounded integer.

Return Value
The **lrint** and **llrint** functions return the rounded integer value.

Parameters
x Argument.

Description
The **lrint** and the **llrint** functions round their arguement to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Example
Output
See Also

lround

Synopsis
#include <math.h>
long int lround(**double** *x*);
long int lroundf(**float** *x*);
long int lroundl(**long double** *x*);
long long int llround(**double** *x*);
long long int llroundf(**float** *x*);
long long int llroundl(**long double** *x*);

Purpose
Calculate rounded integer.

Return Value
The **lround** and **llround** functions return the rounded integer value.

Parameters
x Argument.

Description
The **lround** and the **llround** functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Example
Output
See Also

modf

Synopsis
#include <math.h>
double modf(**double** *value*, **double** **iptr*);
float modff(**float** *value*, **float** **iptr*);
long double modfl(**long double** *value*, **long double** **iptr*);

Purpose
Return fractional value.

Return Value
The **modf** functions return the value of the signed fractional part of *value*.

Parameters
value Argument.

iptr Pointer.

Description
The **modf** functions break the argument *value* into integral and fractional parts, each of which has the same type and sign as the argument. They store the integral part (in floating-point format) in the object pointed to by *iptr*. The function returns the following values related to special numbers:
— **modf**(*value*, *iptr*) returns a result with the same sign as the argument *value*.
— **modf**($\pm\infty$, *iptr*) returns ± 0 and stores $\pm\infty$ in the object pointed to by *iptr*.
— **modf** of a NaN argument stores a NaN in the object pointed to by *iptr* (and returns a NaN).

Example
Output
See Also

nan

Synopsis
#include <math.h>
double nan(**const char** **tagp*);
float nanf(**const char** **tagp*);
long double nanl(**const char** **tagp*);

Purpose
Return a NaN.

Return Value
The **nan** functions return a quiet NaN, if available, with content indicated through *tagp*. If the implementation does not support quiet NaNs, the functions return zero.

Parameters
tagp Argument.

Description
The call **nan(”***n-char-sequence***”)** is equivalent to **strtod**(`"nan(n-char-sequence)", (char**) NULL`); the call **nan(””)** is equivalent to **strtod**`"nan()", (char**) NULL`. IF *tagp* does not point to an n-char-sequence or an empty string, the call is equivalent to **strtod**`("nan", (char**) NULL)`. Calls to **nanf** and **nanl** are equivalent to the corresponding calls to **strtof** and **strtold**.

Example
Output
See Also

nearbyint

Synopsis
#include <math.h>
double nearbyint(double *x*);
float nearbyintf(float *x*);
long double nearbyintl(long double *x*);

Purpose
Calculate rounded integer.

Return Value
The **nearbyint** functions return the rounded integer value.

Parameters
x Argument.

Description
The **nearbyint** functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the *inexact* exception.

Example
Output
See Also

nextafter

Synopsis
#include <math.h>
double nextafter(**double** *x*, **double** *y*);
float nextafterf(**float** *x*, **float** *y*);
long double nextafterl(**long double** *x*, **long double** *y*);

Purpose
Returns next value.

Return Value
The **nextafter** functions return the next representable value in the specified format after *x* in the direction of *y*.

Parameters
x Argument.

y Argument.

Description
The **nextafter** functions determine the next representable value, in the type of the function, after *x* in the direction of *y*, where *x* and *y* are first converted to the type of the function. The **nextafter** functions return *y* if *x* equals *y*. A range error may occur if the magnitude of *x* is the largest finite value representable in the type, or the result is infinite or not representable in the type.

Example

```
/* a sample program that computes the next representable value */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 5.0;
    y = 10.0;
    z = nextafter(x, y);
    printf("%f is the next representable value.\n",z);
}
```

Output

```
5.000000 is the next representable value.
```

See Also

nexttoward

Synopsis
#include <math.h>
double nexttoward(**double** *x*, **long double** *y*);
float nexttowardf(**float** *x*, **long double** *y*);
long double nexttowardl(**long double** *x*, **long double** *y*);

Purpose
Return fractional value.

Return Value

Parameters
x Argument.

y Argument.

Description
The **nexttoward** functions are equivalent to the **nextafter** functions except that the second parameter has type **long double**.

Example
Output
See Also

pow

Synopsis
#include <math.h>
type pow(**type** *x*, **type** *y*);
double pow(**double** *x*, **double** *y*);
float powf(**float** *x*, **float** *y*);
long double powl(**long double** *x*, **long double** *y*);

Purpose
Calculate value to a power.

Return Value
The **pow** functions return the value of *x* raised to the power *y*.

Parameters
x Argument.

y Argument.

Description
The **pow** functions compute *x* raised to the power *y*. A domain error occurs if *x* is negative and *y* is finite and not an integer value. A domain error occurs if the result cannot be represented when *x* is zero and *y* is less than or equal to zero. The function returns the following values related to special numbers:
— **pow**$(x, \pm 0)$ returns 1 for any x, even a NaN.
— **pow**$(x, +\infty)$ returns $+\infty$ for $|x| > 1$.
— **pow**$(x, +\infty)$ returns +0 for $|x| < 1$.
— **pow**$(x, -\infty)$ returns +0 for $|x| > 1$.
— **pow**$(x, -\infty)$ returns $+\infty$ for $|x| < 1$.
— **pow**$(+\infty, y)$ returns $+\infty$ for $|y| > 0$.
— **pow**$(+\infty, y)$ returns +0 for $|y| < 0$.
— **pow**$(-\infty, y)$ returns $-\infty$ for y an odd integer > 0.
— **pow**$(-\infty, y)$ returns $+\infty$ for $y > 0$ and not an odd integer.
— **pow**$(-\infty, y)$ returns -0 for y an odd integer < 0.
— **pow**$(-\infty, y)$ returns +0 for $y < 0$ and not an odd integer.
— **pow**$(\pm 1, \pm\infty)$ returns a NaN.
— **pow**(x, y) returns a NaN for finite $x < 0$ and finite non-integer y.
— **pow**$(\pm 0, y)$ returns $\pm\infty$ for y an odd integer< 0.
— **pow**$(\pm 0, y)$ returns $+\infty$ for $y < 0$ and not an odd integer.
— **pow**$(\pm 0, y)$ returns $\pm$ 0 for y an odd integer > 0.
— **pow**$(\pm 0, y)$ returns $+0$ for $y > 0$ and not an odd integer.

Example
Output

```
pow(2.000000 3.000000) = 8.000000
```

See Also

remainder

Synopsis
#include <math.h>
double remainder(**double** *x*, **double** *y*);
float remainderf(**float** *x*, **float** *y*);
long double remainderl(**long double** *x*, **long double** *y*);

Purpose
Calculate remainder.

Return Value
The **remainder** functions return the value of *x* REM *y*.

Parameters
x Argument.

y Argument.

Description
The **remainder** functions compute the remainder *x* REM *y*.

Example

```
/* The function returns the value of r = x REM y.  r=x-ny, where n is the
   integer nearest the exact value  of  x/y. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, z;

    x = 5.0;
    y = 2.0;
    z =remainder(x, y);
    printf("x =  %f\n", x);
    printf("y = %f\n", y);
    printf("Remainder = %f.\n",z);
}
```

Output

```
x =  5.000000
y = 2.000000
Remainder = 1.000000.
```

See Also

remquo

Synopsis
#include <math.h>
double remquo(**double** *x*, **double** *y*, **int***quo*);
float remquof(**float** *x*, **float** *y*, **int***quo*);
long double remquol(**long double** *x*, **long double** *y*, **int***quo*);

Purpose
Return fractional value.

Return Value
The **remquo** functions return the value of *x* REM *y*.

Parameters
x Argument.

y Argument.

quo Pointer.

Description
The **remquo** functions compute the same remainder as the **remainder** functions. In the object pointed to by *quo* they store a value whose sign is the sign of *x*/*y* and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of *x*/*y*, where *n* is an implementation-defined integer greater than or equal to 3.

Example
Output
See Also

rint

Synopsis
#include <math.h>
double rint(**double** *x*);
float rintf(**float** *x*);
long double rintl(**long double** *x*);

Purpose
Calculate rounded integer.

Return Value
The **rint** functions return the rounded integer value.

Parameters
x Argument.

Description
The **rint** functions differ from the **nearbyint** functions only in that the **rint** functions may raise the inexact exception if the result differs in value from the argument. The function returns the following values related to special numbers:
— **rint**(± 0) returns ± 0 (for all rounding directions).
— **rint**($\pm \infty$) returns $\pm \infty$ (for all rounding directions).

Example

```
/* a sample program that rounds a double number. */
#include <stdlib.h>
#include <math.h>

int main() {
    double x = 213.645;
    printf("%3.3f is rounded to %f \n",x,rint(x));
}
```

Output

```
213.645 is rounded to 214.000000
```

See Also

round

Synopsis
#include <math.h>
double round(**double** *x*);
float roundf(**float** *x*);
long double roundl(**long double** *x*);

Purpose
Calculate rounded integer.

Return Value
The **round** functions return the rounded integer value.

Parameters
x Argument.

Description
The **round** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Example
Output
See Also

scalbn and scalbln

Synopsis
#include <math.h>
double scalbn(**double** *x*, **int** *n*);
float scalbnf(**float** *x*, **int** *n*);
long double scalbnl(**long double** *x*, **int** *n*);
double scalbln(**double** *x*, **int** *n*);
float scalblnf(**float** *x*, **int** *n*);
long double scalblnl(**long double** *x*, **int** *n*);

Purpose
Return the value of $x \times \mathbf{FLT\_RADIX}^n$.

Return Value
The **scalbn** and **scalbln** functions return the value of $x \times \mathbf{FLT\_RADIX}^n$.

Parameters
x Argument.

n Integer.

Description
The **scalbn** and **scalbln** functions compute $x \times \mathbf{FLT\_RADIX}^n$ efficiently, not normally, by computing $\mathbf{FLT\_RADIX}^n$ explicitly. The function returns the following values related to special numbers:
— **scalbn**(x, n) returns x if x is infinite or zero.
— **scalbn**$(x, 0)$ returns x.

Example

```
/* a sample program that decides which of the two values is a max. */
#include <math.h>
#include <stdio.h>

int main() {
    double x, z;
    int y;
    x = 5.0;
    y = 2;
    z =scalbn(x, y);
    printf("Value 1: %f\n", x);
    printf("Value 2: %d\n", y);
    printf("%f is the maximum numeric value.\n",z);
}
```

Output

```
Value 1: 5.000000
Value 2: 2
20.000000 is the maximum numeric value.
```

See Also

signbit

Synopsis
#include <math.h>
int signbit(*real-floating x*);

Purpose
Determines if argument has a negative value.

Return Value
The **signbit** macro returns a nonzero value if and only if its argument has a negative value.

Parameters
x Argument.

Description
The **signbit** macro determines whether its argument value is negative.

Example

```
#include <stdio.h>
#include <math.h>

int main() {
    printf("signbit(3) = %d\n",  signbit(3));
    printf("signbit(-3) = %d\n",  signbit(-3));
}
```

Output
See Also

sin

Synopsis
#include <math.h>
type sin(**type** *x*);
double sin(**double** *x*);
float sinf(**float** *x*);
long double sinl(**long double** *x*);

Purpose
Calculate the sine.

Return Value
The **sin** functions return the sine value.

Parameters
x Argument.

Description
The **sin** functions compute the sine of *x* (measured in radians). The function returns the following values related to special numbers:
— **sin**(± 0) returns ± 0.
— **sin**($\pm \infty$) returns a NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("sin(x) = %f\n", sin(x));
}
```

Output

```
sin(x) = 0.295520
```

See Also
asin(), **asinh**(), **sinh**().

sinh

Synopsis
#include <math.h>
type sinh(**type** *x*);
double sinh(**double** *x*);
float sinhf(**float** *x*);
long double sinhl(**long double** *x*);

Purpose
Calculate the hyperbolic sine.

Return Value
The **sinh** functions return the hyperbolic sine value.

Parameters
x Argument.

Description
The **sinh** functions compute the hyperbolic sine of *x*. A range error occurs if the magnitude of *x* is too large. The function returns the following values related to special numbers:
— **sinh**(± 0) returns ± 0.
— **sinh**($\pm \infty$) returns $\pm \infty$.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("sinh(x) = %f\n", sinh(x));
}
```

Output

```
sinh(x) = 0.304520
```

See Also
sin(), **asinh**(), **asin**().

sqrt

Synopsis
#include <math.h>
double sqrt(**double** *x*);
float sqrtf(**float** *x*);
long double sqrtl(**long double** *x*);

Purpose
Calculate square root.

Return Value
The **sqrt** functions return the value of the square root.

Parameters
x Argument.

Description
The **sqrt** functions compute the non-negative square root of *x*. A domain error occurs if the argument is less than zero.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("sqrt(x) = %f\n", sqrt(x));
}
```

Output

```
sqrt(x) = 0.547723
```

See Also

tan

Synopsis
#include <**math.h**>
type tan(**type** *x*);
double tan(**double** *x*);
float tanf(**float** *x*);
long double tanl(**long double** *x*);

Purpose
Calculate the tangent.

Return Value
The **tan** functions return the tangent value.

Parameters
x Argument.

Description
The **tan** functions compute the tangent of *x* (measured in radians). The function returns the following values related to special numbers:
— **tan**(± 0) returns ± 0.
— **tan**($\pm \infty$) returns NaN.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("tan(x) = %f\n", tan(x));
}
```

Output

```
tan(x) = 0.309336
```

See Also
atan(), **atanh**(), **tanh**().

tanh

Synopsis
#include <math.h>
type tanh(**type** *x*);
double tanh(**double** *x*);
float atanhf(**float** *x*);
long double atanhl(**long double** *x*);

Purpose
Calculate the hyperbolic tangent.

Return Value
The **tanh** functions return the hyperboilc tangent value.

Parameters
x Argument.

Description
The **tanh** functions compute the hyperboilc tangent of *x*. The function returns the following values related to special numbers:
— **tanh**(± 0) returns ± 0.
— **tanh**($\pm \infty$) returns ± 1.

Example

```
#include <math.h>
#include <stdio.h>

int main () {
    double x = 0.3;
    printf("tanh(x) = %f\n", tanh(x));
}
```

Output

```
tanh(x) = 0.291313
```

See Also
tan(), **atanh**(), **atan**().

tgamma

Synopsis
#include <math.h>
double tgamma(**double** *x*);
float tgammaf(**float** *x*);
long double tgammal(**long double** *x*);

Purpose
Calculate gamma value.

Return Value
The **tgamma** functions return the gamma function value.

Parameters
x Argument.

Description
The **tgamma** functions compute the gamma function of *x*: $\Gamma(x)$. A domain error occurs if *x* is a negative integer of zero. A range error may occur if the magnitude of *x* is too large or too small. The function returns the following values related to special numbers:
— **tgamma**($+\infty$) returns $+\infty$.
— **tgamma**(x) returns $+\infty$ if x is a negative integer or zero.
— **tgamma**($-\infty$) returns a NaN.

Example
Output
See Also

trunc

Synopsis
#include <math.h>
double trunc(**double** *x*);
float truncf(**float** *x*);
long double trunc(**long double** *x*);

Purpose
Calculate truncated integer.

Return Value
The **trunc** functions return the truncated integer value.

Parameters
x Argument.

Description
The **trunc** functions round their argument to the integer value, in floating-point format, nearest to but no larger in magnitude than the argument.

Example
Output
See Also

Chapter 11

Numeric Analysis — <numeric.h>

Header file **numeric.h** contains the definitions of high-level numeric analysis functions, which are available only in Ch Professional Edition. It includes the following header files **math.h, stdarg.h, array.h** and **dlfcn.h**. The following high-level functions are designed for numeric analysis.

Function	Description
balance()	Balances a general real matrix to improve accuracy of computed eigenvalues.
ccompanionmatrix()	Calculate companion matrix with complex number.
cdeterminant()	Calculate the determinant of a complex matrix.
cdiagonal()	Get a vector with diagonals of a complex matrix.
cdiagonalmatrix()	Create diagonal matrix from a complex vector.
cfevalarray()	Complex array function evaluation.
cfunm()	Evaluate general complex matrix function.
charpolycoef()	Calculate the coefficients of characteristic polynomial of a matrix.
choldecomp()	Computes the Cholesky factorization of a symmetric positive definite matrix A.
cinverse()	Calculate the inverse of a complex square matrix.
cmean()	Calculate the mean value of all the elements and mean values of each row of a complex array.
combination()	Calculate the number of combination of *n* things taken *k* at a time.
companionmatrix()	Calculate a companion matrix.
complexsolve()	Solve a complex equation.
condnum()	Calculate condition number of a matrix.
conv()	One-dimensional Discrete Fourier Transform (DFT) based convolution.
conv2()	Two-dimensional Discrete Fourier Transform (DFT) based convolution.
corrcoef()	Correlation coefficients calculation.
correlation2()	Obtain a two-dimensional correlation coefficient.
cpolyeval()	Calculate the value of a complex polynomial at a complex point.
cproduct()	Calculate product of all elements or products of the elements of a complex array of any dimension and the products of of each row of a two-dimesional complex array.
cross()	Calculate vector cross product.

csum()	Calculate the sum of all elements of a complex array of any dimension or sums of elements in each row of a two-dimensional complex array.
ctrace()	Calculate the sum of diagonal elements of a complex matrix.
ctriangularmatrix()	Get the upper or lower triangular part of a complex matrix.
cumprod()	Calculate cumulative product of elements in an array
cumsum()	Calculate cumulative sum of elements in an array
curvefit()	Fit a set of data points *x* and *y* to a linear combination of specified base functions.
covariance()	Covariance matrix.
deconv()	One-dimensional Discrete Fourier Transform (DFT) based deconvolution.
derivative()	Calculate numerical derivative of a function at a given point.
derivatives()	Calculate numerical derivatives of a function at multiple points.
determinant()	Calculate the determinant of a matrix.
diagonal()	Get a vector with diagonals of a matrix.
diagonalmatrix()	Create diagonal matrix of a vector.
difference()	Calculate differences between adjacent elements of array.
dot()	Calculate vector dot product.
eigensystem()	Find eigenvalues and eigenvectors.
expm()	Computes a matrix exponential.
factorial()	Factorial function.
fevalarray()	Array function evaluation.
fft()	N-dimensional Fast Fourier Transform (FFT) calculation.
filter()	Filters the data.
filter2()	Two-dimensional Discrete Fourier Transform based FIR filter.
findvalue()	Obtain indices of nonzero elements of an array.
fliplr()	Flip matrix in left/right direction.
flipud()	Flip matrix in up/down direction.
fminimum()	Find the minimum value of a one-dimensional function and its corresponding position for the minimum value.
fminimums()	Find the minimum position and value of an n-dimensional function.
fsolve()	Find a zero position of a nonlinear system of equations.
funm()	Evaluate general real matrix function.
fzero()	Find a zero position of a nonlinear function with one variable.
gcd()	Obtain the greatest common divisor of the corresponding value of two arrays of integral type.
getnum()	Get a number with default value from the console.
hessdecomp()	Reduces a real general matrix to upper Hessenberg form by an orthogonal/unitary matrix similarity transformation.
histogram()	Calculate and plot histograms.
householdermatrix()	Get the Householder matrix.
identitymatrix()	Create an identity matrix.
ifft()	N-dimensional inverse Fast Fourier Transform (FFT) calculation.
integral1()	Numerical integration of a one-dimensional function.
integral2()	Numerical integration of a two-dimensional function.
integral3()	Numerical integration of a three-dimensional function.
integration2()	Numerical integration of a two-dimensional function.
integration3()	Numerical integration of a three-dimensional function.
interp1()	One-dimensional interpolation.

interp2()	Two-dimensional interpolation.
inverse()	Calculate the inverse of a square matrix.
lcm()	Obtain the least common multiple of the corresponding value of two arrays of integral type.
linsolve()	Solve the linear set of equations by LU decomposition.
linspace()	Generate a linearly spaced array.
llsqcovsolve()	Solve linear system of equations based on linear least-squares with known covariance.
llsqnonnegsolve()	Solve linear system of equations with non-negative values based on linear least-squares method.
llsqsolve()	Least squares solution of sets of linear equations.
logm()	Computes matrix natural logarithm.
logspace()	Generate a logarithmically spaced array.
ludecomp()	LU decomposition of general m-by-n matrix.
maxloc()	Find the location of maximum value of an array.
maxv()	Find the maximum values of the elments of each row of a matrix.
mean()	Calculate the mean value of all the elements and mean values of each row in an array.
median()	Find the median value of all the elements and median values of elements in each row of an array.
minloc()	Find the location of the minimum value of an array.
minv()	Find the minimum values of each row in a two-dimensional array.
norm()	Calculate vector and matrix norms.
nullspace()	Calculate null space of a matrix.
oderungekutta()	Solve ordinary differential equation using Runge-Kutta Method.
odesolve()	Solve ordinary differential equation. This function is obsolete.
orthonormalbase()	Find orthonormal bases of a matrix.
pinverse()	Calculate Moore-Penrose pseudoinverse of a matrix.
polyder()	Take derivative of a polynomial.
polyder2()	Calculate the derivative of product or quotient of two polynomials.
polyeval()	Calculate the value of a polynomial and its derivatives.
polyevalarray()	Polynomial evaluation for a sequence of points.
polyevalm()	Calculate the value of a matrix polynomial.
polyfit()	Fit a set of data points to a polynomial function.
product()	Calculate product of all elements of an array of any dimension or products of the elements of each row of a two-dimensional array.
qrdecomp()	Compute the orthogonal-triangular QR decomposition of a matrix.
qrdelete()	Remove a column in a QR factorized matrix.
qrinsert()	Insert a column in a QR factorized matrix.
rank()	Find the rank of a matrix.
residue()	Partial-fraction expansion or residue computation.
rcondnum()	Matrix reciprocal condition number estimate.
roots()	Find the roots of a polynomial.
rot90()	Rotate matrix 90 degrees.
rsf2csf()	Convert a real Schur form to a complex Schur form.
specialmatrix()	Generate a special matrix.
Cauchy	Generate a Cauchy matrix.
ChebyshevVandermonde	Generate a Vandermonde-like matrix for the Chebyshev polynomials.

Chow	Generate a Chow matrix.
Circul	Generate a Circul matrix.
Clement	Generate a Clement matrix.
DenavitHartenberg	Generate a Denavit-Hartenberg matrix.
DenavitHartenberg2	Generate a Denavit-Hartenberg matrix.
Dramadah	Generate a Dramadah matrix.
Fiedler	Generate a Fiedler matrix.
Frank	Generate a Frank matrix.
Gear	Generate a Gear matrix.
Hadamard	Generate a Hadamard matrix.
Hankel	Generate a Hankel matrix.
Hilbert	Generate a Hilbert matrix.
InverseHilbert	Generate a inverse Hilbert matrix.
Magic	Generate a Magic matrix.
Pascal	Generate a Pascal matrix.
Rosser	Generate a Rosser matrix.
Toeplitz	Generate a Toeplitz matrix.
Vandermonde	Generate a Vandermonde matrix.
Wilkinson	Generate a Wilkinson matrix.
schurdecomp()	Compute Schur decomposition.
sign()	Sign function.
sort()	Sorting and ranking elements in ascending order.
sqrtm()	Computes the square root of a matrix.
std()	Calculate the standard deviation of a data set.
sum()	Calculate the sum of all elements or sums of elements in each row of an array.
svd()	Singular value decomposition.
trace()	Calculate the sum of diagonal elements.
triangularmatrix()	Get the upper or lower triangular part of a matrix.
unwrap()	Unwrap radian phase of each element of input array by changing its absolute jump greater than π to its $2 * \pi$ complement.
urand()	Uniform random-number generator.
xcorr()	Obtain a one-dimensional cross-correlation vector.

balance

Synopsis
#include <numeric.h>
int balance(**array double complex** *a*[&][&], **array double complex** *b*[&][&], ...
/* [**array double complex** *d*[&][&]] */);

Syntax
balance(*a*, *b*); balance(*a*, *b*, *d*);
Purpose
Balances a general matrix to improve accuracy of computed eigenvalues.

Return Value
This function returns 0 on success and negative value on failure.

Parameters
a An $n \times n$ square matrix to be balanced.

l An output square matrix which contains the balanced matrix of matrix *a*.

d An optional output square matrix which contains a diagonal matrix.

Description
This function computes the balance matrix *b* and diagonal matrix *d* from input matrix *a* so that $b = d^{-1} * a * d$. If *a* is symmetric, then $b == a$ and *d* is the identity matrix.
In this function, square matrix *a* could be any supported arithmetic data type. Output *b* is the same dimension and data type as input *a*. If the input *a* is a real matrix, the optional output *d* shall only be **double** data type. If the input *a* is a complex matrix, *d* shall be **complex** or **double complex**.

Example1
balance a real general matrix.

```
#include <numeric.h>
int main() {
     int m = 5;

     array double a[5][5] = {     1,  10, 100, 1000, 10000,
                                0.1,   1,  10,  100,  1000,
                               0.01, 0.1,   1,   10,   100,
                              0.001,0.01, 0.1,    1,    10,
                             0.0001,0.001,0.01, 0.1,     1};

     int status;
     array double b[m][m], d[m][m];
     status = balance(a, b, d);
     if (status == 0) {
        printf("a =\n%f\n", a);
        printf("b =\n%f\n", b);
        printf("d =\n%f\n", d);
        printf("inverse(d)*a*d - b =\n%f\n", inverse(d)*a*d-b);
     }
     else
        printf("error: calculation error in balance().\n");
```

```
}
```

Output1

```
a =
1.000000 10.000000 100.000000 1000.000000 10000.000000
0.100000 1.000000 10.000000 100.000000 1000.000000
0.010000 0.100000 1.000000 10.000000 100.000000
0.001000 0.010000 0.100000 1.000000 10.000000
0.000100 0.001000 0.010000 0.100000 1.000000

b =
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000

d =
1000.000000 0.000000 0.000000 0.000000 0.000000
0.000000 100.000000 0.000000 0.000000 0.000000
0.000000 0.000000 10.000000 0.000000 0.000000
0.000000 0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.100000

inverse(d)*a*d - b =
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
-0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
-0.000000 0.000000 0.000000 0.000000 0.000000
```

Example2
Balance a complex general matrix.

```
#include <numeric.h>

int main() {
     int m = 4;

     array double complex a[4][4] = {     0, 0, complex(1,1), 0,
                                          0, 0, 0, complex(1,1),
                                         complex(11,1),complex(10,1), 0, 0,
                                         complex(10,1),complex(11,1), 0, 0};

     int status;
     array double complex b[m][m];
     array double d[m][m];
     status = balance(a, b, d);
     if (status == 0) {
        printf("a =\n%4.2f\n", a);
        printf("b =\n%4.2f\n", b);
        printf("d =\n%4.2f\n", d);
        printf("inverse(d)*a*d -b =\n%4.2f\n", inverse(d)*a*d-b);
     }
     else
        printf("error: calculation error in balance().\n");
```

```
}
```

Output2

```
a =
complex(0.00,0.00) complex(0.00,0.00) complex(1.00,1.00) complex(0.00,0.00)

complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(1.00,1.00)

complex(11.00,1.00) complex(10.00,1.00) complex(0.00,0.00) complex(0.00,0.00)

complex(10.00,1.00) complex(11.00,1.00) complex(0.00,0.00) complex(0.00,0.00)


b =
complex(0.00,0.00) complex(0.00,0.00) complex(10.00,10.00) complex(0.00,0.00)

complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(10.00,10.00)

complex(1.10,0.10) complex(1.00,0.10) complex(0.00,0.00) complex(0.00,0.00)

complex(1.00,0.10) complex(1.10,0.10) complex(0.00,0.00) complex(0.00,0.00)


d =
0.10 0.00 0.00 0.00
0.00 0.10 0.00 0.00
0.00 0.00 1.00 0.00
0.00 0.00 0.00 1.00

inverse(d)*a*d -b =
complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)

complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)

complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)

complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00) complex(0.00,0.00)
```

See Also
eigensystem().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

ccompanionmatrix

Synopsis
#include <**numeric.h**>
array double complex ccompanionmatrix(**array double complex** *v*[&])[:][:];

Purpose
Find companion matrix with complex number.

Return Value
This function returns the complex companion matrix.

Parameters
v Input array containing the complex coefficients of a polynomial.

Description
This function returns the corresponding complex companion matrix of the complex array *v* which contains the coefficients of a polynomial. The eigenvalues of companion matrix are roots of the polynomial.

Example

```
#include <numeric.h>
int main() {
    array double complex v[4] = {complex(1,2),3,4,5}; /* (1+i2)x^3 + 3x^2 + 4x + 5  */
    int n = 4;
    array double complex a[n-1][n-1];

    a = ccompanionmatrix(v);
    printf("ccompanionmatrix(a) =\n%f\n", a);
}
```

Output

```
ccompanionmatrix(a) =
complex(-0.600000,1.200000) complex(-0.800000,1.600000) complex(-1.000000,2.000000)

complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)

complex(0.000000,0.000000) complex(1.000000,0.000000) complex(0.000000,0.000000)
```

See Also
companionmatrix(), **roots**(), **eigensystem**(), **polycoef**().

References

cdeterminant

Synopsis
#include <**numeric.h**>
double complex cdeterminant(**array double complex** *a*[&][&]);

Purpose
Calculate the determinant of a complex matrix.

Return Value
This function returns the determinant of the complex matrix *a*.

Parameters
a The input complex matrix.

Description
The function **cdeterminant**() returns the determinant of complex matrix *a*. If the input matrix is not a square matrix, the function will return NaN.

Example
Calculate the determinant of complex matrices with different data type.

```
#include <numeric.h>
int main() {
    array double complex a[2][2] = {complex(1,2), 4,
                                    3, 7};
    /* a2 is an ill-detition matrix  */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};
    /* a3 is singular */
    array double a3[2][2] = {2, 4,
                             4, 8};
    array complex b[2][2] = {2, 4,
                             3, 7};
    array double c[3][3] =  {-1,5,6,
                             3,-6,1,
                             6,8, 9} ; /* n-by-n matrix */
    double complex det;

    det = cdeterminant(a);
    printf("cdeterminant(a) = %g\n", det);
    det = cdeterminant(a2);
    printf("cdeterminant(a2) = %g\n", det);
    det = cdeterminant(a3);
    printf("cdeterminant(a3) = %g\n", det);
    det = cdeterminant(b);
    printf("cdeterminant(b) = %g\n", det);
    det = cdeterminant(c);
    printf("cdeterminant(c) = %g\n", det);
}
```

Output

```
cdeterminant(a) = complex(-5,14)
cdeterminant(a2) = complex(-0.0038,-0)
```

```
cdeterminant(a3) = complex(-0,-0)
cdeterminant(b) = complex(-2,0)
cdeterminant(c) = complex(317,-0)
```

See Also
determinant(), **inverse**(), **diagonal**(), **ludecomp**(), **rcondnum**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

cdiagonal

Synopsis
#include <**numeric.h**>
array double complex cdiagonal(**array double complex** *a*[&][&],... /* [**int** *k*] */)[:];

Syntax
cdiagonal(*a*);
cdiagonal(*a*, *k*);

Purpose
Form a vector with diagonals of a complex matrix.

Return Value
This function returns a column vector formed from the elements of the *k*th diagonal of the complex matrix *a*.

Parameters
a An input complex matrix.

k An input integer indicating which element the vector is formed from.

Description
The function **cdiagonal**() produces a row vector formed from the elements of *k*th diagonal of the complex matrix *a*.

Example1

```
#include <numeric.h>
int main() {
    array double complex a[3][3] = {complex(1,2), 2, 3,
                            4, 5, 6,
                            7, 8, 9};
    int n =3, k = 1;
    array double complex d[n], d2[n-(abs(k))];

    d = cdiagonal(a);
    printf("cdiagonal(a) = %0.2f\n", d);

    d2 = cdiagonal(a, k);
    printf("cdiagonal(a, 1) = %0.2f\n", d2);

    k = -1;
    d2 = cdiagonal(a, k);
    printf("cdiagonal(a, -1) = %0.2f\n", d2);
}
```

Output1

```
cdiagonal(a) = complex(1.00,2.00) complex(5.00,0.00) complex(9.00,0.00)

cdiagonal(a, 1) = complex(2.00,0.00) complex(6.00,0.00)

cdiagonal(a, -1) = complex(4.00,0.00) complex(8.00,0.00)
```

Example2

```
#include <numeric.h>
int main() {
    array double complex a[4][3] =  {complex(1,2), 2, 3,
                                4, 5, 6,
                                7, 8, 9,
                                4, 4, 4};
    int m=4, n = 3, k1 = 1, k2 = -1;
    array double complex d[min(m, n)];
    array double complex d1[min(m, n-k1)];
    array double complex d2[min(m+k2, n)];

    d = cdiagonal(a);
    printf("cdiagonal(a) = %0.2f\n", d);

    d1 = cdiagonal(a, k1);
    printf("cdiagonal(a, 1) = %0.2f\n", d1);

    d2 = cdiagonal(a, k2);
    printf("cdiagonal(a, -1) = %0.2f\n", d2);
}
```

Output2

```
cdiagonal(a) = complex(1.00,2.00) complex(5.00,0.00) complex(9.00,0.00)

cdiagonal(a, 1) = complex(2.00,0.00) complex(6.00,0.00)

cdiagonal(a, -1) = complex(4.00,0.00) complex(8.00,0.00) complex(4.00,0.00)
```

Example3

```
#include <numeric.h>
int main() {
    array double complex a[3][4] =  {complex(1,2), 2, 3, 4,
                                5, 6, 7, 8,
                                9, 4, 4, 4};
    int m=3, n = 4, k1 = 1, k2 = -1;
    array double complex d[min(m, n)];
    array double complex d1[min(m, n-k1)];
    array double complex d2[min(m+k2, n)];

    d = cdiagonal(a);
    printf("cdiagonal(a) = %0.2f\n", d);

    d1 = cdiagonal(a, k1);
    printf("cdiagonal(a, 1) = %0.2f\n", d1);

    d2 = cdiagonal(a, k2);
    printf("cdiagonal(a, -1) = %0.2f\n", d2);
}
```

Output3

```
cdiagonal(a) = complex(1.00,2.00) complex(6.00,0.00) complex(4.00,0.00)

cdiagonal(a, 1) = complex(2.00,0.00) complex(7.00,0.00) complex(4.00,0.00)

cdiagonal(a, -1) = complex(5.00,0.00) complex(4.00,0.00)
```

See Also
diagonal(), **cdiagonalmatrix**().

References

cdiagonalmatrix

Synopsis
#include <numeric.h>
array double complex cdiagonalmatrix(**array double complex** *v*[&],... /* [int k] */)[:][:];

Syntax
cdiagonalmatrix(*v*)
cdiagonalmatrix(*v, k*)

Purpose
Form a complex diagonal matrix.

Return Value
This function returns a complex square matrix with specified diagonal elements.

Parameters
v An input complex vector.

k An input integer indicating specified diagonal elements of the matrix.

Description
The function **cdiagonalmatrix**() returns a complex square matrix of order $n + abs(k)$, with the elements of *v* on the the *k*th diagonal. $k = 0$ represents the main diagonal, $k > 0$ above the main diagonal, and $k < 0$ below the main diagonal.

Example

```
#include <numeric.h>
int main() {
    array double complex v[3] = {1,2,3};
    int n = 3, k;
    array double complex a[n][n];

    a = cdiagonalmatrix(v);
    printf("cdiagonal matrix a =\n%f\n", a);

    k = 0;
    a = cdiagonalmatrix(v,k);
    printf("cdiagonalmatrix(a, 0) =\n%f\n", a);

    k = 1;
    array double a2[n+abs(k)][n+abs(k)];
    a2 = cdiagonalmatrix(v,k);
    printf("cdiagonalmatrix(a2, 1) =\n%f\n", a2);

    k = -1;
    array double a3[n+abs(k)][n+abs(k)];
    a3 = cdiagonalmatrix(v,k);
    printf("cdiagonalmatrix(a3, -1) =\n%f\n", a3);
}
```

Output

```
cdiagonal matrix a =
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)

complex(0.000000,0.000000) complex(2.000000,0.000000) complex(0.000000,0.000000)

complex(0.000000,0.000000) complex(0.000000,0.000000) complex(3.000000,0.000000)


cdiagonalmatrix(a, 0) =
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)

complex(0.000000,0.000000) complex(2.000000,0.000000) complex(0.000000,0.000000)

complex(0.000000,0.000000) complex(0.000000,0.000000) complex(3.000000,0.000000)


cdiagonalmatrix(a2, 1) =
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 2.000000 0.000000
0.000000 0.000000 0.000000 3.000000
0.000000 0.000000 0.000000 0.000000

cdiagonalmatrix(a3, -1) =
0.000000 0.000000 0.000000 0.000000
1.000000 0.000000 0.000000 0.000000
0.000000 2.000000 0.000000 0.000000
0.000000 0.000000 3.000000 0.000000
```

See Also

diagonalmatrix(), **diagonal**().

References

cfevalarray

Synopsis
#include <**numeric.h**>
int cfevalarray(**array double** &*y*, **double complex** (**func*)(**double complex** *x*), **array double** &*x*, ...
/* [**array int** &*mask*, **double complex** *value*] */);

Syntax
cfevalarray(*y, func, x*)
cfevalarray(*y, func, x, mask, value*)

Purpose
Complex array function evaluation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y A **complex** output array which contains calculated function values.

x A **complex** input array at which the function values will be evaluated.

func A function routine given by the user.

mask An input array of integer. An entry with 1 indicates the corresponding function value will be evaluated, and an entry with 0 indicates that its output value is given by argument *value*.

value The default **complex** value which is used to replace the function value when the entry of *mask* is 0.

Description
This function evaluates the value of a user given function corresponding to the input array *x*. The input array can be of any data type and any dimension. The given function prototype shall be **double complex** *func*(**double complex**).

Example
Evaluation of function $f(x) = x^2$ for complex array of different dimensions.

```
#include <numeric.h>
#define N 2
#define M 3

double complex func(double complex x) {
   return x*x;
}

int main() {
  array float complex y2[N][M], x2[N][M]; /* diff data types, diff dim */
  array int mask1[N][M] = {1, 0, 1, 0, 1, 1};

  linspace(x2, 1, N*M);
  x2[1][2]=complex(2,3);
  cfevalarray(y2, func, x2);
```

```
  printf("y2 = %5.2f\n", y2);

  cfevalarray(y2, func, x2, mask1, -1);
  printf("y2 = %5.2f\n", y2);
}
```

Output

```
y2 = complex( 1.00, 0.00) complex( 4.00, 0.00) complex( 9.00, 0.00)
complex(16.00, 0.00) complex(25.00, 0.00) complex(-5.00,12.00)

y2 = complex( 1.00, 0.00) complex(-1.00, 0.00) complex( 9.00, 0.00)
complex(-1.00, 0.00) complex(25.00, 0.00) complex(-5.00,12.00)
```

See Also
fevalarray().

cfunm

Synopsis
#include <**numeric.h**>
int cfunm(**array double complex** *y[&][&]*, **double complex** (**func*)(**double complex**),
array double complex *x[&][&]*);

Syntax
cfunm(*y*, *func*, *x*)

Purpose
Evaluate general complex matrix function.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input square matrix. It contains data to be evaluated.

func A function routine given by the user.

y Output square matrix which contains data of the calculated function values.

Description
This function evaluates the complex matrix version of the function specified by the parameter *func*. In this function, The input matrix *x* shall be **double complex** data type and the specified function prototype shall be **double complex** *func*(**double complex**). The output matrix *y* could be a **real** or **complex** data type as required.

Example
A complex matrix evaluation.

```
#include <numeric.h>
double complex mylog(double complex x) {
   return log(x);
}

int main() {
   array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                          3,complex(4,1),complex(2,5),
                          0,0,0};
   array double complex zy[3][3];

   expm(zy,zx);
   printf("zx = \n%5.1f",zx);
   printf("zy = \n%5.1f",zy);
   cfunm(zx,mylog,zy);
   printf("zx = \n%5.1f",zx);
}
```

Output

```
zx =
complex(  1.0,  1.0) complex(  2.0,  2.0) complex(  0.0,  0.0)
complex(  3.0,  0.0) complex(  4.0,  1.0) complex(  2.0,  5.0)
complex(  0.0,  0.0) complex(  0.0,  0.0) complex(  0.0,  0.0)
zy =
complex(-44.9, 56.8) complex(-87.5, 70.9) complex(-101.5,-18.9)
complex(-12.5,118.7) complex(-57.4,175.5) complex(-155.5, 67.8)
complex(  0.0,  0.0) complex(  0.0,  0.0) complex(  1.0,  0.0)
zx =
complex(  1.0,  1.0) complex(  2.0,  2.0) complex(  0.0,  0.0)
complex(  3.0,  0.0) complex(  4.0,  1.0) complex(  2.0,  5.0)
complex(  0.0,  0.0) complex(  0.0,  0.0) complex(  0.0,  0.0)
```

See Also

cfunm(), **expm**(), **logm**(), **funm**(), **sqrtm**().

References

G. H. Golub, C. F. Van Loan, Matrix Computations Third edition, The Johns Hopkins University Press, 1996

charpolycoef

Synopsis
#include <numeric.h>
int charpolycoef(array double complex *p*[&], **array double complex** *a*[&][&]);

Purpose
Calculates the coefficients of characteristic polynomial of a matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
p An output array which contains the calculated coefficients of the characteristic polynomial of a matrix.

a An input square matrix.

Description
This function calculates the coefficients of the characteristic polynomial of matrix **A** which is defined as $det(x\mathbf{I} - \mathbf{A})$. The coefficients are in the order of descending powers. The polynomial it represents is $p_0x^n + ... + p_{n-1}x + p_n$.

Example
This example calculates the polynomial coefficients for three different matrices *a1*, *a2* and *a3*. *a1* is symmetrical with real eigenvalues, *a2* is non-symmetrical with real eigenvalues, *a3* is non-symmetrical with complex eigenvalues, and *a4* is a complex matrix.

```
#include <numeric.h>
int main() {
     /* eigenvalues of a symmtrical matrix are always real numbers */
     array double a1[3][3] = {0.8, 0.2, 0.1,
                              0.2, 0.7, 0.3,
                              0.1, 0.3, 0.6};
     /* eigenvalues of a non-symmtrical matrix can be either real numbers or
        complex numbers */
     /* this non-symmtrical matrix has real eigenvalues */
     array double a2[3][3] = {0.8, 0.2, 0.1,
                             0.1, 0.7, 0.3,
                             0.1, 0.1, 0.6};
     /* this non-symmtrical matrix has complex eigenvalues */
     array double a3[3][3] = {3, 9, 23,
                              2, 2, 1,
                              -7, 1, -9};
    array double complex a4[2][2] = {complex(1,2), 4,
                                     3, 7};



     array double charpolynomial[4];
     array double complex zcharpolynomial[4];
     array double complex zcharpolynomial2[3];

     charpolycoef(charpolynomial, a1);
     printf("charpolynomial from charpolycoef(a1) =\n%f\n", charpolynomial);
```

```
    charpolycoef(charpolynomial, a2);
    printf("charpolynomial from charpolycoef(a2) =\n%f\n", charpolynomial);

    charpolycoef(charpolynomial, a3);
    printf("charpolynomial from charpolycoef(a3) =\n%f\n", charpolynomial);

    charpolycoef(zcharpolynomial, a3);
    printf("zcharpolynomial from charpolycoef(zcharpolynomial, a3) =\n%f\n",
            zcharpolynomial);

    charpolycoef(zcharpolynomial2, a4);
    printf("zcharpolynomial2 from charpolycoef(zcharpolynomial2, a4) =\n%f\n",
           zcharpolynomial2);
}
```

Output

```
charpolynomial from charpolycoef(a1) =
1.000000 -2.100000 1.320000 -0.245000

charpolynomial from charpolycoef(a2) =
1.000000 -2.100000 1.400000 -0.300000

charpolynomial from charpolycoef(a3) =
1.000000 4.000000 103.000000 -410.000000

zcharpolynomial from charpolycoef(zcharpolynomial, a3) =
complex(1.000000,0.000000) complex(4.000000,0.000000) complex(103.000000,0.000000)
complex(-410.000000,0.000000)

zcharpolynomial2 from charpolycoef(zcharpolynomial2, a4) =
complex(1.000000,0.000000) complex(-8.000000,-2.000000) complex(-5.000000,14.000000)
```

See Also

roots(), **polycoef**().

References

choldecomp

Synopsis
#include <numeric.h>
int choldecomp(**array double complex** *a*[&][&], **array double complex** *l*[&][&], ...
/* [**char** *mode*] */);

Syntax
choldecomp(*a*, *l*); choldecomp(*a*, *l*, *"mode"*);
Purpose
Computes the Cholesky factorization of a symmetric ,positive ,definite matrix A.

Return Value
This function returns 0 on success, a negative value on failure, and positive value i indicates that the leading minor of order i is not positive definite and the factorization could not be completed.

Parameters
a A symmetric positive definite two-dimensional matrix.

l A two-dimensional output matrix which contains the upper or lower triangle of the symmetric matrix *a*.

mode The it mode specifies the calculation of upper or lower triangle matrix.
'L' or 'l'- lower triangle is calculated.
otherwise the upper triangle matrix is calculated. By default, the upper triangle matrix is calculated.

Description
This function computes Cholesky factorization of a symmetric positive definite matrix **A**. Cholesky decomposition factors a symmetric positive definite matrix into two matrices. For a symmetric positive finite matrix **A** of real type, Cholesky decomposition can produce matrix **L** so that

$$\mathbf{A} = \mathbf{L^T L}$$

for upper trangle factorization or

$$\mathbf{A} = \mathbf{L L^T}$$

for lower triangle factorization. where L^T is the transpose of matrix **L**. For a symmetric positive finite matrix **A** of complex type, instead of L^T, the Hermitian L^H of matrix **L** shall be used.
Example1
Cholesky factorization of a real matrix.

```
#include <numeric.h>
int main() {
     int m = 5;
     array double a[5][5] = { 1, 1, 1, 1, 1,
                              1, 2, 3, 4, 5,
                              1, 3, 6,10,15,
                              1, 4, 10,20,35,
                              1, 5, 15,35,70};
     array double a1[5][5] = { -1, 1, 1, 1, 1,
                              1, 2, 3, 4, 5,
                              1, 3, 6,10,15,
                              1, 4, 10,20,35,
```

```
                              1, 5, 15,35,70};
     int status;
     array double l[m][m];

     status = choldecomp(a, l);
     if (status == 0) {
        printf("upper triangle l =\n%f\n", l);
        printf("transpose(l)*l - a =\n%f\n", transpose(l)*l-a);
     }
     else
        printf("error: Matrix must be positive definite.\n"
                "The %d order of matrix is not positive definite.\n",status);

     status = choldecomp(a, l, 'l');
     if (status == 0) {
        printf("lower triangle l =\n%f\n", l);
        printf("l*transpose(l) - a =\n%f\n", l*transpose(l)-a);
     }
     else
        printf("error: Matrix must be positive definite.\n"
                "The %d order of matrix is not positive definite.\n",status);

     status = choldecomp(a1, l);
     if (status == 0) {
        printf("l =\n%f\n", l);
     }
     else {
        printf("a =\n%f", a1);
        printf("error: Matrix must be positive definite.\n"
                "The %d order of matrix is not positive definite.\n",status);
     }
}
```

Output1

```
upper triangle l =
1.000000 1.000000 1.000000 1.000000 1.000000
0.000000 1.000000 2.000000 3.000000 4.000000
0.000000 0.000000 1.000000 3.000000 6.000000
0.000000 0.000000 0.000000 1.000000 4.000000
0.000000 0.000000 0.000000 0.000000 1.000000

transpose(l)*l - a =
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000

lower triangle l =
1.000000 0.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 0.000000 0.000000
1.000000 2.000000 1.000000 0.000000 0.000000
1.000000 3.000000 3.000000 1.000000 0.000000
1.000000 4.000000 6.000000 4.000000 1.000000

l*transpose(l) - a =
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
```

```
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000 0.000000

a =
-1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 2.000000 3.000000 4.000000 5.000000
1.000000 3.000000 6.000000 10.000000 15.000000
1.000000 4.000000 10.000000 20.000000 35.000000
1.000000 5.000000 15.000000 35.000000 70.000000
error: Matrix must be positive definite.
The 1 order of matrix is not positive definite.
```

Example2
Cholesky factorization of a complex matrix.

```
#include <numeric.h>

int main() {
     int m = 3;
     array double complex a[3][3] =
           { complex(2,0), complex(0,-1), complex(0,0),
             complex(0,1), complex(2,0), complex(0,0),
             complex(0,0), complex(0,0), complex(3,0)};

     int status;
     array double complex l[m][m];

     status = choldecomp(a, l);
     if (status == 0) {
        printf("upper triangle l =\n%5.3f\n", l);
        printf("transpose(l)*l - a =\n%5.3f\n", conj(transpose(l))*l-a);
     }
     else
        printf("error: Matrix must be positive definite.\n"
                "The %d order of matrix is not positive definite.\n",status);

     status = choldecomp(a, l, 'l');
     if (status == 0) {
        printf("lower triangle l =\n%5.3f\n", l);
        printf("l*transpose(l) - a =\n%5.3f\n", l*conj(transpose(l))-a);
     }
     else
        printf("error: Matrix must be positive definite.\n"
                "The %d order of matrix is not positive definite.\n",status);

}
```

Output2

```
upper triangle l =
complex(1.414,0.000) complex(0.000,-0.707) complex(0.000,0.000)
complex(0.000,0.000) complex(1.225,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(1.732,0.000)

transpose(l)*l - a =
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(-0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(-0.000,0.000)
```

```
lower triangle l =
complex(1.414,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.707) complex(1.225,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(1.732,0.000)

l*transpose(l) - a =
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(-0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(-0.000,0.000)
```

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

cinverse

Synopsis
#include <**numeric.h**>
array double complex cinverse(**array double complex** *a*[&][&], ... /* [**int** **status*] */)[:][:];

Syntax
cinverse(*a*)
cinverse(*a, status*)

Purpose
Calculate the inverse of a complex square matrix.

Return Value
This function returns the inverse matrix.

Parameters
a Input complex square matrix.

status Output integer indicating the status of calculation.

Description
This function calculates the inverse matrix of a complex square matrix. If calculation is successful,$status = 0$, otherwise $status \neq 0$.

Example

```
#include <numeric.h>
int main() {
    array double complex a[2][2] = {complex(2,-3), 4,
                                                    3, 7}; // n-by-n matrix
    array double complex inv[2][2];
    int status;

    inv = cinverse(a);
    printf("cinverse(a) =\n%f\n", inv);
    printf("a =\n%f\n", a);
    printf("cinverse(a)*a =\n%f\n", inv*a);
    printf("a*cinverse(a) =\n%f\n", a*inv);

    inv = cinverse(a, &status);
    if(status == 0)
        printf("cinverse(a, &status) = %f\n", inv);
    else
        printf("error: numerical error in cinverse()\n");
}
```

Output

```
cinverse(a) =
complex(0.031461,0.330337) complex(-0.017978,-0.188764)
complex(-0.013483,-0.141573) complex(0.150562,0.080899)
```

```
a =
complex(2.000000,-3.000000) complex(4.000000,0.000000)
complex(3.000000,0.000000) complex(7.000000,0.000000)

cinverse(a)*a =
complex(1.000000,-0.000000) complex(0.000000,0.000000)
complex(-0.000000,-0.000000) complex(1.000000,0.000000)

a*cinverse(a) =
complex(1.000000,-0.000000) complex(0.000000,0.000000)
complex(0.000000,-0.000000) complex(1.000000,0.000000)

cinverse(a, &status) = complex(0.031461,0.330337) complex(-0.017978,-0.188764)
complex(-0.013483,-0.141573) complex(0.150562,0.080899)
```

See Also
inverse(), **ludecomp**(), **linsolve**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadephia, 1992.

cmean

Synopsis
#include <**numeric.h**>
double complex cmean(**array double complex** *&a*, ... /*[**array double complex** *v[&]*] */);

Syntax
cmean(*a*)
cmean(*a*, *v*)

Purpose
Calculate the mean value of all elements of an array of complex type and mean values of the elements of each row of a two-dimensional array of complex type.

Return Value
This function returns the mean value of all the elements of complex type.

Parameters
a An input array of any dimension.

v An output array which contains the mean values of each row of a two-dimensional array of complex type. *v* shall be an array of double complex type.

Description
This function calculates the complex mean value of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate complex mean values of each row. The complex mean values of each row are passed out by argument *v*.

Example
Calculate the mean values of all elements of arrays of complex type with different dimensions. The mean values of each row are passed by argument *v*.

```
#include <numeric.h>
#define N 2
#define M 3

int main() {
    complex a[3] = {1,2,complex(3.435,6.35765)};
    double complex c[2][3][5];
    c[0][0][0] = complex(53.327,92.5310);
    c[0][0][1] = 20;
    complex a1[N][M] = {1,complex(3.54,54.43),3,
                        4,5,complex(3.455,466)};
    array double complex b1[3][4] = {1,2,3,complex(4,64.54),
                                     complex(5,6.45),6,7,8,
                                     1,complex(4.35,54.43),3,4};
    array double complex meana1[N], meanb1[3];
    double complex meanval;

    meanval = cmean(a);
    printf("mean(a) = %f\n", meanval);
```

```
    meanval = cmean(c);
    printf("mean(c) = %f\n", meanval);

    /* Note: second argument of cmean() must be double complex data type */

    meanval = cmean(a1, meana1);
    printf("meanval = mean(a1, meana1) = %f\n", meanval);
    printf("mean(a1, meana1) = %f\n", meana1);

    meanval = cmean(b1, meanb1);
    printf("meanval = mean(b1, meanb1) = %f\n", meanval);
    printf("mean(b1, meanb1) = %0.2f\n", meanb1);
}
```

Output

```
mean(a) = complex(2.145000,2.119217)
mean(c) = complex(2.444233,3.084367)
meanval = mean(a1, meana1) = complex(3.332500,86.738333)
mean(a1, meana1) = complex(2.513333,18.143333) complex(4.151667,155.333333)

meanval = mean(b1, meanb1) = complex(4.029167,10.451667)
mean(b1, meanb1) = complex(2.50,16.14) complex(6.50,1.61) complex(3.09,13.61)
```

See Also

mean().

combination

Synopsis
#include <numeric.h>
unsigned long long combination(**unsigned int** *n*, **unsigned int** *k*);

Purpose
Calculate the number of combinations of *n* different things taken *k* at a time without repetitions.

Return Value
This function returns the number of combinations.

Parameters
n The number of different things.

k The number of items taken from *n* different things.

Description
This function calculates the number of combination of *n* different things taken *k* at a time without repetitions. The number of combination is the number of sets that can be made up from *n* things, each set containing *k* different things and no two sets containing exactly the same *k* things.

Algorithm
The number of combination of *n* different things taken *k* at a time without repetitions is defined as

$$C_k^n = \frac{n!}{(n-k)!k!}$$

Example

```
#include <numeric.h>
int main() {
    unsigned long long f;

    f = combination(3, 2);
    printf("combination(3, 2) = %llu\n", combination(3, 2));
}
```

Output

```
combination(3, 2) = 3
```

See Also
factorial().

References

companionmatrix

Synopsis
#include <**numeric.h**>
array double companionmatrix(**array double** *v*[&])[:][:];

Purpose
Find a companion matrix.

Return Value
This function returns a companion matrix.

Parameters
v Input array containing the coefficients of a polynomial.

Description
This function returns the corresponding companion matrix of array *v* which contains the coefficients of a polynomial. The eigenvalues of companion matrix are roots of the polynomial.

Example

```
#include <numeric.h>
int main() {
    array double v[4] = {2,3,4,5}; /* 2x^3 + 3x^2 + 4x + 5 */
    int n = 4;
    array double a[n-1][n-1];

    a = companionmatrix(v);
    printf("companionmatrix(a) =\n%f\n", a);
}
```

Output

```
companionmatrix(a) =
-1.500000 -2.000000 -2.500000
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
```

See Also
ccompanionmatrix(), **eigensystem**(), **polycoef**().

References

complexsolve

Synopsis
#include <numeric.h>
int complexsolve(**int** *n1*, **int** *n2*, **double** *phi_or_r1*, **double** *phi_or_r2*, **double complex** *z3*,
double &*xx1*, **double** &*xx2*, **double** &*xx3*, **double** &*xx4*);

Purpose
Solve a complex equation in polar format.

Return Value
This function returns the number of solutions with 0, 1, or 2.

Parameters
n1 The first position of the two unknowns on the left hand side of equation(1). It's value can be 1, 2, 3, or 4.

n2 The second position of the two unknowns on the left hand side of equation(1). It's value can be 1, 2, 3, or 4.

phi_or_r1 The value of the first known on the left hand side.

phi_or_r2 The value of the second known on the left hand side.

z3 Complex number on the right hand side.

xx1 First unknown on the left hand side.

xx2 Second unknown on the left hand side.

xx3 First unknown on the left hand side for the second solution, if there are more than one set of solution.

xx4 Second unknown on the left hand side for the second solution, if there are more than one set of solution.

Description
This function is used to solve a complex equation in polar format. The equation is in the form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = x_3 + iy_3 \tag{11.1}$$

Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns out of four parameters R_1, ϕ_1, R_2, and ϕ_2 can be solved in this equation. The parameters R_1, ϕ_1, R_2, and ϕ_2 are in positions 1, 2, 3, and 4, respectively.

Algorithm
Two unknowns can be solved by decomposing equation (11.1) into real and imaginary parts

$$R_1 \cos\phi_1 + R_2 \cos\phi_2 = x_3 \tag{11.2}$$

$$R_1 \sin\phi_1 + R_2 \sin\phi_2 = y_3 \tag{11.3}$$

Case 1: $n_1 = 1, n_2 = 2$, Solve for R_1 and ϕ_1, given R_2, ϕ_2, R_3, ϕ_3 or R_2, ϕ_2, x_3, y_3. Where, n_1 and n_2 are the first and second positions of two unknowns on the left hand side of equation (11.1), respectively.

From equations (11.2) and equations (11.3), we get

$$R_1 \cos\phi_1 = x_3 - R_2 \cos\phi_2 = a \tag{11.4}$$

$$R_1 \sin\phi_1 = y_3 - R_2 \sin\phi_2 = b \tag{11.5}$$

R_1 and ϕ_1 can be calculated as

$$R_1 = \sqrt{a^2 + b^2} \tag{11.6}$$

$$\phi_1 = \text{atan2}(b, a) \tag{11.7}$$

Case 2: $n_1 = 1, n_2 = 3$, Given ϕ_1 and ϕ_2, R_1 and R_2 are solved.

Multiplying equation (11.1) by $e^{-i\phi_2}$ and $e^{-i\phi_1}$ gives

$$R_1 e^{i(\phi_1 - \phi_2)} + R_2 = R_3 e^{i(\phi_3 - \phi_2)} \tag{11.8}$$

$$R_1 + R_2 e^{i(\phi_2 - \phi_1)} = R_3 e^{i(\phi_3 - \phi_1)} \tag{11.9}$$

Imaginary parts of equation (11.7) and equation (11.8) are

$$R_1 \sin(\phi_1 - \phi_2) = R_3 \sin(\phi_3 - \phi_2) \tag{11.10}$$

$$R_2 \sin(\phi_2 - \phi_1) = R_3 \sin(\phi_3 - \phi_1) \tag{11.11}$$

From equation (11.10) and equation (11.11) we get

$$R_1 = R_3 \frac{\sin(\phi_3 - \phi_2)}{\sin(\phi_1 - \phi_2)} \tag{11.12}$$

$$R_2 = R_3 \frac{\sin(\phi_3 - \phi_1)}{\sin(\phi_2 - \phi_1)} \tag{11.13}$$

Case 3: $n_1 = 1, n_2 = 4$, Given ϕ_1 and R_2, R_1 and ϕ_2 are solved.

From equation (11.2) we have

$$R_1 = \frac{x_3 - R_2 \cos\phi_2}{\cos\phi_1} \tag{11.14}$$

substitute equation (11.14) into equation (11.3) we get

$$(x_3 - R_2 \cos\phi_2) \sin\phi_1 + R_2 \sin\phi_2 \cos\phi_1 = y_3 \cos\phi_1 \tag{11.15}$$

equation (11.15) can be simplified as

$$\sin(\phi_2 - \phi_1) = \frac{y_3 \cos\phi_1 - x_3 \sin\phi_1}{R_2} = a \tag{11.16}$$

then

$$\phi_2 = \phi_1 + \sin^{-1}(a) \phi_2 = \phi_1 + \pi - \sin^{-1}(a) \tag{11.17}$$

If $\cos\phi_1$ is larger than machine epsilon,ε, R_1 can be obatined using equation (11.14). Otherwise R_1 will be obtained using equation (11.3)
if ($|\cos\phi_1)| >$ FLT_EPSILON

$$R_1 = \frac{(x_3 - R_2\cos\phi_2)}{\cos\phi_1} \tag{11.18}$$

else

$$R_1 = \frac{(y_3 - R_2\sin\phi_2)}{\sin\phi_1} \tag{11.19}$$

Case 4: $n_1 = 2, n_2 = 4$, given R_1 and R_2, ϕ_1 and ϕ_2 in equation (11.1) can be solved.

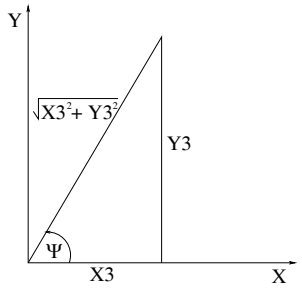
From equation (11.1), we get

$$\cos\phi_1 = \frac{x_3 - R_2\cos\phi_2}{R_1}, \quad \sin\phi_1 = \frac{y_3 - R_2\sin\phi_2}{R_1} \tag{11.20}$$

Substituting these results into the identity equation $\sin^2\phi_1 + \cos^2\phi_1 = 1$ and simplifying the resultant equation, we get

$$y_3\sin\phi_2 + x_3\cos\phi_2 = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2} \tag{11.21}$$

from equation (11.21), we can derive the formulas for ϕ_1 and ϕ_2 as follows:



$$\tan(\psi) = \frac{y_3}{x_3}, \cos(\psi) = \frac{x_3}{\sqrt{x_3^2 + y_3^2}}, \sin(\psi) = \frac{y_3}{\sqrt{x_3^2 + y_3^2}}$$

Equation (11.21) becomes,

$$\frac{y_3}{\sqrt{x_3^2 + y_3^2}}\sin(\phi_2) + \frac{x_3}{\sqrt{x_3^2 + y_3^2}}\cos(\phi_2) = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}} \tag{11.22}$$

$$a = \sin(\psi)\sin(\phi_2) + \cos(\psi)\cos(\phi_2)$$

Let,

$$a = \frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}} \tag{11.23}$$

Becomes,

$$\cos(\phi_2 - \psi) = a \tag{11.24}$$

$$\phi_2 = \psi \pm \cos^{-1}(a) \tag{11.25}$$

$$= \operatorname{atan2}(y_3, x_3) \pm \operatorname{acos}\left(\frac{x_3^2 + y_3^2 + R_2^2 - R_1^2}{2R_2\sqrt{x_3^2 + y_3^2}}\right) \tag{11.26}$$

ϕ_1 can be obtained using equation (11.20)

$$\phi_1 = \operatorname{atan2}(\sin\phi_2, \cos\phi_1) \tag{11.27}$$

or use identity equation,

$$\tan(\frac{\phi}{2}) = \frac{1-\cos(\phi)}{\sin(\phi)} \text{ or } \tan(\frac{\phi}{2}) = \frac{\sin(\phi)}{1+\cos(\phi)} \tag{11.28}$$

$$\phi_1 = 2\tan^{-1}(\frac{1-\cos(\phi)}{\sin(\phi)}) \text{ or } \phi_1 = 2\tan^{-1}(\frac{\sin(\phi)}{1+\cos(\phi)} \tag{11.29}$$

Case 5: $n_1 = 2, n_2 = 3$, given R_1 and ϕ_2, ϕ_1 and R_2 are solved. This case is similar to Case 3.

Case 6: $n_1 = 3, n_2 = 4$, given R_1 and ϕ_1, R_2 and ϕ_2 are solved. This case is similar to case 1.

Example 1
A complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2 \tag{11.30}$$

is solved for the 6 cases described above.

```
#include <numeric.h>
#include <complex.h>

int main() {
    double x1, x2, x3, x4;
    int n1, n2, num;
    double phi_or_r1, phi_or_r2;
    complex z3, error;

    phi_or_r1 = 3;
    phi_or_r2 = 4;
    z3 = complex(1,2);

    /************* test n1 = 1, n2 = 2 ***********/
    n1 = 1,
    n2 = 2;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    error = polar(x1, x2)  + polar(phi_or_r1, phi_or_r2) - z3;
    printf("For n1=1, n2=2, num = %d\n", num);
    printf("For n1=1, n2=2, the output x is %f %f %f %f  \n", x1, x2, x3, x4);
    printf("the residual error is %f \n", error);
```

```
    /************* test n1 = 1, n2 = 3 **********/
    n1 = 1,
    n2 = 3;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=1, n2=3, num = %d\n", num);
    printf("For n1=1, n2=3, the output x is %f %f %f %f  \n", x1, x2, x3, x4);

    /************* test n1 = 1, n2 = 4 **********/
    n1 = 1,
    n2 = 4;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=1, n2=4, num = %d\n", num);
    printf("For n1=1, n2=4, the output x is %f %f %f %f  \n", x1, x2, x3, x4);

    /************* test n1 = 2, n2 = 3 **********/
    n1 = 2,
    n2 = 3;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=2, n2=3, num = %d\n", num);
    printf("For n1=2, n2=3, the output x is %f %f %f %f  \n", x1, x2, x3, x4);

    /************* test n1 = 2, n2 = 4 **********/
    n1 = 2,
    n2 = 4;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=2, n2=4, num = %d\n", num);
    printf("For n1=2, n2=4, the output x is %f %f %f %f  \n", x1, x2, x3, x4);

    /************* test n1 = 3, n2 = 4 **********/
    n1 = 3,
    n2 = 4;
    num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
    printf("For n1=3, n2=4, num = %d\n", num);
    printf("For n1=3, n2=4, the output x is %f %f %f %f  \n", x1, x2, x3, x4);
}
```

Output

Example 2

For a complex equation

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = 1 + i2 \tag{11.31}$$

(1) Given $R_1 = 3$ and $\phi_1 = 4$ radian, find R_2 and ϕ_2
(2) Given $R_1 = 3$ and $R_2 = 4$, find ϕ_1 and ϕ_2

```
#include <numeric.h>
#include <complex.h>

int main () {
  double x1, x2, x3, x4;
  int n1, n2, num;
  double phi_or_r1, phi_or_r2;
  complex z3, error;

  /* problem (1) n1 = 3, n2 = 4 */
  phi_or_r1 = 3;
  phi_or_r2 = 4;
  z3 = complex(1,2);
```

```
  n1 = 3,
  n2 = 4;
  num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
  printf("For n1=3, n2=4, the number of solution is = %d\n", num);
  printf("R2 = %f phi2 = %f\n", x1, x2);
  printf("R2 = %f phi2 = %f\n", x3, x4);

  /* problem (2), n1 = 2, n2 = 4 */
  n1 = 2,
  n2 = 4;
  num = complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, x1, x2, x3, x4);
  printf("For n1=2, n2=4, the number of solution is = %d\n", num);
  printf("phi1 = %f phi2 = %f\n", x1, x2);
  printf("phi1 = %f phi2 = %f\n", x3, x4);
}
```

Output

```
For n1=3, n2=4, the number of solution is = 1
R2 = 5.196488 phi2 = 0.964540
R2 = NaN phi2 = NaN
For n1=2, n2=4, the number of solution is = 2
phi1 = -0.613277 phi2 = 1.942631
phi1 = 2.827574 phi2 = 0.271667
```

See Also
polar().

References

condnum

Synopsis
#include <numeric.h>
double condnum(**array double complex** *a*[&][&]);

Purpose
Calculate the condition number of a matrix.

Return Value
This function returns the condition number.

Parameters
a Input matrix.

Description
The condition number of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and numerical solution of the linear system of equations solution. A value of the condition number near 1 indicates a well-conditioned matrix.

Algorithm
The function **condnum**() uses the the singular value decomposition function **svd**(). By function **svd**(), the matrix a is decomposed as

$$\mathbf{a} = \mathbf{usv}^{\mathbf{t}}$$

The condition number is defined as $\max(s)/\min(s)$.

Example

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                            3, 7};
    /* a2 is an ill-conditioned matrix  */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};
    array float b[2][2] = {2, 4,
                           3, 7};

    array double complex z[2][2] = {2, complex(4, 3),
                                    3, 7};
    /* z2 is an ill-conditioned matrix */
    array double complex z2[2][2] = {2,     complex(4, 3),
                                     2.001, complex(4.0001,3)};
    array complex z3[2][2] = {2, complex(4, 3),
                              3, 7};
    double cond;

    cond = condnum(a);
    printf("condnum(a) = %g\n", cond);
```

```
    cond = condnum(a2);
    printf("condnum(a2) = %g\n", cond);

    cond = condnum(b);
    printf("condnum(b) = %g\n", cond);

    cond = condnum(z);
    printf("condnum(z) = %g\n", cond);
    cond = condnum(z2);
    printf("condnum(z2) = %g\n", cond);
    cond = condnum(z3);
    printf("condnum(z3) = %g\n", cond);
}
```

Output

```
condnum(a) = 38.9743
condnum(a2) = 10527.6
condnum(b) = 38.9743
condnum(z) = 9.32929
condnum(z2) = 11980.8
condnum(z3) = 9.32929
```

See Also

svd(), **norm**(), **rank**(), **rcondnum**().

References

conv

Synopsis
#include <numeric.h>
int conv(array double complex *c*[&], **array double complex** *x*[&], **array double complex** *y*[&]);

Syntax
conv(*c*, *x*, *y*)

Purpose
One-dimensional Discrete Fourier Transform (DFT) based convolution.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
c A one-dimensional array of size $n + m - 1$. It contains the result of convolution of *x* and *y*.

x A one-dimensional array of size n. It contains data used for convolution.

y A one-dimensional array of size m. It contains data used for convolution.

Description
The input **array** *x* and *y* can be of any supported arithmetic data type and sizes n and m, respectively. Conversion of the data to **double complex** type is performed internally. If both *x* and *y* are real type, the result is a one-dimensional array *c* of size $n + m - 1$. If either one of *x* and *y* is complex type, the result *c* is complex type.
If *x* and *y* are considered as two vectors of polynomial coefficients, the convolution of *x* and *y* is equivalent to the multiplication of these two polynomials.

Algorithm
The convolution of two functions $x(t)$ and $y(t)$, denoted as $x * y$, is defined by

$$x * y \equiv \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau$$

in the time domain with $x * y = y * x$. According to the theorem of convolution, if $X(f)$ and $Y(f)$ are Fourier transforms of $x(t)$ and $y(t)$, that is,

$$x(t) \Longleftrightarrow X(f) \qquad \text{and} \qquad y(t) \Longleftrightarrow Y(f)$$

then

$$x * y \Longleftrightarrow X(f)Y(f)$$

According to the theorem of discrete convolution,
if $X_d(f)$ and $Y_d(f)$ are discrete Fourier transforms of sequences $x[nT]$ and $y[nT]$, that is,

$$x[nT] \Longleftrightarrow X_d(f) \qquad \text{and} \qquad y[nT] \Longleftrightarrow Y_d(f)$$

then

$$x[nT] * y[nT] \Longleftrightarrow X_d(f)Y_d(f)$$

Where T is the sample interval and n is an index used in the summation.
Based on the theorem of discrete convolution, DFT can be used to compute discrete convolutions. The procedure is first to compute the DFT of $x[nT]$ and $y[nT]$ and then use the theorem to compute

$$c[nT] = x[nT] * y[nT] = F_d^{-1}[X_d(f)Y_d(f)]$$

to get the convolution result.
Suppose $x[nT]$ and $y[nT]$ have the same length N for n from 0 to $N-1$. Then the DFT of $x[nT]$ and $y[nT]$ have N points, while $c[nT]$ has length of $N+N-1$. Therefore, the signal produced by $x[nT] * y[nT]$ has length that is longer than the DFT length. The length difference produces time aliasing when the inverse DFT of $X_d(f)Y_d(f)$ is used to obtain $x[nT] * y[nT]$. Thus, $c[nt] = x[nT]y[nT]$ is a time-aliased version of the sequences convolution. To solve the time-aliasing problem, DFT can be used to compute discrete convolution of two signals if zero padded to the signals to extend their lengths to equal the discrete convolution length. That is, if the lengths of $x[nT]$ and $y[nT]$ are N_x and N_y, respectively, then $x[nT]$ and $y[nT]$ can be padded with $N_y - 1$ and $N_x - 1$ zeroes, respectively. This zero padding produces sample signals that are both of length $N_x + N_y - 1$. DFT can be used with $(N_x + N_y - 1)$ points to compute $X_d(f)$ and $Y_d(f)$. The length, $N_x + N_y - 1$, of the sample sequence obtained from the inverse DFT $F_d^{-1}[X_d(f)Y_d(f)]$ equals the length of the discrete convolution so that the discrete convolution is obtained.
In this numeric function, the sizes of two convolution arrays *x* and *y* are expanded to $m + n - 1$ and zero padded internally. The DFT algorithm is used to compute the discrete Fourier transform of *x* and *y*. Multiplying two transforms together component by component, then using the inverse DFT algorithm to take the inverse discrete Fourier transform of the products, the answer is the convolution $x(nT) * y(nT)$.

Example 1
Assume $x(t)$ and $y(t)$ are polynomial functions, convolution of *x* and *y* is equivalent to multiplication of the two polynomials.

$$\begin{aligned} x(t) &= t^5 + 2 * t^4 + 3 * t^3 + 4 * t^2 + 5 * t + 6; \\ y(t) &= 6 * t + 7; \end{aligned}$$

Convolution of $x(t) * y(t)$ or multiplication of polynomials $x(t)$ and $y(t)$ is equal to

$$c(t) = x(t) * y(t) = 6 * t^6 + 19 * t^5 + 32 * t^4 + 45 * t^3 + 58 * t^2 + 71 * t + 42$$

```
#include <stdio.h>
#include <numeric.h>

#define N 6        /* data x array size */
#define M 2        /* data y array size */
#define N2 (N+M-1)

int main() {
   int i;
   array double c[N2],x[N]={1,2,3,4,5,6},y[M]={6,7};

   conv(c,x,y); /* float data convolution */
   printf("Polynomial multiplication\n");
   printf("x=%6.3f\n",x);
   printf("y=%6.3f\n",y);
   printf("c=%6.3f\n",c);
}
```

Output

```
Polynomial multiplication
x= 1.000  2.000  3.000  4.000  5.000  6.000

y= 6.000  7.000

c= 6.000 19.000 32.000 45.000 58.000 71.000 42.000
```

Example 2

The sequences corresponding to the input and unit pulse response of a filter in a sample data control system are given as

$$x[n] = \begin{cases} n+1 & 0 \le n < 2 \\ 5-n & 2 \le n \le 3 \\ 0 & \text{elsewhere} \end{cases}$$

$$y[n] = \begin{cases} -n/2 & 2 \le n \le 4 \\ 0 & \text{elsewhere} \end{cases}$$

The convolution sum or output sequence becomes

$$\begin{aligned} c[n] = x[n] * y[n] &= \sum_{m=-\infty}^{\infty} x[m]y[n-m] \\ &= \begin{cases} \sum_{m=2}^{4} x[m]y[n-m] & 2 \le n \le 7 \\ 0 & \text{elsewhere} \end{cases} \end{aligned}$$

The six values of $c[n]$ for $i = 2$ to 7 can be calculated as follows

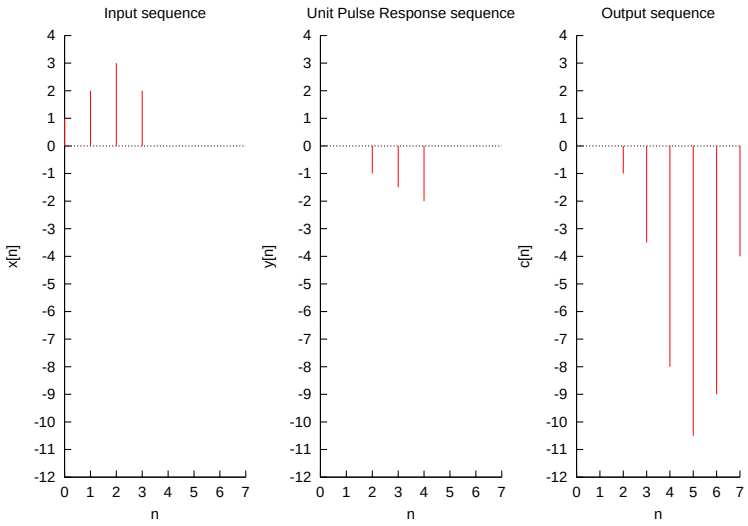
$$\begin{aligned} c[2] &= x[2]h[0] + x[3]h[-1] + x[4]h[-2] = -1 \\ c[3] &= x[2]h[1] + x[3]h[0] + x[4]h[-1] = -3.5 \\ c[4] &= x[2]h[2] + x[3]h[1] + x[4]h[0] = -8 \\ c[5] &= x[2]h[3] + x[3]h[2] + x[4]h[1] = -10.5 \\ c[6] &= x[2]h[4] + x[3]h[3] + x[4]h[2] = -9 \\ c[7] &= x[2]h[5] + x[3]h[4] + x[4]h[3] = -4 \end{aligned}$$

```
#include <stdio.h>
#include <chplot.h>
#include <numeric.h>

#define N 4        /* data x array size */
#define M 5        /* data y array size */
#define N2 (N+M-1)

int main() {
   int i,n1[N],n2[M],n3[N2];
   array double c[N2],x[N]={1,2,3,2},y[M]={0,0,-1,-1.5,-2};
   class CPlot plot;
   string_t labelx="x",
            labely="y";
```

```
   linspace(n1,0,N-1);
   linspace(n2,0,M-1);
   linspace(n3,0,N2-1);
   conv(c,x,y);
   printf("x=%6.3f\n",x);
   printf("y=%6.3f\n",y);
   printf("c=%6.3f\n",c);

   plot.data2D(n1,x);
   plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
   plot.axisRange(PLOT_AXIS_X, 0,7,1);
   plot.axisRange(PLOT_AXIS_Y, -12,4,1);
   plot.label(PLOT_AXIS_X,"n");
   plot.label(PLOT_AXIS_Y,"x[n]");
   plot.title("Input sequence");
   plot.plotting();
   plot.deletePlots();
   plot.data2D(n2,y);
   plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
   plot.axisRange(PLOT_AXIS_X, 0,7,1);
   plot.axisRange(PLOT_AXIS_Y, -12,4,1);
   plot.label(PLOT_AXIS_X,"n");
   plot.label(PLOT_AXIS_Y,"y[n]");
   plot.title("Unit Pulse Response sequence");
   plot.plotting();
   plot.deletePlots();
   plot.data2D(n3,c);
   plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
   plot.axisRange(PLOT_AXIS_X, 0,7,1);
   plot.axisRange(PLOT_AXIS_Y, -12,4,1);
   plot.label(PLOT_AXIS_X,"n");
   plot.label(PLOT_AXIS_Y,"c[n]");
   plot.title("Output Response sequence");
   plot.plotting();
}
```

Output

```
x= 1.000  2.000  3.000  2.000

y= 0.000  0.000 -1.000 -1.500 -2.000

c= 0.000  0.000 -1.000 -3.500 -8.000 -10.500 -9.000 -4.000
```



See Also
fft(), **ifft**(), **conv2**(), **deconv**().

References
Nussbaumer, H. J, *Fast Fourier Transform and Convolution Algorithms*, New York: Springer-Verlay, 1982

conv2

Synopsis
#include <**numeric.h**>
int conv2(**array double complex** *c*[&][&], **array double complex** *f*[&][&],
array double complex *g*[&][&], ... /* [**string_t** *method*] */);

Syntax
conv2(*c*, *f*, *g*)
conv2(*c*, *g*, *g*, *method*);

Purpose
Two-dimensional Discrete Fourier Transform (DFT) based convolution.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
c A two-dimensional **array** with size of a section of $(na+nb-1) \times (ma+mb-1)$ specified by "method". It contains the result of convolution of *f* and *g*.

f A two-dimensional **array** of size $(na) \times (ma)$ used for convolution.

g A two-dimensional **array** of size $(nb) \times (mb)$ used for convolution.

method The **string_t** *method* used to specify the convolution result.
"full" - (default) returns the full 2-D convolution. The size is $(na + nb - 1) \times (ma + mb - 1)$.
"same" - returns the central part of 2-D convolution. That is the same size as *f*.
"valid" - returns only those parts of 2-D convolution that are computed without the zero-padded edges, the size of *c* is $(na - nb + 1) \times (ma - mb + 1)$ where the size of *f* must be bigger than the size of *g*. That is, size(*f*) $>$ size(*g*).

Description
The input two-dimensional **array** *f* and *g* can be of any supported arithmetic data type and sizes, respectively. Conversion of the data to **double complex** is performed internally. If both of *f* and *g* are **real** data, the result is a two-dimensional **real array** *c*. If either one of *f* and *g* is **complex** , the result **array** *c* will be **complex** data type.
If size of *f* and size of *g* are $(ma) \times (na)$ and $(mb) \times (nb)$, respectively, then
method = "full" (default) the size of *c* is $(ma + mb - 1) \times (na + nb - 1)$;
method = "same" the size of *c* is $(ma) \times (na)$;
method = "valid" the size of *c* is $(ma - mb + 1) \times (na - nb + 1)$ where the size of *f* must be bigger than the size of *g*. That is, size(*f*)$>$ size(*g*).

Algorithm
Two-dimensional convolution is analogous in form to one-dimensional convolution. Thus for two functions $f(x, y)$ and $g(x, y)$, we define

$$f(x, y) * g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\alpha, \beta) g(x - \alpha, y - \beta) d\alpha d\beta$$

The convolution theorem in two dimensions, then, is given by the relations

$$f(x,y) * g(x,y) \Longleftrightarrow F(u,v)G(u,v)$$

where $F(u,v)$ and $G(u,v)$ are two-dimensional Fourier transform of $f(x,y)$ and $g(x,y)$, respectively. The discrete convolution of the two functions $f(mT,nT)$ and $g(mT,nT)$ is given by the relation

$$f(mT,nT) * g(mT,nT) = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(mT,nT)g(k_1-m,k_2-n)$$

for $k_1 = 0,1,2,\cdots,M-1$ and $k_2 = 0,1,2,\cdots,N-1$. Where T is the sampling interval, M and N are periods in x and y directions, respectively.
Similarly, the discrete convolution theorem in two dimensions is given as

$$c(mT,nT) = f(mT,nT) * g(mT,nT) \Longleftrightarrow F_d(f_1,f_2)G_d(f_1,f_2)$$

where $F_d(f_1,f_2)$ and $G_d(f_1,f_2)$ are discrete Fourier transform of $f(mT,nT)$ and $g(mT,nT)$, respectively. We can compute the DFT of $f(mT,nT)$ and $g(mT,nT)$ and then use the convolution theorem to compute

$$c(mT,nT) = f(mT,nT) * g(mT,nT) = F_d^{-1}[F_d(f_1,f_2)G_d(f_1,f_2)]$$

to get the two-dimensional result.
Similar to the one-dimensional convolution, if periods M and N are used to compute the convolution, there is a time-aliasing in the signals. To solve the time-aliasing problem, zeroes are padded to the signal to extend their lengths to equal the discrete convolution length. That is, if the lengths of $f(nT,mT)$ is $(na)\times(ma)$ and $g(nT,mT)$ is $(nb)\times(mb)$, then $f(nT,mT)$ and $g(nT,mT)$ are padded with $(nb-1)\times(mb-1)$ and $(na-1)\times(ma-1)$ zeroes, respectively. This zero padding produces sample signals that are both of length $(na+nb-1)\times(ma+mb-1)$. A two-dimensional DFT with $(na+nb-1)\times(ma+mb-1)$ points is used to compute $F_d(f_1,f_2)$ and $G_D(F_1,F_2)$. The length, $(na+nb-1)\times(ma+mb-1)$, of the sample sequence obtained from the inverse DFT $F_d^{-1}[F_d(f_1,f_2)G_d(f_1,f_2)]$ equals the length of the discrete convolution so that the discrete convolution is produced.
In this numeric function, the size of two convolution arrays *f* and *g* are expanded to $(na+nb-1)\times(ma+mb-1)$ and zero padded internally. The DFT algorithm is used to compute the two-dimensional discrete Fourier transform of *f* and *g*. Multiplying two transforms together component by component, then using the inverse DFT algorithm to take the inverse discrete Fourier transform of the products, the answer is the two-dimensional convolution $f(mT,nT) * g(mT,nT)$.

Example
In image processing, a Sobel filter is a simple approximation to the concept of edge detection. Convoluting the original two-dimensional image data with a Sobel filter

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

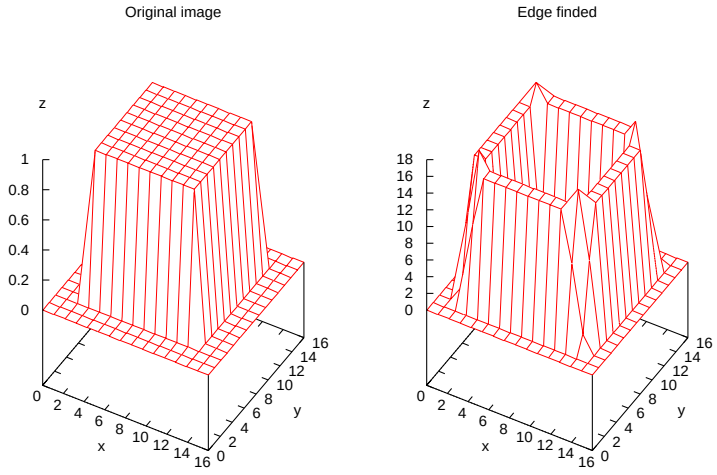
detects the edges in x-direction. Similarly, convoluting a Sobel filter

$$g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

which is the transpose of matrix g_x, edges in y-direction can be detected.

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
   int i, j;
   array double g[3][3]={{-1,0,1},{-2,0,2},{-1,0,1}};
   array double x[16], y[16], z1[256], f[16][16], H[18][18], V[18][18],Z[18][18], Z1[256];

   linspace(x,0,16);
   linspace(y,0,16);
   for(i=3; i<13; i++)
      for(j=3; j<13; j++) {
         z1[i*16+j]=1;
         f[i][j] = 1;
   }
   plotxyz(x,y,z1);                    /* original image */
   conv2(H,f,g);
   conv2(V,f,transpose(g));
   Z = H .* H + V .* V;                /* magnitude of the pixel value */
   for(i = 0; i<16; i++)
      for(j=0; j<16; j++)
         Z1[i*16+j] = Z[i+1][j+1];
   plotxyz(x,y,Z1);                    /* edge finded image */
}
```

Output



See Also

fft(), **ifft**(), **conv**(), **deconv**().

References

Rafael C. Gonzalez, Paul Wintz, Digital Image Processing, Second Edition, Addison-Wesley Publishing Company, 1987.

corrcoef

Synopsis
#include <numeric.h>
int corrcoef(array double &*c*, **array double** &*x*, ... /* [**array double** &*y*] */);

Syntax
corrcoef(*c*, *x*)
corrcoef(*c*, *x*, *y*)

Purpose
Correlation coefficients calculation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
c Correlation coefficient matrix formed from array *x*.

x A vector of observation or a two-dimensional **array** whose column is an observation and row is a variable.

y Optional argument. *y* is a vector or two-dimensional array whose length or column length shall be the same as the column length of *x*.

Description
The input array *x* can be of any supported arithmetic data type of vector or two-dimensional array of any size $n \times m$ (if *x* is a vector, regard the size as $1 \times m$). Each column of *x* is an observation and each row is a variable. Optional input *y* can only be of **double** data type and same column length as *x*. That is, the size of *y* is $n_1 \times m$ or regard as $1 \times m$ if *y* is a vector. Attachment of *y* to row a of *x* as a new expanding matrix [**X**] is performed internally. **corrcoef**(*c*, *x*, *y*) is equivalent to **corrcoef**(*c*,[**X**]) where **X** is defined as $\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$. The size of [**X**] is $(n + n_1) \times m$. Conversion of the data to **double** is performed internally. The result *c* is a matrix of correlation coefficients.

Algorithm
For array *x* of size $N \times M$, which has N variable and each variable has total M observation, correlation coefficient matrix *c* is related to the covariance matrix C by

$$c_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

where C is a covariance matrix. It is calculated by

$$C = \frac{1}{N-1} u * u'$$

where element u is defined as

$$u_{ij} = x_{ij} - \mu_i; \qquad i = 0, 1, \cdots, N; j = 0, 1, \cdots, M$$

and

$$\mu_i = \frac{1}{M}\sum_{j=1}^{M} x_{ij}; \qquad\qquad i = 0, 1, \cdots, N$$

where x_{ij} is the element of the input matrix.

Example
Consider an array $x[4][5]$.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 1 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

where each row of the x is a variable and column is an observation. The result array c is the correlation coefficient matrix.

```
#include <stdio.h>
#include <numeric.h>

#define N 4        /* data array size */
#define N1 5
#define N2 4

int main(){
   int i,j;
   array double  c[N2][N2];
   array double  x[N][N1]={{1,2,3,4,5},
                           {1,2,3,4,5},
                           {0,0,0,0,1},
                           {2,3,4,5,6}};
   corrcoef(c,x);
   printf("x=%6.3f\n",x);
   printf("correlation coefficient matrix\n");
   printf("%6.3f\n",c);
}
```

Output

```
x= 1.000  2.000  3.000  4.000  5.000
 1.000  2.000  3.000  4.000  5.000
 0.000  0.000  0.000  0.000  1.000
 2.000  3.000  4.000  5.000  6.000

correlation coefficient matrix
 1.000  1.000  0.707  1.000
 1.000  1.000  0.707  1.000
 0.707  0.707  1.000  0.707
 1.000  1.000  0.707  1.000
```

See Also
covariance().

correlation2

Synopsis
#include <**numeric.h**>
double correlation2(**array double** *x*[&][&], **array double** *y*[&][&]);

Syntax
correlation2(*x*, *y*)

Purpose
Two-dimensional correlation coefficient.

Return Value
This function returns the two-dimensional correlation coefficient.

Parameters
x A square matrix of size $n \times n$. It contains the original date for correlation coefficient calculation.

y A square matrix of the same size as *x*. It contains the original date for correlation coefficient calculation.

Algorithm
This function is used to calculate the two-dimensional correlation coefficient. The two-dimensional correlation coefficient is defined as

$$c = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(xx_{ij} * yy_{ij})}{\sqrt{(\sum_{i=1}^{n}\sum_{j=1}^{n} xx_{ij}^2) * (\sum_{i=1}^{n}\sum_{j=1}^{n} yy_{ij}^2)}}$$

where

$$xx_{ij} = x_{ij} - \mu_x$$
$$yy_{ij} = y_{ij} - \mu_y$$

μ_x and μ_y are the mean values of the matrix *x* and *y*.

$$\mu_x = \frac{1}{n * n}\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}$$
$$\mu_y = \frac{1}{n * n}\sum_{i=1}^{n}\sum_{j=1}^{n} y_{ij}$$

Example
Calculate two square matrix correlation coefficient.

```
#include <stdio.h>
#include <numeric.h>

#define NC 6
#define ND 3
```

```
#define N 20

int main() {
   int i,j,k;
   array double x[3][3]={1,2,3,
                         3,4,5,
                         6,7,8};
   array double y[3][3]={3,2,2,
                         3,8,5,
                         6,2,5};
   double c;
   c = correlation2(x,y);
   printf("x=\n%f",x);
   printf("y=\n%f",y);
   printf("c=\n%f\n",c);
}
```

Output

```
x=
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y=
3.000000 2.000000 2.000000
3.000000 8.000000 5.000000
6.000000 2.000000 5.000000
c=
0.326637
```

See Also
correlation().

covariance

Synopsis
#include <**numeric.h**>
int covariance(**array double** &*c*, **array double** &*x*, ... /* [**array double** &*y*] */);

Syntax
covariance(*c*, *x*)
covariance(*c*, *x*, *y*)

Purpose
Covariance matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
c Covariance matrix. If there is only one input *x* and it is a vector, this is the variance. If *x* is a matrix, where each column is an observation and each row is a variable, *c* is the covariance matrix. The diagonal of *c* is a vector of variances for each row. The square root of diagonal of *c* is a vector of standard deviations.

x A vector whose elements are an observation, or a two-dimensional **array** whose column is an observation and row is a variable.

y Optional argument. *y* could be a vector or a two-dimensional **array** whose length or column length shall be same as the column length of *x*.

Description
The input **array** *x* can be of any supported arithmetic data type of vector or matrix of any size $n \times m$ (if *x* is a vector, regard the size as $1 \times m$). Each column of *x* is an observation and each row is a variable. Optional input *y* can only be of **double** data type and same column length as *x*. That is, the size of *y* is $n_1 \times m$ or regard as $1 \times m$ if *y* is a vector. Attachment of *y* to row of *x* as a new expanding matrix [**X**] is performed internally. **covariance**(*c*, *x*, *y*) is equivalent to **covariance**(*c*,[**X**]) where **X** is defined as $\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$. The size of [**X**] is $(n + n_1) \times (m)$. Conversion of the data to **double** is performed internally. The result *c* is a variance or a covariance matrix depending on *x* and *y*. If *x* is a vector and no optional input *y*, the result *c* is a variance. Otherwise it is a covariance. Diagonal of *c* is a vector of variances for each row. The square root of the diagonal of *c* is a vector of standard deviations.

Algorithm
For each of **array** $x[N][M]$ of N variable which has total M observation, covariance function is defined as

$$covariance(x_i, x_j) = E[(x_i - \mu_i)(x_j - \mu_j)] \qquad i, j = 1, 2, \cdots, N;$$

where x_i and x_j are the *ith* and *jth* row elements. E is the mathematical expectation and $\mu_i = Ex_i$.
In function **covariance**(), the mean from each column is removed before calculating the result.

$$u_{ij} = x_{ij} - \mu_i; \qquad i = 0, 1, \cdots, N; j = 0, 1, \cdots, M$$

where

$$\mu_i = \frac{1}{M} \sum_{j=1}^{M} x_{ij}; \qquad i = 0, 1, \cdots, N$$

Then the covariance matrix is calculated by

$$c = \frac{1}{N-1} u * u'$$

Example
Consider an array $x[4][5]$.

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 0 & 0 & 1 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

where each row of the x is a variable and column is an observation. The result **array** c is the covariance matrix. The diagonal elements $c(i, j)$ represent the variances for the rows of x. The off-diagonal element $c(i, j)$ represents the covariances of rows i and j.

```
#include <stdio.h>
#include <numeric.h>

#define N 4        /* data array size */
#define N1 5
#define N2 4

int main() {
   int i,j;
   array double  c[N2][N2];
   array double  x[N][N1]={{1,2,3,4,5},
                           {1,2,3,4,5},
                           {0,0,0,0,1},
                           {2,3,4,5,6}};

   covariance(c,x);     /* covariance of matrix */

   printf("x=%8.3f\n",x);
   printf("c=%8.3f\n",c);
}
```

Output

```
x=   1.000    2.000    3.000    4.000    5.000
   1.000    2.000    3.000    4.000    5.000
   0.000    0.000    0.000    0.000    1.000
   2.000    3.000    4.000    5.000    6.000

c=   2.500    2.500    0.500    2.500
   2.500    2.500    0.500    2.500
   0.500    0.500    0.200    0.500
   2.500    2.500    0.500    2.500
```

See Also
corrcoef().

cpolyeval

Synopsis
#include <numeric.h>
double complex cpolyeval(**array double complex** *c*[&], **double complex** *z*);

Syntax
cpolyeval(*c*, *z*)

Purpose
Calculate the value of a polynomial at point *z*.

Return Value
This function returns the **complex** value of a polynomial at a given complex point *z*.

Parameters
c A vector of coefficients of a polynomial.

z A value of point x in which polynomial is evaluated.

Description
The vector of coefficient of polynomial *c* can be of any supported arithmetic data type and extent size. The value of *z* can be any data type. Conversion of the data to **double complex** is performed internally. The return value is **double complex** data of the value of the polynomial.

Algorithm
For polynomial

$$P(z) = c_0 z^n + c_1 z^{n-1} + c_2 z^{n-1} + \cdots + c_{n-1} z + c_n$$

The calculation of the polynomial value is implemented using the follow algorithm

$$P = (((\cdots (c_0 z + c_1)z + c_2)z + c_3)z + \cdots + c_{n-1})z + c_n$$

Example
Evaluate polynormal

$$P(z) = z^5 - 5z^4 + 10z^3 - 10z^2 + 5z - 1$$

at complex point $z = 0.1 + i0.1$.

```
#include <stdio.h>
#include <numeric.h>

#define NC 6
#define ND 3

int main() {
   int i,j,k;
```

```
   double c[NC]={1.0,-5.0,10.0,-10.0,5.0,-1.0};
   double complex z=complex(0.1,1),val;

   val = cpolyeval(c,z);
   printf("z = %f   function value = %f\n",z,val);
}
```

Output

```
z = complex(0.100000,1.000000)   function value = complex(2.199510,-3.819500)
```

see also
polyeval(), **polyevalarray**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

cproduct

Synopsis
#include <**numeric.h**>
double complex cproduct(**array double complex** *&a*, ... /* [**array double complex** *v*[:]] */);

Syntax
cproduct(a)
cproduct(a, v)

Purpose
Calculate products of all elements and products of the elements of each rows of a complex array.

Return Value
This function returns the product of all elements.

Parameters
a The input complex array for which the products are calculated.

v The output array which contains the products of each row of a two-dimensional complex array.

Description
This function calculates the product of all elements in a complex array of any dimension. If the array is a two-dimensional matrix, the function can calculate the products of each row. The product of all elements of the array is returned by the function and the products of the elements of each row are passed out by the argument *v*. The input array can be of any dimension and any arithmetic data types.

Example1
Calculate the products of all elements of the complex arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    complex double a[3] = {complex(1,2), complex(2,3), complex(1,2)};
    complex double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double complex products;

    products = cproduct(a);
    printf("cproduct(a) = %f\n", products);
    products = cproduct(b);
    printf("cproduct(b) = %f\n", products);
    products = cproduct(b1);
    printf("cproduct(b) = %f\n", products);
    products = cproduct(c);
    printf("cproduct(c) = %f\n", products);
}
```

Output1

```
cproduct(a) = complex(-18.000000,-1.000000)
cproduct(b) = complex(720.000000,0.000000)
cproduct(b) = complex(720.000000,0.000000)
cproduct(c) = complex(0.000000,0.000000)
```

Example2

Calculate the products of the elements of each row of two-dimensional complex arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    double complex a[2][3] = {complex(1,2),2,3,
                        4,5,6};
    array double complex b[3][4] = {1,2,3,4,
                             5,6,7,8,
                             1,2,3,4};
    array double complex cproductv1[2], cproductv2[3];

    cproduct(a,cproductv1);
    printf("cproduct(a, cproductv1) = %f\n", cproductv1);
    cproduct(b,cproductv2);
    printf("cproduct(b,cproductv2) = %f\n", cproductv2);
}
```

Output2

```
cproduct(a, cproductv1) = complex(6.000000,12.000000) complex(120.000000,0.000000)

cproduct(b,cproductv2) = complex(24.000000,0.000000) complex(1680.000000,0.000000) complex(24.000000,
```

See Also

product(), **cumprod**(), **mean**(), **median**(), **sum**(), **cumsum**().

References

cross

Synopsis
#include <**numeric.h**>
array double cross(**array double** *a*[&], **array double** *b*[&])[3];

Purpose
Calculate the cross product of two vectors.

Return Value
This function returns the cross product of vectors *a* and *b*.

Parameters
a An input array which contains the first vector.

b An input array which contains the second vector.

Description
The function **cross**() returns the cross product of vectors *a* and *b*. The number of elements of each vector shall be 3.

Example

```
#include <numeric.h>
int main() {
    array double a[3] = {1, 2, 3};
    array double b[ ] = {1, 2, 3};
    array double c[3] = {2, 3, 4};
    array double crossprod[3];

    crossprod = cross(a, b);
    printf("cross(a,b) = %f\n", crossprod);
    crossprod = cross(a, c);
    printf("cross(a,c) = %f\n", crossprod);
}
```

Output

```
cross(a,b) = 0.000000 0.000000 0.000000

cross(a,c) = -1.000000 2.000000 -1.000000
```

See Also
dot().

References

csum

Synopsis
#include <**numeric.h**>
double complex csum(**array double complex** &*a*, ... /*[**array double complex** *v*[:]]*/);

Syntax
csum(a)
csum(a, v)

Purpose
Calculate the sum of all elements of an array and the sums of each row of a two-dimensional complex array.

Return Value
This function returns the sum of all elements of the complex array.

Parameters
a The input complex array.

v The output array which contains the sums of each row of array.

Description
The function calculates the sum of all elements in a complex array of any dimension. If the array is a two-dimensional matrix, the function can calculate the sum of each row. The input array can be any arithmetic data type and of any dimension.

Example1
Calculate the sum of all elements of a complex arrays of different data types.

```
#include <numeric.h>
int main() {
    complex double a[3] = {complex(1,2), complex(2,3)};
    complex double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double complex sums;

    sums = csum(a);
    printf("csum(a) = %f\n", sums);
    sums = csum(b);
    printf("csum(b) = %f\n", sums);
    sums = csum(b1);
    printf("csum(b) = %f\n", sums);
    sums = csum(c);
    printf("csum(c) = %f\n", sums);
}
```

Output1

```
csum(a) = complex(3.000000,5.000000)
csum(b) = complex(21.000000,0.000000)
csum(b) = complex(21.000000,0.000000)
csum(c) = complex(30.000000,0.000000)
```

Example2
Calculate the sums of each elements of the complex arrays

```
#include <numeric.h>
int main() {
    double complex a[2][3] = {complex(1,2),2,3,
                        4,5,6};
    array double complex b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array int    b1[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double complex csumv1[2], csumv2[3];

    csum(a, csumv1);
    printf("csum(a, csumv1) = %.3f\n", csumv1);
    csum(b, csumv2);
    printf("csum(b, csumv2) = %.3f\n", csumv2);
    csum(b1, csumv2);
    printf("csum(b1, csumv2) = %.3f\n", csumv2);
}
```

Output2

```
csum(a, csumv1) = complex(6.000,2.000) complex(15.000,0.000)

csum(b, csumv2) = complex(10.000,0.000) complex(26.000,0.000) complex(10.000,0.000)

csum(b1, csumv2) = complex(10.000,0.000) complex(26.000,0.000) complex(10.000,0.000)
```

See Also
sum(), **cumsum**(), **trace**(), **product**().

References

ctrace

Synopsis
#include <**numeric.h**>
double complex ctrace(**array double complex** *a*[&][&]);

Purpose
Calculate the sum of diagonal elements of a complex matrix.

Return Value
This function returns the sum.

Parameters
a Input two-dimensional complex array.

Description
This function calculates the sum of the diagonal elements of the complex matrix.

Algorithm
For a matrix *a*, the trace is defined as

$$trace = \sum_{i=1}^{n} a_{ii}$$

Example

```
#include <numeric.h>
int main() {
    array double complex a[2][2] = {complex(2,3), -4,
                                    3, -7};
    array double complex a2[3][2] = {complex(2,3), -4,
                                     3, -7,
                                     1, 1};
    array double b[2][2] = {2, -4,
                            3, -7};
    double complex t;

    t = ctrace(a);
    printf("ctrace(a) = %f\n", t);
    t = ctrace(a2);
    printf("ctrace(a2) = %f\n", t);
    t = ctrace(b);
    printf("ctrace(b) = %f\n", t);
}
```

Output

```
ctrace(a) = complex(-5.000000,3.000000)
ctrace(a2) = complex(-5.000000,3.000000)
ctrace(b) = complex(-5.000000,0.000000)
```

See Also
trace(), **sum**(), **mean**(), **median**().

References

ctriangularmatrix

Synopsis
#include <numeric.h>
array complex double ctriangularmatrix(**string_t** *pos*, **array double complex** *a*[&][&], ... /* [**int** k] */)[:][:];

Syntax
ctriangularmatrix(*pos, a*)
ctriangularmatrix(*pos, a, k*)

Purpose
Get the upper or lower triangular part of a complex matrix.

Return Value
This function returns the upper or lower triangular part of a complex matrix.

Parameters
pos Input string indicating which part of the matrix to be returned.

a Input 2-dimensional array.

k Input integer. A matrix is returned on and below the *k*th diagonal of the input matrix.

Description
This function gets the triangular matrix on and below the *k*th diagonal of matrix *a*. For *pos* of "upper" the function returns the upper complex triangular part of the matrix, and for *pos* of "lower" the function returns the lower complex part of the matrix. *k* indicates the offset of the triangular matrix to the lower *k*th diagonal of the matrix.

Example

```
#include <numeric.h>
int main() {
    array double complex a[4][3] = {1,2,3,
                                    4,5,6,
                                    7,8,9,
                                    3,6,4};
    int n = 4, m=3, k;
    array double complex t[n][m];


    t = ctriangularmatrix("upper",a);
    printf("ctriangularmatrix(\"upper\", t) =\n%5.3f\n", t);

    k = 0;
    t = ctriangularmatrix("upper",a,k);
    printf("ctriangularmatrix(\"upper\", t, 0) =\n%5.3f\n", t);

    k = 1;
    t = ctriangularmatrix("upper",a,k);
    printf("ctriangularmatrix(\"upper\", t, 1) =\n%5.3f\n", t);
```

```
    k = -1;
    t = ctriangularmatrix("upper",a,k);
    printf("ctriangularmatrix(\"upper\", t, -1) =\n%5.3f\n", t);


    t = ctriangularmatrix("lower",a);
    printf("ctriangularmatrix(\"lower\", t) =\n%5.3f\n", t);

    k = 0;
    t = ctriangularmatrix("lower",a,k);
    printf("ctriangularmatrix(\"lower\", t, 0) =\n%5.3f\n", t);

    k = 1;
    t = ctriangularmatrix("lower",a,k);
    printf("ctriangularmatrix(\"lower\", t, 1) =\n%5.3f\n", t);

    k = -1;
    t = ctriangularmatrix("lower",a,k);
    printf("ctriangularmatrix(\"lower\", t, -1) =\n%5.3f\n", t);
}
```

Output

```
ctriangularmatrix("upper", t) =
complex(1.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(0.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(9.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

ctriangularmatrix("upper", t, 0) =
complex(1.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(0.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(9.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

ctriangularmatrix("upper", t, 1) =
complex(0.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)

ctriangularmatrix("upper", t, -1) =
complex(1.000,0.000) complex(2.000,0.000) complex(3.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(0.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(0.000,0.000) complex(0.000,0.000) complex(4.000,0.000)

ctriangularmatrix("lower", t) =
complex(1.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(0.000,0.000)
complex(7.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)

ctriangularmatrix("lower", t, 0) =
complex(1.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(0.000,0.000)
complex(7.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)
```

```
ctriangularmatrix("lower", t, 1) =
complex(1.000,0.000) complex(2.000,0.000) complex(0.000,0.000)
complex(4.000,0.000) complex(5.000,0.000) complex(6.000,0.000)
complex(7.000,0.000) complex(8.000,0.000) complex(9.000,0.000)
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)

ctriangularmatrix("lower", t, -1) =
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(4.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
complex(7.000,0.000) complex(8.000,0.000) complex(0.000,0.000)
complex(3.000,0.000) complex(6.000,0.000) complex(4.000,0.000)
```

See Also
triangularmatrix(), **diagonal**(), **cdiagonal**()

cumprod

Synopsis
#include <numeric.h>
int cumprod(**array double complex** *&y*, **array double complex** *&x*);

Syntax
cumprod(*y*, *x*)

Purpose
Calculate cumulative product of elements in an array.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input array with dimension less than three. It contains data to be calculated.

y Output array of same dimension as *x*. It contains the result of cumulative product.

Description
This function computes the cumulative product of the input array *x*. If the input *x* is a vector, it calculates the cumulative product of the elements of *x*. If the input *x* is a two-dimensional matrix, it calculates the cumulative product over each row. If the input *x* is a three-dimensional array, it calculates the cumulative product over the the first dimension. It is invalid for calculation of cumulative product for arrays with more than three dimensions.

Algorithm
For a vector $x = [x_1, x_2, \cdots, x_n]$, cumulative productive vector $y = [y_1, y_2, \cdots, y_n]$ is defined by

$$y_i = x_1 * x_2 * \cdots * x_i \qquad i = 1, 2, \cdots, n$$

Example
Calculation of cumulative products of arrays of different dimensions.

```
#include <numeric.h>
int main() {
   array double x[6]={1,2,3,4,5,6}, y[6];
   array double complex zx[2][3]={complex(1,1),2,3,complex(2,2),5,6}, zy[2][3];
   array double x1[2][3][4]={{1,2,3,4,
                              5,6,7,8,
                              5,6,7,8},
                              {10,11,12,13,
                               10,11,12,13,
                               14,15,16,17}};
   array double y1[2][3][4];

   cumprod(y,x);
   printf("x=%f",x);
   printf("y=%f",y);
   printf("\n");
```

```
   cumprod(zy,zx);
   printf("zx=%5.2f",zx);
   printf("zy=%5.2f",zy);
   printf("\n");

   cumprod(y1,x1);
   printf("x1=%f",x1);
   printf("y1=%f",y1);
}
```

Output

```
x=1.000000 2.000000 3.000000 4.000000 5.000000 6.000000
y=1.000000 2.000000 6.000000 24.000000 120.000000 720.000000

zx=complex( 1.00, 1.00) complex( 2.00, 0.00) complex( 3.00, 0.00)
complex( 2.00, 2.00) complex( 5.00, 0.00) complex( 6.00, 0.00)
zy=complex( 1.00, 1.00) complex( 2.00, 2.00) complex( 6.00, 6.00)
complex( 2.00, 2.00) complex(10.00,10.00) complex(60.00,60.00)

x1=1.000000 2.000000 3.000000 4.000000
5.000000 6.000000 7.000000 8.000000
5.000000 6.000000 7.000000 8.000000

10.000000 11.000000 12.000000 13.000000
10.000000 11.000000 12.000000 13.000000
14.000000 15.000000 16.000000 17.000000
y1=1.000000 2.000000 6.000000 24.000000
5.000000 30.000000 210.000000 1680.000000
5.000000 30.000000 210.000000 1680.000000

10.000000 110.000000 1320.000000 17160.000000
10.000000 110.000000 1320.000000 17160.000000
14.000000 210.000000 3360.000000 57120.000000
```

See Also

cumsum(), **product**(), **cproduct**(), **sum**(), **csum**().

cumsum

Synopsis
#include <numeric.h>
int cumsum(**array double complex** *&y*, **array double complex** *&x*);

Syntax
cumsum(*y*, *x*)

Purpose
Caculatate cumulative sum of elements in an array.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input array whose dimension are no more than three. It contains data to be calculated.

y Output array of same dimensional as *x*. It contains data the result of cumulative sum.

Description
This function computes the cumulative sum of the input array *x*. If the input *x* is a vector, it calculates the cumulative sum of the elements of *x*. If the input *x* is a two-dimensional matrix, it calculates the cumulative sum over each row. If the input *x* is a three-dimensional array, it calculates the cumulative sum over the first dimension. It is invalid for calculation of cumulative sum for arrays with more than three dimensions.

Algorithm
For a vector $x = [x_1, x_2, \cdots, x_n]$, cumulative sum vector $y = [y_1, y_2, \cdots, y_n]$ is defined by

$$y_i = x_1 + x_2 + \cdots + x_i \qquad i = 1, 2, \cdots, n$$

Example
Calculation of cumulative sum of arrays of different dimensions.

```
#include <numeric.h>
int main() {
   array double x[6]={1,2,3,4,5,6}, y[6];
   array double complex zx[2][3]={complex(1,1),2,3,complex(2,2),5,6}, zy[2][3];
   array double x1[2][3][4]={{1,2,3,4,
                              5,6,7,8,
                              5,6,7,8},
                              {10,11,12,13,
                               10,11,12,13,
                               14,15,16,17}};
   array double y1[2][3][4];

   cumsum(y,x);
   printf("x=%f",x);
   printf("y=%f",y);
   printf("\n");
```

```
   cumsum(zy,zx);
   printf("zx=%5.2f",zx);
   printf("zy=%5.2f",zy);
   printf("\n");

   cumsum(y1,x1);
   printf("x1=%f",x1);
   printf("y1=%f",y1);
}
```

Output

```
x=1.000000 2.000000 3.000000 4.000000 5.000000 6.000000
y=1.000000 3.000000 6.000000 10.000000 15.000000 21.000000

zx=complex( 1.00, 1.00) complex( 2.00, 0.00) complex( 3.00, 0.00)
complex( 2.00, 2.00) complex( 5.00, 0.00) complex( 6.00, 0.00)
zy=complex( 1.00, 1.00) complex( 3.00, 1.00) complex( 6.00, 1.00)
complex( 2.00, 2.00) complex( 7.00, 2.00) complex(13.00, 2.00)

x1=1.000000 2.000000 3.000000 4.000000
5.000000 6.000000 7.000000 8.000000
5.000000 6.000000 7.000000 8.000000

10.000000 11.000000 12.000000 13.000000
10.000000 11.000000 12.000000 13.000000
14.000000 15.000000 16.000000 17.000000
y1=1.000000 3.000000 6.000000 10.000000
5.000000 11.000000 18.000000 26.000000
5.000000 11.000000 18.000000 26.000000

10.000000 21.000000 33.000000 46.000000
10.000000 21.000000 33.000000 46.000000
14.000000 29.000000 45.000000 62.000000
```

See Also
cumprod(), **product**(), **cproduct**(), **sum**(), **csum**().

curvefit

Synopsis
#include <numeric.h>
int curvefit(**double** *a*[&], **double** *x*[&], **double** *y*[&], **void** (**funcs*)(**double**, **double** [ ]), ...
/*[**double** *sig*[ ], **int** *ia*[ ], **double** *covar*[:][:], **double** **chisq*]*/);

Syntax
curvefit(*a, x, y, funcs*)
curvefit(*a, x, y, funcs, sig*)
curvefit(*a, x, y, funcs, sig, ia*)
curvefit(*a, x, y, funcs, sig, ia, covar*)
curvefit(*a, x, y, funcs, sig, ia, covar, chisq*)

Purpose
Fit a set of data points *x* and *y* to a linear combination of specified functions of *x*.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a An output array which contains coefficients for curve fitting.

x An input array which contains variable values of data set.

y An input array which contains function values of data set.

funcs The routine given by the user that passes the values of base functions evaluated at *x*.

sig An input array which contains individual standard (measurement error) deviations of data set. It can be set to 1 if unknown.

ia An input array. Those components of nonzero entries shall be fitted, and those components of zero entries shall be held at their input values.

covar The covariance matrix of fitting results, describing the fit of data to the model. See algorithm for definition.

chisq Chi-square of fitting.

Description
Given a set of data points in arrays *x* and *y* with individual standard deviations in array *sig*, use χ^2 minimization to fit for some or all of the coefficients in array *a* of a function that depends linearly on, $y = a_i * funcs_i(x)$. The fitted coefficients are passed by array *a*. The program can also pass χ^2 and covariance matrix *covar*. If the values for array argument *ia* are constants, the corresponding components of the covariance matrix will be zero. The user supplies a routine *funcs*(*x*,*func*) that passes the function values evaluated at *x* by the array *func*.

Algorithm
The general form of the fitting formula is

$$y(x) \quad = \quad \sum_{k=1}^{M} a_k f_k(x)$$

where $f_1(x), ..., f_M(x)$ are arbitrary fixed function of *x*, called base functions. $f_k(x)$ can be a nonlinear function of *x*. The coefficients a_k are determined by minimizing chi-square which is defined as

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - \sum_{k=1}^{M} a_k f_k(x_i)}{\sigma_i} \right)^2$$

where σ_i is the standard deviation of the ith data point. If the standard deviations are unknown, they can be set to the constant value $\sigma = 1$. By defining a $n \times m$ matrix α with element (k, j) as

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{f_j(x_i) f_k(x_i)}{\sigma_i^2}$$

The element (k, j) of the covariance matrix can be expressed as

$$cover_{kj} = [\alpha]^{-1}_{kj}$$

Example 1
Fitting of the data points generated by $f(x) = 4\cos(x) + 3\sin(x) + 2e^x + 1$ with uniform random deviation to the base functions $\cos(x)$, $\sin(x)$, e^x and 1. This example uses **curvefit**(*a*, *x*, *y*, *funcs*) only.

```
#include <stdio.h>
#include <numeric.h>
#define NPT 100
#define SPREAD 0.1
#define NTERM 4

void funcs(double x, double func[]) {
    func[0]=cos(x);
    func[1]=sin(x);
    func[2]=exp(x);
    func[3]=1.0;
}
int main(void) {
    int i,j;
    array double a[NTERM],x[NPT],y[NPT],sig[NPT],u[NPT];

    linspace(x, 0, 10);
    urand(u);
    y = 4*cos(x) + 3*sin(x) +2*exp(x) +(array double [NPT])1.0 + SPREAD*u;
    curvefit(a,x,y,funcs);
    printf("\n%11s\n","parameters");
    for (i=0;i<NTERM;i++)
       printf(" a[%1d] = %f\n", i,a[i]);
}
```

Output1

```
 parameters
 a[0] = 3.990255
 a[1] = 2.994660
 a[2] = 2.000000
 a[3] = 1.051525
```

Example 2
Fitting of the data points generated by $f(x) = 4\cos(x) + 3\sin(x) + 2e^x + 1$ with uniform random deviation to the base functions of $\cos(x), \sin(x), e^x$ and 1. This example uses all the options of arguments of function **curvefit**(). The values of 3 and 1 for coefficients *a*[2] and *a*[0] are fixed, respectively.

```
#include <stdio.h>
#include <numeric.h>

#define NPT 100
#define SPREAD 0.1
#define NTERM 4

void funcs(double x, double func[]) {
    func[0]=cos(x);
    func[1]=sin(x);
    func[2]=exp(x);
    func[3]=1.0;
}

int main(void) {
    int i,j;
    array int ia[NTERM];
    array double a[NTERM],x[NPT],y[NPT],sig[NPT],covar[NTERM][NTERM],u[NPT];
    double chisq;

    linspace(x, 0, 10);
    urand(u);
    y = 4*cos(x) + 3*sin(x) +2*exp(x) +(array double [NPT])1.0 + SPREAD*u;
    sig=(array double[NPT])(SPREAD);

    ia[0] = 0; ia[1] = 1; ia[2] = 0; ia[3] = 1;
    a[0] = 4; a[1] = 3; a[2] = 2; a[3] = 1;

    curvefit(a,x,y,funcs,sig,ia,covar,&chisq);
    printf("\n%11s %21s\n","parameter","uncertainty");
    for (i=0;i<NTERM;i++)
        printf("  a[%1d] = %8.6f %12.6f\n",i,a[i],sqrt(covar[i][i]));
    printf("chi-square = %12f\n",chisq);
    printf("full covariance matrix\n");
    printf("%12f",covar);
}
```

Output 2

```
  parameter           uncertainty
  a[0] = 4.000000     0.000000
  a[1] = 2.994009     0.015044
  a[2] = 2.000000     0.000000
  a[3] = 1.052622     0.010357
chi-square =     7.538080
full covariance matrix
    0.000000     0.000000     0.000000     0.000000
```

```
0.000000     0.000226     0.000000    -0.000041
0.000000     0.000000     0.000000     0.000000
0.000000    -0.000041     0.000000     0.000107
```

See Also
polyfit(), **std**().

References
William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannerya, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

deconv

Synopsis
#include <numeric.h>
int deconv(array double complex *u*[&], **array double complex** *v*[&], **array double complex** *q*[&], ...
/* [**array double complex** *r*[&]] */);

Syntax
deconv(*u*, *v*, *q*)
deconv(*u*, *v*, *q*, *r*)

Purpose
One-dimensional Discrete Fourier Transform (DFT) based deconvolution.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
u A one-dimensional array of size n. It contains data used for deconvolution.

v A one-dimensional array of size m. It contains data used for deconvolution.

q A one-dimensional array of size $(n - m + 1)$. It contains the result of deconvolution of *u* and *v*.

r A one-dimensional array of size n. It contains the remainder item if user specified.

Description
The input arrays *u* and *v* can be of any supported arithmetic data type and sizes n and m. Conversion of the data to **double complex** type is performed internally. If both *u* and *v* are **real** data type, the deconvolution result is a one-dimensional **real array** *q* of size $(n + m - 1)$ and the remaining item is a **real array** *r* if specified. If either one of *u* and *v* is a **complex** type, then the result **array** *q* and remained item *r* will be a **complex** data type.
If *u* and *v* are vectors of polynomial coefficients, deconvolution of *u* and *v* is equivalent to the division the two polynomials.

Algorithm
According to the theorem of convolution, if $X(f)$ and $Y(f)$ are Fourier transforms of $x(t)$ and $y(t)$, that is,

$$x(t) \Longleftrightarrow X(f) \qquad \text{and} \qquad y(t) \Longleftrightarrow Y(f)$$

then

$$x * y \Longleftrightarrow X(f)Y(f)$$

The theorem of convolution of discrete convolution is described as,
if $X_d(f)$ and $Y_d(f)$ are discrete Fourier transforms of sequences $x[nT]$ and $y[nT]$, that is,

$$x[nT] \Longleftrightarrow X_d(f) \qquad \text{and} \qquad y[nT] \Longleftrightarrow Y_d(f)$$

then

$$x[nT] * y[nT] \Longleftrightarrow X_f(f)Y_d(f)$$

where T is the sample interval and n is an index used in the summation.
In function **deconv**(), the convolution result *u* is given and one sequence of convolution data *v* is known. Based on the theorem of discrete convolution, DFT can be used to compute discrete deconvolution. The procedure is first to compute the DFT of $u[nT]$ and $v[nT]$ and then use the theorem to compute

$$q[nT] + r[nT] = F_d^{-1}[U_d(f)]/F_d^{-1}[V_d(f)]$$

where $U_d(f)$ and $V_d(f)$ are DFT of sequences $u[nT]$ and $v[nT]$, respectively. In the algorithm the long division of two polynomial is used to get the result. The quotient is the convolution sequence $q[nT]$ and the remainder is $r[nT]$.

Example
There are two polynomial *U* and *V* whose coefficients *u* and *v* are $[6, 19, 32, 35, 58, 71, 42]$ and $[6, 7]$ respectively. The deconvolution of *u* and *v* is equivalent to the division of polynomial *U* by polynomial *V*.

$$\begin{aligned} U &= 6 * t^6 + 19 * t^5 + 32 * t^4 + 45 * t^3 + 58 * t^2 + 71 * t + 42 \\ V &= 6 * t + 7 \end{aligned}$$

then U/V equals

$$Q = t^5 + 2 * t^4 + 3 * t^3 + 4 * t^2 + 5 * t + 6$$

```
#include <stdio.h>
#include <numeric.h>

#define N 7        /* u array size */
#define M 2        /* response function dimension  */
#define N2 N-M+1

int main() {
   int i,j,n,m;
   array double u[N] = {6,19,32,45,58,71,42},
                v[M] = {6,7};
   array double q[N2], r[N];

   printf("u=%f\n",u);
   printf("v=%f\n",v);

   deconv(u,v,q);  /* real v, not specify remaind result */
   deconv(u,v,q,r); /* real v, specified remaind result */

   printf("q=%f\n",q);
   printf("r=%f\n",r);
}
```

Output

```
u=6.000000 19.000000 32.000000 45.000000 58.000000 71.000000 42.000000

v=6.000000 7.000000
```

```
q=1.000000 2.000000 3.000000 4.000000 5.000000 6.000000

r=0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 -0.000000
```

See Also
conv(), **conv2**(), **filter**().

References
Nussbaumer, H. J, *Fast Fourier Transform and Convolution Algorithms*, New York: Springer-Verlay, 1982.

derivative

Synopsis
#include <**numeric.h**>
double derivative(**double** (**func*)(**double**), **double** *x*, ... /* [**double** &*err*], [**double** *h*] */);

Syntax
derivative(*func*, *x*)
derivative(*func*, *x*, &*err*)
derivative(*func*, *x*, &*err*, *h*)

Purpose
Calculate numerical derivative of a function at a given point.

Return Value
This function returns a derivative value of function *func* at point *x* in **double** data type.

Parameters
func The function taking derivative.

x A point at which the derivative is evaluated.

err An estimated error in the derivative will return as *err* if this argument is passed.

h An initial estimated step size. It does not have to be small, but it should be an increment in *x* over which *func* changes substantially. If the user does not specify this value, the value of $0.02 * x$ or 0.0001 (if *x* < 0.0001) is used by default.

Description
The function taken derivative, *func*, is specified as a pointer to a function that takes a **double** as an argument and returns a **double** value at *x*. The returned value is the derivative of the function at point *x*. The optional argument *err* contains the estimate of the error in the calculation of derivative. It helps the user to estimate the result. If the optional argument *h* is given, calculation of the derivative uses the initial step size of *h*. Otherwise it uses the default value.

Algorithm
The symmetrized form of the definition of the derivate is defined as the limit

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

when $h \to 0$. For this formula, the truncation and roundoff errors are $e_t \sim h^2 f'''$ and $e_r \sim \epsilon_f |f(x)/h|$, respectively. The optimal choice of h is

$$h \sim (\frac{\epsilon_f f}{f'''})^{1/3} \sim (\epsilon_f)^{1/3} x_c$$

and the relative error is

$$(e_r + e_t)/|f'| \sim (\epsilon_f)^{2/3} f^{2/3} (f''')^{1/3}/f' \sim (\epsilon_f)^{2/3}$$

Example
Calculate the derivative of function $x\sin(x)$ at point $x = 2.5$

$$\frac{d}{dx}x\sin(x)|_{x=2.5} = \{\sin(x) - x\cos(x)\}|_{x=2.5} = -1.404387$$

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double func(double x) {
    return x*sin(x);
}

int main() {
    double yprime, func(double), x, h, err;

    x = 2.5;
    yprime = derivative(func, x);
    /* derivative of func = x*sin(x) is sin(x)+x*cos(x) */
    printf("yprime = %f\n", sin(x)+x*cos(x));
    printf("derivative(func, x) = %f\n", yprime);
    yprime  = derivative(func, x, &err);
    printf("derivative(func, x, &err) = %f, err = %f\n", yprime, err);
    h = 0.1; // default h = 0.001
    yprime  = derivative(func, x, &err, h);
    printf("derivative(func, x, &err, h) = %f, err = %f\n", yprime, err);
}
```

Output

```
yprime = -1.404387
derivative(func, x) = -1.404387
derivative(func, x, &err) = -1.404387, err = 0.000000
derivative(func, x, &err, h) = -1.404387, err = 0.000000
```

see also
derivatives(), **integral1**(), **integral2**(), **integral3**(), **integration2**(), **integration3**().
References
Ridder, C. J. F., *Advances in Engineering Software*, 1982, vol. 4, no. 2, pp. 75-76. [2].

derivatives

Synopsis
#include <**numeric.h**>
array double derivatives(**double** (**func*)(**double**), **double** *x*[&], ... /* [**double** &*err*], [**double** *h*] */)[:];

Syntax
derivative(*func*, *x*)
derivative(*func*, *x*, &*err*)
derivative(*func*, *x*, &*err*, *h*)

Purpose
Calculate derivatives of a function numerically at multiple points.

Return Value
This function returns an **array** of derivative values of function *func* at points *x*[&].

Parameters
func Function its derivatives are calculated.

x A series of points in which derivatives are calculated.

err An estimated error in the derivative which will return as *err* if the user specified.

h An initial estimated step size. It does not have to be small, but rather should be an increment in *x* over which *func* changes substantially. If the user do not specify this value, the value of $0.02 * x$ or 0.0001 (if $x < 0.0001$) is used by default.

Description
The function taking derivative, *func*, is specified as a pointer to a function that takes an **array** of **double** data as an argument and returns an **array double** values at *x*[&]. The values of *x* can be of any real type. Conversion of the data to **double** is performed internally. The return value is an **array** of derivatives of the function at points specified by **array** *x*. If the optional argument *h* is specified, which shall be **double** data type, the derivative uses initial step size of *h*. Otherwise it uses the default value. Optional argument *err* shall be a **double** data and it means the estimate of the error in the derivate. This helps the user to estimate the result.

Algorithm
Using a for-loop to evaluate the derivative values specified in **array** *x* and in the loop using the same algorithm as **derivative**().

Example
Take the derivatives of function

$$f(x) = x * \sin(x) * \cos(x)$$

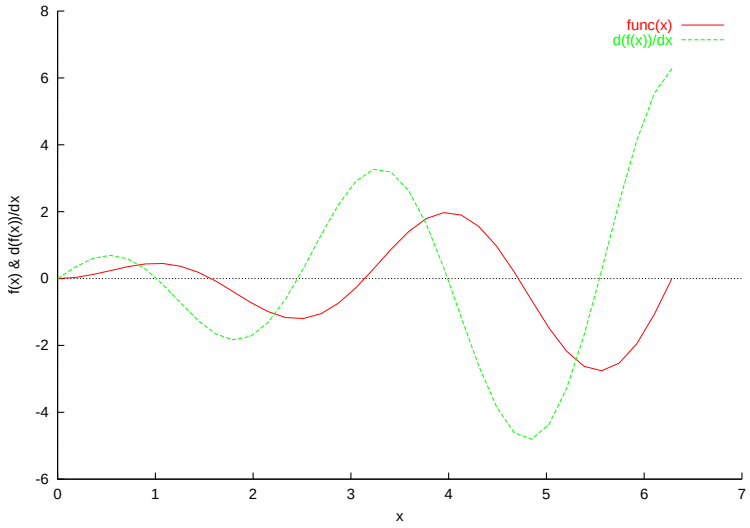
at 36 different positions meanly spaced between 0 and 2π.
Analytically, the first derivative of this function is

$$f'(x) = 0.5 * \sin(2x) + x\cos(2x)$$

```
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 36

double func(double x) {
   return x*sin(x)*cos(x);
}

int main() {
   array double x[N], y[N], yprime[N];
   double err, h;
   int i;
   class CPlot plot;

   linspace(x, 0, 2*M_PI);
   printf("       x      derivative() value       Cal. value\n");
//   /* Specify the initial step h and return estimate error */
//   yprime =derivatives(func,x,&err,0.8);
   yprime =derivatives(func,x);    /* use default step value and do not return err */

   fevalarray(y,func,x);

   for (i=0; i<2; i++)
      printf("%12.6f    %12.6f        %12.6f\n",x[i],yprime[i],
             (0.5*sin(2*x[i])+x[i]*cos(2*x[i])));
   printf("     --------     ---------            --------\n");
   for (i=34; i<36; i++)
      printf("%12.6f    %12.6f        %12.6f\n",x[i],yprime[i],
             (0.5*sin(2*x[i])+x[i]*cos(2*x[i])));
   plot.label(PLOT_AXIS_X,"x");
   plot.label(PLOT_AXIS_Y,"f(x) & d(f(x))/dx");
   plot.data2D(x,y);
   plot.data2D(x,yprime);
   plot.legend("f(x)",0);
   plot.legend("d(f(x))/dx",1);
   plot.plotting();
}
```

Output

```
       x derivative() value  Cal. value
    0.000000     0.000000     0.000000
    0.179520     0.343760     0.343760
     -------- -----------------
    6.103666     5.538781     5.538781
    6.283186     6.283187     6.283187
```



see also

derivative(), **integral1**(), **integral2**(), **integral3**(), **integration2**(), **integration3**().

References

Ridder, C. J. F., *Advances in Engineering Software*, 1982, vol. 4, no. 2, pp. 75-76.

determinant

Synopsis
#include <**numeric.h**>
double determinant(**array double** *a*[&][&]);

Purpose
Calculate the determinant of a matrix.

Return Value
This function returns the determinant of matrix *a*.

Parameters
a The input matrix.

Description
The function **determinant**() returns the determinant of matrix *a*. If the input matrix is not a square matrix, the function will return NaN.

Example
Calculate the determinant of matrices with different data type.

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                            3, 7};
    /* a2 is an ill-detition matrix  */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};
    /* a3 is singular */
    array double a3[2][2] = {2, 4,
                             4, 8};
    array float b[2][2] = {2, 4,
                           3, 7};
    array double c[3][3] = {-1,5,6,
                            3,-6,1,
                            6,8, 9} ; /* n-by-n matrix */
    array double d[3][3] = {2,1,-2,
                            4,-1,2,
                            2,-1,1} ; /* n-by-n matrix */

    double det;

    det = determinant(a);
    printf("determinant(a) = %g\n", det);
    det = determinant(a2);
    printf("determinant(a2) = %g\n", det);
    det = determinant(a3);
    printf("determinant(a3) = %g\n", det);
    det = determinant(b);
    printf("determinant(b) = %g\n", det);
    det = determinant(c);
    printf("determinant(c) = %g\n", det);
    det = determinant(d);
```

```
    printf("determinant(d) = %g\n", det);
}
```

Output

```
determinant(a) = 2
determinant(a2) = -0.0038
determinant(a3) = -0
determinant(b) = 2
determinant(c) = 317
determinant(d) = 6
```

See Also
cdeterminant(), **inverse**(), **diagonal**(), **ludecomp**(), **rcondnum**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

diagonal

Synopsis
#include <**numeric.h**>
array double diagonal(**array double** *a*[&][&], ... /* [**int** *k*] */)[:];

Syntax
diagonal(*a*);
diagonal(*a*, *k*);

Purpose
Form a vector with diagonals of a matrix.

Return Value
This function returns a column vector formed from the elements of the *k*th diagonal of the matrix *a*. By default, *k* is 0.

Parameters
a An input matrix.

k An input integer indicating which element the vector is formed from.

Description
The function **diagonal**() produces a colum vector formed from the elements of the *k*th diagonal of the matrix *a*.

Example1

```
#include <numeric.h>
int main() {
    array double a[3][3] = {1, 2, 3,
                            4, 5, 6,
                            7, 8, 9};
    int n =3, k = 1;
    array double d[n], d2[n-(abs(k))];

    d = diagonal(a);
    printf("diagonal(a) = %f\n", d);

    d2 = diagonal(a, k);
    printf("diagonal(a, 1) = %f\n", d2);

    k = -1;
    d2 = diagonal(a, k);
    printf("diagonal(a, -1) = %f\n", d2);
}
```

Output1

```
diagonal(a) = 1.000000 5.000000 9.000000

diagonal(a, 1) = 2.000000 6.000000

diagonal(a, -1) = 4.000000 8.000000
```

Example2

```
#include <numeric.h>
int main() {
    array double a[4][3] =  {1, 2, 3,
                             4, 5, 6,
                             7, 8, 9,
                             4, 4, 4};
    int m=4, n = 3, k1 = 1, k2 = -1;
    array double d[min(m, n)];
    array double d1[min(m, n-k1)];
    array double d2[min(m+k2, n)];

    d = diagonal(a);
    printf("diagonal(a) = %f\n", d);

    d1 = diagonal(a, k1);
    printf("diagonal(a, 1) = %f\n", d1);

    d2 = diagonal(a, k2);
    printf("diagonal(a, -1) = %f\n", d2);
}
```

Output2

```
diagonal(a) = 1.000000 5.000000 9.000000

diagonal(a, 1) = 2.000000 6.000000

diagonal(a, -1) = 4.000000 8.000000 4.000000
```

Example3

```
#include <numeric.h>
int main() {
    array double a[3][4] =  {1, 2, 3, 4,
                             5, 6, 7, 8,
                             9, 4, 4, 4};
    int m=3, n = 4, k1 = 1, k2 = -1;
    array double d[min(m, n)];
    array double d1[min(m, n-k1)];
    array double d2[min(m+k2, n)];

    d = diagonal(a);
    printf("diagonal(a) = %f\n", d);

    d1 = diagonal(a, k1);
    printf("diagonal(a, 1) = %f\n", d1);

    d2 = diagonal(a, k2);
    printf("diagonal(a, -1) = %f\n", d2);
}
```

Output3

```
diagonal(a) = 1.000000 6.000000 4.000000

diagonal(a, 1) = 2.000000 7.000000 4.000000

diagonal(a, -1) = 5.000000 4.000000
```

See Also
diagonalmatrix().
cdiagonal(), **cdiagonalmatrix**().

References

diagonalmatrix

Synopsis
#include <numeric.h>
array double diagonalmatrix(**array double** *v*[&], ... /* [int k] */)[:][:];

Syntax
diagonalmatrix(*v*)
diagonalmatrix(*v, k*)

Purpose
Form a diagonal matrix.

Return Value
This function returns a square matrix with specified diagonal elements.

Parameters
v An input vector.

k An input integer indicating specified diagonal elements of the matrix.

Description
The function **diagonalmatrix**() returns a square matrix of order $n + abs(k)$, with the elements of *v* on the *k*th diagonal. $k = 0$ represents the main diagonal, $k > 0$ above the main diagonal, and $k < 0$ below the main diagonal.

Example

```
#include <numeric.h>
int main() {
    array double v[3] = {1,2,3};
    int n = 3, k;
    array double a[n][n];

    a = diagonalmatrix(v);
    printf("diagonal matrix a =\n%f\n", a);

    k = 0;
    a = diagonalmatrix(v,k);
    printf("diagonalmatrix(a, 0) =\n%f\n", a);

    k = 1;
    array double a2[n+abs(k)][n+abs(k)];
    a2 = diagonalmatrix(v,k);
    printf("diagonalmatrix(a2, 1) =\n%f\n", a2);

    k = -1;
    array double a3[n+abs(k)][n+abs(k)];
    a3 = diagonalmatrix(v,k);
    printf("diagonalmatrix(a3, -1) =\n%f\n", a3);
}
```

Output

```
diagonal matrix a =
1.000000 0.000000 0.000000
0.000000 2.000000 0.000000
0.000000 0.000000 3.000000

diagonalmatrix(a, 0) =
1.000000 0.000000 0.000000
0.000000 2.000000 0.000000
0.000000 0.000000 3.000000

diagonalmatrix(a2, 1) =
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 2.000000 0.000000
0.000000 0.000000 0.000000 3.000000
0.000000 0.000000 0.000000 0.000000

diagonalmatrix(a3, -1) =
0.000000 0.000000 0.000000 0.000000
1.000000 0.000000 0.000000 0.000000
0.000000 2.000000 0.000000 0.000000
0.000000 0.000000 3.000000 0.000000
```

See Also

cdiagonalmatrix(), **diagonal**() **cdiagonal**().

References

difference

Synopsis
#include <**numeric.h**>
array double difference(**array double** *a*[&])[:];

Purpose
Calculate differences between the adjacent elements of an array.

Return Value
This function returns the array of differences of adjacent elements of array *a*.

Parameters
a The input array.

Description
The function **difference**() calculates the differences of adjacent elements of an array. The input array can be of any arithmetic data types.

Example
Calculate the differences of the adjacent elements of the arrays with different data types.

```
#include <numeric.h>
#define N 6
int main() {
    array double a[N] = {1, 2, 10, 4, 5, 6};
    array float f[N] = {1, 2, 10, 4, 5, 6};
    array double d[N-1];

    d = difference(a);
    printf("difference(a) = %f\n", d);
    d = difference(f);
    printf("difference(f) = %f\n", d);
}
```

Output

```
difference(a) = 1.000000 8.000000 -6.000000 1.000000 1.000000

difference(f) = 1.000000 8.000000 -6.000000 1.000000 1.000000
```

See Also
derivative(), **derivatives**(), **sum**(), **product**().

References

dot

Synopsis
#include <**numeric.h**>
double dot(**array double** *a*[&], **array double** *b*[&]);

Purpose
Calculate the dot product of a vector.

Return Value
This function returns the dot product of two vectors.

Parameters
a An input one-dimensional array.

b Another input one-dimensional array.

Description
The function calculates the dot product of two vectors. The number of elements of two vectors should be the same. Otherwise the function returns NaN.

Example

```
#include <numeric.h>
int main() {
    array double a[4] = {1, 2, 3, 4};
    array double b[ ] = {1, 2, 3, 4};
    double dotprod;

    dotprod = dot(a, b);
    printf("dot(a,b) = %f\n", dotprod);
}
```

Output

```
dot(a,b) = 30.000000
```

See Also
cross().

References

eigensystem

Synopsis
#include <**numeric.h**>
int eigensystem(**array double complex** *evalues*[&], **array double complex** *evectors*[&][&],
array double complex *a*[&][&], ... /* [**string_t** *mode*] */);

Syntax
eigensystem(*evalues*, NULL, *a*); eigensystem(*evalues*, *evectors*, *a*);
Purpose
Find eigenvalues and eigenvectors.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
evalues Output one-dimensional array which contains the calculated eigenvalues.

evectors Output square matrix which contains the calculated eigenvectors.

a Input square matrix for which the eigenvalues and eigenvectors are calculated.

mode The optional **string_t** *mode* used to specify if using a preliminary balancing step before the calculation of eigenvalues and eigenvectors.

Description
This function calculates the eigenvalues and eigenvectors of n-by-n matrix *a* so that $a*v = \lambda*v$, where λ and v are the vector of eigenvalues and eigenvectors, respectively. The computed eigenvectors are normalized so that the norm of each eigenvector equals 1. *mode* is used to specify if a preliminary balancing step before the calculation is performed or not. Generally, balancing improves the condition of the input matrix, enabling more accurate computation of the eigenvalues and eigenvectors. But, it also may lead to incorrect eigenvectors in special cases. By default, a preliminary balancing step is taken.
Example1
This example calculates the eigenvalues and eigenvectors of a non-symmetrical matrix. The eigenvalues of this matrix is real number. The formula $a * v - evalues * v$ is calculated to verify the results.

```
#include <numeric.h>
#define N 5

int main() {
array double a[5][5] = {2,      30,    100,  1000, 10000,
                        0.3,    1,     10,   100,  1000,
                        0.04,   0.1,   1,    10,   100,
                        0.001,  0.01,  0.1,  1,    10,
                        0.0002, 0.003, 0.01, 0.1,  1};
     array double evalues[N];
     array double evectors[N][N];
     array double *v, vv[N][N];
     int i, status;

     eigensystem(evalues, evectors, a);
     printf("eigennvalues of a =\n%e\n", evalues);
```

```
     printf("eigenvectors of a =\n%e\n", evectors);

     eigensystem(evalues, evectors, a, "nobalance");
     printf("no balance: eigennvalues of a =\n%e\n", evalues);
     printf("no balance: eigenvectors of a =\n%e\n", evectors);

     vv = transpose(evectors);
       for(i=0; i<N; i++) {
          v = (array double [:])(double [N])&vv[i][0];
          printf("A*v-evalues[i]*v = %f\n", a*v-evalues[i]*v);
       }

}
```

Output1

```
eigennvalues of a =
7.327477e+00 -1.508424e+00 1.809473e-01 -1.555781e-15 2.032652e-16

eigenvectors of a =
-9.959858e-01 -9.945522e-01 -9.966546e-01 -7.564396e-14 4.879875e-14
-8.892060e-02 1.028639e-01 -5.396722e-02 -9.891903e-15 -2.815928e-14
-1.025131e-02 1.687971e-02 -6.047655e-02 9.948856e-01 8.268104e-01
-6.173564e-04 -2.900250e-04 1.047629e-02 -1.010080e-01 5.588107e-01
-9.959858e-05 -9.945522e-05 -9.966546e-05 1.519397e-04 -6.414918e-02

no balance: eigennvalues of a =
7.327477e+00 1.809473e-01 -1.508424e+00 1.227397e-15 -4.649855e-16

no balance: eigenvectors of a =
9.959858e-01 9.966546e-01 -9.945522e-01 1.586027e-13 -2.911653e-16
8.892060e-02 5.396722e-02 1.028639e-01 2.746367e-15 -1.029631e-14
1.025131e-02 6.047655e-02 1.687971e-02 9.842796e-01 9.714797e-01
6.173564e-04 -1.047629e-02 -2.900250e-04 -1.764452e-01 -2.367118e-01
9.959858e-05 9.966546e-05 -9.945522e-05 7.801720e-03 1.395638e-02

A*v-evalues[i]*v = -0.000000 0.000000 0.000000 0.000000 -0.000000

A*v-evalues[i]*v = 0.000000 0.000000 0.000000 0.000000 -0.000000

A*v-evalues[i]*v = 0.000000 0.000000 0.000000 0.000000 0.000000

A*v-evalues[i]*v = 0.000000 0.000000 0.000000 0.000000 0.000000

A*v-evalues[i]*v = -0.000000 0.000000 0.000000 0.000000 -0.000000
```

Example2

The eigenvalues and eigenvectors of symmetrical and non-symmetrical matrices are calculated. The symmetrical matrix has real eigenvalues and a non-symmetrical matrix has real or complex eigenvalues.

```
#include <numeric.h>
int main() {
     /* eigenvalues of a symmtrical matrix are always real numbers */
     array double a[3][3] = {0.8, 0.2, 0.1,
                             0.2, 0.7, 0.3,
                             0.1, 0.3, 0.6};
     /* eigenvalues of a non-symmtrical matrix can be either real numbers or
        complex numbers */
```

```
     /* this non-symmtrical matrix has real eigenvalues */
     array double a2[3][3] = {0.8, 0.2, 0.1,
                              0.1, 0.7, 0.3,
                              0.1, 0.1, 0.6};
     /* this non-symmtrical matrix has complex eigenvalues and eigenvectors*/
     array double a3[3][3] = {3, 9, 23,
                               2, 2, 1,
                               -7, 1, -9};


     array double evalues[3],evectors[3][3];
     array double complex zevalues[3], zevectors[3][3];

     eigensystem(evalues, NULL, a);
     printf("evalues =\n%f\n", evalues);

     eigensystem(evalues, NULL, a2);
     printf("evalues =\n%f\n", evalues);
     eigensystem(zevalues, evectors, a2);
     printf("zevalues =\n%f\n", zevalues);
     printf("evectors =\n%f\n", evectors);

     eigensystem(evalues, NULL, a3);
     printf("evalues =\n%f\n", evalues);
     eigensystem(zevalues, zevectors, a3);
     printf("evalues =\n%f\n", zevalues);
     printf("evectors =\n%f\n", zevectors);
}
```

Output2

```
evalues =
1.108807 0.652624 0.338569

evalues =
1.000000 0.600000 0.500000

zevalues =
complex(1.000000,0.000000) complex(0.600000,0.000000) complex(0.500000,0.000000)


evectors =
-0.744845 -0.707107 0.408248
-0.579324 0.707107 -0.816497
-0.331042 0.000000 0.408248

evalues =
NaN NaN 3.241729

evalues =
complex(-3.620864,10.647303) complex(-3.620864,-10.647303) complex(3.241729,0.000000)


evectors =
complex(0.854461,0.000000) complex(0.854461,0.000000) complex(0.604364,0.000000)

complex(-0.024440,-0.125397) complex(-0.024440,0.125397) complex(0.744064,0.000000)

complex(-0.236405,0.444621) complex(-0.236405,-0.444621) complex(-0.284803,0.000000)
```

Example3
The eigenvalues and eigenvectors of real and complex matrices are calculated in this example.

```
#include <numeric.h>
int main() {
     array double complex a[3][3] = {0.8, 0.2, 0.1,
                                     0.1, 0.7, 0.3,
                                     0.1, 0.1, 0.6};

     array double complex a2[3][3] = {complex(0.8,-1), 0.2, 0.1,
                                                0.1, 0.7, 0.3,
                                                0.1, 0.1, 0.6};
    /* this non-symmtrical matrix has complex eigenvalues and eigenvectors*/
    array double complex a3[3][3] = {3, 9, 23,
                                     2, 2, 1,
                                     -7, 1, -9};


     array double complex evalues[3];
     array double complex evectors[3][3];

     eigensystem(evalues, evectors, a);
     printf("eigenvalues of a =\n%f\n", evalues);
     printf("eigenvectors of a =\n%f\n", evectors);

     eigensystem(evalues, evectors, a2);
     printf("eigenvalues of a =\n%f\n", evalues);
     printf("eigenvectors of a =\n%f\n", evectors);

     eigensystem(evalues, evectors, a3);
     printf("eigenvalues of a =\n%f\n", evalues);
     printf("eigenvectors of a =\n%f\n", evectors);
}
```

Output3

```
eigenvalues of a =
complex(1.000000,0.000000) complex(0.600000,0.000000) complex(0.500000,0.000000)


eigenvectors of a =
complex(0.744845,0.000000) complex(-0.707107,0.000000) complex(-0.408248,0.000000)

complex(0.579324,0.000000) complex(0.707107,0.000000) complex(0.816497,0.000000)

complex(0.331042,0.000000) complex(0.000000,0.000000) complex(-0.408248,0.000000)


eigenvalues of a =
complex(0.796911,-0.968453) complex(0.832118,-0.037024) complex(0.470970,0.005477)


eigenvectors of a =
complex(0.989164,0.000000) complex(0.003912,-0.225655) complex(-0.025122,-0.090331)

complex(-0.019749,0.106886) complex(0.891481,0.000000) complex(0.794571,0.000000)

complex(0.008946,0.098280) complex(0.391299,-0.034802) complex(-0.598227,0.044616)
```

```
eigenvalues of a =
complex(-3.620864,10.647303) complex(-3.620864,-10.647303) complex(3.241729,-0.000000)


eigenvectors of a =
complex(0.854461,0.000000) complex(0.854461,0.000000) complex(0.604364,0.000000)

complex(-0.024440,-0.125397) complex(-0.024440,0.125397) complex(0.744064,0.000000)

complex(-0.236405,0.444621) complex(-0.236405,-0.444621) complex(-0.284803,-0.000000)
```

See Also
balance(), **choldecomp**(), hessdecomp().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

expm

Synopsis
#include <numeric.h>
int expm(**array double complex** *y[&][&]*, **array double complex** *x[&][&]*);

Syntax
expm(*y*, *x*)

Purpose
Computes the matrix exponential.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input square matrix. It contains data to be calculated.

y Output square matrix which contains data of the result of *x* exponential.

Description
This function computes the matrix exponential of the input matrix *x* using a scaling and squaring algorithm with a Pade approximation. If any of the eigenvalues of input matrix *x* are positive, the output matrix *y* is a real matrix. If *x* has nonpositive eigenvalues, the results *y* shall be a complex matrix.

Example
Calculation of exponentials of real and complex matrix.

```
#include <numeric.h>
int main() {
   array double x[3][3]={1,2,3,
                         3,4,5,
                         6,7,8};
   array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                         3,complex(4,1),complex(2,5),
                         0,0,0};
   array double complex zy[3][3];
   array double y[3][3];

   expm(y,x);
   printf("x = \n%f",x);
   printf("y = \n%f",y);
   printf("\n");

   expm(zy,zx);
   printf("zx = \n%5.1f",zx);
   printf("zy = \n%5.1f",zy);
}
```

Output

```
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y =
157372.953093 200034.605129 242697.257164
290995.910241 369883.552084 448769.193928
491431.845963 624654.472518 757878.099072

zx =
complex(  1.0,  1.0) complex(  2.0,  2.0) complex(  0.0,  0.0)
complex(  3.0,  0.0) complex(  4.0,  1.0) complex(  2.0,  5.0)
complex(  0.0,  0.0) complex(  0.0,  0.0) complex(  0.0,  0.0)
zy =
complex(-44.9, 56.8) complex(-87.5, 70.9) complex(-101.5,-18.9)
complex(-12.5,118.7) complex(-57.4,175.5) complex(-155.5, 67.8)
complex(  0.0,  0.0) complex(  0.0,  0.0) complex(  1.0,  0.0)
```

See Also

logm(), **funm**(), **cfunm**(), **sqrtm**().

References

G. H. Golub, C. F. Van Loan, Matrix Computations Third edition, The Johns Hopkins University Press, 1996

factorial

Synopsis
#include <**numeric.h**>
unsigned long long factorial(**unsigned int** *n*);

Purpose
Calculate factorial of an unsigned integer.

Return Value
This function returns factorial of integer *n*.

Parameters
n Input unsigned integer.

Description
The function **factorial** calculates the product of all the integers from 1 to *n*. The factorial is defined as

$$f = n!$$

Example

```
#include <numeric.h>
int main() {
    unsigned long long f;

    f = factorial(3);
    printf("factorial(3) = %llu\n", factorial(3));
}
```

Output

```
factorial(3) = 6
```

See Also
combination().

References

fevalarray

Synopsis
#include <**numeric.h**>
int fevalarray(**array double** &*y*, **double** (**func*)(**double** *x*), **array double** &*x*, ... /* [**array int** &*mask*, **double** *value*] */);

Syntax
fevalarray(*y, func, x*)
fevalarray(*y, func, x, mask, value*)

Purpose
Array function evaluation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y An output array which contains the result of function evaluation.

x An input array at which the function values will be evaluated.

func A pointer to function given by the user.

mask An input array of integer. If the value of an element of array *mask* is 1, the function evaluation will be applied to that element. If the value of an element of array is 0, the value from the optional fisth argument *value* will be used for that element.

value The default value used to replace the function evaluation when the entry of *mask* is 0.

Description
This function evaluates a user function applied to each element of the input array *x*. The input array can be of any arithmetic data type and dimension.

Example
Evaluation of function $f(x) = x^2$ for arrays of different dimensions.

```
#include <numeric.h>
#define N 4
#define M 5

double func(double x) {
   return x*x;
}

int main() {
  array double y1[N],    x1[N];
  array float y2[N][M], x2[N][M]; /* diff data types, diff dim */
  array int mask1[N] = {1, 0, 1, 0};

  linspace(x1, 1, N);
  fevalarray(y1, func, x1);
```

```
  printf("y1 = %f\n", y1);

  linspace(x1, 1, N);
  fevalarray(y1, func, x1, mask1, -1);
  printf("y1 = %f\n", y1);


  linspace(x2, 1, N*M);
  fevalarray(y2, func, x2);
  printf("y2 = %f\n", y2);
}
```

Output

```
y1 = 1.000000 4.000000 9.000000 16.000000

y1 = 1.000000 -1.000000 9.000000 -1.000000

y2 = 1.000000 4.000000 9.000000 16.000000 25.000000
36.000000 49.000000 64.000000 81.000000 100.000000
121.000000 144.000000 169.000000 196.000000 225.000000
256.000000 289.000000 324.000000 361.000000 400.000000
```

See Also
cfevalarray(), **streval**().

References

fft

Synopsis
#include <numeric.h>
int fft(array double complex *&y*, **array double complex** *&x*, ... /* [**int** *n* [**int** *dim*[&]]] */);

Syntax
fft(*y*, *x*)
fft(*y*, *x*, *n*)
fft(*y*, *x*, *dim*);

Purpose
N-dimensional Fast Fourier Transform (FFT) calculation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y Result of FFT, the same dimension and size **array** as *x*.

x An n-dimensional array of data used for FFT.

n Optional argument, user specified FFT points for one-dimensional data.

dim A one-dimensional array of optional arguments with **int** type. It contains user specified FFT points. Each element corresponds to data in that dimension. For example, there is a three-dimensional data *x[m][n][l]*. The FFT points of the array is specified by m, n, l. Then the array *dim* is given values of $\text{dim}[0] = m, \text{dim}[1] = n$ and $\text{dim}[2] = l$.

Description
The multi-dimensional (maximum dimension is three) array *x* can be of any supported arithmetic data type and size. Conversion of the data to double complex is performed internally. Array *y* with the same dimension as array *x* contains the result of the fast Fourier transform. The optional argument *n* of int type is used to specify the number of points for FFT of one-dimensional data For multi-dimensional data, The optional array argument *dim* of int type. contains the value for user specified FFT points. The dimensions of the input array *x* are contained in array *dim*. If no optional argument is passed, the number of FFT points is obtained from the input array *x*.

Algorithm
Mixed-radix Fast Fourier Transform algorithm developed by R. C. Singleton (Stanford Research Institute, Sept, 1968). A discrete Fourier transform is defined by

$$Y(n) = \sum_{j=0}^{N-1} X(j) exp(\frac{-2\pi ijn}{N}), (n = 0, 1, \cdots, N-1);$$

and the inverse discrete Fourier transform is defined by

$$Z(n) = \sum_{j=0}^{N-1} X(j) exp(\frac{2\pi ijn}{N}), (n = 0, 1, \cdots, N-1);$$

satisfying the condition of $Z(j) = NX(j), (j = 0, 1, \cdots, N-1)$. Function **fft**() evaluates these sums using fast Fourier transform technique. It is not required that N be a power of 2. One-, two-, and three-dimensional transforms can be performed.

Example 1
This example illustrates the use of the FFT. There is a signal

$$f(t) = \begin{cases} 9e^{-9t} & t \geq 0 \\ 0 & t < 0 \end{cases}$$

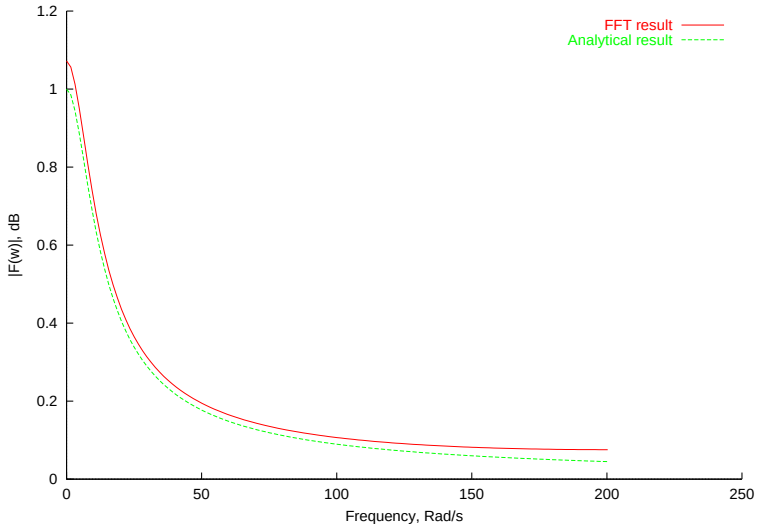
Analytically, the Fourier transform of function $f(t)$ is given by

$$F(\omega) = \frac{9}{9 + j\omega}$$

This example illustrates an approach to estimate the Fourier transform of less common signals because the sample frequency is limited. In this example, total 256 points of signal are sampled in 4 seconds. So the sample rate is $256/4 = 64$Hz.

```
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 256    /* total sample points */
#define M 4      /* sample time, 4 seconds */

int main() {
   int i;
   array double t[N], f[N], E[N/2], W[N/2+1],y[2][N/2+1];
   array double complex F[N], Fp[N/2+1], Fa[N/2+1];
   double Ts, Ws;

   linspace(t, 0, M);
   f = 9*exp(-9*t);
   Ts = t[1] - t[0];
   Ws = 2*M_PI/Ts;
   fft(F,f);
   linspace(W,0,N/2);
   W = W*Ws/N;
   for (i=0; i<N/2+1; i++) {
      Fp[i] = F[i]*Ts;
      Fa[i] = 9.0/(9+W[i]*complex(0,1));
      y[0][i] = abs(Fp[i]);
      y[1][i] = abs(Fa[i]);
   }
   plotxy(W,y,"FFT example","Frequency, Rad/s","|F(w)|, dB");
}
```

Output



Example 2
This example shows the one-dimensional fft usage. When the optional argument *n* is specified, it takes *n* points FFT. If it is not specified, a total number of elements of *x* is taken to FFT by default.

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

#define N 16
#define M 4

int main() {
   int i,j,k,retval;
   array double x1[N];
   array double complex y1[N],y2[M],x2[N],x3[M];
   int n = M;

   linspace(x1, 0, 1);

   fft(y1,x1);
   ifft(x2,y1);
   printf("FFT number of points by default\n");
   printf("Original data    FFT result    FFT + iFFT result\n");
   for (i=0; i<N; i++)
   printf("%5.3f  %5.3f  %5.3f\n",x1[i],y1[i],x2[i]);

   printf("FFT number of points is specified by n=%d\n",n);
   fft(y2,x1,n);
   ifft(x3,y2,n);
   printf("\n\nOriginal data    FFT result    FFT + iFFT result\n");
   for (i=0; i<M; i++)
   printf("%5.3f  %5.3f  %5.3f\n",x1[i],y2[i],x3[i]);
}
```

Output

```
FFT number of points by default
Original data    FFT result    FFT + iFFT result
0.000  complex(8.000,0.000)  complex(-0.000,-0.000)
```

```
0.067  complex(-0.533,-2.681)  complex(0.067,0.000)
0.133  complex(-0.533,-1.288)  complex(0.133,-0.000)
0.200  complex(-0.533,-0.798)  complex(0.200,0.000)
0.267  complex(-0.533,-0.533)  complex(0.267,0.000)
0.333  complex(-0.533,-0.356)  complex(0.333,0.000)
0.400  complex(-0.533,-0.221)  complex(0.400,-0.000)
0.467  complex(-0.533,-0.106)  complex(0.467,0.000)
0.533  complex(-0.533,0.000)  complex(0.533,0.000)
0.600  complex(-0.533,0.106)  complex(0.600,0.000)
0.667  complex(-0.533,0.221)  complex(0.667,0.000)
0.733  complex(-0.533,0.356)  complex(0.733,0.000)
0.800  complex(-0.533,0.533)  complex(0.800,0.000)
0.867  complex(-0.533,0.798)  complex(0.867,0.000)
0.933  complex(-0.533,1.288)  complex(0.933,0.000)
1.000  complex(-0.533,2.681)  complex(1.000,-0.000)
FFT number of points is specified by n=4


Original data    FFT result    FFT + iFFT result
0.000  complex(0.400,0.000)  complex(0.000,0.000)
0.067  complex(-0.133,-0.133)  complex(0.067,0.000)
0.133  complex(-0.133,0.000)  complex(0.133,0.000)
0.200  complex(-0.133,0.133)  complex(0.200,0.000)
```

Example 3

This example shows the multi-dimensional fft usage. When the optional array argument *dim* is specified, the value for each element is used to specify the number of elements for each dimension of the array for performing FFT. If it is not specified, a total number of elements of *x* is taken to FFT by default.

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

#define N1 4
#define N2 3
#define M 3

int main() {
   int i,j,k,retval;
   array double complex x1[N1][N2],y1[N1][N2],x3[N1][N2];
   array double complex y[M][M],x2[M][M];
   int dim[2];

   linspace(x1, 0, 1);
   dim[0]=M;dim[1]=M;
   fft(y,x1,dim);
   dim[0]=M;dim[1]=M;
   ifft(x2,y,dim);

   printf("Input data =\n");
   printf("%5.3f",x1);
   printf("\nFFT result data =\n");
   printf("%5.3f",y);
   printf("\nFFT + iFFT result data =\n");
   printf("%5.3f",x2);

   fft(y1,x1);
   ifft(x3,y1);

   printf("\n\nInput data =\n");
```

```
   printf("%5.3f",x1);
   printf("\nFFT result data =\n");
   printf("%5.3f",y1);
   printf("\nFFT + iFFT result data =\n");
   printf("%5.3f",x3);
}
```

Output

```
Input data =
complex(0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)
complex(0.818,0.000) complex(0.909,0.000) complex(1.000,0.000)

FFT result data =
complex(3.273,0.000) complex(-0.409,-0.236) complex(-0.409,0.236)
complex(-1.227,-0.709) complex(0.000,0.000) complex(0.000,0.000)
complex(-1.227,0.709) complex(0.000,-0.000) complex(0.000,-0.000)

FFT + iFFT result data =
complex(0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)


Input data =
complex(0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)
complex(0.818,0.000) complex(0.909,0.000) complex(1.000,0.000)

FFT result data =
complex(6.000,0.000) complex(-0.545,-0.315) complex(-0.545,0.315)
complex(-1.636,-1.636) complex(0.000,-0.000) complex(0.000,-0.000)
complex(-1.636,0.000) complex(-0.000,-0.000) complex(-0.000,0.000)
complex(-1.636,1.636) complex(0.000,0.000) complex(0.000,0.000)

FFT + iFFT result data =
complex(-0.000,0.000) complex(0.091,0.000) complex(0.182,0.000)
complex(0.273,0.000) complex(0.364,0.000) complex(0.455,0.000)
complex(0.545,0.000) complex(0.636,0.000) complex(0.727,0.000)
complex(0.818,0.000) complex(0.909,0.000) complex(1.000,0.000)
```

See Also

ifft().

References

R. C. Singleton, *An Algorithm for Computing the Mixed Radix F. F. T.*, IEEE Trans, Audio Electroacoust., AU-1(1969) 93-107.

MJ Oleson, *Netlib Repository at UTK and ORNL* , go/fft-olesen.tar.gz, http://www.netlib.org.

filter

Synopsis
#include <numeric.h>
int filter(array double complex *v*[&], **array double complex** *u*[&],
array double complex *x*[&], **array double complex** *y*[&], ...
/* [**array double complex** *zi*[&], [**array double complex** *zf*[&]] */);

Syntax
filter(*v*, *u*, *x*, *y*)
filter(*v*, *u*, *x*, *y*, *zi*, *zf*)

Purpose
Filters the data in vector *x* with a filter represented by vectors *u* and *v* to create the filtered data *y*. The filter is a direct form II transposed implementation of the standard difference equation:

$$\begin{aligned} y(n) &= v_0 * x(n) + v_1 * x(n-1) + ... + v_{nb} * x(n-nb-1) \\ &\quad -u_1 * y(n-1) - ... - u_{na} * y(n-na-1) \end{aligned}$$

The input-output description of this filtering operation in the z-transform domain is a rational transfer function

$$Y(z) = \frac{v_0 + v_1 z^{-1} + \cdots + v_{nb-1} z^{-nb-1}}{1 + u_1 z^{-1} + \cdots + u_{na-1} z^{-na-1}} X(z)$$

Return Value
This function returns 0 on success and -1 on failure.

Parameters
v A vector of size nb. It contains the numerator of polynomial coefficients of the filter.

u A vector of size na. It contains the denominator of polynomial coefficients of the filter.

x A vector of size N. It contains the raw data.

y A vector of size N. It contains the filtered data.

zi A vector of size $(\max(na, nb) - 1)$. It contains initial delays of the filter.

zf A vector of size max(na, nb). It contains the final delays of the filter.

Description
The numerator coefficients with zeros of the system transfer function *v*, denominator coefficients with poles of the system transfer function *u*, and vector *x* of input data can be of any supported arithmetic data type and size. Conversion of the data to double complex is performed internally. The vector *y* which is the same size as *x* contains the result of filtered output. The optional arguments *zi* and *zf* are used to set the initial values of delays and get the final delays of the filter. They shall be double complex data type. The leading coefficient of denominator u_0 must be non-zero, as the other coefficients are divided by u_0.

Algorithm
The block diagram of the direct form II transposed implementation is shown in Figure 1.

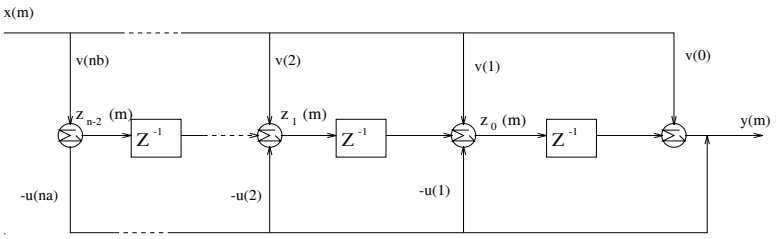


Figure 11.1: Direct form II transposed implementation

The operation of filter at sample m is given by the time domain difference equations

$$\begin{aligned} y(m) &= v_0 x(m) + z_0(m-1) \\ z_0(m) &= v_1 x(m) - u_1 y(m) + z_1(m-1) \\ \vdots &= \vdots \quad \vdots \\ z_{n-3}(m) &= v_{n-2} x(m) - u_{n-2} y(m) + z_{n-2}(m-1) \\ z_{n-2}(m) &= v_{n-1} x(m) - u_{n-1} y(m) \end{aligned}$$

where $n = \max(na, nb)$ and $m = 0, 1, \cdots, N$. Initial delay state $z_0(0), z_0(1), \cdots, z_0(n-2)$ are contained in vector *zi*. Final delay states $z_{(N-1)}(0), z_{(N-1)}(1), and z_{(N-1)}(n-1)$ are kept in vector *zf*.

Example 1
This example shows the usage of function of **filter**(), specification of the initial delays *zi* and the final delays *zf* are passed back to the calling function.

```
#include <stdio.h>
#include <numeric.h>

#define N1 6        /* x array size */
#define N2 1        /* B term coefficient number */
#define N3 4        /* A term coefficient number */

int main()
{
   int i,j,n,m;
   array double complex x[N1],v[N2],u[N3],zi[N3-1],zf[N3-1];
   array double x1[N1],v1[N2],u1[N3],y1[N1];
   array double complex y[N1];
   for (i=0;i<N1;i++){
      real(x[i])=i+1;
      imag(x[i])=i-1;
      x1[i] = i+1;
   }
   for (i=0;i<N3;i++) {
      real(u[i])=i+1;
      imag(u[i])=i+1;
      u1[i] = i+1;
   }
   for (i=0;i<N2;i++) {
      v[i]=i+1;
      v1[i] = i+1;
```

```
   }
   for (i=0;i<N3-1;i++) {
      zi[i]=1;
   }

   filter(v,u,x,y,zi,zf);  /* User specified initial delays */

   printf("u=%6.3f\n",u);
   printf("v=%6.3f\n",v);
   printf("x=%6.3f\n",x);
   printf("y=%6.3f\n",y);
   printf("zi=%6.3f\n",zi);
   printf("zf=%6.3f\n",zf);

   filter(v1,u1,x1,y1);        /* User did not specify initial delays */

   printf("u1=%6.3f\n",u1);
   printf("v1=%6.3f\n",v1);
   printf("x1=%6.3f\n",x1);
   printf("y1=%6.3f\n",y1);

   return 0;
}
```

Output

```
u=complex( 1.000, 1.000) complex( 2.000, 2.000) complex( 3.000, 3.000)
complex( 4.000, 4.000)

v=complex( 1.000, 0.000)

x=complex( 1.000,-1.000) complex( 2.000, 0.000) complex( 3.000, 1.000)
complex( 4.000, 2.000) complex( 5.000, 3.000) complex( 6.000, 4.000)


y=complex( 1.000,-1.000) complex( 0.000, 1.000) complex( 0.000, 0.000)
complex(-1.000, 0.000) complex( 6.000,-5.000) complex(-4.000, 9.000)


zi=complex( 1.000, 0.000) complex( 1.000, 0.000) complex( 1.000, 0.000)


zf=complex(-6.000,-3.000) complex(-12.000,-7.000) complex(16.000,-36.000)


u1= 1.000  2.000  3.000  4.000

v1= 1.000

x1= 1.000  2.000  3.000  4.000  5.000  6.000

y1= 1.000  0.000  0.000  0.000  5.000 -4.000
```

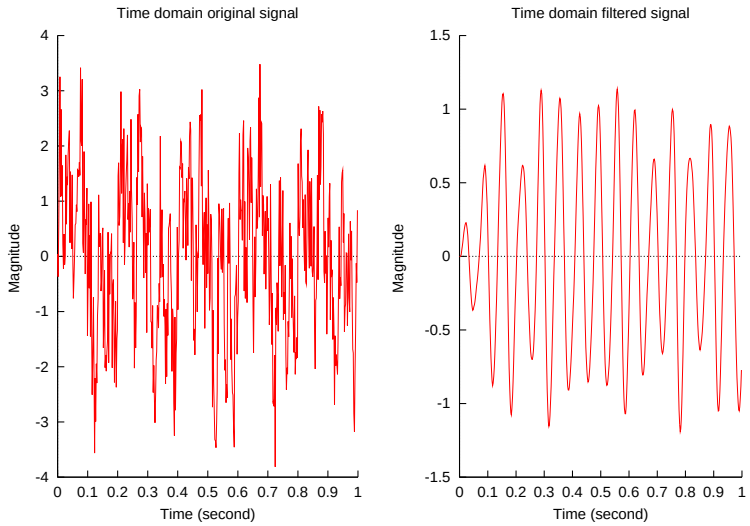
Example 2
This example shows how to use the FFT algorithm to find the spectrum of signals which is buried in noises and use the filter algorithm to filter unwanted signals.

```
#include <stdio.h>
```

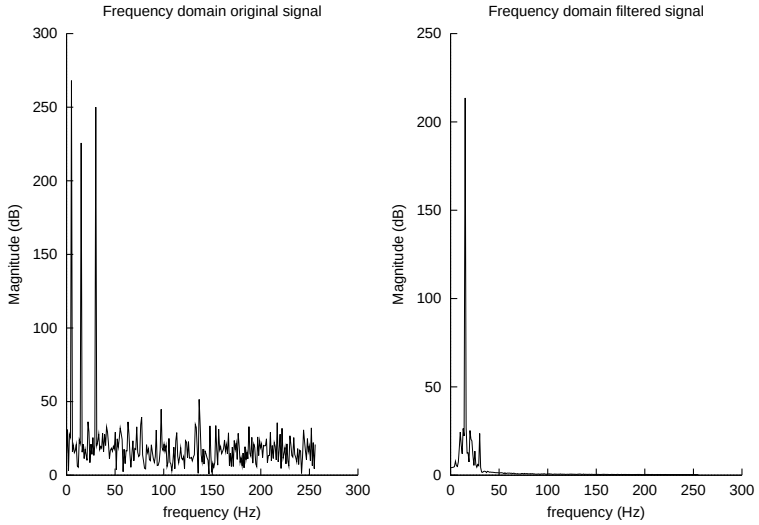
```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 512

int main() {
   array double t[N], x[N], y[N], Pyy[N/2], f[N/2], u[7], v[7];
   array double complex Y[N];
   int i;
   class CPlot plot;

   /* The filter arguments u[i] and v[i] are designed according to the characteristic
    of filter specified. In this example program, we use the designed filter to
   filter the raw data. */
   u[0]=1;u[1]=-5.66792131;u[2]=13.48109005;u[3]=-17.22250511;
   u[4]=12.46418230;u[5]=-4.84534157;u[6]=0.79051978;
   v[0]=0.00598202; v[1]=-0.02219918; v[2]=0.02645738; v[3]=0;
   v[4]=-0.02645738;v[5]=0.02219918;v[6]=-0.00598202;

   linspace(t,0,N-1);
   t = t/N;
   for (i=0; i<N; i++) {
      x[i] = sin(2*M_PI*5*t[i]) + sin(2*M_PI*15*t[i]) + sin(2*M_PI*t[i]*30);
      x[i]=x[i]+3*(urand(NULL)-0.5);
   }

   filter(v,u,x,y);
   plotxy(t,x,"Time domain original signal","Time (second)","Magnitude ");
   plotxy(t,y,"Time domain filtered signal","Time (second)","Magnitude ");

   fft(Y,x);
   for (i=0; i<N/2; i++)
      Pyy[i] = abs(Y[i]);
   linspace(f,0,N/2);
   plotxy(f,Pyy,"Frequency domain original signal","frequency (Hz)","Magnitude (db)");
   fft(Y,y);
   for (i=0; i<256; i++)
      Pyy[i] = abs(Y[i]);
   linspace(f,0,255);
   plotxy(f,Pyy,"Frequency domain filtered signal","frequency (Hz)","Magnitude (db)");
}
```

Output

This figure shows singals in the time domain. The figure on the left hand side is the original signals with three sinusoidal components at frequency of 5, 15, 30 Hz which are buried in white noises. The figure on the right hand side is signals after a sixth order IIR filter with a passband of 10 to 20 Hz. We hope to keep the 15 Hz sinusoidal signals and get rid of the 5 and 30 Hz sinusoids and other white noise using a filter with coefficients *u* and *v* given in the program.



The signals in the frequency domain are obtained by a fast Fourier transform algorthm using function **fft**(). This figure shows the original and filtered signals in the frequency domain. The figure on the left hand side shows the original signals with three separate main frequency spectrums and some noise frequency. After filtering, the signals, as shown on the right hand side of the figure , contain mainly a frequency of 15 Hz and a few noise components.



See Also
filter2().

References
Alon V. Oppenheim, *Discrete-time Signal Processing*, Prentice Hall, 1998.

filter2

Synopsis
#include <numeric.h>
int filter2(**array double complex** *y*[&][&], **array double complex** *u*[&][&], **array double complex** *x*[&][&],
... /* [**string_t** *method*] */);

Syntax
filter2(*y*, *u*, *x*)
filter2(*y*, *u*, *x*, *method*);

Purpose
Two-dimensional Discrete Fourier Transform based FIR filter.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y Two-dimensional **array** of filtered data.

u Two-dimensional **array** of size $nu \times mu$ of FIR filter.

x Two-dimensional **array** of size $nx \times mx$ of raw data.

method A string which describes the filter output method.

Description
Filter the data *x* with the 2-D FIR filter in the matrix *u*. By default, the result *y* is computed using 2-D convolution with the same size as *x*. If option *method* is specified, *y* is computed via 2-D convolution with size specified by "method":
"same" - (default) returns the central part of the convolution that is the same size as *x*.
"valid" - returns only those parts of the convolution that are computed without the zero-padded edges, the size of *y* is $(nx - nu + 1) \times (mx - mu + 1)$. The size of *x* must be bigger than the size of *u*.
"full" - returns the full 2-D convolution, that is the size of *y* is $(nx + nu - 1) \times (mx + mu - 1)$.

Algorithm
This function calls **conv2**(), the two-dimensional convolution function, to implement the filtering operation. **See conv2**().

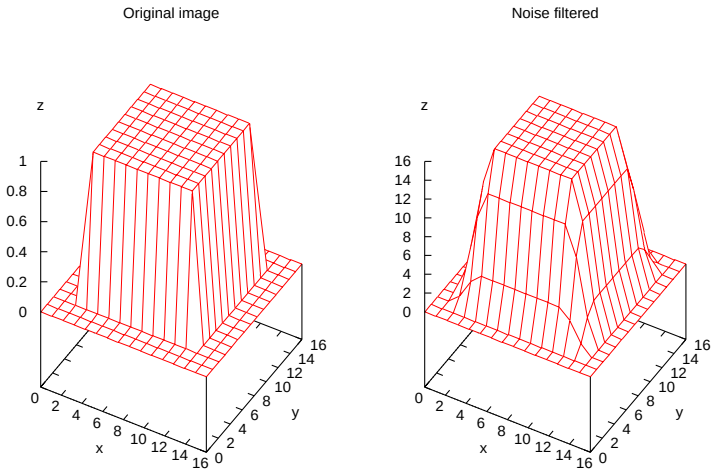
Example
In image processing, Sobel filter often used to smoothing image. Convoluting the original two-dimensional image data with Sobel mask

$$s = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The image can be smoothed.

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
   int i, j;
   array double u[3][3]={{1,2,1},{2,4,2},{1,2,1}};
   array double x[16],y[16],z1[256],z[16][16],Z[18][18],Z1[256];

   linspace(x,0,16);
   linspace(y,0,16);
   for(i=3; i<13; i++)
      for(j=3; j<13; j++) {
         z1[i*16+j]=1;
         z[i][j] = 1;
   }
   plotxyz(x,y,z1);
   filter2(Z,u,z,"full");
   for(i = 0; i<16; i++)
      for(j=0; j<16; j++)
         Z1[i*16+j] = Z[i+1][j+1];
   plotxyz(x,y,Z1);
}
```

Output



See Also
filter(), **conv2**().

findvalue()

Synopsis
#include <numeric.h>
int findvalue(array int *y[&]*, **array double complex** *&x*);

Syntax
findvalue(*y*, *x*)

Purpose
Obtain indices of nonzero elements of an array.

Return Value
This function returns the number of nonzero elements of an array.

Parameters
y A vector the size of the total number of elements of array *x*. It contains the indices of nonzero elements of array *x*.

x Any-dimensional input array which contains the original data.

Description
The original data **array** *x* can be of any supported arithmetic data type and extent sizes. The total number of nonzero elements is returned by the function call of **findvalue**(). Vector *y* is of integer data type with the total number of elements of *x*. It contains the indices of nonzero elements of array *x*. The remaining elements of *y* contain a value of -1. If *x* is an array of complex data type, a zero element has a value of zero for both its real and imaginary parts.

Example

```
#include <numeric.h>
int main() {
   array double x1[5]={0,43.4,-7,-2.478,7};
   array double x[4][2]={1,3,
                         2.45,8.56,
                         -3,5,
                         6,8};
   array double complex zx[3][2]={complex(1,1),0,
                         3,complex(4,1),
                         complex(1,0),complex(0,1)};
   array int zy[6], y1[5],y[8];
   int i,j;

   j=findvalue(y1,x1);
   printf("x1 = \n%f",x1);
   printf("Total %d of x1 are not eq. 0. They are located at y1 = \n",j);
   for (i=0; i<j; i++) printf("%d   ",y1[i]);
   printf("\n");
   j=findvalue(y,x<2.0);
   printf("x = \n%f",x);
   printf("Total %d of x are less than 2.0. They are located at y = \n",j);
```

```
   for (i=0; i<j; i++) printf("%d   ",y[i]);
   printf("\n");
   j=findvalue(zy,zx);
   printf("zx = \n%f",zx);
   printf("Total %d of zx are not eq. 0. They are located at zy = \n",j);
   for (i=0; i<j; i++) printf("%d   ",zy[i]);
   printf("\n");
}
```

Output

```
x1 =
0.000000 43.400000 -7.000000 -2.478000 7.000000
Total 4 of x1 are not eq. 0. They are located at y1 =
1   2   3   4
x =
1.000000 3.000000
2.450000 8.560000
-3.000000 5.000000
6.000000 8.000000
Total 2 of x are less than 2.0. They are located at y =
0   4
zx =
complex(1.000000,1.000000) complex(0.000000,0.000000)
complex(3.000000,0.000000) complex(4.000000,1.000000)
complex(1.000000,0.000000) complex(0.000000,1.000000)
Total 5 of zx are not eq. 0. They are located at zy =
0   2   3   4   5
```

fliplr()

Synopsis
#include <numeric.h>
int fliplr(array double complex *y[&][&]*, **array double complex** *x[&][&]*);

Syntax
fliplr(*y*, *x*)

Purpose
Flip matrix in left/right direction.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Two-dimensional matrix which contains the original data.

y Two-dimensional matrix of the same data type and size as *x*, which contains the flipped result of input matrix *x*.

Description
This function flips matrix *x* in the left/right direction with rows being preserved and columns flipped.

Example

```
#include <numeric.h>
int main() {
   array double x[2][4]={1,3, 2.45,8.56,
                         3,5, 6,8};
   array double complex zx[2][3]={complex(1,1),0,3,
                         complex(4,1), 6,0};
   array double complex zy[2][3];
   array double y[2][4];

   fliplr(y,x);
   printf("x = \n%f",x);
   printf("y = \n%f",y);
   printf("\n");

   fliplr(zy,zx);
   printf("zx = \n%5.1f",zx);
   printf("zy = \n%5.1f",zy);
}
```

Output

```
x =
1.000000 3.000000 2.450000 8.560000
3.000000 5.000000 6.000000 8.000000
y =
```

```
8.560000 2.450000 3.000000 1.000000
8.000000 6.000000 5.000000 3.000000

zx =
complex(  1.0,  1.0) complex(  0.0,  0.0) complex(  3.0,  0.0)
complex(  4.0,  1.0) complex(  6.0,  0.0) complex(  0.0,  0.0)
zy =
complex(  3.0,  0.0) complex(  0.0,  0.0) complex(  1.0,  1.0)
complex(  0.0,  0.0) complex(  6.0,  0.0) complex(  4.0,  1.0)
```

See Also
flipud(), **rot90**().

flipud()

Synopsis
#include <**numeric.h**>
int flipud(**array double complex** *y[&][&]*, **array double complex** *x[&][&]*);

Syntax
flipud(*y*, *x*)

Purpose
Flip matrix in up/down direction.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Two-dimensional matrix which contains the original data.

y Two-dimensional matrix of the same data type and size as *x*, which contains the flipped result of input matrix *x*.

Description
This function flips matrix *x* in the up/down direction with columns being preserved and rows flipped.

Example

```
#include <numeric.h>
int main() {
   array double x[4][2]={1,3,
                         2.45,8.56,
                         3,5,
                         6,8};
   array double complex zx[3][2]={complex(1,1),0,
                         3,complex(4,1),
                         6,0};
   array double complex zy[3][2];
   array double y[4][2];

   flipud(y,x);
   printf("x = \n%f",x);
   printf("y = \n%f",y);
   printf("\n");

   flipud(zy,zx);
   printf("zx = \n%5.1f",zx);
   printf("zy = \n%5.1f",zy);
}
```

Output

```
x =
```

```
1.000000 3.000000
2.450000 8.560000
3.000000 5.000000
6.000000 8.000000
y =
6.000000 8.000000
3.000000 5.000000
2.450000 8.560000
1.000000 3.000000

zx =
complex(  1.0,  1.0) complex(  0.0,  0.0)
complex(  3.0,  0.0) complex(  4.0,  1.0)
complex(  6.0,  0.0) complex(  0.0,  0.0)
zy =
complex(  6.0,  0.0) complex(  0.0,  0.0)
complex(  3.0,  0.0) complex(  4.0,  1.0)
complex(  1.0,  1.0) complex(  0.0,  0.0)
```

See Also
fliplr(), **rot90**().

fminimum

Synopsis
#include <**numeric.h**>
int fminimum (**double** **fminval*, **double** **xmin*, **double** (**func*)(**double**), **double** *x0*, **double** *xf*, ...
/* [**double** *rel_tol*, **double** *abs_tol*] */);

Syntax
fminimum(*fminval, xmin, func, x0, xf*)
fminimum(*fminval, xmin, func, x0, xf, rel_tol*)
fminimum(*fminval, xmin, func, x0, xf, rel_tolm, abs_tol*)

Purpose
Find the minimum value of a one-dimensional function and its corresponding position for the minimum value.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
fminval A pointer to the variable which passes the calculated minimum value of the function.

xmin A pointer to the variable which contains the calculated position with the minimum value of the function.

func The function given by user for minimization.

x0 The begining-point of the interval on which a minimum value is searched for.

xf The end-point of the interval on which a minimum value is searched for.

rel_tol The relative tolerance of the calculation.

abs_tol The absolute tolerance of the calculation.

Description
Determines a point between *x0* and *xf* at which the real function *func* assumes a minimum value. The function *func* given by the user has an argument for the *x* value input. The argument *fminval* passes the calculated minimum value of the function. The argument *xmin* contains the position where the minimum value of the function is found. The tolerance is defined as a function of x: $|x|rel\_tol + abs\_tol$, where *rel_tol* is the relative precision and *abs_tol* is the absolute precision which should not be chosen equal to zero. The default value for *rel_tol* and *abs_tol* is 10^{-6}.

Limitations
In the interval at which the function assumes a minimum value, it is assumed that for some point *u* either (a) *func* is strictly monotonically decreasing on [a,u) and strictly monotonically increasing on [u,b] or (b) these two intervals may be replaced by [a,u] and (u,b] respectively.

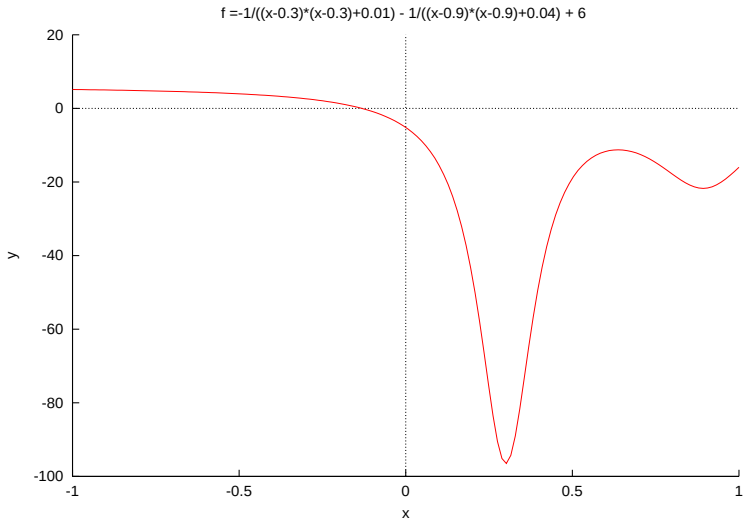
Example

Find the minimum of the function $f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} - 6$ at intervals [-1,0], [0,0.8], [0.4,1] and [0,1] with the default value for tolerance. For interval [0,1] this function can only find the local minimum at 0.892716. This example also finds the minimum of the function at the interval [0,0.8] with the given tolerance rel_tol and abs_tol equal to 0.01.

```
#include <numeric.h>
#include <stdio.h>

double func(double x) {
    double y;
    y = 1/((x-0.3)*(x-0.3)+0.01) + 1/((x-0.9)*(x-0.9)+0.04) - 6;
    return -y;
}

int main() {
    double x0, xf, rel_tol, abs_tol;
    double func(double);
    double xmin, fminval;
    int status;
    string_t title = "f =-1/((x-0.3)*(x-0.3)+0.01) - 1/((x-0.9)*(x-0.9)+0.04) + 6 ";

    fplotxy(func, -1, 1, 50, title, "x", "y");
    x0 = -1;  xf = 0; /* fminimum is at 0.0 */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    x0 = 0;  xf = 0.8;  /* fminimum is around 0.3 */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    rel_tol = 1E-2; /* default tolerance is 1E-6 */
    abs_tol = 1E-2; /* default tolerance is 1E-6 */
    x0 = 0;  xf = 0.8;  /* fminimum is around 0.6 */
    status = fminimum(&fminval, &xmin, func, x0, xf,rel_tol,abs_tol);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    x0 = 0.4;  xf = 1; /* fminimum is at 0.4 */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);

    x0 = 0;  xf = 1; /* Only local minimum at 0.892716 is found */
    status = fminimum(&fminval, &xmin, func, x0, xf);
    printf("Interval = [%3.2f, %3.2f]\n", x0, xf);
    printf("status = %d\n", status);
```

```
    printf("xmin = %f\n", xmin);
    printf("m = %f\n\n", fminval);
}
```

Output




```
Interval = [-1.00, 0.00]
status = 0
xmin = 0.000000
m = -5.176471

Interval = [0.00, 0.80]
status = 0
xmin = 0.300375
m = -96.501409

Interval = [0.00, 0.80]
status = 0
xmin = 0.305573
m = -96.232705

Interval = [0.40, 1.00]
status = 0
xmin = 0.400000
m = -47.448276

Interval = [0.00, 1.00]
status = 0
xmin = 0.892717
m = -21.734573
```

See Also

fminimums().

References

H. T. Lau, *A Numerical Library in C for Scientists and Engineers*, CRC Press, Inc. 1995.

fminimums

Synopsis
#include <**numeric.h**>
int fminimums(**double** **fminval*, **double** *xmin*[&], **double** (**func*)(**int**, **double**[ ]), **double** *x0*[&], ...
/*[**double** *rel_tol*, **double** *abs_tol*, **int** *numfuneval*] */);

Syntax
fminimums(*fminimumval, xmin, func, x0)*
fminimums(*fminimumval, xmin, func, x0, rel_tol)*
fminimums(*fminimumval, xmin, func, x0, rel_tol, abs_tol)*
fminimums(*fminimumval, xmin, func, x0, rel_tol, abs_tol, numfuneval)*

Purpose
Find the minimum position and value of an n-dimensional function.

Return Value
This function returns one of the following values: **Parameters**
0 On successful.

-1 The process is broken off because at the end of an iteration step the number of calls of *func* exceeded the value of *numfuneval*.

-2 The process is broken off because the condition of the problem is too bad.

-3 Function not supported.

Parameters
fminval A pointer to the variable which contains the calculated minimum value of the function.

xmin A pointer to the array which contains the calculated positions with the minimum value of the function.

func An n-dimensional function given by the user for minimization.

x0 An input array of approximation for variables *x* where the function gives the minimum value.

rel_tol The relative tolerance of the calculation.

abs_tol The absolute tolerance of the calculation.

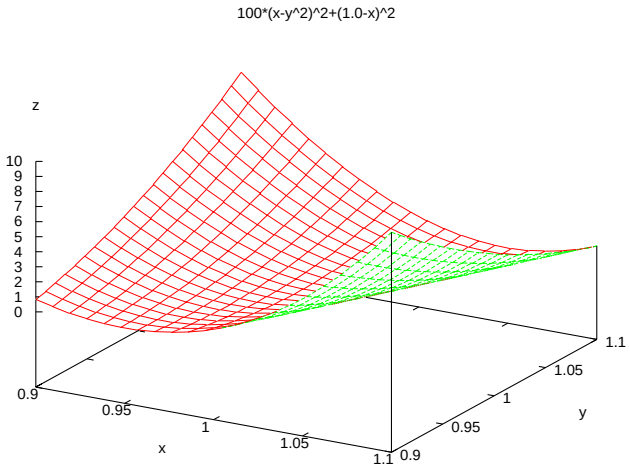
numfunceval The maximum number of function evaluation allowed.

Description
Given an n-dimensional function and an array which contains initial estimate by the user as input, this function calculates the array of the position at which the function has a minimum value. The number of dimension is taken from input array *x* internally. The function *func* should deliver the value of the function to be minimized, at the points given by *x*. The tolerance is defined as a function of *x* as $|x|rel\_tol + abs\_tol$, where *rel_tol* is the relative precision and *abs_tol* is the absolute precision which should not be chosen equal to zero. The argumnet *numfunceval* is the maximum number of function evaluations allowed. The default values for *rel_tol, abs_tol* and *numfuneval* are $10^{-6}, 10^{-6}$, and 250, respectively.

Example
Find the minimum value of two dimension function $100(x_1 - x_0^2)^2 + (1.0 - x_0)^2$ with the initial guess of $x_0 = -1.2$ and $x_1 = 1.0$.

```
#include <numeric.h>
#include <stdio.h>
#define N 2

double func(double x[]) {
    double temp;
    temp=x[1]-x[0]*x[0];
    return 100*temp*temp+(1.0-x[0])*(1.0-x[0]);
}

int main () {
    double func(double x[]);
    double fminval,xmin[N],x0[N],rel_tol, abs_tol;
    int numfuneval, status;
    x0[0] = -1.2;
    x0[1]=1.0;

    /* Plot a 2 dimensional graph of the function
       for which the minimum will be found */
    double funcp(double x, y)
    {
        double temp;
        temp=y-x*x;
        return 100*temp*temp+(1.0-x)*(1.0-x);
    }
    fplotxyz(funcp, 0.9, 1.1, 0.9, 1.1, 22,22,"100*(x-y*y)*(x-y*y)+(1.0-x)*(1.0-x)",
             "x","y","z");

    status = fminimums(&fminval, xmin, func,x0);
    printf("status = %d\n", status);
    printf("fminval = %f\n", fminval);
    printf("xmin[0]  = %f\n", xmin[0]);
    printf("xmin[1]  = %f\n", xmin[1]);

    rel_tol = 1.0e-3;   /* not default value */
    abs_tol = 1.0e-3;   /* not default value */
    numfuneval = 250;   /* not default value */
    status = fminimums(&fminval, xmin, func,x0,rel_tol, abs_tol,numfuneval);
    printf("status = %d\n", status);
    printf("fminval = %f\n", fminval);
    printf("xmin[0]  = %f\n", xmin[0]);
    printf("xmin[1]  = %f\n", xmin[1]);
}
```

Output




```
status = 0
fminval = 0.000000
xmin[0]  = 1.000000
xmin[1]  = 1.000000
status = 0
fminval = 0.000000
xmin[0]  = 0.999999
xmin[1]  = 1.000028
```

See Also
fminimum().

References
H. T. Lau, *A Numerical Library in C for Scientists and Engineers*, CRC Press, Inc. 1995.

fsolve

Synopsis
#include <numeric.h>
int fsolve (**double** *x*[&], **void**(**func*)(**double** *x*[ ], **double** *y*[ ]), **double** *x0*[&]);

Purpose
Find a zero position of a nonlinear system of equations.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x An n-dimensional array which contains the calculated zero position.

x0 An n-dimensional array which contains the initial guess for a zero position.

func A pointer to the function given by the user.

Description
Function **fsolve**() finds a zero position of function *func*(**x**) provided by the user. The user function has two arguments, the first one for input and the second one for output. The input argument is an n-dimensional array, and the function values calculated will be delivered by the second array argument of the same dimension as output. The number of dimension is taken from the function given by the user internally.

Algorithm
The multidimensional nonlinear system of equations is solved by a multidimensional secant method: Broyden's method. Since the Jacobian matrix which is needed for solution of the equations is approximated by Quisi-Newton method inside the function, the user does not need to provide analytic formula for derivatives of equations. Broyden's method converges superlinearly once you get close enough to the root.

Example
Solve the following nonlinear system of two equations

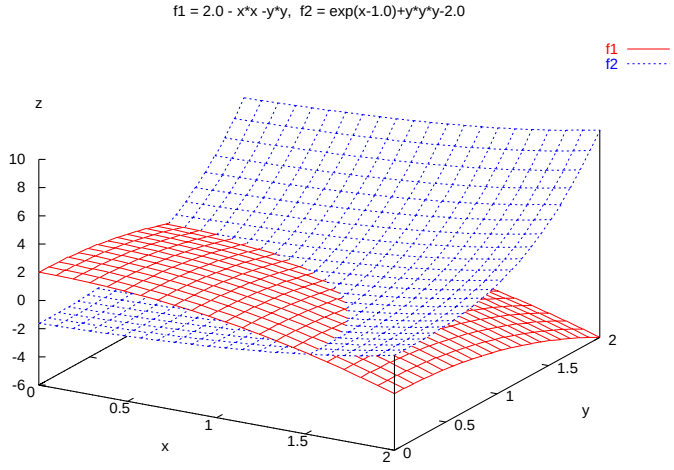
$$\begin{aligned} f_0 &= -(x_0^2 + x_1^2 - 2.0) = 0 \\ f_1 &= e^{x_0 - 1.0} + x_1^3 - 2.0 = 0 \end{aligned}$$

with the initial guesses of zero point at $x_0 = 2.0$, and $x_1 = 0.5$.

```
/* Driver for routine fsolve: for 2 dimentional equation*/
#include <stdio.h>
#include <numeric.h>
#include <chplot.h>
#define N 2
#define NP 22

void func(double x[], double f[]){
    f[0]=-(x[0]*x[0]+x[1]*x[1]-2.0);
    f[1]=exp(x[0]-1.0)+x[1]*x[1]*x[1]-2.0;
}
```

```
int main(){
    double x[N], x0[N], f[N];
    double x1[NP], x2[NP], f1[NP*NP], f2[NP*NP];
    int i,j,status;
    class CPlot plot;

    linspace(x1, 0, 2);
    linspace(x2, 0, 2);

    for(i=0; i<NP; i++) {
        for(j=0; j<NP; j++){
            f1[NP*i+j] = -(x1[i]*x1[i]+x2[j]*x2[j] - 2.0);
            f2[NP*i+j] = exp(x1[i]-1.0) + x2[j]*x2[j]*x2[j]-2.0;
        }
    }
    plot.data3D(x1, x2, f1);
    plot.data3D(x1, x2, f2);
    plot.legend("f1", 0);
    plot.legend("f2", 1);
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");
    plot.label(PLOT_AXIS_Z, "z");
    plot.title("f1 = 2.0 - x*x -y*y,  f2 = exp(x-1.0)+y*y*y-2.0");
    plot.ticsLevel(0);
    plot.plotting();

    x0[0]=2.0;
    x0[1]=0.5;
    status=fsolve(x, func, x0);

    func(x,f);

    printf("status = %d\n",status);
    if (status==-1)
        fprintf(stderr, "Convergence problems.\n");
    printf("%7s %3s %12s\n","Index","x","f");
    for (i=0;i<2;i++)
        printf("%5d %12.6f %12.6f\n",i,x[i],f[i]);
}
```

Output




```
status = 0
  Index    x            f
    0      1.000016    -0.000020
    1      0.999994    -0.000002
```

See Also
fzero().

References
William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

funm

Synopsis
#include <**numeric.h**>
int funm(**array double** *y[&][&]*, **double** (**func*)(**double**), **array double** *x[&][&]*);

Syntax
funm(*y*, *func*, *x*)

Purpose
Evaluate general real matrix function.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input square matrix. It contains data to be evaluated.

func A function routine given by the user.

y Output square matrix which contains data of the calculated function values.

Description
This function evaluates the matrix version of the function specified by argument *func*. In this function, The input matrix *x* shall be **double** data type and the specified function prototype shall be **double** *func*(**double**). The output matrix *y* could be **real** or **complex** data type as required.

Example
A real matrix evaluation.

```
#include <numeric.h>
double mylog(double x) {
   return log(x);
}

int main() {
   array double x[3][3]={1,2,3,
                         3,4,5,
                         6,7,8};
   array double y[3][3];

   expm(y,x);
   printf("x = \n%f",x);
   printf("y = \n%f",y);
   funm(x,mylog,y);
   printf("x = \n%f",x);

}
```

Output

```
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y =
157372.953093 200034.605129 242697.257164
290995.910241 369883.552084 448769.193928
491431.845963 624654.472518 757878.099072
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
```

See Also

expm(), **logm**(), **cfunm**(), **sqrtm**().

References

G. H. Golub, C. F. Van Loan, Matrix Computations Third edition, The Johns Hopkins University Press, 1996

fzero

Synopsis

#include <numeric.h>

int fzero(**double** *x, **double**(**func*)(**double** *x*), ... /* [**double** *x0*] | [**double** *x02*[2]*/);

Purpose

Find a zero position of a nonlinear function with one variable.

Return Value

This function returns 0 on success and -1 on failure.

Parameters

x Output of the calculated zero position.

func A pointer to the function given by the user.

x0 Input of the initial guess for zero position.

x02 Input of a vector of length 2 and double type. The function shall be bracketed in the interval of [*x02*[0], *x02*[1]] so that the sign of *func*(*x02*[0]) differs from the sign of *func*(*x02*[1]). Otherwise, an error occurs.

Description

Function **fzero**() finds a zero position of function *func*(*x*) provided by the user. The input argument of function *func*() is *x* value.

Algorithm

If x0 is given, the algorithm of one-dimensional zero finding is based on the algorithm for finding zero of the multi-dimensional nonlinear system of equations. Based on **fsolve**(), the number of dimension is set to 1 for **fzero**(). See algorithm for **fsolve**(). If x02 is given, a bisection method is used to find the zero position.

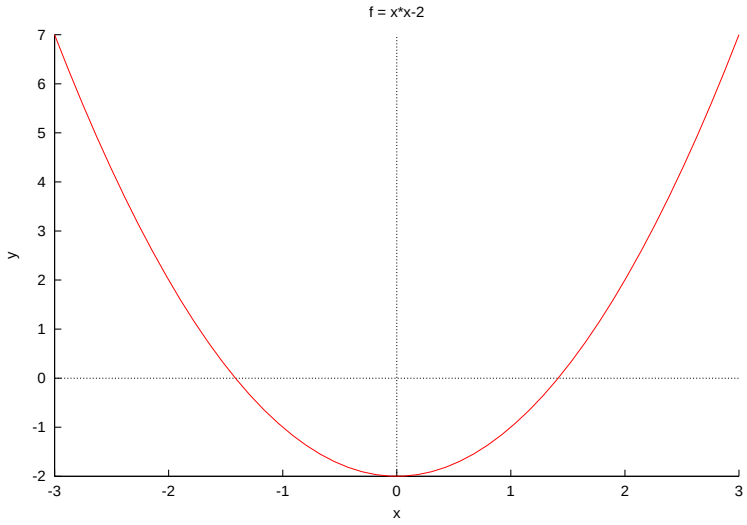
Example

Find zero of function $f = x^2 - 2$ with initial guess $x = 2.0$.

```
#include <stdio.h>
#include <chplot.h>
#include <numeric.h>

double func(double x) {
    return x*x-2.0;
}

int main() {
    double x, x0, f, x02[2];
    int i,status;
    double func(double);
    fplotxy(func, -3, 3, 50, "f = x*x-2", "x", "y");

    x0=-2.0;
    status=fzero(&x, func, x0); // x0 is a scalar
    f=func(x);
    if (status<0)
        printf("fzero() failed.\n");
    printf(" %6s %12s\n","x","f");
```

```
    printf("%10.6f %12.6f\n",x,f);

    x02[0]=-2.0;
    x02[1]=0;
    status=fzero(&x, func, x02); // x02 is an array
    f=func(x);
    if (status<0)
        printf("fzero() failed.\n");
    printf(" %6s %12s\n","x","f");
    printf("%10.6f %12.6f\n",x,f);
}
```

Output




```
     x            f
 1.414213    -0.000000
     x            f
-1.414213    -0.000000
```

See Also
fsolve().

References
William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

gcd()

Synopsis
#include <numeric.h>
int gcd(array int *&u*, **array int** *&v*, **array int** *&g*, ... /* [**array int** *c[&]*, **array int** *d[&]*]*/);

Syntax
gcd(*u*, *v*, *g*)
gcd(*u*, *v*, *g*, *c*, *d*)

Purpose
Obtain the greatest common divisor of the corresponding elements of two arrays of integer type.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
u Any-dimensional array which contains non-negative integer elements for the greatest common divisor calculation.

v An array the same dimension and size as *u*. It contains another non-negative integer elements for the greatest common divisor calculation.

g An output array the same dimension and size as *u*. It contains the result of the greatest common divisor calculation of *u* and *v*.

c An output array the same dimension and size as *u*.

d An output array the same dimension and size as *u*.

Description
This function calculates the greatest common divisor of corresponding elements *u* and *v*. The arrays *u* and *v* shall be the same size and contain non-negative integer data. The output array *g* is a positive integer array of the same size as *u*. The optional output arrays *c* and *d* are of the same size as *u* and satisfy the equation $u. * c + v. * d = g$.

Example

```
#include <numeric.h>
int main() {
   array int u[2][3] = {1,2,7,15,3,4};
   array int v[2][3] = {2,4,8,3,8,3};
   array int g[2][3],c[2][3],d[2][3];
   gcd(u,v,g,c,d);
   printf("g=%d",g);
   printf("c=%d",c);
   printf("d=%d",d);
   printf("u.*c+v.*d = %d",u.*c+v.*d);
}
```

Output

```
g=1 2 1
3 1 1
c=1 1 -1
0 3 1
d=0 0 1
1 -1 -1
u.*c+v.*d = 1 2 1
3 1 1
```

See Also
lcm().

References
Knuth, Donald Ervin, *The art of computer programming*, Vol. 2, Addison-Wesley Pub. Co., 1973.

getnum

Synopsis
#include <**numeric.h**>
double getnum(**string_t** *msg*, **double** *d*);

Purpose
Obtain a number from the console through standard input stream *stdin*.

Return Value
This function returns the default number or input number from the console.

Parameters
msg Message printed out in the standard output stream *stdout*.

d Default number.

Description
This function returns the default number when carriage RETURN is entered as input or new number from *stdin* as a double-precision floating-point number. However, if an invalid number is entered, a number is requested. The message in string *msg* will be printed out.
Example

```
#include <numeric.h>
/* The input numbers typed in on the console are 100 200 */

int main() {
    double d=2.0;

    printf("\nEnter a number [%lf]: ",d);
    d = getnum(NULL, d);
    printf("\nThe number you entered: %lf\n\n",d);
    d = getnum("Please enter a number[10.0]: ", 10);
    printf("\nThe number you entered: %lf\n\n",d);
}
```

Output

```
Enter a number [2.000000]:
The number you entered: 100.000000

Please enter a number[10.0]:

The number you entered: 200.000000
```

See Also
getline().

References

hessdecomp

Synopsis
#include <numeric.h>
int hessdecomp(**array double complex** *a*[&][&], **array double complex** *h*[&][&], ...
/* [**array double complex** *p*[&][&]] */);

Syntax
hessdecomp(*a*, *h*);
hessdecomp(*a*, *h*, *p*);

Purpose
Reduces a real general matrix *a* to upper Hessenberg form *h* by an orthogonal/unitary matrix *p* similarity transformation: $p^T * a * p = h$ or $p^H * a * p = h$ for real and complex, respectively. The p^H is Hermitian of matrix *p*.

Return Value
This function returns 0 on success and negative value on failure.

Parameters
a A $n \times n$ square matrix to be decomposed.

h An output two-dimensional matrix which contains the upper Hessenberg matrix of matrix *a*.

p An optional output two-dimensional array which contains an orthogonal or unitary matrix.

Description
This function computes the Hessenberg matrix *h* and an orthogonal/unitary matrix *p* so that $h = p^T * a * p$ and $p^T * p = I$ for real matrix, and $h = p^H * a * p$ and $p^H * p = I$ for complex matrix. Each element of a Hessenberg matrix below the first subdiagonal is zero. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original one, but less computation is needed to calculate them.
In this function, square matrix *a* could be any supported arithmetic data type. Output *h* is the same dimension and data type as input *a*. If the input *a* is of real type, the optional output *p* shall only be double type. If the input *a* is complex type, *p* shall be **complex** or **double complex** type.

Example1
Reduce a real general matrix to upper Hessenberg form.

```
#include <numeric.h>
int main() {
     int m = 3;
     array double a[3][3] = { 0.8,0.2,0.1,
                              0.1,0.7,0.3,
                              0.1,0.1,0.6};

     int status;
     array double p[m][m], h[m][m];
     status = hessdecomp(a, h);
     if (status == 0) {
        printf("h =\n%f\n", h);
```

```
     }
     else
        printf(" Hessenberg matrix calculation error.\n");
     status = hessdecomp(a, h, p);
     if (status == 0) {
        printf("h =\n%f\n", h);
        printf("p =\n%f\n", p);
        printf("transpose(p)*p =\n%f\n",transpose(p)*p );
        printf("transpose(p)*a*p - h =\n%f\n", transpose(p)*a*p - h);
     }
     else
        printf(" Hessenberg matrix calculation error.\n");
}
```

Output1

```
h =
0.800000 -0.212132 -0.070711
-0.141421 0.850000 -0.050000
0.000000 0.150000 0.450000

h =
0.800000 -0.212132 -0.070711
-0.141421 0.850000 -0.050000
0.000000 0.150000 0.450000

p =
1.000000 0.000000 0.000000
0.000000 -0.707107 -0.707107
0.000000 -0.707107 0.707107

transpose(p)*p =
1.000000 0.000000 0.000000
0.000000 1.000000 -0.000000
0.000000 -0.000000 1.000000

transpose(p)*a*p - h =
0.000000 -0.000000 -0.000000
0.000000 0.000000 -0.000000
0.000000 0.000000 0.000000
```

Example2
Reduce a complex general matrix to upper Hessenberg form.

```
#include <numeric.h>
int main() {
     int m = 3;

     array double complex a[3][3] = { complex(-149,1), -50, -154,
                              537,  180,  546,
                              -27,   -9,  -25};
     int status;
     array double complex h[m][m];
     array double complex p[m][m];
     status = hessdecomp(a, h);
     if (status == 0) {
        printf("h =\n%5.2f\n", h);
     }
```

```
     else
        printf("Hessenberg matrix calculation error.\n");
     status = hessdecomp(a, h, p);
     if (status == 0) {
        printf("h =\n%5.2f\n", h);
        printf("p =\n%5.2f\n", p);
        printf("transpose(p)*p =\n%5.2f\n", transpose(p)*p);
        printf("conj(transpose(p))*a*p - h =\n%5.2f\n", conj(transpose(p)*a*p - h));
     }
     else
        printf("Hessenberg matrix calculation error.\n");

}
```

Output2

```
h =
complex(-149.00, 1.00) complex(42.20, 0.00) complex(-156.32, 0.00)
complex(-537.68, 0.00) complex(152.55, 0.00) complex(-554.93, 0.00)
complex( 0.00, 0.00) complex( 0.07, 0.00) complex( 2.45, 0.00)

h =
complex(-149.00, 1.00) complex(42.20, 0.00) complex(-156.32, 0.00)
complex(-537.68, 0.00) complex(152.55, 0.00) complex(-554.93, 0.00)
complex( 0.00, 0.00) complex( 0.07, 0.00) complex( 2.45, 0.00)

p =
complex( 1.00, 0.00) complex( 0.00, 0.00) complex( 0.00, 0.00)
complex( 0.00, 0.00) complex(-1.00, 0.00) complex( 0.05, 0.00)
complex( 0.00, 0.00) complex( 0.05, 0.00) complex( 1.00, 0.00)

transpose(p)*p =
complex( 1.00, 0.00) complex( 0.00, 0.00) complex( 0.00, 0.00)
complex( 0.00, 0.00) complex( 1.00, 0.00) complex( 0.00, 0.00)
complex( 0.00, 0.00) complex( 0.00, 0.00) complex( 1.00, 0.00)

conj(transpose(p))*a*p - h =
complex( 0.00,-0.00) complex( 0.00,-0.00) complex( 0.00,-0.00)
complex( 0.00,-0.00) complex( 0.00,-0.00) complex( 0.00,-0.00)
complex( 0.00,-0.00) complex( 0.00,-0.00) complex(-0.00,-0.00)
```

See Also
eigensystem().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

histogram

Synopsis
#include <**numeric.h**>
int histogram(**array double** &*y*, **array double** *x*[&], ... /* [**array double** *hist*[:]] */);

Syntax
histogram(*y, x*)
histogram(*y, x, hist*)

Purpose
Calculate and plot histograms.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y An input array of data set.

x An input array which contains the bins of the histogram.

hist An output array which contains the calculated data of the histogram.

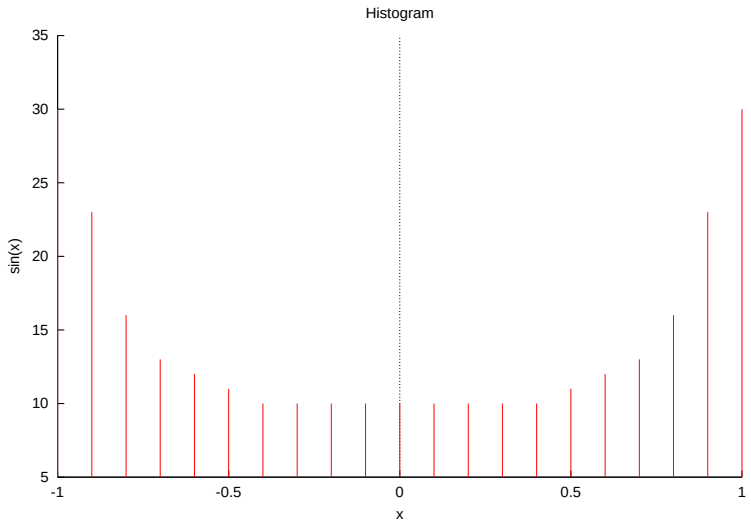
Description
This function calculates and plots the histogram of data set *y* with the bins defined in array *x*. The array of data set *y* can be of any dimensions. If the function is called without the optional argument *hist*, the histogram will be plotted. Otherwise the function only outputs the data of histogram without plotting.

Example1
The histograms of the data set generated from function $\sin(y)$ are calculated and plotted in this example. Note that the histograms are the same for the data set *y1* and *y2* due to the symmetric nature of the $\sin()$ function, even though their periods of the data are different.

```
#define N 300
#define N1 10
#define N2 30
#define M 21

#include<numeric.h>
int main() {
  array double y1[N], x[M], hist[M];
  array double y2[N1][N2];

  linspace(x, -1, 1);
  linspace(y1, 0, 2*M_PI);
  linspace(y2, 0, 4*M_PI);

  y1 = sin(y1);
  y2 = sin(y2);
  histogram(y1, x); // chplot.histogram
  histogram(y2, x); // chplot.histogram

  histogram(y1, x, hist);
```

```
  printf("x hist = %f %f\n", x, hist);
  histogram(y2, x, hist);
  printf("x hist = %f %f\n", x, hist);
}
```

Output1




```
x hist = -1.000000 -0.900000 -0.800000 -0.700000 -0.600000 -0.500000 -0.400000 -0.300000
-0.200000 -0.100000 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000
0.700000 0.800000 0.900000 1.000000
 30.000000 23.000000 16.000000 13.000000 12.000000 11.000000 10.000000 10.000000
10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 11.000000
12.000000 13.000000 16.000000 23.000000 30.000000

x hist = -1.000000 -0.900000 -0.800000 -0.700000 -0.600000 -0.500000 -0.400000 -0.300000
-0.200000 -0.100000 0.000000 0.100000 0.200000 0.300000 0.400000 0.500000 0.600000
0.700000 0.800000 0.900000 1.000000
 30.000000 23.000000 16.000000 13.000000 12.000000 11.000000 10.000000 10.000000
10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 11.000000
12.000000 13.000000 16.000000 23.000000 30.000000
```

Example2

The histograms of the same data sets *y1* and *y2* as in example 1 are calculated in this example. The histograms ara drawn by function **plotxy**() according to the data calculated by function **histogram**(). The histograms are the same as that of example 1.

```
#define N 300
#define N1 10
#define N2 30
#define M 21

#include <numeric.h>
#include <chplot.h>
int main() {
  array double y1[N], x[M], hist[M];
  array double y2[N1][N2];
```

```
  class CPlot plot;

  linspace(x, -1, 1);
  linspace(y1, 0, 2*M_PI);
  linspace(y2, 0, 4*M_PI);

  y1 = sin(y1);
  y2 = sin(y2);

  histogram(y1, x, hist);
  plotxy(x, hist, "Histogram", "x", "sin(x)", &plot);
  plot.axisRange(PLOT_AXIS_Y, min(hist)-5, max(hist)+5);
  plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
  plot.plotting();

  histogram(y2, x, hist);
  plotxy(x, hist, "Histogram", "x", "sin(x)", &plot);
  plot.axisRange(PLOT_AXIS_Y, min(hist)-5, max(hist)+5);
  plot.plotType(PLOT_PLOTTYPE_IMPULSES, 0);
  plot.plotting();
}
```

Output2

The output histograms are the same as that of example 1.

See Also

plotxy().

References

householdermatrix

Synopsis
#include <numeric.h>
int householdermatrix(**array double complex** *x*[&], **array double complex** *v*[&], ...
/* [**double** **beta*] */);

Syntax
householdermatrix(*x, v*)
householdermatrix(*x, v, beta*)

Purpose
Get the Householder matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input vetctor of n elements.

v Output vector of n elements.

beta Optional output value of pointer to double.

Description
This function passes a vector *x* as an input argument and gets the vector *v* and optional output value *beta* such that

$$H = I - \text{beta} * v * v^T$$

where H is a Householder matrix, and I is an identity matrix. A Householder matrix H satisfies the equation

$$H * x = -\text{sign}(x[0]) * \text{norm}(x) * E$$

where vector $E = [1, 0, 0, \cdots, 0]$ has n elements. If *x* is a complex vector, then sign(x[0]) is defined

$$\text{sign}(x[0]) = \frac{x[0]}{\text{abs}(x[0])}$$

Example

```
#include <numeric.h>
int main() {
   array double complex x[5] = {complex(-0.3, 0.5),54,25.3,25.46,83.47};
   array double complex y[5], h[5][5];
   array double x1[5] = {-0.3,54,25.3,25.46,83.47};
   array double e[5]={1,0,0,0,0};
   array double y1[5], h1[5][5];
   double beta;

   householdermatrix(x1,y1);
   printf("x1=\n%5.2f",x1);
```

```
   printf("y1=\n%5.2f",y1);
   householdermatrix(x1,y1,&beta);
   h1 = identitymatrix(5) - beta*y1*transpose(y1);
   printf("y1=\n%5.2f",y1);
   printf("beta=%5.2f\n",beta);
   printf("h1*x1+sign(x1[0])*norm(x1)*e =\n%5.2f",h1*x1+sign(x1[0])*norm(x1,"2")*e);

   householdermatrix(x,y);
   printf("x=\n%5.2f",x);
   printf("y=\n%5.2f",y);
   householdermatrix(x,y,&beta);
   h = identitymatrix(5) - beta*y*conj(transpose(y));
   printf("y=\n%5.2f",y);
   printf("beta=%5.2f\n",beta);
   printf("h*x+sign(x[0])*norm(x)*e =\n%5.2f",h*x+x[0]/abs(x[0])*norm(x,"2")*e);
}
```

Output

```
x1=
-0.30 54.00 25.30 25.46 83.47
y1=
-106.00 54.00 25.30 25.46 83.47
y1=
-106.00 54.00 25.30 25.46 83.47
beta= 0.00
h1*x1+sign(x1[0])*norm(x1)*e =
 0.00  0.00  0.00 -0.00 -0.00
x=
complex(-0.30, 0.50) complex(54.00, 0.00) complex(25.30, 0.00) complex(25.46, 0.00)
complex(83.47, 0.00)
y=
complex(-54.68,91.13) complex(54.00, 0.00) complex(25.30, 0.00) complex(25.46, 0.00)
complex(83.47, 0.00)
y=
complex(-54.68,91.13) complex(54.00, 0.00) complex(25.30, 0.00) complex(25.46, 0.00)
complex(83.47, 0.00)
beta= 0.00
h*x+sign(x[0])*norm(x)*e =
complex( 0.00, 0.00) complex( 0.00,-0.00) complex( 0.00,-0.00) complex( 0.00,-0.00)
complex(-0.00,-0.00)
```

References

G. H. Golub and C. F. van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press, Baltimore, Maryland, 1996

identitymatrix

Synopsis
#include <**numeric.h**>
array double identitymatrix(**int** *n*)[:][:];

Purpose
Generate an identity matrix.

Return Value
This function returns the identity matrix.

Parameters
n Input integer.

Description
The function returns an $n \times n$ indentity matrix.

Example

```
#include <numeric.h>
int main() {
    int n = 3;
    array double a[n][n];

    a = identitymatrix(n);
    printf("identitymatrix(n) =\n%f\n", a);
}
```

Output

```
identitymatrix(n) =
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

See Also
rank().

References

ifft

Synopsis
#include <numeric.h>
int ifft(**array double complex** &*y*, **array double complex** &*x*, ... /* [**int** *n* [**int** *dim*[&]]] */);

Syntax
ifft(*y*, *x*)
ifft(*y*, *x*, *n*)
ifft(*y*, *x*, *dim*);

Purpose
N-dimensional inverse Fast Fourier Transform (FFT) calculation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y Result of inverse FFT, the array of the same dimension and size as *x*.

x An n-dimensional array used for calculating the inverse FFT.

n Optional argument, user specified inverse FFT points for one-dimensional data.

dim A one-dimensional array of optional arguments with int type. It contains the user specified inverse FFT points. Every element corresponds to a data dimension. For example, there is a three-dimensional data *x[M][N][L]*. The FFT points are specified m, n, l, respectively. Then the array *dim* is given values of $\text{dim}[0] = m, \text{dim}[1] = n, \text{dim}[2] = l$.

Description
The multi-dimensional **array** *x* can be of any supported arithmetic data type and size. Conversion of the data to **double complex** is performed internally. The same multi-dimensional, **double complex array** *y* contains the result of inverse fast Fourier transform. The result normalized by the total number of points of the transform is performed internally. The optional argument *n* or *dim*[] is used for specifying the number of points of the inverse FFT. If the user does not specify the number of points of FFT, it will be initialized according to the length of the input data.

Algorithm
see fft().

Example
see fft().

See Also
fft().

References

R. C. Singleton, *An Algorithm for Computing the Mixed Radix F. F. T.*, IEEE Trans, Audio Electroacoust., AU-1(1969) 93-107.

MJ Oleson, *Netlib Repository at UTK and ORNL*, go/fft-olesen.tar.gz, http://www.netlib.org

integral1

Synopsis
#include <**numeric.h**>
double integral1(**double** (**func*)(**double**), **double** *x1*, **double** *x2*, ... /* [**double** *tol*] */);

Syntax
integral1(*func*, *x1*, *x2*)
integral1(*func*, *x1*, *x2*, *tol*)

Purpose
Numerical integration of a function.

Return Value
This function returns an integral value of a function *func* integrated from *x1* to *x2*.

Parameters
func Function to be integrated.

x1 An initial point of integral.

x2 A final point of integral.

tol The user specified tolerance. If the user does not specify this optional value, the value of $10*$FLT_EPSILON is used by default, where FLT_EPSILON is defined in header file **float.h**.

Description
The function to be integrated, *func*, is specified as a pointer to a function that takes a **double** as an argument and returns a **double** functional value. *x1* and *x2* are the end-points of the range to be integrated. Conversion of the data to **double** are performed internally. The return value is a **double** data of integral evaluation. If the optional argument *tol* is specified, the integral uses the user specified tolerance to decide iteration stop. Otherwise the default value is used.

Algorithm
The following trapezoidal algorithm for numerical integration of func $f(x)$ is used

$$\int_{x_1}^{x_2} f(x)dx = h[\frac{1}{2}f_1 + \frac{1}{2}f_2] + O(h^3 f'')$$

where f'' represents the second derivative of the function *f*(*x*) evaluated at a point within $x_1 \leq x \leq x_2$.

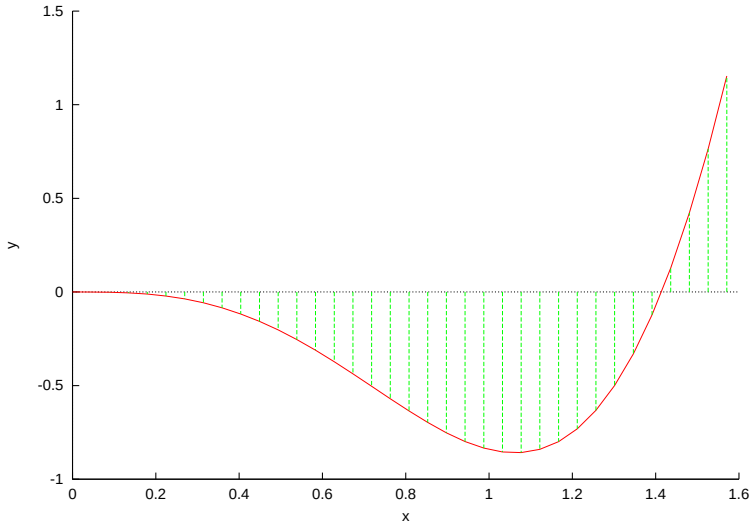
Example
Evaluate the integration

$$\int_0^{\frac{\pi}{2}} x^2(x^2-2)\sin(x)dx$$

```
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

/* Test function */
double func(double x) {
   return x*x*(x*x-2.0)*sin(x);
}

/* Integral of test function */
double fint(double x) {
   return 4.0*x*(x*x-7.0)*sin(x)-(pow(x,4.0)-14.0*x*x+28.0)*cos(x);
}

int main() {
   double x1=0.0,x2=M_PI/2,s;
   int numpoints = 36;
   array double x[numpoints], y[numpoints];
   class CPlot plot;

   linspace(x, x1, x2);
   fevalarray(y, func, x);
   printf("Actual value of the integration is %12.6lf\n",fint(x2)-fint(x1));
   s=integral1(func,x1,x2);
   printf("Result from the function integral1 is %11.6lf\n",s);

   plot.data2D(x, y);
   plot.data2D(x, y);
   plot.plotType(PLOT_PLOTTYPE_IMPULSES, 1);
   plot.plotting();
}
```

Output

```
Actual value of integraion is    -0.479159
Result from the function integration is   -0.479158
```



see also
integral2(), **integral3**(), **integration2**(), **integration3**(), **derivative**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integral2

Synopsis
#include <numeric.h>
double integral2(**double** (**func*)(**double** *x*, **double** *y*), **double** *x1*, **double** *x2* **double** *y1*, **double** *y2*);

Syntax
integral2(*func*, *x1*, *x2*, *y1*, *y2*)

Purpose
Numerical integration of a function of two variables with constant limits for both *x* and *y*.

Return Value
This function returns an integral value of a function *func* integrated from $y1$ to $y2$ in *y* and *x1* to *x2* in *x*.

Parameters
func Function to be integrated.

x1 The lower limit of integration in *x*.

x2 The upper limit of integration in *x*.

y1 The lower limit of integration in *y*.

y2 The upper limit of integration in *y*.

Description
The function to be integrated, *func*, is specified as a pointer to a two-dimemsional function that takes two arguments and returns the functional value of double type. The values *x1* and *x2* are the end-points of the range in *x* to be integrated. The values $y1$ and $y2$ are the upper and lower limits in *y*, respectively. The return value is the integral of double type.

Algorithm
The integration is carried out by (1) its lower and upper limits in *y* denoted as *y1* and *y2*; (2) its lower and upper limits in *x* denoted as *x1* and *x2*, respectively, so that

$$I = \int\int dxdyf(x,y) = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dyf(x,y)$$

Example
Integrate $f(x,y) = \sin(x)\cos(y) + 1$ over the limits $0 \le x \le \pi$ and $-\pi \le y \le \pi$ by

$$I = \int_0^{\pi} \int_{-\pi}^{\pi} (\sin(x) * \cos(y) + 1)dxdy$$

```
#include <math.h>
#include <numeric.h>

double func(double x,double y) {
    return sin(x)*cos(y)+1;
```

```
}
int main() {
    double x1 = 0, x2 = M_PI;
    double y1 = -M_PI, y2 = M_PI;
    double v;
    v = integral2(func,x1, x2, y1,y2);
    printf("integral2() gives %f\n", v);
}
```

Output

```
integral2() gives 19.739212
```

see also
integral1(), **integral3**(), **integration2**(), **integration3**(), **derivative**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integral3

Synopsis
#include <**numeric.h**>
double integral3(**double** (**func*)(**double** *x*, **double** *y*, **double** it z), **double** *x1*, **double** *x2*, **double** *y1*,
double *y2*, **double** *z1*, **double** *z2*);

Syntax
integration3(*func*, *x1*, *x2*, *y1*, *y2*, *z1*, *z2*)

Purpose
Numerical integration of a function of three-demensions with constant limits in *x*, *y* and *z*.

Return Value
This function returns an integral value of a function *func* integrated from *z1* to *z2* in *z*, *y1* to *y2* in *y* and *x1* to *x2* in *x*.

Parameters
func Function to be integrated.

x1 The lower limit of integration in *x*.

x2 The upper limit of integration in *x*.

y1 The lower limit of integration in *y*.

y2 The upper limit of integration in *y*.

z1 The lower limit of integration in *z*.

z2 The upper limit of integration in *z*.

Description
The function to be integrated, *func*, is specified as a pointer to a three-dimemsional function that takes three arguments and returns a functional value of double type. The values $y1$ and $y2$ are the upper and lower limits in *y*, the values $z1$ and $z2$ are the upper and lower limits in *z*, and the values *x1* and *x2* are the end-points of the range in *x* to be integrated, respectively . The return value is the integral of double type.

Algorithm
The integration is carried out by (1) its lower and upper limits in *z*, denoted *z1* and *z2*, (2)its lower and upper limits in *y*, denoted *y1* and *y2*; and (3)its lower and upper limits in *x*, which we will denote *x1* and *x2*, so that

$$I = \int\int\int dxdydzf(x,y,z) = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy \int_{z_1}^{z_2} dzf(x,y,z)$$

Example
Integrate $f(x,y,z) = \sin(x)\cos(y)\sin(z) + 1$ over the limits of $0 \leq x \leq \pi$, $-\pi \leq y \leq \pi$ and $0 \leq z \leq \pi$.

$$I = \int_0^{\pi} \int_{-\pi}^{\pi} \int_0^{\pi} (\sin(x)\cos(y)\sin(z) + 1)dxdydz$$

```
#include <math.h>
#include <numeric.h>

double func(double x,double y, double z) {
    return sin(x)*cos(y)*sin(z)+1;
}
int main() {
    double x1 = 0, x2 = M_PI;
    double y1 = -M_PI, y2 = M_PI;
    double z1 = 0, z2 = M_PI;
    double v;
    v = integral3(func, x1,x2, y1, y2, z1, z2);
    printf("integral3() gives %f\n", v);
}
```

Output

```
integral3() gives 62.012571
```

see also
integral1(), **integral2**(), **integral3**(), **integration2**(), **integration3**(), **derivative**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integration2

Synopsis
#include <**numeric.h**>
double integration2(**double** (******func*)(**double** *x*, **double** *y*), **double** *x1*, **double** *x2* **double** (******y1*)(**double** *x*), **double** (******y2*)(**double** *x*));

Syntax
integration2(*func*, *x1*, *x2*, *y1*, *y2*)

Purpose
Numerical integration of a function of two variables.

Return Value
This function returns an integral value of a function *func* integrated from $y_1(x)$ to $y_2(x)$ and *x1* to *x2* in *x*.

Parameters
func Function to be integrated.

x1 Function denated to the lower limit of integration in *x*.

x2 Function denated to the upper limit of integration in *x*.

y1 Function denated to the lower limit of integration in *y*.

y2 Function denated to the upper limit of integration in *y*.

Description
The function to be integrated, *func*, is specified as a pointer to a two-dimemsional function that takes two arguments and returns the functional value of double type. The values *x1* and *x2* are the end-points of the range in *x* to be integrated. The values $y_1(x)$ and $y_2(x)$ are user-supplied functions for the upper and lower limits in *y*. The return value is the integral of double type.

Algorithm
The integration is carried out by (1) its lower and upper limits in *y*, denoted as *y1(x)* and *y2(x)*; (2) its lower and upper limits in *x*, denoted as *x1* and *x2*, respectively, so that

$$I = \int\int dxdyf(x,y) = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dyf(x,y)$$

Now we can define a function *H(x,y)* that does the innermost integral,

$$H(x,y) = \int_{y_1(x)}^{y_2(x)} f(x,y)dy$$

and finally our answer as an integral over *H(x)*

$$I = \int_{x_1}^{x_2} H(x)dx$$

Example
Integrate r^2 over a circular area with a radius of $r = 2$.

$$I = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} (x^2 + y^2) dx dy = \int_0^r \int_0^\pi \rho^2 d\theta d\rho = \frac{2\pi r^4}{4}$$

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double r = 2;
double func(double x,double y) {
   return x*x+y*y;
}
double y1(double x) {
   return -sqrt(r*r-x*x);
}
double y2(double x) {
   return sqrt(r*r-x*x);
}

int main() {
  double x1=-r, x2=r, s;
  s=integration2(func,x1,x2, y1,y2);
  printf("integration2() = %f\n", s);
  printf("actual integral = %f\n", M_PI*pow(r,4.0)/2.0);
}
```

Output

```
integration2() = 25.156167
actual integral = 25.132744
```

see also
integral1(), **integral2**(), **integral3**(), **integration3**(), **derivative**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

integration3

Synopsis
#include <numeric.h>
double integration3(**double** (**func*)(**double** *x*, **double** *y*, **double** it z), **double** *x1*, **double** *x2*,
double (**y1*)(**double** *x*), **double** (**y2*)(**double** *x*),
double(**z1*)(**double** *x*, **double** *y*), **double**(**z2*)(**double** *x*, **double** *y*));

Syntax
integration3(*func*, *x1*, *x2*, *y1*, *y2*, *z1*, *z2*)

Purpose
Numerical integration of a function of three-demensions.

Return Value
This function returns an integral value of a function *func* integrated from *z1* to *z2* in *z*, *y1* to *y2* in *y* and *x1* to *x2* in *x*.

Parameters
func Function to be integrated.

x1 Function denated to the lower limit of integration in *x*.

x2 Function denated to the upper limit of integration in *x*.

y1 Function denated to the lower limit of integration in *y*.

y2 Function denated to the upper limit of integration in *y*.

z1 Function denated to the lower limit of integration in *z*.

z2 Function denated to the upper limit of integration in *z*.

Description
The function to be integrated, *func*, is specified as a pointer to a three-dimemsional function that takes three arguments and returns a functional value of double type. The values $y_1(x)$ and $y_2(x)$ are user-supplied functions for the upper and lower limits in *y*, the values $z_1(x)$ and $z_2(x)$ are user-supplied functions for the upper and lower limits in *z*, and the values *x1* and *x2* are the end-points of the range in *x* to be integrated, respectively. The return value is the integral of double type.

Algorithm
The integration is carried out by (1) its lower and upper limits in *z* at specified *x* and *y*, denoted *z1(x,y)* and *z2(x,y)*, (2)its lower and upper limits in *y*, denoted *y1(x)* and *y2(x)*; (3)its lower and upper limits in *x*, which we will denote *x1* and *x2*, so that

$$I = \int\int\int dxdydzf(x,y,z) = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dzf(x,y,z)$$

Now we can define a function *G(x,y)* that does the innermost integral,

$$G(x,y) = \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dz$$

and a function *H(x)* that does the integral of *G(x,y)*,

$$H(x,y) = \int_{y_1(x)}^{y_2(x)} G(x,y)dy$$

and finally our answer as an integral over *H(x)*

$$I = \int_{x_1}^{x_2} H(x)dx$$

Example
Integrate r^2 over a spherical volume with a radius of $r = 2$.

$$I = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-x^2-y^2}}^{\sqrt{r^2-x^2-y^2}} (x^2+y^2+z^2)dxdydz = \int_0^{2\pi} \int_0^{\pi} \int_0^{r} \rho^4 \sin(\theta)d\rho d\theta d\phi = \frac{4\pi r^5}{5}$$

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double r;
double func(double x,double y,double z) {
   return x*x+y*y+z*z;
}
double y1(double x) {
   return -sqrt(r*r-x*x);
}
double y2(double x) {
   return sqrt(r*r-x*x);
}
double z1(double x,double y) {
   return -sqrt(r*r-x*x-y*y);
}
double z2(double x,double y) {
   return sqrt(r*r-x*x-y*y);
}

int main() {
   double x1=-2, x2= 2, s;
   r = 2;
   s=integration3(func,x1,x2,y1,y2,z1,z2);
   printf("integration3() = %f\n", s);
   printf("actual integral = %f\n", 4.0*M_PI*pow(r,5.0)/5.0);
}
```

Output

```
integration3() = 80.486994
actual integral = 80.424781
```

see also
integral1(), **integral2**(), **integral3**(), **integration2**(), **derivative**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

interp1

Synopsis
#include <numeric.h>
int interp1(**double** *y*[&], **double** *x*[&], **double** *xa*[&], **double** *ya*[&], **char** **method*);

Purpose
This function finds the values of the function, which is expressed in terms of two arrays *xa* and *ya*, at points expressed in array *x* by linear or cubic spline interpolation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y An output array of function values.

x An input array of variable values.

ya An array which contains function values corresponding to *xa*.

xa An array which contains tabulated x values of the function.

method A string which specifies the method of interpolation. The string `"linear"` or `"spline"` is for linear or spline interpolation, respectively.

Description
This function takes two arrays *xa* and *ya* of the same length that express a known function, and calculates the values, at some other points *x* by linear or cubic spline interpolation. The interpolation method can be chosen by input argument *method* of string with `"linear"` or `"spline"`.

Example
Calculate function values of $\sin(x)$ by linear and cubic spline interpolation.

```
#include <stdio.h>
#include <numeric.h>
#define N 10
#define NUM_X 5

int main() {
    int i,nfunc;
    array double f[NUM_X],x[NUM_X],y[NUM_X];
    array float xa[N],ya[N];

    /* Generation of interpolation tables of sin(x) function */
    linspace(xa, 0, M_PI);
    ya = sin(xa);
    linspace(x, 0.15, M_PI-0.15);
    f=sin(x);

    /*Spline interpolation */
    printf("\nCubic spline interpolation of function sin(x)\n");
    printf("\n%9s %13s %17s\n","x","f(x)","interpolation");
    interp1(y, x, xa, ya, "spline");
```

```
    for(i=0;i<NUM_X;i++)
        printf("%12.6f %12.6f %12.6f\n",x[i],f[i],y[i]);

    /*linear interpolation */
    printf("\nlinear interpolation of function sin(x)\n");
    printf("\n%9s %13s %17s\n","x","f(x)","interpolation");
    interp1(y, x, xa, ya, "linear");
    for(i=0;i<NUM_X;i++)
        printf("%12.6f %12.6f %12.6f\n",x[i],f[i],y[i]);
}
```

Output

```
Cubic spline interpolation of function sin(x)

        x          f(x)     interpolation
    0.150000     0.149438     0.149431
    0.860398     0.758102     0.758072
    1.570797     1.000000     0.999960
    2.281195     0.758102     0.758072
    2.991593     0.149438     0.149430

linear interpolation of function sin(x)

        x          f(x)     interpolation
    0.150000     0.149438     0.146972
    0.860398     0.758102     0.746562
    1.570797     1.000000     0.984808
    2.281195     0.758102     0.746562
    2.991593     0.149438     0.146972
```

See Also
interp2(), **CSpline**::**Interp**(), **CSpline**::**Interpm**().

References
William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

interp2

Synopsis
#include <numeric.h>
int interp2(**double** *z*[&][&], **double** *x*[&], **double** *y*[&], **double** *xa*[&], **double** *ya*[&],
double *za*[&][&], **char** **method*);

Purpose
This function finds the values of a two-dimensional function, at points indicated by two-one-dimensional arrays *x* and *y*, by two dimensional linear or cubic spline interpolation. The function is expressed in terms of tabulated values.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
z An output two-dimensional matrix which contains calculated function values

x An array of *x* values at which the function values will be calculated by interpolation.

y An array of *y* values at which the function values will be calculated by interpolation.

xa An input one-dimensional array which contains tabulated values of the first variable.

ya An input one-dimensional array which contains tabulated values of the second variable.

za An input two-dimensional matrix which contains function values corresponding to *xa* and *ya*.

method A string which specifies the method of interpolation. The string `"linear"` and `"spline"` is for linear or spline interpolation, respectively.

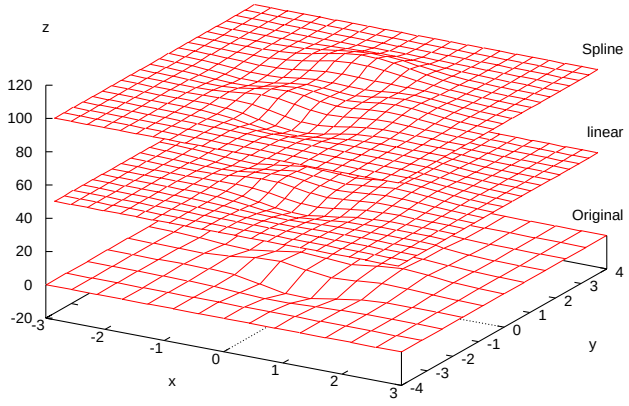
Description
This function takes two arrays, *xa* of dimension *m*, and *ya* of dimension *n*, and a matrix of function value *za* of dimension $m \times n$ tabulated at the grid points defined by *xa* and *ya*, and calculates the values of the function, at points defined by arrays of *x* and *y* by linear or cubic spline interpolation. The dimensions for arrays *xa*, *ya*, *x* and *y* can be different.

Example
Interpolate a two- dimensional function
$f(x,y) = 3(1-x)^2 e^{-x^2-y^2+1} - 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$
by linear and cubic spline interpolations.

```
#include <stdio.h>
#include <chplot.h>
#include <numeric.h>
#define M 15
#define N 11
#define NUM_X 22
#define NUM_Y 22

int main() {
```

```
    int i,j;
    array double z_s1[NUM_X*NUM_Y], z_l1[NUM_X*NUM_Y];
    array double x[NUM_X],y[NUM_Y],xa[M],ya[N];
    array double za[M][N],za1d[M*N], z_s[NUM_X][NUM_Y], z_l[NUM_X][NUM_Y];
    class CPlot plot;

    /* Construct data set of the peaks function */
    linspace(xa, -3, 3);
    linspace(ya, -4, 4);
    for(i=0; i<M; i++) {
        for(j=0; j<N; j++) {
            za[i][j] = 3*(1-xa[i])*(1-xa[i])*
                       exp(-(xa[i]*xa[i])-(ya[j]+1)*(ya[j]+1))
                       - 10*(xa[i]/5 - xa[i]*xa[i]*xa[i]-
                       pow(ya[j],5))*exp(-xa[i]*xa[i]-ya[j]*ya[j])
                       - 1/3*exp(-(xa[i]+1)*(xa[i]+1)-ya[j]*ya[j]);
        }
    }
    za1d = (array double[M*N])za;

    linspace(x, -2.9, 2.9);
    linspace(y, -3.9, 3.9);

    /* test 2-dimensional cubic spline interpolation*/
    interp2(z_s,x,y,xa,ya,za,"spline");

    /* test 2-dimensional linear interpolation */
    interp2(z_l,x,y,xa,ya,za,"linear");

    /* add offset for display */
    z_l1 = (array double[NUM_X*NUM_Y])z_l + (array double[NUM_X*NUM_Y])50;
    z_s1 = (array double[NUM_X*NUM_Y])z_s + (array double[NUM_X*NUM_Y])100;

    plot.data3D(xa, ya, za1d);
    plot.data3D(x, y, z_s1);
    plot.data3D(x, y, z_l1);
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "y");
    plot.label(PLOT_AXIS_Z, "z");
    plot.ticsLevel(0);
    plot.text("Spline", PLOT_TEXT_RIGHT,3.5,3.5,120);
    plot.text("linear", PLOT_TEXT_RIGHT,3.5,3.5,70);
    plot.text("Original", PLOT_TEXT_RIGHT,3.5,3.5,20);
    plot.plotType(PLOT_PLOTTYPE_LINES,0,1,1);
    plot.plotType(PLOT_PLOTTYPE_LINES,1,1,1);
    plot.plotType(PLOT_PLOTTYPE_LINES,2,1,1);
    plot.plotting();
}
```

Output



See Also
interp1(), **CSpline**::Interp(), **CSpline**::Interpm().

References
William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

inverse

Synopsis
#include <**numeric.h**>
array double inverse(**array double** *a*[&][&], ... /* [**int** **status*] */)[:][:];

Syntax
inverse(*a*)
inverse(*a*, *status*)

Purpose
Calculate the inverse of a square matrix of real numbers.

Return Value
This function returns the inverse matrix.

Parameters
a Input square matrix.

status Output integer indicating the status of calculation.

Description
This function calculates the inverse matrix of a square matrix. If calculation is successful,$status = 0$, otherwise $status \neq 0$. To calculate the inverse of a matrix of complex numbers, use function **cinverse**().

Example1

```
#include <numeric.h>
int main() {
    array double a[2][2]  = {2, 4,
                             3, 7};
    array double inv[2][2];
    int status;

    inv = inverse(a);
    printf("inverse(a) =\n%f\n", inv);
    printf("inverse(a)*a =\n%f\n", inv*a);
    printf("a*inverse(a) =\n%f\n", a*inv);

    inv = inverse(a, &status);
    if(status == 0)
        printf("inverse(a, &status) = %f\n", inv);
    else
        printf("error: numerical error in inverse()\n");
}
```

Output1

```
inverse(a) =
3.500000 -2.000000
-1.500000 1.000000
```

```
inverse(a)*a =
1.000000 0.000000
0.000000 1.000000

a*inverse(a) =
1.000000 0.000000
-0.000000 1.000000

inverse(a, &status) = 3.500000 -2.000000
-1.500000 1.000000
```

Example2

```
#include <numeric.h>
array double func(array double B[2][2])[2][2] {
  return inverse(B);
}

int main() {
    array double a[2][2]  = {2, 4,
                             3, 7};
    array double inv[2][2];
    int status;

    inv = func(a);
    printf("inverse(a) =\n%f\n", inv);
    printf("inverse(a)*a =\n%f\n", inv*a);
    printf("a*inverse(a) =\n%f\n", a*inv);
}
```

Output2

```
inverse(a) =
3.500000 -2.000000
-1.500000 1.000000

inverse(a)*a =
1.000000 0.000000
0.000000 1.000000

a*inverse(a) =
1.000000 0.000000
-0.000000 1.000000
```

See Also

cinverse(), **ludecomp**(), **linsolve**().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadephia, 1992.

lcm()

Synopsis
#include <**numeric.h**>
int lcm(**array int** &*g*, **array int** &*u*, **array int** &*u*);

Syntax
lcm(*g*, *u*, *v*)

Purpose
Obtain the least common multiple of the corresponding elements of two arrays of integer type.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
g An output array which is the same dimension and size as *u*. It contains the result of the least common multiple calculation of *u* and *v*.

u Any-dimensional array which contains positive integer elements for the least common multiple calculation.

v An array which is the same dimension and size as *u*. It contains another positive integer elements for the lease common multiple calculation.

Description
This function calculates the least common multiple of corresponding elements *u* and *v*. The arrays *u* and *v* shall be the same size and contains positive integer values. The output array *g* is a positive integer array of same size as *u*.

Example

```
#include <numeric.h>
int main() {
   array int u[2][3] = {1,2,7,15,3,4};
   array int v[2][3] = {2,4,8,3,8,3};
   array int g[2][3];
   lcm(g,u,v);
   printf("g=%d",g);
}
```

Output

```
g=2 4 56
15 24 12
```

See Also
gcd().

References
Knuth, Donald Ervin, *The art of computer programming*, Vol. 2, Addison-Wesley Pub. Co., 1973.

linsolve

Synopsis
#include <numeric.h>
int linsolve(**array double complex** *x*[&], **array double complex** *a*[&][&], **array double complex** *b*[&]);

Purpose
Solve the linear system of equations by LU decomposition.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Output array which contains the solution of equations.

a Input two-dimensional array contains the coefficients of a linear system of equations.

b Input one-dimensional array for linear equations.

Description
This function solves the system of linear equations by LU decomposition. The input matrix *a* should be n-by-n. The function can handle the equation set with complex numbers. The function returns 0 if there is a solution for the equation set, otherwise it returns -1.

It calculates the solution to a real/complex system of linear equations $a*X = b$, where a is an n-by-n matrix. The LU decomposition with partial pivoting and row interchanges is used to factor *a* as $a = P * L * U$, where P is a permulation matrix, L is unit lower triangular, and U is upper triangular. The factored form of *a* is then used to solve the system of equation $a * X = b$.

Example1
Solution of a linear system of equations with real coefficients.

```
#include <numeric.h>
int main() {
     array double a[3][3] = {3, 0, 6,
                             0, 2, 1,
                             1, 0, 1};
     array double b[3] = {2,
                          13,
                          25};
     array float a1[3][3] = {3, 0, 6,
                             0, 2, 1,
                             1, 0, 1};
     array float b1[3] = {2,
                          13,
                          25};
     array double x[3];
     int status;

     linsolve(x, a, b);
     printf("linsolve(a,b) =\n%f\n", x);
     linsolve(x, a1, b1);
     printf("linsolve(a1,b1) =\n%f\n", x);
```

```
     status = linsolve(x, a, b);
     if(status == 0)
         printf("linsolve(x,a,b) =\n%f\n", x);
     else
         printf("error: numerical error in linsolve()\n");
}
```

Output1

```
linsolve(a,b) =
49.333333 18.666667 -24.333333

linsolve(a1,b1) =
49.333333 18.666667 -24.333333

linsolve(x,a,b) =
49.333333 18.666667 -24.333333
```

Example2
Solution of a linear system of equations with complex coefficients.

```
#include <numeric.h>
int main() {
     array double complex a[3][3] = {complex(3,4), 0, 6,
                              0, 2, 1,
                              1, 0, 1};
     array double complex b[3] = {2,
                                  13,
                                  25};
     array double complex x[3];
     int status;

     linsolve(x, a, b);
     printf("linsolve(x, a,b) =\n%f\n", x);

     status = linsolve(x, a, b);
     if(status == 0)
         printf("linsolve(x,a,b) =\n%f\n", x);
     else
         printf("error: numerical error in linsolve()\n");
}
```

Output2

```
linsolve(x, a,b) =
complex(17.760000,23.680000) complex(2.880000,11.840000) complex(7.240000,-23.680000)


linsolve(x,a,b) =
complex(17.760000,23.680000) complex(2.880000,11.840000) complex(7.240000,-23.680000)
```

See Also
llsqsolve(), **linsolve**(), **inverse**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

linspace

Synopsis
#include <**numeric.h**>
int linspace(**array double** &*x*, **double** *first*, **double** *last*);

Purpose
Get linearly spaced values for an array.

Return Value
This function returns the number of elements in array *x*.

Parameters
x An output with linearly spaced values.

first Starting point of the linearly spaced array.

last Ending point of the linearly spaced array.

Description
This function obtains a linearly spaced values starting with *first* and ending with *last* for the array *x*. The number of points is taken internally from the array *x*.

Example

```
#include <numeric.h>
int main() {
  array double x[11], x2[2][3], x3[2][3][2];
  array float y[11];
  array int z[11];
  array unsigned int ui[11];
  array unsigned short uh[11];
  array short h[11];
  array unsigned char uc[11];
  array char c[11];
  int n;

  linspace(x, 1, sizeof(x)/sizeofelement(elementtype(x)));
  printf("x = %f\n", x);

  linspace(x2, 1, sizeof(x2)/sizeofelement(elementtype(x2)));
  printf("x2 = %f\n", x2);

  linspace(x3, 1, sizeof(x3)/sizeofelement(elementtype(x3)));
  printf("x3 = %f\n", x3);

  linspace(x, 0, M_PI);
  printf("x = %f\n", x);
  n = linspace(x, 0, M_PI);
  printf("n = %d\n", n);

  linspace(y, 0, M_PI); /* y is float */
  printf("y = %f\n", y);
```

```
  linspace(z, 0, 90);
  printf("z = %d\n", z);

  linspace(ui, 0, 90);
  printf("ui = %d\n", ui);

  linspace(uh, 0, 90);
  printf("uh = %d\n", uh);

  linspace(h, 0, 90);
  printf("h = %d\n", h);

  linspace(c, 0, 90);
  printf("c = %d\n", c);

  linspace(uc, 0, 90);
  printf("uc = %d\n", uc);
}
```

Output

```
x = 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000 11.000000

x2 = 1.000000 2.000000 3.000000
4.000000 5.000000 6.000000

x3 = 1.000000 2.000000
3.000000 4.000000
5.000000 6.000000

7.000000 8.000000
9.000000 10.000000
11.000000 12.000000

x = 0.000000 0.314159 0.628319 0.942478 1.256637 1.570797 1.884956 2.199115 2.513274
2.827434 3.141593

n = 11
y = 0.000000 0.314159 0.628319 0.942478 1.256637 1.570796 1.884956 2.199115 2.513274
2.827434 3.141593

z = 0 9 18 27 36 45 54 63 72 81 90

ui = 0 9 18 27 36 45 54 63 72 81 90

uh = 0 9 18 27 36 45 54 63 72 81 90

h = 0 9 18 27 36 45 54 63 72 81 90

c = 0 9 18 27 36 45 54 63 72 81 90

uc = 0 9 18 27 36 45 54 63 72 81 90
```

See Also
logspace().

References

llsqcovsolve()

Synopsis
#include <numeric.h>
int llsqcovsolve(array double *x[&]*, **array double** *a[&][&]*, **array double** *b[&]*,
array double *v[&][&]*, ... /* [**array double** *p[&]*] */);

Syntax
llsqcovsolve(*a*, *b*, *v*, *x*)
llsqcovsolve(*a*, *b*, *v*, *x*, *p*)

Purpose
Solve linear system of equations based on linear least-squares with known covariance.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a A two-dimensional array size of $m \times n$ which contains input data. It is an overdetermined least-squares problem. So m must be larger than n.

b A vector with m elements. It contains input data.

v An input array of size $m \times m$. It contains covariance value.

x An output vector with n elements. It contains the linear least-squares results.

p An optional output vector with n elements. It contains the standard errors of *x*.

Description
This function calculates the vector *x* that satisfies $a * x = b + e$ where e is a normally distributed error with zero mean and covariance *v* using the linear least-squares method. This is an overdetermined linear least-squares problem. So the number of rows m must be larger than the number of column n. If optional output vector *p* is specified, it will pass out the standard errors of *x*.

Algorithm
The vector *x* can be calculated through the minimizes the quantity $(ax - b)^T v^{-1}(ax - b)$. The classical linear algebra solution to this problem is

$$x = (a^T v^{-1} a)^{-1} a^T v^{-1} b$$

and the standard errors of *x* is

$$\begin{aligned} \text{mse} &= b^T(v^{-1} - v^{-1}a(a^T v^{-a}a)^{-1}a^T v^{-1})b/(m-n) \\ p &= \sqrt{\textbf{diagonal}(a^T v^{-1} a)^{-1} * \text{mse}} \end{aligned}$$

Example

```
#include <numeric.h>

int main() {
     array double a[4][3] = {1, 2, 3,
                             4, 5, 6,
                             7, 8, 9,
                             11,12,3}; /* a[m][n] */
     array double b[4] = {4,4,54,6};
     array double v[4][4] ={ 1, 1, 1, 1,
                             1, 2, 3, 4,
                             1, 3, 6,10,
                             1, 4,10,20};
     array double x[3], dx[3];

     llsqcovsolve(x, a, b, v, dx);
     printf("a = \n%f", a);
     printf("b = \n%f", b);
     printf("v = \n%f", v);
     printf("x = \n%f", x);
     printf("a*x = \n%f", a*x);
     printf("dx = \n%f", dx);
}
```

Output

```
a =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
11.000000 12.000000 3.000000
b =
4.000000 4.000000 54.000000 6.000000
v =
1.000000 1.000000 1.000000 1.000000
1.000000 2.000000 3.000000 4.000000
1.000000 3.000000 6.000000 10.000000
1.000000 4.000000 10.000000 20.000000
x =
10.800000 -25.600000 14.800000
a*x =
4.000000 4.000000 4.000000 -144.000000
dx =
61.237244 54.772256 5.270463
```

See Also

qrdecomp(), **llsqnonnegsolve**(), **llsqsolve**().

References

Franklin A. Graybill, *Theory and Application of the Linear Model*, Duxbury Press, 1976.
Gilbert Strang, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, 1986.

llsqnonnegsolve()

Synopsis
#include <numeric.h>
int llsqnonnegsolve(**array double** *x[&]*, **array double** *a[&][&]*, **array double** *b[&]*, ...
/* [**double** tol, **array double** *w[&]*] */);

Syntax
llsqnonnegsolve(*a*, *b*, *x*)
llsqnonnegsolve(*a*, *b*, *x*, *tol*)
llsqnonnegsolve(*a*, *b*, *x*, *tol*, *p*)

Purpose
Solve linear system of equation with non-negative values based on linear least-squares method.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a A two-dimensional array size of $m \times n$ which contains input data.

b A vector with m elements. It contains input data.

x An output vector with n elements. It contains the non-negative least squares results.

tol An optional input which specifies the tolerance of the solution. If the user does not specify this argument or specify zero, $tol = 10 * max(m, n) * norm(u, "1") *$ FLT_EPSILON is used by default. FLT_EPSILON is defined in header file **float.h**.

w An optional output vector with n elements. It contains a vector where $w[i] < 0$ when $x[i] = 0$ and $w[i] \cong 0$ when $x[i] > 0$.

Description
This function solves the equations $ax = b$ using the linear least-squares method, while it is subject to the constraint that the solution vector *x* has non-negative elements. That is, $x[i] \geq 0$ for $i = 0, 1, \cdots, n - 1$.

Algorithm
This function uses the algorithm described in the reference book, Chapter 23.

Example

```
#include <numeric.h>
int main () {

array double a[15][5]={-0.1340555,-0.2016283,-0.1693078,-0.1897199,-0.1738723,
                       -0.1037948,-0.1576634,-0.1334626,-0.1484855,-0.1359769,
                       -0.0877960,-0.1288387,-0.1068301,-0.1201180,-0.1093297,
                       0.02058554,0.00335331,-0.0164127,0.00078606,0.00271659,
                       -0.0324809,-0.0187680,0.00410639,-0.0140589,-0.0138439,
                       0.05967662,0.06667714,0.04352153,0.05740438,0.05024962,
```

```
                        0.06712457,0.07352437,0.04489770,0.06471862,0.05876445,
                        0.08687186,0.09368296,0.05672327,0.08141043,0.07302320,
                        0.02149662,0.06222662,0.07213486,0.06200069,0.05570931,
                        0.06687407,0.10344506,0.09153849,0.09508223,0.08393667,
                        0.15879069,0.18088339,0.11540692,0.16160727,0.14796479,
                        0.17842887,0.20361830,0.13057860,0.18385729,0.17005549,
                        0.11414080,0.17259611,0.14816471,0.16007466,0.17005549,
                        0.07846038,0.14669563,0.14365800,0.14003842,0.12571277,
                        0.10803175,0.16994623,0.14971519,0.15885312,0.14301547},
    b[15]={-0.4361,-0.3437,-0.2657,-0.0392,0.0193,0.0747,0.0935,0.1079,0.1930,
           0.2058,0.2606,0.3142,0.3529,0.3615,0.3647};
   array double x[5], w[5];
   llsqnonnegsolve(x,a,b,0.0,w);
   printf("a=%6.4f",a);
   printf("b=%6.4f",b);
   printf("x=%6.4f",x);
   printf("w=%6.4f",w);
}
```

Output

```
a=-0.1341 -0.2016 -0.1693 -0.1897 -0.1739
-0.1038 -0.1577 -0.1335 -0.1485 -0.1360
-0.0878 -0.1288 -0.1068 -0.1201 -0.1093
0.0206 0.0034 -0.0164 0.0008 0.0027
-0.0325 -0.0188 0.0041 -0.0141 -0.0138
0.0597 0.0667 0.0435 0.0574 0.0502
0.0671 0.0735 0.0449 0.0647 0.0588
0.0869 0.0937 0.0567 0.0814 0.0730
0.0215 0.0622 0.0721 0.0620 0.0557
0.0669 0.1034 0.0915 0.0951 0.0839
0.1588 0.1809 0.1154 0.1616 0.1480
0.1784 0.2036 0.1306 0.1839 0.1701
0.1141 0.1726 0.1482 0.1601 0.1701
0.0785 0.1467 0.1437 0.1400 0.1257
0.1080 0.1699 0.1497 0.1589 0.1430
b=-0.4361 -0.3437 -0.2657 -0.0392 0.0193 0.0747 0.0935 0.1079 0.1930 0.2058 0.2606
0.3142 0.3529 0.3615 0.3647
x=0.0000 0.0000 2.4393 0.0000 0.0000
w=-0.0055 -0.0035 0.0000 -0.0024 -0.0023
```

See Also
llsqcovsolve(), **llsqsolve**().

References
Lawson, C. L. and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, 1974.

llsqsolve

Synopsis
#include <numeric.h>
int llsqsolve(**array double complex** *x*[&], **array double complex** *a*[&][&],
array double complex *b*[&]);

Purpose
Linear least-squares solution of a linear system of equations.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Output array which contains the solution of a linear system of equations.

a Input matrix for a linear system of equations.

b Input array for constants of a linear system of equations.

Description
This function solves the linear system of equations by the method called *linear least-squares* method. The input matrix *x* can be of different number of rows and columns. But, the number of elements for *x* shall be the same as the number of columns in matrix *a*. The number of elements for *b* shall be the same as the number of rows in matrix *a*. The function can handle the equation set with complex numbers. The function returns 0 if there is a solution for the system of linear equations, otherwise it returns -1.
Algorithm
The linear system of equations

$$a * X = b$$

is solved by minimizing the squared error in $(a * X - b)$.

Example1
Solutions of linear system of equations with *a*[3][3] and *a*[3][4].

```
#include <numeric.h>
#define N1 3
#define N2 4
int main() {
     array double a[N1][3] = {3, 0, 6,
                              0, 2, 1,
                              1, 0, 1};
     array double a2[N2][3] = {3, 0, 6,
                               0, 2, 1,
                               1, 0, 1,
                               4, 5, 2};
     array double b[N1] = {2,
                           13,
                           25};
     array double b2[N2] = {2,
```

```
                          13,
                          25,
                          1};
     array double x[3], x2[3];
     int status;

     llsqsolve(x, a, b);
     printf("llsqsolve(x,a,b) = \n%f\n", x);
     printf("a*x = \n%f\n", a*x);
     llsqsolve(x2, a2, b2);
     printf("llsqsolve(x2, a2,b2) = \n%f\n", x2);
     printf("a2*x2 = \n%f\n", a2*x2);

     status = llsqsolve(x, a, b);
     if(status == 0)
         printf("llsqsolve(x, a,b) =\n%f\n", x);
     else
         printf("error: numerical error in llsqsolve()\n");
}
```

Output1

```
llsqsolve(x,a,b) =
49.333333 18.666667 -24.333333

a*x =
2.000000 13.000000 25.000000

llsqsolve(x2, a2,b2) =
-2.256881 1.715596 2.198777

a2*x2 =
6.422018 5.629969 -0.058104 3.948012

llsqsolve(x, a,b) =
49.333333 18.666667 -24.333333
```

Example2
Solution of linear system of equations with complex numbers.

```
#include <numeric.h>
int main() {
     array double complex a[3][3] = {complex(3,2), 0, 6,
                          0, 2, 1,
                          1, 0, 1};
     array double complex a2[3][4] = {complex(3,2), 0, 6, 0,
                                  2, 1, 1, 0,
                                  1, 4, 5, 2};
     array double complex b[3] = {2,
                         13,
                         25};
     array double complex b2[3] = {2,
                         13,
                         25};
     array double complex x[3], x2[4];
     int status;

     llsqsolve(x, a, b);
     printf("llsqsolve(x, a,b) = \n%f\n", x);
```

```
    llsqsolve(x2, a2, b2);
    printf("a*x = \n%f\n", a*x);
    printf("llsqsolve(x2, a2,b2) = \n%f\n", x2);
    printf("a2*x2 = \n%f\n", a2*x2);

    status = llsqsolve(x, a, b);
    if(status == 0)
        printf("llsqsolve(x,a,b) is ok\n");
    else
        printf("error: numerical error in llsqsolve()\n");
}
```

Output2

```
llsqsolve(x, a,b) =
complex(34.153846,22.769231) complex(11.076923,11.384615) complex(-9.153846,-22.769231)


a*x =
complex(2.000000,0.000000) complex(13.000000,0.000000) complex(25.000000,0.000000)


llsqsolve(x2, a2,b2) =
complex(3.953216,0.102924) complex(6.702534,1.163353) complex(-1.608967,-1.369201)
complex(1.140741,1.044834)

a2*x2 =
complex(2.000000,0.000000) complex(13.000000,-0.000000) complex(25.000000,0.000000)


llsqsolve(x,a,b) is ok
```

See Also
linsolve().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

logm

Synopsis
#include <**numeric.h**>
int logm(**array double complex** *y[&][&]*, **array double complex** *x[&][&]*);

Syntax
logm(*y*, *x*)

Purpose
Computes the matrix natural logarithm.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input square matrix. It contains data to be calculated.

y Output square matrix which contains data of the result of *x* natural logarithm..

Description
This function computes the matrix natural logarithm of the input matrix *x*. It is the inverse function of **expm**(). The complex results of *y* will be produced if *x* has negative eigenvalues.

Example
This example shows the usage of **expm**() and **logm**(). A real or complex matrix will change back to itself when a matrix is calculated by **expm**() and then followed by **logm**().

```
#include <numeric.h>
int main() {
   array double x[3][3]={1,2,3,
                         3,4,5,
                         6,7,8};
   array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                         3,complex(4,1),complex(2,5),
                         0,0,0};
   array double complex zy[3][3];
   array double complex y[3][3];

   expm(y,x);
   printf("x = \n%f",x);
   printf("y = \n%5.3f",y);
   logm(x,y);
   printf("x = \n%f",x);
   printf("\n");

   expm(zy,zx);
   printf("zx = \n%5.3f",zx);
   printf("zy = \n%5.3f",zy);
   logm(zx,zy);
   printf("zx = \n%5.3f",zx);
}
```

Output

```
x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000
y =
complex(157372.953,0.000) complex(200034.605,0.000) complex(242697.257,0.000)

complex(290995.910,0.000) complex(369883.552,0.000) complex(448769.194,0.000)

complex(491431.846,0.000) complex(624654.473,0.000) complex(757878.099,0.000)

x =
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000

zx =
complex(1.000,1.000) complex(2.000,2.000) complex(0.000,0.000)
complex(3.000,0.000) complex(4.000,1.000) complex(2.000,5.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
zy =
complex(-44.901,56.755) complex(-87.478,70.853) complex(-101.511,-18.910)

complex(-12.469,118.748) complex(-57.370,175.502) complex(-155.470,67.839)

complex(0.000,0.000) complex(0.000,0.000) complex(1.000,0.000)
zx =
complex(1.000,1.000) complex(2.000,2.000) complex(0.000,0.000)
complex(3.000,0.000) complex(4.000,1.000) complex(2.000,5.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
```

See Also

expm(), **funm**(), **cfunm**(), **sqrtm**().

References

G. H. Golub, C. F. Van Loan, Matrix Computations Third edition, The Johns Hopkins University Press, 1996

logspace

Synopsis
#include <**numeric.h**>
int logspace(**array double** &*x*, **double** *first*, **double** *last*);

Purpose
Get logarithmically spaced values for an array.

Return Value
This function returns the number of elements in array *x*.

Parameters
x An output array with logarithmically spaced values.

first Starting point of the logarithmically spaced array.

last Ending point of the logarithmically spaced array.

Description
This function obtains a logarithmically spaced values starting with *first* and ending with *last* for array *x*. The number of points is taken internally from the array *x*.

Example

```
#include <numeric.h>
int main() {
  array double x[11], x2[2][3];
  array float y[11];
  array complex z[11];
  int n;

  logspace(x, 0, 2);
  printf("x = %f\n", x);
  logspace(x2, 0, 2);
  printf("x2 = %f\n", x2);
  n = logspace(x, 0, 2);
  printf("n = %d\n", n);
  n = logspace(y, 0, 2);
  printf("y = %f\n", y);
  logspace(z, 0, 2);
  printf("z = %f\n", z);
}
```

Output

```
x = 1.000000 1.584893 2.511886 3.981072 6.309573 10.000000 15.848932 25.118864
39.810717 63.095734 100.000000

x2 = 1.000000 2.511886 6.309573
15.848932 39.810717 100.000000

n = 11
```

```
y = 1.000000 1.584893 2.511886 3.981072 6.309574 10.000000 15.848932 25.118864
39.810719 63.095734 100.000000

z = complex(1.000000,0.000000) complex(1.584893,0.000000) complex(2.511886,0.000000)
complex(3.981072,0.000000) complex(6.309574,0.000000) complex(10.000000,0.000000)
complex(15.848932,0.000000) complex(25.118864,0.000000) complex(39.810719,0.000000)
complex(63.095734,0.000000) complex(100.000000,0.000000)
```

See Also
linspace().

References

ludecomp

Synopsis
#include <**numeric.h**>
int ludecomp(**array double complex** *a*[&][&], **array double complex** *l*[&][&],
array double complex *u*[&][&],... /***array int** *p*[:][:] */);

Syntax
ludecomp(a, l, u)
ludecomp(a, l, u, p)

Purpose
LU decomposition of a general m-by-n matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a Input matrix.

l Output L matrix.

u Output U matrix.

p Output P matrix.

Description
An LU decomposition of a square matrix *a* is calculated using partial pivoting with row interchanges. The factorization has the form $a = P * L * U$, where P is a permutation matrix, L a lower triangular matrix with unit diagonal elements and U an upper triangular matrix.

Example1
LU decomposition of a real 3×3 matrix.

```
#include <numeric.h>
int main() {
     int n = 3;
     array double a[3][3] = {2,1,-2,
                             4,-1,2,
                             2,-1,1} ; /* n-by-n matrix */
     array double a2[3][3] = {-1,5,6,
                              3,-6,1,
                              6,8, 9} ; /* n-by-n matrix */
     array double l[n][n], u[n][n];
     array int p[n][n];
     int status;

     status = ludecomp(a, l, u);
     printf("l =\n%f\n", l);
     printf("u =\n%f\n", u);
     printf("l*u =\n%f\n", l*u);
```

```
    /* pass the address of p[0][0]
       or use ludecomp(a, l, u, &p);
       or use ludecomp(a, l, u, &p[0][0]); */
    status = ludecomp(a, l, u, p);
    printf("l =\n%f\n", l);
    printf("u =\n%f\n", u);
    printf("p =\n%d\n", p);
    printf("p*l*u =\n%f\n", p*l*u);

    status = ludecomp(a2, l, u);
    if(status == 0) {
        printf("l =\n%f\n", l);
        printf("u =\n%f\n", u);
        printf("l*u =\n%f\n", l*u);
    }
    else
        printf("error: numerical error in ludecomp()\n");

    status = ludecomp(a2, l, u, p);
    printf("l =\n%f\n", l);
    printf("u =\n%f\n", u);
    printf("p =\n%d\n", p);
    printf("p*l*u =\n%f\n", p*l*u);
}
```

Output1

```
l =
0.500000 1.000000 0.000000
1.000000 0.000000 0.000000
0.500000 -0.333333 1.000000

u =
4.000000 -1.000000 2.000000
0.000000 1.500000 -3.000000
0.000000 0.000000 -1.000000

l*u =
2.000000 1.000000 -2.000000
4.000000 -1.000000 2.000000
2.000000 -1.000000 1.000000

l =
1.000000 0.000000 0.000000
0.500000 1.000000 0.000000
0.500000 -0.333333 1.000000

u =
4.000000 -1.000000 2.000000
0.000000 1.500000 -3.000000
0.000000 0.000000 -1.000000

p =
0 1 0
1 0 0
0 0 1

p*l*u =
```

```
2.000000 1.000000 -2.000000
4.000000 -1.000000 2.000000
2.000000 -1.000000 1.000000

l =
-0.166667 -0.633333 1.000000
0.500000 1.000000 0.000000
1.000000 0.000000 0.000000

u =
6.000000 8.000000 9.000000
0.000000 -10.000000 -3.500000
0.000000 0.000000 5.283333

l*u =
-1.000000 5.000000 6.000000
3.000000 -6.000000 1.000000
6.000000 8.000000 9.000000

l =
1.000000 0.000000 0.000000
0.500000 1.000000 0.000000
-0.166667 -0.633333 1.000000

u =
6.000000 8.000000 9.000000
0.000000 -10.000000 -3.500000
0.000000 0.000000 5.283333

p =
0 0 1
0 1 0
1 0 0

p*l*u =
-1.000000 5.000000 6.000000
3.000000 -6.000000 1.000000
6.000000 8.000000 9.000000
```

Example2
LU decomposition of 3×3 matrix with complex number.

```
#include <numeric.h>
int main() {
     int n = 3;
     array double complex a[3][3] = {complex(-1,1),5,6,
                                      3,-6,1,
                                      6,8, 9} ; /* n-by-n matrix */
     array double complex l[n][n], u[n][n];

     ludecomp(a, l, u);
     printf("l =\n%f\n", l);
     printf("u =\n%f\n", u);
     printf("l*u =\n%f\n", l*u);
}
```

Output2

```
l =
complex(-0.166667,0.166667) complex(-0.633333,0.133333) complex(1.000000,0.000000)

complex(0.500000,0.000000) complex(1.000000,0.000000) complex(0.000000,0.000000)

complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,0.000000)


u =
complex(6.000000,0.000000) complex(8.000000,0.000000) complex(9.000000,0.000000)

complex(0.000000,0.000000) complex(-10.000000,0.000000) complex(-3.500000,0.000000)

complex(0.000000,0.000000) complex(0.000000,0.000000) complex(5.283333,-1.033333)


l*u =
complex(-1.000000,1.000000) complex(5.000000,0.000000) complex(6.000000,0.000000)

complex(3.000000,0.000000) complex(-6.000000,0.000000) complex(1.000000,0.000000)

complex(6.000000,0.000000) complex(8.000000,0.000000) complex(9.000000,0.000000)
```

See Also
linsolve, **inverse**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

maxloc

Synopsis
#include <**numeric.h**>
int maxloc(**array double** &*a*);

Purpose
Find the index of element with the maximum value in an array.

Return Value
This function returns the index for the elements with the maximum value in an array.

Parameters
a The input array in which the maximum value shall be found.

Description
The function finds the maximum element in the array *a*.

Example

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,10,5,6};
    float  b1[2][3] = {1,2,3,10,5,6};
    int    b2[2][3] = {1,2,3,10,5,6};
    double c[2][3][5];
    c[0][0][3] = 10;
    c[1][0][3] = -10;
    int loc;

    loc = maxloc(a);
    printf("maxloc(a) = %d\n", loc);
    loc = maxloc(b);
    printf("maxloc(b) = %d\n", loc);
    loc = maxloc(b1);
    printf("maxloc(b1) = %d\n", loc);
    loc = maxloc(b2);
    printf("maxloc(b2) = %d\n", loc);
    loc = maxloc(c);
    printf("maxloc(c) = %d\n", loc);
}
```

Output

```
maxloc(a) = 2
maxloc(b) = 3
maxloc(b1) = 3
maxloc(b2) = 3
maxloc(c) = 3
```

See Also
minloc(), **maxv**().

References

maxv

Synopsis
#include <**numeric.h**>
array double maxv(**array double** *a*[&][&])[:];

Purpose
Find the maximum values of the elements of each row of a matrix.

Return Value
This function returns an array which contains the maximum values of each row of the matrix.

Parameters
a Input matrix.

Description
This function finds the maximum values of each row of a matrix. The matrix can be of any arithmetic data type.

Example

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double maxv1[2], maxv2[3];

    maxv1 = maxv(a);
    printf("maxv(a) = %f\n", maxv1);
    maxv2 = maxv(b);
    printf("maxv(b) = %f\n", maxv2);
}
```

Output

```
maxv(a) = 3.000000 6.000000

maxv(b) = 4.000000 8.000000 4.000000
```

See Also
maxloc(), **minloc**().

References

mean

Synopsis
#include <**numeric.h**>
double mean(**array double** &*a*, ... /*[**array double** *v*[:]] */);

Syntax
mean(*a*)
mean(*a*, *v*)

Purpose
Calculate the mean value of all elements of an array and mean values of the elements of each row of a two-dimensional array.

Return Value
This function returns the mean value of all the elements.

Parameters
a An input array of any dimension.

v An output array which contains the mean values of each row of a two-dimensional array.

Description
The function calculates the mean value of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate mean values of each row. The mean values of each row are passed out by argument *v*.

Example1
Calculate the mean values of all elements of arrays with different dimensions.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    int b[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double meanval;

    meanval = mean(a);
    printf("mean(a) = %f\n", meanval);
    meanval = mean(b);
    printf("mean(b) = %f\n", meanval);
    meanval = mean(c);
    printf("mean(c) = %f\n", meanval);
}
```

Output1

```
mean(a) = 2.000000
mean(b) = 3.500000
mean(c) = 1.000000
```

Example2

Calculate the mean values of all elements of arrays with different dimensions. The mean values of each row are passed by argument *v*.

```
#include <numeric.h>
#define N 2
#define M 3
int main() {
    double a[N][M] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double meana1[N], meanb1[3];
    double meanval;

    /* Note: second argument of mean() must be double data type */

    meanval = mean(a, meana1);
    printf("meanval = mean(a, meana1) = %f\n", meanval);
    printf("mean(a, meana1) = %f\n", meana1);


    meanval = mean(b, meanb1);
    printf("meanval = mean(b, meanb1) = %f\n", meanval);
    printf("mean(b, meanb1) = %f\n", meanb1);
}
```

Output2

```
meanval = mean(a, meana1) = 3.500000
mean(a, meana1) = 2.000000 5.000000

meanval = mean(b, meanb1) = 3.833333
mean(b, meanb1) = 2.500000 6.500000 2.500000
```

Example3

Calculate the mean values of each column of an array. The calculation is carried out by transposing the array.

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double meana1[3], meanb1[4];
    double meanval;


    mean(transpose(a), meana1);
    printf("mean(traspose(a), meana1) = %f\n", meana1);

    meanval = mean(transpose(b), meanb1);
    printf("mean(transpose(b), meanb1) = %f\n", meanb1);
}
```

Output3

```
mean(traspose(a), meana1) = 2.500000 3.500000 4.500000

mean(transpose(b), meanb1) = 2.333333 3.333333 4.333333 5.333333
```

See Also
median(), **minloc**(), **maxloc**().

References

median

Synopsis
#include <**numeric.h**>
double median(**array double** &*a*, ... /*[**array double** *v*[:]] */);

Syntax
median(a)
median(a, v)

Purpose
Find the median value of all elements of an array and median values of the elements in each row of a two-dimensional array.

Return Value
This function returns the median value of all the elements.

Parameters
a An input array of any dimension.

v An output array which contains the median values of each row.

Description
The function calculates the median value of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate the median values of each row. The median values of each row are passed out by argument *v*.

Example1
Find the median value of all the elements of arrays.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double medianval;

    medianval = median(a);
    printf("median(a) = %f\n", medianval);
    medianval = median(b);
    printf("median(b) = %f\n", medianval);
    medianval = median(c);
    printf("median(c) = %f\n", medianval);
}
```

Output

```
median(a) = 2.000000
median(b) = 3.500000
median(c) = 0.000000
```

Example2
Find the median values of each row of arrays.

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double medianv1[2], medianv2[3];

    median(a, medianv1);
    printf("median(a, medianv1) = %f\n", medianv1);
    median(b, medianv2);
    printf("median(b, medianv2) = %f\n", medianv2);
}
```

Output

```
median(a, medianv1) = 2.000000 5.000000

median(b, medianv2) = 2.500000 6.500000 2.500000
```

See Also
mean(), **minv**().

References

minloc

Synopsis
#include <**numeric.h**>
int minloc(**array double** &*a*);

Purpose
Find the index of the element with the minimum value in an array.

Return Value
This function returns the index for the element with the minimum value in an array.

Parameters
a The input array in which the index of the element with the minimum value shall be found.

Description
The function finds the index of the element with the minimum value in the array *a*. The array can be of any dimension.

Example

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,-10,5,6};
    double c[2][3][5];
    c[0][0][3] = 10;
    c[1][0][3] = -10;
    int loc;

    loc = minloc(a);
    printf("minloc(a) = %d\n", loc);
    loc = minloc(b);
    printf("minloc(b) = %d\n", loc);
    loc = minloc(c);
    printf("minloc(c) = %d\n", loc);
}
```

Output

```
minloc(a) = 0
minloc(b) = 3
minloc(c) = 18
```

See Also
maxloc().

References

minv

Synopsis
#include <**numeric.h**>
array double minv(**array double** *a*[&][&])[:];

Purpose
Find the minimum values of each row in a two-dimensional array.

Return Value
This function returns the array of minimum values.

Parameters
a Input two-dimensional array in which the minimum values of each row will be found.

Description
The function finds the minimum values of each row of a two-dimensional array. The input array can be of any arithmetic data type.

Example

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double minv1[2], minv2[3];

    minv1 = minv(a);
    printf("minv(a) = %f\n", minv1);
    minv2 = minv(b);
    printf("minv(b) = %f\n", minv2);
}
```

Output

```
minv(a) = 1.000000 4.000000

minv(b) = 1.000000 5.000000 1.000000
```

See Also
maxloc(), **minloc**(), **median**().

References

norm

Synopsis
#include <**numeric.h**>
double norm(**array double complex** &*a*, **char** * mode);

Purpose
Calculate norm of a vector or matrix.

Return Value
This function returns a norm.

Parameters
a Input, one or two-dimensional array for which a norm is calculated.

mode A character array indicating the type of norm to be calculated.

Description
The norm of a vector or matrix is a scalar that gives some measure of the magnitude of the elements of the vector or matrix. The **norm** function calculates norms of different types for a vector or matrix according to the argument *mode*.

Algorithm
The mode of various different norms for both vectors and matrices are defined below.

Mode	Norm Type	Algorithm
	Norm for Vectors	
`"1"`	1-norm	$\|\|a\|\|_1 = \|a_1\| + \|a_1\| + ... + \|a_n\|$
`"2"`	2-norm	$\|\|a\|\|_2 = (\|a_1\|^2 + \|a_2\|^2 + ... + \|a_n\|^2)^{1/2}$
`"p"`	p-norm	$\|\|a\|\|_p = (\|a_1\|^p + \|a_2\|^p + ... + \|a_n\|^p)^{1/p}$ `"p"` is a floating-point number.
`"i"`	infinity norm	$\|\|a\|\|_\infty = \max_i \|a_i\|$
`"-i"`	negative infinity norm	$\|\|a\|\|_{-\infty} = \min_i \|a_i\|$
	Norm for m-by-n Matrices	
`"1"`	1-norm	$\|\|a\|\|_1 = \max_j \sum_{i=1}^{m} \|a_{ij}\|$
`"2"`	2-norm	$\|\|a\|\|_2 =$ maximum singular value of a
`"i"`	infinity norm	$\|\|a\|\|_\infty = \max_i \sum_{j=1}^{m} \|a_{ij}\|$
`"f"`	Frobenius norm	$\|\|a\|\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} \|a_{ij}\|^2$
`"m"`	norm	$\|\|a\|\| = \max(\mathbf{abs}(A[i][j]))$

Example 1
Calculate the norms of real vectors and matrices.

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, -4,
                            3, -7};
    array double b[3][2] = {2, -4,
                            3, -7,
                            1, 5};
```

```
    array double c[2][3] = {2, -4, 3,
                            -7, 1, 5};
    array double complex d[3] = {2, -4, 3};
    double anorm;

/* a is nxn */
    anorm = norm(a, "1");
    printf("1-norm of a = %f\n", anorm);
    anorm = norm(a, "2");
    printf("2-norm of a = %f\n", anorm);
    anorm = norm(a, "i");
    printf("infinity-norm of a = %f\n", anorm);
    anorm = norm(a, "f");
    printf("Frobenius-norm of a = %f\n", anorm);
    anorm = norm(a, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

/* a is nxm */
    anorm = norm(b, "1");
    printf("1-norm of b = %f\n", anorm);
    anorm = norm(b, "2");
    printf("2-norm of b = %f\n", anorm);
    anorm = norm(b, "i");
    printf("infinity-norm of b = %f\n", anorm);
    anorm = norm(b, "f");
    printf("Frobenius-norm of b = %f\n", anorm);
    anorm = norm(b, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

/* a is mxn */
    anorm = norm(c, "1");
    printf("1-norm of c = %f\n", anorm);
    anorm = norm(c, "2");
    printf("2-norm of c = %f\n", anorm);
    anorm = norm(c, "i");
    printf("infinity-norm of c = %f\n", anorm);
    anorm = norm(c, "f");
    printf("Frobenius-norm of c = %f\n", anorm);
    anorm = norm(c, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

/* a is 1-dim */
    anorm = norm(d, "1");
    printf("1-norm of d = %f\n", anorm);
    anorm = norm(d, "2");
    printf("2-norm of d = %f\n", anorm);
    anorm = norm(d, "2.1");
    printf("2.1-norm of d = %f\n", anorm);
    anorm = norm(d, "i");
    printf("infinity-norm of d = %f\n", anorm);
    anorm = norm(d, "-i");
    printf("negative infinity-norm of d = %f\n", anorm);
}
```

Output1

```
1-norm of a = 11.000000
2-norm of a = 8.828855
infinity-norm of a = 10.000000
```

```
Frobenius-norm of a = 8.831761
max(abs(a)) = 7.000000

1-norm of b = 16.000000
2-norm of b = 8.671495
infinity-norm of b = 10.000000
Frobenius-norm of b = 10.198039
max(abs(a)) = 7.000000

1-norm of c = 9.000000
2-norm of c = 9.846035
infinity-norm of c = 13.000000
Frobenius-norm of c = 10.198039
max(abs(a)) = 7.000000

1-norm of d = 9.000000
2-norm of d = 5.385165
2.1-norm of d = 5.263628
infinity-norm of d = 4.000000
negative infinity-norm of d = 2.000000
```

Example 2
Calculate the norms of complex vectors and matrices.

```
#include <numeric.h>
int main() {
    array double complex z[2][2] = {2, -4,
                                    3, -7};
    array complex z1[2][2]       = {2, -4,
                                    3, -7};
    array double complex z2[2][2] = {2, -4,
                                    3, complex(1,-7)};
    array double complex z3[4] = {2, complex(-4,1),3};
    double anorm;

    anorm = norm(z, "1");
    printf("1-norm of a = %f\n", anorm);
    anorm = norm(z, "i");
    printf("infinity-norm of a = %f\n", anorm);
    anorm = norm(z, "f");
    printf("Frobenius-norm of a = %f\n", anorm);
    anorm = norm(z, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

    anorm = norm(z1, "1");
    printf("1-norm of a = %f\n", anorm);
    anorm = norm(z1, "i");
    printf("infinity-norm of a = %f\n", anorm);
    anorm = norm(z1, "f");
    printf("Frobenius-norm of a = %f\n", anorm);
    anorm = norm(z1, "m");
    printf("max(abs(a)) = %f\n\n", anorm);

    anorm = norm(z2, "1");
    printf("1-norm of z2 = %f\n", anorm);
    anorm = norm(z2, "i");
    printf("infinity-norm of z2 = %f\n", anorm);
    anorm = norm(z2, "f");
    printf("Frobenius-norm of z2 = %f\n", anorm);
```

```
    anorm = norm(z2, "m");
    printf("max(abs(z2)) = %f\n\n", anorm);

    anorm = norm(z3, "1");
    printf("1-norm of d = %f\n", anorm);
    anorm = norm(z3, "2");
    printf("2-norm of d = %f\n", anorm);
    anorm = norm(z3, "2.1");
    printf("2.1-norm of d = %f\n", anorm);
    anorm = norm(z3, "i");
    printf("infinity-norm of d = %f\n", anorm);
    anorm = norm(z3, "-i");
    printf("negative infinity-norm of d = %f\n", anorm);

}
```

Output2

```
1-norm of a = 11.000000
infinity-norm of a = 10.000000
Frobenius-norm of a = 8.831761
max(abs(a)) = 7.000000

1-norm of a = 11.000000
infinity-norm of a = 10.000000
Frobenius-norm of a = 8.831761
max(abs(a)) = 7.000000

1-norm of z2 = 11.071068
infinity-norm of z2 = 10.071068
Frobenius-norm of z2 = 8.888194
max(abs(z2)) = 7.071068

1-norm of d = 9.123106
2-norm of d = 5.477226
2.1-norm of d = 5.355310
infinity-norm of d = 4.123106
negative infinity-norm of d = 0.000000
```

See Also

condnum(), **rcondnum**(), **svd**(), **maxloc**(), **maxv**(), **minloc**(), **minv**().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

nullspace

Synopsis
#include <numeric.h>
int nullspace(array double complex *null*[&][&], **array double complex** *a*[&][&]);

Purpose
Calculate null space of matrix *a*.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a Input matrix.

null Output array which contains the null space of matrix *a*.

Description
This function calculates an orthonormal bases for null space of matrix *a*. The null space *s* meets the following condition

$$s^t * s = I$$

and

$$a * s = 0$$

Algorithm
The computation of null space of matrix *a* is based on the singular value decomposition of matrix *a*. The SVD is formulated as

$$A = USV^T$$

Where S is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The null space can be obtained by

$$orth(i, j) = V^T(i, j),$$

with i=0, 1, 2, ..., n-1, and j=r, r+1, ... , n-1, where r is the number of non-zero singular values with tolerance $tol = \max(m, n) \times \max(S) \times$ `DBL_EPSILON`.

Example
Calculate null space of a singular matrix with rank(a) == 2.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 2 */
    array double a[M][N] = {1, 2, 3,
                            4, 5, 6,
                            7, 8, 9};
```

```
    int r = rank(a);
    array double null[M][N-r];

    nullspace(null, a);
    printf("rank(a) = %d\n", r);
    printf("null from nullspace(null, a) = \n%f\n", null);
    printf("transpose(null)*null = \n%f\n", transpose(null)*null);
}
```

Output

```
rank(a) = 2
null from nullspace(null, a) =
-0.408248
0.816497
-0.408248

transpose(null)*null =
1.000000
```

Example2
Calculate null space of a singular matrix with rank(a) == 1.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 1 */
    array double a[M][N] = {1, 2, 3,
                            1, 2, 3,
                            2, 4, 6};
    int r = rank(a);
    array double null[M][N-r];

    nullspace(null, a);
    printf("rank(a) = %d\n", r);
    printf("null from nullspace(null, a) = \n%f\n", null);
    printf("transpose(null)*null = \n%f\n", transpose(null)*null);
}
```

Output2

```
rank(a) = 1
null from nullspace(null, a) =
-0.872872 0.408248
-0.218218 -0.816497
0.436436 0.408248

transpose(null)*null =
1.000000 -0.000000
-0.000000 1.000000
```

Example3
Calculate null space of a non-singular matrix with rank(a) == 3.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
```

```
    /* non-singular matrix rank(a) == 3 */
    array double a[M][N] = {1, 2, 3,
                            5, 5, 6,
                            7, 8, 9};
    int r = rank(a);
    if(r == 3) {
      printf("rank(a) is 3, empty null space\n");
      return 0;
    }
    else  {
      array double null[M][N-r];
      nullspace(null, a);
      printf("rank(a) = %d\n", r);
      printf("null from nullspace(null, a) = \n%f\n", null);
      printf("transpose(null)*null = \n%f\n", transpose(null)*null);
    }
}
```

Output3

```
rank(a) is 3, empty null space
```

See Also

svd(), **rank**().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

oderungekutta

Synopsis
#include <numeric.h>
int oderungekutta(**double** *t*[&], **double** &*y*, **void** **param*,
double (**func*)(**double**, **double**[], **double**[], **void***),
double *t0*, **double** *tf*, **double** *y0*[&], ... /* [**double** *eps*] */);

Syntax
oderungekutta(*t*, *y*, `param`, *func*, *t0*, *tf*, *y0*)
oderungekutta(*t*, *y*, `param`, *func*, *t0*, *tf*, *y0*, *eps*)

Purpose
Numerical solution for ordinary differential equations (ODE) using Runge-Kutta method.

Return Value
If successful, this function returns the number of points calculated in the interval between *t0* and *tf*. Otherwise, it returns -1.

Parameters
t A vector which contains the *t* value in the interval between *t0* and *xf*.

y A one or two-dimensional array which contains the solution of the ODE function. For a two-dimensional array, each row of *y* corresponds to each derivative function and each column corresponds to the interval value t_i

param An option parameter for passing information to ODE functions.

func A pointer to function, the function shall be first-order differential equation.

t0 A start point of *t*.

tf A final point of *t*.

y0 A vector which contains the initial values of ODE functions.

eps The user specified tolerance. If the user does not specify this option, the value of 10^{-8} is used by default.

Description
The derivative functions *func* is specified as a pointer to a function which shall be first-order differential equation. The end-points *t0* and *tf* and initial value of differential equations *y0* can be any **real** data type. Conversion of the data to double type is performed internally. The returned **int** value is the effective number of points calculated in the interval between *t0* and *tf*. Vector *t* contains the values between *t0* and *tf* in which the ODE function is solved and the results are stored in a one-dimensional array *y* for an ordinary differential equation or in a two-dimensional array for a system of ordinary differential equations. The argument `param` is used to pass information from an application program to the ODE functions. If the optional argument *eps* is passed, the algorithm uses the user specified tolerance to decide when to stop the iterations. Otherwise,

the value of 10^{-8} is used by default.

Algorithm
Fifth-order Runge-Kutta with adaptive stepsize control.

$$y_{n+1} = y_n + c_1k_1 + c_2k_2 + c_3k_3 + c_4k_4 + c_5k_5 + c_6k_6 + O(h^6)$$

Where

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + a_2h, y_n + b_{21}k_1) \\ &\vdots \quad \vdots \quad \vdots \\ k_6 &= hf(t_n + a_6h, y_n + b_{61}k_1 + \cdots + b_{65}k_5) \end{aligned}$$

The particular values of the various constants are obtained by Cash and Karp, and given in the table below.

Cash-Karp Parameters for Embedded Runga-Kutta Method

i	a_i	b_i					c_i
1							$\frac{37}{378}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$
j =		1	2	3	4	5	

Example 1
This example shows how to solve the first-order derivative function using function **oderungekutta**(). We want to solve ODE function

$$\frac{dy}{dt} = cos(t)$$

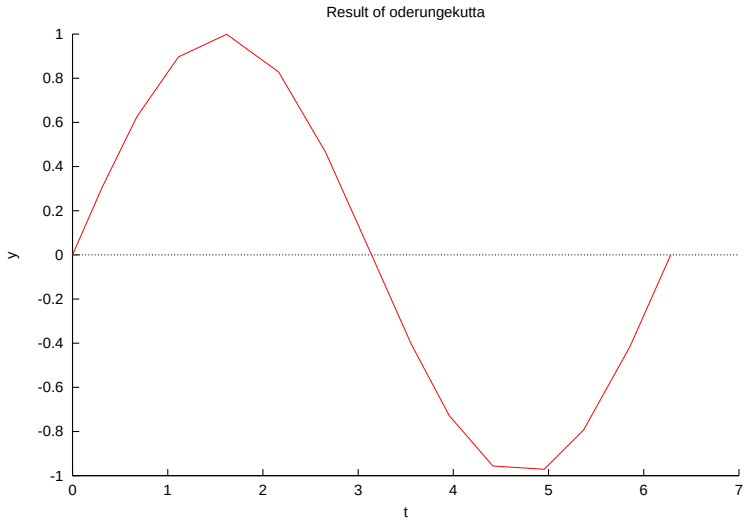
with $t_0 = 0, t_f = 2\pi, y_0 = 0$ and no more than 50 points.

```
#include <stdio.h>
#include <numeric.h>

#define NVAR 1
#define POINTS 50

int main() {
    void func(double t, double y[], double dydt[], void *param);
    double t0=0, tf=2*M_PI, y0[NVAR]={0};
    double t[POINTS], y[POINTS];
    int i, points;

/* return -1: divergent
   return >1: ok, num of points
*/
```

```
    points = oderungekutta(t, y, NULL, func, t0, tf, y0);
    printf("points = %d\n", points);
    if(points > 0) {
      printf("\n%8s %18s %15s\n","t","oderungekutta()","sin(t)");
      for (i=0; i<points; i++)
         printf("%10.4f %14.6f %14.6f\n", t[i],y[i],sin(t[i]));
      plotxy(t, y, "result of oderungekutta", "t", "y");
    }
    else
      printf("oderungekutta() failed\n");
}

void func(double t, double y[], double dydt[], void *param) {
    /* y' = cos(t) */
    dydt[0] = cos(t);
}
```

Output

```
points = 15

       t         odesolve()         sin(t)
    0.0000       0.000000       0.000000
    0.3100       0.305059       0.305059
    0.6766       0.626141       0.626141
    1.1126       0.896865       0.896865
    1.6194       0.998818       0.998818
    2.1649       0.828672       0.828671
    2.6578       0.465114       0.465114
    3.1593      -0.017658      -0.017658
    3.5575      -0.404016      -0.404016
    3.9573      -0.728193      -0.728194
    4.4151      -0.956132      -0.956133
    4.9543      -0.970888      -0.970888
    5.3682      -0.792540      -0.792540
    5.8549      -0.415294      -0.415294
    6.2832       0.000001       0.000001
```



Example 2
This example shows how to specify the tolerance. We want to solve ODE function

$$\frac{dy}{dt} = cos(t)$$

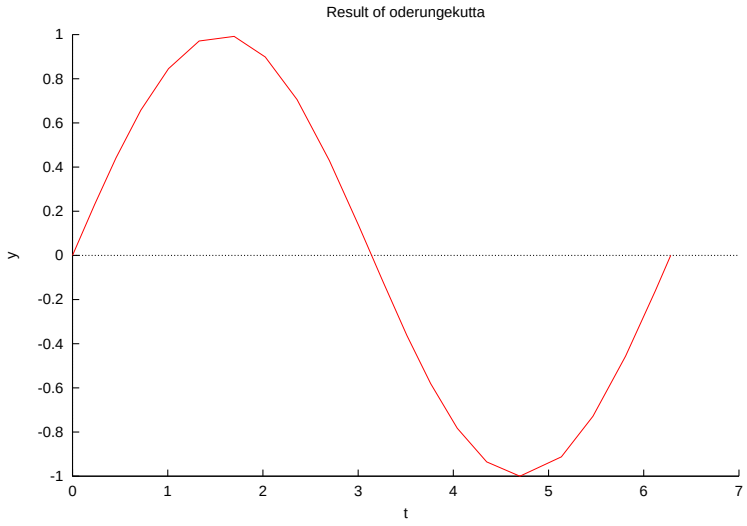
with $t_0 = 0, t_f = 2\pi, y_0 = 0$ and no more than 50 points. The numerical tolerance is FLT_EPSILON instead of 10^{-8} by default.

```
#include <stdio.h>
#include <float.h>
#include <numeric.h>

#define NVAR 1
#define POINTS 50

int main() {
    void func(double t, double y[], double dydt[], void *param);
    float t0=0, tf=2*M_PI, y0[NVAR]={0}; /* use float */
    float t[POINTS], y[POINTS];
    float tol;
    int i, points;

    tol = FLT_EPSILON; /* default tolerance is 1e-8 */
    points = oderungekutta(t, y, NULL, func, t0, tf, y0, tol);
    printf("points = %d\n", points);
    printf("\n%8s %18s %15s\n","t","oderungekutta()","sin(t)");
    for (i=0;i<points;i++)
      printf("%10.4f %14.6f %14.6f\n", t[i],y[i],sin(t[i]));
    plotxy(t, y, "result of oderungekutta", "t", "y");
}

void func(double t, double y[], double dydt[], void *param) {
    /* y' = cos(t) */
    dydt[0] = cos(t);
}
```

Output

```
points = 22

       t         odesolve()          sin(t)
    0.0000       0.000000       0.000000
    0.2319       0.229803       0.229803
    0.4576       0.441772       0.441772
    0.7181       0.657918       0.657918
    1.0080       0.845762       0.845762
    1.3295       0.971026       0.971026
    1.6979       0.991935       0.991935
    2.0250       0.898604       0.898604
    2.3583       0.705591       0.705591
    2.6974       0.429731       0.429730
    3.0073       0.133918       0.133918
    3.2598      -0.117907      -0.117907
    3.5128      -0.362781      -0.362781
    3.7617      -0.581135      -0.581135
    4.0411      -0.783034      -0.783034
```

```
4.3504      -0.935198      -0.935198
4.6982      -0.999900      -0.999900
5.1342      -0.912350      -0.912350
5.4667      -0.728744      -0.728744
5.8095      -0.456146      -0.456146
6.1219      -0.160581      -0.160580
6.2832       0.000001       0.000001
```



Example 3
This example shows how to solve the Van der Pol equation

$$\frac{d^2u}{dt^2} - \mu(1-u^2)\frac{du}{dt} + u = 0$$

where $\mu = 2, t_0 = 1, t_f = 30, u(t_0) = 1$, and $u'(t_0) = 0$;
The Van der Pol equation is reformulated as a set of first-order differential equations first.
Let

$$\begin{aligned} y_0 &= u \\ y_1 &= \frac{du}{dt} \end{aligned}$$

then

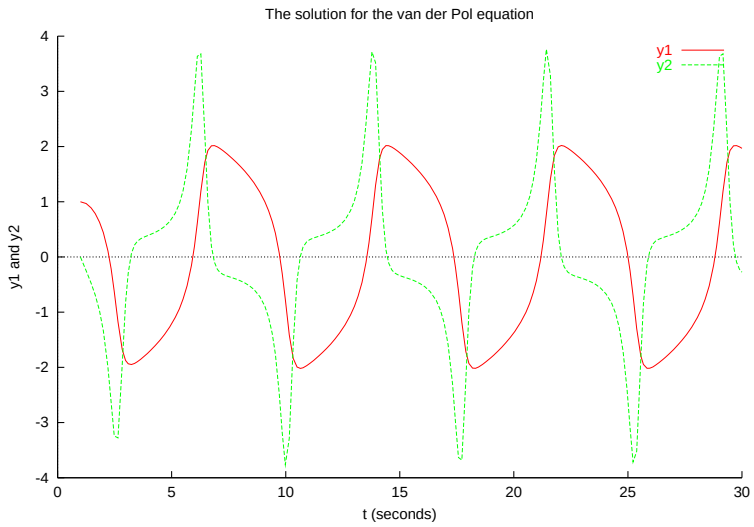
$$\begin{aligned} \frac{dy_0}{dt} &= \frac{du}{dt} \\ \frac{dy_1}{dt} &= \frac{d^2u}{dt^2} = \mu(1-u^2)\frac{du}{dt} + u = \mu(1-y_0^2)y_1 - y_0 \end{aligned}$$

Use **oderungekutta**() to solve this system of ordinary differential equations. The parameter `param` is used to pass μ from the function main() to the ODE function `func()`.

```
#include <chplot.h>
#include <numeric.h>
```

```
#define NVAR 2
#define POINTS 256
void func(double t, double y[], double dydt[], void *param) {
    double mu;
    mu = *(double*)param;
    dydt[0] = y[1];
    dydt[1]=mu*(1-y[0]*y[0])*y[1] - y[0];
}

int main() {
    double t0=1, tf=30, y0[NVAR] = {1, 0};
    double t[POINTS], y[NVAR][POINTS];
    double mu = 2;

    oderungekutta(t, y, &mu, func, t0, tf, y0);
    plotxy(t, y, "The solution for the van der Pol equation", "t (seconds)", "y1 and y2");
}
```

Output



Example 4

Function **oderungekutta**() returns the number of points calculated in the interval *t0* and *tf*. Generally the user should guess how many points it would produce and define the size of **array** *t* and *y* a little bit bigger than the guessed number of points. Function **oderungekutta**() will automatically pad the leftover space in *t* and *y* with results at *tf*.

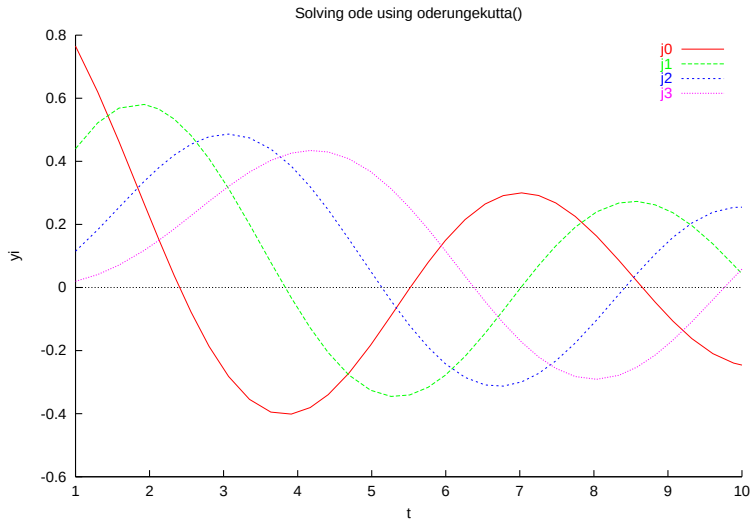
```
#include <numeric.h>
#include <stdio.h>
#include <chplot.h>

#define NVAR 4
#define POINTS 50

int main() {
    double t0=1, tf=10, y0[NVAR];
    void func(double t, double y[], double dydt[], void *param);
    double t[POINTS], y[NVAR][POINTS];
```

```
    int i, points;
    class CPlot plot;

    y0[0]=j0(t0);
    y0[1]=j1(t0);
    y0[2]=jn(2,t0);
    y0[3]=jn(3,t0);

    points = oderungekutta(t, y, NULL, func, t0, tf, y0);
    printf("points = %d\n", points);
    printf("\n%8s %18s %15s\n","t","integral","bessj(2,t)");
    for (i=0; i<points; i++)
       printf("%10.4f %14.6f %14.6f\n", t[i],y[2][i],jn(2,t[i]));

    printf("\nThe leftover space in t and y are padded with results at tf\n");
    for (i=points; i<POINTS; i++)
       printf("%10.4f %14.6f %14.6f\n", t[i],y[2][i],jn(2,t[i]));

    plotxy(t, y, "solving ode using oderungekutta()", "t", "yi", &plot);
    plot.legend("j0", 0);
    plot.legend("j1", 1);
    plot.legend("j2", 2);
    plot.legend("j3", 3);
    plot.plotting();
}

void func(double t, double y[], double dydt[], void *param) {
    dydt[0] = -y[1];
    dydt[1]=y[0]-(1.0/t)*y[1];
    dydt[2]=y[1]-(2.0/t)*y[2];
    dydt[3]=y[2]-(3.0/t)*y[3];
}
```

Output

```
points = 36

       t           integral      bessj(2,t)
    1.0000       0.114903       0.114903
    1.3003       0.183091       0.183091
    1.5864       0.253575       0.253575
    1.9283       0.336506       0.336506
    2.1222       0.379289       0.379289
    2.3324       0.419650       0.419650
    2.5595       0.454012       0.454011
    2.8041       0.477962       0.477962
    3.0669       0.486476       0.486476
    3.3491       0.474305       0.474305
    3.6380       0.438846       0.438846
    3.9140       0.384657       0.384657
    4.1688       0.319384       0.319384
    4.4123       0.246199       0.246199
    4.6909       0.153679       0.153679
    4.9838       0.052188       0.052188
    5.2611      -0.042008      -0.042008
    5.5128      -0.121128      -0.121128
    5.7616      -0.189497      -0.189496
    5.9936      -0.241616      -0.241616
```

```
    6.2600      -0.285009      -0.285009
    6.5309      -0.309122      -0.309122
    6.7762      -0.312831      -0.312831
    7.0242      -0.299369      -0.299369
    7.2551      -0.272343      -0.272343
    7.4869      -0.232833      -0.232832
    7.7487      -0.176103      -0.176103
    8.0381      -0.102923      -0.102924
    8.3387      -0.021395      -0.021396
    8.5820       0.044090       0.044090
    8.8270       0.105731       0.105730
    9.0676       0.158871       0.158870
    9.3226       0.204252       0.204251
    9.6060       0.238586       0.238586
    9.8881       0.253934       0.253934
   10.0000       0.254631       0.254630

The leftover space in t and y are padded with results at tf
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
   10.0000       0.254631       0.254630
```



See Also

derivative(), **integral1**(), **integral2**(), **integral3**(), **integration2**(), **integration3**().

References

Cash, J. R., and Karp, A. H., *ACM Transactions on Mathematical Software*, 1990. vol. 16, pp. 201-222.

odesolve

Synopsis
#include <numeric.h>
int odesolve(**double** *t*[&], **double** &*y*, **double** (**func*)(**double**, **double**[], double[]), **double** *t0*, **double** *tf*, **double** *y0*[&],
... /* [**double** *eps*] */);

Syntax
odesolve(*t*, *y*, *func*, *t0*, *tf*, *y0*)
odesolve(*t*, *y*, *func*, *t0*, *tf*, *y0*, *eps*)

Purpose
Numerical solution for ordinary differential equations (ODE). This function is obsolete. Use function **oderungekutta**().

Return Value
If successful, this function returns the number of points calculated in the interval between *t0* and *tf*. Otherwise, it returns -1.

Parameters
t A vector which contains the *t* value in the interval between *t0* and *xf*.

y A one or two-dimensional array which contains the solution of the ODE function. For a two-dimensional array, each row of *y* corresponds to each derivative function and each column corresponds to the interval value t_i

func A pointer to function, the function shall be first-order differential equation.

t0 A start point of *t*.

tf A final point of *t*.

y0 A vector which contains the initial values of ODE functions.

eps The user specified tolerance. If the user does not specify this option, the value of 10^{-8} is used by default.

Description
The derivative functions *func* is specified as a pointer to a function which shall be first-order differential equation. The end-points *t0* and *tf* and initial value of differential equations *y0* can be any **real** data type. Conversion of the data to double type is performed internally. The returned **int** value is the effective number of points calculated in the interval between *t0* and *tf*. Vector *t* contains the values between *t0* and *tf* in which the ODE function is solved and the results are stored in a one-dimensional array *y* for an ordinary differential equation or in a two-dimensional array for a system of ordinary differential equations. If the optional argument *eps* is passed, the algorithm uses the user specified tolerance to decide when to stop the iterations. Otherwise, the value of 10^{-8} is used by default.

Algorithm
Fifth-order Runge-Kutta with adaptive stepsize control.

$$y_{n+1} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6)$$

Where

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + a_2 h, y_n + b_{21} k_1) \\ &\vdots \quad\quad \vdots \quad\quad \vdots \\ k_6 &= hf(t_n + a_6 h, y_n + b_{61} k_1 + \cdots + b_{65} k_5) \end{aligned}$$

The particular values of the various constants are obtained by Cash and Karp, and given in the table below.

Cash-Karp Parameters for Embedded Runga-Kutta Method

i	a_i	b_i					c_i
1							$\frac{37}{378}$
2	$\frac{1}{5}$	$\frac{1}{5}$					0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{594}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$
j =		1	2	3	4	5	

References
Cash, J. R., and Karp, A. H., *ACM Transactions on Mathematical Software*, 1990. vol. 16, pp. 201-222.

orthonormalbase

Synopsis
#include <numeric.h>
int orthonormalbase(array double complex *orth*[&][&], **array double complex** *a*[&][&]);

Purpose
Find orthonormal bases of a matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a The input matrix.

orth An output matrix of orthonormal bases for the matrix *a*.

Description
This function calculates the orthonormal bases for a matrix. The columns of *orth* are orthonormal, and have the same space as columns of *a*. The number of columns of *orth* is the rank of *a*.

Algorithm
The computation of orthonormal bases for matrix *a* is based on the singular value decomposition(SVD) of matrix *a*. The SVD is formulated as

$$A = USV^T$$

Where S is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The orthonormal bases can be obtained by

$$orth(i,j) = U(i,j),$$

with i=0, 1, 2, ... n-1, and j=0, 1, 2, ... r-1, where r is the number of non-zero sigular values with tolerance $tol = \max(m,n) \times \max(S) \times$ `DBL_EPSILON`.

Example1
Calculate the orthonormal bases of a singular 3-by-3 matrix with rank(a) == 2.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 2 */
    array double a[M][N] = {1, 2, 3,
                            4, 5, 6,
                            7, 8, 9};
    int r = rank(a);
    array double orth[M][r];

    orthonormalbase(orth, a);
    printf("rank(a) = %d\n", r);
    printf("orth from orthonormalbase(orth, a) = \n%f\n", orth);
    printf("transpose(orth)*orth = \n%f\n", transpose(orth)*orth);
}
```

Output1

```
rank(a) = 2
orth from orthonormalbase(orth, a) =
-0.214837 0.887231
-0.520587 0.249644
-0.826338 -0.387943

transpose(orth)*orth =
1.000000 -0.000000
-0.000000 1.000000
```

Example2
Calculate the orthonormal bases of a singular 3-by-3 matrix with rank(a) == 1.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 1 */
    array double a[M][N] = {1, 2, 3,
                            1, 2, 3,
                            2, 4, 6};
    int r = rank(a);
    array double orth[M][r];

    orthonormalbase(orth, a);
    printf("rank(a) = %d\n", r);
    printf("orth from orthonormalbase(orth, a) = \n%f\n", orth);
    printf("transpose(orth)*orth = \n%f\n", transpose(orth)*orth);
}
```

Output2

```
rank(a) = 1
orth from orthonormalbase(orth, a) =
-0.408248
-0.408248
-0.816497

transpose(orth)*orth =
1.000000
```

Example3
Calculate the orthonormal bases of a singular 3-by-3 matrix with rank(a) == 3.

```
#include <numeric.h>
#define M 3
#define N 3
int main() {
    /* singular matrix rank(a) == 3 */
    array double a[M][N] = {1, 2, 3,
                            5, 5, 6,
                            7, 8, 9};
    int r = rank(a);
    array double orth[M][r];

    orthonormalbase(orth, a);
```

```
    printf("rank(a) = %d\n", r);
    printf("orth from orthonormalbase(orth, a) = \n%f\n", orth);
    printf("transpose(orth)*orth = \n%f\n", transpose(orth)*orth);
}
```

Output3

```
rank(a) = 3
orth from orthonormalbase(orth, a) =
-0.210138 0.955081 -0.208954
-0.541507 -0.291649 -0.788486
-0.814010 -0.052541 0.578470

transpose(orth)*orth =
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

See Also

nullspace(), **svd**(), **rank**().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

pinverse

Synopsis
#include <**numeric.h**>
array double pinverse(**array double** *a*[&][&])[:][:];

Purpose
Calculate MoorePenrose pseudo inverse of a matrix.

Return Value
This function returns the pseudo inverse of a matrix.

Parameters
a An input matrix.

Description
The Moore_Penrose pseudo inverse matrix *b* of matrix *a* has to meet the following four conditions:

$$a * b * a = a,$$
$$b * a * b = b,$$
$$a * b \text{ is Hermitian}$$
$$b * a \text{ is Hermitian}$$

Algorithm
The algorithm of this function is based on the function **svd**(). Any singular value less than DBL_EPSILON is treated as zero.

Example1
Calculate the Moore-Penrose pseudo inverse of a 3-by-3 singular matrix with rank(a) == 2.

```
#include <numeric.h>
#define M 3
#define N 3
/* pseudoinverse */
int main() {
    /* singular matrix rank(a) == 2 */
    array double a[M][N] = {1, 2, 3,
                            4, 5, 6,
                            7, 8, 9};
    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output1

```
rank(a) = 2
pinverse(a) =
-0.638889 -0.166667 0.305556
-0.055556 0.000000 0.055556
0.527778 0.166667 -0.194444

a*pinverse(a)*a =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000

pinverse(a)*a*pinverse(a) =
-0.638889 -0.166667 0.305556
-0.055556 0.000000 0.055556
0.527778 0.166667 -0.194444
```

Example2
Calculate the Moore-Penrose pseudo inverse of a 3-by-3 singular matrix with rank(a) == 1.

```
#include <numeric.h>
#define M 3
#define N 3
/* pseudoinverse */
int main() {
    /* singular matrix rank(a) == 1 */
    array double a[M][N] = {1, 2, 3,
                            1, 2, 3,
                            2, 4, 6};
    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output2

```
rank(a) = 1
pinverse(a) =
0.011905 0.011905 0.023810
0.023810 0.023810 0.047619
0.035714 0.035714 0.071429

a*pinverse(a)*a =
1.000000 2.000000 3.000000
1.000000 2.000000 3.000000
2.000000 4.000000 6.000000

pinverse(a)*a*pinverse(a) =
0.011905 0.011905 0.023810
0.023810 0.023810 0.047619
0.035714 0.035714 0.071429
```

Example3
Calculate the Moore-Penrose pseudo inverse of a 3-by-3 non-singular matrix with rank(a) == 3.

```
#include <numeric.h>
#define M 3
#define N 3
/* pseudoinverse */
int main() {
    /* non-singular matrix rank(a) == 3 */
    array double a[M][N] = {1, 2, 3,
                            5, 5, 6,
                            7, 8, 9};
    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output3

```
rank(a) = 3
pinverse(a) =
-0.500000 1.000000 -0.500000
-0.500000 -2.000000 1.500000
0.833333 1.000000 -0.833333

a*pinverse(a)*a =
1.000000 2.000000 3.000000
5.000000 5.000000 6.000000
7.000000 8.000000 9.000000

pinverse(a)*a*pinverse(a) =
-0.500000 1.000000 -0.500000
-0.500000 -2.000000 1.500000
0.833333 1.000000 -0.833333
```

Example4
Calculate the Moore-Penrose pseudo inverse of a 2-by-3 matrix.

```
#include <numeric.h>
#define M 2
#define N 3
/* pseudoinverse */
int main() {
     array double a[M][N] = {7, 8, 1,
                             3, 6, 4}; /* m-by-n matrix */
    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
```

```
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output4

```
rank(a) = 2
pinverse(a) =
0.128000 -0.104000
0.030769 0.061538
-0.142154 0.235692

a*pinverse(a)*a =
7.000000 8.000000 1.000000
3.000000 6.000000 4.000000

pinverse(a)*a*pinverse(a) =
0.128000 -0.104000
0.030769 0.061538
-0.142154 0.235692
```

Example5
Calculate the Moore-Penrose pseudo inverse of a 3-by-2 matrix.

```
#include <numeric.h>
#define M 3
#define N 2
/* pseudoinverse */
int main() {
     array double a[M][N] = {7, 8,
                             1, 3,
                             6, 4}; /* m-by-n matrix */
    int r;
    array double pinv[N][M];

    r = rank(a);
    pinv = pinverse(a);
    printf("rank(a) = %d\n", r);
    printf("pinverse(a) = \n%f\n", pinv);
    printf("a*pinverse(a)*a = \n%f\n", a*pinv*a);
    printf("pinverse(a)*a*pinverse(a) = \n%f\n", pinv*a*pinv);
}
```

Output5

```
rank(a) = 2
pinverse(a) =
-0.053595 -0.209150 0.264052
0.139869 0.228758 -0.201307

a*pinverse(a)*a =
7.000000 8.000000
1.000000 3.000000
6.000000 4.000000

pinverse(a)*a*pinverse(a) =
-0.053595 -0.209150 0.264052
0.139869 0.228758 -0.201307
```

See Also
rank(), **svd**(), **qrdecomp**(), **inverse**().
cinverse().

References

polycoef

Synopsis
#include <numeric.h>
int polycoef(array double complex *p*[&], **array double complex** *x*[&]);

Purpose
Calculates the coefficients of the polynomial with specified roots.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
p An output array containing the calculated coefficients of the polynomial.

x An input one-dimensional array containing the roots of the polynomial.

Description
This function calculates the coefficients of a polynomial with specified roots expressed as an array *x*. The coefficients are in the order of descending powers. The polynomial it represents is $p_0x^n + ... + p_{n-1}x + p_n$.

Example1
Calculate the coefficients of polynomials

$$x^3 - 6x^2 - 72x - 27 = 0$$

and

$$x^4 - 12x^3 - 25x + 116 = 0$$

The roots of the polynomials are calculated by function **roots**().

```
#include <numeric.h>
int main() {
     array double x1[3], p1[4]  = {1, -6, -72, -27}; /* xˆ3-6xˆ2-72x-27 =0*/
     array double x2[4], p2[5]  = {1, -12, 0, 25, 116}; /* xˆ4-12xˆ3-25x+116 =0 */
     array double complex z2[4], zp2[5];
     array double complex z3[4], p3[5] = {complex(3,4), complex(4,2),
                              complex(5,3), complex(2,4), complex(1,5)};
     int status;


     roots(x1, p1);
     polycoef(p1, x1);
     printf("p1 from polycoef(p1, x1) = %0.2f\n", p1);

     /* xˆ4-12xˆ3-25x+116 =0  has two complex roots and two real roots */
     roots(x2, p2);
     roots(z2, p2);
     polycoef(p2, x2);
```

```
     printf("p2 from polycoef(p2, x2) =  %0.2f\n", p2);
     polycoef(zp2, z2);
     printf("zp2 from polycoef(zp2, z2) =\n %0.2f\n", zp2);

     roots(z3, p3);
     status = polycoef(p3, z3);
     if(status == 0) {
       printf("p3 from polycoef(p3, z3) = \n %0.2f\n", p3);
       printf("complex(3,4)*p3 from polycoef(p3, z3) = \n %0.2f\n", complex(3,4)*p3);
       roots(z3, p3);
       printf("z3 from roots(z3, p3) = \n %0.2f\n", z3);
     }
     else
       printf("polycoef(p3, z3) failed\n");
}
```

Output1

```
p1 from polycoef(p1, x1) = 1.00 -6.00 -72.00 -27.00

p2 from polycoef(p2, x2) =  1.00 NaN NaN NaN NaN

zp2 from polycoef(zp2, z2) =
 complex(1.00,0.00) complex(-12.00,0.00) complex(0.00,0.00) complex(25.00,0.00)
complex(116.00,0.00)

p3 from polycoef(p3, z3) =
 complex(1.00,0.00) complex(0.80,-0.40) complex(1.08,-0.44) complex(0.88,0.16)
complex(0.92,0.44)

complex(3,4)*p3 from polycoef(p3, z3) =
 complex(3.00,4.00) complex(4.00,2.00) complex(5.00,3.00) complex(2.00,4.00)
complex(1.00,5.00)

z3 from roots(z3, p3) =
 complex(0.23,1.28) complex(0.43,-0.73) complex(-0.75,-0.71) complex(-0.70,0.55)
```

Example2
The coefficients of polynomial with linearly spaced roots are calculated.

```
#include <numeric.h>
/*  Wilkinson's famous example */
#define N 10
int main() {
     array double x[N], p[N+1];
     linspace(x, 1, N);
     printf("x = %f\n", x);
     polycoef(p, x);
     printf("p = %f\n", p);
     roots(x, p);
     printf("x = %f\n", x);
}
```

Output2

```
x = 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000
```

```
p = 1.000000 -55.000000 1320.000000 -18150.000000 157773.000000 -902055.000000
3416930.000000 -8409500.000000 12753576.000000 -10628640.000000 3628800.000000


x = 9.000000 10.000000 8.000000 7.000000 6.000000 5.000000 4.000000 3.000000 2.000000
1.000000
```

Example3
The roots of a characteristic polynomial are calculated by function **eigensystem**() from a matrix, and then the coefficients are calculated again using function **polycoef**().

```
#include <numeric.h>
int main() {
     array double a[3][3] = {0.8, 0.2, 0.1,
                             0.1, 0.7, 0.3,
                             0.1, 0.1, 0.6};
     array double evalues[3],evectors[3][3];
     array double p[4];

     eigensystem(evalues, NULL, a);
     polycoef(p, evalues);
     printf("evalues = %f\n", evalues);
     printf("characteristic polynomial of matrix a = %f\n", p);
}
```

Output3

```
evalues = 1.000000 0.600000 0.500000

characteristic polynomial of matrix a = 1.000000 -2.100000 1.400000 -0.300000
```

See Also
roots(), **charpolycoef**().

References

polyder

Synopsis
#include <numeric.h>
int polyder(**array double complex** *y*[&], **array double complex** *x*[&]);

Syntax
polyder(*y*, *x*)

Purpose
Take derivative of a polynomial.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y A vector of coefficients of the derivative of the original polynomial.

x A vector of coefficients of the original polynomial.

Description
The vector of coefficient of polynomial *x* can be of any supported arithmetic data type and size. Conversion of the data to double complex type is performed internally. If vector *x* is of real type with size n, then the vector *y* for the derivative is of real type with size $(n-1)$. If vector *x* is of complex type, then the vector *y* for the derivative is of complex type.

Algorithm
Given a polynomial $P(t)$ with coefficients of x_i, for $i = 0, 1, \cdots, N$. That is

$$P(t) = x_0 t^n + x_1 t^{n-1} + x_2 t^{n-2} + \cdots + x_{n-1} t + x_n$$

Then the first derivative of this polynomial is

$$\begin{aligned} P'(t) &= n * x_0 t^{n-1} + (n-1) * x_1 t^{n-2} + (n-2) * x_2 t^{n-3} + \cdots + x_{n-1} \\ &= y_0 t^{n-1} + y_1 t^{n-1} + y_2 t^{n-3} + \cdots + y_{n-1} \end{aligned}$$

Example
In this example:
polynomial: $(1-i)x^2 + 2x + (3+i) = 0$
derivative : $(2+i2)x + 2 = 0$

polynomial: $x^2 + 2x + 3 = 0$
derivative : $2x + 2 = 0$

```
#include <stdio.h>
#include <numeric.h>
```

```
#define N 3        /* data array size */

int main() {
   int i;
   array double complex x[N],y[N-1];
   array double x1[N],y1[N-1];
   for (i=0;i<N;i++) {
      x[i]=complex(1+i,i-1);
      x1[i]=i+1;
   }

   printf("Coef. of polynomial b\n");
   printf("x=%6.3f\n", x);

   polyder(y,x);            /* derivative of x polynomial */
   printf("Coef. of derivative of polynomial of b\n");
   printf("y=%6.3f\n", y);

   printf("Coef. of polynomial d\n");
   printf("x1=%6.3f",x1);

   polyder(y1,x1);            /* derivative of B polynomial */
   printf("Coef. of derivative of polynomial of d\n");

   printf("y1=%6.3f",y1);
}
```

Output

```
Coef. of polynomial b
x=complex( 1.000,-1.000) complex( 2.000, 0.000) complex( 3.000, 1.000)


Coef. of derivative of polynomial of b
y=complex( 2.000,-2.000) complex( 2.000, 0.000)

Coef. of polynomial d
x1= 1.000  2.000  3.000
Coef. of derivative of polynomial of d
y1= 2.000  2.000
```

See Also
polyder2().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyder2()

Synopsis
#include <numeric.h>
int polyder2(**array double complex** *q*[&], **array double complex** *r*[&], **array double complex** *u*[&],
array double complex *v*[&]);

Syntax
polyder2(*q*, *NULL*, *u*, *v*)
polyder2(*q*, *r*, *u*, *v*)

Purpose
Calculate the derivative of product or quotient of two polynomials.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
q A vector contains the coefficients of the derivative of the product $(u * v)'$ of two polynomials *u* and *v* or coefficients of the numerator of the quotient $(u/v)'$ of two polynomials *u* and *v*.

r If input *r* is NULL the function implements the derivative of the product $(u * v)'$ of two polynomials *u* and *v*. Otherwise, it is a vector which contains the coefficients of the denominator from the derivative of the quotient $(u/v)'$ of two polynomials *u* and *v*.

u A vector of polynomial coefficients.

v A vector of polynomial coefficients.

Description
The operation of the derivative of a product or quotient of two polynomials *u* and *v* depends on argument *r*. If NULL is passed to argument *r*, functon **polyder**2() calculates the derivative of the product $(u * v)'$ of two polynomials *u* and *v*. Otherwise, it calculates the derivative of the quotient $(u/v)'$. The coefficient vector of polynomials *u* and *v* can be of any supported arithmetic data type with sizes n and m, respectively. Conversion of the data to double complex type is performed internally. If both vectors *u* and *v* are real type, the vector *q* with size $n + m - 2$ and *r* with size $2 * m - 1$ are real type. If one of vectors *u* and *v* is complex type, vectors *q* and *r* are complex type.

Algorithm
Given two polynomials $U(x)$ and $V(x)$

$$\begin{aligned} U(x) &= u_0x^n + u_1x^{n-1} + u_2x^{n-2} + \cdots + u_{n-1}x + u_n \\ V(x) &= v_0x^m + v_1x^{m-1} + v_2x^{m-2} + \cdots + v_{m-1}x + v_m \end{aligned}$$

The derivative of the product of two polynomials $U(x) * V(x)$ becomes

$$\begin{aligned} Q(x) &= (U(x)V(x))' = U'(x)V(x) + V'(x)U(x) \\ R(x) &= NULL \end{aligned}$$

The derivative of the quotient of two polynomials $U(x)/V(x)$ becomes

$$\left(\frac{U(x)}{V(x)}\right)' = \frac{Q(x)}{R(x)} = \frac{U'(x)V(x) - V'(x)U(x)}{V^2(x)}$$

where $Q(x)$ and $R(x)$ are the numerator and denominator of the derivative, respectively. The polynomials for $Q(x)$, applicable to derivative of either the product or quotient of two polynomials, and $R(x)$ can be represented as

$$\begin{aligned} Q(x) &= q_0 x^{n+m-1} + q_1 x^{n+m-2} + \cdots + q_{n+m-2}x + q_{n+m-1} \\ R(x) &= r_0 x^{2m-1} + r_1 x^{2m-2} + \cdots + r_{2m-2}x + r_{2m-1} \end{aligned}$$

Example
Given

$$\begin{aligned} U(x) &= x + 2 \\ V(x) &= x^2 + 2x \end{aligned}$$

then

$$\begin{aligned} (U(x)V(x))' &= 3x^2 + 8x + 4 \\ \left(\frac{U(x)}{V(x)}\right)' &= \frac{-x^2 - 4x - 4}{x^4 + 4x^3 + 4x^2} \end{aligned}$$

```
#include <stdio.h>
#include <numeric.h>

#define N 3        /* data array size */
#define M 2        /* response array size */

int main() {
   int i;
   array double complex v[N],u[M],q[N+M-2],r[2*N-1];
   array double v1[N],u1[M],q1[N+M-2],r1[2*N-1];
      v[0]=complex(1,1);v[1]=complex(2,2);v[2]=0;
      v1[0]=1;v1[1]=2;v1[2]=0;
   for (i=0;i<M;i++) {
      u[i]=complex(i+1,i);
      u1[i]=i+1;
   }

   printf("Coef. of polynomial v=%4.2f\n", v);
   printf("Coef. of polynomial u=%4.2f\n", u);

   polyder2(q, NULL, u, v);         /* derivative of v*u polynomial */
   printf("Coef. of deriv of poly v*u=%4.2f\n", q);
   printf("\n");

   polyder2(q, r, u, v);      /* derivative of u/v polynomial */
   printf("Coef. of derivative of polynomial of u/v\n");
   printf("Numerator: %4.2f\n", q);
   printf("Denormator:%4.2f\n", r);

   printf("Coef. of polynomial v=%6.3f\n", v1);
   printf("Coef. of polynomial u=%6.3f\n", u1);
```

```
   polyder2(q1, NULL, u1, v1);         /* derivative of b*a polynomial */
   printf("Coef. of derivative of polynomial of b*a=%6.3f\n", q1);
   printf("\n");

   polyder2(q1, r1, u1, v1);      /* derivative of a/b polynomial */
   printf("Coef. of derivative of polynomial of a/b=\n");
   printf("Numerator: %6.3f\n", q1);
   printf("Denormator:%6.3f\n", r1);
}
```

Output

```
Coef. of polynomial v=complex(1.00,1.00) complex(2.00,2.00) complex(0.00,0.00)

Coef. of polynomial u=complex(1.00,0.00) complex(2.00,1.00)

Coef. of deriv of poly v*u=complex(3.00,3.00) complex(6.00,10.00) complex(2.00,6.00)


Coef. of derivative of polynomial of u/v
Numerator: complex(-1.00,-1.00) complex(-2.00,-6.00) complex(-2.00,-6.00)

Denormator:complex(0.00,2.00) complex(0.00,8.00) complex(0.00,8.00) complex(0.00,0.00)
complex(0.00,-0.00)

Coef. of polynomial v= 1.000  2.000  0.000

Coef. of polynomial u= 1.000  2.000

Coef. of derivative of polynomial of b*a= 3.000  8.000  4.000


Coef. of derivative of polynomial of a/b=
Numerator: -1.000 -4.000 -4.000

Denormator: 1.000  4.000  4.000  0.000 -0.000
```

See Also
polyder().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyeval

Synopsis
#include <numeric.h>
double polyeval(**array double** *c*[&], **double** *x*, ... /* [**array double** *dp*[&] */);

Syntax
polyeval(*c*, *x*)
polyeval(*c*, *x*, *dp*)

Purpose
Calculate the value of a polynomial and its derivatives.

Return Value
This function returns the value of a polynomial and its derivatives at a given point.

Parameters
c A vector of coefficients of a polynomial.

x A value of point x in which the value of polynomial is evaluated.

dp A vector of dimension m, it contains the values from the 1st to mth order derivatives of the polynomial at point *x*.

Description
The vector of coefficients of polynomial *c* can be of any supported real type and dimension n. The value of *x* can be of real type. Conversion of the data to double is performed internally. The return data is double type of the value of the polynomial. If the optional argument *dp* is specified, which shall be double type of dimension m, it passes back the values of 1st to mth order derivatives of the polynomial.

Algorithm
Given a polynomial

$$P(x) = c_0x^n + c_1x^{n-1} + \cdots + c_{n-3}x^3 + c_{n-2}x^2 + c_{n-1}x + c_n$$

The calculation of the polynomial value and its derivatives are implemented using the following algorithm

$$\begin{aligned}
P &= (((\cdots(c_0x + c_1)x + \cdots + c_{n-3})x + c_{n-2})x + c_{n-1})x + c_n \\
P' &= ((\cdots(nc_0x + (n-1)c_1)x + \cdots + 3c_{n-3})x + 2c_{n-2})x + c_{n-1} \\
P'' &= (\cdots(n(n-1)c_0x + (n-1)(n-2)c_1)x + \cdots + 3*2c_{n-3})x + 2c_{n-2} \\
\vdots &= \vdots \quad \vdots \\
P^{(n-1)} &= n(n-1)(n-2)\cdots 3*2c_0x + (n-1)(n-2)\cdots 3*2*1c_1 \\
P^{(n)} &= n(n-1)(n-2)\cdots 3*2*1c_0 \\
P^{(n+1)} &= 0
\end{aligned}$$

Example
Calculate the value of the polynormal

$$C(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

at point $x = 0.1$ and its first, second and third order derivatives.

```
#include <numeric.h>
#include <stdio.h>

#define NC 6
#define ND 3
#define NP 10

int main() {
   int i,j,k;
   float c[NC]={1.0,-5.0,10.0,-10.0,5.0,-1.0};
   float vp[ND], d[ND+1],x;
   string_t title = "   x    polynomial    first deriv   second deriv    third deriv";

   x=0.1;
   d[0] = polyeval(c,x,vp);
   for (j=0;j<ND;j++) d[j+1]=vp[j];
   printf("%s\n", title);
   printf("%4.3f",x);
   for (i=0;i<=ND;i++) {
      printf("%14.6f",d[i]);
   }
   printf("\n\n");

   x=0.1;
   printf("x=%f\tf(x)=%f\n",x,polyeval(c,x,vp));
}
```

Output

```
   x    polynomial    first deriv   second deriv    third deriv
0.100     -0.590490      3.280500    -14.580000     48.599998

x=0.100000 f(x)=-0.590490
```

see also
polyevalarray(), **cpolyeval**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press,1997.

polyevalarray

Synopsis
#include <numeric.h>
int polyevalarray(array double complex *&val*, *c*[&], *&x*);

Syntax
polyevalarray(*val*, *c*, *x*)

Purpose
Polynomial evaluation for a sequence of points.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
val The array of output which contains the polynomial evaluation values for points *x*. The dimension of array *val* is the same as that of **array** *x*.

c A vector of coefficients for the polynomial.

x An array which contains the value of points in which the polynomial is evaluated.

Description
The vector *c* with the coefficients of a polynomial can be of any supported arithmetic data type and size. The value of *x* can be any arithmetic type. Conversion of the data to double complex is performed internally. Array *val*, the same dimension and size as *x*, contains the evaluation results. If both *c* and *x* are real type, *val* is real type. Otherwise, it shall be complex type.

Algorithm
Given polynomial

$$Y(x_i) = c_0 x_i^n + c_1 x_i^{n-1} + c_2 x_i^{n-1} + \cdots + c_{n-1} x_i + c_n$$

the polynomial evaluation is implemented

$$Y_i = (((\cdots (c_0 x_i + c_1) x_i + c_2) x_i + c_3) x_i + \cdots + c_{n-1}) x_i + c_n$$

where subscript i traverses all elements of array *x*.

Example
Evaluate polynormal

$$C(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

at points $x = \{0.1, 0.2, \cdots, 1.9, 2.0\}$.

```
#include <numeric.h>
#include <stdio.h>

#define NC 6
#define N 10

int main()
{
   int i,j,k;
   array double c[NC]={1.0,-5.0,10.0,-10.0,5.0,-1.0};
   array double x[N],val[N];

   for (i=0; i<N; i++)
      x[i]=0.2*(i+1);
   printf("x value    function evalulate\n");
   polyevalarray(val,c,x);
   for (i=0; i<N; i++)
      printf("%f    %f\n",x[i],val[i]);
}
```

Output

```
x value    function evalulate
0.200000    -0.327680
0.400000    -0.077760
0.600000    -0.010240
0.800000    -0.000320
1.000000    0.000000
1.200000    0.000320
1.400000    0.010240
1.600000    0.077760
1.800000    0.327680
2.000000    1.000000
```

see also
polyeval(), **cpolyeval**().

References
William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

polyevalm

Synopsis
#include <numeric.h>
int polyevalm(array double complex *y[&][&]*, **array double complex** *c[&]*, **array double complex** *x[&][&]*);

Syntax
polyevalm(*y*, *c*, *x*)

Purpose
Evaluate a real or complex polynomial with matrix argument.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input square matrix. It contains data to be evaluated.

c A vector of coefficients of a polynomial. It can be **real** or **complex** data type.

y Output square matrix which contains data of the evaluated polynomial with given matrix argument *x*.

Description
This function evaluates the polynomial with given matrix argument *x*. In this function, The input square matrix *x* can be of any supported arithemetic data type. The polynomial is specified by the coefficient vector *c*, which can be any supported arithemetic data type. The output matrix *y* could be **real** or **complex** data type as required.

Example
Evaluation of real/complex polynomials of real/complex matrices.

```
#include <numeric.h>
#include <stdio.h>

#define NC 6
#define ND 3
#define N 20

int main()
{
   int i,j,k;
   array double c[4]={1,2,3,4};
   array double complex c1[4]={complex(2,1),2,3,4};
   array double x[3][3]={1,2,3,
                         3,4,5,
                         6,7,8};
   array double y[3][3];
   array double complex x1[3][3]={1,2,complex(2,3),
                         3,4,5,
                         6,7,complex(5,8)};
```

```
   array double complex y1[3][3];
   polyevalm(y,c,x);
   printf("x=\n%f\n",x);
   printf("y=\n%f\n",y);
   polyevalm(y1,c1,x1);
   printf("x1=\n%5.2f\n",x1);
   printf("y1=\n%5.2f\n",y1);


   return 0;
}
```

Output

```
x=
1.000000 2.000000 3.000000
3.000000 4.000000 5.000000
6.000000 7.000000 8.000000

y=
397.000000 501.000000 609.000000
729.000000 931.000000 1125.000000
1233.000000 1566.000000 1903.000000

x1=
complex( 1.00, 0.00) complex( 2.00, 0.00) complex( 2.00, 3.00)
complex( 3.00, 0.00) complex( 4.00, 0.00) complex( 5.00, 0.00)
complex( 6.00, 0.00) complex( 7.00, 0.00) complex( 5.00, 8.00)

y1=
complex(-82.00,685.00) complex(-64.00,878.00) complex(-706.00,603.00)

complex(849.00,1110.00) complex(1137.00,1361.00) complex(-195.00,1912.00)

complex(-12.00,2034.00) complex(105.00,2594.00) complex(-1864.00,2023.00)
```

See Also
logm(), **cfunm**(), **polyeval**(), **polyevalarray**(), **cpolyeval**(), **sqrtm**().

References
G. H. Golub, C. F. Van Loan, Matrix Computations Third edition, The Johns Hopkins University Press, 1996

polyfit

Synopsis
#include <**numeric.h**>
int polyfit(**double** *a*[&], **double** *x*[&], **double** *y*[&], ... /* [**double** *sig*[ ], **int** *ia*[ ],
double *covar*[ ][ ], **double** **chisq*] */);

Syntax
polyfit(*a, x, y*)
polyfit(*a, x, y, sig*)
polyfit(*a, x, y, sig, ia*)
polyfit(*a, x, y, sig, ia, covar*)
polyfit(*a, x, y, sig, ia, covar, chisq*)

Purpose
Fit a set of data points *x*, *y* to a polynomial function.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a An output array which contains coefficients of polynomial.

x An array which contains variable values of the data set.

y An array which contains function values of the data set.

sig An input array which contains individual standard (measurement error) deviations of data set. It can be set to 1 if unknown.

ia An input array. Those components of nonzero entries shall be fitted, and those components of zero entries shall be held at their input values.

covar The Covariance matrix of fitting results, describing the fit of data to the model. See algorithm for definition.

chisq Chi-square of fitting.

Description
Given a set of data points expressed in arrays *x* and *y* with the same dimension, use χ^2 minimization to fit the data set using a polynomial coefficients *a*. The fitted coefficients are passed by array *a*. The program also passes χ^2, covariance matrix *covar*. If the parameters of *ia* are held fixed, the corresponding components of covariance matrix will be zero.

Algorithm
The algorithm of the function **polyfit**() is based on that of the function **curvefit**(). For polynomial fitting, the base functions for **curvefit**() are set to the terms of polynomials internally. The general form of the fitting formula is

$$y(x) \quad = \quad \sum_{k=1}^{M} a_k x^k$$

The coefficients a_k are determined by minimizing chi-square which is defined as

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - \sum_{k=1}^{M} a_k x_i^k)}{\sigma_i} \right)^2$$

where σ_i is the standard deviation of the ith data point. If the standard deviations are unknown, they can be set to the constant value $\sigma = 1$. By defining a $n \times m$ matrix α with element (k, j) as

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{x_i^j x_i^k}{\sigma_i^2}$$

The element (k, j) of the covariance matrix can be expressed as

$$cover_{kj} = [\alpha]_{kj}^{-1}$$

Example1

Fit the data set created by the polynomial

$$8x^4 + 5x^3 + 3x^2 + 6x + 7$$

The deviation with a fraction of a uniform random number is added to data set.

```
#include <numeric.h>
#include <stdio.h>
#define NPT 100                 /* Number of data points */
#define NTERM 5                 /* Number of terms */

int main() {
    int i,j,status;
    array double  u[NPT],x[NPT],y[NPT],a[NTERM];

    /* Create a data set of NTERM order polynomial with uniform random deviation*/
    linspace(x,0.1,0.1*NPT);
    y = 8*x.*x.*x.*x + 5*x.*x.*x + 3*x.*x + 6*x  + (array double[NPT])(7);

    /* put uniform random deviation on data set */
    urand(u);
    y += 0.1*u;
    status=polyfit(a,x,y);
    if(status) printf("Abnormal fit");
    printf("\n%11s\n","Coefficients");
    for (i=0;i<NTERM;i++)
        printf("  a[%1d] =  %8.6f \n",i,a[i]);
    printf("y = %f at x = %f\n", polyeval(a, 2.0), 2.0);
}
```

Output1

```
Coefficients
  a[0] =  8.000048
  a[1] =  4.999537
  a[2] =  2.998743
  a[3] =  6.015915
  a[4] =  7.033636
y = 199.057500 at x = 2.000000
```

Example2

Fit the data set created by the polynomial

$$8x^4 + 5x^3 + 3x^2 + 6x + 7$$

The deviation with a fraction of a uniform random number is added to data set. This example uses all the options of arguments of function **polyfit**(). The coefficients $a[1]$ and $a[3]$ are fixed and the standard deviation of data set is assumed to be a constant 0.1. The uncertainty which is defined as $dev[i] = \sqrt{covar[i][i]}$ is calculated.

```
#include <stdio.h>
#include <numeric.h>
#define NPT 100                 /* Number of data points */
#define NTERM 5                 /* Number sof term */
#define SPREAD 0.1              /* Number sof term */

int main() {
    int i,j,status,ia[NTERM];
    array double u[NPT],x[NPT],y[NPT],a[NTERM],dev[NTERM];
    array double sig[NPT],covar[NTERM][NTERM];
    double chisq;

    /* Create a data set of NTERM order polynomial with gaussian deviation*/
    linspace(x,0.1,0.1*NPT);
    y = 8*x.*x.*x.*x + 5*x.*x.*x + 3*x.*x + 6*x  + (array double[NPT])(7);

    /* put uniform random deviation on data set */
    urand(u);
    y += 0.1*u;
    sig=(array double[NPT])(SPREAD);

    ia[0] = 1; ia[1] = 0; ia[2] = 1; ia[3] = 0, ia[4] = 1;
    a[1] = 5; a[3] = 6;

    status=polyfit(a,x,y,sig,ia,covar,&chisq);
    dev = sqrt(diagonal(covar));
    if(status) printf("Abnormal fit");
    printf("\n%11s%23s\n","Coefficients");
    printf("\n%11s %21s\n","parameter","uncertainty");
    for (i=0;i<NTERM;i++)
        printf("  a[%1d] = %8.6f %12.6f\n",
            i,a[i],dev[i]);
    printf("chi-square = %12f\n",chisq);
    printf("full covariance matrix\n");
    printf("%12f",covar);
}
```

Output2

```
Coefficients
  parameter           uncertainty
  a[0] = 8.000007     0.000013
  a[1] = 5.000000     0.000000
  a[2] = 2.999413     0.001165
  a[3] = 6.000000     0.000000
  a[4] = 7.057378     0.018870
chi-square =     7.414554
```

```
full covariance matrix
    0.000000     0.000000    -0.000000     0.000000     0.000000
    0.000000     0.000000     0.000000     0.000000     0.000000
   -0.000000     0.000000     0.000001     0.000000    -0.000016
    0.000000     0.000000     0.000000     0.000000     0.000000
    0.000000     0.000000    -0.000016     0.000000     0.000356
```

See Also
curvefit(), **polyeval**(), **std**().

References
William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numeric Recipes in C*, Second Edition, Cambridge University Press, 1997.

product

Synopsis
#include <**numeric.h**>
double product(**array double** &*a*, ... /* [**array double** *v*[:]] */);

Syntax
product(a)
product(a, v)

Purpose
Calculate products of all elements and products of the elements of each rows of an array.

Return Value
This function returns the product of all elements.

Parameters
a The input array for which the products are calculated.

v The output array which contains the products of each row of a two-dimensional array.

Description
This function calculates the product of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate the products of each row. The product of all elements of the array is returned by the function and the products of the elements of each row are passed out by the argument *v*. The input array can be of any dimension and any arithmetic data types.

Example1
Calculate the products of all elements of arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    double prod;

    prod = product(a);
    printf("prod(a) = %f\n", prod);
    prod = product(b);
    printf("product(b) = %f\n", prod);
    prod = product(b1);
    printf("product(b1) = %f\n", prod);
    prod = product(c);
    printf("product(c) = %f\n", prod);
}
```

Output1

```
prod(a) = 6.000000
product(b) = 720.000000
product(b1) = 720.000000
product(c) = 0.000000
```

Example2

Calculate the products of the elements of each row of the two-dimensional arrays with different data types and dimensions.

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double productv1[2], productv2[3];

    product(a, productv1);
    printf("product(a, productv1) = %f\n", productv1);
    product(b,productv2);
    printf("product(b,productv2) = %f\n", productv2);
}
```

Output2

```
product(a, productv1) = 6.000000 120.000000

product(b,productv2) = 24.000000 1680.000000 24.000000
```

See Also

cproduct(), **cumprod**(), **mean**(), **median**(), **sum**().

References

qrdecomp

Synopsis
#include <**numeric.h**>
int qrdecomp(**array double complex** *a*[&][&], **array double complex** *q*[&][&],
array double complex *r*[&][&]);

Purpose
Perform the orthogonal-triangular QR decomposition of a matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a Input matrix.

q Output unitary matrix.

r Output upper triangular matrix.

Description
This function performs a QR decomposition of a matrix *a* so that $a = qr$. The output *q* is a unitary matrix and r is an upper triangular matrix. Assume the size of input matrix *a* is $m \times n$. If $m \leq n$, there is only one type of output matrices *q* and *r*. The size of output matrix *q* is $m \times m$ and matrix *r* is $m \times n$. If $m > n$, there are two types of output matrices *q* and *r*. One is full size in which matrix *q* is $m \times m$ and *r* is $m \times n$. Another is an economy size in which matrix *q* is $m \times n$ and *r* is $n \times n$. This function checks the size of arguments and automatically selects the corresponding output type.

Example1
Perform QR decomposition of $4 \times 4, 2 \times 4$ and 4×2 matrices. The 4×2 matrices have two different types of output corresponding to the argument sizes passed in.

```
#include <numeric.h>
int main() {
     int m = m, n = 4;
     array double a[4][4] = {1, 5, -7, 4,
                             3, 2,  5, 2,
                             3, 2,  5, 2,
                             3, 9,  5, 2}; /* a[m][n] */
     array double q[n][n];
     array double r[n][n];

     qrdecomp(a, q, r);
     printf("q = \n%f\n", q);
     printf("r = \n%f\n", r);
     printf("q*r = \n%f\n", q*r);

     m = 2;
     n = 4;
     array double a1[2][4] = {1, 5, -7, 4,
                              3, 2,  5, 2}; /* a[m][n] */
     array double q1[m][m];    // m<n
     array double r1[m][n];
```

```
    qrdecomp(a1, q1, r1);
    printf("q1^T*q1 = \n%f\n", transpose(q1)*q1);
    printf("q1 = \n%f\n", q1);
    printf("r1 = \n%f\n", r1);
    printf("q1*r1 = \n%f\n", q1*r1);

    array double a2[4][2] = {1, 5,
                            -7, 4,
                             3, 2,
                             5, 2}; /* a[m][n] */
    m = 4;
    n = 2;
    array double q2[m][n], q21[m][m];     // m>=n
    array double r2[n][n], r21[m][n];
    int status;

    status = qrdecomp(a2, q21, r21);
    if(status == 0) {
        printf("q21^T*q21 = \n%f\n", transpose(q21)*q21);
        printf("q21 = \n%f\n", q21);
        printf("r21 = \n%f\n", r21);
        printf("q21*r21 = \n%f\n", q21*r21);
    }
    else
        printf("error: numerical error in qrdecomp()\n");

    status = qrdecomp(a2, q2, r2);
    if(status == 0) {
        printf("q2^T*q2 = \n%f\n", transpose(q2)*q2);
        printf("q2 = \n%f\n", q2);
        printf("r2 = \n%f\n", r2);
        printf("q2*r2 = \n%f\n", q2*r2);
    }
    else
        printf("error: numerical error in qrdecomp()\n");
}
```

Output1

```
q =
-0.188982 0.511914 0.837991 0.000000
-0.566947 -0.405266 0.119713 0.707107
-0.566947 -0.405266 0.119713 -0.707107
-0.566947 0.639893 -0.518756 -0.000000

r =
-5.291503 -8.315218 -7.181325 -4.157609
0.000000 6.697548 -4.436592 1.706382
0.000000 0.000000 -7.262591 2.793304
0.000000 0.000000 0.000000 0.000000

q*r =
1.000000 5.000000 -7.000000 4.000000
3.000000 2.000000 5.000000 2.000000
3.000000 2.000000 5.000000 2.000000
3.000000 9.000000 5.000000 2.000000

q1^T*q1 =
1.000000 0.000000
```

```
0.000000 1.000000

q1 =
-0.316228 -0.948683
-0.948683 0.316228

r1 =
-3.162278 -3.478505 -2.529822 -3.162278
0.000000 -4.110961 8.221922 -3.162278

q1*r1 =
1.000000 5.000000 -7.000000 4.000000
3.000000 2.000000 5.000000 2.000000

q21^T*q21 =
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000

q21 =
-0.109109 -0.730552 -0.408074 -0.536530
0.763763 -0.491027 0.173257 0.381499
-0.327327 -0.323359 0.873744 -0.157694
-0.545545 -0.347312 -0.200072 0.736021

r21 =
-9.165151 0.763763
0.000000 -6.958209
0.000000 0.000000
0.000000 0.000000

q21*r21 =
1.000000 5.000000
-7.000000 4.000000
3.000000 2.000000
5.000000 2.000000

q2^T*q2 =
1.000000 0.000000
0.000000 1.000000

q2 =
-0.109109 -0.730552
0.763763 -0.491027
-0.327327 -0.323359
-0.545545 -0.347312

r2 =
-9.165151 0.763763
0.000000 -6.958209

q2*r2 =
1.000000 5.000000
-7.000000 4.000000
3.000000 2.000000
5.000000 2.000000
```

Example2
Perform QR decomposition of 2×4 complex matrix

```
#include <numeric.h>
int main() {
     array double complex a[2][4] = {1, 5, -7,  4,
                                     3, 3,  2, -13}; /* a[m][n] */
     array double complex a1[2][4] = {complex(1,2), 5, -7,  4,
                                     3, 3,  2, -13}; /* a[m][n] */
     int m = 2, n = 4;
     array double complex q[m][m];    // in the case of m<n
     array double complex r[m][n] ;

     qrdecomp(a, q, r);
     printf("q^H*q = \n%f\n", conj(transpose(q))*q);
     printf("q = \n%f\n", q);
     printf("r = \n%f\n", r);
     printf("q*r = \n%f\n", q*r);

     qrdecomp(a1, q, r);
     printf("q^H*q = \n%f\n", conj(transpose(q))*q);
     printf("q = \n%f\n", q);
     printf("r = \n%f\n", r);
     printf("q*r = \n%f\n", q*r);

     m = 4;
     n = 2;
     array double complex a2[4][2] = {complex(1,2), complex(3,5),
                                      complex(-7,2), complex(3,4),
                                      complex(3,0), complex(2,3),
                                      complex(2,0), complex(2,-13)};
                                          // a[m][n]
     array double complex q2[m][m];     // in the case of m>=n
     array double complex r2[m][n] ;
     qrdecomp(a2, q2, r2);
     printf("q2^H*q2 = \n%f\n", conj(transpose(q2))*q2);
     printf("q2 = \n%f\n", q2);
     printf("r2 = \n%f\n", r2);
     printf("q2*r2 = \n%f\n", q2*r2);
}
```

Output2

```
q^H*q =
complex(1.000000,0.000000) complex(0.000000,0.000000)
complex(0.000000,0.000000) complex(1.000000,0.000000)

q =
complex(-0.316228,0.000000) complex(-0.948683,0.000000)
complex(-0.948683,-0.000000) complex(0.316228,0.000000)

r =
complex(-3.162278,0.000000) complex(-4.427189,0.000000) complex(0.316228,0.000000)
complex(11.067972,0.000000)
complex(0.000000,0.000000) complex(-3.794733,0.000000) complex(7.273239,0.000000)
complex(-7.905694,0.000000)

q*r =
complex(1.000000,0.000000) complex(5.000000,0.000000) complex(-7.000000,0.000000)
complex(4.000000,0.000000)
```

```
complex(3.000000,0.000000) complex(3.000000,0.000000) complex(2.000000,0.000000)
complex(-13.000000,0.000000)

q^H*q =
complex(1.000000,0.000000) complex(-0.000000,0.000000)
complex(-0.000000,-0.000000) complex(1.000000,0.000000)

q =
complex(-0.267261,-0.534522) complex(0.717137,-0.358569)
complex(-0.801784,0.000000) complex(-0.000000,0.597614)

r =
complex(-3.741657,0.000000) complex(-3.741657,2.672612) complex(0.267261,-3.741657)
complex(9.354143,2.138090)
complex(0.000000,0.000000) complex(3.585686,0.000000) complex(-5.019960,-3.705209)
complex(2.868549,9.203260)

q*r =
complex(1.000000,2.000000) complex(5.000000,0.000000) complex(-7.000000,0.000000)
complex(4.000000,-0.000000)
complex(3.000000,-0.000000) complex(3.000000,0.000000) complex(2.000000,0.000000)
complex(-13.000000,-0.000000)

q2^H*q2 =
complex(1.000000,0.000000) complex(0.000000,0.000000) complex(0.000000,-0.000000)
complex(-0.000000,0.000000)
complex(0.000000,-0.000000) complex(1.000000,0.000000) complex(0.000000,-0.000000)
complex(0.000000,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000) complex(1.000000,0.000000)
complex(-0.000000,0.000000)
complex(-0.000000,-0.000000) complex(0.000000,-0.000000) complex(-0.000000,-0.000000)
complex(1.000000,0.000000)

q2 =
complex(-0.118678,-0.237356) complex(-0.097267,-0.380224) complex(-0.535487,-0.061199)
complex(0.376291,-0.586055)
complex(0.830747,-0.237356) complex(-0.175866,0.098249) complex(0.279893,0.090757)
complex(0.027154,-0.354437)
complex(-0.356034,0.000000) complex(-0.110039,-0.362539) complex(0.784519,0.011697)
complex(0.238045,-0.239634)
complex(-0.237356,0.000000) complex(-0.119864,0.804660) complex(0.041031,0.075111)
complex(0.490320,-0.184604)

r2 =
complex(-8.426150,0.000000) complex(-1.186782,6.171265)
complex(0.000000,0.000000) complex(-14.335517,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000)
complex(0.000000,0.000000) complex(0.000000,0.000000)

q2*r2 =
complex(1.000000,2.000000) complex(3.000000,5.000000)
complex(-7.000000,2.000000) complex(3.000000,4.000000)
complex(3.000000,-0.000000) complex(2.000000,3.000000)
complex(2.000000,0.000000) complex(2.000000,-13.000000)
```

See Also
orthonormalbase(), **ludecomp**(), **nullspace**() **qrdelete**(), **qrinsert**().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

qrdelete

Synopsis
#include <numeric.h>
int qrdelete(array double complex *qout*[&][&], **array double complex** *rout*[&][&],
array double complex *q*[&][&], **array double complex** *r*[&][&], **int** *j*);

Purpose
Remove a column in QR factorized matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
qout Output Q matrix in QR factorization of the matrix with a column removed.

rout Output R matrix in QR factorization of the matrix with a column removed.

q Input Q matrix in QR factorization of the original matrix.

r Input R matrix in QR factorization of the original matrix.

j Index specifying which column to be removed.

Description
This function changes Q and R to be the factorization of the matrix a with one of its columns deleted. Assume that Q and R is the original factorization of the matrix a which has n columns, this function returns $qout$ and $rout$ that are the factorization of the matrix obtained by removing the $(j+1)$th column of a.

Example1
Change QR decomposition of 6×6 complex matrix a to the decomposition of the matrix obtained by removing the third column of a.

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double complex a[6][6] = {1,  2,  3,   7.8, 8.9, 10,
                                    4,  5,  6,   1.2, 2.3, 3,
                                    7,  8,  9,   4.5, 5.6, 6,
                                    11, 15, 2.5, 5.5, 8.5, 0.1,
                                    13, 10, 7.5, 4.5, 1.5, 10,
                                    3.5,6.5,9.5, 10.5,0.2, 31};
    array double complex q[6][6], r[6][6];
    array double complex qout[6][6], rout[6][5];

    qrdecomp(a, q, r);
    qrdelete(qout, rout, q, r, 2);

    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output1

```
qout = complex(-0.0521,0.0000) complex(-0.1565,0.0000) complex(0.6493,0.0000)
complex(0.7075,0.0000) complex(-0.0466,0.0000) complex(0.2201,0.0000)

complex(-0.2084,-0.0000) complex(-0.1186,0.0000) complex(-0.1817,0.0000)
complex(0.0216,0.0000) complex(0.8093,0.0000) complex(0.5040,0.0000)

complex(-0.3648,-0.0000) complex(-0.0807,0.0000) complex(0.0181,0.0000)
complex(0.2167,0.0000) complex(0.3968,0.0000) complex(-0.8098,0.0000)

complex(-0.5732,-0.0000) complex(-0.5377,0.0000) complex(-0.4142,0.0000)
complex(0.1481,0.0000) complex(-0.4111,0.0000) complex(0.1408,0.0000)

complex(-0.6774,-0.0000) complex(0.6717,0.0000) complex(0.1903,0.0000)
complex(-0.1300,0.0000) complex(-0.1219,0.0000) complex(0.1479,0.0000)

complex(-0.1824,-0.0000) complex(-0.4633,0.0000) complex(0.5807,0.0000)
complex(-0.6428,0.0000) complex(0.0396,0.0000) complex(-0.0112,0.0000)


rout = complex(-19.1898,0.0000) complex(-20.6229,0.0000) complex(-10.4144,0.0000)
complex(-8.9110,0.0000) complex(-15.8209,0.0000)
complex(0.0000,0.0000) complex(-5.9115,0.0000) complex(-6.5255,0.0000)
complex(-5.7735,0.0000) complex(-10.1036,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(9.6041,0.0000)
complex(2.3431,0.0000) complex(25.9217,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(8.4954,0.0000) complex(-12.7715,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(4.3108,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000)

qout*rout = complex(1.0000,0.0000) complex(2.0000,0.0000) complex(7.8000,0.0000)
complex(8.9000,0.0000) complex(10.0000,0.0000)
complex(4.0000,0.0000) complex(5.0000,0.0000) complex(1.2000,0.0000)
complex(2.3000,0.0000) complex(3.0000,0.0000)
complex(7.0000,0.0000) complex(8.0000,0.0000) complex(4.5000,0.0000)
complex(5.6000,0.0000) complex(6.0000,0.0000)
complex(11.0000,0.0000) complex(15.0000,0.0000) complex(5.5000,0.0000)
complex(8.5000,0.0000) complex(0.1000,0.0000)
complex(13.0000,0.0000) complex(10.0000,0.0000) complex(4.5000,0.0000)
complex(1.5000,0.0000) complex(10.0000,0.0000)
complex(3.5000,0.0000) complex(6.5000,0.0000) complex(10.5000,0.0000)
complex(0.2000,0.0000) complex(31.0000,0.0000)
```

Example2

Change QR decomposition of 4×4 complex matrix a to the decomposition of the matrix obtained by removing the first column of a.

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double complex a[4][4] = {
       -0.7605, -0.2862, complex(-0.2748, 0.6488),  complex(-0.2748,-0.6488),
       -0.4394, 0.6886,  complex(-0.4194, -0.4555), complex(-0.4194,0.4555),
       0.1572, 0.3920,   complex(0.0360, 0.0915),   complex(0.0360,-0.0915),
       -0.4514, -0.5388, complex(-0.3322,-0.0089),  complex(-0.3322, 0.0089)
```

```
    };
    array double complex q[4][4], r[4][4];
    array double complex qout[4][4], rout[4][3];

    qrdecomp(a, q, r);
    qrdelete(qout, rout, q, r, 0);
    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output2

```
qout = complex(-0.2862,0.0000) complex(-0.3148,0.5825) complex(-0.3580,-0.5464)
complex(0.2299,0.0000)
complex(0.6886,0.0000) complex(-0.4589,-0.1572) complex(-0.4933,0.1320)
complex(0.1727,-0.0000)
complex(0.3920,0.0000) complex(0.0480,0.3054) complex(0.0434,-0.2804)
complex(-0.8187,0.0000)
complex(-0.5388,0.0000) complex(-0.3843,-0.2882) complex(-0.4086,0.2550)
complex(-0.4970,0.0000)

rout = complex(1.0000,0.0000) complex(-0.0170,-0.4587) complex(-0.0170,0.4587)

complex(0.0000,0.0000) complex(0.8884,0.0000) complex(-0.0717,-0.0252)

complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.8852,0.0000)

complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)


qout*rout = complex(-0.2862,0.0000) complex(-0.2748,0.6488) complex(-0.2748,-0.6488)

complex(0.6886,0.0000) complex(-0.4194,-0.4555) complex(-0.4194,0.4555)

complex(0.3920,0.0000) complex(0.0360,0.0915) complex(0.0360,-0.0915)

complex(-0.5388,0.0000) complex(-0.3322,-0.0089) complex(-0.3322,0.0089)
```

Example3

The same as Example 1, except that all matrices contain real numbers.

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                            4, 5, 6, 1.2, 2.3, 3,
                            7, 8, 9, 4.5, 5.6, 6,
                            11, 15, 2.5, 5.5, 8.5, .1,
                            13, 10, 7.5, 4.5, 1.5, 10,
                            3.5, 6.5, 9.5, 10.5, .2, 31};
    array double q[6][6], r[6][6];
    array double qout[6][6], rout[6][5];

    qrdecomp(a, q, r);
    qrdelete(qout, rout, q, r, 2);
```

```
    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output3

```
qout = -0.0521 -0.1565 0.6493 0.7075 -0.0466 0.2201
-0.2084 -0.1186 -0.1817 0.0216 0.8093 0.5040
-0.3648 -0.0807 0.0181 0.2167 0.3968 -0.8098
-0.5732 -0.5377 -0.4142 0.1481 -0.4111 0.1408
-0.6774 0.6717 0.1903 -0.1300 -0.1219 0.1479
-0.1824 -0.4633 0.5807 -0.6428 0.0396 -0.0112

rout = -19.1898 -20.6229 -10.4144 -8.9110 -15.8209
0.0000 -5.9115 -6.5255 -5.7735 -10.1036
0.0000 0.0000 9.6041 2.3431 25.9217
0.0000 0.0000 0.0000 8.4954 -12.7715
0.0000 0.0000 0.0000 0.0000 4.3108
0.0000 0.0000 0.0000 0.0000 0.0000

qout*rout = 1.0000 2.0000 7.8000 8.9000 10.0000
4.0000 5.0000 1.2000 2.3000 3.0000
7.0000 8.0000 4.5000 5.6000 6.0000
11.0000 15.0000 5.5000 8.5000 0.1000
13.0000 10.0000 4.5000 1.5000 10.0000
3.5000 6.5000 10.5000 0.2000 31.0000
```

See Also
qrinsert(), **qrdecomp**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

qrinsert

Synopsis
#include <**numeric.h**>
int qrinsert(**array double complex** *qout*[&][&], **array double complex** *rout*[&][&],
array double complex *q*[&][&], **array double complex** *r*[&][&],
int *j*, **array double complex** *x*[&]);

Purpose
Insert a column in QR factorized matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
qout Output Q matrix in QR factorization of the matrix with an inserted column.

rout Output R matrix in QR factorization of the matrix with an inserted column.

q Input Q matrix in QR factorization of the original matrix.

r Input R matrix in QR factorization of the original matrix.

j Index specifying where to insert the extra column.

x The column to be inserted.

Description
This function changes Q and R to be the factorization of the matrix a with an extra column inserted. Assume that Q and R is the orignial factorization of the matrix a which has n columns, this function returns $qout$ and $rout$ that are the factorization of the matrix obtained by inserting an extra column x right before the $(j+1)$th column of a. If j is n, then x is inserted after the last column of a.

Example1
Change QR decomposition of 6×6 complex matrix a to the decomposition of the matrix obtained by inserting x before the third column of a.

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double complex a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                                    4, 5, 6, 1.2, 2.3, 3,
                                    7, 8, 9, 4.5, 5.6, 6,
                                    11, 15, 2.5, 5.5, 8.5, .1,
                                    13, 10, 7.5, 4.5, 1.5, 10,
                                    3.5, 6.5, 9.5, 10.5, .2, 31};
    array double complex q[6][6], r[6][6];
    array double complex qout[6][6], rout[6][7];
    array double complex x[6] = {10, 11, 12, 13, 14, 15};

    qrdecomp(a, q, r);
    qrinsert(qout, rout, q, r, 2, x);
    printf("qout = %.4f\nrout = %.4f\n", qout, rout);
```

```
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output1

```
qout = complex(-0.0521,0.0000) complex(-0.1565,0.0000) complex(0.5705,0.0000)
complex(-0.6514,0.0000) complex(-0.1766,0.0000) complex(0.4379,0.0000)

complex(-0.2084,-0.0000) complex(-0.1186,0.0000) complex(0.3449,0.0000)
complex(-0.0204,0.0000) complex(0.8873,0.0000) complex(-0.1891,0.0000)

complex(-0.3648,-0.0000) complex(-0.0807,0.0000) complex(0.1193,0.0000)
complex(0.6106,0.0000) complex(0.0177,0.0000) complex(0.6877,0.0000)

complex(-0.5732,-0.0000) complex(-0.5377,0.0000) complex(-0.5401,0.0000)
complex(-0.3008,0.0000) complex(-0.0048,0.0000) complex(-0.0062,0.0000)

complex(-0.6774,-0.0000) complex(0.6717,0.0000) complex(0.1117,0.0000)
complex(-0.1035,0.0000) complex(-0.1586,0.0000) complex(-0.2039,0.0000)

complex(-0.1824,-0.0000) complex(-0.4633,0.0000) complex(0.4869,0.0000)
complex(0.3180,0.0000) complex(-0.3950,0.0000) complex(-0.5078,0.0000)


rout = complex(-19.1898,0.0000) complex(-20.6229,0.0000) complex(-26.8632,0.0000)
complex(-12.9365,0.0000) complex(-10.4144,0.0000) complex(-8.9110,0.0000)
complex(-15.8209,0.0000)
complex(0.0000,0.0000) complex(-5.9115,0.0000) complex(-8.3741,0.0000)
complex(-2.6154,0.0000) complex(-6.5255,0.0000) complex(-5.7735,0.0000)
complex(-10.1036,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(12.7767,0.0000)
complex(8.9680,0.0000) complex(8.0453,0.0000) complex(2.2131,0.0000)
complex(23.6127,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(4.9123,0.0000) complex(-1.1385,0.0000) complex(-5.0738,0.0000)
complex(5.8830,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(-5.1202,0.0000) complex(0.2106,0.0000)
complex(-12.8281,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(6.8539,0.0000)
complex(-9.8438,0.0000)

qout*rout = complex(1.0000,0.0000) complex(2.0000,0.0000) complex(10.0000,0.0000)
complex(3.0000,0.0000) complex(7.8000,0.0000) complex(8.9000,0.0000)
complex(10.0000,0.0000)
complex(4.0000,0.0000) complex(5.0000,0.0000) complex(11.0000,0.0000)
complex(6.0000,0.0000) complex(1.2000,0.0000) complex(2.3000,0.0000)
complex(3.0000,0.0000)
complex(7.0000,0.0000) complex(8.0000,0.0000) complex(12.0000,0.0000)
complex(9.0000,0.0000) complex(4.5000,0.0000) complex(5.6000,0.0000)
complex(6.0000,0.0000)
complex(11.0000,0.0000) complex(15.0000,0.0000) complex(13.0000,0.0000)
complex(2.5000,0.0000) complex(5.5000,0.0000) complex(8.5000,0.0000)
complex(0.1000,0.0000)
complex(13.0000,0.0000) complex(10.0000,0.0000) complex(14.0000,0.0000)
complex(7.5000,0.0000) complex(4.5000,0.0000) complex(1.5000,0.0000)
complex(10.0000,0.0000)
```

```
complex(3.5000,0.0000) complex(6.5000,0.0000) complex(15.0000,0.0000)
complex(9.5000,0.0000) complex(10.5000,0.0000) complex(0.2000,0.0000)
complex(31.0000,0.0000)
```

Example2
The same as Example1, Execpt that all matrices contain numbers.

```
#include <numeric.h>
#include <stdio.h>
int main() {
    array double a[6][6] = {1, 2, 3, 7.8, 8.9, 10,
                            4, 5, 6, 1.2, 2.3, 3,
                            7, 8, 9, 4.5, 5.6, 6,
                            11, 15, 2.5, 5.5, 8.5, .1,
                            13, 10, 7.5, 4.5, 1.5, 10,
                            3.5, 6.5, 9.5, 10.5, .2, 31};
    array double q[6][6], r[6][6];
    array double qout[6][6], rout[6][7];
    array double x[6] = {10, 11, 12, 13, 14, 15};

    qrdecomp(a, q, r);
    qrinsert(qout, rout, q, r, 2, x);
    printf("qout = %f\nrout = %f\n", qout, rout);
    printf("qout*rout = %.4f\n", qout*rout);

    return 0;
}
```

Output2

```
qout = -0.052111 -0.156528 0.570518 -0.651416 -0.176603 0.437905
-0.208444 -0.118631 0.344933 -0.020393 0.887315 -0.189091
-0.364776 -0.080733 0.119349 0.610631 0.017715 0.687745
-0.573220 -0.537685 -0.540132 -0.300849 -0.004805 -0.006178
-0.677442 0.671704 0.111661 -0.103480 -0.158569 -0.203877
-0.182388 -0.463268 0.486904 0.318045 -0.394971 -0.507826

rout = -19.189841 -20.622891 -26.863172 -12.936532 -10.414365 -8.910965 -15.820871

0.000000 -5.911545 -8.374077 -2.615442 -6.525484 -5.773472 -10.103602
0.000000 0.000000 12.776730 8.968006 8.045269 2.213064 23.612683
0.000000 0.000000 0.000000 4.912276 -1.138537 -5.073806 5.882958
0.000000 0.000000 0.000000 0.000000 -5.120200 0.210568 -12.828086
0.000000 0.000000 0.000000 0.000000 0.000000 6.853920 -9.843752

qout*rout = 1.0000 2.0000 10.0000 3.0000 7.8000 8.9000 10.0000
4.0000 5.0000 11.0000 6.0000 1.2000 2.3000 3.0000
7.0000 8.0000 12.0000 9.0000 4.5000 5.6000 6.0000
11.0000 15.0000 13.0000 2.5000 5.5000 8.5000 0.1000
13.0000 10.0000 14.0000 7.5000 4.5000 1.5000 10.0000
3.5000 6.5000 15.0000 9.5000 10.5000 0.2000 31.0000
```

See Also
qrdelete(), **qrdecomp**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

rank

Synopsis
#include <numeric.h>
int rank(array double complex *a*[&][&]);

Purpose
Find the rank of a matrix.

Return Value
This function returns the rank of a matrix.

Parameters
a Input two-dimensional array.

Description
This function provides an estimate of the number of linearly independent rows or columns of a matrix. The function can handle complex matrices.

Algorithm
The method used to compute the rank of a matrix in this function is based on the singular value decomposition by function **svd**(). The number of non-zero singular values with tolerance $tol = \max(m, n) \times \max(S) \times$ `DBL_EPSILON`.

Example1

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                            3, 7};
    /* a2 is an ill-condition matrix  */
    array double a2[2][2] = {2, 4,
                             2.001, 4.0001};
    /* singular matrix */
    array float b[3][3] = {2, 4, 4,
                           3, 7, 3,
                           3, 7, 3};

    array double complex z[2][2] = {2, complex(4, 3),
                                    3, 7};
    /* a2 is an ill-condition matrix  */
    array double complex z2[2][2] = {2,     complex(4, 3),
                                     2.001, complex(4.0001,3)};
    array complex z3[2][2] = {2, complex(4, 3),
                              3, 7};
    int r;

    r = rank(a);
    printf("rank(a) = %d\n", r);

    r = rank(a2);
    printf("rank(a2) = %d\n", r);
```

```
    r = rank(b);
    printf("rank(b) = %d\n", r);

    r = rank(z);
    printf("rank(z) = %d\n", r);
    r = rank(z2);
    printf("rank(z2) = %d\n", r);
    r = rank(z3);
    printf("rank(z3) = %d\n", r);
}
```

Output1

```
rank(a) = 2
rank(a2) = 2
rank(b) = 2
rank(z) = 2
rank(z2) = 2
rank(z3) = 2
```

Example2

```
#include <numeric.h>
int main() {
    /* singular matrix */
    array float b[3][3] = {1, 2, 3,
                           4, 5, 6,
                           7, 8, 9};
    int r;

    r = rank(b);
    printf("rank(b) = %d\n", r);
    printf("determinant(b) = %f\n", determinant(b));
}
```

Output2

```
rank(b) = 2
determinant(b) = 0.000000
```

See Also
svd().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

rcondnum

Synopsis
#include <**numeric.h**>
double rcondnum(**array double complex** *a*[&][&]);

Purpose
Estimate reciprocal of the condition number of a matrix.

Return Value
This function returns the reciprocal of the condition number of a matrix.

Parameters
a Input matrix.

Description
This function calculates an estimate for the reciprocal of the condition of matrix *a* in 1-norm. If *a* is well conditioned , **rcondnum**(*a*) is near 1.0. If *a* is badly conditioned, **rcondnum**(*a*) is near 0.0. Compared to **condnum**(), function **rcondnum**() is a more efficient, but less reliable, method of estimating the condition of matrix.

Algorithm
The reciprocal of the condition number of a general real/complex matrix A is estimated in either the 1-norm or the infinity-norm, using the LU decomposition.

Example

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, 4,
                            3, 7};
    /* a2 is an ill-condition matrix  */
    array double a2[3][3] = {2, 4, 5,
                             2.001, 4.0001, 5.001,
                             6, 8, 3};
    array double complex z[2][2] = {2, 4,
                                    3, 7};
    array double complex z2[2][2] = {complex(2, 3), complex(4,2),
                                     3, 7};
    double rcond;

    rcond = rcondnum(a);
    printf("rcondnum(a) = %f\n", rcond);
    rcond = rcondnum(a2);
    printf("rcondnum(a2) = %f\n", rcond);

    rcond = rcondnum(z);
    printf("rcondnum(z) = %f\n", rcond);
    rcond = rcondnum(z2);
    printf("rcondnum(z2) = %f\n", rcond);
}
```

Output

```
rcondnum(a) = 0.018182
rcondnum(a2) = 0.000035
rcondnum(z) = 0.018182
rcondnum(z2) = 0.131909
```

See Also
condnum(), **norm**(), **rank**(), **svd**().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

residue

Synopsis
#include <numeric.h>
int residue(**array double** *u*[&], **array double** *v*[&], **array double complex** *r*[&],
array double complex *p*[&], **array double** *k*[&]);

Syntax
residue(*u*, *v*, *r*, *p*, *k*)

Purpose
Partial-fraction expansion or residue computation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
u A vector of size m which contains the numerator of the ratio of two polynomials.

v A vector of size n which contains the denominator of the ratio of two polynomials.

r A vector of size $(n-1)$ which contains the residues of a partial fraction expansion of the ratio of two polynomials.

p A vector of size $(n-1)$ which contains the poles of a partial fraction expansion of the ratio of two polynomials.

k A vector of size $(m-n+1)$ which contains the direct term of a partial fraction expansion of the ratio of two polynomials. If $m \leq n$, then *k* contains NULL.

Description
Function **residue**() finds the residues, poles and direct term of a partial fraction expansion of the ratio of two polynomials $V(s)$ and $U(s)$. Given the ratio of two polynomials

$$\frac{U(s)}{V(s)} = \frac{u_0 s^m + u_1 s^{m-1} + \cdots + u_{m-1} s + u_m}{s^n + v_1 s^{n-1} + \cdots + v_{n-1} s + v_n}$$

If there are no multiple roots, the expansion becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_{n-1}}{s - p_{n-1}} + K(s)$$

If p_i is a pole of multiplicity l, then the expansion becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_i}{(s - p_i)^l} + \cdots + \frac{r_{i+1}}{(s - p_i)^2} + \frac{r_{i+l}}{s - p_i} + \cdots + \frac{r_{n-1}}{s - p_{n-1}} + K(s)$$

where $K(x)$ is the direct term. If $m > n$, $K(s)$ is

$$K(s) = k_0 s^{m-n} + k_1 s^{m-n-1} + \cdots + k_{m-n} s + k_{m-n+1}$$

Otherwise, $K(s)$ is empty. Vectors *u* and *v* specify the coefficients of the polynomials in descending powers of s. They can be any real type. Conversion of the data to double type is performed internally. The residues *r* and poles *p* are any compatible data type according to the residue computation. If the argument of real type is passed and the result is complex type, the value of NaN will be passed out. The direct term *k* is always real type.

Algorithm

Given the ratio of two polynomials $U(x)$ and $V(x)$

$$\frac{U(s)}{V(s)} = \frac{u_0 s^m + u_1 s^{m-1} + \cdots + u_{m-1} s + u_m}{s^n + v_1 s^{n-1} + \cdots + v_{n-1} s + v_n}$$

If there are no multiple roots, it can be written as

$$\frac{U(s)}{V(s)} = \frac{U(s)}{(s-p_0)(s-p_1)\cdots(s-p_{n-1})}$$

where p_i can be real or complex number. Then it becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s-p_0} + \frac{r_1}{s-p_1} + \cdots + \frac{r_{n-1}}{s-p_{n-1}}$$

where

$$r_i = [(s-p_i)\frac{U(s)}{V(s)}]_{s=p_i}$$

If l of the (n - 1) poles are identical, or say, the root at $s = p_i$ is of multiplicity l, then

$$\frac{U(s)}{V(s)} = \frac{U(s)}{(s-p_0)(s-p_1)\cdots(s-p_i)^l\cdots(s-p_i)^2(s-p_i)\cdots(s-p_{n-l})}$$

The equation can be expanded as

$$\frac{U(s)}{V(s)} = \frac{r_0}{s-p_0} + \frac{r_1}{s-p_1} + \cdots + \frac{r_i}{(s-p_i)^l} + \cdots + \frac{r_{i+1}}{(s-p_i)^2} + \frac{r_{i+l}}{s-p_i} + \cdots + \frac{r_{n-1}}{s-p_{n-l}}$$

where the coefficients correspond to simple poles could be calculated by the method described above. The determination of the coefficients that correspond to the multiple-order is given below.

$$r_{i+k} = \frac{1}{(k-1)!}\frac{d^{k-1}}{ds^{k-1}}[(s-p_i)^l\frac{U(s)}{V(s)}]_{s=p_i} \qquad k = 1, 2, \cdots, l$$

Example 1

Partial-fraction expansion with single real roots and without direct term.

$$\frac{10s+6}{2s^3+12s^2+22s+12} = \frac{-1}{s+1} + \frac{7}{s+2} + \frac{-6}{s+3}$$

```
#include <stdio.h>
#include <numeric.h>

#define M 2        /* data array size */
#define N 4        /* response array size */
```

```
int main() {
   array double u[M] = {10, 6};
   array double v[N] = {2, 12, 22, 12};
   array double r[N-1],p[N-1];
   array double k[1];

   residue(u,v,r,p,k);
   printf("coef. of u=%f\n",u);
   printf("coef. of v=%f\n",v);
   printf("coef. of r=%f\n",r);
   printf("coef. of p=%f\n",p);
   printf("coef. of k=%f\n",k);
}
```

Output

```
coef. of u=10.000000 6.000000

coef. of v=2.000000 12.000000 22.000000 12.000000

coef. of r=-1.000000 7.000000 -6.000000

coef. of p=-1.000000 -2.000000 -3.000000

coef. of k=0.000000
```

Example 2

A partial-fraction expansion with single roots of complex numbers and without direct term.

$$\frac{x+3}{x^2+2x+5} = \frac{0.5+i0.5}{s+(1+i2)} + \frac{0.5-i0.5}{s+(1-i2)}$$

```
#include <stdio.h>
#include <numeric.h>

#define M 2        /* data array size */
#define N 3        /* response array size */

int main() {
   array double u[M] = {1, 3};
   array double v[N] = {1, 2, 5};
   array double complex r[N-1],p[N-1];
   array double r1[N-1],p1[N-1];
   array double k[1];

   residue(u,v,r,p,k);
   printf("coef. of u=%f\n",u);
   printf("coef. of v=%f\n",v);
   printf("coef. of r=%f\n",r);
   printf("coef. of p=%f\n",p);
   printf("coef. of k=%f\n",k);

   printf("\n r1, p1 pass by real data, but the result are complex data.\n");
   residue(u,v,r1,p1,k);
   printf("coef. of r=%f\n",r1);
   printf("coef. of p=%f\n",p1);
}
```

Output

```
coef. of u=1.000000 3.000000

coef. of v=1.000000 2.000000 5.000000

coef. of r=complex(0.500000,0.500000) complex(0.500000,-0.500000)

coef. of p=complex(-1.000000,-2.000000) complex(-1.000000,2.000000)

coef. of k=0.000000


 r1, p1 pass by real data, but the result are complex data.
coef. of r=NaN NaN

coef. of p=NaN NaN
```

Example 3

A partial-fraction expansion with single roots and has a direct term.

$$\frac{2s^3 + 12s^2 + 22s + 12}{10s + 6} = \frac{0.2688}{s + 0.6} + 0.2s^2 + 1.08s + 1.552$$

```
#include <stdio.h>
#include <numeric.h>

#define N 2        /* data array size */
#define M 4        /* response array size */

int main() {
   array double v[N] = {10, 6};
   array double u[M] = {2, 12, 22, 12};
   array double r[N-1],p[N-1];
   array double k[M-N+1];

   residue(u,v,r,p,k);
   printf("coef. of u=%f\n",u);
   printf("coef. of v=%f\n",v);
   printf("coef. of r=%f\n",r);
   printf("coef. of p=%f\n",p);
   printf("coef. of k=%f\n",k);
}
```

Output

```
coef. of u=2.000000 12.000000 22.000000 12.000000

coef. of v=10.000000 6.000000

coef. of r=0.268800

coef. of p=-0.600000

coef. of k=0.200000 1.080000 1.552000
```

Example 4
A partial-fraction expansion with multiplicity roots and without direct term.

$$\frac{s^2+2s+3}{s^5+5s^4+9s^3+7s^2+2s} = \frac{3}{2(s+2)} - \frac{3}{(s+1)^3} - \frac{2}{s+1} + \frac{3}{2s}$$

```
#include <stdio.h>
#include <numeric.h>

#define M 3        /* data array size */
#define N 6        /* response array size */

int main() {
   array double u[M] = {1,2,3};
   array double v[N] = {1, 5, 9, 7, 2, 0};
   array double r[N-1],p[N-1];
   array double k[1];

   residue(u,v,r,p,k);
   printf("coef. of u=%f\n",u);
   printf("coef. of v=%f\n",v);
   printf("coef. of r=%f\n",r);
   printf("coef. of p=%f\n",p);
   printf("coef. of k=%f\n",k);
}
```

Output

```
coef. of u=1.000000 2.000000 3.000000

coef. of v=1.000000 5.000000 9.000000 7.000000 2.000000 0.000000

coef. of r=1.500000 -3.000000 0.000000 -2.000000 1.500000

coef. of p=-2.000000 -1.000000 -1.000000 -1.000000 0.000000

coef. of k=0.000000
```

References

Benjamin C. Kuo, *Automatic Control System*, fifth edition, Prentice-Hall, 1987, p. 28-32.

roots

Synopsis
#include <numeric.h>
int roots(**array double complex** *x*[&], **array double complex** *p*[&]);

Purpose
Find the roots of a polynomial.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x An output which contains the roots of the polynomial.

p An input which contains the coefficients of the polynomial.

Description
This function finds the roots of the polynomial. The function can handle the polynomial with complex coefficients.

Algorithm
The roots of a polynomial are obtained by computing the eigenvalues of the corresponding companion matrix of the polynomial.

Example

```
#include <numeric.h>
int main() {
     array double x1[3], p1[4]  = {1, -6, -72, -27}; /* x^3-6x^2-72x-27 =0*/
     array double x2[4], p2[5]  = {1, -12, 0, 25, 116}; /* x^4-12x^3-25x+116 =0 */
     array double complex z2[4];
     array double complex z3[4], p3[5] = {complex(3,4), complex(4,2),
                          complex(5,3), complex(2,4), complex(1,5)};
     int status, i;


     roots(x1, p1);
     printf("x1 from roots(x1, p1) = %.3f\n", x1);
     for(i=0; i<3; i++)
       printf("polyeval(p1, x1[%d]) = %.3f\n", i,  polyeval(p1, x1[i]));


     /* x^4-12x^3-25x+116 =0  has two complex roots and two real roots */
     roots(x2, p2);
     printf("x2 from roots(x2, p2) = %.3f\n", x2);
     roots(z2, p2);
     printf("z2 from roots(z2, p2) = \n%.3f\n", z2);

     status = roots(z3, p3);
     if(status == 0)
       printf("z3 from roots(z3, p3) = \n%.3f\n", z3);
     else
```

```
       printf("roots(z3, p3) failed\n");

     for(i=0; i<4; i++)
       printf("cpolyeval(p3, z3[%d]) = %.3f\n", i,  cpolyeval(p3, z3[i]));
}
```

Output

```
x1 from roots(x1, p1) = 12.123 -5.735 -0.388

polyeval(p1, x1[0]) = 0.000
polyeval(p1, x1[1]) = -0.000
polyeval(p1, x1[2]) = 0.000
x2 from roots(x2, p2) = 11.747 2.703 NaN NaN

z2 from roots(z2, p2) =
complex(11.747,0.000) complex(2.703,0.000) complex(-1.225,1.467) complex(-1.225,-1.467)


z3 from roots(z3, p3) =
complex(0.226,1.281) complex(0.431,-0.728) complex(-0.754,-0.708) complex(-0.703,0.554)


cpolyeval(p3, z3[0]) = complex(0.000,-0.000)
cpolyeval(p3, z3[1]) = complex(0.000,-0.000)
cpolyeval(p3, z3[2]) = complex(0.000,-0.000)
cpolyeval(p3, z3[3]) = complex(0.000,-0.000)
```

See Also

fzero(), **residue**().

References

rot90()

Synopsis
#include <numeric.h>
int rot90(**array double complex** *y[&][&]*, **array double complex** *x[&][&]*, ... /* [**int** *k*] */);

Syntax
rot90(*y*, *x*)

Purpose
Rotate matrix 90 degrees.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Two-dimensional matrix which contains the original data.

y Two-dimensional matrix of the same data type and size correspondence with the size of rotation of matrix *x*. It contains the rotation of input matrix *x*.

k Integral data which indicates the $k*90$ degrees rotation of *x*. For a positive *k*, the matrix is rotated counter clockwise. For a negative *k*, the matrix is rotated clockwise.

Description
This function rotates matrix *x* $k*90$ degrees in counter clockwise direction.

Example

```
#include <numeric.h>
int main() {
   array double x[4][2]={1,3,
                         2.45,8.56,
                         3,5,
                         6,8};
   array double complex zx[3][2]={complex(1,1),0,
                         3,complex(4,1),
                         0,0};
   array double complex zy[3][2];
   array double y[2][4], y1[4][2];

   rot90(y,x);
   printf("x = \n%f",x);
   printf("y = rotate 90 degrees\n%f",y);
   rot90(y1,x,2);
   printf("y1 = rotate 180 degrees\n%f",y1);
   rot90(y1,x,8);
   printf("y1 = rotate 720 degrees\n%f",y1);
   rot90(y,x,-1);
   printf("y = totate -90/270 degrees\n%f",y);
   printf("\n");
```

```
   rot90(zy,zx,2);         /* complex data rotation */
   printf("zx = \n%5.1f",zx);
   printf("zy = \n%5.1f",zy);

}
```

Output

```
x =
1.000000 3.000000
2.450000 8.560000
3.000000 5.000000
6.000000 8.000000
y = rotate 90 degrees
3.000000 8.560000 5.000000 8.000000
1.000000 2.450000 3.000000 6.000000
y1 = rotate 180 degrees
8.000000 6.000000
5.000000 3.000000
8.560000 2.450000
3.000000 1.000000
y1 = rotate 720 degrees
1.000000 3.000000
2.450000 8.560000
3.000000 5.000000
6.000000 8.000000
y = totate -90/270 degrees
6.000000 3.000000 2.450000 1.000000
8.000000 5.000000 8.560000 3.000000

zx =
complex(  1.0,  1.0) complex(  0.0,  0.0)
complex(  3.0,  0.0) complex(  4.0,  1.0)
complex(  0.0,  0.0) complex(  0.0,  0.0)
zy =
complex(  0.0,  0.0) complex(  0.0,  0.0)
complex(  4.0,  1.0) complex(  3.0,  0.0)
complex(  0.0,  0.0) complex(  1.0,  1.0)
```

See Also
flipud(), **fliplr**().

rsf2csf

Synopsis
#include <numeric.h>
int rsf2csf(**array double complex** *uc*[:][:], **array double complex** *tc*[:][:],
array double complex *ur*[&][&], **array double complex** *tr*[&][&]);

Purpose
Convert a real upper quasi-triangular Schur form to a complex upper triangular Schur form.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
uc Output complex unitary matrix.

tc Output complex quasi-triangular Schur matrix.

ur Input real unitary matrix.

tr Input real upper triangular Schur matrix.

Description
This function converts a real upper quasi-triangular Schur form, which is the result of Schur decomposition, to a complex upper triangular Schur form. Assume that (u_r, t_r) is the real Schur form of the real square matrix x. This function can produce the complex Schur form (u_c, t_c) from the real one (u_r, t_r), such that $x = u_c t_c u_c'$. The matrix t_c is upper triangular with the eigenvalues of x on the diagonal.

Example
Transform the output of function **schurdecomp**() from real Schur form (ur, tr) to complex Schur form (uc, tc).

```
#include <array.h>

int main() {
    array double x1[4][4] = {1,1,1,3, 1,2,1,1, 1,1,3,1, -2,1,1,4};
    array double ur1[4][4], tr1[4][4];
    array double complex uc1[4][4], tc1[4][4];

    array double x2[3][3] = {2, 3, 4, 5, 6, 7, 8, 9, 10};
    array double ur2[3][3], tr2[3][3];
    array double complex uc2[3][3], tc2[3][3];

    schurdecomp(x1, ur1, tr1);
    printf("ur1 = %.4f\ntr1 = %.4f\n", ur1, tr1);

    rsf2csf(uc1, tc1, ur1, tr1);
    printf("uc1= %.4f\ntc1 = %.4f\n", uc1, tc1);
    printf("norm(uc1) = %.4f\n\n", norm(uc1, "2"));
    printf("uc1*tc1*uc1' = %.4f\n\n", uc1*tc1*transpose(conj(uc1)));

    schurdecomp(x2, ur2, tr2);
    printf("ur2 = %.4f\ntr2 = %.4f\n", ur2, tr2);
```

```
    rsf2csf(uc2, tc2, ur2, tr2);
    printf("uc2 = %.4f\ntc2 = %.4f\n", uc2, tc2);
    printf("norm(uc2) = %.4f\n\n", norm(uc2, "2"));
    printf("uc2*tc2*uc2' = %.4f\n\n", uc2*tc2*transpose(conj(uc2)));

    return 0;
}
```

Output

```
ur1 = 0.8835 0.3186 -0.0197 -0.3428
0.4458 -0.3651 0.1666 0.8001
0.0480 -0.5469 0.7191 -0.4260
-0.1355 0.6827 0.6743 0.2466

tr1 = 1.9202 0.9542 2.9655 0.8572
-2.2777 1.9202 1.1416 1.5630
0.0000 0.0000 4.8121 1.1314
0.0000 0.0000 0.0000 1.3474

uc1= complex(-0.2675,0.4801) complex(0.7417,-0.1731) complex(-0.0197,0.0000)
complex(-0.3428,0.0000)
complex(0.3065,0.2422) complex(0.3743,0.1984) complex(0.1666,0.0000)
complex(0.8001,0.0000)
complex(0.4591,0.0261) complex(0.0403,0.2971) complex(0.7191,0.0000)
complex(-0.4260,0.0000)
complex(-0.5732,-0.0736) complex(-0.1138,-0.3710) complex(0.6743,0.0000)
complex(0.2466,0.0000)

tc1 = complex(1.9202,1.4742) complex(1.3236,0.0000) complex(-0.9583,-1.6113)
complex(-1.3122,-0.4658)
complex(0.0000,0.0000) complex(1.9202,-1.4742) complex(2.4895,0.6203)
complex(0.7196,0.8493)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(4.8121,0.0000)
complex(1.1314,0.0000)
complex(0.0000,0.0000) complex(0.0000,0.0000) complex(0.0000,0.0000)
complex(1.3474,0.0000)

norm(uc1) = 1.0000

uc1*tc1*uc1' = complex(1.0000,-0.0000) complex(1.0000,-0.0000) complex(1.0000,-0.0000)
complex(3.0000,0.0000)
complex(1.0000,0.0000) complex(2.0000,-0.0000) complex(1.0000,0.0000)
complex(1.0000,0.0000)
complex(1.0000,0.0000) complex(1.0000,-0.0000) complex(3.0000,-0.0000)
complex(1.0000,-0.0000)
complex(-2.0000,-0.0000) complex(1.0000,0.0000) complex(1.0000,-0.0000)
complex(4.0000,0.0000)


ur2 = 0.2826 0.8680 0.4082
0.5383 0.2087 -0.8165
0.7939 -0.4505 0.4082

tr2 = 18.9499 4.8990 0.0000
0.0000 -0.9499 -0.0000
0.0000 0.0000 -0.0000

uc2 = complex(0.2826,0.0000) complex(0.8680,0.0000) complex(0.4082,0.0000)
```

```
complex(0.5383,0.0000) complex(0.2087,0.0000) complex(-0.8165,0.0000)

complex(0.7939,0.0000) complex(-0.4505,0.0000) complex(0.4082,0.0000)


tc2 = complex(18.9499,0.0000) complex(4.8990,0.0000) complex(0.0000,0.0000)

complex(0.0000,0.0000) complex(-0.9499,0.0000) complex(-0.0000,0.0000)

complex(0.0000,0.0000) complex(0.0000,0.0000) complex(-0.0000,0.0000)


norm(uc2) = 1.0000

uc2*tc2*uc2' = complex(2.0000,0.0000) complex(3.0000,0.0000) complex(4.0000,0.0000)

complex(5.0000,0.0000) complex(6.0000,0.0000) complex(7.0000,0.0000)

complex(8.0000,0.0000) complex(9.0000,0.0000) complex(10.0000,0.0000)
```

See Also
schurdecomp().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

schurdecomp

Synopsis
#include <**numeric.h**>
int schurdecomp(**array double complex** *a*[&][&], **array double complex** *q*[&][&],
array double complex *t*[&][&]);

Purpose
Compute Schur decomposition of a matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a An input two-dimensional array.

q An output two-dimensional array which contains *q* matrix.

t An output two-dimensional array which contains *t* matrix.

Description
This function computes Schur form of a square matrix. The Schur matrix **T** for a square matrix **A** of real type is defined as

$$A = QTQ^T$$

where matrix **Q** is an orthogonal matrix with Q^TQ=**I**. For matrix **A** of complex type, the Schur matrix *T* is defined as

$$A = QTQ^H$$

where matrix **Q** is unitary with Q^HQ=**I**.

Example1
Compute Schur decomposition of a real matrix.

```
#include <numeric.h>
int main() {
     int n = 2;
     array double a[2][2] = {8, -3,
                             -5, 9};
     array double t[n][n];
     array double q[n][n];
     int status;

     status = schurdecomp(a, q, t);
     if(status == 0) {
         printf("Schur vectors q =\n%f\n", q);
         printf("Schur form t =\n%f\n", t);
         printf("q*t*qˆT =\n%f\n", q*t*transpose(q));
         printf("qˆT*q =\n%f\n", transpose(q)*q);
     }
     else
         printf("error: numerical error in schurdecomp()\n");
     schurdecomp(a, q, NULL);
     printf("Schur vectors q =\n%f\n", q);
}
```

Output1

```
Schur vectors q =
0.661061 -0.750332
0.750332 0.661061

Schur form t =
4.594875 2.000000
0.000000 12.405125

q*t*qˆT =
8.000000 -3.000000
-5.000000 9.000000

qˆT*q =
1.000000 0.000000
0.000000 1.000000

Schur vectors q =
0.661061 -0.750332
0.750332 0.661061
```

Example2
Compute Schur decomposition of complex matrix.

```
#include <numeric.h>
int main() {
     int n = 2;
     array double complex a[2][2] = {8, -3,
                                     -5, 9};
     array double complex z[2][2] = {complex(1,2), -3,
                                     -5, 9};
     array double complex t[n][n];
     array double complex q[n][n];

     schurdecomp(a, q, t);
     printf("Schur vectors q =\n%f\n", q);
     printf("Schur form t =\n%f\n", t);
     printf("q*t*qˆH =\n%f\n", q*t*conj(transpose(q)));
     printf("qˆH*q =\n%f\n", conj(transpose(q))*q);


     schurdecomp(z, q, t);
     printf("Schur vectors q =\n%f\n", q);
     printf("Schur form t =\n%f\n", t);
     printf("q*t*qˆH =\n%f\n", q*t*conj(transpose(q)));
     printf("qˆH*q =\n%f\n", conj(transpose(q))*q);
}
```

Output2

```
Schur vectors q =
complex(-0.661061,0.000000) complex(-0.750332,0.000000)
complex(-0.750332,0.000000) complex(0.661061,0.000000)

Schur form t =
complex(4.594875,0.000000) complex(-2.000000,0.000000)
complex(0.000000,0.000000) complex(12.405125,0.000000)
```

```
q*t*q^H =
complex(8.000000,0.000000) complex(-3.000000,0.000000)
complex(-5.000000,0.000000) complex(9.000000,0.000000)

q^H*q =
complex(1.000000,0.000000) complex(-0.000000,0.000000)
complex(-0.000000,0.000000) complex(1.000000,0.000000)

Schur vectors q =
complex(-0.874263,0.158241) complex(-0.452441,0.076946)
complex(-0.458937,0.000000) complex(0.888419,0.009421)

Schur form t =
complex(-0.524869,1.723999) complex(-1.983020,1.403899)
complex(0.000000,0.000000) complex(10.524869,0.276001)

q*t*q^H =
complex(1.000000,2.000000) complex(-3.000000,0.000000)
complex(-5.000000,0.000000) complex(9.000000,0.000000)

q^H*q =
complex(1.000000,0.000000) complex(0.000000,-0.000000)
complex(0.000000,0.000000) complex(1.000000,0.000000)
```

See Also
eigensystem().

References
E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

sign

Synopsis
#include <**numeric.h**>
double sign(**double** *x*)

Purpose
Sign function.

Return Value
This function returns 0, 1, -1 to indicate the sign of real number *x*;

Parameters
x The input **double** type real number.

Description
The function returns the 1, -1 and 0 for $x > 0$, $x < 0$ and $x = 0$, respectively.

Example

```
#include <numeric.h>
int main() {
    int i = -10;
    float f = -10;
    double d = 10;
    double d2 = 0;
    printf("sign(i) = %d\n", sign(i));
    printf("sign(f) = %d\n", sign(f));
    printf("sign(d) = %d\n", sign(d));
    printf("sign(d2) = %d\n", sign(d2));
}
```

Output

```
sign(i) = -1
sign(f) = -1
sign(d) = 1
sign(d2) = 0
```

See Also
abs(), **real**(), **conj**().

References

sort()

Synopsis
#include <**numeric.h**>
int sort(**array double complex** &*y*, **array double complex** &*x*, ...
/* [**string_t** *method*], [**array int** &*ind*] */);

Syntax
sort(*y*, *x*)
sort(*y*, *x*, *method*)
sort(*y*, *x*, *method*, *ind*)

Purpose
Sorting and ranking elements in ascending order.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
y An array of any dimension and size which contains the sorted data.

x The same array as *x* which contains the original data.

method Sorting method.
method = "array" - (default) sorted by total elements. That is, if *x* is an array of $n \times m$, sort in total $n \times m$ elements.
method = "row" - sorted by row for two-dimensional array.
method = "column" - sorted by column for two-dimensional array.

ind Index array . Each elements presents the ranking index, starting with 0, corresponding to array *x*.

Description
The original data in array *x* can be of any supported arithmetic data type and dimension. Array *y* is the same data type and size as *x* and it contains sorted data. If *x* is a complex data type, it is sorted by the magnitude of each element. If *x* includes any NaN or ComplexNaN elements, **sort**() places these at the end. The index in array *ind* contains the ranking index corresponding to array *x* and optional argument *method* specifies the sorting method. When *x* is a two-dimensional array, this parameter specifies the sorting method defined as follows: "array" - sorted by total elements; "row" - sorted by row for two-dimensional array; "column" - sorted by column for two-dimensional array. By default, two-dimensional arrays are sorted by total elements. If *x* is not a two-dimensional array, The array is sorted by total elements.

Example

```
#include <numeric.h>

int main() {
    array double x1[4] = {0.1, NaN, -0.1, 3}, y1[4];
```

```
    array double complex x2[2][3] = {5, NaN, -3, -6, 3, 5}, y2[2][3];
    array int inde1[2][3];
    array double x4[3][3] = {1, 3, -3, -2, 3, 5, 9, 2, 4}, y4[3][3];
    array int inde[3][3];
    array double x3[2][3][4],y3[2][3][4];
    x3[1][1][2] = 10;
    x3[1][2][2] = -10;

    sort(y1, x1);           // sort by total elements
    printf("sort by total elements\n");
    printf("x1 = %5.2f\n", x1);
    printf("sort(x1) = %5.2f\n", y1);

    sort(y2, x2, "row", inde1);
    printf("sort by row, gives index array\n");
    printf("x2 = %f\n", x2);
    printf("sort(x2)\n");
    printf("%f\n", y2);
    printf("index(x2)\n");
    printf("%d\n", inde1);

    sort(y3, x3, "array");
    printf("sort by total elements \n");
    printf("x3 = %f\n", x3);
    printf("sort(x3)\n");
    printf("%f\n", y3);

    sort(y4, x4, "column", inde);
    printf("sort by column elements, gives index array\n");
    printf("x4 = %f\n", x4);
    printf("sort(x4)\n");
    printf("%f\n", y4);
    printf("inde(x4)\n");
    printf("%d\n", inde);
}
```

Output

```
sort by total elements
x1 =  0.10   NaN -0.10  3.00

sort(x1) = -0.10  0.10  3.00   NaN

sort by row, gives index array
x2 = complex(5.000000,0.000000) ComplexNaN complex(-3.000000,0.000000)
complex(-6.000000,0.000000) complex(3.000000,0.000000) complex(5.000000,0.000000)


sort(x2)
complex(-3.000000,0.000000) complex(5.000000,0.000000) ComplexNaN
complex(3.000000,0.000000) complex(5.000000,0.000000) complex(-6.000000,0.000000)


index(x2)
2 0 1
1 2 0

sort by total elements
x3 = 0.000000 0.000000 0.000000 0.000000
```

```
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000

0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 10.000000 0.000000
0.000000 0.000000 -10.000000 0.000000

sort(x3)
-10.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000

0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 0.000000
0.000000 0.000000 0.000000 10.000000

sort by column elements, gives index array
x4 = 1.000000 3.000000 -3.000000
-2.000000 3.000000 5.000000
9.000000 2.000000 4.000000

sort(x4)
-2.000000 2.000000 -3.000000
1.000000 3.000000 4.000000
9.000000 3.000000 5.000000

inde(x4)
1 2 0
0 0 2
2 1 1
```

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

specialmatrix()

Synopsis
#include <**numeric.h**>
array double specialmatrix(**string_t** *name*, ... /* [**type1** *arg1*, **type2** *arg2*, ...] */)[:][:];

Syntax
specialmatrix(*name*)
specialmatrix(*name*, *arg1*)
specialmatrix(*name*, *arg1*, *arg2*, *arg3*)

Purpose
Generate a special matrix.

Return Value
This function returns a special matrix.

Parameters
name The special matrix name selected from the table below.

arg1, arg2, **...** Input arguments required by a particular special matrix. The number of arguments and their data types are different for different special matrices.

Special Matrices

Cauchy	ChebyshevVandermonde	Chow	Circul	Celement
Dramadah	DenavitHartenberg	DenavitHartenberg2	Fiedler	Frank
Gear	Hadamard	Hankel	Hilbert	InverseHilbert
Magic	Pascal	Rosser	Toeplitz	Vandermonde
Wilkinson				

. . . . . .

Cauchy

Synopsis
#include <**numeric.h**>
array double specialmatrix("Cauchy", **array double** *c[&]*, ... /* [**array double** *r[&]*] */);

Syntax
specialmatrix("Cauchy", *c*)
specialmatrix("Cauchy", *c*, *r*)

Purpose
Generate a Cauchy matrix.

Return Value
This function returns an $n \times n$ Cauchy matrix.

Parameters

c Input vector with n elements.

c An optional input vector with n elements.

Description

This function generates the $n \times n$ Cauchy matrix whose elements are $y[i][j] = 1/(c[i] + r[j])$. When optional input *r* is not specified, then inside the function, *c* is assigned to *r*. That is, element (i, j) is defined by $y[i][j] = 1/(c[i] + c[j])$.

Example

```
#include <numeric.h>
int main() {
    array double r[5] ={3,4,8,6,2};
    array double c[5] ={8,2.3,9,6,2};
    printf("c=\n%f",c);
    printf("r=\n%f",r);
    printf("specialmatrix(\"Cauchy\", c) =\n%5.4f\n", specialmatrix("Cauchy", c));
    printf("specialmatrix(\"Cauchy\", c, r) =\n%5.4f\n", specialmatrix("Cauchy", c, r));

}
```

Output

```
c=
8.000000 2.300000 9.000000 6.000000 2.000000
r=
3.000000 4.000000 8.000000 6.000000 2.000000
specialmatrix("Cauchy", c) =
0.0625 0.0971 0.0588 0.0714 0.1000
0.0971 0.2174 0.0885 0.1205 0.2326
0.0588 0.0885 0.0556 0.0667 0.0909
0.0714 0.1205 0.0667 0.0833 0.1250
0.1000 0.2326 0.0909 0.1250 0.2500

specialmatrix("Cauchy", c, r) =
0.0909 0.0833 0.0625 0.0714 0.1000
0.1887 0.1587 0.0971 0.1205 0.2326
0.0833 0.0769 0.0588 0.0667 0.0909
0.1111 0.1000 0.0714 0.0833 0.1250
0.2000 0.1667 0.1000 0.1250 0.2500
```

References

Knuth, Donald, *The art of computer programming*, Vol. 1, Addison-Wesley Pub. Co., 1973.

ChebyshevVandermonde

Synopsis

#include <numeric.h>

array double specialmatrix("ChebyshevVandermonde", **array double** *c[&]*, ... /* [**int** *m*] */);

Syntax
specialmatrix("ChebyshevVandermonde", *c*)
specialmatrix("ChebyshevVandermonde", *c*, *m*)

Purpose
Generate a Vandermonde-like matrix for the Chebyshev polynomials.

Return Value
This function returns a Vandermonde-like matrix.

Parameters
c Input vector with n elements.

m An optional input integral number which defines the rows of the output Vandermonde-like matrix.

Description
This function generates the $n \times m$ Chebyshev Vandermonde matrix whose elements are $y[i][j] = T_i(P[j])$. T_{i-1} is the Chebyshev polynomial of ith degree. When the optional input *m* is not specified, the output is a square matrix of $n \times n$. If the user specifies the optional input *m*, then a rectangular version of $n \times m$ Chebyshev Vandermonde matrix is generated.

Example

```
#include <numeric.h>
int main() {
    array double v[5] = {2,3,4,5,6};
    printf("v=\n%f",v);
    printf("specialmatrix(\"ChebyshevVandermonde\", v) =\n%5.4f\n",
            specialmatrix("ChebyshevVandermonde", v));
    printf("specialmatrix(\"ChebyshevVandermonde\", v, 3) =\n%5.4f\n",
            specialmatrix("ChebyshevVandermonde", v, 3));
}
```

Output

```
v=
2.000000 3.000000 4.000000 5.000000 6.000000
specialmatrix("ChebyshevVandermonde", v) =
1.0000 1.0000 1.0000 1.0000 1.0000
2.0000 3.0000 4.0000 5.0000 6.0000
7.0000 17.0000 31.0000 49.0000 71.0000
26.0000 99.0000 244.0000 485.0000 846.0000
97.0000 577.0000 1921.0000 4801.0000 10081.0000

specialmatrix("ChebyshevVandermonde", v, 3) =
1.0000 1.0000 1.0000 1.0000 1.0000
2.0000 3.0000 4.0000 5.0000 6.0000
7.0000 17.0000 31.0000 49.0000 71.0000
```

References
N.J. Higham, *Stability analysis of algorithms for solving confluent Vandermonde-like systems*, SIAM J.

Matrix Anal. Appl., 11 (1990).
Knuth, Donald Ervin, *The art of computer programming*, Vol. 1, Addison-Wesley Pub. Co., 1973.

Chow

Synopsis
#include <numeric.h>
array double specialmatrix("Chow", **int** *n*, ... /* [**double** *u*, **double** *v*] */);

Syntax
specialmatrix("Chow", *n*)
specialmatrix("Chow", *n*, *u*)
specialmatrix("Chow", *n*, *u*, *v*)

Purpose
Generate Chow matrix.

Return Value
This function returns a Chow matrix.

Parameters
n Input integral number. It specifies the degree of the matrix. That is, the size of the Chow matrix is $n \times n$.

u An optional input value.

v An optional input value.

Description
This function generates the $n \times n$ Chow matrix. It is a singular Toeplitz lower Hessenberg matrix. This function returns a square matrix y such that $y = h + v * I$, where h is a square matrix whose elements are $h[i][j] = u^{(i-j)+1}$ and I is a $n \times n$ identity matrix. If the user does not specify the optional arguments *u* and *v*, $u = 1.0$ and $v = 0.0$ are used by default.

Example
Generate a 5th order Chow matrix with $u = 1, v = 0$(default), $u = 0.5, v = 0$ and $u = 0.5, v = 0.67$. Verify the result with the formulation $y = h + v * I$.

```
#include <numeric.h>
int main() {
    int i, j, n=5;
    double u=0.5, v=0.67;
    array double h[n][n], y[n][n];

    printf("specialmatrix(\"Chow\", n) =\n%5.4f\n", specialmatrix("Chow", n));
    printf("specialmatrix(\"Chow\", n, u) =\n%5.4f\n", specialmatrix("Chow", n, u));
    printf("specialmatrix(\"Chow\", n, u, v) =\n%5.4f\n", specialmatrix("Chow", n, u, v));
    for (i=0; i<n; i++)
       for (j=0; j<=min(i+1,n-1); j++)
          h[i][j] = pow(u,(i-j+1));
    y = h + v*identitymatrix(n);
    printf("varify matrix=\n%5.4f",y);
}
```

Output

```
specialmatrix("Chow", n) =
1.0000 1.0000 0.0000 0.0000 0.0000
1.0000 1.0000 1.0000 0.0000 0.0000
1.0000 1.0000 1.0000 1.0000 0.0000
1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000

specialmatrix("Chow", n, u) =
0.5000 1.0000 0.0000 0.0000 0.0000
0.2500 0.5000 1.0000 0.0000 0.0000
0.1250 0.2500 0.5000 1.0000 0.0000
0.0625 0.1250 0.2500 0.5000 1.0000
0.0312 0.0625 0.1250 0.2500 0.5000

specialmatrix("Chow", n, u, v) =
1.1700 1.0000 0.0000 0.0000 0.0000
0.2500 1.1700 1.0000 0.0000 0.0000
0.1250 0.2500 1.1700 1.0000 0.0000
0.0625 0.1250 0.2500 1.1700 1.0000
0.0312 0.0625 0.1250 0.2500 1.1700

varify matrix=
1.1700 1.0000 0.0000 0.0000 0.0000
0.2500 1.1700 1.0000 0.0000 0.0000
0.1250 0.2500 1.1700 1.0000 0.0000
0.0625 0.1250 0.2500 1.1700 1.0000
0.0312 0.0625 0.1250 0.2500 1.1700
```

References

T.S. Chow, A class of Hessenberg matrices with known eigenvalues and inverses, SIAM Review, 11 (1969), pp. 391-395.

. . . . . .

Circul

Synopsis

#include <numeric.h>
array double specialmatrix("Circul", **array double** *c[&]*);

Syntax

specialmatrix("Circul", *c*)

Purpose

Generate a Circul matrix.

Return Value

This function returns a $n \times n$ Circul matrix.

Parameters

c Input vector with n elements. It is the first row of Circul matrix.

Description

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting

the entries one step forward. specifies the degree of the matrix.

Example

```
#include <numeric.h>
int main() {
    array double c[5] = {2,7,1,9,6};
    printf("c=\n%f",c);
    printf("specialmatrix(\"Circul\", c) =\n%5.4f\n", specialmatrix("Circul", c));
}
```

Output

```
c=
2.000000 7.000000 1.000000 9.000000 6.000000
specialmatrix("Circul", c) =
2.0000 7.0000 1.0000 9.0000 6.0000
6.0000 2.0000 7.0000 1.0000 9.0000
9.0000 6.0000 2.0000 7.0000 1.0000
1.0000 9.0000 6.0000 2.0000 7.0000
7.0000 1.0000 9.0000 6.0000 2.0000
```

References

P.J. Davis, Circulant Matrices, John Wiley, 1977.

. . . . . .

Clement

Synopsis

#include <numeric.h>
array double specialmatrix(“Clement”, **int** *n* /* [**int** *k*] */);

Syntax

specialmatrix(“Clement”, *n*)
specialmatrix(“Clement”, *n*, *k*)

Purpose

Generate a Clement matrix.

Return Value

This function returns an $n \times n$ Clement matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Integral number 0 or 1. For $k = 0$ or by default, the matrix is unsymmetric. For $k = 1$, it is a symmetric matrix.

Description

Clement matrix is a tridiagonal matrix with zero diagonal elements and known eigenvalues. The eigenvalues

of the matrix are positive and negative numbers of $n-1, n-3, n-5, \cdots, 1$or 0.

Example

```
#include <numeric.h>
int main() {
    int n1=5, n2=6;
    array double y1[n1][n1], eigenvalue1[n1],eigenvector1[n1][n1];
    array double y2[n2][n2], eigenvalue2[n2],eigenvector2[n2][n2];

    printf("specialmatrix(\"Clement\", n1, 0) =\n%5.4f\n",
           y1=specialmatrix("Clement", n1, 0));
    printf("specialmatrix(\"Clement\", n1, 1) =\n%5.4f",
           y1=specialmatrix("Clement", n1, 1));
    eigensystem(eigenvalue1,eigenvector1, y1);
    printf("eigenvalue=\n%f\n",eigenvalue1);
    printf("specialmatrix(\"Clement\", n2, 1) =\n%5.4f",
           y2=specialmatrix("Clement", n2, 1));
    eigensystem(eigenvalue2,eigenvector2, y2);
    printf("eigenvalue=\n%f",eigenvalue2);
}
```

Output

```
specialmatrix("Clement", n1, 0) =
0.0000 1.0000 0.0000 0.0000 0.0000
4.0000 0.0000 2.0000 0.0000 0.0000
0.0000 3.0000 0.0000 3.0000 0.0000
0.0000 0.0000 2.0000 0.0000 4.0000
0.0000 0.0000 0.0000 1.0000 0.0000

specialmatrix("Clement", n1, 1) =
0.0000 2.0000 0.0000 0.0000 0.0000
2.0000 0.0000 2.4495 0.0000 0.0000
0.0000 2.4495 0.0000 2.4495 0.0000
0.0000 0.0000 2.4495 0.0000 2.0000
0.0000 0.0000 0.0000 2.0000 0.0000
eigenvalue=
4.000000 -4.000000 0.000000 2.000000 -2.000000

specialmatrix("Clement", n2, 1) =
0.0000 2.2361 0.0000 0.0000 0.0000 0.0000
2.2361 0.0000 2.8284 0.0000 0.0000 0.0000
0.0000 2.8284 0.0000 3.0000 0.0000 0.0000
0.0000 0.0000 3.0000 0.0000 2.8284 0.0000
0.0000 0.0000 0.0000 2.8284 0.0000 2.2361
0.0000 0.0000 0.0000 0.0000 2.2361 0.0000
eigenvalue=
5.000000 -5.000000 3.000000 -3.000000 1.000000 -1.000000
```

References

P.A. Clement, A class of triple-diagonal matrices for test purposes, SIAM Review, 1 (1959), pp. 50-52.

. . . . . .

DenavitHartenberg

Synopsis
#include <numeric.h>
array double specialmatrix("DenavitHartenberg", **double** *theta* , **double** *d*, **double** *alpha*, **double** *a*);

Syntax
specialmatrix("DenavitHartenberg", *theta*, *d*, *alpha*, *a*)

Purpose
Generate a Denavit-Hartenberg matrix.

Return Value
This function returns a 4×4 Denavit-Hartenberg matrix.

Parameters
theta Input θ value.

d Input d value.

alpha Input α value.

a Input a value.

Description
The Denavit-Hartenberg matrix is used to describe the coordinate transformation between adjacent links in a three dimensional space. In the Denavit-Hartenberg representation, there are three basic rules for determining and establishing the coordinate frame.

1. The z_{i-1} axis lies along the axis of motion of the ith joint.
2. The x_i axis is normal to the z_{i-1} axis, and pointing away from it.
3. The y_i axis completes the right-handed coordinate system as required.

This first Denavit-Hartenberg representation of a rigid link depends on four geometric parameters associated with each link. These four parameters completely describe any revolute or prismatic joint. Referring to the Figure11.2, these four parameters are defined as follows:

θ_i is the joint angle from the x_{i-1} axis to the x_i axis about the z_{i-1} axis (using the right-hand rule).

d_i is the distance from the origin of the $(i-1)$th coordinate frame to the intersection of the z_{i-1} axis with the x_i axis along the z_{i-1} axis.

a_i is the offset distance from the intersection of the z_i axis with the x_i axis to the origin of the ith frame along the x_i axis, or the shortest distance between the z_{i-1} and z_i axis.

α_i is the offset angle from the z_{i-1} axis to the z_i axis about the x_i axis, using the right-hand rule.

Example

```
#include<numeric.h> // with M_PI
int main() {
    array double t1[4][4], t2[4][4], t3[4][4],
                 t4[4][4], t5[4][4], t6[4][4], t[4][4];
```

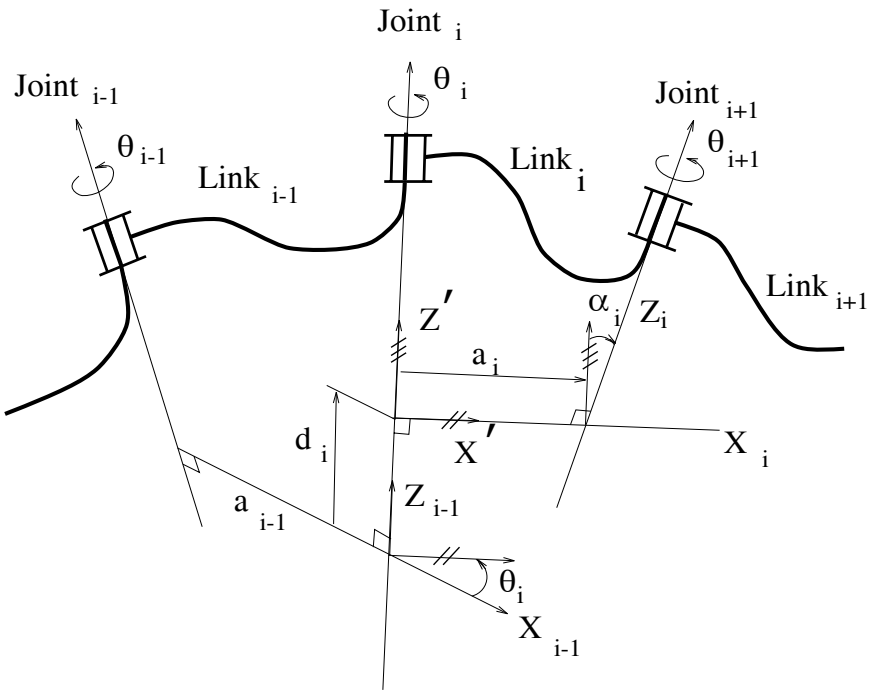


Figure 11.2: Link coordinate system and its parameters

```
    double d2r = M_PI/180;
    double d[6] = {0, 149.09, 0, 433.07, 0, 56.25};
    double alpha[6] = {-90*d2r, 0, 90*d2r, -90*d2r, 90*d2r, 0};
    double a[6] = {0, 431.8, -20.32, 0, 0, 0};
    double theta[6];

    theta[0] = 10*d2r;
    theta[1] = 20*d2r;
    theta[2] = 30*d2r;
    theta[3] = 40*d2r;
    theta[4] = 50*d2r;
    theta[5] = 60*d2r;
    t1 = specialmatrix("DenavitHartenberg",theta[0], d[0], alpha[0], a[0]);
    t2 = specialmatrix("DenavitHartenberg",theta[1], d[1], alpha[1], a[1]);
    t3 = specialmatrix("DenavitHartenberg",theta[2], d[2], alpha[2], a[2]);
    t4 = specialmatrix("DenavitHartenberg",theta[3], d[3], alpha[3], a[3]);
    t5 = specialmatrix("DenavitHartenberg",theta[4], d[4], alpha[4], a[4]);
    t6 = specialmatrix("DenavitHartenberg",theta[5], d[5], alpha[5], a[5]);
    t = t1*t2*t3*t4*t5*t6; // forward kinematics for Puma 560
    printf("t = \n%f\n", t);
}
```

Output

```
t =
-0.636562 0.022716 0.770891 730.916122
0.771180 0.029596 0.635929 308.395147
-0.008369 0.999304 -0.036358 144.208567
0.000000 0.000000 0.000000 1.000000
```

References

K. S. Fu, R. C. Gonzalez, C. S. G. Lee, *Robotics: Control, Seneing, Vision, and Intelligence*, McGraw-Hill Pub. Co., 1987.

. . . . . .

DenavitHartenberg2

Synopsis

#include <numeric.h>

array double specialmatrix("DenavitHartenberg2", **double** *theta* , **double** *d*, **double** *alpha*, **double** *a*);

Syntax

specialmatrix("DenavitHartenberg2", *theta*, *d*, *alpha*, *a*)

Purpose

generate a Denavit-Hartenberg matrix.

Return Value

This function returns a 4×4 Denavit-Hartenberg matrix.

Parameters

theta Input θ value.

d Input d value.

alpha Input α value.

a Input a value.

Description

This second Denavit-Hartenberg representation of a rigid link depends on four geometric parameters associated with each link. These four parameters completely describe any revolute or prismatic joint. Referring to the Figure11.3, these four parameters are defined as follows:

θ_i is the joint angle from the x_{i-1} axis to the x_i axis about the z_i axis, using the right-hand rule.

d_i is the distance from the origin of the $(i-1)$th coordinate frame to the intersection of the z_i axis with the x_i axis along the z_i axis.

a_i is the offset distance from the intersection of the z_i axis with the x_i axis to the origin of the $(i+1)$th frame along the x_i axis, or the shortest distance between the z_i and z_{i+1} axis.

α_i is the offset angle from the z_i axis to the z_{i+1} axis about the x_i axis, using the right-hand rule.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"DenavitHartenberg2\",M_PI, 0.0, -M_PI, 0.0) =\n%5.4f\n",
            specialmatrix("DenavitHartenberg2",M_PI, 0.0, -M_PI, 0.0));
}
```

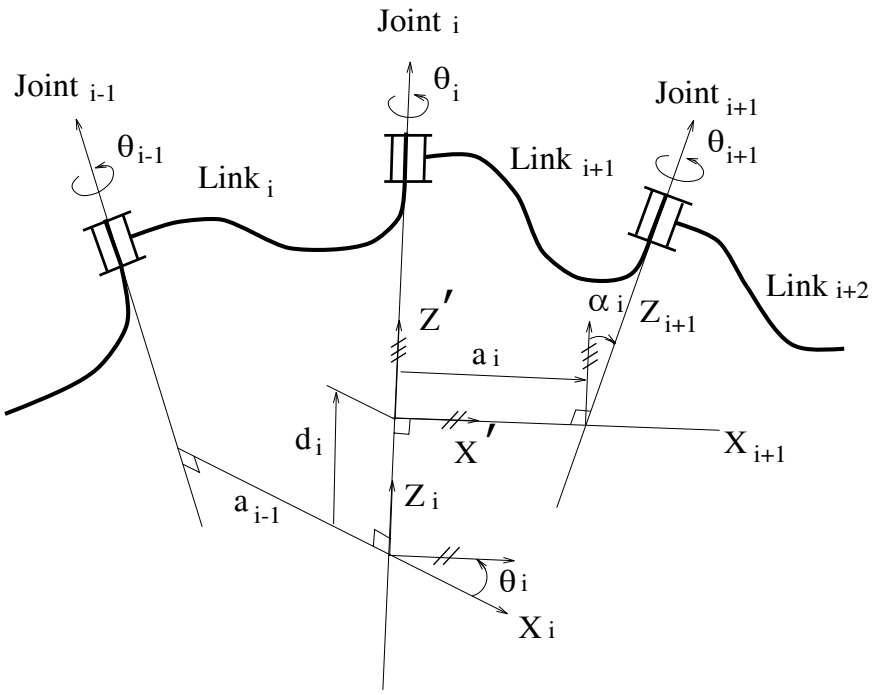


Figure 11.3: Link coordinate system and its parameters

Output

```
specialmatrix("DenavitHartenberg2",M_PI, 0.0, -M_PI, 0.0) =
-1.0000 0.0000 0.0000 0.0000
0.0000 1.0000 -0.0000 -0.0000
-0.0000 -0.0000 -1.0000 -0.0000
0.0000 0.0000 0.0000 1.0000
```

References
John J. Craig, *Robotics: Mechanics & Control*, Addison-Wesley Pub. Co., 1986.

. . . . . .

Dramadah

Synopsis
#include <**numeric.h**>
array double specialmatrix(“Dramadah1”, **int** *n* /* [**int** *k*] */);

Syntax
specialmatrix(“Dramadah1”, *n*)
specialmatrix(“Dramadah1”, *n*, *k*)

Purpose
generate a Dramadah matrix.

Return Value

This function returns an $n \times n$ Dramadah matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Integral number 1, 2, or 3.

$k = 1$ or by default, the returned matrix *y* is Toeplitz, with *abs*$|y| = 1$. Each element of y^{-1} is an integral value.

$k = 2$, *y* is upper triangular and Toeplitz. Each element of y^{-1} is an integral value.

$k = 3$, *y* has a maximal determinant among (0,1) lower Hessenberg matrices.

Description

The Dramadah matrix has elements of zeros and ones. The inverse matrix of Dramadah matrix has large integer elements.

Example

```
#include <numeric.h>
int main() {
    int n=4;
    array double y[n][n];
    printf("specialmatrix(\"Dramadah\", n) =\n%5.4f", y=specialmatrix("Dramadah", n));
    printf("absolute value of determinant of y=%f\n",abs(determinant(y)));
    printf("inverse of y=\n%5.4f\n",inverse(y));
    printf("specialmatrix(\"Dramadah\", n,2) =\n%5.4f", y=specialmatrix("Dramadah", n,2));
    printf("inverse of y=\n%5.4f\n",inverse(y));
    printf("specialmatrix(\"Dramadah\", n,3) =\n%5.4f", y=specialmatrix("Dramadah", n,3));
    printf("determinant of y=%f\n",determinant(y));
}
```

Output

```
specialmatrix("Dramadah", n) =
1.0000 1.0000 0.0000 1.0000
0.0000 1.0000 1.0000 0.0000
0.0000 0.0000 1.0000 1.0000
1.0000 0.0000 0.0000 1.0000
absolute value of determinant of y=1.000000
inverse of y=
-1.0000 1.0000 -1.0000 2.0000
1.0000 0.0000 0.0000 -1.0000
-1.0000 1.0000 0.0000 1.0000
1.0000 -1.0000 1.0000 -1.0000

specialmatrix("Dramadah", n,2) =
1.0000 1.0000 0.0000 1.0000
0.0000 1.0000 1.0000 0.0000
0.0000 0.0000 1.0000 1.0000
0.0000 0.0000 0.0000 1.0000
inverse of y=
1.0000 -1.0000 1.0000 -2.0000
0.0000 1.0000 -1.0000 1.0000
0.0000 0.0000 1.0000 -1.0000
0.0000 0.0000 0.0000 1.0000
```

```
specialmatrix("Dramadah", n,3) =
1.0000 1.0000 0.0000 0.0000
0.0000 1.0000 1.0000 0.0000
1.0000 0.0000 1.0000 1.0000
0.0000 1.0000 0.0000 1.0000
determinant of y=3.000000
```

References

R.L. Graham and N.J.A. Sloane, Anti-Hadamard matrices, Linear Algebra and Appl., 62 (1984), pp. 113-137. L. Ching, The maximum determinant of an nxn lower Hessenberg (0,1) matrix, Linear Algebra and Appl., 183 (1993), pp. 147-153.

. . . . . .

Fiedler

Synopsis

#include <numeric.h>
array double specialmatrix("Fiedler", **array double** *c[&]*);

Syntax

specialmatrix("Fiedler", *c*)

Purpose

Generate a Fiedler matrix.

Return Value

This function returns an $n \times n$ Fiedler matrix.

Parameters

c Input vector with n elements. The length of the vector n specifies the order of the output matrix. That is, the size of the Fiedler matrix is $n \times n$.

Description

A Fiedler matrix has a dominant positive eigenvalue and all the other eigenvalues are negative. A Fiedler matrix is a symmetric matrix with elements *abs(c[i] - c[j])*.

Example

```
#include <numeric.h>
int main() {
    int n=5;
    array double y[n][n], eigenvalue[n], eigenvector[n][n];
    array double c[5] = {2,-7,1,-9,6};
    printf("specialmatrix(\"Fiedler\",c) =\n%5.4f", y=specialmatrix("Fiedler", c));
    eigensystem(eigenvalue,eigenvector, y);
    printf("eigenvalues =\n%f",eigenvalue);
}
```

Output

```
specialmatrix("Fiedler",c) =
0.0000 9.0000 1.0000 11.0000 4.0000
9.0000 0.0000 8.0000 2.0000 13.0000
1.0000 8.0000 0.0000 10.0000 5.0000
11.0000 2.0000 10.0000 0.0000 15.0000
4.0000 13.0000 5.0000 15.0000 0.0000
eigenvalues =
-23.319294 32.070623 -0.904663 -5.933579 -1.913088
```

References

G. Szego, Solution to problem 3705, Amer. Math. Monthly, 43 (1936), pp. 246-259.
J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra, Birkhauser, Basel, and Academic Press, New York, 1977, p. 159.

. . . . . .

Frank

Synopsis

#include <numeric.h>
array double specialmatrix("Frank", **int** *n* /* [**int** *k*] */);

Syntax

specialmatrix("Frank", *n*)
specialmatrix("Frank", *n*, *k*)

Purpose

Generate a Frank matrix.

Return Value

This function returns an $n \times n$ Frank matrix.

Parameters

n Input integral number. It specifies the order of the square matrix produced.

k Integral number 0 or 1. For $k = 0$ or by default, the matrix is an upper Hessenberg with determinant 1. If $k = 1$, the elements are reflected about the anti-diagonal.

Description

A Frank matrix is an upper Hessenberg with ill-conditioned eigenvalues. The eigenvalues of the Frank matrix are positive and occur in reciprocal pairs. If n is odd, 1 is an eigenvalue. The first m, with $m = \mathit{floor}(n/2)$, eigenvalues in the ascending order are ill-conditioned.

Example

```
#include <numeric.h>
int main() {
    int n1=4, n2=5;
    array double y1[n1][n1], eigenvalue1[n1], eigenvector1[n1][n1];
    array double y2[n2][n2], eigenvalue2[n2], eigenvector2[n2][n2];
    printf("specialmatrix(\"Frank\",n1) =\n%5.4f", y1=specialmatrix("Frank", n1));
    eigensystem(eigenvalue1,eigenvector1,y1);
    printf("eigenvalues =\n%5.4f",eigenvalue1);
```

```
    printf("determinant =%5.4f\n",determinant(y1));
    printf("specialmatrix(\"Frank\",n2, 1) =\n%5.4f", y2=specialmatrix("Frank", n2, 1));
    eigensystem(eigenvalue2,eigenvector2,y2);
    printf("eigenvalues =\n%5.4f",eigenvalue2);
    printf("determinant =%5.4f\n",determinant(y2));
}
```

Output

```
specialmatrix("Frank",n1) =
4.0000 3.0000 2.0000 1.0000
3.0000 3.0000 2.0000 1.0000
0.0000 2.0000 2.0000 1.0000
0.0000 0.0000 1.0000 1.0000
eigenvalues =
0.1367 7.3127 2.0666 0.4839
determinant =1.0000
specialmatrix("Frank",n2, 1) =
1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 2.0000 2.0000 2.0000 2.0000
0.0000 2.0000 3.0000 3.0000 3.0000
0.0000 0.0000 3.0000 4.0000 4.0000
0.0000 0.0000 0.0000 4.0000 5.0000
eigenvalues =
3.5566 10.0629 1.0000 0.0994 0.2812
determinant =1.0000
```

References

W.L. Frank, Computing eigenvalues of complex matrices by determinant evaluation and by methods of Danilewski and Wielandt, J. Soc. Indust. Appl. Math., 6 (1958), pp. 378-392 (see pp. 385 and 388).

G.H. Golub and J.H. Wilkinson, Ill-conditioned eigensystems and the computation of the Jordan canonical form, SIAM Review, 18 (1976), pp. 578-619 (Section 13).

H. Rutishauser, On test matrices, Programmation en Mathematiques Numeriques, Editions Centre Nat. Recherche Sci., Paris, 165, 1966, pp. 349-365. Section 9.

J.H. Wilkinson, Error analysis of floating-point computation, Numer. Math., 2 (1960), pp. 319-340 (Section 8).

J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University Press, 1965 (pp. 92-93).

P.J. Eberlein, A note on the matrices denoted by B_n, SIAM J. Appl. Math., 20 (1971), pp. 87-92.

J.M. Varah, A generalization of the Frank matrix, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 835-839.

Gear

Synopsis

#include <numeric.h>

array double specialmatrix(“Gear”, **int** *n*, ... /* [**int** *ni*, **int** *nj*] */);

Syntax

specialmatrix(“Gear”, *n*)

specialmatrix(“Gear”, *n*, *ni*)

specialmatrix(“Gear”, *n*, *ni*, *nj*)

Purpose
Generate a Gear matrix.

Return Value
This function returns an $n \times n$ Gear matrix.

Parameters
n Input integral number. It specifies the order of the square matrix produced.

ni Integral number *abs(ni)*$\leq n$.

nj Integral number *abs(nj)*$\leq n$.

Description
A Gear matrix is an $n \times n$ matrix with ones on the sub- and super-diagonals. The [0][abs(ni)] element is assigned the value *sign(ni)* and the [n-1][n-1-abs(nj)] is assigned *sign(nj)*. All eigenvalues of Gear matrix are of the form $2 * cos(a)$ and the eigenvectors are of the form $[sin(w+a), sin(w+2a), ..., sin(w+Na)]$, where a and w are constants.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"Gear\",7) =\n%5.4f\n", specialmatrix("Gear", 7));
    printf("specialmatrix(\"Gear\",7,3,6) =\n%5.4f\n", specialmatrix("Gear", 7, 3, 6));
}
```

Output

```
specialmatrix("Gear",7) =
0.0000 1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1.0000 0.0000 1.0000 0.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 0.0000 1.0000 0.0000 1.0000 0.0000
0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 1.0000
-1.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0000

specialmatrix("Gear",7,3,6) =
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0000
1.0000 0.0000 1.0000 0.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 1.0000 0.0000 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 0.0000 1.0000 0.0000 1.0000 0.0000
0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 1.0000
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0000
```

References
C.W. Gear, A simple set of test matrices for eigenvalue programs, Math. Comp., 23 (1969), pp. 119-125.

Hadamard

Synopsis
#include <numeric.h>
array double specialmatrix(“Hadamard”, **int** *n*);

Syntax
specialmatrix(“Hadamard”, *n*)

Purpose
Generate a Hadamard matrix.

Return Value
This function returns an $n \times n$ Hadamard matrix.

Parameters
n Input integral number. It specifies the degree of the matrix. That is, the size of Hadamard matrix is $n \times n$.

Description
This function generates an $n \times n$ Hadamard matrix. A Hadamard matrix is a matrix *y* with elements 1 or -1 such that $y^T * y = n * I$, where I is an $n \times n$ identity matrix. An $n \times n$ Hadamard matrix with $n > 2$ exists only if *n%4 = 0*. This function handles only the cases where $n, n/12$ or $n/20$ is a power of 2.

Example

```
#include <numeric.h>
int main() {
    int n=8;
    array double y[n][n];
    printf("specialmatrix(\"Hadamard\", n) =\n%4.1f", y=specialmatrix("Hadamard", n));
    printf("verify: transpose(y)*y-n*identitymatrix(n)=0\n%5.4f",
            transpose(y)*y-n*identitymatrix(n));
}
```

Output

```
specialmatrix("Hadamard", n) =
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0 -1.0  1.0 -1.0  1.0 -1.0  1.0 -1.0
 1.0  1.0 -1.0 -1.0  1.0  1.0 -1.0 -1.0
 1.0 -1.0 -1.0  1.0  1.0 -1.0 -1.0  1.0
 1.0  1.0  1.0  1.0 -1.0 -1.0 -1.0 -1.0
 1.0 -1.0  1.0 -1.0 -1.0  1.0 -1.0  1.0
 1.0  1.0 -1.0 -1.0 -1.0 -1.0  1.0  1.0
 1.0 -1.0 -1.0  1.0 -1.0  1.0  1.0 -1.0
verify: transpose(y)*y-n*identitymatrix(n)=0
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
```

References
Reference: S.W. Golomb and L.D. Baumert, The search for Hadamard matrices, Amer. Math. Monthly, 70 (1963) pp. 12-17.

Hankel

Synopsis
#include <numeric.h>
array double specialmatrix("Hankel", **array double** *c[&]*, ... /* [**array double** *r[&]*] */);

Syntax
specialmatrix("Hankel", *c*)
specialmatrix("Hankel", *c*, *r*)

Purpose
Generate a Hankel matrix.

Return Value
This function returns a Hankel matrix.

Parameters
c Input vector with n elements.

r An optional input vector with m elements.

Description
If the optional input vector *r* is not specified, this function generates a square Hankel matrix whose column is *c* and whose elements are zero below the anti-diagonal. If the optional input vector *r* is specified, it returns an $n \times m$ Hankel matrix whose first column is *c* and last row is *r*. Hankel matrices are symmetric.

Example

```
#include <numeric.h>
int main() {
    array double r[3] ={2,6,8};
    array double c[5] ={8,2.3,9,6,2};
    printf("specialmatrix(\"Hankel\", c) =\n%5.1f\n", specialmatrix("Hankel", c));
    printf("specialmatrix(\"Hankel\", c, r) =\n%5.1f\n", specialmatrix("Hankel", c, r));
    printf("specialmatrix(\"Hankel\", r, c) =\n%5.1f\n", specialmatrix("Hankel", r, c));
}
```

Output

```
specialmatrix("Hankel", c) =
  8.0   2.3   9.0   6.0   2.0
  2.3   9.0   6.0   2.0   0.0
  9.0   6.0   2.0   0.0   0.0
  6.0   2.0   0.0   0.0   0.0
  2.0   0.0   0.0   0.0   0.0
```

```
specialmatrix("Hankel", c, r) =
  8.0   2.3   9.0
  2.3   9.0   6.0
  9.0   6.0   2.0
  6.0   2.0   6.0
  2.0   6.0   8.0

specialmatrix("Hankel", r, c) =
  2.0   6.0   8.0   2.3   9.0
  6.0   8.0   2.3   9.0   6.0
  8.0   2.3   9.0   6.0   2.0
```

References

.....

Hilbert

Synopsis
#include <numeric.h>
array double specialmatrix("Hilbert", **int** *n*);

Syntax
specialmatrix("Hilbert", *n*)

Purpose
Generate a Hilbert matrix.

Return Value
This function returns an $n \times n$ Hilbert matrix.

Parameters
n Input integral number. It specifies the degree of the matrix. That is, the size of Hilbert matrix is $n \times n$.

Description
This function generates an $n \times n$ Hilbert matrix. A Hilbert matrix is a matrix *y* with elements $y[i][j] = 1/(i + j + 1)$. It is a famous example of a badly conditioned matrix.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"Hilbert\", 7) =\n%5.3f\n", specialmatrix("Hilbert", 7));
}
```

Output

```
specialmatrix("Hilbert", 7) =
1.000 0.500 0.333 0.250 0.200 0.167 0.143
0.500 0.333 0.250 0.200 0.167 0.143 0.125
0.333 0.250 0.200 0.167 0.143 0.125 0.111
0.250 0.200 0.167 0.143 0.125 0.111 0.100
```

```
0.200 0.167 0.143 0.125 0.111 0.100 0.091
0.167 0.143 0.125 0.111 0.100 0.091 0.083
0.143 0.125 0.111 0.100 0.091 0.083 0.077
```

References

. . . . . .

InverseHilbert

Synopsis
#include <numeric.h>
array double specialmatrix("InverseHilbert", **int** *n*);

Syntax
specialmatrix("InverseHilbert", *n*)

Purpose
generate an inverse Hilbert matrix.

Return Value
This function returns an $n \times n$ inverse Hilbert matrix.

Parameters
n Input integral number. It specifies the degree of the matrix. That is, the size of the inverse Hilbert matrix is $n \times n$.

Description
This function generates an $n \times n$ inverse Hilbert matrix. The result is exact for n less than 15.

Example

```
#include <numeric.h>
int main() {
    printf("specialmatrix(\"InverseHilbert\", 7) =\n%8.4g\n",
            specialmatrix("InverseHilbert", 7));
}
```

Output

```
specialmatrix("InverseHilbert", 7) =
      49    -1176     8820 -2.94e+04 4.851e+04 -3.881e+04 1.201e+04
   -1176 3.763e+04 -3.175e+05 1.129e+06 -1.94e+06 1.597e+06 -5.045e+05
    8820 -3.175e+05 2.858e+06 -1.058e+07 1.871e+07 -1.572e+07 5.045e+06
-2.94e+04 1.129e+06 -1.058e+07 4.032e+07 -7.276e+07 6.209e+07 -2.018e+07
4.851e+04 -1.94e+06 1.871e+07 -7.276e+07 1.334e+08 -1.153e+08 3.784e+07
-3.881e+04 1.597e+06 -1.572e+07 6.209e+07 -1.153e+08 1.006e+08 -3.33e+07
1.201e+04 -5.045e+05 5.045e+06 -2.018e+07 3.784e+07 -3.33e+07 1.11e+07
```

References

Magic

Synopsis
#include <numeric.h>
array double specialmatrix("Magic", **int** *n*);

Syntax
specialmatrix("Magic", *n*)

Purpose
generate a Magic matrix.

Return Value
This function returns an $n \times n$ Magic matrix.

Parameters
n Input integral number. It specifies the degree of the matrix. That is, the size of Magic matrix is $n \times n$.

Description
This function generates an $n \times n$ Magic matrix. A Magic matrix is a matrix *y* with elements from 1 through n^2 with equal row, column, and diagonal sums.

Example

```
#include <numeric.h>
int main() {
    int i, n=7;
    array double y[n][n], sum1[n];
    printf("specialmatrix(\"Magic\",n) =\n%4.0f", y=specialmatrix("Magic", n));
    sum(y,sum1);
    printf("Sum of columns=\n%4.0f",sum1);
    sum(transpose(y),sum1);
    printf("Sum of rows=\n%4.0f",sum1);
    printf("Sum of diagonal elements=%4.0f\n",trace(y));
    rot90(y, y);
    printf("Sum of anti-diagonal elements=%4.0f\n",trace(y));
}
```

Output

```
specialmatrix("Magic",n) =
  30   38   46    5   13   21   22
  39   47    6   14   15   23   31
  48    7    8   16   24   32   40
   1    9   17   25   33   41   49
  10   18   26   34   42   43    2
  19   27   35   36   44    3   11
  28   29   37   45    4   12   20
```

```
Sum of columns=
 175  175  175  175  175  175  175
Sum of rows=
 175  175  175  175  175  175  175
Sum of diagonal elements= 175
Sum of anti-diagonal elements= 175
```

Pascal

Synopsis
#include <numeric.h>
array double specialmatrix("Pascal", **int** *n* /* [**int** *k*] */);

Syntax
specialmatrix("Pascal", *n*)
specialmatrix("Pascal", *n*, *k*)

Purpose
generate a Pascal matrix.

Return Value
This function returns an $n \times n$ Pascal matrix.

Parameters
n Input integral number. It specifies the order of the square matrix produced.

k Optional input Integral number 0, 1 or 2.

Description
If the optional input *k* is not specified by default, or specified with 0, it returns a symmetric positive definite matrix with integer elements, which is made up from Pascal's triangular. Its inverse matrix has integer elements. If the optional input *k* is specified with 1, it is a lower triangular Choleskey factor of the Pascal matrix. Its inverse matrix is itself. If the optional input *k* is specified with 2, it is a transposed and permuted version of *specialmatrix("Pascal",n,1)* which is a cubic root of the identity.

Example

```
#include <numeric.h>
int main() {
    int n=5;
    array double y[n][n];

    printf("specialmatrix(\"Pascal\", n) =\n%5.1f", y=specialmatrix("Pascal", n));
    printf("inverse matrix=\n%5.1f\n",inverse(y));
    printf("specialmatrix(\"Pascal\", n, 1) =\n%5.1f", y=specialmatrix("Pascal", n, 1));
    printf("inverse matrix=\n%5.1f\n",inverse(y));
    printf("specialmatrix(\"Pascal\", n, 2) =\n%5.1f", y=specialmatrix("Pascal", n, 2));
}
```

Output

```
specialmatrix("Pascal", n) =
  1.0   1.0   1.0   1.0   1.0
  1.0   2.0   3.0   4.0   5.0
  1.0   3.0   6.0  10.0  15.0
  1.0   4.0  10.0  20.0  35.0
  1.0   5.0  15.0  35.0  70.0
inverse matrix=
  5.0 -10.0  10.0  -5.0   1.0
-10.0  30.0 -35.0  19.0  -4.0
 10.0 -35.0  46.0 -27.0   6.0
 -5.0  19.0 -27.0  17.0  -4.0
  1.0  -4.0   6.0  -4.0   1.0

specialmatrix("Pascal", n, 1) =
  1.0   0.0   0.0   0.0   0.0
  1.0  -1.0   0.0   0.0   0.0
  1.0  -2.0   1.0   0.0   0.0
  1.0  -3.0   3.0  -1.0   0.0
  1.0  -4.0   6.0  -4.0   1.0
inverse matrix=
  1.0   0.0   0.0   0.0   0.0
  1.0  -1.0   0.0   0.0   0.0
  1.0  -2.0   1.0   0.0   0.0
  1.0  -3.0   3.0  -1.0   0.0
  1.0  -4.0   6.0  -4.0   1.0

specialmatrix("Pascal", n, 2) =
  0.0   0.0   0.0   0.0  -1.0
  0.0   0.0   0.0  -1.0   4.0
  0.0   0.0   1.0   3.0  -6.0
  0.0  -1.0  -2.0  -3.0   4.0
 -1.0  -1.0  -1.0  -1.0   1.0
```

References

. . . . . .

Rosser

Synopsis
#include <**numeric.h**>
array double specialmatrix(“Rosser”);

Syntax
specialmatrix(“Rosser”)

Purpose
Generate a Rosser matrix.

Return Value
This function returns an 8×8 Rosser matrix.

Description

A Rosser matrix is an 8×8 matrix *R*.

$$\begin{bmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{bmatrix}$$

This matrix was a challenge for many matrix eigenvalue algorithms. It has:

- Two eigenvalues with the same value.
- Three nearly equal eigenvalues.
- Dominant eigenvalues of opposite sign.
- A zero eigenvalue.
- A small, nonzero eigenvalue.

Example

```
#include <numeric.h>
int main() {
    array double y[8][8], eigenvalue[8], eigenvector[8][8];
    printf("specialmatrix(\"Rosser\") =\n%5.1f\n", y=specialmatrix("Rosser"));
    eigensystem(eigenvalue, eigenvector,y);
    printf("eigenvalue =\n%7.3f",eigenvalue);
}
```

Output

```
specialmatrix("Rosser") =
611.0 196.0 -192.0 407.0  -8.0 -52.0 -49.0  29.0
196.0 899.0 113.0 -192.0 -71.0 -43.0  -8.0 -44.0
-192.0 113.0 899.0 196.0  61.0  49.0   8.0  52.0
407.0 -192.0 196.0 611.0   8.0  44.0  59.0 -23.0
 -8.0 -71.0  61.0   8.0 411.0 -599.0 208.0 208.0
-52.0 -43.0  49.0  44.0 -599.0 411.0 208.0 208.0
-49.0  -8.0   8.0  59.0 208.0 208.0  99.0 -911.0
 29.0 -44.0  52.0 -23.0 208.0 208.0 -911.0  99.0

eigenvalue =
-1020.049  -0.000   0.098 1020.049 1020.000 1019.902 1000.000 1000.000
```

 .....

Toeplitz

Synopsis
#include <numeric.h>

array double specialmatrix("Toeplitz", **array double** *c[&]*, ... /* [**array double** *r[&]*] */);

Syntax
specialmatrix("Toeplitz", *c*)
specialmatrix("Toeplitz", *c*, *r*)

Purpose
Generate a Toeplitz matrix.

Return Value
This function returns a Toeplitz matrix.

Parameters
c Input vector with n elements.

r An optional input vector with m elements.

Description
If the optional input vector *r* is not specified, this function generates a square symmetric Toeplitz matrix whose first column is *c*. If the optional input vector *r* is specified, it returns an $n \times m$ non-symmetric Toeplitz matrix whose first column is *c* and first row is *r*.

Example

```
#include <numeric.h>
int main() {
    array double r[3] ={8,6,2};
    array double c[5] ={8,2.3,9,6,2};
    printf("specialmatrix(\"Toeplitz\", c) =\n%5.1f\n", specialmatrix("Toeplitz", c));
    printf("specialmatrix(\"Toeplitz\", c, r) =\n%5.1f\n",
            specialmatrix("Toeplitz", c, r));
    printf("specialmatrix(\"Toeplitz\", r, c) =\n%5.1f\n",
            specialmatrix("Toeplitz", r, c));
}
```

Output

```
specialmatrix("Toeplitz", c) =
  8.0   2.3   9.0   6.0   2.0
  2.3   8.0   2.3   9.0   6.0
  9.0   2.3   8.0   2.3   9.0
  6.0   9.0   2.3   8.0   2.3
  2.0   6.0   9.0   2.3   8.0

specialmatrix("Toeplitz", c, r) =
  8.0   6.0   2.0
  2.3   8.0   6.0
  9.0   2.3   8.0
  6.0   9.0   2.3
  2.0   6.0   9.0

specialmatrix("Toeplitz", r, c) =
```

```
  8.0   2.3   9.0   6.0   2.0
  6.0   8.0   2.3   9.0   6.0
  2.0   6.0   8.0   2.3   9.0
```

Vandermonde

Synopsis
#include <**numeric.h**>
array double specialmatrix("Vandermonde",**array double** *v*[&])[:][:];

syntax specialmatrix("Vandermonde", *v*);
Purpose
generate a Calculate the Vandermonde.

Return Value
This function returns an $n \times n$ Vandermonde matrix.

Parameters
v An input vector of size n.

Description
This function calculates the Vandermonde matrix of a vector of *n* elements. The Vandermonde matrix is an $n \times n$ matrix defined as:

$$a_{i,j} = v_i^{j-1}$$

The Vandermonde matrix can be used in the fitting of polynomial to data.

Example

```
#include <numeric.h>
int main() {
    array double v[3] = {2,3,4};
    printf("specialmatrix(\"Vandermonde\", v) =\n%f\n",
           specialmatrix("Vandermonde", v));
}
```

Output

```
specialmatrix("Vandermonde", v) =
1.000000 2.000000 4.000000
1.000000 3.000000 9.000000
1.000000 4.000000 16.000000
```

See Also
polyfit().

Wilkinson

Synopsis
#include <numeric.h>
array double specialmatrix("Wilkinson", **int** *n*);

Syntax
specialmatrix("Wilkinson", *n*)

Purpose
generate a Wilkinson matrix.

Return Value
This function returns an $n \times n$ Wilkinson's eigenvalue testing matrix.

Parameters
n Input integral number. It specifies the order of the square matrix produced.

Description
Wilkinson's eigenvalue testing matrix is a symmetric, tridiagonal matrix with pairs of nearly, but not exactly, equal eigenvalues.

Example

```
#include <numeric.h>
int main() {
    int n=7;
    array double y[n][n], eigenvalue[n], eigenvector[n][n];
    printf("specialmatrix(\"Wilkinson\", n) =\n%5.1f", y=specialmatrix("Wilkinson", n));
    eigensystem(eigenvalue, eigenvector, y);
    printf("eigenvalues =\n%5.1f",eigenvalue);
}
```

Output

```
specialmatrix("Wilkinson", n) =
  3.0   1.0   0.0   0.0   0.0   0.0   0.0
  1.0   3.0   1.0   0.0   0.0   0.0   0.0
  0.0   1.0   3.0   1.0   0.0   0.0   0.0
  0.0   0.0   1.0   3.0   1.0   0.0   0.0
  0.0   0.0   0.0   1.0   3.0   1.0   0.0
  0.0   0.0   0.0   0.0   1.0   3.0   1.0
  0.0   0.0   0.0   0.0   0.0   1.0   3.0
eigenvalues =
  4.8   1.2   1.6   4.4   3.0   3.8   2.2
```

sqrtm

Synopsis
#include <numeric.h>
int sqrtm(array double complex *y[&][&]*, **array double complex** *x[&][&]*);

Syntax
sqrtm(*y*, *x*)

Purpose
Computes the square root of the matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x Input square matrix. It contains data to be calculated.

y Output square matrix which contains data of the result of the principal square root of the matrix *x*.

Description
This function computes the principal square root of the matrix *x*, that is, $x = y * y$ where *x* and *y* are square matrices. If *x* has non-positive eigenvalues, the result *y* is a real matrix. If *x* has any eigenvalue of negative real part then a complex result *y* will be produced.

Example
Calculation of square roots of matrices of different data types.

```
#include <numeric.h>
int main() {
   array double x[3][3]={0.8,0.2,0.1,
                         0.2,0.7,0.3,
                         0.1,0.3,0.6};
   array double complex zx[3][3]={complex(1,1),complex(2,2),0,
                         3,complex(4,1),complex(2,5),
                         0,0,0};
   array double complex zy[3][3];
   array double y[3][3];

   sqrtm(y,x);
   printf("x = \n%5.3f",x);
   printf("y = \n%5.3f",y);
   printf("\n");

   sqrtm(zy,zx);
   printf("zx = \n%5.3f",zx);
   printf("zy = \n%5.3f",zy);

}
```

Output

```
x =
0.800 0.200 0.100
0.200 0.700 0.300
0.100 0.300 0.600
y =
0.886 0.113 0.048
0.113 0.807 0.189
0.048 0.189 0.750

zx =
complex(1.000,1.000) complex(2.000,2.000) complex(0.000,0.000)
complex(3.000,0.000) complex(4.000,1.000) complex(2.000,5.000)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
zy =
complex(0.690,0.818) complex(1.003,0.353) complex(-3.155,0.649)
complex(1.017,-0.488) complex(1.708,0.331) complex(3.068,1.046)
complex(0.000,0.000) complex(0.000,0.000) complex(0.000,0.000)
```

See Also

logm(), **funm**(), **cfunm**(), **expm**().

References

G. H. Golub, C. F. Van Loan, Matrix Computations Third edition, The Johns Hopkins University Press, 1996

std

Synopsis
#include <**numeric.h**>
double std(**array double** &*a*, ... /* [**array double** *v*[:]] */);

Syntax
std(a)
std(a, v)

Purpose
Calculate the standard deviation of a data set.

Return Value
This function returns the standard deviation.

Parameters
a Input array which contains a data set.

v Output array which contains the standard deviations of each row of the array.

Description
This function calculates the standard deviation of a data set.

Algorithm
The standard deviation of the data set σ_i is defined as

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(X_j - \overline{X})^2}$$

where N is the number of observations of data set *x*, $\overline{x}$ is the mean of data set *x*.

Example1

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    int    b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double stdval;

    stdval = std(a);
    printf("std(a) = %f\n", stdval);
    stdval = std(b);
    printf("std(b) = %f\n", stdval);
    stdval = std(b1);
    printf("std(b1) = %f\n", stdval);
    stdval = std(c);
```

```
    printf("std(c) = %f\n", stdval);
}
```

Output1

```
std(a) = 1.000000
std(b) = 1.870829
std(b1) = 1.870829
std(c) = 4.025779
```

Example2

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double stdv1[2], stdv2[3];

    std(a, stdv1);
    printf("std(a, stdv1) = %f\n", stdv1);
    std(b, stdv2);
    printf("std(b, stdv2) = %f\n", stdv2);
}
```

Output2

```
std(a, stdv1) = 1.000000 1.000000

std(b, stdv2) = 1.290994 1.290994 1.290994
```

See Also
mean(), **median**().

References

sum

Synopsis
#include <**numeric.h**>
double sum(**array double** &*a*, ... /*[**array double** *v*[:]]*/);

Syntax
sum(a)
sum(a, v)

Purpose
Calculate the sum of all elements of an array and the sums of each row of a two-dimensional array.

Return Value
This function returns the sum of all elements of an array.

Parameters
a The input array.

v The output array which contains the sums of each row of array.

Description
This function calculates the sum of all elements in an array of any dimension. If the array is a two-dimensional matrix, the function can calculate the sum of each row. The input array can be any arithmetic data type, and of any dimension.

Example1
Calculate the sum of all elements of arrays of different data types.

```
#include <numeric.h>
int main() {
    double a[3] = {1,2,3};
    double b[2][3] = {1,2,3,4,5,6};
    int b1[2][3] = {1,2,3,4,5,6};
    double c[2][3][5];
    c[0][0][0] = 10;
    c[0][0][1] = 20;
    double sums;

    sums = sum(a);
    printf("sum(a) = %f\n", sums);
    sums = sum(b);
    printf("sum(b) = %f\n", sums);
    sums = sum(b1);
    printf("sum(b) = %f\n", sums);
    sums = sum(c);
    printf("sum(c) = %f\n", sums);
}
```

Output1

```
sum(a) = 6.000000
sum(b) = 21.000000
sum(b) = 21.000000
sum(c) = 30.000000
```

Example2
Calculate the sums of each elements of the arrays

```
#include <numeric.h>
int main() {
    double a[2][3] = {1,2,3,
                      4,5,6};
    array double b[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array int    b1[3][4] = {1,2,3,4,
                            5,6,7,8,
                            1,2,3,4};
    array double sumv1[2], sumv2[3];
    double s;

    s = sum(a, sumv1);
    printf("sum(a) = %f\n", sumv1);
    sum(b, sumv2);
    printf("sum(b) = %f\n", sumv2);
    sum(b1, sumv2);
    printf("sum(b1) = %f\n", sumv2);
}
```

Output2

```
sum(a) = 6.000000 15.000000

sum(b) = 10.000000 26.000000 10.000000

sum(b1) = 10.000000 26.000000 10.000000
```

See Also
csum(), **cumsum**(), cumprod(), **trace**(), **product**().

References

svd

Synopsis
#include <numeric.h>
int svd(**array double complex** *a*[&][&], **array double** *s*[&], **array double complex** *u*[&][&],
array double complex *vt*[&][&]);

Purpose
Compute singular value decomposition of a matrix.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
a Input matrix.

s Output S vector.

u Output U matrix.

vt Output V matrix.

Description
The function computes the singular value of a real /complex m-by-n matrix, and the left and/or right singular vectors. The SVD is formulated as

$$A = USV^T$$

Where S is an m-by-n matrix which is zero except for its min(m,n) diagonal elements, U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of S are the singular value of A; they are real, non-negative, and are in descending order. The first min(m,n) column of U and V are the left and right singular vectors of A.

Example1
Singular value decomposition of a real n-by-n matrix.

```
#include <numeric.h>
int main() {
     array double a[3][3] = {7, 8, 1,
                             3, 6, 4,
                             3, 5, 7};   /* m-by-n matrix */
     array float  b[3][3] = {7, 8, 1,
                             3, 6, 4,
                             3, 5, 7};   /* m-by-n matrix */
     int m = 3, n = 3;
     int lmn = min(m,n);
     array double s[lmn];
     array double u[m][m];
     array double vt[n][n];
     array float fs[lmn];
     array float fu[m][m];
     array float fvt[n][n];
     int status;
```

```
    svd(a, s, NULL, NULL);
    printf("s =\n%f\n", s);

    svd(a, s, u, vt);
    printf("u =\n%f\n", u);
    printf("s =\n%f\n", s);
    printf("vt =\n%f\n", vt);
    printf("u*s*v^t =\n%f\n", u*diagonalmatrix(s)*transpose(vt));

    status = svd(b, fs, fu, fvt);
    if(status ==0) {
        printf("fu =\n%f\n", fu);
        printf("fs =\n%f\n", fs);
        printf("fvt =\n%f\n", fvt);
        printf("u*s*v^t =\n%f\n", fu*diagonalmatrix(fs)*transpose(fvt));
    }
    else
        printf("error: numerical error in svd()\n");

}
```

Output1

```
s =
15.071167 5.471953 0.957939

u =
-0.660443 0.704551 0.259659
-0.511773 -0.169318 -0.842271
-0.549458 -0.689158 0.472395

s =
15.071167 5.471953 0.957939

vt =
-0.517995 0.430638 0.739075
-0.736603 0.214678 -0.641349
-0.434853 -0.876621 0.206007

u*s*v^t =
7.000000 8.000000 1.000000
3.000000 6.000000 4.000000
3.000000 5.000000 7.000000

fu =
-0.660443 0.704551 0.259659
-0.511773 -0.169318 -0.842271
-0.549458 -0.689158 0.472395

fs =
15.071167 5.471953 0.957939

fvt =
-0.517995 0.430638 0.739075
-0.736603 0.214678 -0.641349
-0.434853 -0.876621 0.206007

u*s*v^t =
7.000000 8.000000 1.000000
```

```
3.000000 6.000000 4.000000
3.000000 5.000000 7.000000
```

Example2
Singular value decomposition of a real m-by-n matrix ($m < n$).

```
#include <numeric.h>
int main() {
     array double a[2][3] = {7, 8, 1,
                             3, 6, 4}; /* m-by-n matrix */
     int m = 2, n = 3, i;
     int lmn = min(m,n);
     array double s[lmn], sm[m][n];
     array double u[m][m];
     array double vt[n][n];


     svd(a, s, u, vt);
     printf("u =\n%f\n", u);
     printf("s =\n%f\n", s);
     printf("vt =\n%f\n", vt);
     for(i=0; i<lmn; i++)
       sm[i][i] = s[i];
     printf("u*s*v^t =\n%f\n", u*sm*transpose(vt));
}
```

Output2

```
u =
-0.818910 -0.573922
-0.573922 0.818910

s =
12.851503 3.136698

vt =
-0.580020 -0.497570 0.644981
-0.777715 0.102681 -0.620174
-0.242353 0.861325 0.446525

u*s*v^t =
7.000000 8.000000 1.000000
3.000000 6.000000 4.000000
```

Example3
Singular value decomposition of a real m-by-n matrix ($m > n$).

```
#include <numeric.h>
int main() {
     array double a[3][2] = {7, 8,
                             1, 3,
                             6, 4}; /* m-by-n matrix */
     int m = 3, n = 2, i;
     int lmn = min(m,n);
     array double s[lmn], sm[m][n];
     array double u[m][m];
     array double vt[n][n];
```

```
     svd(a, s, u, vt);
     printf("u =\n%f\n", u);
     printf("s =\n%f\n", s);
     printf("vt =\n%f\n", vt);
     for(i=0; i<lmn; i++)
       sm[i][i] = s[i];
     printf("u*s*v^t =\n%f\n", u*sm*transpose(vt));
}
```

Output3

```
u =
-0.812719 -0.288579 -0.506171
-0.217573 -0.655581 0.723102
-0.540508 0.697808 0.470016

s =
13.058084 2.118123

vt =
-0.700689 0.713467
-0.713467 -0.700689

u*s*v^t =
7.000000 8.000000
1.000000 3.000000
6.000000 4.000000
```

Example4
Singular value decomposition of a real n-by-n matrix with complex number.

```
#include <numeric.h>
int main() {
     int m = 3 , n = 3;
     int lmn = min(m,n);
     array double complex a[3][3] = {7, 8, 1,
                                     3, 6, 4,
                                     3, 5, 7} ; /* m-by-n matrix */
     array complex z[3][3] = {complex(1,2), 8, 1,
                              3, 6, 4,
                              3, 5, 7} ; /* m-by-n matrix */
     array double s[lmn];
     array double complex u[m][m];
     array double complex vt[n][n];
     int status;

     svd(a, s, NULL, NULL);
     printf("s =\n%f\n", s);

     svd(a, s, u, vt);
     printf("u =\n%f\n", u);
     printf("s =\n%f\n", s);
     printf("vt =\n%f\n", vt);
     printf("u*s*v^t =\n%f\n", u*diagonalmatrix(s)*transpose(vt));

     status = svd(z, s, u, vt);
     if(status ==0) {
         printf("u =\n%f\n", u);
```

```
        printf("s =\n%f\n", s);
        printf("vt =\n%f\n", vt);
        printf("u*s*v^t =\n%f\n", u*diagonalmatrix(s)*transpose(vt));
    }
    else
        printf("error: numerical error in svd()\n");

}
```

Output4

```
s =
15.071167 5.471953 0.957939

u =
complex(-0.660443,0.000000) complex(0.704551,0.000000) complex(0.259659,0.000000)

complex(-0.511773,0.000000) complex(-0.169318,0.000000) complex(-0.842271,0.000000)

complex(-0.549458,0.000000) complex(-0.689158,0.000000) complex(0.472395,0.000000)


s =
15.071167 5.471953 0.957939

vt =
complex(-0.517995,-0.000000) complex(0.430638,-0.000000) complex(0.739075,0.000000)

complex(-0.736603,-0.000000) complex(0.214678,-0.000000) complex(-0.641349,-0.000000)

complex(-0.434853,-0.000000) complex(-0.876621,-0.000000) complex(0.206007,0.000000)


u*s*v^t =
complex(7.000000,0.000000) complex(8.000000,0.000000) complex(1.000000,0.000000)

complex(3.000000,0.000000) complex(6.000000,0.000000) complex(4.000000,0.000000)

complex(3.000000,0.000000) complex(5.000000,0.000000) complex(7.000000,0.000000)


u =
complex(-0.504577,-0.182672) complex(0.536523,-0.575026) complex(-0.247592,0.179497)

complex(-0.551746,-0.131934) complex(-0.031610,0.061460) complex(0.625115,-0.531624)

complex(-0.607942,-0.143559) complex(-0.372381,0.487898) complex(-0.372831,0.306730)


s =
13.678758 5.090474 0.989266

vt =
complex(-0.317937,-0.000000) complex(-0.358611,-0.000000) complex(0.877675,0.000000)

complex(-0.759338,-0.217182) complex(0.440159,-0.352023) complex(-0.095224,-0.222508)

complex(-0.509341,-0.125401) complex(-0.431508,0.606249) complex(-0.360819,0.202282)
```

```
u*s*v^t =
complex(1.000000,2.000000) complex(4.932809,1.184115) complex(3.850649,4.943457)

complex(3.000000,0.000000) complex(5.202331,3.116323) complex(3.380806,1.947931)

complex(3.000000,0.000000) complex(6.030656,5.110845) complex(3.373340,-0.361984)
```

Example5
Singular value decomposition of a real n-by-n singular matrix.

```
#include <numeric.h>
int main() {
    /* singular matrix */
     array float a[3][3] = {1, 2, 3,
                            4, 5, 6,
                            7, 8, 9};
     int m = 3, n = 3, i;
     int lmn = min(m,n);
     array double s[lmn], sm[m][n];
     array double u[m][m];
     array double vt[n][n];


     svd(a, s, u, vt);
     printf("u =\n%f\n", u);
     printf("s =\n%f\n", s);
     printf("vt =\n%f\n", vt);
     for(i=0; i<lmn; i++)
       sm[i][i] = s[i];
     printf("u*s*v^t =\n%f\n", u*sm*transpose(vt));
}
```

Output5

```
u =
-0.214837 0.887231 0.408248
-0.520587 0.249644 -0.816497
-0.826338 -0.387943 0.408248

s =
16.848103 1.068370 0.000000

vt =
-0.479671 -0.776691 -0.408248
-0.572368 -0.075686 0.816497
-0.665064 0.625318 -0.408248

u*s*v^t =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
```

See Also
condnum(), **rcondnum**().

References

E. Anderson, et al, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

trace

Synopsis
#include <numeric.h>
double trace(**array double** *a*[&][&]);

Purpose
Calculate the sum of diagonal elements of a matrix.

Return Value
This function returns the sum.

Parameters
a Input two-dimensional array.

Description
This function calculates the sum of the diagonal elements of the matrix.

Algorithm
For a matrix *a*, the trace is defined as

$$trace = \sum_{i=1}^{n} a_{ii}$$

Example

```
#include <numeric.h>
int main() {
    array double a[2][2] = {2, -4,
                            3, -7};
    array double a2[3][2] = {2, -4,
                             3, -7,
                             1, 1};
    array int b[2][2] = {2, -4,
                         3, -7};
    array int b2[3][2] = {2, -4,
                          3, -7,
                          1, 1};
    double t;

    t = trace(a);
    printf("trace(a) = %f\n", t);
    t = trace(a2);
    printf("trace(a2) = %f\n", t);
    t = trace(b);
    printf("trace(b) = %f\n", t);
    t = trace(b2);
    printf("trace(b2) = %f\n", t);
}
```

Output

```
trace(a) = -5.000000
trace(a2) = -5.000000
trace(b) = -5.000000
trace(b2) = -5.000000
```

See Also
ctrace(), **sum**(), **mean**(), **median**().

References

triangularmatrix

Synopsis
#include <numeric.h>
double triangularmatrix(**string_t** *pos*, **array double** *a*[&][&], ... /* [**int**k] */)[:][:];

Syntax
triangularmatrix(*pos, a*)
triangularmatrix(*pos, a, k*)

Purpose
Get the upper or lower triangular part of a matrix.

Return Value
This function returns the upper or lower triangular part of a matrix.

Parameters
pos Input string indicating which part of the matrix to be returned.

a Input 2-dimensional array.

k Input integer. A matrix is returned on and below the *k*th diagonal of the input matrix. By default, *k* is 0.

Description
This function gets a triangular matrix of matrix *a*. For *pos* of `"upper"`, the function returns the upper triangular part of matrix *a*, on and above the *k*th diagonal of the matrix. Optional argument *k* indicates the offset of the triangular matrix to the upper *k*th diagonal of the matrix. For *pos* of `"lower"` the function returns the lower part of matrix *a*, on and below the *k*th diagonal of the matrix. Optional argument *k* indicates the offset of the triangular matrix to the lower *k*th diagonal of the matrix.

Example

```
#include <numeric.h>
int main() {
    array double a[4][3] = {1,2,3,
                            4,5,6,
                            7,8,9,
                            6,3,5};
    int n = 4, m = 3, k;
    array double t[n][m];


    t = triangularmatrix("upper",a);
    printf("triangularmatrix(\"upper\", t) =\n%f\n", t);

    k = 0;
    t = triangularmatrix("upper",a,k);
    printf("triangularmatrix(\"upper\", t, 0) =\n%f\n", t);

    k = 1;
    t = triangularmatrix("upper",a,k);
    printf("triangularmatrix(\"upper\", t, 1) =\n%f\n", t);
```

```
    k = -1;
    t = triangularmatrix("upper",a,k);
    printf("triangularmatrix(\"upper\", t, -1) =\n%f\n", t);


    t = triangularmatrix("lower",a);
    printf("triangularmatrix(\"lower\", t) =\n%f\n", t);

    k = 0;
    t = triangularmatrix("lower",a,k);
    printf("triangularmatrix(\"lower\", t, 0) =\n%f\n", t);

    k = 1;
    t = triangularmatrix("lower",a, k);
    printf("triangularmatrix(\"lower\", t, 1) =\n%f\n", t);

    k = -1;
    t = triangularmatrix("lower",a,k);
    printf("triangularmatrix(\"lower\", t, -1) =\n%f\n", t);
}
```

Output

```
triangularmatrix("upper", t) =
1.000000 2.000000 3.000000
0.000000 5.000000 6.000000
0.000000 0.000000 9.000000
0.000000 0.000000 0.000000

triangularmatrix("upper", t, 0) =
1.000000 2.000000 3.000000
0.000000 5.000000 6.000000
0.000000 0.000000 9.000000
0.000000 0.000000 0.000000

triangularmatrix("upper", t, 1) =
0.000000 2.000000 3.000000
0.000000 0.000000 6.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000

triangularmatrix("upper", t, -1) =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
0.000000 8.000000 9.000000
0.000000 0.000000 5.000000

triangularmatrix("lower", t) =
1.000000 0.000000 0.000000
4.000000 5.000000 0.000000
7.000000 8.000000 9.000000
6.000000 3.000000 5.000000

triangularmatrix("lower", t, 0) =
1.000000 0.000000 0.000000
4.000000 5.000000 0.000000
7.000000 8.000000 9.000000
6.000000 3.000000 5.000000
```

```
triangularmatrix("lower", t, 1) =
1.000000 2.000000 0.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
6.000000 3.000000 5.000000

triangularmatrix("lower", t, -1) =
0.000000 0.000000 0.000000
4.000000 0.000000 0.000000
7.000000 8.000000 0.000000
6.000000 3.000000 5.000000
```

See Also
ctriangularmatrix(), **diagonal**().

unwrap

Synopsis
#include <numeric.h>
int unwrap(**array double** &*y*, **array double** &*x*, ... /* [**double** *cutoff*] */);

Syntax
unwrap(*y, x*)
unwrap(*y, x, cutoff*)

Purpose
Unwrap radian phase of each element of input array *x* by changing its absolute jump greater than π to its $2 * \pi$ complement.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
x A vector or two-dimensional array which contains the original radian phase data.

y An output array which is the same dimension and size as *x*. It contains the unwraped data.

cutoff An optional input data which specifies the jump value. If the user does not specify this input, *cutoff* has a value of π by default.

Description
The input array *x* can be of a vector or a two-dimensional array. If it is a two-dimensional array, the function unwraps it through every row of the array.

Example
In motion analysis of a crank-rocker mechanism, the output range of the rocker is within $0 \sim 2\pi$. For a given mechanism, the output may be as shown on the top part in Figure 11.4. There is a jump when θ_4 is π, because $\theta_4 = \pi$ and $\theta_4 = -\pi$ are the same point for the crank-rocker mechanism. If the **unwrap**() function is used, a smooth curve for output angle θ_4 can be obtained as shown on the lower part in Figure 11.4.

```
#include <numeric.h>
#include <complex.h>
#include <chplot.h>

int main(){
    double r[1:4],theta1,theta31;
    int n1=2,n2=4, i;
    double complex z,p,rb;
    double  x1,x2,x3,x4;
    array double theta2[36],theta4[36],theta41[36];
    class CPlot subplot, *plot;

    /* four-bar linkage*/
    r[1]=5; r[2]=1.5; r[3]=3.5; r[4]=4;
    theta1=30*M_PI/180;
    linspace(theta2,0,2*M_PI);
```

```
    for (i=0;i<36;i++) {
       z=polar(r[1],theta1)-polar(r[2],theta2[i]);
       complexsolve(n1,n2,r[3],-r[4],z,x1,x2,x3,x4);

       theta4[i] = x2;
    }
    unwrap(theta41, theta4);
    subplot.subplot(2,1);
    plot = subplot.getSubplot(0,0);
    plot->data2D(theta2, theta4);
    plot->title("Wraped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");

    plot = subplot.getSubplot(1,0);
    plot->data2D(theta2, theta41);
    plot->title("Unwraped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");
    subplot.plotting();
}
```

Output

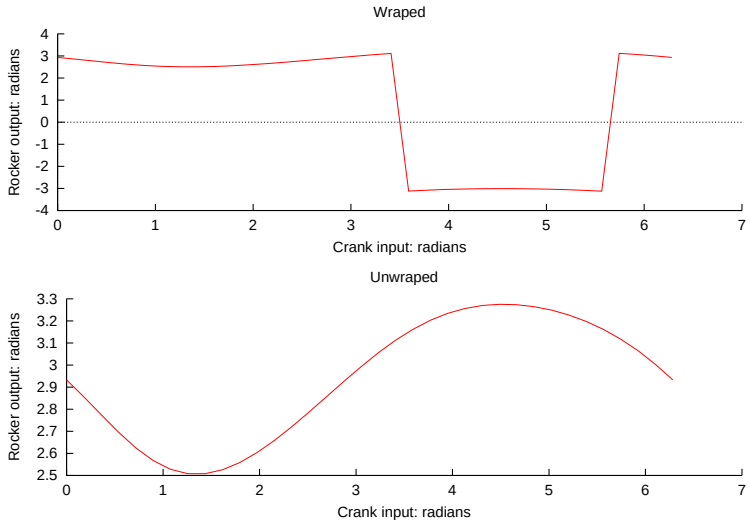


Figure 11.4: Comparison of results with and without using **unwrap**() function.

References

H. H. Cheng, *Computational Kinematics*, 1999.

urand

Synopsis
#include <**numeric.h**>
double urand(**array double** &*x*);

Syntax
urand(&*x*)
urand(*NULL*)

Purpose
Uniform random-number generator.

Return Value
This function returns a **double** uniform random-number.

Parameter
x An array of any dimensional size. It contains the uniform random numbers. The return value is the first element of array. If the argument of **urand**() is *NULL*, it only returns a uniform random number.

Description
Function urand() uses a random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to 1.

Example
The following example generates a two-dimensional array of uniform random number and a single uniform random number.

```
#include <stdio.h>
#include <numeric.h>

int main() {
   array double x[3][2];
   double u;

   u = urand(x);
   printf("urand(x) = %f\n", u);
   printf("x of urand(x) = \n%f\n", x);
   printf("urand(NULL) = %f\n", urand(NULL));
}
```

Output

```
urand(x) = 0.513871
x of urand(x) =
0.513871 0.175726
0.308634 0.534532
0.947630 0.171728

urand(NULL) = 0.702231
```

xcorr

Synopsis
#include <numeric.h>
int xcorr(**array double complex** *c*[&], **array double complex** *x*[&], **array double complex** *y*[&]);

Syntax
xcorr(*c*, *x*, *y*)

Purpose
One-dimensional cross-correlation function estimation.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
c A one-dimensional **array** of size $(n+m-1)$. It contains the result of a cross-correlation function estimate of *x* and *y*.

x A one-dimensional **array** of size n. It contains data used for correlation.

y A one-dimensional **array** of size m. It contains data used for correlation.

Description
The input **array** *x* and *y* can be of any supported arithmetic data type and size. Conversion of the data to **double complex** is performed internally. If both *x* and *y* are **real** data, the result is a one-dimensional **real array** *c* of size $(n+m-1)$. If either one of *x* or *y* is **complex** , then the result **array** *c* will be of **complex** data type.

Algorithm
Given two functions $x(t)$ and $y(t)$, and their corresponding Fourier transforms $X(f)$ and $Y(f)$, the cross correlation of these two functions, denoted as $xcorr(x, y)$, is defined by

$$xcorr(x,y) \equiv \int_{-\infty}^{\infty} x(t+\tau)y(t)d\tau$$

$$xcorr(x,y) \iff X(f)Y^*(f)$$

The asterisk denotes a complex conjugate. The correlation of a function with itself is called its autocorrelation. In this case

$$xcorr(x,x) \iff |X(f)^2|$$

In discrete correlation of two sampled sequences x_k and y_k, each periodic with period N, is defined by

$$xcorr(x,y)_j \equiv \sum_{k=0}^{N-1} x_{j+k}y_k$$

The discrete correlation theorem says that this discrete correlation of two real sequences x and y is one member of the discrete Fourier transform pair

$$xcorr(x,y)_j \iff X_k Y_k^*$$

where X_k and Y_k^* are discrete Fourier transforms of x_k and y_k, and the asterisk denotes complex conjugation. This theorem makes the same presumptions about the functions as those encountered for the discrete convolution theorem. The correlation can be computed using the DFT as follows: DFT the two data sequences, multiplying one resulting transform by the complex conjugate of the other, and inverse transforming the product, the result is the correlation. Just as in the case of convolution, time-aliasing is handled by padding with zeroes.
In the numerical implemenation, the sizes of two data sequences, **array** *x* of size n and *y* of size m, are expanded to $(m+n-1)$ and padded with zeroes internally. The FFT algorithm is used to compute the discrete Fourier transforms of *x* and *y* first. Multiplying the complex conjugate of transform of *y* with the transform of *x* component by component. Then using the inverse FFT algorithm to take the inverse transform of the products, the result is the cross correlation $xcorr(x,y)$ for arrays *x* and y.

Example
For two sequences *x*[m] and *y*[n], analytically, the cross correlation of *x* and *y* can be calculated by

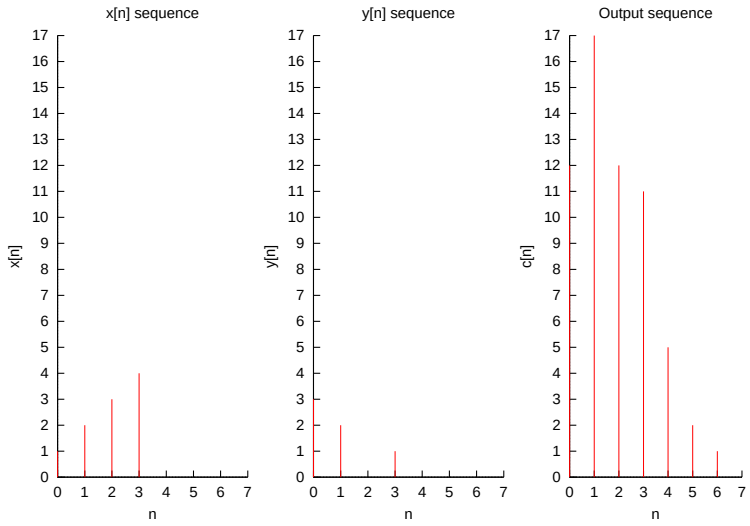
$$c[k] = \sum_{j=\max\{1,k-n+1\}}^{\min\{k,m\}} y[j]x[n+j-k]; \qquad k = 1,2,\cdots,n+m-1$$

Given $x = \{1,2,3,4\}$ and $y = \{3,2,0,1\}$, then

$$\begin{aligned}
c[1] &= y[1]x[4] = 12 \\
c[2] &= y[1]x[3] + y[2]x[4] = 17 \\
c[3] &= y[1]x[2] + y[2]x[3] + y[3]x[4] = 12 \\
c[4] &= y[1]x[1] + y[2]x[2] + y[3]x[3] + y[4]x[4] = 11 \\
c[5] &= y[2]x[1] + y[3]x[2] + y[4]x[3] = 5 \\
c[6] &= y[3]x[1] + y[4]x[2] = 2 \\
c[7] &= y[4]x[1] = 1
\end{aligned}$$

Output

```
x= 1.000  2.000  3.000  4.000

y= 3.000  2.000  0.000  1.000

c=12.000 17.000 12.000 11.000  5.000  2.000  1.000
```



See Also

fft(), **ifft**(), **conv**(), **conv**2(), **deconv**().

References

William H. Press, et al, *Numerical Recipes in C*, second edition, Cambridge University Press, 1997.

Chapter 12

Non-local jumps <setjmp.h>

The header **setjmp.h** defines the macro **setjmp**, and declares one function and one type, for bypassing the normal function call and return discipline.

The type declared is
jmp_buf
which is an array type suitable for holding the information needed to restore a calling enviornment.

The macro defined is
setjmp
which saves the current enviornment.

Functions

The following functions are defined by the **setjmp.h** header file.

Function	Description
longjmp	Restores the enviornment saved by the most recent invocation of the **setjmp** macro.

Macros

The following macros are defined by the **setjmp.h** header file.

Macro	Description
setjmp	Saves its calling environment in its **jmp_buf**argument.

Declared Types

The following functions are defined by the **setjmp.h** header file.

Type	Description
jmp_buf	array - holds information needed to restore a calling enviornment.

Portability
This header has no known portability problem.

setjmp

Synopsis
#include <setjmp.h>
int setjmp(**jmp_buf** *env*);

Purpose
Saves enviornment.

Return Value
If the return is from a direct invocation, the **setjmp** macro returns the value zero. If the return is from a call to the **longjmp** function, the **setjmp** macro returns a nonzero value.
An invocation of the **setjmp** macro shall appear only in one of the following contexts:

– The entire controlling expression of a selection or iteration statement

– The operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement.

– The operand of a unary ! operator with the resulting expression being the entire controlling expression of selection or iteration statement; or

– The entire expression of an expression statement (possibly cast to **void**).

Parameters
env Argument.

Description
The **setjmp** macro saves its calling enviornment in its **jmp_buf** argument for later use by the **longjmp** function.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
    void exit();

    if(setjmp(env) != 0) {
       int k;

       printf("value of i on 2nd return from setjmp: %d\n", i);
       i = 2;
       k = setjmp(env);
       printf("k = %d\n", k);
       if(k==0) {
         printf("reset setjmp() \n");
```

```
            restore_env();
          }
          else {
            printf("value of i on 3rd return from setjmp: %d\n", i);
            exit(0);
          }
        }
        (void) printf("value of i on 1st return from setjmp: %d\n", i);
        i = 1;
        restore_env();
        /* NOTREACHED */
    }
    int restore_env()
    {
        longjmp(env, 1);
        /* NOTREACHED */
        return 0;
    }
```

Output

```
value of i on 1st return from setjmp: 0
value of i on 2nd return from setjmp: 1
k = 0
reset setjmp()
k = 1
value of i on 3rd return from setjmp: 2
```

See Also
longjmp().

longjmp

Synopsis
#include <**setjmp.h**>
void longjmp(**jmp_buf** *env*, **int** *val*);

Purpose
Restores enviornment.

Return Value
After **longjmp** is completed, program execution continues as if the corresponding invocation of the **setjmp** macro had just returned the value specified by *val*. The **longjmp** function cannot cause the **setjmp** macro to return the value 0; if *val* is 0, the **setjmp** macro returns the value 1.

Parameters
env Argument.

val Argument.

Description
The **longjmp** function restores the enviornment saved by the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument. If there has been no such invocation, or if the function containing the invocation of the **setjmp** macros has terminated execution in the interim, or if the invocation of the **setjmp** macro was within the scope of an indentfier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

All accessible objects have values as of the time **longjmp** was called, except that the values of objects of automatic storage duration that are local to the function containg the invocation of the corresponding **setjmp** macro that do not have volatile-qualified type and have been changed between the **setjmp** invocation and the **longjmp** call are indeterminate.

Example
see **setjmp**().

See Also
setjmp().

Chapter 13

Signal handling <signal.h>

The header **signal.h** declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

The type defined is

sig_atomic_t

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity even in the presence of asynchronous interrupts.

The macros defined are

SIG_DFL
SIG_ERR
SIG_IGN

which expand to constant expressions with distinct values that are type compatible with the second argument to, and the return value of, the **signal** function, and whose values compare unequal to the address of any declarable function; and each of the following, which expand to positive integer constant expressions with type **int** and distinct values that are the signal numbers, correspond to the specified condition:

SIGABRT abnormal termination, such as is initiated by the abort function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

Ch may not generate any of these signals, except as a result of explicit calls to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter may also be specified by the different platform. implementation. The complete set of signals, their semantics, and their default handling is platform-dependent; all signal numbers shall be positive.

Functions

The following functions are defined by the **signal**.h header file.

Function	Description
signal()	Determines in which way the signal number sig will be handled.

Macros

The following macros and several other macros starting with **SIG** are defined by the **signal**.h header file.

Macro	Description
SIG_DFL	Specify use of default signal handling.
SIG_ERR	Return value of function **signal**() when it has an error.
SIG_IGN	Ignore the signal.

Declared Types

The following types are defined by the **signal**.h header file.

Type	Description
sig_atomic_t	int - object that can be accessed as an atomic entitym.

Portability

This header has no known portability probelm.

signal

Synopsis
#include <signal.h>
void (*__signal__(**int** *sig*, **void** (∗ *func*)(**int**)))(**int**);

Purpose
Chooses which way to receive a signal.

Return Value
If the request can be honored, the **signal** function returns the value of *func* for the most recent successful call to **signal** for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Parameters
sig Signal number.

func signal handler.

Description
The signal function chooses one of three ways in which receipt of the signal number sig is to be subsequently handled. If the value of *func* is **SIG_DFL** default handling for that signal will occur. If the value of *func* is **SIG_IGN**, the signal will be ignored. Otherwise, *func* shall point to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library), is called a *signal handler*.

When a signal occurs and *func* points to a function, it is implementation-defined whether the equivalent of **signal**(*sig*, **SIG_DFL**); is executed or the implementation prevents some implementation-defined set of signals (at least including *sig*) from occurring until the current signal handling has completed; in the case of **SIGILL**, the implementation may alternatively define that no action is taken. Then the equivalent of (*__func__*)(sig); is executed. If and when the function returns, if the value of *sig* is **SIGFPE SIGILL SIGSEGV** or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

If the signal occurs as the result of calling the **abort** or **raise** function, the signal handler shall not call the **raise** function.

If the signal occurs other than as the result of calling the **abort** or **raise** function, the behavior is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as **volatile sig_atomic_t**, or the signal handler calls any function in the standard library other than the **abort** function or the **signal** function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the **signal** function results in a **SIG_ERR** return, the value of **errno** is indeterminate.

At program startup, the equivalent of
signal(sig, SIG_IGN);

may be executed for some signals selected in an implementation-defined manner; the equivalent of **signal(sig, SIG_DFL);**

is executed for all other signals defined by the implementation.

The implementation shall behave as if no library function calls the **signal** function.

Example

```
/* sample program to show how signal() is used */
#include <stdio.h>
#include <signal.h>
int delay = 1;
void childHandler();

int main()
{
   int pid;
   char *argv[4];

   argv[0]=strdup("ls");
   argv[1]=strdup("-s");
   argv[2]=strdup("signal.c");
   argv[3]=NULL;
   signal(SIGCHLD, childHandler);
   pid=fork();
   if(pid==0)
   {
      execvp("ls",argv);
      perror("limit");
   }
   else
   {
      sleep(delay);
      printf("Child %d exceeded limit and is being killed \n",pid);
      kill(pid, SIGINT);
   }
}

void childHandler()
{
   int childPid, childStatus;
   childPid=wait(&childStatus);
   printf("Child %d terminated within %d seconds \n", childPid, delay);
   exit(0);
}
```

Output

```
   2 signal.c
Child 9211 terminated within 1 seconds
```

See Also
raise().

raise

Synopsis
#include <**signal.h**>
int raise(**int** *sig*);

Purpose
Sends the signal specified by the *sig* to the executing program.

Return Value
The **raise** function returns zero if successful, nonzero if unsuccessful.

Parameters
sig Signal number.

Description
The **raise** function carries out the actions described by the signal function. If a signal handler is called, the **raise** function shall not return until after the signal handler does.

Example

```
#include<stdio.h>
#include<signal.h>

void func(int sig) {
   printf("sig in func() = %d\n", sig);
}

int main() {
  void (*signal (int sig, void (*disp)(int)))(int);
  void (*fptr)(int);

  fptr = signal(SIGTERM, func);
  printf("fptr = %p\n", fptr);
  raise(SIGTERM);
}
```

Output

```
fptr = 0
sig in func() = 15
```

See Also
signal().

Chapter 14

Variable Argument Lists — <stdarg.h>

The header **stdarg.h** declares a type and defines eight macros, for advancing through a list of arguments whose number and types are not known to the called function when it is parsed.

A function may be called with a variable number of arguments of varying types. Its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

The declared type **va_list** is an object type suitable for holding information needed by the macros **va_start**, **va_arg**, **va_end**, **va_copy**, **va_count**, **va_dim**, **va_elementtype**, and **va_extent**. If access to the varying arguments is desired, the called function shall declare an object (referred to as *ap* in this subclause) having type **va_list**. The object *ap* may be passed as an argument to another function; if that function invokes the **va_arg** macro with parameter ap, the value of *ap* in the calling function is indeterminate and shall be passed to the **va_end** macro prior to any further reference to *ap*.

Variable argument list access macros
The **va_start**, **va_arg**, and **va_copy** macros described in this subclause shall be implemented as macros, not functions. The **va_end** is an identifier declared with external linkage. If a program defines an external identifier with the name **va_end**, the behavior is undefined. Each invocation of the **va_start** or **va_copy** macros shall be matched by a corresponding invocation of the **va_end** macro in the function accepting a varying number of arguments.

Macros
The following macros are defined by the **stdarg.h** header file.

Macro	Description
VA_NOARG	Second argument for **va_start**(), if no argument is passed to function
va_arg	Expands to an expression that has the specified type and the value of the next argument in the calling function.
va_copy	Makes a copy of the **va_list**.
va_count	Obtain the number of variable arguments.
va_dim	Obtain the dimension of an array of variable arguments.
va_elementtype	Obtain the data type of variable arguments.
va_end	Facilitates a normal return from the function.

va_extent	Obtain the number of elements of the array of variable argument.
va_start	Initializes *ap* for subsequent use by **va_arg** and **va_end**.

Declared Types

The following types are defined by the **stdarg.h** header file.

Type	Description
va_list	object - holds information needed by **va_start**, **va_arg**, **va_end**, **va_copy**, **va_count**, **va_dim**, **va_elementtype** and **va_extent**.

Portability

This header has no known portability problem.

va_arg

Synopsis
#include <**stdarg.h**>
type **va_arg**(**va_list** *ap*, *type*);

Purpose
Expand a value in the variable argument list.

Return Value
The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

Parameters
ap An object having type **va_list**.

type The data type of passed variable argument.

Description
The **va_arg** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter *ap* shall be the same as the **va_list** *ap* initialized by **va_start**. Each invocation of **va_arg** modifies *ap* so that the values of successive arguments are returned in turn. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a ***** to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.
Example
See **va_start**().

See Also
va_start(), **va_end**().

va_copy

Synopsis
#include <**stdarg.h**>
void va_copy(**va_list** *dest*, **va_list** *src*);

Purpose
Makes **va_list** *src* a copy of **va_list** *dest*.

Return Value
The **va_copy** macro returns no value.

Parameters
dest A pointer to destination object.

src A pointer to source object.

Description
The **va_copy** macro makes the **va_list** *dest* be a copy of the **va_list** *src*, as if the **va_start** macro had been applied to it followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of *src*.

Example

```
#include <stdarg.h>

void fun (int i, ...) {
    int i, j, k;;
    float f;
    va_list ap, ap2;

    va_start(ap, i);
    j = va_arg(ap, int);
    printf("j = %d\n", j);
    va_copy(ap2, ap);

    k = va_arg(ap, int);
    printf("k = %d\n", k);
    f = va_arg(ap, float);
    printf("f = %f\n", f);
    va_end(ap);

    /* use ap2 copied from ap */
    k = va_arg(ap2, int);
    printf("k = %d\n", k);
    f = va_arg(ap2, float);
    printf("f = %f\n", f);
    va_end(ap2);
}

int main() {
   fun(1, 2, 3, 4.0);
}
```

Output

See Also
va_arg(), **va_start**(), **va_end**().

va_count

Synopsis
#include <**stdarg.h**>
int va_count(**va_list** *ap*);

Purpose
Obtain the number of variable arguments.

Return Value
The **va_count** macro returns the number of variable arguments.

Parameters
ap An object having type **va_list**.

Description
The **va_count** macro returns the number of variable arguments.

Example
See **va_start**().

See Also
va_dim(), **va_extent**(), **va_elementtype**().

va_dim

Synopsis
#include <**stdarg.h**>
int va_dim(**va_list** *ap*);

Purpose
Obtain the dimension of an array which is a variable argument.

Return Value
The **va_dim** macro returns the dimensions of the array.

Parameters
ap An object having type **va_list**.

Description
The **va_dim** macro returns the dimensions of an array which is a variable argument.

Example
See **va_start**().

See Also
va_count(), **va_elementtype**(), **va_extent**().

va_elementtype

Synopsis
#include <**stdarg.h**>
int va_elementtype(**va_list** *ap*);

Purpose
Obtain the data type of a variable argument.

Return Value
The **va_elementtype** macro returns the data type of a variable argument.

Parameters
ap An object having type **va_list**.

Description
The **va_elementtype** macro returns the data type of a variable argument.

Example
See **va_start**().

See Also
va_count(), **va_dim**(), **va_extent**().

va_end

Synopsis
#include <**stdarg.h**>
void va_end(**va_list** *ap*);

Purpose
Creates a normal return from a function.

Return Value
The **va_end** macro returns no value.

Parameters
ap An object having type **va_list**.

Description
The **va_end** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of **va_start** that initialized the **va_list** *ap*. The **va_end** macro may modify *ap* so that it is no longer usable (without an intervening invocation of **va_start**). If there is no corresponding invocation of the **va_start** macro, or if the **va_end** macro is not invoked before the return, the behavior is undefined.

Example
See **va_start**().

See Also
va_start(), **va_arg**().

va_extent

Synopsis
#include <**stdarg.h**>
int va_extent(**va_list** *ap*, **int** *i*);

Purpose
Obtain the number of elements in the specified dimension of a variable argument.

Return Value
The **va_extent** macro returns the number of elements in the specified dimension of a variable argument.

Parameters
ap An object having type **va_list**.

i An integer specifying in which dimension the number of elements will be obtained.

Description
The **va_extent** macro returns the number of elements in the specified dimension of a variable argument.

Example
See **va_start**().

See Also
va_count(), **va_dim**(), **va_elementtype**().

va_start

Synopsis
#include <**stdarg.h**>
void va_start(**va_list** *ap*, *parmN*);

Purpose
Initializes *ap*.

Return Value
The **va_start** macro returns no value.

Parameters
ap An object having type **va_list**.

parmN The identifier of the rightmost parameter in the variable parameter list in the function definition.

Description
The **va_start** macro shall be invoked before any access to the unnamed arguments.

The **va_start** macro initializes *ap* for subsequent use by **va_arg** and **va_end**. **va_start**() shall not be invoked again for the same *ap* without an intervening invocation of **va_end**() for the same *ap*.

The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). If the parameter *parmN* can be declared with the **register** storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions. In case there is no parameter in the function, macro VA_NOARG shall be used.

Example

```
#include <stdarg.h>

void fun (int k, ...) {
    int i, j, vacount;
    va_list ap;

    va_start(ap, k);
    vacount = va_count(ap);
    printf("vacount = %d\n", vacount);
    printf("va_dim(ap)= %d\n", va_dim(ap));
    i = va_extent(ap, 0);
    printf("va_extent(ap, 0)= %d\n", i);
    if(va_dim(ap) > 1) {
      j = va_extent(ap, 1);
      printf("va_extent(ap, 1)= %d\n", j);
    }
    if(elementtype(int) == va_elementtype(ap))
      printf("array element is int\n");
    else if(elementtype(double) == va_elementtype(ap))
      printf("array element is double\n");
```

```
    else if(elementtype(int *) == va_elementtype(ap))
      printf("array element is pointer to int\n");
    va_end(ap);
}

int main() {
   int i, a[4], *p;
   double b[5][6];

   p = &i;
   fun(i, a);
   fun(i, b,a);
   fun(i, p);
}
```

Output

```
vacount = 1
va_dim(ap)= 1
va_extent(ap, 0)= 4
array element is int
vacount = 2
va_dim(ap)= 2
va_extent(ap, 0)= 5
va_extent(ap, 1)= 6
array element is double
vacount = 1
va_dim(ap)= 0
va_extent(ap, 0)= 0
array element is pointer to int
```

See Also

va_arg(), **va_end**().

Chapter 15

Boolean type and values <stdbool.h>

The header **stdbool.h** defines four macros.

The macro
bool
expands to **_Bool**.

The remaining three macros are suitable for use in **#if** preprocessing directives. They are
true
which expands to the integer constant 1,
false
which expands to the integer constant 0, and
__bool_true_false_are_defined
which expands to the decimal constant 1.

Notwithstanding the reserved indentifiers, a program is permitted to undefine and perhaps then redefine the macros **bool, true,** and **false**.

Macros

The following macros are defined by the **stdbool.h** header file.

Macro	Description
bool	Expands to **_Bool**.
true	Expands to integer constant 1.
false	Expands to integer constant 0.
__bool_true_false_are_defined	Expands to decimal point constant 1.

Chapter 16

Common Definitions <stddef.h>

The following types and macros are defined in the standard header **stddef.h**. Some are also defined in other headers, as noted in their respective subclauses.

The types are

ptrdiff_t

which is the signed integer type of the result of subtracting two pointers;

size_t

which is the unsigned integer type of the result of the **sizeof** operator; and

wchar_t

which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant.

The macros are

offsetof(*type, member-designator***)**

which expands to an integer constant expression that has type **size_t**, the value of which is the offset in bytes, to the structure member (designated by member-designator), from the beginning of its structure (designated by type). The type and member designator shall be such that given

static *type* **t**;

then the expression **&(t.member-designator)** evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

Macros

The following macros are defined by the **stddef.h** header file.

Macro	Description
offsetof	Expands to integer constant..

Declared Types

The following types are defined by the **stddefh** header file.

Type	Description
ptrdiff_t	int - results from subtracting two pointers.
size_t	unsigned int - results from the **sizeof** operator.
wchar_t	int - range of values can represent distinct codes for all members of the largest extended character set specified by the supported locales.

Portability

This header has no known portability problem.

Chapter 17

Input/Output <stdio.h>

The header **stdio.h** declares three types, several macros, and many functions for performing input and output.

The types declared are **size_t** (described by **stddef.h**);

FILE

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

fpos_t

which is an object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

The macros are

_IOFBF
_IOLBF
_IONBF

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **setvbuf** function;

BUFSIZ

which expands to an integer constant expression, which is the size of the buffer used by the setbuf function;

EOF

which expands to an integer constant expression, with type **int** and a negative value, that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream;

FOPEN_MAX

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

FILENAME_MAX

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation guarantees can be opened;

L_tmpnam

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam** function;

SEEK_CUR
SEEK_END

SEEK_SET
which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;
TMP_MAX
which expands to an integer constant expression that is the minimum number of unique file names that can be generated by the **tmpnam** function;
stderr
stdin
stdout
which are expressions of type "pointer to **FILE**" that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

The header **wchar.h** declares a number of functions useful for wide-character input and output. The wide-character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of "generalized" multibyte characters.

The input/output functions are given the following collective terms:

– The *wide-character input functions* – those functions described that perform input into wide characters and wide strings: **fgetwc, fgetws, getwc, getwchar, fwscanf, wscanf, vfwscanf,** and **vwscanf**.

– The *wide-character output functions* – those functions that perform output from wide characters and wide strings: **fputwc, fputws, putwc, putwchar, fwprintf, wprintf, vfwprintf**, and **vwprintf**.

– The *wide-character input/output functions* – the union of the **ungetwc** function, the wide- character input functions, and the wide-character output functions.

– The *byte input/output functions* – those functions described in this subclause that perform input/output: **fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, getc, getchar, gets, printf, putc, putchar, puts, scanf, ungetc, vfprintf, vfscanf, vprintf**, and **vscanf**.

Streams
Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.

A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is platform-dependent. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is platform-dependent.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have a platform-dependent number of null characters appended to the end of the stream.

Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide-character input/output function has been applied to a stream without orientation, the stream becomes a *wide-oriented stream.* Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)

Byte input/output functions shall not be applied to a wide-oriented stream and wide-character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream's orientation, except for the following additional restrictions:

– Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

– For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide-character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

Environmental limits
Each platform supports text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Files
A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is platform-depedent whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

Binary files are not truncated, except as defined later. Whether a write on a text stream causes the associated file to be truncated beyond that point is platform-dependent.
When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is

filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is platform-dependent,and may be affected via the **setbuf** and **setvbuf** functions.

A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is platform-dependent.

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **main** function returns to its original caller, or if the **exit** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **abort** function, need not close all files properly.

The address of the FILE object used to control a stream may be significant; a copy of a FILE object need not serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly – *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are platform-dependent. Whether the same file can be simultaneously open multiple times is also platform-dependent.

Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

– Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

– A file need not begin nor end in the initial shift state.

Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are platform-dependent.

The wide-character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's own **mbstate_t** object. The byte input functions read characters from the stream as if by successive calls to the **fgetc** function.

The wide-character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the **fputwc** function. Each conversion occurs as if by a call to the **wcrtomb** function, with the conversion state described by the stream's own **mbstate_t** object.

The byte output functions write characters to the stream as if by successive calls to the **fputc** function.

In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the **mbrtowc** and **wcrtomb** functions.

An *encoding error* occurs if the character sequence presented to the underlying **mbrtowc** function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying **wcrtomb** does not correspond to a valid (generalized) multibyte character. The wide- character input/output functions and the byte input/output functions store the value of the macro **EILSEQ** in **errno** if and only if an encoding error occurs.

Enviornmental limits
The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Functions

The following functions are defined by the **stdio.h** header file.

Function	Description
clearerr	Clears the end-of-file and error indicators from a stream.
fclose	Flushes a stream and closes the associated file(s).
feof	Tests the end-of-file for a stream.
ferror	Tests the error indicator for a stream.
fflush	Flushes a file.
fgetc	Reads the next character from the input stream.
fgetpos	Stores values of file position indicators.
fgets	Reads `n-1` characters from the input stream ending with a carriage return character.
fopen	Opens a file.
fprintf	Prints output to given stream.
fputc	Writes a character to the output stream.
fputs	Writes a string to the output stream.
fread	Reads from the stream into an array.
freopen	Opens a file.
fscanf	Reads input from a given stream.
fseek	Sets file postion indicator for a string.
fsetpos	Sets the **mbstate_t** object and the file position indicator for a stream.
ftell	Obtains the current value of the file position indicator for a stream.
fwrite	Writes from an array into a stream.
getc	Reads a character from the input stream.
getchar	Reads next character from string.
getline()	Get an input line ending with a carriage return character.
getline	Reads a line from an input stream.
gets	Reads a character from the input stream.
perror	Maps the error number in **errno** to an error message.
printf	Prints output to the standard output.
putc	Writes a character to the output stream.

putchar	Writes a character to **stdout**.
puts	Writes a character to a string.
remove	Removes a file.
rename	Renames a file.
rewind	Sets the file position indicator to tthe beginning of a stream.
scanf	Reads input from the standard input.
setbuf	Sets buffer of a stream.
setvbuf	Sets buffer mode and size of a stream.
snprintf	Prints output to a specified arrary.
sprintf	Prints output to a specified arrary.
sscanf	Reads input from a string.
tmpfile	Creates a temporary binary file.
tmpnam	Creates a temporary file name.
ungetc	Pushes character back onto input stream.
vfprintf	Prints output to specified stream.
vprintf	Prints output to specified stream.
vsnprintf	Prints output to specified array.
vsprintf	Prints output to specified array.

Macros

The following macros are defined by the **stdio.h** header file.

Macro	Description
_IOFBF	Expands to an integer constant expression suitable for third argument of **setvbuf**.
_IOLBF	Expands to an integer constant expression suitable for third argument of **setvbuf**.
_IONBF	Expands to an integer constant expression suitable for third argument of **setvbuf**.
BUFSIZ	Expands to an integer constant expression, which is the size of the buffer used by **setbuf**.
EOF	Expands to an an integer constant expression, with type **int** and a negative value, that is returned by several functions to indicate *end-of-file*.
FILENAME_MAX	Expands to an integer constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation guarantees can be opened.
FOPEN_MAX	Expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously.
L_tmpnam	Expands to an integer constant expression large enough to hold the temporary file name string generated by **tmpnam**.
SEEK_CUR	Expands to an integer constant expression with distinct value suitable for use in **fseek**.
SEEK_END	Expands to an integer constant expression with distinct

	value suitable for use in **fseek**.
SEEK_SET	Expands to an integer constant expression with distinct value suitable for use in **fseek**.
TMP_MAX	Expands to an integer constant expression that is the minimum number of file names that can be generated by **tmpnam**.
stderr	Points to file object associated with the standard error.
stdin	Points to file object associated with the standard input.
stdout	Points to file object associated with the standard output.

Declared Types

The following types are defined by the **stdio.h** header file.

Type	Description
size_t	unsigned int - results from the **sizeof** operator.
FILE	object - records all information needed to control a stream.
fpos_t	object - records all information needed to specify uniquely every position within a file.

Portability

The following functions are not supported in Ch: **vsscanf**(), **vfscanf**(), **vscanf**().

clearerr

Synopsis
#include <**stdio.h**>
void clearerr(**FILE** **stream*);

Purpose
Clear the end-of-file and error indicators from a stream.

Return Value
The **clearerr** function returns no value.

Parameters
stream Pointer to a stream.

Description
The **clearerr** function clears the end-of-file and error indicators for the stream pointed to by *stream*.

Example

```
/* clearerr() resets  the error indicator and  EOF
   to 0 on the same stream */
#include <stdio.h>

int main() {
    FILE *fp;
    char ch;

    if((fp = fopen("clearerr.c","r")) == NULL) {
        fprintf(stderr, "Error: cannot open file clearerr.c for reading\n");
        exit(1);
    }
    while(!feof(fp)) {
        ch = fgetc(fp);
    }
    printf("feof(fp) is %2d before clearerr(fp) is called.\n", feof(fp));
    clearerr(fp);
    printf("feof(fp) is %2d after clearerr(fp) is called.\n", feof(fp));
 }
```

Output

```
feof(fp) is 16 before clearerr(fp) is called.
feof(fp) is  0 after clearerr(fp) is called.
```

See Also

fclose

Synopsis
#include <stdio.h>
int fclose(**FILE** **stream*);

Purpose
Flushes a stream and closes the associated file(s).

Return Value
The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

Parameters
stream Pointer to a stream.

Description
The **fclose** function causes the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host enviornment to be written to the file; and unread buffered data are discarded. The stream is disassociated from the file. If the asociated buffer was automatically allocated, it is deallocated.

Example see **clearerr**().
Output
See Also

feof

Synopsis
#include <stdio.h>
int feof(**FILE** **stream*);

Purpose
Test the end-of-file for a stream.

Return Value
The **feof** function returns nonzero if and only if the end-of-file indicator is set for *stream*.

Parameters
stream Pointer to a stream.

Description
The **feof** function tests the end-of-file indicator for the stream pointed to by *stream*.

Example see **clearerr**().
Output
See Also

ferror

Synopsis
#include <stdio.h>
int ferror(**FILE** **stream*);

Purpose
Test the error indicator for a stream.

Return Value
The **ferror** function returns nonzero if and only if the end-of-file indicator is set for *stream*.

Parameters
stream Pointer to a stream.

Description
The **ferror** function tests the error indicator for the stream pointed to by *stream*.

Example see **clearerr**().
Output
See Also

fflush

Synopsis
#include <stdio.h>
int fflush(**FILE** **stream*);

Purpose
Flush a file.

Return Value
The **fflush** function sets the error indicator for the stream and returns **EOF** if a write error occurs, otherwise, it returns zero.

Parameters
stream Pointer to a stream.

Description
If *stream* points to an output stream or an update stream in which the most recent operation was no input, the **fflush** function causes any unwrittern data for that stream to be delievered to the host enviornment to be written to the file; otherwise, the behavior is undefined.

If *stream* is a null pointer, the **fflush** function performs this flushing action on all streams for which the behavior is defined above.

Example
see **fwrite**().
Output
See Also

fgetc

Synopsis
#include <stdio.h>
int fgetc(**FILE** **stream*);

Purpose
Read input from stream.

Return Value
If the end-of-file indicator is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetc** returns **EOF**. Otherwise, the **fgetc** function returns the next character from the input stream pointed to by *stream*. If a read error occurs, the error indicator for the stream is set and **fgetc** returns **EOF**.

Parameters
stream Pointer to a stream.

Description
If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next character is present, the **fgetc** function obtains that character as an **unsigned char** converted to an **int** and advances the associated file position indicator for the stream (if defined).

Example

```
/* a sample program that reads a file then
displays it's content. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch;

    /* open file for reading only */
    if((stream=fopen("fgetc.c","r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    ch = fgetc(stream);
    printf("%c\n",ch);
    fclose(stream);
}
```

Output

```
/
```

See Also

fgetpos

Synopsis
#include <stdio.h>
int fgetpos(**FILE** * *restrict stream*, **fpos_t** * *resitrict pos*);

Purpose
Store current values of file position indicators.

Return Value
If successful, the **fgetpos** function returns 0; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Parameters
stream Pointer to a stream.

pos Pointer to an object.

Description
The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The values stored contain unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Example

```
/* a sample program that uses the fgetpos() to read the fourth
   character from a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    fpos_t pos;
    char ch;
    long int offset = 3;
    /* open file for reading */
    if((stream = fopen("fgetpos.c","r")) == NULL){
        printf("cannot open file\n");
        exit(1);
    }
    fseek(stream, offset ,SEEK_SET);
    fgetpos(stream,&pos);
    fsetpos(stream,&pos);
    fread(&ch,sizeof(char),1,stream);   //read a character
    printf("The current character in file is '%c'\n",ch);
    fclose(stream);
}
```

Output

```
The current character in file is 'a'
```

See Also

fgets

Synopsis
#include <stdio.h>
int *__**fgets**__(__**char** * *restrict s*, **int** *n*, **FILE** **restrict stream*__);

Purpose
Read input from a stream into the memory.

Return Value
The **fgets** function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during operation, the array contents are indeterminate and a null pointer is returned.

Parameters
s Pointer to an array.

n Pointer to an integer.

stream Pointer to a stream.

Description
The **fgets** function reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Example
Output
See Also

fopen

Synopsis
#include <**stdio.h**>
FILE ***fopen**(**const char** **filename*, **const char** **mode*);

Purpose
Open a file.

Return Value
The **fopen** function returns a pointer to the object controlling the stream. If the open operation fails, **fopen** retruns a null pointer.

Parameters
filename Pointer to a file.

mode Points to a string.

Description
The **fopen** function opens the file whose name is the string pointed to by *filename*, and associates a stream with it.

The argument *mode* points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.

r	open a test file for reading
w	truncate to zero length or create a test file for writing
a	append; open or create a text file for writing at end-of-file
rb	open a binary file for reading
wb	truncate to zero length or create a binary file for writing
ab	append; open or create a binary file for writing at end-of-file
r+	open a text file for update (reading and writing)
w+	truncate to zero length or create a text file for update
a+	append; open or create a text file for update, writing at-end of-file
r+b *or* **rb+**	open a binary file for update (reading and writing)
w+b *or* **wb+**	truncate to zero length or create a binary file for updating
a+b *or* **ab+**	append; open or create a binary file for update, writing at end-of-file

Opening a file with read mode (**'r'** as the first character in the mode argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (**'a'** as the first character in the *mode* argument) causes all subsequent writes to the file to be foreced to the then current end-of-file, regardless of intervening calls to the **fseek** function. In some implementations, opening a binary file with the append mode (**'b'** as the second or third character in the above list of *mode* argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

When a file is opened with update mode (**'+'** as the second or third character in the above list of *mode* argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **fflush** function or to a file positioning function (**fseek**, **fsetpos**, or **rewind**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Example

see **clearerr**().

Output

See Also

fprintf

Synopsis
#include <stdio.h>
int fprintf(**FILE** **restrict stream*, **const char** **restrict format*, ...);

Purpose
Write output to a stream using the given format.

Return Value
The **fprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Parameters
stream Pointer to a stream.

format Points to a character.

Description
The **fprintf** function writes output to the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

– Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

– An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (described later) or a decimal integer.

– An optional precision that gives the minimum number of digits to appear for the **d, i, o, u, x**, and **X** conversions, the number of digits to appear after the decimal-point character for **a, A, e, E, f**, and **F** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in **s** conversions. The precision takes the form of a period (.) followed either by an asterisk (described later) or by an optional decimal integer; if only

the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

– An optional length modifier that specifies the size of the argument.

– A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified).
+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified).
space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and + flags both appear, the *space* flag is ignored.
The result is converted to an "alternative form". For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a, A, e, E, f, F, g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.
0 For **d, i, o, u, x, X, a, A, e, E, f, F, g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d, i, o, u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

hh Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following **n** conversion specifier applies to a pointer to a **signed char** argument.
h Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing); or that a following **n** conversion specifier applies to a pointer to a **short int** argument.

l (ell) Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **n** conversion specifier applies to a pointer to a **long int** argument; that a following **c** conversion specifier applies to a **wint_t** argument; that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following **a, A, e, E, f, F, g**, or **G** conversion specifier.

ll (ell-ell) Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following **n** conversion specifier applies to a pointer to a **long long int** argument.

j Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer to an **intmax_t** argument.

z Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a **size_t** argument.

t Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **ptrdiff_t** or the corresponding unsigned integer type argument; or that a following **n** conversion specifier applies to a pointer to a **ptrdiff_t** argument.

L Specifies that a following **a, A, e, E, f, F, g**, or **G** conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d,i The **int** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o,u,x,X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X)** in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f,F A **double** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **double** argument representing an infinity is converted in one of the styles *[-]***inf** or *[-]***infinity** – which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[-]***nan** or *[-]***nan***(n-char-sequence)* – which style, and the meaning of any n-char- sequence, is implementation-defined. The **F** conversion specifier produces **INF, INFINITY**, or **NAN** instead of **inf, infinity**, or **nan**, respectively.

e,E A double argument representing a floating-point number is converted in the style *[-]d.ddd***e**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the # flag is specified; a decimal-point character appears only if it is followed by a digit.

A double argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

a,A A **double** argument representing a floating-point number is converted in the style *[-]***0x***h.hhhh***p**±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating- point number and is otherwise unspecified) before the decimal-point character221) and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the # flag is not specified, no decimal- point character appears. The letters **abcdef** are used for a conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

c If no **l** length modifier is present, the **int** argument is converted to an **unsigned char**, and the resulting character is written.

If an **l** length modifier is present, the **wint_t** argument is converted as if by an ls conversion specification with no precision and an argument that points to the initial element of a two-element array of **wchar_t**, the first element containing the **wint_t** argument to the **lc** conversion specification and the second a null wide character.

s If no **l** length modifier is present, the argument shall be a pointer to the initial element of an array of character type.223) Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.
If an **l** length modifier is present, the argument shall be a pointer to the initial element of an array of **wchar_t** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A % character is written. No argument is converted. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding coversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

If **FLT_RADIX** is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For **e, E, f, F, g**, and **G** conversions, if the number of significant decimal digits is at most **DECIMAL_DIG**, then the result should be correctly rounded. If the number of significant decimal digits is more than **DECIMAL_DIG** but the source value is exactly representable with **DECIMAL_DIG** digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Example
Output

See Also

fputc

Synopsis
#include <**stdio.h**>
int ***fputc**(**int** *c*, **FILE** **stream*);

Purpose
Print output to **stdout**.

Return Value
The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.

Parameters
c Pointer to an integer.

stream Pointer to a stream.

Description
The **fputc** function writes the character specified by *c* (converted to an **unsigned char**) to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opennded with append mode, the character is appended to the ouput stream.

Example

```
/* a sample program that opens a temporary file and
write a character and a string into it. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch='A';
    char *name;

    if((name = tmpnam(NULL)) == NULL)
        printf("Error in creating temp file.\n");
    /* open file for reading only */
    if((stream=fopen(name,"w")) == NULL) {
      printf("cannot open file\n");
      exit(1);
    }
    else {
        fputc(ch,stream);
        fputs("This is a test\n",stream);
        fputs("This is a test\n",stdout);
        fclose(stream);
    }
}
```

Output

```
This is a test
```

See Also

fputs

Synopsis
#include <**stdio.h**>
int ***fputs**(**const char** * *restrict s*, **FILE** * *restrict stream*);

Purpose
Print output to a string.

Return Value
The **fputs** function writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null character is not written.

Parameters
s Pointer to a character.

stream Pointer to a stream.

Description
The **fputs** function writes the string pointed to by *s* pointed to by *stream*. The terminating null character is not written.

Example
see **fputc**().
Output

See Also
fputc()

fread

Synopsis
#include <stdio.h>
int fread(**void** * *restrict ptr*, **size_t** *size*, **size_t** *nmemb*, **FILE** * *restrict stream*);

Purpose
Read input from string into an array.

Return Value
The **fread** function returns the number of elements successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, *fread* returns zero and the contents of the array and the state of the stream remain unchanged.

Parameters
ptr Pointer.

size Argument.

nmemb Argument.

stream Pointer to a stream.

Description
The **fread** function reads, into the array pointed to by *ptr*, up to *nmemb* elements whose size is specified by *size*, from the stream pointed to by *stream*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Example
Output
See Also

freopen

Synopsis
#include <stdio.h>
FILE ***freopen**(**const char** **filename*, **const char** **mode*, **FILE** * *restrict stream*);

Purpose
Open a file.

Return Value
The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of *stream*.

Parameters
filename Pointer to a file.

mode Points to a string.

stream Points to a stream.

Description
The **freopen** function opens the file whose name is the string pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used just as in the **fopen** function.

If the *filename* is a null pointer, the **freopen** function attempts to change the mode of the stream to that specified by *mode*, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.
The **freopen** function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Example
Output
See Also

fscanf

Synopsis
#include <stdio.h>
int fscanf(**FILE** **restrict stream*, **const char** **restrict format*, ...);

Purpose
Read the input from a stream using the given format.

Return Value
The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of any early matching failure.

Parameters
stream Pointer to a stream.

format Points to a character.

Description
The **fscanf** function reads input from the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither a % nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character
After the %, the following appear in sequence:

- An optional assignment-suppresing character *****.
- An optional nonzero decimal integer that specifies the maximum field width (in characters).
- An optional *length modifier* that specifies the size of the receiving object.
- A *conversion specifier* character that specifies the type of conversion to be applied.

The **fscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[, c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item (or, in the case of a %**n** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

The length modifiers and their meanings are:

hh Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

l (ell) Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **long int** or **unsigned long** int; that a following **a, A, e, E, f, F, g**, or **G** conversion specifier applies to an argument with type pointer to **double**; or that a following **c, s**, or **[** conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

j Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

t Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

L Specifies that a following **a, A, e, E, f, F, g**, or **G** converson specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating-point number.

c Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).

If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

s Matches a sequence of non-white-space characters.
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[Matches a nonempty sequence of characters from a set of expected characters (the scanset).
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.
The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a **-** character is in the scanlist and is not the first, nor the second where the first character is a ^, nor the last character, the behavior is implementation-defined.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **fscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment- suppressing character or a field width, the behavior is undefined.

% Matches a single % character; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **A, E, F, G**, and **X** are also valid and behave the same as, respectively, **a, e, f, g**, and **x**.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure.

Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.

Example
Output
See Also

fseek

Synopsis
#include <stdio.h>
int fseek(**FILE** **stream*, **long int** *offset*, **int** *whence*);

Purpose
Store current values of file position indicators.

Return Value
the **fseek** function returns nonzero only for a request that cannot be satisfied.

Parameters
stream Pointer to a stream.

offset Argument.

whence Argument.

Description
The **fseek** function sets the file position indicator for the stream pointed to by *stream*. If a read or write error occurs, the error indicator for the stream is set and **fseek** fails.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified position is the beginning of the file if *whence* is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**. A binary stream need not meaningfully support **fseek** calls with a *whence* value of **SEEK_END**.

For a text stream, either *offset* shall be zero, or *offset* shall be a value returned by an earlier successful call to the **ftell** function on a stream associated with the same file, and *whence* shall be **SEEK_SET**.

After determining the new position, a successful call to the **fseek** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful **fseek** call, the next operation on an update stream may be either input or output.

Example

```
/* a sample program that reads the third charter
in a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch;
    long int offset;

    offset = 2*sizeof(char);
    /* open file for reading and writing */
    if((stream = fopen("fseek.c","r")) == NULL){
```

```
      printf("cannot open file\n");
      exit(1);
    }
    fseek(stream,offset,0);     //search the third character
    fread(&ch,sizeof(char),1,stream);   //read a character
    printf("The third character in file is '%c'\n",ch);
    rewind(stream);                     //set file indicator to first
    fread(&ch,sizeof(char),1,stream);
    printf("The first character in file is '%c'\n",ch);
    fclose(stream);
}
```

Output

```
The third character in file is ' '
The first character in file is '/'
```

See Also

fsetpos

Synopsis
#include <stdio.h>
int fsetpos(**FILE** * *restrict stream*, **fpos_t** * *resitrict pos*);

Purpose
Store current values of file position indicators.

Return Value
If successful, the **fsetpos** function returns 0; on failure, the **fsetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Parameters
stream Pointer to a stream.

pos Pointer to an object.

Description
The **fsetpos** function sets the **mbstate_t** object (if any) and file position indicator for the *stream* pointed to by stream according to the value of the object pointed to by *pos*, which shall be a value obtained from an earlier successful call to the **fgetpos** function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and **fsetpos** fails.

A successful call to the **fsetpos** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful **fsetpos** call, the next operation on an update stream may be either input or output.

Example see **fgetpos**().
Output
See Also

ftell

Synopsis
#include <stdio.h>
long int ftell(**FILE** **stream*);

Purpose
Obtain the current value of the file position indicator for a stream.

Return Value
If successful, the **ftell** function returns the current value of the file position indicator for the stream. On failure, the **ftell** function returns -1L and stores an implementation-defined positive value in **errno**.

Parameters
stream Pointer to a stream.

Description
The **ftell** function obtains the current value of the file position indicator for the stream pointed to by *stream*. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, its file position indicator contains unspecified information, usable by the **fseek** function for returning the file position indicator for the stream to its position at the time of the **ftell** call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

Example

```
/* a sample program that reads the third charter
in a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char ch;
    int cur_indicator;
    long int offset = 2*sizeof(char);
    /* open file for reading and writing */
    if((stream = fopen("ftell.c","r")) == NULL){
      printf("cannot open file\n");
      exit(1);
    }
    fseek(stream,offset,0);     //search the third character
    cur_indicator = ftell(stream);      // read current indicator
    fread(&ch,sizeof(char),1,stream);   //read a character
    printf("The third character in file is '%c'\n",ch);
    fseek(stream,cur_indicator-1,0);   // set the file indicator to
                                       // the number before the current
    fread(&ch,sizeof(char),1,stream);   //read a character
    printf("The second character in file is '%c'\n",ch);
    rewind(stream);                     //set file indicator to first
    fread(&ch,sizeof(char),1,stream);
    printf("The first character in file is '%c'\n",ch);
    fclose(stream);
```

```
}
```

Output

```
The third character in file is ' '
The second character in file is '*'
The first character in file is '/'
```

See Also

fwrite

Synopsis
#include <stdio.h>
int fwrite(void * *restrict ptr*, **size_t** *size*, **size_t** *nmemb*, **FILE** * *restrict stream*);

Purpose
Read input from string into an array.

Return Value
The **fwrite** function returns the number of elements successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, *fwrite* returns zero and the contents of the array and the state of the stream remain unchanged.

Parameters
ptr Pointer.

size Argument.

nmemb Argument.

stream Pointer to a stream.

Description
The **fwrite** function writes, from the array pointed to by *ptr*, up to *nmemb* elements whose size is specified by *size*, from the stream pointed to by *stream*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Example

```
/* a sample program that flushes the buffer
after write operation. */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 80

int main() {
    FILE *stream;
    char buf[SIZE];
    char *name;
    size_t num;

    if((name = tmpnam(NULL)) == NULL)
        printf("Error in creating temp file.\n");
    /* open file for reading and writing */
    if((stream = fopen(name,"w")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    strcpy(buf,"1111111111111111");
    num = fwrite(buf,sizeof(char),10,stream);
```

```
    printf("Number of elements written to temp file: %d\n", num);
    fflush(stream);      // flush the buffer
    fclose(stream);
    remove(name);
}
```

Output

```
Number of elements written to temp file: 10
```

See Also

getchar

Synopsis
#include <**stdio.h**>
int getchar(**void**);

Purpose
Read next character from stream.

Return Value
The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

Parameters
No argument.

Description
The **getchar** function is equivalnet to **getc** with the argument **stdin**.

Example
Output
See Also

getc

Synopsis
#include <**stdio.h**>
int getc(**FILE** **stream*);

Purpose
Read input from a stream.

Return Value
The **getc** funciton returns the next character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

Parameters
stream Pointer to a stream.

Description
The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Example

```
/* a sample program that flushes the buffer
after write operation. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc , char *argv[]) {
    char choice;

    do {
      printf("1: Check spelling\n");
      printf("2: Correct spelling\n");
      printf("3: Lookup a word in the directory\n");
      printf("4: Quit\n");

      printf("\nEnter your selection: ");
      choice = getc(stdin);
    } while (!strchr("1234",choice));
    printf("You entered: %c\n", choice);
    return (0);
}
```

Output

```
1: Check spelling
2: Correct spelling
3: Lookup a word in the directory
4: Quit

Enter your selection: You entered: 4
```

See Also

getline

Synopsis
#include <**stdio.h**>
int getline(**FILE** **fp*, **char** **line*, **int** *n*);

Purpose
Get the input line ending with a carriage return character.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
fp A pointer to the file stream.

line A string into which the characters of the input line will be put.

n The size of string.

Description
This function puts the characters of the input line ending with a return character into string *line*.

Example

```
#include<stdio.h>

int main() {
  char l[40];
  getline(stdin, l, sizeof(l));
  printf("l = %s\n", l);
}
```

Output

```
l = abcd1234ABCD
```

See Also
getnum().

gets

Synopsis
#include <**stdio.h**>
int ***gets**(**char** **s*);

Purpose
Read characters from input streams into arrays.

Return Value
The **gets** function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during operation, the array contents are indeterminate and a null pointer is returned.

Parameters
s Pointer to a character.

Description
The **gets** function reads characters from the input stream pointed to by *stdin* into the array pointed to by *s*, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

Example
Output
See Also

perror

Synopsis
#include <**stdio.h**>
void perror(**const char** *s);

Purpose
Map the error number in **errno** to an error message.

Return Value
The **perror** function returns no value.

Parameters
stream Pointer to a stream.

Description
The **perror** function maps the error number in the integer expression **errno** to an error message. It writes a sequence of characters to the standard error stream thus: first (if *s* is not a null pointer and the character pointed to by *s* is not the null character), the string pointed to by s followed by a colon (**:**) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the **strerror** function with argument **errno**.

Example

```
/* Uses the perror function to display an error message.  */
#include <stdio.h>
#include <signal.h>
int delay = 1;
void childHandler();

int main()
{
   int pid;
   char *argv[3];

   argv[0]="-l";
   signal(SIGCHLD, childHandler);
   pid=fork();
   if(pid==0)
   {
      execvp("i",argv);
      perror("Error:");
   }
   else
   {
      sleep(delay);
      printf("Child %d exceeded limit and is being killed \n",pid);
      kill(pid, SIGINT);
   }
}

void childHandler()
```

```
{
   int childPid, childStatus;
   childPid=wait(&childStatus);
   printf("Child %d terminated within %d seconds \n", childPid, delay);
   exit(0);
}
```

Output

```
Error:: No such file or directory
Child 10698 terminated within 1 seconds
```

See Also

printf

Synopsis
#include <**stdio.h**>
int printf(**const char** *restrict format*, ...);

Purpose
Print output to **stdout**.

Return Value
The **printf** function returns the number of characters transmitted, or a negative value if an output or encoding error occured.

Parameters
format Pointer to a character.

Description
The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.

Example

```
/* this program prints a string.
*/

#include <stdio.h>

int main() {
    char *c = "This is a test string";
    printf("%s\n", c);
}
```

Output

```
This is a test string
```

See Also

putchar

Synopsis
#include <**stdio.h**>
int putchar(**int** *c*);

Purpose
Read next character from stream.

Return Value
The **putchar** function returns the character written. If a read error occurs, the error indicator for the stream is set and **putchar** returns **EOF**.

Parameters
c Character.

Description
The **putchar** function is equivalnet to **putc** with the argument **stdout**.

Example

```
/* a sample program that writes a string
to stdout. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char str[] = "This is a test.\n";
    char *p;

    p = str;
    for(; *p; p++) putchar(*p);
}
```

Output

```
This is a test.
```

See Also

putc

Synopsis
#include <stdio.h>
int putc(**int** *c*, **FILE** **stream*);

Purpose
Print output to stream.

Return Value
The **putc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

Parameters
c Pointer to an integer.

stream Pointer to a stream.

Description
The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate *stream* more than once, so that argument should never be an expression with side effects.

Example

```
/* a sample program that writes a string
to a file. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char *name;
    int c, i;

    c = 'a';
    if((name = tmpnam(NULL)) == NULL)
        printf("temp file name could not be made. \n");
    /* open file for reading and writing */
    if((stream = fopen(name,"w")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    for(i =0; i <10; i++) {
        putc(c, stream);
        printf("%c", (char)c);
    }
    printf("\n");
    fclose(stream);
    remove(name);
}
```

Output

```
aaaaaaaaaa
```

See Also

puts

Synopsis
#include <**stdio.h**>
int puts(**const char** **s*);

Purpose
Write output to a string.

Return Value
The **puts** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

Parameters
s Pointer to a character.

Description
The **puts** function writes the string pointed to by *s* to the stream pointed to by **stdout**, and appends a newline character to the output. The terminating null character is not written.

Example

```
/* a sample program that writes a string
to stdout. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char str[80];

    strcpy(str,"this is a test.\n");
    puts(str);
}
```

Output

```
this is a test.
```

See Also

remove

Synopsis
#include <**stdio.h**>
int remove(**const char** **filename*);

Purpose
Remove a file.

Return Value
The **remove** function returns zero if the operation succeeds, nonzero if it fails.

Parameters
filename Pointer to a file.

Description
The **remove** function causes the file whose name is the string pointed to by *filename* to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Example
Output
See Also

rename

Synopsis
#include <**stdio.h**>
int rename(**const char** **old*, **const char** **new*);

Purpose
Remove a file.

Return Value
The **rename** function returns zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

Parameters
old Pointer to the old file name.

new Pointer to the new file name.

Description
The **rename** function causes the file whose name is the string pointed to by *old* to be henceforth known by the name given by the string pointed to by *new*. The file named *old* is no longer accessible by that name. If a file named by the string pointed to by *new* exists prior to the call to the **rename** function, the behavior is implementation-defined.

Example

```
/* a sample program that renames the file specified as the first
command line argument to that specified by the second command
line argument. this program rename.c is executed by command
       rename.c < rename.in
   Input file rename.in contains the following data: rename.c temp
*/
#include <stdio.h>

int main() {
    char name1[80], name2[80];

    printf("Enter filename: ");
    scanf("%s", name1);
    printf("%s\n", name1);
    printf("Enter new name: ");
    scanf("%s", name2);
    printf("%s\n", name2);
    if(rename(name1, name2)!=0)
        printf("Rename error\n");
    else {
printf("Old file name: %s\n",name1);
printf("New file name: %s\n", name2);
    }
    if(rename(name2, name1) !=0)
        printf("Rename error\n");
    else {
```

```
printf("Old file name: %s\n", name2);
printf("New file name: %s\n", name1);
    }
    return (0);
}
```

Output

```
Enter filename: rename.c
Enter new name: temp
Old file name: rename.c
New file name: temp
Old file name: temp
New file name: rename.c
```

See Also

rewind

Synopsis
#include <stdio.h>
void rewind(**FILE** **stream*);

Purpose
Set the file position indicator to the beginning of a stream.

Return Value
The **rewind** function returns no value.

Parameters
stream Pointer to a stream.

Description
The **rewind** function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to

(void)fseek(*stream, 0L, SEEK_SET***)**

except that the error indicator for the stream is also cleared.

Example see **fseek**().
Output
See Also

scanf

Synopsis
#include <**stdio.h**>
int scanf(**const char** *restrict format*, ...);

Purpose
Scan input from **stdin**.

Return Value
The **scanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, even zero, in the event of an early matching failure.

Parameters
format Pointer to a character.

Description
The **scanf** function is equivalent to **fscanf** with the argument **stdin** interposed before the arguments to **scanf**.

Example

```
/* this program scanf.c is executed by command
       scanf.c < scanf.in
   Input file scanf.in contains the following data
       1.0 2.0 3.0 4.0
*/
#include <stdio.h>

int main() {
    float f;
    double d;

    printf("input f\n");
    scanf("%f",&f);
    printf("Your input of f is %f\n", f);
    printf("input d\n");
    scanf("%lf",&d);
    printf("Your input of d is %f\n", d);
    printf("input f\n");
    scanf("%lf",&f); /* ERROR: should use %f for float */
    printf("Your input of d is not what you typed in f = %f\n", f);
    printf("input d\n");
    scanf("%f",&d); /* ERROR: should use %lf for double */
    printf("Your input of d is not what you typed in d = %f\n", d);
}
```

Output

```
input f
Your input of f is 1.000000
input d
```

```
Your input of d is 2.000000
input f
Your input of d is not what you typed in f = 2.125000
input d
Your input of d is not what you typed in d = 512.000000
```

See Also

setbuf

Synopsis
#include <**stdio.h**>
void setbuf(**FILE** **restrict stream*, **char** * *restrict buf*);

Purpose
Specify which buffer a stream will use.

Return Value
The **setbuf** function returns no value.

Parameters
stream Points to a stream.

buf Points to a character.

Description
Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for *mode* and **BUFSIZ** for *size*, or (if *buf* is a null pointer), with the value **_IONBF** for *mode*.

Example

```
/* a sample program that reads a file then
displays it's content. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char buffer[BUFSIZ], ch;

    /* open file for reading only */
    if((stream=fopen("setbuf.c","r")) == NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    setbuf(stream,buffer); /* associate a buffer with the stream */
    ch = fgetc(stream);
    printf("%c\n",ch);
    fclose(stream);
}
```

Output

```
/
```

See Also

setvbuf

Synopsis
#include <stdio.h>
void setvbuf(**FILE** * *restrict stream*, **char** * *restrict buf*);

Purpose
Specify the mode and buffer size of the given stream.

Return Value
The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for mode or if the request cannot be honored.

Parameters
stream Points to a stream.

buf Points to a character.

mode Holds how stream will be buffered.

size Holds the size of the buffer.

Description
The **setvbuf** function may be used only after the *stream* pointed to by stream has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument *mode* determines how the stream will be buffered, as follows: **_IOFBF** causes input/output to be fully buffered; **_IOLBF** causes input/output to be line buffered;**_IONBF** causes input/output to be unbuffered. If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function and the argument *size* specifies the size of the array; otherwise, *size* may determine the size of a buffer allocated by the **setvbuf** function. The contents of the array at any time are indeterminate.

Example

```
/* a sample program that reads a file then
displays it's content. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *stream;
    char buffer[500], ch;
    size_t size = 500;

    /* open file for reading only */
    if((stream=fopen("setvbuf.c","r")) == NULL){
      printf("cannot open file\n");
      exit(1);
    }
    setvbuf(stream,buffer,_IOFBF,size); /* associate a buffer with the stream */
    ch = fgetc(stream);
    printf("%c\n",ch);
```

```
    fclose(stream);
}
```

Output

```
/
```

See Also

snprintf

Synopsis
#include <stdio.h>
int snprintf(**char** * *restrict s*, **size_t** *n*, **const char** * *restrict format*, ...);

Purpose
Print output to an array.

Return Value
The **snprintf** function returns the number of characters that would have been written had **n** been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than **n**.

Parameters
s Pointer to an array.

n Argument.

format Pointer to a character.

Description
The **snprintf** function is equivalent to **fprintf**, except that the ouput is written into an array (specified by arguments *s*) rather than to a stream. If *n* is zero, nothing is written, and *s* may be a null pointer. Otherwise, output characters beyond the **n**-1st are discarded rather than being written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

sprintf

Synopsis
#include <stdio.h>
int sprintf(**char** * *restrict s*, **const char** * *restrict format*, ...);

Purpose
Print output to an array.

Return Value
The **sprintf** function returns the number of characters written in the array, not counting the terminating null character, or negative value if an encoding error occurred.

Parameters
s Pointer to an array.

format Pointer to a character.

Description
The **sprintf** function is equivalent to **fprintf**, except that the ouput is written to an array (specified by arguments **s**) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

sscanf

Synopsis
#include <stdio.h>
int sscanf(FILE * *restrict s*, **const char** * *restrict format*, ...);

Purpose
Read input from a stream.

Return Value
The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Parameters
s Pointer to an array.

format Pointer to a character.

Description
The **sscanf** function is equivalent to **fscanf**, except that the input is obatined from a string (specified by argument **s**) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf** function. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

tmpfile

Synopsis
#include <**stdio.h**>
FILE ***tmpfile**(**void**);

Purpose
Create a temporary file.

Return Value
The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

Parameters
No argument.

Description
The **tmpfile** function creates a temporary file that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with the **"wb+"** mode.

Example

```
/* a sample program that creates a temporary working file. */
#include <stdlib.h>
#include <stdio.h>

int main() {
    FILE *temp;

    if((temp = tmpfile()) == NULL) {
        printf("Cannot open temporary work file.\n");
        exit(1);
    }
}
```

Output

See Also

tmpnam

Synopsis
#include <**stdio.h**>
char ***tmpnam**(**char** **s*);

Purpose
Create a temporary string for a file name.

Return Value
If the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the **tmpnam** function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam char**'s; the **tmpnam** function writes its result in that array and returns the argument as its value.

Parameters
s Pointer to a character.

Description
The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file.

The **tmpnam** function generates a different string each time it is called, up to **TMP_MAX** times. If it is called more than **TMP_MAX** times, the behavior is implementation-defined.

The implementation shall behave as if no library function calls the **tmpnam** function.

Example

```
/* a sample program that displays three unique temporary
file name. */
#include <stdio.h>

int main() {
    char name[40];
    int i;

    for(i = 0; i < 3; i++) {
      tmpnam(name);
      printf("%s\n",name);
    }
}
```

Output

```
/var/tmp/aaaxraG7u
/var/tmp/baayraG7u
/var/tmp/caazraG7u
```

See Also

ungetc

Synopsis
#include <stdio.h>
int ungetc(**int** *c*, **FILE** **stream*);

Purpose
Push a character back onto the input stream.

Return Value
The **ungetc** funciton returns the next character pushed back after conversion, or **EOF** if the operation fails.

Parameters
c Pointer to an integer.

stream Pointer to a stream.

Description
The **ungetc** function pushes the character specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file positioning function (**fseek, fsetpos**, or **rewind**) discards any pushed- back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **ungetc** function; if its value was zero before a call, it is indeterminate after the call.

Example

```
/* a sample program that inserts a character into the
buffer of a file then reads it and displays it. */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#define BUF_SIZE 40


int main() {
    char buf[BUF_SIZE], c;
    FILE *stream;
```

```
    if((stream = fopen("ungetc.c", "r")) != NULL) {
        c = fgetc(stream);
        printf("The character pulled from the file is: %c\n", c);
        printf("ungetc(c,stream) = ");
        fputc( ungetc(c,stream), stdout);
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
The character pulled from the file is: /
ungetc(c,stream) = /
```

See Also

vfprintf

Synopsis
#include <stdio.h>
int vfprintf(**FILE** * *restrict stream*, **const char** * *restrict format*, **va_list** *arg*);

Purpose
Print output to a given stream.

Return Value
The **vfprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Parameters
stream Pointer to a stream.

format Pointer to a character.

arg Argument.

Description
The **vfprintf** function is equivalent to **fprintf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfprintf** function does not invoke the **va_end** macro.

Example
Output
See Also

vprintf

Synopsis
#include <**stdio.h**>
int vprintf(**const char** * *restrict format*, **va_list** *arg*);

Purpose
Print output to a given stream.

Return Value
The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Parameters
format Pointer to a character.

arg Argument.

Description
The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_ arg** calls). The **vprintf** function does not invoke the **va_end** macro.

Example
Output
See Also

vsnprintf

Synopsis
#include <stdio.h>
int vsnprintf(**char** * *restrict s*, **size_t** *n*, **const char** * *restrict format*, **va_list** *arg*);

Purpose
Print output to an array.

Return Value
The **vsnprintf** function returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than *n*.

Parameters
s Pointer to a character.

n Argument.

format Pointer to a character.

arg Argument.

Description
The **vsnprintf** function is equivalent to **snprintf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsnprintf** function does not invoke the **va_end** macro. If copying takes place between objects that overlap, the behavior is undefined.

See Also

vsprintf

Synopsis
#include <**stdio.h**>
int vsprintf(**char** * *restrict s*, **const char** * *restrict format*, **va_list** *arg*);

Purpose
Print output to an array.

Return Value
The **vsprintf** function returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred.

Parameters
s Pointer to a character.

format Pointer to a character.

arg Argument.

Description
The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by *arg*, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsprintf** function does not invoke the **va_end** macro. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

Chapter 18

String Handling — <stdlib.h>

The header **stdlib.h** declares five types and several functions of general utility, and defines several macros.

The types declared are **size_t** and **wchar_t**,

div_t

which is a structure type that is the type of the value returned by the **div** function,

ldiv_t

which is a structure type that is the type of the value returned by the **ldiv** function, and

lldiv_t

which is a structure type that is the type of the value returned by the **lldiv** function.

The macros defined are

EXIT_FAILURE

and

EXIT_SUCCESS

which expand to integer constant expressions that may by be used as the argument to the **exit** function to return unsuccessful or successful termination status, respectively, to the host environment;

RAND_MAX

which expands to an integer constant expression, the value of which is the maximum value returned by the **rand** function; and

MB_CUR_MAX

which expands to a positive integer expression with type **size_ t** whose value is the maximum number of bytes

in a multibyte character for the extended character set specified by the current locale (category **LC_TYPE**), and whose value is never greater than **MB_LEN_MAX**.

Public Data
None.

Functions
The following functions are prototyped in the header file **stdlib.h**.

Functions	Description
abort()	Terminates the program.
abs()	Computes an absolute value.
atexit()	Registers the termination of a program.
atoc()	Converts portion of a string to double complex representation.
atof()	Converts portion of a string to double representation.
atoi()	Converts portion of a string to int representation.
atol()	Converts portion of a string to long int representation.
atoll()	Converts portion of a string to long long int representation.
bsearch()	Searches for matching objects.
calloc()	Allocates space for an array.
div()	Computes a quotient and a remainder.
exit()	Causes normal program termination.
free()	Deallocates a space.
getenv()	Searches for matching strings.
iscnum()	Find out if a string is a valid complex number.
isenv()	Find out if a string is an environment variable.
isnum()	Find out if a string is a valid numerical number.
iswnum()	Find out if a wide character string is a valid numerical number.
labs() ()	Computes an absolute value.
llabs()	Computes an absolute value.
malloc()	Allocates space for an object.
mblen()	Determine the number of bytes in a multibyte character.
mbstowcs()	Converts a sequence of multibyte characters into corresponding codes.
mbtowc()	Determines the number of bytes in a multibyte character.
qsort()	Sorts an array of objects.
rand()	Returns a psuedo-random integer.
realloc()	Deallocates pointer from old object to new object.
remenv()	Remove an environment variable.
srand()	Returns a new psuedo-random integer.
strtod()	Converts initial portion of a string.
strtof()	Converts initial portion of a string.
strtol()	Converts initial portion of a string.
strtold()	Converts initial portion of a string.
strtoll()	Converts initial portion of a string.
strtoul()	Converts initial portion of a string.

strtoull()	Converts initial portion of a string.
system()	Execute a program through a command processor.
wcstombs()	Converts a sequence of codes into corresponding multibyte characters.
wctomb()	Determines the number of bytes needed to represent a multibyte character.

Macros
The following macros are defined for the **stdlib** header.

Macro	Description
EXIT_FAILURE	Returns unsuccessful termination status.
EXIT_SUCCESS	Returns successful termination status.
RAND_MAX	Integer which holds the maximum value returned by the **rand** function.
MB_CUR_MAX	Unsigned integer which holds maximum number of bytes specified by the current locale.

Declared Types
The following types are declared in the **stdlib** header.

Declared Types	Description
size_t	Unsigned integer.
wchar_t	Integer.
div_t	Structure of type returned by **div**.
ldiv_t	Structure of type returned by **ldiv**.
lldiv_t	Structure of type returned by **lldiv**.

Portability
This header has no known portability problem.

abort

Synopsis
#include <stdlib.h>
void abort(*void***);**

Purpose
Terminates the program.

Return Value
The **abort**() function does not return to its caller.

Parameters
no argument.

Description
The **abort**() function causes abnormal program termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open output streams are flushed or open streams closed or temporary files removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise(SIGABRT)**.

Example

```
/* a sample program that use abort()*/
#include <stdlib.h>
#include <stdio.h>

int main() {
    printf("abort() is called\n");
    abort();
    printf("This is not called\n");
}
```

Output

```
abort() is called
```

See Also

abs

Synopsis
#include <stdlib.h>
int abs(int *j*);
long int labs(long int *j*);
long long int llabs(long long int *j*);

Purpose
Computes an absolute value.

Return Value
The **abs**(), **labs**(), and **llabs**() functions return the absolute value.

Parameters
j Integer argument.

Description
The **abs**(), **labs**(), and **llabs**() functions compute the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.

Example

```
/* A sample program to compute absolute values.*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num = -2;
    printf("Number is %d\n", num);
    printf("but after running abs()\n");
    num = abs(num);
    printf("Number is %d\n", num);
}
```

Output
See Also

atexit

Synopsis
#include <stdlib.h>
int atexit(void (*func)(*void*));

Purpose
Registers the termination of a program.

Return Value
The **atexit**() function returns zero if the registration succeeds, nonzero if it fails.

Parameters
func A registered function to be called at normal program termination.

Description
The **atexit**() function registers the function pointed to by *func*, to be called without arguments at normal program termination.

Environmental Limits
The number of functions that can be registered is not limited.

Example

```
/* this is a sample that use atexit() */
#include <stdio.h>
#include <stdlib.h>
void atexitHandler();

int main()
{
    printf("The program is running. \n");
    atexit(atexitHandler);   /* call atexitHandler before exit */
}

void atexitHandler()
{
    printf("Call function to exit. \n ");
}
```

Output

```
The program is running.
Call function to exit.
```

See Also
exit()

atoc

Synopsis
#include <stdlib.h>
double complex atoc(const char ***nptr*****);**

Purpose
Converts portion of a string to double complex representation.

Return Value
The **atoc**() function returns the converted value.

Parameters
nptr Pointer to a character.

Description
The **atoc**() function converts the initial portion of the string pointed to by *nptr* to **double complex** representation. The complex number can be represented as complex(re, im) or in conventional mathematical notation such as $a, a + ib, a + i * b, ib + a, i * b$, where imaginary number i can be replaced by j or I, and a and b are constants, and plus sign '+' can be replaced by minus sign '-'.

Example

```
/* a sample program that changes a string to double complex*/
#include <stdlib.h>
#include <complex.h>

int main() {
    char *number = "1.27859-I*32.6532";
    char *number2 = "1.27859+i32.6532";
    char *number3 = "complex(3,4)";
    double complex outnumber;

    outnumber = atoc(number);
    printf("the complex number is %f \n",outnumber);
    outnumber = atoc(number2);
    printf("the complex number is %f \n",outnumber);
    outnumber = atoc(number3);
    printf("the complex number is %f \n",outnumber);
}
```

Output

```
the complex number is complex(1.278590,-32.653200)
the complex number is complex(1.278590,32.653200)
the complex number is complex(3.000000,4.000000)
```

See Also
atof(), **atoi**(), **strtod**(), **strtof**(), **strtold**().

atof

Synopsis
#include <stdlib.h>
double atof(const char **nptr***);**

Purpose
Converts portion of a string to double representation.

Return Value
The **atof**() function returns the converted value.

Parameters
nptr Pointer to a character.

Description
The **atof**() function converts the initial portion of the string pointed to by *nptr* to **double** representation. Except for the behavior on error, it is equivalent to

strtod(nptr, (char **)NULL)

Example

```
/* a sample program that changes a string to double */
#include <stdlib.h>

int main() {
    char *number = "1.27859";
    double outnumber;

    outnumber = atof(number);
    printf("the out double number is %f \n",outnumber);
}
```

Output

```
the out double number is 1.278590
```

See Also
atoc(), **atoi**(), **strtod**(), **strtof**(), **strtold**().

atoi

Synopsis
#include <stdlib.h>
int atoi(const char **nptr****);**
long int atol(const char **nptr****);**
long long int atoll(const char **nptr****);**

Purpose
Converts portion of a string to int, long int, or long long int representation.

Return Value
The **atoi**(), **atol**(), and **atoll**() functions return the converted value.

Parameters
nptr Pointer to a character.

Description
The **atoi**(), **atol**(), and **atoll**() functions convert the initial portion of the string pointed to by *nptr* to **int, long int**, and **long long int** representation, repectively. Except for the behavior on error, they are equivalent to

atoi(): (int)strtol(nptr, (char **)NULL, 10)

atol(): strtol(nptr, (char **)NULL, 10)

atoll(): strtoll(nptr, (char **)NULL, 10)

Example

```
/* a sample program that changes a string to integer */
#include <stdlib.h>

int main() {
    char *number = "1278";
    int outnumber;

    outnumber = atoi(number);
    printf("the out integer number is %d \n",outnumber);
}
```

Output

```
the out integer number is 1278
```

See Also
atoc(), **atof**(), **strtol**(), **strtoll**(), **strtoul**(), **strtoull**().

bsearch

Synopsis
#include <stdlib.h>
void * bsearch(const void **key*, **const void** **base*, **size_t** *nmemb*, **size_t** *size*, **int (****compar*)**(const void *, const void *));**

Purpose
Searches for matching objects.

Return Value
The **bsearch**() function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

Parameters
key Pointer to a void.
base Pointer to a void.
nmemb Integer type argument.
size Integer type argument.
compar Pointer to a function with two arguments and return value of int type.

Description
The **bsearch**() function searches an array of *nmemb* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element of the array is specified by *size*.

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the *key* object is considered, respectively, to be less than, equal to, or greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

Example

```
/* a sample program that decides if the character
   is in the alphabet */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *alpha="abcdefghijklmnopqrstuvwxyz";
    char ch, *p;
    int comp();

    ch='m';
    p=(char *) bsearch(&ch,alpha,26,1,comp);
    if(p)
        printf("%c is in alphabet\n", ch);
    else
```

```
        printf("%c is not in alphabet\n", ch);
}

/* compare two characters */
int comp(char *ch, char *s)
{
    return (*ch)-(*s);
}
```

Output

```
m is in alphabet
```

See Also

calloc

Synopsis
#include <stdlib.h>
void * calloc(size_t *nmemb***, size_t** *size***);**

Purpose
Allocates space for an array.

Return Value
The **calloc**() function returns either a null pointer or a pointer to the allocated space.

Parameters
nmemb Unsigned integer argument.
size Integer argument.

Description
The **calloc**() function allocates space for an array of *nmemb* objects, each of whose size is *size*. The space is initialized to all bits zero.

Example

```
/* a sample program that allocate 100 floats in memory */
#include <stdio.h>
#include <malloc.h>

int main() {
    float *p;

    p = (float *) calloc(100,sizeof(float));
    if(!p){
      printf("allocation failure - aborting\n");
      exit(1);
    }
    printf ("%f\n", p[8]);
    p[8] = 1.2;
    printf ("%f\n", p[8]);
    free(p);
}
```

Output

```
0.000000
1.200000
```

See Also

div

Synopsis
#include <**stdlib.h**>
div_tdiv(int *numer***, int** *denom***);**
ldiv_tdiv(long int *numer***, long int** *denom***);**
lldiv_tdiv(long long int *numer***, long long int** *denom***);**

Purpose
Computes a quotient and a remainder.

Return Value
The **div**(), **ldiv**(), and **lldiv**() functions return a structure of type **div_t**, **ldiv_t**, and **lldiv_t**, respectively, comprising both the quotient and the remainder. The structures shall contain (in either order) the members *quot* (the quotient) and *rem* (the remainder), each of which have the same type as the arguments *numer* and *denom*. If either part of the result cannot be represented, the behavior is undefined.

Parameters
numer Integer argument.
denom Integer argument.

Description
The **div**(), **ldiv**(), and **lldiv**() functions compute *numer* **/** *denom* and *numer* **%** *denom* in a single operation.

Example

```
/* a sample program that computes the quotient
and remainder of the division of the numerator
by the denominator. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int number=400;
    int denom=5;
    div_t result;

    result=div(number,denom);
    printf("%d is divided by %d\n",number,denom);
    printf("the quotient is %d\n",result.quot);
    printf("the remainder is %d\n",result.rem);
}
```

Output

```
400 is divided by 5
the quotient is 80
the remainder is 0
```

See Also

exit

Synopsis
#include <stdlib.h>
void exit(int *status***);**

Purpose
Causes normal program termination.

Return Value
The **exit**() function can not return to its caller.

Parameters
status Integer argument.

Description
The **exit**() function causes normal program termination to occur. If more than one call to the **exit**() function is executed by a program, the behavior is undefined.

First, all functions registered by the **atexit**() function are called, in the reverse order of their registration.

Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **tmpfile**() function are removed.

Finally, control is returned to the host environment. If the value of *status* is zero or **EXIT_SUCCESS**, an implementation-defined form of the status *successful termination* is returned. If the value of *status* is **EXIT_FAILURE**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Example

```
/* a sample program that use exit()*/
#include <stdlib.h>
#include <stdio.h>

int main() {
    printf("exit() is called\n");
    exit(EXIT_SUCCESS);
    printf("This is not called\n");
}
```

Output

```
exit() is called
```

See Also

free

Synopsis
#include <stdlib.h>
void free(void **ptr***);**

Purpose
Deallocates a space.

Return Value
The **free**() function returns either a null pointer or a pointer to the allocated space.

Parameters
ptr Pointer to an argument of void type.

Description
The **free**() function causes the space pointed to by *ptr* to be deallocated, that is, made available for further allocation. If *ptr* is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the **calloc**(), **malloc**(), or **realloc**() function, or if the space has been deallocated by a call to **free**() or **realloc**(), the behavior is undefined.

Example

```
/* a sample program that allocate 100 floats in memory */
#include <stdio.h>
#include <malloc.h>

int main() {
    float *p;

    p = (float *) calloc(100,sizeof(float));
    if(!p){
        printf("allocation failure - aborting\n");
        exit(1);
    }
    printf("Will now free 100 floats from memory.\n");
    free(p);
}
```

Output

```
Will now free 100 floats from memory.
```

See Also

getenv

Synopsis
#include <stdlib.h>
char * getenv(const char **name****);**

Purpose
Searches for matching strings.

Return Value
The **getenv**() function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **getenv**() function. If the specified **name** cannot be found, a null pointer is returned.

Parameters
name Pointer to a character.

Description
The **getenv**() function searched an *environmental list*, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

The implementation shall behave as if no library function calls the **getenv**() function.

Example
Output
See Also

iscnum

Synopsis
#include <stdlib.h>
int iscnum(**string_t** *pattern*);

Purpose
Find out if a string is a complex number or real number.

Return Value
This function returns 1 if the string is a valid complex number or real number or 0 if not.

Parameters
name The input string.

Description
This function determines if the input string *pattern* is a valid complex number, a pure imaginary number, or a real number. The complex number can be represented as complex(re, im) or in conventional mathematical notation such as $a, a+ib, a+i*b, ib+a, i*b$, where imaginary number i can be replaced by j or I, and a and b are constants, and plus sign '+' can be replaced by minus sign '-'.

Example

```
#include <complex.h>
#include <stdlib.h>
int main() {
   string_t s="3+I*4";

   printf("isnum(\"0x30\") = %d\n", isnum("0x30"));
   printf("isnum(\"abc\") = %d\n", isnum("abc"));
   printf("iswnum(L\"0x30\") = %d\n", iswnum(L"0x30"));
   printf("iswnum(L\"abc\") = %d\n", iswnum(L"abc"));

   printf("iscnum(\"10+I*4\") = %d\n",iscnum(s));
   printf("atoc(s) = %f\n",atoc(s));
   printf("iscnum(\"I*4\") = %d\n",iscnum("I*4"));
   printf("atoc(\"I*4\") = %f\n",atoc("I*4"));
   printf("iscnum(\"complex(1,2)\") = %d\n",iscnum("complex(1,2)"));
   printf("atoc(\"complex(1,2)\") = %f\n",atoc("complex(1,2)"));
   printf("iscnum(\"4\") = %d\n",iscnum("4"));
}
```

Output

```
isnum("0x30") = 1
isnum("abc") = 0
iswnum(L"0x30") = 1
iswnum(L"abc") = 0
iscnum("10+I*4") = 1
atoc(s) = complex(3.000000,4.000000)
iscnum("I*4") = 1
atoc("I*4") = complex(0.000000,4.000000)
```

```
iscnum("complex(1,2)") = 1
atoc("complex(1,2)") = complex(1.000000,2.000000)
iscnum("4") = 0
```

See Also
isnum(), **iswnum**(), **isenv**(), **iskey**(), **isvar**(), **isstudent**().

isenv

Synopsis
#include <**stdlib.h**>
int isenv(**string_t** *name*);

Purpose
Find out if a string is an environment variable.

Return Value
This function returns 1 if the string is an environment variable or 0 if it is not.

Parameters
name The input string.

Description
This function determines if the input string *name* is an environmetal variable.

Example

```
/* test if a string is an environment variable */
#include <stdlib.h>
#include <string.h>

int main() {
  int i;

  printf("isenv(\"CHHOME\") = %d\n", isenv("CHHOME"));
  printf("isenv(\"unknown\") = %d\n", isenv("unknown"));
}
```

Output

```
isenv("CHHOME") = 1
isenv("unknown") = 0
```

See Also
iskey(), **isnum**(), **isstudent**(), **isvar**().

isnum

Synopsis
#include <stdlib.h>
int isnum(**string_t** *pattern*);

Purpose
Find out if a string is a number.

Return Value
This function returns 1 if the string is a valid number or 0 if not.

Parameters
name The input string.

Description
This function determines if the input string *pattern* is a valid number.

Example
See **isenv**().

See Also
iscnum(), **iswnum**(), **isenv**(), **iskey**(), **isvar**(), **isstudent**().

iswnum

Synopsis
#include <**stdlib.h**>
int iswnum(**wchar_t** *pattern*);

Purpose
Find out if a wide character string is a number.

Return Value
This function returns 1 if the string is a valid number or 0 if not.

Parameters
name The input string.

Description
This function determines if the input string *pattern* is a valid number.

Example
See **isenv**().

See Also
isnum(), **iscnum**(), **isenv**(), **iskey**(), **isvar**(), **isstudent**().

malloc

Synopsis
#include <stdlib.h>
void * malloc(size_t*size*);

Purpose
Allocates space for an object.

Return Value
The **malloc**() function returns either a null pointer or a pointer to the allocated space.

Parameters
size Integer argument.

Description
The **malloc**() function allocates space for an object whose size is specified by *size* and whose value is indeterminate.

Example

```
/* a sample program that allocates space for a string
   dynamically, request user input, and then print the
   string backwards.  This is executed with the following
   statement:  malloc.c < malloc.in.  The contents of
   malloc.in are: abcd */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
    char *s;
    int t;

    s = malloc(80);
    printf("Please enter a string: ");
    if(!s) {
      printf("Memory request failed.\n");
      exit(1);
    }
    fgets(s, 80, stdin);
    printf("%s", s);
    printf("Converted string: ");
    for(t = strlen(s)-2; t>=0; t--)
      putchar(s[t]);
    printf("\n");
    free(s);
    return (0);
}
```

Output

```
Please enter a string: abcd
```

```
Converted string: dcba
```

See Also

mblen

Synopsis
#include <stdlib.h>
int mblen(const char *s**, size_t** *n**);**

Purpose
Determines the number of bytes in a multibyte character.

Return Value
If *s* is a null pointer, the **mblen**() function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, the **mblen**() function either returns 0 (if *s* points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

Parameters
s Pointer to a character.
n Unsigned integer argument.

Description
If *s* is not a null pointer, the **mblen**() function determines the number of bytes contained in the multibyte character pointed to by *s*. Except that the shift state of the **mbtowc** function is not affected, it is equivalent to

mbtowc((wchar_t *)0, s, n);

The implementation shall behave as if no library function calls the **mblen**() function.

Example

```
/* a sample program that displays the length of the
multibyte character pointed by mb. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    char mb[] = "This is a test";

    printf("The length of the multibyte character pointed by str is %d\n" \
    ,mblen(mb,sizeof(char)));
}
```

Output

```
The length of the multibyte character pointed by str is 1
```

See Also

mbstowcs

Synopsis
#include <stdlib.h>
size_t mbstowcs(wchar_t * restrict *pwcs***, const char restrict** **s***, size_t** *n***);**

Purpose
Converts a sequence of multibyte characters into corresponding codes.

Return Value
If an invalid multibyte character is encountered, the **mbstowcs**() function returns $(\mathbf{size\_t}) - \mathbf{1}$. Otherwise, the **mbstowcs**() function returns the number of array elements modified, not including a terminating zero code, if any.

Parameters
pwcs Integer argument.
s Pointer to a character.
n Unsigned integer argument.

Description
The **mbstowcs**() function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the —mbtowc function, except that the shift state of the **mbtowc**() function is not effected.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

Example

```
/* a sample program that converts the first 4 characters
in the multibyte character pointed by mb and puts the result
in str. */
#include <wchar.h>
#include <stdlib.h>

int main() {
    char s[] = "This is a test";
    wchar_t pwcs[20];
    size_t n;

    n = strlen(s)+1;
    printf("Original string: %s\n", s);
    printf("Number of characters to convert: %d\n", n);
    if(mbstowcs(pwcs,s,n) == -1) {
        printf("error convert\n");
        exit(1);
    }
    printf("The converted characters are: ");
```

```
    fputws(pwcs, stdout);
    fputs("\n", stdout);
}
```

Output

```
Original string: This is a test
Number of characters to convert: 15
The converted characters are: This is a test
```

See Also

mbtowc

Synopsis
#include <stdlib.h>
int mbtowc(wchar_t * restrict *pwc***, const char restrict** *s**, size_t** *n***);**

Purpose
Determines the number of bytes in a multibyte character.

Return Value
If *s* is a null pointer, the **mbtowc**() function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, the **mbtowc**() function either returns 0 (if *s* points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than *n* or the value of the **MB_CUR_MAX** macro.

Parameters
pwc Integer argument.
s Pointer to a character.
n Unsigned integer argument.

Description
The *s* is not a null pointer, the **mbtowc**() function determines the number of bytes that are contained in the multibyte character pointed to by *s*. It then determines the code for the value of type **wchar_t** that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and *pwc* is not a null pointer, the **mbtowc**() function stores the code in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

The implementation shall behave as if no library function calls the **mbtowc**() function.

Example

```
/* a sample program that converts the multibyte character in s
into its equivalent wide character and puts the result in the
array  pointed to by pwcs. */
#include <wchar.h>
#include <stdlib.h>

int main() {
    char s[] = "This is a test";
    wchar_t pwcs[8];
    size_t n;

    n = 1;
    if(mbtowc(pwcs,s,n) == -1) {
      printf("error convert\n");
      exit(1);
```

```
    }
    printf("The original string is: %s\n", s);
    printf("The converted characters are: ");
    fputws(pwcs, stdout);
    fputs("\n", stdout);
}
```

Output

```
The original string is: This is a test
The converted characters are: T
```

See Also

qsort

Synopsis
#include <stdlib.h>
void qsort(const void **base*, **size_t** *nmemb*, **size_t** *size*, **int (****compar*)**(ocnst void *, const void *));**

Purpose
Sorts an array of objects.

Return Value
The **qsort**() function returns no value.

Parameters
base Pointer to a void.
nmemb Integer type argument.
size Integer type argument.
compar Pointer to a function.

Description
The **qsort**() function sorts an array of *nmemb* objects, the initial element of which is pointed to by *base*. The size of each object is specified by *size*.

The contents of the array are sorted into ascending order according to a comparison function pointed to by *compar*, the function is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Example

```
/* a sample program that sorts a list of integers and
displays the result. */
#include <stdio.h>
#include <stdlib.h>

int num[10] = { 1,3,6,5,8,7,9,6,2,0};
int comp(const void *,const void *);

int main() {
    int i;

    printf("Original array:\t");
    for(i=0; i<10; i++)
        printf("%d ",num[i]);
    qsort(num,10,sizeof(int),comp);
    printf("\nSorted array:\t");
    for(i=0; i<10; i++)
        printf("%d ",num[i]);
    printf("\n");
}
```

```
/* compare the integers */
int comp(const void *i, const void *j) {
    return *(int *)i - *(int *)j;
}
```

Output

```
Original array: 1 3 6 5 8 7 9 6 2 0
Sorted array: 0 1 2 3 5 6 7 6 8 9
```

See Also

rand

Synopsis
#include <**stdlib.h**>
int rand(*void***);**

Purpose
Returns a psuedo-random integer.

Return Value
The **rand**() function returns a psuedo-random integer.

Parameters
no argument.

Description
The **rand**() function computes a sequence of psuedo-random integers in the range of 0 to **RAND_MAX**.

Example

```
/* a sample program that displays ten pseudorandom numbers. */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;

    for(i=0; i<10; i++) printf("%d ",rand());
    printf ("\n");
}
```

Output

```
16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
```

See Also

realloc

Synopsis
#include <stdlib.h>
void * realloc(void **ptr*, **size_t** *size*);

Purpose
Deallocates pointer from old object to new object.

Return Value
The **realloc**() function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

Parameters
ptr Pointer to a void.
size Integer argument.

Description
The **realloc**() function deallocates the old object pointed to by *ptr* and returns a pointer to a new object that has the size specified by *size*. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

If *ptr* is a null pointer, the **realloc**() function behaves like the **malloc**() function for the specified size. Otherwise, if *ptr* does not match a pointer earlier returned by the **calloc**(), **malloc**(), or **realloc**() function, or if the space has been deallocated by a call to the **free**() or **realloc**() function, the behavior is undefined. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

Example

```
/* a sample program that first allocates 16 characters,
copies the string "This is 16 chars" into them, and then
use realloc() to increase the size to 17 in order to place
a period at the end. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char *p;

    p = malloc(16);
    if(!p) {
        printf("Allocation error - aborting.");
        exit(1);
    }
    strcpy(p,"This is 16 chars");
    p = realloc(p,17);
    if(!p) {
        printf("allocation error - aborting.");
```

```
        exit(1);
    }
    strcat(p,".");
    printf(p);
    printf("\n");
    free(p);
}
```

Output

```
This is 16 chars.
```

See Also

remenv

Synopsis
#include <**stdlib.h**>
int remenv(**string_t** *nmae*);

Purpose
Remove an environment variable.

Return Value
This function returns 0 on success and -1 on failure.

Parameters
name A string containing the name of the environment variable to be removed.

Description
This function removes an environment variable with the name in string *name*.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  string_t s;
  putenv("TESINGENV=testingenv");
  s = getenv("TESINGENV");
  printf("s = %s\n", s);

  remenv("TESINGENV");
  s = getenv("TESINGENV");
  if(s == NULL)
    printf("ok: s = NULL\n");
  else
    printf("error: s = %s\n", s);
}
```

Output

```
s = testingenv
ok: s = NULL
```

See Also
putenv().

srand

Synopsis
#include <stdlib.h>
void srand(unsigned int *seed***);**

Purpose
Returns a new psuedo-random integer.

Return Value
The **srand**() function returns no value.

Parameters
seed Unsigned integer argument.

Description
The **srand**() function uses the argument as a seed for a new sequence of psuedo-random numbers to be returned by subsequent calls to **rand**(). If **srand**() is then called with the same seed value, the sequence of psuedo-random numbers shall be repeated. If **rand**() is called before any calls to **srand**() have been made, the same sequence shall be generated as when **srand**() is first called with a seed value of 1.

The implementation shall behave as if no library calls the **srand**() function.

Example

```
/* a sample program that displays ten pseudorandom numbers. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i,stime;
    long ltime;

    ltime = time(NULL); /* get current calendar time */
    stime = (unsigned int) ltime/2;
    srand(stime);
    for(i=0; i<10; i++) printf("%d ",rand());
    printf("\n");
}
```

Output

```
17875 24317 11838 16984 8536 4774 910 5469 11847 28683
```

See Also

strtod

Synopsis
#include <stdlib.h>
double strtod(const char * *restrict nptr***, char** ** *restrict endptr***);**
float strtof(const char * *restrict nptr***, char** ** *restrict endptr***);**
long double strtold(const char * *restrict nptr***, char** ** *restrict endptr***);**

Purpose
Converts portion of a string to double representation.

Return Value
The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return type and sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the result underflows, the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno**() acquires the value **ERANGE**is implemetation-defined.

Parameters
nptr Pointer to a character.
endptr Pointer to a character.

Description
The **strtod**(), **strtof**(), and **strtold**() functions convert the initial portion of the string pointed to by **nptr** to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possiby empty, sequence of white-space characters (as specified by the **isspace**() function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part.

— a **0x** or **0X**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part.

— one of **inf** or **infinity**, ignoring case.

— one of **NAN**or **NAN(***n-char-sequence*$_{opt}$**)**, ignoring case in the **NAN**part, where:

n-char-sequence
digit
nondigit
n-char-sequence digit
n-char-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point charcter is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating-point number, or is a binary exponent part does not appear in a binary floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence **INF** or **INFINITY** is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence **NAN** or **NAN (***n-char-sequence*$_{opt}$**)**, is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the menaing of the n-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, the value resulting from the conversion is correctly rounded.

In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Recommended Practice
If the subject sequence has the hexadecimal form and **FLT_RADIX**is not a power of 2, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most **DECIMAL_DIG** (defined in **float.h**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than **DECIMAL_DIG** significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having **DECIMAL_DIG** significant digits, such that the values of *L*, *D*, and *U* satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

Example
Output
See Also

strtol

Synopsis
#include <stdlib.h>
long int strtol(const char * *restrict nptr***, char** ** *restrict endptr***, int** *base***);**
long long int strtoll(const char * *restrict nptr***, char** ** *restrict endptr***, int** *base***);**
unsigned long int strtoul(const char * *restrict nptr***, char** ** *restrict endptr***, int base);**
unsigned long long int strtoull(const char * *restrict nptr***, char** ** *restrict endptr***, int base);**

Purpose
Converts a portion of a string to double representation.

Return Value
The **strtol**(), **strtoll**(), **strtoul**(), and **strtoull**() functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Parameters
nptr Pointer to a character.
endptr Pointer to a character.
base Integer argument.

Description
The **strtol**(), **strtoll**(), **strtoul**() and **strtoull**() functions convert the initial portion of the string pointed to by *nptr* to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possiby empty, sequence of white-space characters (as specified by the **isspace** function), a subject sequence resembling an integer represented in some radix determined by the value of *base*, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to an integer, and return the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values of 10 through 35; only letters and digits whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other

than a sign or a permissionible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Example
Output
See Also

system

Synopsis
#include <stdlib.h>
int system(const char **string***);**

Purpose
Execute a program through a command processor.

Return Value
If the argument is a null pointer, the **system**() function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the **system**() function does return, it returns an implementation-defined value.

Parameters
string Pointer to a character.

Description
If *string* is a null pointer, the **system**() function determines whether the host environment has a *command processor*. If *string* is not a null pointer, the **system**() function passes the string pointed to by *string* to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling **system**() to behave in a non-conforming manner or to terminate.

Example
Output
See Also

wcstombs

Synopsis
#include <stdlib.h>
size_t wcstombs(char restrict **s*, **const wchar_t * restrict** *pwcs*, **size_t** *n*);

Purpose
Converts a sequence of codes into corresponding multibyte characters.

Return Value
If a code is encountered that does not correspond to a valid multibyte character, the **wcstombs**() function returns $(\mathbf{size\_t}) - \mathbf{1}$. Otherwise, the **wcstombs**() function returns the number of bytes modified, not including a teminating null character, if any.

Parameters
pwcs Integer argument.
s Pointer to a character.
n Unsigned integer argument.

Description
The **wcstombs**() function converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the **wctomb**() function, except that the shift state of the **wctomb**() function is not affected.

No more than *n* elements will be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char s[8];
    wchar_t pwcs[8] = L"123abc\n";
    size_t retval;

    retval = wcstombs(s,pwcs, wcslen(pwcs)+1);
    if(retval != (size_t) -1) {
        printf("the original string is: ");
        fputws(pwcs, stdout);
     printf("The converted result is: %s\n",s);
    }
    else
printf("Error\n");
}
```

Output

```
the original string is: 123abc
The converted result is: 123abc
```

See Also

wctomb

Synopsis
#include <stdlib.h>
int wctomb(char **s*, **wchar_t** *wchar***);**

Purpose
Determines the number of bytes needed to represent a multibyte character.

Return Value
If **s** is a null pointer, the **wctomb**() function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **wctomb**() function returns -1 if the value of *wchar* does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of *wchar*.

In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

Parameters
s Pointer to a character.
wchar Integer argument.

Description
The **wctomb**() function determines the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* (including any change in shift state). It stores the multibyte character representation in the array object pointed to by *s* (if *s* is not a null pointer). At most **MB_CUR_MAX** characters are stored. If the value of *wchar* is zero, the wctomb function is left in the initial shift state.

The implementation shall behave as if no library function calls the **wctomb**() function.

Example

```
/* This is an example of wctomb(char*, wchar_t).  This function converts
   a wide character to the corresponding multibyte character.
*/
#include <stdio.h>
#include <stdlib.h>

int main() {
    char s[128];
    wchar_t pwcs;
    int retval,x;
    pwcs = 'T';
    wctomb(s,pwcs);
    if(retval != -1) {
      printf("The converted result is: %s\n",s);
    }
    else
printf("Error\n");
}
```

Output

```
The converted result is: T
```

See Also

Chapter 19

String Functions — <string.h>

The header **string.h** declares one type and several functions. The type is **size_t**. Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

Where an argument declared as **size_t n** specifies the length of the array for a function, **n** can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values. On such a call, if a function that locates a character finds no occurance, a function that copies characters copies zero characters.

Functions
The following functions are implemented using the **string** header.

Function	Description
memchr	Locates the first occurance of a specified character in a stream.
memcmp	Compares the first characters in two objects.
memcpy	Copies characters from one object to another.
memmove	Copies characters from one object to another.
memset	Copies a single value into specified number of locations in an object.
str2ascii	Returns the ASCII value for each character.
str2mat	Create a matrix of char from individual strings.
stradd	Adds two strings together.
strcasecmp	Compares two strings after converting all characters to lowercase
strcat	Appends a copy of one string to another string.
strchr	Locates the first occurance of a character in a string.
strcmp	Compares two strings.
strcoll	Compares two strings.
strconcat	Appends copies of strings to a string.
strcpy	Copies a string to an array.
strcspn	Computes the length of the maximum initial segment of a string.
strdup	Duplicates a string.
strerror	Maps values of **errnum**.

strgetc	Returns a character from a specified location.
strjoin	Combines strings as a string.
strlen	Computes the length of a string.
strncasecmp	Compares two strings after converting all characters to lowercase.
strncat	Appends characters from an array to a wide string.
strncmp	Compares characters between two arrays.
strncpy	Copies characters from one array to another array.
strpbrk	Locates first occurance of a character from one string in another string.
strputc	Replaces a character in a specified location.
strrchr	Locates last occurance of a character in a string.
strrep	Search and replace strings inside another string.
strspn	Computes the length of the maximum initial segment of a string.
strstr	Locates the first occurance of an object in one string in another string.
strtok	Breaks a string into a sequence of tokens.
strtok_r	Breaks a string into a sequence of tokens, reentrant.
strxfrm	Transforms string and places it in an array.

Macros

No macro is defined for the **string** header.

Declared Types

The following types are declared in the **string** header.

Declared Types	Description
size_t	Unsigned integer.

Portability

Differences between C and Ch

Functions **strcpy**, **strncpy**, **strcat**, **strncat**, **strcmp**, **strcoll**, **strtolc**, **memcpy**, and **memmove** are built-in functions in Ch.

Functions **str2ascii**, **str2mat**, **strgetc**, **strputc**, and **strrep** are available only in Ch.

memchr

Synopsis
#include <string.h>
void *memchr(const void *s**, int** *c***, size_t** *n***);**

Purpose
Locates the first occurance of a specified character in a stream.

Return Value
The **memchr**() function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

Parameters
s Pointer to void argument.
c Integer argument.
n Unsigned integer argument.

Description
The **memchr**() function locates the first occurance of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**.

Example

```
/* a sample program that prints 'is a test' on the screen.*/
#include <stdio.h>
#include <string.h>

int main() {
    char p[] = "this is a test\n";
    char *s;

    s = memchr(p,' ',strlen(p));
    printf(s);
}
```

Output

```
 is a test
```

See Also

memcmp

Synopsis
#include <string.h>
int memcmp(const void **s1***, const void** **s2***, size_t** *n***);**

Purpose
Compares the first characters in two objects.

Return Value
The **memcmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

Parameters
s1 Pointer to void argument.
s2 Pointer to void argument.
n Unsigned integer argument.

Description
The **memcmp**() function compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abcdefghijkl";
    char s2[] = "abcdefkl";
    int outcome;
    int l1,l2;
    int len;

    len = (l1 = strlen(s1)) < (l2 = strlen(s2)) ? l1:l2;
    outcome = memcmp(s1,s2,len);
    if(! outcome) printf("equal\n");
    else if(outcome<0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
first less than second
```

See Also

memcpy

Synopsis
#include <string.h>
void *memcpy(void * restrict *s1***, const void * restrict** *s2***, size_t** *n***);**

Purpose
Copies characters from one object to another.

Return Value
The **memcpy**() returns the value of **s1**.

Parameters
s1 Pointer to void argument.
s2 Pointer to void argument.
n Unsigned integer argument.

Description
The **memcpy**() function copies **n** characters from the oject pointed to by **s2** to the object pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

memmove

Synopsis
#include <string.h>
void *memmove(void **s1*, **const void** **s2*, **size_t** *n*);

Purpose
Copies characters from one object to another.

Return Value
The **memmove**() returns the value of **s1**.

Parameters
s1 Pointer to void argument.
s2 Pointer to void argument.
n Unsigned integer argument.

Description
The **memmove**() function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** or **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

Example
Output
See Also

memset

Synopsis
#include <wchar.h>
void *memset(void *s**, int** *c***, size_t** *n***);**

Purpose
Copies a single value into specified number of locations in an object.

Return Value
The **memset**() function returns the value of **s**.

Parameters
s Pointer to a void argument.
c Integer argument.
n Unsigned integer argument.

Description
The **memset**() function copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**.

Example
Output
See Also

str2ascii

Synopsis
#include <**string.h**>
unsigned int str2ascii(**string_t** *s*);

Purpose
Get the ASCII number of a string.

Return Value
This function returns the sum of ASCII value for each character of a string.

Parameters
s The input string.

Description
The function **str2ascii**() returns the sum of ASCII value for each character of a string.

Example

```
#include <stdio.h>
#include <string.h>

int main() {
   int ascii;
   ascii = str2ascii("abcd");
   printf("ASCII num for string ‘abcd‘ is %d\n", ascii);
}
```

Output

```
ASCII num for string ‘abcd‘ is 394
```

See Also

str2mat

Synopsis
#include <**string.h**>
int str2mat(**array char** *mat*[:][:], **string_t** *s*, ...);

Purpose
Create a matrix of char from individual strings.

Return Value
This function returns 0 on successful or -1 on failure.

Parameters
mat A two-dimensional array containing the created matrix.
s The individual string.

Description
This function creates a matrix of char containing the string *s* and strings passed from variable number arguments. The total number of strings including *s* and variable number arguments must equal the number of elements in the first dimension of array *mat*. If the number of strings is less than that of elements of array *mat*, the function call will fail and return -1. The strings more than the number of elements will be ignored.

Example

```
#include <stdio.h>
#include <string.h>
#include <array.h>

int main() {
   array char mat[3][10];
   int status;

   mat = (array char [3][10])' '; /* blank space */
   status = str2mat(mat, "abcd");
   printf("status = %d\nmat = %c\n", status, mat);
   str2mat(mat, "abcd", "efghi", "0123456789");
   printf("mat = \n%c\n", mat);
   status = str2mat(mat, "ABCD", "EFGH", "ab23456789", "too many strings");
   printf("status = %d\nmat = \n%c\n", status, mat);
}
```

Output

```
status = 0
mat = a b c d



mat =
a b c d
e f g h i
0 1 2 3 4 5 6 7 8 9
```

```
status = -1
mat =
A B C D
E F G H
a b 2 3 4 5 6 7 8 9
```

See Also
strputc().

stradd

Synopsis
#include <string.h>
string_t stradd(string_t *s*, ...);

Purpose
Adds strings together.

Return Value
The **stradd**() function returns the new string.

Parameters
s string argument.

Description
The **stradd**() function adds the string **s** and subsequent strings passed by variable number of arguments.

Example

```
/** add strings */
#include <stdio.h>
#include <string.h>

int main() {
  string_t s, s1 = "ABCD";
  char *s2="cbcd";
  char s3[]="1234";

  s = stradd(s1, s2, s3);
  printf("s = %s\n", s);
  s = stradd(s2, s3);
  printf("s = %s\n", s);
}
```

Output

```
s = ABCDcbcd1234
s = cbcd1234
```

See Also

strcasecmp

Synopsis
#include <string.h>
int strcasecmp(const char **s1*, **const char** **s2*);

Purpose
Compares two strings.

Return Value
The **strcasecmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strcasecmp**() function compares the string pointed to by **s1** to the string pointed to by **s2**. Before the stings are compared, the **tolower()** functions is called to change uppercase characters to lowercase.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abc";
    char s2[] = "abc";
    int outcome;
    int l1,l2;

    outcome = strcasecmp(s1,s2);
    if(! outcome) printf("equal\n");
    else if(outcome < 0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
equal
```

See Also
strcmp(), **strncmp**(), **strncasecmp**().

strcat

Synopsis
#include <string.h>
char *strcat(char * restrict *s1***, const char * restrict s2);**

Purpose
Appends a copy of one string to another string.

Return Value
The **strcat**() returns the value of **s1**.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strcat**() function appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also
strconcat(), **strjoin**(), **strncat**().

strchr

Synopsis
#include <string.h>
char strchr(const char *s, int c)

Purpose
Locates the first occurance of a character in a string.

Return Value
The **strchr**() function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Parameters
s Pointer to a character argument.
c Integer argument.

Description
The **strchr**() function locates the first occurance of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Example

```
/* a sample program that uses strchr to return the remaining
string after and including the first space */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;
    int c;

    c = ' ';
    p = "this This is a test";
    printf("Original string: %s\n", p);
    printf("Character to search for: '%c'\n", (char)c);
    p = strchr(p,c);
    printf("After: %s\n",p);
}
```

Output

```
Original string: this This is a test
Character to search for: ' '
After:  This is a test
```

See Also

strcmp

Synopsis
#include <string.h>
int strcmp(const char *s1, const char *s2);

Purpose
Compares two strings.

Return Value
The **strcmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strcmp**() function compares the string pointed to by **s1** to the string pointed to by **s2**.

Example
Output
See Also

strcoll

Synopsis
#include <string.h>
int strcoll(const char **s1***, const char** **s2***);**

Purpose
Compares two strings.

Return Value
The **strcoll**() function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2** when both are interpreted as appropriate to the current locale.

Parameters
s1 Pointer to a character.
s2 Pointer to a character.

Description
The **strcoll**() function compares the string pointed to by **s1** to the string pointed to by **s2**, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Example
Output
See Also

strconcat

Synopsis
#include <string.h>
char *strconcat(const char * string1, ...);

Purpose
Concatenates strings.

Return Value
The **strconcat**() returns a string with dynamically allocated memory.

Parameters
string1 Pointer to character argument.

Description
The **strconcat**() function concatenates all strings in the argument list, and puts the result into the returned string with a dynamically allocated memory. The dynamically allocated memory needs to be freed by the user.

Example

```
#include <string.h>
#include <stdarg.h>

int main() {
  char *buffer;
  char test1[90] = "abcd";

  buffer = strconcat(test1, "test2", "test3");
  printf("strconcat=%s\n",buffer);
  printf("test1=%s\n",test1);
  free (buffer);
  return 0;
}
```

Output

```
strconcat=abcdtest2test3
test1=abcd
```

See Also
strcat(), **strncat**(), **strjoin**().

strcpy

Synopsis
#include <**string.h**>
char *strcpy(char * restrict *s1***, const char * restrict** *s2***);**

Purpose
Copies a string to an array.

Return Value
The **strcpy**() returns the value of **s1**.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **strcpy**() function copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

strcspn

Synopsis
#include <string.h>
size_t *strcspn(const char *s1, const char *s2);

Purpose
Computes the length of the maximum initial segment of a string.

Return Value
The **strcspn**() function returns the length of the segment.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strcspn**() function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters *not* from the string pointed to by **s2**.

Example

```
/* a sample program that prints the number 8 which is
the index of the first charcacter in first string. */
#include <stdio.h>
#include <string.h>

int main() {
    int len;

    len = strcspn("this is a test","ab");
    printf("%d\n",len);
}
```

Output

```
8
```

See Also

strdup

Synopsis
#include <string.h>
char *strdup(const char * *s***);**

Purpose
The function duplicates a string.

Return Value
The **strdup**() returns a pointer to the duplicate string.

Parameters
s Character pointer argument.

Description
The **strdup**() function creates a duplicate of a string pointed to by *s* and stores it in memory obtained by using **malloc**.

Example

```
/* a sample program that displays a string. */
#include <stdio.h>
#include <string.h>

int main() {
    char str[80],*p;

    strcpy(str,"this is a test");
    p = strdup(str);   /* copy the content of str */
    if(p) {
      printf("%s\n",p);
      free(p);
    }
    else {
      printf("strdup() failed\n");
    }
}
```

Output

```
this is a test
```

See Also

strerror

Synopsis
#include <string.h>
char *strerror(int *errnum***);**

Purpose
Maps values of **errnum**.

Return Value
The **strerror**() function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **strerror** function.

Parameters
errnum Integer argument.

Description
The **strerror**() function maps the number in **errnum** to a message string. Typically, the values for **errnum** come from **errno**, but **strerror** shall map any value of type **int** to a message.

The implementation shall behave as if no library function calls the **strerror** function.

Example
Output
See Also

strgetc

Synopsis
#include <**string.h**>
int strgetc(**string_t** & *s*, **int** *i*);

Purpose
Get a character in a specified position of a string.

Return Value
This function returns the character.

Parameters
s A reference to a string.
i An integer indicating the position of character to be obtained in the string.

Description
This function returns the *(i+1)th* character of the string *s*. This function is for Ch **string_t**.

Example

```
/** access element of a string */
#include <stdio.h>
#include <string.h>

int main() {
  string_t s = "ABCD";

  printf("strgetc(s, 1) = %c \n", strgetc(s, 1));
  strputc(s, 2, 'm');
  printf("s = %s\n", s);
}
```

Output

```
strgetc(s, 1) = B
s = ABmD
```

See Also
strputc().

strjoin

Synopsis
#include <string.h>
char *strjoin(const char * separator, ...);

Purpose
Combines strings to a string separated by the specified delimiter string.

Return Value
The **strjoin**() returns a string with dynamically allocated memory.

Parameters
separator Pointer to character argument.

Description
The **strjoin**() function combines all strings in the argument list, and puts the result into the returned string with dynamically allocated memory. The returned string is separated by the delimiter specified by the first argument **separator**. The dynamically allocated memory needs to be freed by the user.

Example

```
#include <string.h>
#include <stdarg.h>

int main() {
  char *buffer;

  buffer = strjoin("+", "test1", "test2", "test3");
  printf("strjoin =%s\n",buffer);
  free (buffer);
  return 0;
}
```

Output

```
strjoin =test1+test2+test3
```

See Also
strcat(), **strncat**(), **strconcat**().

strlen

Synopsis
#include <string.h>
size_t strlen(const char *s**);**

Purpose
Computes the length of a string.

Return Value
The **strlen**() function returns the number of characters that precede the terminating null character.

Parameters
s Pointer to a character argument.

Description
The **strlen**() function computes the length of the string pointed to by **s**.

Example
Output
See Also

strncasecmp

Synopsis
#include <string.h>
int strncasecmp(const char **s1*, **const char** **s2*, **size_t** *n*);

Purpose
Compares characters between two arrays.

Return Value
The **strncasecmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**. Before the arguments are compared, all of the characters are converted to lowercase by **tolower**().

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.
n Unsigned integer argument.

Description
The **strncasecmp()** function compares not more than **n** characters (characters that follow a null character are not compared) from the array pointed to by **s1** to the array pointed to by **s2**.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abcdefG8";
    char s2[] = "abcdefg";
    int outcome;
    int l1,l2;
    int len;

    len = (l1 = strlen(s1)) < (l2 = strlen(s2)) ? l1:l2;
    outcome = strncasecmp(s1,s2,len);
    if(! outcome) printf("equal\n");
    else if(outcome < 0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
equal
```

See Also

strncat

Synopsis
#include <**string.h**>
char *strncat(char * restrict *s1***, const char * restrict** *s2***, size_t** *n***);**

Purpose
Appends characters from an array to a wide string.

Return Value
The **strncat**() returns the value of **s1**.

Parameters
s1 Pointer to character argument.
s2 Pointer to character argument.
n Unsigned integer argument.

Description
The **strncat**() function appends not more than **n** characters (a null character and characters that follow it are not appended) from the array pointed to by **s2** to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also
strcat(), **strconcat**(), **strjoin**().

strncmp

Synopsis
#include <string.h>
int strncmp(const char **s1***, const char** **s2***, size_t** *n***);**

Purpose
Compares characters between two arrays.

Return Value
The **strncmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the possibly null-terminated array pointed to by **s2**.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.
n Unsigned integer argument.

Description
The **strncmp**() function compares not more than **n** characters (characters that follow a null character are not compared) from the array pointed to by **s1** to the array pointed to by **s2**.

Example

```
/* a sample program that shows the outcome of a comparison
of its two strings.*/
#include <stdio.h>
#include <string.h>

int main() {
    char s1[] = "abcdefghijkl";
    char s2[] = "abcdefkl";
    int outcome;
    int l1,l2;
    int len;

    len = (l1 = strlen(s1)) < (l2 = strlen(s2)) ? l1:l2;
    outcome = strncmp(s1,s2,len);
    if(! outcome) printf("equal\n");
    else if(outcome < 0) printf("first less than second\n");
    else printf("first greater than second\n");
}
```

Output

```
first less than second
```

See Also

strncpy

Synopsis
#include <string.h>
char *strncpy(char * restrict *s1***, const char * restrict** *s2***, size_t** *n***);**

Purpose
Copies characters from one array to another array.

Return Value
The **strncpy**() returns the value of **s1**.

Parameters
s1 Character pointer argument.
s2 Character pointer argument.
n Unsigned integer argument.

Description
The **strncpy**() function copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

Example
Output
See Also

strpbrk

Synopsis
#include <string.h>
char *strpbrk(const char *s1, const char *s2);

Purpose
Locates first occurance of a character from one string in another string.

Return Value
The **strpbrk**() function returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strpbrk**() function locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

Example

```
/* a sample program that prints the message 's is a test'
on the screen. */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;

    p = strpbrk("this is a test"," absj");
    printf("%s\n",p);
}
```

Output

```
s is a test
```

See Also

strputc

Synopsis
#include <**string.h**>
int strputc(**string_t** &*s*, **int** *i*, **char** *c*);

Purpose
Replace a character in a specified position of a string with another character.

Return Value
This function returns 0 on successful and -1 in failure.

Parameters
s Reference to the string.
i Input integer indicating the position of the character to be replaced.
c Input character to be used to replace the character in the string.

Description
This function replaces the *(i+1)th* character of string *s* with character *c*. This function is for Ch **string_t**

Example
See **strgetc**.

See Also
strgetc().

strrchr

Synopsis
#include <string.h>
char *strrchr(const char *s, int c);

Purpose
Locates last occurance of a character in a string.

Return Value
The **strrchr**() function returns a pointer to the character, or a null pointer if **c** does not occur in the string.

Parameters
s Pointer to a character.
c Integer argument.

Description
The **strrchr**() function locates the last occurance of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Example

```
/* a sample program that prints the message 'is a test'
on the screen. */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;

    p = strrchr("this is a test",'i');
    printf("%s\n",p);
}
```

Output

```
is a test
```

See Also

strrep

Synopsis
#include <**string.h**>
string_t strrep(**string_t** *s1*, **string_t** *s2*, **string_t** *s3*);

Purpose
Search and replace strings inside another string.

Return Value
This function returns the new string in which all matched strings are replaced.

Parameters
s1 A string in which the string *s2* will be searched and replaced.
s2 A string to be searched.
s3 A string used to replace the string *s2*.

Description
This function searches and replaces the string *s2* with string *s3* inside the string *s1*. The replaced string is returned and string *s1* is not changed.

Example

```
#include <stdio.h>
#include <string.h>

int main() {
   string_t s, s1 = "He is smart, but you are smarter",
             s2 ="smart", s3="great", s4="tall";

   printf("s1 = %s\n", s1);
   s = strrep(s1, s2, s3);
   printf("s1 = %s\n", s1);
   printf("s = %s\n", s);
   s = strrep(s1, s2, s3);
   printf("s1 = %s\n", s1);
   printf("s = %s\n", s);
   s = strrep(s1, s2, s4);
   printf("s1 = %s\n", s1);
   printf("s = %s\n", s);
}
```

Output

```
s1 = He is smart, but you are smarter
s1 = He is smart, but you are smarter
s = He is great, but you are greater
s1 = He is smart, but you are smarter
s = He is great, but you are greater
s1 = He is smart, but you are smarter
s = He is tall, but you are taller
```

See Also
strgetc().

strspn

Synopsis
#include <string.h>
size_t strspn(const char *s1, const char *s2);

Purpose
Computes the length of the maximum initial segment of a string.

Return Value
The **strspn**() function returns the length of the segment.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strspn**() function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Example

```
/* a sample program that prints the index of the first
character in the string pointed to by str1 that does not
match any of the characters in str2.*/
#include <stdio.h>
#include <string.h>

int main() {
    unsigned int len;

    len = strspn("this is a test","siht ");
    printf("%d\n",len);
}
```

Output

```
8
```

See Also

strstr

Synopsis
#include <string.h>
char *strstr(const char *s1, const char *s2);

Purpose
Locates the first occurance of an object in one string in the other string.

Return Value
The **strstr**() function returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, the function returns **s1**.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
The **strstr**() function locates the first occurance in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

Example

```
/* a sample program that prints 'is is a test'. */
#include <stdio.h>
#include <string.h>

int main() {
    char *p;

    p = strstr("this is a test","is");
    printf("%s\n",p);
}
```

Output

```
is is a test
```

See Also

strtok_r

Synopsis
#include <string.h>
char *strtok_r(char * *s***, const char *** *sep***, char **** *lasts***);**

Purpose
Finds all of the character strings that are seperated by spaces.

Return Value
The **strtok_r**() function returns a pointer to the token found, or a null pointer if there is no token.

Parameters
s Pointer to a character argument.
sep Pointer to a character argument.
lasts Pointer to a user-provided pointer.

Description
The function considers the null-terminated string *s* as a sequence of zero or more text tokens seperated by one or more characters from the string *sep*. *lasts* points to stored information necessary for /strtokr/ to continue scanning the same string.

In the first call, *s* points to a null-terminated string, *sep* points to a null-terminated string of seperator characters, and *lasts* is ignored. The function returns a pointer to the first character of the token, writes a null character into *s* followed by the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s* is a **NULL** pointer and *lasts* is unchanged from previous calls so that subsequent calls will move through the string *s*, returning successive tokens until no tokens remain. The seperator string *sep* may vary from call to call. When no tokens remain in *s*, a **NULL** pointer is returned.

Example

```
/* a sample program that tokenizes the string, "The summer soldirer,
the sunshine patriot" with spaces and commas being the delimiters.
The output is The|summer|soldier|the sunshine|patriot. */
#include <stdio.h>
#include <string.h>

int main() {
    char *token, *str1, *delimit;
    char *endptr = NULL;

    str1 = "The summer soldier, the sunshine patriot";
    delimit = " ,";
    for(token = strtok_r(str1, delimit, &endptr); token!= NULL;
        token = strtok_r(NULL, delimit, &endptr)) {
        printf("token = %s\n", token);
    }
}
```

Output

```
token = The
token = summer
token = soldier
token = the
token = sunshine
token = patriot
```

See Also

strtok

Synopsis
#include <string.h>
char *strtok(char * restrict *s1***, const char * restrict** *s2***);**

Purpose
Breaks a string into a sequence of tokens.

Return Value
The **strtok**() function returns a pointer to the first character of a token, or a null pointer if there is no token.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.

Description
A sequence of calls to the **strtok**() function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The seperator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is *not* contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.

The **strtok**() function then searches from there for a character that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and the subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The **strtok** function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved-pointer and behaves as described above.

The implementation shall behave as if no library function calls the **strtok** function.

Example

```
/* a sample program that tokenizes the string, "The summer soldirer,
the sunshine patriot" with spaces and commas being the delimiters.*/
#include <stdio.h>
#include <string.h>

int main() {
    char *token, *str1, *delimit;
```

```
    str1 = "The summer soldier, the sunshine patriot";
    delimit = " ,";
    for(token = strtok(str1, delimit); token!= NULL;
        token = strtok(NULL, delimit)) {
        printf("token = %s\n", token);
    }
}
```

Output

```
token = The
token = summer
token = soldier
token = the
token = sunshine
token = patriot
```

See Also

strxfrm

Synopsis
#include <string.h>
size_t strxfrm(char * restrict *s1***, const char * restrict** *s2***, size_t** *n***);**

Purpose
Transform a string and place it in an array.

Return Value
The **strxfrm**() function returns the length of the transformed string (not including the terminating null character). If the value returned is **n** or greater, the contents of the array pointed to by **s1** are indeterminate.

Parameters
s1 Pointer to a character argument.
s2 Pointer to a character argument.
n Unsigned integer argument.

Description
The **strxfrm**() function transforms the string pointed to by **s2** and places the resulting string into the array pointed to by **s1**. The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings. No more than **n** characters are placed into the resulting array pointed to by **s1**, including the terminating null character, if **n** is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Example
Output
See Also

Chapter 20

Date and Time Functions — <time.h>

The header **time.h** defines one macro, and declares several types and functions for manipulating time. Many functions deal with a *calender time* that represents the current date (according to the Gregorian calender) and time. Some functions deal with *local time*, which is the calender time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local timezone and Daylight Saving Time are implementation-defined.

The macro defined is **CLOCKS_PER_SEC** which expands to a constant expression with type **clock_t** described below, and which is the number per second of the value returned by the **clock** function.

The types declared are **size_t**;

clock_t

and

time_t

which are arithmetic types capable of representing times; and

struct tm

which holds the components of a calender time, called the *broken-down time*.

The **tm** structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.

int tm_sec; // *seconds after the minute—[0,60]*
int tm_min; // *minutes after the hour—[0,59]*
int tm_hour; // *hours since midnight—[0,23]*
int tm_mday; // *day of the month—[1,31]*
int tm_mon; // *months since January—[0,11]*
int tm_year; // *years since 1900*
int tm_wday; // *days since Sunday—[0,6]*
int tm_yday; // *days since January 1—[0,365]*

int tm_isdst; // *Daylight Saving Time flag*

The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

Public Data
None.

Functions
The following functions are implemented using the **time** header.

Functions	Description
asctime()	Converts broken-down time into a string.
clock()	Determines the processor time used.
ctime()	Converts from calendar time to local time.
difftime()	Computes the difference between two calender times.
gmtime()	Converts from calendar time to broken time.
localtime()	Converts from calendar time to broken-down time.
mktime()	Converts broken-down time to calender time.
strftime()	Places characters into an array.
time()	Determines the current calendar time.

Macros
The following macros are defined for the **time** header.

Macro	Description
CLOCKS_PER_SEC	Holds the value of number per second of the value returned.

Declared Types
The following types are declared in the **time** header.

Declared Types	Description
size_t	Unsigned integer.
clock_t	Arithmetic type capable of representing times.
time_t	Arithmetic type capable of representing times.
struct tm	Structure which holds various date and time elements.

Portability
This header has no known portability problem.

asctime

Synopsis
#include <time.h>
char * asctime(const struct tm ***timeptr*****);**

Purpose
Converts broken-down time into a string.

Return Value
The **asctime**() function returns a pointer to the string.

Parameters
timeptr Pointer to a structure of type tm

Description
The **asctime**() function converts the broken-down time in the structure pointed to by *timeptr* into a string in the form

```
   Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
char *asctime(const struct tm *timeptr)
{
     static const char wday_name[7][3] = {
          "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
     };
     static const char mon_name[12][3] = {
          "Jan", "Feb", "Mar", "Apr", "May", "Jun",
          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
     };
     static char result[26];

     sprintf(result, "%.3s %.3s%3d %.2d:%.2d%.2d %d\n",
          wday_name[timeptr->tm_wday],
          mon_name[timeptr->tm_mon],
          timeptr->tm_mday, timeptr->tm_hour,
          timeptr->tm_min, timeptr->tm_sec,
          1900 + timeptr->tm_year;
     return result;
}
```

Example 1

```
/* a sample program that return local time. */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm *ptr;
```

```
    time_t lt;

    lt = time(NULL);       //return system time
    printf("time(NULL) = %d\n", lt);
    ptr = localtime(&lt);  // return time in the form of tm structure
    printf(asctime(ptr));
}
```

Output

```
time(NULL) = 941830207
Fri Nov  5 14:30:07 1999
```

Example 2

```
/* a sample program that return local time. */
#include <stdio.h>
#include <time.h>
#define SIZE 26

int main() {
    struct tm *ptr;
    time_t lt;
    char buf[26];

    lt = time(NULL);       //return system time
    printf("time(NULL) = %d\n", lt);
    ptr = localtime(&lt);  // return time in the form of tm structure
    asctime_r(ptr,buf,SIZE);
    printf(buf);
}
```

Output

```
time(NULL) = 941830207
Fri Nov  5 14:30:07 1999
```

See Also

clock

Synopsis
#include <time.h>
clock_t clock(*void***);**

Purpose
Determines the processor time used.

Return Value
The **clock**() function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the **clock**() function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, the function returns the value **(clock_t) – 1**.

Parameters
No argument.

Description
The **clock**() function determines the processor time used.

Example

```
/* a sample program that displays the current execution time. */
#include <stdio.h>
#include <time.h>
void elapsed_time(void);

int main() {
    int i;

    clock();    /// first call clock
    for(i = 0; i<100000;i++) {};
    elapsed_time();
}

void elapsed_time(void) {
    clock_t time;

    time = clock();
    printf("Elapsed time: %d micro seconds or %f seconds.\n",time,
          (double)time/CLOCKS_PER_SEC);  //  return the time between first call
                                         // and this call .
}
```

Output

```
Elapsed time: 30000 microsecs.
```

ctime

Synopsis
#include <time.h>
char * ctime(const time_t **timer****);**

Purpose
Converts from calendar time to local time.

Return Value
The **ctime**() function returns the pointer returned by the **asctime**() function with that broken-down time as the argument.

Parameters
timer Variable pointer used to hold a calender time.

Description
The **ctime**() function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to

asctime(**localtime**(**timer**))

Example

```
/* a sample program that return local time. */
#include <stdio.h>
#include <time.h>

int main() {
    time_t *timer;

    timer = malloc(sizeof(time_t));
    *timer = time(NULL);       //return system time
    printf("%s", ctime(timer));
}
```

Output

```
Fri Nov  5 14:30:09 1999
```

See Also

difftime

Synopsis
#include <**time.h**>
double difftime(time_t *time1,* **time_t** *time0***);**

Purpose
Computes the difference between two calender times.

Return Value
The **difftime**() function returns the difference expressed in seconds as a **double**.

Parameters
time1 Variable used to hold a calender time.
time0 Variable used to hold a calender time.

Description
The **difftime**() function computes the difference between two calender times: *time1 – time0*.

Example

```
/* a sample program that returns the difference in
seconds between two calendar times. */
#include <stdio.h>
#include <time.h>

int main() {
    time_t time0, time1;
    int i;

    time0 = time(NULL);       /* return system time */
    for(i = 0;i<100000;i++) {};
    time1 = time(NULL);
    printf("the 100000 loops time is %f sec\n",difftime(time1,time0));
}
```

Output

```
the 100000 loops time is 8.000000 sec
```

See Also

gmtime

Synopsis
#include <time.h>
struct tm * gmtime(const time_t **timer****);**

Purpose
Converts from calendar time to broken time.

Return Value
The **gmtime**() function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

Parameters
timer Variable pointer used to hold a calender time.

Description
The **gmtime**() function converts the calendar time pointed to by *timer* into a broken-down time, expressed as UTC.

Example

```
/* a sample program that prints both the local time and
the UTC of the system. */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm *local_time, *gm;
    time_t t;

    t = time(NULL);
    local_time = localtime(&t);
    printf("Local time and date: %s\n",asctime(local_time));
    gm = gmtime(&t);
    printf("Coordinated Universal Time and date: %s\n",asctime(gm));
}
```

Output

```
Local time and date: Fri Nov  5 14:30:18 1999

Coordinated Universal Time and date: Fri Nov  5 19:30:18 1999
```

See Also

localtime

Synopsis
#include <time.h>
char * localtime(const time_t **timer****);**

Purpose
Converts from calendar time to broken-down time.

Return Value
The **localtime**() function returns a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

Parameters
timer Variable pointer used to hold a calender time.

Description
The **localtime**() function converts the calendar time pointed to by *timer* into a broken-down time, expressed as local time.

Example

```
/* a sample program that prints both the local time and
the UTC of the system. */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm *local_time, *gm;
    time_t t;

    t = time(NULL);
    local_time = localtime(&t);
    printf("Local time and date: %s\n",asctime(local_time));
    gm = gmtime(&t);
    printf("Coordinated Universal Time and date: %s\n",asctime(gm));
}
```

Output

```
Local time and date: Fri Nov  5 14:30:19 1999

Coordinated Universal Time and date: Fri Nov  5 19:30:19 1999
```

See Also

mktime

Synopsis
#include <time.h>
time_t mktime(struct tm*timeptr);

Purpose
Converts broken-down time to calender time.

Return Value
The **mktime**() function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value **(time_t) – 1**.

Parameters
timeptr Pointer to a structure of type tm

Description
The **mktime**() function converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr* into a calender time value with the same encoding as that of the values returned by the **time**() function. the original **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the **tm_wday** and **tm_yday** components of the structure are set appropriately, and the other components are set to represent the specified calender time, but with their values forced to the ranges indicated above; the final value of the **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

If the call is successful, a second call to the **mktime**() function with the resulting **struct tm** value shall always leave it unchanged and return the same value as the first call. Furthermore, if the normalized time is exactly representable as a **time_t** value, then the normalized broken-down time and the broken-down time generated by converting the result of the **mktime**() function by a call to **localtime**() shall be identical.

Example

```
/* a sample program that shows the use of mktime() */
#include <stdio.h>
#include <time.h>

int main() {
    struct tm t;
    time_t t_of_day;

    t.tm_year = 1999-1900;
    t.tm_mon = 6;
    t.tm_mday = 12;
    t.tm_hour = 2;
    t.tm_min = 30;
    t.tm_sec = 1;
    t.tm_isdst = 0;
    t_of_day = mktime(&t);
    printf(ctime(&t_of_day));
```

```
}
```

Output

```
Mon Jul 12 03:30:01 1999
```

strftime

Synopsis
#include <time.h>
char strftime(char * restrict *s***, size_t** *maxsize***, const char * restrict** *format***, const struct tm * restrict** *timeptr***);**

Purpose
Places characters into an array.

Return Value
If the total number of resulting characters including the terminating null character is not more than *maxsize*, the **strftime**() function returns the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Parameters
s Character pointer argument.
maxsize Unsigned integer argument.
format Character pointer argument.
timeptr Pointer to structure.

Description
The **strftime**() function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The *format* string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a **%** character, possibly followed by an **E** or **O** modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* characters are placed into the array.

Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the **LC_TIME** category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by **timeptr**, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

%a is replaced by the locale's abbreviated weekday name. [**tm_wday**]

%A is replaced by the locale's full weekday name. [**tm_wday**]

%b is replaced by the locale's abbreviated month name. [**tm_mon**]

%B is replaced by the locale's full month name. [**tm_mon**]

%c is replaced by the locale's appropriate date and time representation.

%C is replaced by the year divided by 100 and truncated to an integer, as a decimal number **(00–99)**. [**tm_year**]

%d is replaced by the day of the month as a decimal number **(01–31)**. [**tm_mday**]

%D is equivalent to "%m\%d\%y". [**tm_mon**, **tm_mday**, **tm_year**]

%e is replaced by day of the month as a decimal number **(01–31)**; a single digit is preceded by a space. [**tm_mday**]

%F is equivalent to "%Y − %m − %d" (the ISO 8601 date format). [**tm_year**, **tm_mon**, **tm_mday**]

%g is replaced by the last two digits of the week-based year (see below) as a decimal number **(00–99)**. [**tm_year**, **tm_wday**, **tm_yday**]

%G is replaced by the week-based year (see below) as a decimal number (e.g., 1997). [**tm_year**, **tm_wday**, **tm_yday**]

%h is equivalent to **" %b "**. [**tm_mon**]

%H is replaced by the hour (24-hour clock) as a decimal number **(01–23)**. [**tm_hour**]

%I is replaced by the hour (12-hour clock) as a decimal number **(01–12)**. [**tm_hour**]

%j is replaced by the day of the year as a decimal number **(001–366)**. [**tm_yday**]

%m is replaced by the month as a decimal number **(01–12)**. [**tm_mon**]

%M is replaced by the minute as a decimal number **(00–59)**. [**tm_min**]

%n is replaced by a new-line character.

%p is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock. [**tm_hour**]

%r is replaced by the locale's 12-hour clock time. [**tm_hour**, **tm_min**, **tm_sec**]

%R is equivalent to **"%H:%M"**. [**tm_hour**, **tm_min**]

%S is replaced by the second as a decimal number **(00–60)**. [**tm_sec**]

%t is replaced by a horizontal-tab character.

%T is equivalent to **"%H:%M:%S"** (the ISO 8601 time format). [**tm_hour**, **tm_min**, **tm_sec**]

%u is replaced by the ISO 8601 weekday as a decimal number **(1–7)**, where Monday is 1. [**tm_wday**]

%U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number **(00–53)**. [**tm_year**, **tm_wday**, **tm_yday**]

%V is replaced by the ISO 8601 week number (see below) as a decimal number **(01–53)**. [**tm_year**, **tm_wday**, **tm_yday**]

%w is replaced by the weekday as a decimal number **(0–6)**, where Sunday is 0. [**tm_wday**]

%W is replaced by the week number of the year (the first Monday as the first day of week 1) s a decimal number **(00–53)**. [**tm_year**, **tm_wday**, **tm_yday**]

%x is replaced by the locale's appropriate date representation.

%X is replaced by the locale's appropriate time representation.

%y is replaced by the last 2 digits of the year as a decimal number **(00–99)**. [**tm_year**]

%Y is replaced by the year as a decimal number (e.g., **1997**). [**tm_year**]

%z is replaced by the offset from UTC in the ISO 8601 format "**-0430**" (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [**tm_isdst**]

%Z is replaced by the locale's time zone name or abbreviation, or by no characters if no time zone is determinable. [**tm_isdst**]

%% is replaced by **%**.

Some conversion specifiers can be modified by the inclusion of an **E** or **O** modifier character to indicate an alternative format or specification. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

%Ec is replaced by the locale's alternative date and time representation.

%EC is replaced by the name of the base year (period) in the locale's alternative representation.

%Ex is replaced by the locale's alternative date representation.

%EX is replaced by the locale's alternative time representation.

%Ey is replaced by the offset from **%EC** (year only) in the locale's alternative representation.

%EY is replaced by the locale's full alternative year representation.

%Od is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero).

%Oe is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading spaces).

%OH is replaced by the hour (24-hour clock), using the locale's alternative numeric symbols.

%OI is replaced by the hour (12-hour clock), using the locale's alternative numeric symbols.

%Om is replaced by the month, using the locale's alternative numeric symbols.

%OM is replaced by the minutes, using the locale's alternative numeric symbols.

%OS is replaced by the seconds, using the locale's alternative numeric symbols.

%Ou is replaced by the ISO 8601 weekday as a number in the locale's alternative representation, where Monday is 1.

%OU is replaced by the week number, using the locale's alternative numeric symbols.

%OV is replaced by the ISO 8601 week number, using the locale's alternative numeric symbols.

%Ow is replaced by the weekday as a number, using the locale's alternative numeric symbols.

%OW is replaced by the week number of the year, using the locale's alternative numeric symbols.

%Oy is replaced by the last 2 digits of the year, using the locale's alternative numeric symbols.

%g, **%G**, and **%V** give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, **%G** is replaced by **1998** and **%V** is replaced by **53**. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, **%G** is replaced by **1998** and **%V** is replaced by **1**.

If a conversion specifier is not one of the above, the behavior is undefined.

In the **"C"** locale, the **E**and **O** modifiers are ignored and the replacement strings for the following specifiers are:

%a the first three characters of **%A**.

%A one of **"Sunday", "Monday", ... ,"Saturday"**.

%b one of the first three characters of **%B**.

%B one of **"January", "February", ... , "December"**.

%c equivalent to **"%A %B %d %T %Y"**.

%p one of **"am"** or **"pm"**.

%r equivalent to **"%I:%M:%S:%p"**.

%x equivalent to **"%A:%B:%d:%Y"**.

%X equivalent to **"%T"**.

%Z implementation-defined.

Example

```
/* a sample program that displays current time. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    struct tm *ptr;
    time_t ltime;
    char str[80];

    ltime = time(NULL); /* get current calendar time */
    ptr = localtime(&ltime);  // return time in the form of tm structure
    strftime(str,80,"It is now %H:%M %p.",ptr);
    printf("%s\n",str);
}
```

Output

```
It is now 14:30 PM.
```

time

Synopsis
#include <time.h>
time_t time(time_t *timer***);**

Purpose
Determines the current calender time.

Return Value
The **time**() function returns the implementation's best approximation to the current calendar time. The value `(time_t)-1` is returned if the calendar time is not available. If *timer* is not a null pointer, the return value is also assigned to the object it points to.

Parameters
timer Variable used to hold a calender time.

Description
The **time**() function determines the current calendar time. The encoding of the value is unspecified.

Example

```
See asctime().
```

Output

Chapter 21

Extended Multibyte and Wide Character Functions — <wchar.h>

The header **wchar.h** declares four data types, one tag, four macros, and many functions.

The types declared are **wchar_t** and **size_t** described in Chapter 16;

mbstate_t

which is an object type other than an array type that can hold the conversion state informantion necessary to convert between sequences of multibyte characters and wide characters;

wint_t described in Chapter 22;

and

struct tm

which is declared as an incomplete structure type.
the contents of which are described in Chapter 20;

The macros defined are

WCHAR_MAX

which is the maximum value representable by an object of type **wchar_t**;

WCHAR_MIN

which is the minimum value representable by an object of type **wchar_t**; and

WEOF described in Chapter 22.

The functions declared are grouped as follows:

— Functions that perform input and output of wide characters, or multibyte characters, or both;

— Functions that provide wide-string numeric conversions;

— Functions that perform general wide-string manipulation;

— Functions for wide-string date and time conversions; and

— Functions that provide extended capabilities for conversion between multibyte and wide-character sequences.

Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

Public Data
None.

Functions
The following functions are implemented using the **wchar** header.

Functions	Description
btowc()	Determines if character is multibyte character.
fgetwc()	Obtains wide character and advances one position.
fgetws()	Reads wide characters from stream.
fputwc()	Writes the wide character to the output stream.
fputws()	Writes a wide string to stream.
fwide()	Determines orientation of stream.
getwc()	Gets wide character and advances pointer.
getwchar()	Gets wide character and advances file position indicator.
mbrlen()	Determines the number of bytes in a multibyte character.
mbrtowc()	Determines the number of bytes needed to complete the next multibyte character.
mbsinit()	Determines if pointed-to object describes an initial conversion state.
mbsrtowcs()	Converts a sequence of multibyte charaters from an array into a sequence.
putwc()	Writes wide character to output stream.
putwchar()	Writes wide character to output stream.
ungetwc()	Pushes wide character back onto input stream.
wcrtomb()	Determines the number of bytes needed to represent the next multibyte character.
wcscat()	Appends a copy of one wide string to another wide string.
wcschr()	Locates first occurance of a wide character in a string.
wcscmp()	Compares two wide strings.
wcscoll()	Compares two wide strings.
wcscpy()	Copies a wide string to an array.
wcscspn()	Computes the length of the maximum initial segment of a wide string.
wcsftime()	Places wide characters into an array.
wcslen()	Computes the length of a wide string.
wcsncat()	Appends wide characters from an array to a wide string.
wcsncmp()	Compares wide characters between two arrays.

wcsncpy()	Copies wide characters from one array to another array.
wcspbrk()	Locates first occurance of a wide character from one string in another string.
wcsrchr()	Locates last occurance of a wide character in a wide string.
wcsrtombs()	Converts sequence of wide characters from array into a sequence of multibyte characters.
wcsspn()	Computes the length of the maximum initial segment of a wide string.
wcsstr()	Locates the first sequence found in one wide string in the other wide string.
wcstod()	Converts initial portion of wide string to double, float, and long double.
wcstok()	Breaks a wide string into a sequence of tokens.
wcstol()	Converts wide string to long int, long long int, unsigned long int, or unsigned long long int.
wcsxfrm()	Transforms wide string and places it in an array.
wctob()	Determines if character corresponds to member of extended character set.
wmemchr()	Locates the first occurance of a specified wide character in a stream.
wmemcmp()	Compares the first wide characters in two objects.
wmemcpy()	Copies wide characters from one object to another.
wmemmove()	Copies wide characters from one object to another.
wmemset()	Copies a single value into specified number of locations in an object.

Macros

The following macros are defined for the **wchar** header.

Macro	Description
WCHAR_MAX	Maximum value representable by object of type **wchar_t**.
WCHAR_MIN	Minimum value representable by object of type **wchar_t**.
WEOF	Used to indicate *end-of-file*.

Declared Types

The following types are declared in the **wchar** header.

Declared Types	Description
wchar_t	int-can hold codes for all the extended character set.
size_t	unsigned int
wint_t	int-holds values corresponding to members of the extended set.
mbstate_t	object type-holds conversion state information.
struct tm	incomplete structure type-holds the components of calender time.

Portability

The following functions are not supported in the current release of Ch: **fwprintf**(), **fwscanf**(), **swprintf**(), **swscanf**(), **vfwprintf**(), **vfwscanf**(), **vswprintf**(), **vswscanf**(), **vwprintf**(), **vwscanf**(), **wprintf**(), **wscanf**().

btowc

Synopsis
#include <wchar.h>
#include <stdio.h>
wint_t btowc(int *c***);**

Purpose
Determines if character is multibyte character.

Return Value
The **btowc**() function returns **WEOF** if *c* has the value **EOF** or if **(unsigned char)** *c* does not constitute a valid (one-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of the character.

Parameters
c Integer argument.

Description
The **btowc**() function determines whether *c* constitutes a valid (one-byte) multibyte character in the initial shift state.

Example

```
/*  This is an example of btowc(int).  The function determines whether c is a
    valid (one-byte) character in the initial shift state.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int c;
    wint_t retval;
    c='r';

    retval = btowc(c);
    if(retval != WEOF) {
     printf("btowc() = ");
     fputwc(c, stdout);
     printf("\n");
    }
    else
        printf("Error\n");
}
```

Output

```
btowc() = r
```

See Also

fgetwc

Synopsis
#include <wchar.h>
#include <stdio.h>
wint_t fgetwc(FILE ***stream*****);**

Purpose
Obtains wide character and advances one position.

Return Value
The **fgetwc**() function returns the next wide character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetwc**() returns **WEOF**. If a read error occurs, the error indicator for the stream is set and **fgetwc**() returns **WEOF**. If an encoding error occurs (including too few bytes), the value of the macro **EILSEQ** is stored in **errno**() and **fgetwc**() returns **WEOF**.

Parameters
stream FILE pointer argument.

Description
If a next wide character is present from the input stream pointed to by *stream*, the **fgetwc**() function obtains that wide character and advances the associated file position indicator for the stream (if defined).

Example

```
/*This is an example of how to use the function fgetwc(FILE *).  It will
  retrieve a wide character from an input stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;

    printf("Displays the first character from the file, ""wctype.c""\n");
    if((stream = fopen("fgetwc.c", "r")) != NULL) {
        printf("fgetwc(stream) = ");
        fputwc( fgetwc(stream), stdout);
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
Displays the first character from the file, wctype.c
fgetwc(stream) = /
```

See Also

fgetws

Synopsis
#include <wchar.h>
#include <stdio.h>
wchar_t fgetws(wchar_t * restrict *s***, int** *n***, FILE * restrict** *stream***);**

Purpose
Reads wide characters from stream.

Return Value
The **fgetws**() function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read or encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Parameters
s Integer pointer argument.
n Integer argument.
stream FILE pointer argument.

Description
The **fgetws**() function reads at most one less than the number of wide characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional wide characters are read after a new-line wide character (which is retained) or after end-of-file. A null wide character is written immediately after the last wide character read into the array.

Example

```
/* This is an example of the function fgetws(wchar_t*, int, FILE *).
   This will retrieve an entire string of the max length specifed by the user
   from an input stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    int i;
    wchar_t str[100];

    i =10;
    printf("Retrieving a string of length %d from \"Makefile\".\n", i);
    if((stream = fopen("Makefile", "r")) != NULL) {
     fgetws(str, i, stream);
        fputws(str, stdout);
        fputs("\n", stdout);
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
Retrieving a string of length 10 from "Makefile".
default a
```

See Also

fputwc

Synopsis
#include <wchar.h>
#include <stdio.h>
wint_t fputwc(wchar_t *c***, FILE** **stream**);**

Purpose
Writes the wide character to the output stream.

Return Value
The **fputwc**() function returns the next wide character written. If a write error occurs, the error indicator for the stream is set and **fputwc**() returns **WEOF**. If an encoding error occurs, the value of the macro **EILSEQ** is stored in **errno**() and **fputwc**() returns **WEOF**.

Parameters
c Integer argument.
stream FILE pointer argument.

Description
The **fputwc**() function writes the wide character specified by *c* to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Example

```
/* This is an example of the function fputwc(wint_t, FILE *).  This will insert
   a wide character into the specifed output destination.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wint_t wc;
    char *name;

    if((name = tmpnam(NULL)) == NULL)
    printf("Error in creating temp file.\n");
    else {
wc = 'i';
     printf("Will write the character 'i' to to a temporary file.\n");
     if((stream = fopen(name, "w")) != NULL) {
            printf("fputwc(stream) = ");
            fputwc(wc,stream);
      fputwc(wc,stdout);
            printf("\n");
        }
      else
    printf("Error in opening file!\n");
    }
```

```
    fclose(stream);
    remove(name);
}
```

Output

```
Will write the character 'i' to to a temporary file.
fputwc(stream) = i
```

See Also

fputws

Synopsis
#include <wchar.h>
#include <stdio.h>
wchar_t fputws(const wchar_t * restrict *s***, FILE * restrict** *stream***);**

Purpose
Writes a wide string to stream.
Return Value
The **fputws**() function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

Parameters
s Integer pointer argument.
stream FILE pointer argument.

Description
The **fputws**() function writes the wide string pointed to by *s* to the stream pointed to by *stream*. The terminating null wide character is not written.

Example

```
/* This is an example for the function fputws(const wchar_t *, FILE *).
   It will place a wide character string into the specifed file.
   If the file does not exist, one will be created.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wchar_t line[100];
    char *temp, *name;
    int len;

    if((name = tmpnam(NULL)) == NULL)
     printf("temp file name could not be made. \n");
    else {
     temp ="Have a Nice Day!";
     len = strlen(temp);
     mbstowcs(line, temp, len);
     if((stream = fopen(name, "w")) != NULL) {
            fputws(line, stream);
            fputws(line, stdout);
            fputs("\n", stdout);
     }
        else
            printf("Error in opening file!\n");
    }
    fclose(stream);
    remove(name);
}
```

Output

```
Have a Nice Day!
```

See Also

fwide

Synopsis
#include <wchar.h>
#include <stdio.h>
int fwide(FILE ***stream*****, int** ***mode*****);**

Purpose
Determines orientation of stream.

Return Value
The **fwide**() function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

Parameters
stream FILE pointer argument.
mode Integer argument.

Description
The **fwide**() function determines the orientation of the stream pointed to by *stream*. If *mode* is greater than zero, the function first attempts to make the stream wide oriented. If *mode* is less than zero, the function first attemps to make the stream byte oriented. Otherwise, *mode* is zero and the function does not alter the orientation of the stream.

Example

```
/*  This is an example of the function fwide(FILE *, int).  The function will
    alter the stream according to the desired mode.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    int c,retval;

    c = 11;
    if((stream = fopen("fwide.c", "r")) != NULL)
    {
        retval = fwide(stream, c);
        if(retval < 0)
            printf("Stream has byte orientation.\n");
        else if(retval > 0)
            printf("Stream has wide orientation.\n");
        else
            printf("Stream has no orientation.\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
Stream has wide orientation.
```

See Also

getwc

Synopsis
#include <wchar.h>
#include <stdio.h>
wint_t getwc(FILE *stream);

Purpose
Gets a wide character and advances the pointer.

Return Value
The **getwc**() function returns the next wide character from the input stream pointed to by *stream*, or **WEOF**.

Parameters
stream FILE pointer argument.

Description
The **getwc**() function is equivalent to **fgetwc**(), except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Example

```
/* This is an example for the function getwc(FILE *).  This retrieves a wide
   character from the input stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    int i;
    wint_t ch;

    printf("Retrieves the first 10 characters from the file ""wctype.c"".\n");
    if((stream = fopen("getwc.c", "r")) != NULL) {
  for(i =0; i<10; i++) {
          ch = getwc(stream);
         fputwc(ch, stdout);
     }
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
Retrieves the first 10 characters from the file wctype.c.
/* This is
```

See Also

getwchar

Synopsis
#include <wchar.h>
wint_t getwchar(*void***);**

Purpose
Gets a wide character and advances the file position indicator.

Return Value
The **getwchar**() function returns the next wide character from the input stream pointed to by **stdin**, or **WEOF**.

Parameters
void Void argument.

Description
The **getwchar**() function is equivalent to **getwc** with the argument **stdin**.

Example

```
/* This is an example for getwchar(void).  This function will retrieve a wide
   character from the input stream, such as from stdin.  This program is
   executed by: getwchar.c < getwchar.in
   The input file getwchar.in contains: 'i'
*/
#include <wchar.h>

int main() {
    int i, ch;
    char buffer[80];

    printf("Enter a character. \n");
    for(i =0; (i<80) && (ch !='\n'); i++) {
     ch = getwchar();
        buffer[i]= (char)ch;

    }
    printf("You entered: %s\n",buffer);
}
```

Output

```
Enter a character.
You entered: i
```

See Also

mbrlen

Synopsis
#include <wchar.h>
size_t mbrlen(const char * restrict *s***, size_t** *n***, mbstate_t * restrict** *ps***);**

Purpose
Determines the number of bytes in a multibyte character.

Return Value
The **mbrlen**() function returns a value between zero and *n*, inclusive, $(\mathbf{size\_t}) - \mathbf{2}$, or $(\mathbf{size\_t}) - \mathbf{1}$.

Parameters
s Integer pointer argument.
n Unsigned integer argument.
ps Object type pointer argument.

Description
The **mbrlen**() function is equivalent to the call:

$$\mathbf{mbrtowc}(\mathbf{NULL}, s, n, ps! = \mathbf{NULL}?ps : \&\mathbf{internal})$$

where **internal** is the **mbstate_t** object for the **mbrlen**() function, except that the expression designated by *ps* is evaluated only once.

Example

```
/* This is an example of the function mbrlen(const char*, size_t, mbstate_t).
   This returns a value between zero and length.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str;
    size_t retval, length;
    mbstate_t *ps;

    str = "hello";
    ps = NULL;
    length = 5;
    retval = mbrlen(str, length, ps);
    if(retval != (size_t) -1) {
     printf("Number of bytes of next character:  %d\n", retval);
    }
    else
        printf("Error\n");
}
```

Output

```
Number of bytes of next character:  1
```

See Also
mbrtowc()

mbrtowc

Synopsis
#include <wchar.h>
size_t mbrtowc(const char * restrict *pwc***, const char * restrict** *s***, size_t** *n***, mbstate_t * restrict** *ps***);**

Purpose
Determines the number of bytes needed to complete the next multibyte character.

Return Value
The **mbrtowc**() function returns the first of the following that applies (given the current conversion state)

0 if the next *n* or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

positive if the next *n* or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

$(\mathbf{size\_t}) - \mathbf{2}$ if the next *n* bytes contribute to an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed (no value is stored).

$(\mathbf{size\_t}) - \mathbf{1}$ if an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **EILSEQ** is stored in **errno**(), and the conversion state is undefined.

Parameters
pwc Character pointer argument.
s Character pointer argument.
n Unsigned integer argument.
ps Integer argument.

Description
If *s* is a null pointer, the **mbrtowc**() function is equivalent to the call:

mbrtowc(NULL, "", 1, *ps***)**

In this case, the values of the parameters *pwc* and *n* are ignored.

If *s* is not a null pointer, the **mbrtowc**() function inspects at most *n* bytes beginning with the byte pointed to by *s* to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is completed, it determines the value of the corresponding wide character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding wide character is the ull wide character, the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function mbrtowc(const char*, size_t, mbstate_t).
   the mbrtowc function inspects at most n bytes beginning with the byte
   pointed  to by s to determine the number of bytes needed to complete the
   next multibyte character
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str2;
    wchar_t *str;
    size_t retval, length;
    mbstate_t *ps;

    str2 = "hello";
    ps = NULL;
    length = 5;
    retval = mbrtowc(str, str2, length, ps);
    if(retval != (size_t) -1)
     printf("Number of bytes of the next character:  %d\n", retval);
    else
        printf("error\n");
}
```

Output

```
Number of bytes of the next character:  1
```

See Also
mbrtowc

mbsinit

Synopsis
#include <wchar.h>
#include <stdio.h>
int mbsinit(const mbstate_t **ps***);**

Purpose
Determines if pointed-to object describes an initial conversion state.

Return Value
The **mbsinit**() function returns nonzero if *ps* is a null pointer or if the pointed-to object describesan initial state; otherwise it returns zero.

Parameters
ps Integer pointer argument.

Description
If *ps* is not a null pointer, the **mbsinit**() function determines whether the pointed-to **mbstate_t** object describes an initial conversion state.

Example

```
/* This is an example of the function mbsinit(mbstate_t).
   This function determines whether the object pointed to by ps describes an
   initial conversion state.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int retval;
    mbstate_t *ps;

    ps = NULL;
    retval = mbsinit(ps);
    if(retval != 0) {
     printf("mbsinit(ps) returns nonzero if ps is a null pointer \n");
     printf("or ps is pointing to an initial conversion state.\n");
     printf("mbsinit(ps)= %d\n", retval);
    }
    else
        printf("Error \n");
}
```

Output

```
mbsinit(ps) returns nonzero if ps is a null pointer
or ps is pointing to an initial conversion state.
mbsinit(ps)= 1
```

See Also

mbsrtowcs

Synopsis
#include <wchar.h>
size_t mbsrtowcs(wchar_t * restrict *dst***, const char ** restrict** *src***, size_t** *len***, mbstate_t * restrict** *ps***);**

Purpose
Converts a sequence of multibyte charaters from an array into a sequence.

Return Value
If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **mbsrtowcs**() function stores the value of the macro **EILSEQ** in **errno**() and returns (**size_t**) **- 1**; the conversion state is undefined. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null (if any).

Parameters
dst Character pointer argument.
src Character pointer argument.
len Unsigned integer argument.
ps Object type pointer.

Description
The **mbsrtowcs**() function converts a sequence of multibyte characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if *dst* is not a null pointer) when *len* codes have been stored into the array pointed to by *dst* Each conversion takes place as if by call to the **mbrtowc** function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if *dst* is not a null pointer, the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function mbsrtowcs(wchar_t*, const char**,
                        size_t, mbstate_t). Converts a sequence of characters
   into a sequence of corresponding wide-characters.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100];
    const char *str2;
    size_t retval, length;
```

```
    mbstate_t *ps;

    str2 = "hello";
    ps = NULL;
    length = 5;
    retval = mbsrtowcs(str1, &str2,length, ps);
    if(retval != (size_t) -1) {
     printf("Number of multibye characters");
     printf(" successfully converted: %d\n",retval);
    }
    else
        printf("Error\n");
}
```

Output

```
Number of multibye characters successfully converted: 5
```

See Also

putwc

Synopsis
#include <wchar.h>
#include <stdio.h>
wint_t putwc(wchar_t *c***, FILE** **stream**);**

Purpose
Writes a wide character to an output stream.

Return Value
The **putwc**() function returns the next wide character written, or **WEOF**.

Parameters
c Integer argument.
stream FILE pointer argument.

Description
The **putwc**() function is equivalent to **fputwc**(), except that if it is implemented as a macro, it may evaluate *stream* more than once, so that argument should never be an expression with side effects.

Example

```
/* This is an example for putwc(wint_t, FILE*).  This function will write a
  character to a stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wint_t wc;
    char *name;

    wc = 'i';
    if((name = tmpnam(NULL)) == NULL)
         printf("Error in creating temp file.\n");
    else {
      fputs("Will write the letter '",stdout);
     fputwc(wc, stdout);
     fputs("' to a temporary file.\n", stdout);
     if((stream = fopen(name, "w")) != NULL) {
            printf("putwc(stream) = %c\n", putwc(wc,stream));
        }
     else
            printf("Error in opening file!\n");
    }
    fclose(stream);
    remove(name);
}
```

Output

```
Will write the letter 'i' to a temporary file.
putwc(stream) = i
```

See Also

putwchar

Synopsis
#include <wchar.h>
wint_t putwchar(wchar_t *c***);**

Purpose
Writes a wide character to an output stream.

Return Value
The **putwchar**() function returns the character written, or **WEOF**.

Parameters
c Integer argument.

Description
The **putwchar**() function is equivalent to **putwc**() with the second argument **stdout**.

Example

```
/* This is an example of the function fputwchar(wint_t).  This will write a
   wide character to stdout.
*/
#include <wchar.h>

int main() {
    wint_t wc;
    wc = 's';

    printf("Will place the character '");
    fputwc(wc, stdout);
    printf("' on to stdout.\n");
    printf("putwchar(wc) = ");
    putwchar(wc);
    printf("\n");
    }
}
```

Output

```
Will place the character 's' on to stdout.
putwchar(wc) = s
```

See Also

ungetwc

Synopsis
#include <wchar.h>
#include <stdio.h>
wint_t ungetwc(wint_t *c***, FILE** **stream***);*

Purpose
Pushes a wide character back onto an input stream.

Return Value
The **ungetwc**() function returns the character pushed back, or **WEOF** if the operation fails.

Parameters
c Integer argument.
stream FILE pointer argument.

Description
The **ungetwc**() function pushes the wide character specified by *c* back onto the input stream pointed to by *stream*. Pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file positioning function (**fseek**(), **fsetpos**(), or **rewind**()) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

One wide character of pushback is guaranteed, even if the call to the **ungetwc**() function follows just after a call to a formatted wide character input function **fwscanf**(), **vfwscanf**(), **vwscanf**(), or **wscanf**(). If the **ungetwc**() function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro **WEOF**, the operation fails and the input stream is unchanged.

A successful call to the **ungetwc**() function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back. For a text or binary stream, the value of its file position indicator after a successful call to the **ungetwc**() function is unspecified until all pushed-back wide characters are read or discarded.

Example

```
/* This is an example of the function ungetwc(wint_t, FILE *).  This pushes
   a character back onto the stream.
*/
#include <wchar.h>

int main() {
    FILE *stream;
    wint_t wc;
```

```
    if((stream = fopen("ungetwc.c", "r")) != NULL) {
        wc = fgetwc(stream);
        printf("ungetwc(wc,stream) = ");
        fputwc( ungetwc(wc,stream), stdout);
        printf("\n");
    }
    else
        printf("Error in opening file!\n");
    fclose(stream);
}
```

Output

```
ungetwc(wc,stream) = /
```

See Also

wcrtomb

Synopsis
#include <wchar.h>
size_t wcrtomb(const char * restrict *s*, wchar_t *n*, mbstate_t * restrict *ps*);

Purpose
Determines the number of bytes needed to represent the next multibyte character.

Return Value
The **wcrtomb**() function returns the number of bytes stored in the array object (including any shift sequences). When *wc* in not a valid wide character, an encoding error occurs: the function stores the value of the macro **EILSEQ** in **errno**() and returns $(size\_t)$ **- 1**; the conversion state is undefined.

Parameters
s Character pointer argument.
n Character argument.
ps Integer pointer argument.

Description
If *s* is a null pointer, the **wcrtomb**() fucntion is equivalent to the call:

$$\mathbf{wcrtomb}(\mathbf{buf}, \mathbf{L'\backslash 0'}, \mathbf{1}, ps)$$

where **buf** is an internal buffer.

If *s* is not a null pointer, the **wcrtomb**() function determines the number of bytes needed to represent the multibyte character given by *wc* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most $\mathbf{MB\_CUR\_MAX}$ bytes are stored. If *wc* is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function wcrtomb(char*, wchar_t, mbstate_t).
   The wcrtomb function determines the number  of  bytes  needed  to represent
   the multibyte  character  that corresponds to the wide character given by wc.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *str;
    size_t retval;
    wchar_t c;
    mbstate_t *ps;

    str = "hello";
    c = 'e';
```

```
    ps = NULL;
    retval = wcrtomb(str, c, ps);
    if(retval != (size_t) -1) {
     printf("Number of bytes needed to represent the \n");
     printf("next character:  %d\n", retval);
    }
    else
        printf("Error\n");
}
```

Output

```
Number of bytes needed to represent the
next character:  1
```

See Also

wcscat

Synopsis
#include <wchar.h>
wchar_t wcscat(wchar_t * restrict *s1*, const wchar_t * restrict s2);

Purpose
Appends a copy of one wide string to another wide string.

Return Value
The **wcscat**() returns the value of *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcscat**() function appends a copy of the wide string pointed to by *s2* (including the terminating null wide character) to the end of the wide string pointed to by *s1*. The initial wide character of *s2* overwrites the null character at the end of *s1*.

Example

```
/* This is an example of the function wcscat(wchar_t*, const wchar_t*).  This
   appends a string with another wide character string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100];
    wchar_t str2[100];
    char *line;
    int retval, len;
    line = "Hello";
    len = strlen(line);

    mbstowcs(str1, line, len );
    mbstowcs(str2, line, len);
    fputs("String1: ", stdout);
    fputws(str1, stdout);
    fputs("\nString2: ", stdout);
    fputws(str2, stdout);
    wcscat(str1, str2);
    fputs("\nAfter wide character appending: ", stdout);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
String1: Hello
```

```
String2: Hello
After wide character appending: HelloHello
```

See Also

wcschr

Synopsis
#include <wchar.h>
wchar_t wcschr(const wchar_t *s, wchar_t *c*);

Purpose
Locates first occurance of a wide character in a string.

Description
The **wcschr**() function locates the first occurance of *c* in the wide string pointed to by *s*. The terminating null wide character is considered to be part of the wide string.

Parameters
s Integer pointer argument.
c Integer argument.

Return Value
The **wcschr**() function returns a pointer to the located wide string character, or a null pointer if the wide character does not occur in the wide string.

Example

```
/* This is an example of the function wcschr(const wchar_t*, wchar_t*).
   This function finds a character in a string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str[100], c;
    char *line;
    wchar_t *retval;
    int result, len;
    line = "Hello";
    c = 'l';

    len = strlen(line);
    mbstowcs(str, line, len);
    retval = wcschr(str,c);
    result = retval - str +1;
    if(result <= 0) {
  printf("There is no '");
        fputwc(c, stdout);
        printf("' in this string.\n");
    }
    else {
     printf("The first occurence of '");
        fputwc(c,stdout);
        printf("' is at position: %d\n", result);
    }
}
```

Output

```
The first occurence of 'l' is at position: 3
```

See Also

wcscmp

Synopsis
#include <wchar.h>
int wcscmp(const wchar_t **s1***, const wchar_t** **s2***, size_t** *n***);**

Purpose
Compares two wide strings.

Return Value
The **wcscmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by *s1* is greater than, equal to, or less than the wide string pointed to by *s2*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wcscmp**() function compares the wide string pointed to by *s1* to the wide string pointed to by *s2*.

Example

```
/* This is an example of the function wcscmp(const wchar_t*, const wchar_t*).
   This function compares two strings.  If string1 < string2, then a negative
   number will be returned.  If string1 = string2, zero will be returned.
   If string1 > string2, a positive number will be returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    const char *line, *line2;
    wchar_t str1[100], str2[100];
    int retval,len1, len2;

    line = "hello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcscmp(str1, str2);
    if(retval > 0)
        printf("String1 is greater than String2\n");
    else if(retval == 0)
        printf("String1 is identical to String2\n");
    else if(retval < 0)
     printf("String1 is less than String2\n");
    else
        printf("Error!\n");
}
```

Output

```
String1 is identical to String2
```

See Also

wcscoll

Synopsis
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);

Purpose
Compares two wide strings.

Return Value
The **wcscoll**() function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by *s1* is greater than, equal to, or less than the wide string pointed to by *s2* when both are interpreted as appropriate to the current locale.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcscoll**() function compares the wide string pointed to by *s1* to the wide string pointed to by *s2*, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Example

```
/* This is an example of the function wcscoll(const wchar_t*, const wchar_t*).
   If string1 < string2, a negative number will be returned.
   If string1 = string2, zero will be returned.  If
   string1 > string2, a positive number will be returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    const char *line, *line2;
    wchar_t str1[100], str2[100];
    int retval,len1,len2;

    line = "hello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcscoll(str1, str2);
    if(retval > 0)
        printf("String1 is greater than String2\n");
    else if(retval == 0)
        printf("String1 is identical to String2\n");
    else if(retval < 0)
        printf("String1 is less than String2\n");
    else
        printf("Error!\n");
```

```
}
```

Output

```
String1 is identical to String2
```

See Also

wcscpy

Synopsis
#include <wchar.h>
wchar_t wcscpy(wchar_t * restrict *s1***, const wchar_t * restrict** *s2***);**

Purpose
Copies a wide string to an array.

Return Value
The **wcscpy**() returns the value of *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcscpy**() function copies the wide string pointed to by *s2* (including the terminating null wide character) into the array pointed to by *s1*.

Example

```
/* This is an example of the function wcscpy(wchar_t*, const wchar_t*).  This
   make a copy of the second string and store into the first parameter.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line;
    int retval,len;
    line = "Hello";
    len = strlen(line);

    mbstowcs(str2, line, len);
    wcscpy(str1, str2);
    fputws(str1, stdout);
    fputs("\n",stdout);
}
```

Output

```
Hello
```

See Also

wcscspn

Synopsis
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);

Purpose
Computes the length of the maximum initial segment of a wide string.

Return Value
The **wcscspn**() function returns the length of the segment.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcscspn**() function computes the length of the maximum initial segment of the wide string pointed to by *s1* which consists entirely of wide characters *not* from the wide string pointed to by *s2*.

Example

```
/* This is an example of the function wcscspn(const wchar_t*, const wchar_t*).
   This finds a substring.  The function returns the length of the first
   substring that is made up of characters from string 2.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t retval;
    int len1,len2;
    line = "hello";
    line2 = "ll";

    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line,len1);
    mbstowcs(str2, line2, len2);
    retval = wcscspn(str1, str2);
    printf("Substring starts at position:  %d\n", retval);
}
```

Output

```
Substring starts at position:  2
```

See Also

wcsftime

Synopsis
#include <wchar.h>
size_t wcsftime(wchar_t * restrict *s***, size_t** *maxsize***, const wchar_t * restrict** *format***, const struct tm * restrict** *timeptr***);**

Purpose
Places wide characters into an array.

Return Value
If the total number of resulting wide characters, including the terminating null wide character is not more than *maxsize*, the **wcsftime**() function returns the number of wide characters placed into the array pointed to by *s* not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

Parameters
s Integer pointer argument.
maxsize Unsigned integer argument.
format Integer pointer argument.
timeptr Struct argument.

Description
The **wcsftime**() function is equivalent to the **strftime**() function, except that:

— The argument *s* points to the initial element of an array of wide characters into which the generated output is to be placed.

— The argument *maxsize* indicates the limiting number of wide characters into which the generated output is to be placed.

— The argument *format* is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.

— The return value indicates the number of wide characters.

Example

```
/* This is an example of the function wcsftime(wchar_t*,size_t,const wchar_t*,
    const struct tm*).  This function formats a time string.
*/
#include <wchar.h>
#include <stdlib.h>
#include <time.h>
```

```
int main() {
    size_t retval, maxsize;
    wchar_t str1[100];
    wchar_t format[100];
    char *line;
    struct tm *timeptr;
    int len;
    time_t aclock;
    time(&aclock);
    maxsize = 100;

    line = "%c";
    len = strlen(line);
    mbstowcs(format, line, len);
    timeptr = localtime(&aclock);
    retval = wcsftime(str1, maxsize, format,timeptr );
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
Mon Feb 21 14:56:31 2000
```

See Also

wcslen

Synopsis
#include <wchar.h>
size_t wcslen(const wchar_t *s);

Purpose
Computes the length of a wide string.

Return Value
The **wcslen**() function returns the number of wide characters that precede the terminating null wide character.

Parameters
s Integer pointer argument.

Description
The **wcslen**() function computes the length of the wide string pointed to by *s*.

Example

```
/* This is an example of the function wcslen(const wchar_t*).  This
   returns the length of the string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line;
    wchar_t str1[100];
    size_t retval;
    int len;

    line = "hello";
    len = strlen(line);
    mbstowcs(str1, line, len);
    retval = wcslen(str1);
    printf("Number of characters in str1:  %d\n", retval);
}
```

Output

```
Number of characters in str1:  5
```

See Also

wcsncat

Synopsis
#include <wchar.h>
wchar_t wcsncat(wchar_t * restrict *s1***, const wchar_t * restrict** *s2***, size_t** *n***);**

Purpose
Appends wide characters from an array to a wide string.

Return Value
The **wcsncat**() returns the value of *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wcsncat**() function appends not more than *n* wide characters (a null wide character and those that follow it are not appended) from the array pointed to by *s2* to the end of the wide string pointed to by *s1*. The initial wide character of *s2* overwrites the null wide character at the end of *s1*. A terminating null wide character is always appended to the result.

Example

```
/* This is an example of the function wcsncat(wchar_t*, const wchar_t*, size_t).
    This function appends the specified number of characters from str2 to str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line, *line2;
    wchar_t  *retval;
    size_t count;
    int len1,len2;

    line = "Hello, ";
    line2 ="how are you today?";
    len1 = strlen(line);
    len2 = strlen(line2);
    count =18;
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    wcsncat(str1,str2,count);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
Hello, how are you today?
```

See Also

wcsncmp

Synopsis
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);

Purpose
Compares wide characters between two arrays.

Return Value
The **wcsncmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wcsncmp**() function compares not more than *n* wide characters (those that follow a null wide character are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

Example

```
/* This is an example of the function wcsncmp(const wchar_t*, const wchar_t*,
   size_t).  This compares the specifed number of characters of two strings.
   If string1 substring < string2 substring, then a negative value is returned.
   If string1 substring = string2 substring, then zero is returned.
   If string1 substring > string2 substring, then a positive value is returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t count;
    int retval,len1,len2;
    count = 5;
    line = "hellohello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2,len2);
    fputs("String1: ", stdout);
    fputws(str1, stdout);
    fputs("\nString2: ", stdout);
    fputws(str2, stdout);
```

```
    printf("\nNumber of characters to compare: %d\n", count);
    retval = wcsncmp(str1, str2, count);
    if(retval > 0)
        printf("String1 is greater than String2\n");
    else if(retval == 0)
        printf("String1 is identical to String2\n");
    else if(retval < 0)
        printf("String1 is less than String2\n");
    else
        printf("Error!\n");
    printf("wcsncmp(str1,str2,count) = %d\n", retval);
}
```

Output

```
String1: hellohello
String2: hello
Number of characters to compare: 5
String1 is identical to String2
wcsncmp(str1,str2,count) = 0
```

See Also

wcsncpy

Synopsis
#include <wchar.h>
wchar_t wcsncpy(wchar_t * restrict *s1***, const wchar_t * restrict** *s2***, size_t** *n***);**

Purpose
Copies wide characters from one array to another array.

Return Value
The **wcsncpy**() returns the value of *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wcsncpy**() function copies not more than *n* wide characters (those that follow a null wide character are not copied) from the array pointed to by *s2* to the array pointed to by *s1*.

If the array pointed to by *s2* is a wide string that is shorter than *n* wide characters, null wide characters are appended to the copy in the array pointed to by *s1*, until *n* wide characters in all have been written.

Example

```
/*  This is an example of the function wcsncpy(wchar_t*, const wchar_t*,
    size_t). This copies the specified number of characters from string2
    to string1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line;
    wchar_t  *retval;
    size_t count;
    int len;

    line ="How are you today?";
    count =18;
    len = strlen(line);
    mbstowcs(str2, line,len);
    wcsncpy(str1,str2,count);
    fputws(str1, stdout);
    fputs("\n", stdout);
}
```

Output

```
How are you today?
```

See Also

wcspbrk

Synopsis
#include <wchar.h>
wchar_t wcspbrk(const wchar_t *s1, const wchar_t *s2);

Purpose
Locates the first occurance of a wide character from one string in another string.

Return Value
The **wcspbrk**() function returns a pointer to the wide character in *s1*, or a null pointer if no wide character from *s2* occurs in *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcspbrk**() function locates the first occurrence in the wide string pointed to by *s1* of any wide character from the wide string pointed to by *s2*.

Example

```
/* This is an example of the function wcspbrk(const wchar_t*, const wchar_t).
   Returns a pointer to the first character in string1 that also belongs to
   string2.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line, *line2;
    wchar_t  *retval;
    int len1, len2;
    line = "Today is a very fine day.\n";
    line2 ="abc";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcspbrk(str1,str2);
    fputs("Original: ", stdout);
    fputws(str1, stdout);
    fputs("Characters to search for: ", stdout);
    fputws(str2, stdout);
    fputs("\nFirst occurrence of a character belonging to string2: ", stdout);
    fputws(retval, stdout);
    fputs("\n", stdout);
}
```

Output

```
Original: Today is a very fine day.
Characters to search for: abc
First occurrence of a character belonging to string2: ay is a very fine day.
```

See Also

wcsrchr

Synopsis
#include <wchar.h>
wchar_t wcsrchr(const wchar_t *s, wchar_t *c*);

Purpose
Locates the last occurance of a wide character in a wide string.

Return Value
The **wcsrchr**() function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

Parameters
s Integer pointer argument.
c Integer argument.

Description
The **wcsrchr**() function locates the last occurance of *c* in the wide string pointed to by *s*. The terminating null wide character is considered to be part of the wide string.

Example

```
/* This is an example of the function wcsrchr(const char_t*, wchar_t*).
   This returns a pointer to the last occurence of the search character.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100];
    wchar_t c, *retval;
    char *line;
    int result, len;
    line = "Hello, how are you doing?";
    c = 'l';

    len = strlen(line);
    mbstowcs(str1, line, len);
    fputs("String to be searched: ", stdout);
    fputws(str1, stdout);
    printf("\nCharacter to be searched for: ");
    fputwc(c, stdout);
    printf("\n");
    retval = wcsrchr(str1,c);
    result = retval - str1 +1;
    if(result <= 0) {
  printf("There is no '");
        fputwc(c, stdout);
        printf("' in this string.\n");
    }
    else {
```

```
     printf("The last occurence of ’");
        fputwc(c, stdout);
        printf("’ is at position: %d\n", result);
    }
}
```

Output

```
String to be searched: Hello, how are you doing?
Character to be searched for: l
The last occurence of ’l’ is at position: 4
```

See Also

wcsrtombs

Synopsis
#include <wchar.h>
size_t wcsrtombs(wchar_t * restrict *dst***, const wchar_t ** restrict** *src***, size_t** *len***, mbstate_t * restrict** *ps***);**

Purpose
Converts a sequence of wide characters from an array into a sequence of multibyte characters.

Return Value
If conversion stops because a code is reached that does not correspond to a valid multibyte character, an encoding error occurs: the **wcsrtombs**() function stores the value of the macro **EILSEQ** in **errno**() and returns $(size\_t)$ **- 1**; the conversion state is undefined. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null (if any).

Parameters
dst Character pointer argument.
src Character pointer argument.
len Unsigned integer argument.
ps Object type pointer.

Description
The **wcsrtombs**() function converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if *dst* is not a null pointer) when the next multibyte character would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst*. Each conversion takes place as if by call to the **wcrtomb**() function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null character, the resulting state described is the initial conversion state.

Example

```
/* This is an example of the function wcsrtombs(char*, const wchar_t**,
                        size_t, mbstate_t).
   The wcsrtombs function  converts  a  sequence  of  wide
   characters  from the array indirectly pointed to by src into
   a sequence of corresponding multibyte characters,  beginning
   in  the  conversion state described by the object pointed to
   by ps.
*/
#include <wchar.h>
#include <stdlib.h>
```

```
int main() {
    char *str1, *line;
    const wchar_t *str2;
    wchar_t str3[100];
    size_t retval, length;
    mbstate_t *ps;
    int len;

    line = "hello";
    len = strlen(line);
    mbstowcs(str3, line, len);
    str2 = str3;
    ps = NULL;
    length = 5;
    retval = wcsrtombs(str1, &str2,length, ps);
    if(retval != (size_t) -1) {
     printf("Number of multibye characters successfully");
        printf(" converted: %d\n",retval);
    }
    else
printf("Error\n");
}
```

Output

```
Number of multibye characters successfully converted: 5
```

See Also

wcsspn

Synopsis
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);

Purpose
Computes the length of the maximum initial segment of a wide string.

Return Value
The **wcsspn**() function returns the length of the segment.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcsspn**() function computes the length of the maximum initial segment of the wide string pointed to by *s1* which consists entirely of wide characters from the wide string pointed to by *s2*.

Example

```
/* This is an example of the function wcsspn(const wchar_t*, const wchar_t*).
   This returns the length of the substring found in str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t retval;
    int len1, len2;

    line = "hello, how are you?";
    line2 = "hel";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    fputs("String: ", stdout);
    fputws(str1, stdout);
    fputs("\nSubstring to search for: ", stdout);
    fputws(str2, stdout);
    retval = wcsspn(str1, str2);
    printf("\nThe length of the substring is:  %d\n", retval);
}
```

Output

```
String: hello, how are you?
Substring to search for: hel
```

```
The length of the substring is:  4
```

See Also

wcsstr

Synopsis
#include <**wchar.h**>
wchar_t wcsstr(const wchar_t **s1***, const wchar_t** **s2***);**

Purpose
Locates the first sequence found in one wide string in the other wide string.

Return Value
The **wcsstr**() function returns a pointer to the located wide string, or a null pointer if the wide string is not found. If *s2* points to a wide string with zero length, the function returns *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.

Description
The **wcsstr**() function locates the last occurance in the wide string pointed to by *s1* of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by *s2*.

Example

```
/*  This is the function wcsstr(const wchar_t*, const wchar_t*).
    This function finds the first occurence of a wide string from str2 in str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    wchar_t *retval;
    char *line, *line2;
    int len1, len2;
    line = "Today is a very fine day.";
    line2 ="is";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    retval = wcsstr(str1, str2);
    if(retval != NULL) {
     fputs("Original: ", stdout);
     fputws(str1, stdout);
     fputs("\nCharacters to search for: ", stdout);
     fputws(str2, stdout);
     printf("\nFirst occurrence of a character ");
        printf("belonging to string2: ");
        fputws(retval,stdout);
     fputs("\n", stdout);
    }
```

```
    else
        printf("Error\n");
}
```

Output

```
Original: Today is a very fine day.
Characters to search for: is
First occurrence of a character belonging to string2: is a very fine day.
```

See Also

wcstod

Synopsis
#include <wchar.h>
double wcstod(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***);**
float wcstof(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***);**
long double wcstold(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***);**

Purpose
Converts an initial portion of a wide string to double, float, and long double, respectively.

Return Value
The functions return the converted value, if any. If no conversions could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the return type and the sign of the value) and the value of the macro **ERANGE** is stored in **errno**(). If the result underflows, the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno**() acquires the value **ERANGE** is implementation defined.

Parameters
nptr Integer pointer argument.
endptr Integer pointer argument.

Description
The **wcstod**(), **wcstof**(), and **wcstold**() functions convert the initial portion of the wide string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **iswspace**() function) a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters;

— a **0x** or **0X**, then a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional binary exponent part;

— one of **INF** or **INFINITY**, or any other wide string equivalent.

— one of **NAN** or **NAN(**$n - wchar - sequence_{opt}$**)**, or any other wide string equivalent except for case in the NAN part, where:
n-wchar-sequence:
digit

nondigit
n-wchar-sequence digit
n-wchar-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating point number, or if a binary exponent part does not appear in a binary floating point number, an exponent part of the appropriate type with the value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide character sequence **INF** or **INFINITY** is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A wide character sequence **NAN** or **NAN(**$n - wchar - sequence_{opt}$**)** is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequences is implementation-defined. A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, the value resulting from the conversion is correctly rounded.

In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Recommended practice If the subject sequence has the hexadecimal form and **FLT_RADIX** is not a power of 2, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at the most **DECIMAL_DIG** (defined in **float.h**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than **DECIMAL_DIG** significant digits, such that the values of *L* and *U*, both having **DECIMAL_DIG** significant digits, such that the values of *L*, *D*, and *U* satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

Example

```
/* This is an example of the function wcstod(const wchar_t*, wchar_t**).
   This converts a strint to a double value.
*/
#include <wchar.h>
```

```
#include <stdlib.h>

int main() {
    wchar_t nptr[100];
    wchar_t *endptr;
    double x;
    char *line;
    int len;

    line = "3.1415926This stopped it";
    len = strlen(line);
    mbstowcs(nptr, line, len);
    x = wcstod(nptr, &endptr);
    printf("Original string: ");
    fputws(nptr, stdout);
    printf("\nDouble representation: %f\n", x);
}
```

Output

```
Original string: 3.1415926This stopped it
Double representation: 3.141593
```

See Also

wcstok

Synopsis
#include <wchar.h>
wchar_t wcstok(wchar_t * restrict *s1***, const wchar_t * restrict** *s2***, wchar_t ** restrict** *ptr***);**

Purpose
Breaks a wide string into a sequence of tokens.

Return Value
The **wcstok**() function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
ptr Integer pointer argument.

Description
A sequence of calls to the **wcstok**() function breaks the wide string pointed to by *s1* into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by *s2*. The third argument points to a caller-provided **wchar_t** pointer into which the **wcstok**() function stores information necessary for it to continue scanning the same wide string.

The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by *ptr*. Subsequent calls in the sequence have a null first argument and the object pointed to by *ptr* is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by *s2* may be different from call to call.

The first call in the sequence searches the wide string pointed to by *s1* for the first wide character that is *not* contained in the current separator wide string pointed to by *s2*. If no such wide character is found, then there are no tokens in the wide string pointed to by *s1* and the **wcstok** function returns a null pointer. If such a wide character is found, it is the start of the first token.

The **wcstok**() function then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by *s1*, and the subsequent searches in the same wide string for a token returns a null pointer. If such a wide character is found, it is overwritten by a null character, which terminates the current token.

In all cases, the **wcstok**() function stores sufficient information in the pointer pointed to by *ptr* so that subsequent calls, with a null pointer for *s1* and the unmodified pointer value for *ptr*, shall start searching just past the element overwritten by a null wide character (if any).

Example

```
/* This is an example of the function wcstok(wchar_t*, const wchar_t*,
                                              wchar_t**).  This pulls out
```

```
   the tokens from a given string. There is a difference between the number
   of arguments depending on where you compile.  Portability difference
   between HP and Sun.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    wchar_t *endptr = NULL;
    wchar_t *token;
    int len1, len2;

    line = "hello goodbye now";
    line2 = " ";
    len1 = strlen(line);
    len2 = strlen(line2);
    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2, len2);
    fputs("Original string: ", stdout);
    fputws(str1, stdout); fputs("\n", stdout);
    for (token = wcstok(str1, str2, &endptr);
         token != NULL; token= wcstok(NULL, str2, &endptr)) {
      fputs("token = ", stdout); fputws(token, stdout); fputs("\n", stdout);
   }
}
```

Output

```
Original string: hello goodbye now
token = hello
token = goodbye
token = now
```

See Also

wcstol

Synopsis
#include <wchar.h>
long int wcstol(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***, int** *base***);**
long long int wcstoll(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***, int** *base***);**
unsigned long int wcstoul(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***, int** *base***);**
unsigned long long int wcstoull(const wchar_t * restrict *nptr***, wchar_t ** restrict** *endptr***, int** *base***);**

Purpose
Converts wide string to long int, long long int, unsigned long int, or unsigned long long int.

Return Value
The **wcstol**(), **wcstoll**(), **wcstoul**(), and **wcstoull**() functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type sign of the value, if any) and the value of the macro **ERANGE** is stored in errno.

Parameters
nptr Integer pointer argument.
endptr Integer pointer argument.
base Integer argument.

Description
The **wcstol**(), **wcstoll**(), **wcstoul**(), and **wcstoull**() functions convert the initial portion of the wide string pointed to by *nptr* to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the **iswspace**() function) a subject sequence resembling an integer represented in some radix determined by the value of *base*, and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to an integer, and return the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36 (inclusive) the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters **a** (or **A**) through **z** (or **Z**) are ascribed the values of 10 through 35; only letters and digits whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white space, or if the first non-white-space

wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the **"C"** locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Example

```
/* This is an example of the function wcstol(const wchar_t*,wchar_t** ,int).
    This converts a string to a long integer in the specified base.
*/
#include <wchar.h>

int main() {
    wchar_t nptr[100];
    wchar_t *endptr;
    int base, len;
    long retval;
    char *line;

    line= "-10110134932This stopped it";
    base = 10;
    len = strlen(line);
    mbstowcs(nptr, line, len);
    fputs("Original string: ", stdout);
    fputws(nptr, stdout);
    printf("\nLong integer representation in base %d : ", base);
    retval = wcstol(nptr, &endptr, base);
    printf("%ld\n", retval);
}
```

Output

```
Original string: -10110134932This stopped it
Long integer representation in base 10 : -2147483648
```

See Also

wcsxfrm

Synopsis
#include <wchar.h>
int wcsxfrm(const wchar_t * restrict *s1***, const wchar_t * restrict** *s2***, size_t** *n***);**

Purpose
Transforms a wide string and places it in an array.

Return Value
The **wcsxfrm**() function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned in *n* or greater, the contents of the array pointed to by *s1* are indeterminate.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wcsxfrm**() function transforms the wide string pointed to by *s2* and places the resulting wide string into the array pointed to by *s1*. The transformation is such that if the **wcscmp**() function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **wcscoll**() function applied to the same two original wide strings. No more than *n* wide characters are placed into the resulting array pointed to by *s1*, including the terminating null wide character. If *n* is permitted to be a null pointer.

Example

```
/* This is an example of the function wcsxfrm(wchar_t*, const wchar_t*,size_t).
   This trnasforms a string based on locale-specific information.  It returns
   the length of the transformed string.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t retval,count;
    int len1;

    line = "abc";
    len1 = strlen(line);
    mbstowcs(str2, line, len1);
    fputs("Source string: ", stdout);
    fputws(str2, stdout);
    retval = wcsxfrm(str1, str2,count);
    printf("\nThe length of the transformed string is:  %d\n", retval);
}
```

Output

```
Source string: abc
The length of the transformed string is:  3
```

See Also

wctob

Synopsis
#include <wchar.h>
#include <stdio.h>
int wctob(wint_t i*c*);

Purpose
Determines if a character corresponds to member of extended character set.

Description
The **wctob**() function determines whether *c* corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Parameters
c Integer argument.

Return Value
The **wctob**() function returns **EOF** if *c* does not correspond to multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an **unsigned char** converted to an **int**.

Example

```
/*  This is an example of the function wctomb(char*, wchar_t).  This function
    determines the number of bytes needed to represent the equivalent mulitbyte
    character.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wint_t wc;
    int retval;
    wc=L'r';

    retval = wctob(wc);
    if(retval != EOF) {
        printf("wctob() = ");
      fputc(retval, stdout);
     printf("\n");
    }
    else
   printf("error\n");
}
```

Output

```
wctob() = r
```

See Also

wmemchr

Synopsis
#include <wchar.h>
wchar_t wmemchr(const wchar_t *s, wchar_t *c*, size_t *n*);

Purpose
Locates the first occurance of a specified wide character in a stream.

Return Value
The **wmemchr**() function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

Parameters
s Integer pointer argument.
c Integer argument.
n Unsigned integer argument.

Description
The **wmemchr**() function locates the first occurance of *c* in the initial *n* wide characters of the object pointed to by *s*.

Example

```
/*  This is an example of wmemchr(const wchar_t*, wchar_t, size_t).
    This funtion finds and returns the located specified wide character 'c'.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int len, result;
    size_t count;
    wchar_t *retval, str1[100], c;
    char *line;
    c='r';
    count = 20;

    line = "Hello how are you today?";
    len = strlen(line);
    mbstowcs(str1, line, len);
    retval = wmemchr(str1,c, count);
    if(retval != NULL) {
     printf("%s\n", line);
     result = retval - str1 +1;
     if(result <= 0)
            printf("There is no %c in this string.\n",c);
     else
            printf("The first '%c' is at position: %d\n", c, result);
    }
    else
        printf("Error\n");
```

```
}
```

Output

```
Hello how are you today?
The first 'r' is at position: 12
```

See Also

wmemcmp

Synopsis
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);

Purpose
Compares the first wide characters in two objects.

Return Value
The **wmemcmp**() function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wmemcmp**() function compares the first *n* wide characters of the object pointed to by *s1* to the first *n* wide characters of the object pointed to by *s2*.

Example

```
/* This is an example of the function wmemcmp(const wchar_t*, const wchar_t*,
   size_t).  This compares the specifed number of characters of two strings.
   If string1 substring < string2 substring, then a negative value is returned.
   If string1 substring = string2 substring, then zero is returned.
   If string1 substring > string2 substring, then a positive value is returned.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    const char *line, *line2;
    wchar_t str1[100], str2[100];
    size_t count;
    int retval,len1,len2;
    count = 6;
    line = "hellohello";
    line2 = "hello";
    len1 = strlen(line);
    len2 = strlen(line2);

    mbstowcs(str1, line, len1);
    mbstowcs(str2, line2,len2);
    retval = wmemcmp(str1, str2, count);
    if(retval != NULL) {
     fputs("String1: ", stdout);
     fputws(str1, stdout);
     fputs("\nString2: ", stdout);
     fputws(str2, stdout);
```

```
     printf("\nNumber of characters to compare: %d\n", count);
     if(retval > 0)
            printf("String1 is greater than String2\n");
     else if(retval == 0)
            printf("String1 is identical to String2\n");
     else if(retval < 0)
            printf("String1 is less than String2\n");
     else
            printf("Error!\n");
     printf("wmemcmp(str1,str2,count) = %d\n", retval);
    }
    else
        printf("Error\n");
}
```

Output

```
String1: hellohello
String2: hello
Number of characters to compare: 6
String1 is greater than String2
wmemcmp(str1,str2,count) = 104
```

See Also

wmemcpy

Synopsis
#include <wchar.h>
wchar_t wmemcpy(wchar_t * restrict *s1***, const wchar_t * restrict** *s2***, size_t** *n***);**

Purpose
Copies wide characters from one object to another.

Return Value
The **wmemcpy**() returns the value of *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wmemcpy**() function copies *n* wide characters from the object pointed to by *s2* to the object pointed to by *s1*.

Example

```
/* This is an example of the function wmemcpy(wchar_t*, const wchar_t*,
                                                                  size_t).
   This copies the specified number of characters from string2 to string1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line;
    wchar_t  *retval;
    size_t count;
    int len;

    line ="How are you today?";
    count =11;
    len = strlen(line);
    mbstowcs(str2, line,len);
    retval = wmemcpy(str1,str2,count);
    if(retval != NULL) {
     printf("Original string: ");
     fputws(str2, stdout);
     printf("\nNumber of characters to copy: %d", count);
     printf("\nCopied string: ");
     fputws(str1, stdout);
     fputs("\n", stdout);
    }
    else
        printf("Error\n");
```

```
}
```

Output

```
Original string: How are you today?
Number of characters to copy: 11
Copied string: How are you
```

See Also

wmemmove

Synopsis
#include <wchar.h>
wchar_t wmemmove(wchar_t * restrict *s1***, const wchar_t * restrict** *s2***, size_t** *n***);**

Purpose
Copies wide characters from one object to another.

Return Value
The **wmemmove**() returns the value of *s1*.

Parameters
s1 Integer pointer argument.
s2 Integer pointer argument.
n Unsigned integer argument.

Description
The **wmemmove**() function copies *n* wide characters from the object pointed to by *s2* to the object pointed to by *s1*. Copying takes place as if the *n* wide characters from the object pointed to by *s2* are first copied into a temporary array of *n* wide characters that does not overlap the objects pointed to by *s1* or *s2*, and then the *n* wide characters from the temporary array are copied into the object pointed to by *s1*.

Example

```
/* This is an example of the function wmemmove(wchar_t*, const wchar_t*,
                                                                  size_t)
   This copies the specified number of characters from string2 to string1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    wchar_t str1[100], str2[100];
    char *line, *line2;
    wchar_t  *retval;
    size_t count;
    int len;

    line ="What is your name?";
    count =15;
    len = strlen(line);
    retval = wmemmove(str1,str2,count);
    if(retval != NULL) {
     printf("Original line: %s\n", line );
     mbstowcs(str2, line,len);
     printf("Number of characters to move: %d",count);
     printf("\nNew string1: ");
     fputws(wmemmove(str1, str2, count), stdout);
     fputs("\n", stdout);
    }
```

```
    else
        printf("Error\n");
}
```

Output

```
Original line: What is your name?
Number of characters to move: 15
New string1: What is your na
```

See Also

wmemset

Synopsis
#include <wchar.h>
wchar_t wmemset(const wchar_t *s, wchar_t *c*, size_t *n*);

Purpose
Copies a single value into specified number of locations in an object.

Return Value
The **wmemset**() function returns the value of *s*.

Parameters
s Integer pointer argument.
c Integer argument.
n Unsigned integer argument.

Description
The **wmemset**() function copies the value of *c* into each of the first *n* wide characters of the object pointed to by *s*.

Example

```
/*  This is an example of wmemset(wchar_t*, wchar_t, size_t).  This function
    copies the specified wide character into the first n positions of str1.
*/
#include <wchar.h>
#include <stdlib.h>

int main() {
    int len;
    size_t count;
    wchar_t *retval, str1[100], c;
    char *line;
    c='r';
    count = 10;

    line = "Hello how are you today?";
    len = strlen(line);
    mbstowcs(str1, line, len);
    retval = wmemset(str1,c, count);
    if(retval != NULL) {
     printf("Original string: %s\n", line);
     printf("Character to replace with: ");
     fputwc(c, stdout);
     printf("\nThe number of characters to replace: %d\n", count);
     printf("The new string is: ");
     fputws(retval, stdout);
     fputs("\n", stdout);
    }
    else
        printf("Error\n");
```

```
}
```

Output

```
Original string: Hello how are you today?
Character to replace with: r
The number of characters to replace: 10
The new string is: rrrrrrrrrrare you today?
```

See Also

Chapter 22

Wide Character Functions — <wctype.h>

The header **wctype.h** declares three data types, one macro, and many functions.

The types declared are

wint_t

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see **WEOF** below);

wctrans_t

which is a scalar type that can hold values which represent locale-specific character mappings; and

wctype_t

which is a scalar type that can hold values which represent locale-specific character classifications.

The macro defined is

WEOF

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set. It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide-character value that does not correspond to any member of the extended character set.

The functions declared are grouped as follows:

—Functions that provide wide-character classification;

—Extensible functions that provide wide-character classification;

—Functions that provide wide-character case mapping;

—Extensible functions that provide wide-character mapping.

For all functions described in this subclause that accept an argument of type **wint_t**, the value shall be representalble as a **whar_t** or shall equal the value of the macro **WEOF**. If this argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the **LC_TYPE** category of the current locale.

Public Data
None.

Functions
The following functions are prototyped in the header file **wctype.h**.

Functions	Description
iswalnum()	Tests for alpha-numeric wide character.
iswalpha()	Tests for upper or lower case alphabetic wide character.
iswcntrl()	Tests for control wide character.
iswctype()	Determines whether character has properties to be a wide character.
iswdigit()	Tests for decimal-digit wide characters.
iswgraph()	Tests for printing wide character.
iswlower()	Tests for lowercase wide characters.
iswprint()	Tests for printing wide character.
iswpunct()	Tests for punctuation wide characters.
iswspace()	Tests for white-space wide characters.
iswupper()	Tests for uppercase letter wide characters.
iswxdigit()	Tests for hexadecimal-digit wide characters.
towctrans()	Maps wide characters.
towlower()	Converts uppercase letter to corresponding lowercase letter.
towupper()	Converts lowercase letter to corresponding uppercase letter.
wctrans()	Describes mapping between wide characters.
wctype()	Describes a class of wide characters.

Macros
The following macros are defined for the **wctype** header.

Macro	Description
WEOF	Used to indicate *end-of-file*.

Declared Types
The following types are declared in the **wctype** header.

Declared Types	Description
wint_t	int-holds values corresponding to members of the extended set.
wctrans_t	scalar-holds values which represent locale-specific character mappings.
wctype_t	scalar-holds values which represent locale-specific character classifications.

Portability
This header has no known portability problem.

iswalnum

Synopsis
#include <wctype.h>
int iswalnum(wint_t *wc***);**

Purpose
Tests to see if a character is a wide alpha-numeric character.

Return Value
The **iswalnum**() function returns a boolean value. It returns true if the character is a wide alpha-numeric character.

Parameters
wc Integer argument.

Description
The **iswalnum**() function tests for any wide character for which **iswalpha**() or **iswdigit**() is true.

Example

```
/* This is an example of the function iswalnum(wint_t ).  This function checks
   if the parameter is an alphanumeric character.  If true, the function
   returns a positive number, or zero if false.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswalnum(wc) != 0)
     printf(": is a alphanumeric character.\n");
    else
        printf(": is not an alphanumeric character.\n");
}
```

Output

```
i: is a alphanumeric character.
```

See Also

iswalpha

Synopsis
#include <wctype.h>
int iswalpha(wint_t *wc***);**

Purpose
Tests to see if a character is an upper or lower case alphabetic wide character.

Return Value
The **iswalpha**() function returns a boolean value. It returns true if the character is an upper or lower case alphabetic wide character.

Parameters
wc Integer argument.

Description
The **iswalpha**() function tests for any wide character for which **iswupper**() or **iswlower**() is true, or any wide character that is one of a locale-specific set of alphabetic wide characters for which none of **iswcntrl**(), **iswdigit**(), **iswpunct**(), or **iswspace**() is true.

Example

```
/* This is an example for iswalpha(wint_t).  This function will return a
   non-zero value if the parameter is an alphabetic character.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswalpha(wc) > 0)
        printf(": is an alphabetic character\n");
    else
        printf(": is not an alphabetic character\n");
}
```

Output

```
i: is an alphabetic character
```

See Also

iswcntrl

Synopsis
#include <wctype.h>
int iswcntrl(wint_t *wc***);**

Purpose
Tests for a control wide character.

Return Value
The **iswcntrl**() function returns a boolean value. If the character is a control wide character the function returns a true.

Parameters
wc Integer argument.

Description
The **iswcntrl**() function tests for any control wide character.

Example

```
/* This is an example for iswcntrl(wint_t).  This returns a non-zero number
   if it is a control wide character.  If not, it will return zero.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswcntrl(wc) > 0)
        printf(": is a wide control character.\n");
    else
        printf(": is not a wide control character.\n");
}
```

Output

```
i: is not a wide control character.
```

See Also

iswctype

Synopsis
#include <wctype.h>
int iswctype(wint_t *wc***, wctype_t** *desc***);**

Purpose
Determines whether a character has the properties to be a wide character.

Return Value
The **iswctype**() function returns nonzero (true) if and only if the value of the wide character *wc* has the property described by *desc*.

Parameters
wc Integer argument.
desc Scalar argument.

Description
The **iswctype**() function determines whether the wide character *wc* has the property described by *desc*. The current setting of the **LC_CTYPE** category shall be the same as during the call to **wctype**() that returned the value *desc*.

Each of the following expressions has a truth-value equivalent to the call to the wide-cha racter classification function in the comment that follows the expression:

iswctype(*wc***, wctype("alnum")) // iswgraph(***wc***)**
iswctype(*wc***, wctype("graph")) // iswgraph(***wc***)**
iswctype(*wc***, wctype("cntrl")) // iswcntrl(***wc***)**
iswctype(*wc***, wctype("digit")) // iswdigit(***wc***)**
iswctype(*wc***, wctype("graph")) // iswgraph(***wc***)**
iswctype(*wc***, wctype("lower")) // iswlower(***wc***)**
iswctype(*wc***, wctype("print")) // iswprint(***wc***)**
iswctype(*wc***, wctype("punct")) // iswpunct(***wc***)**
iswctype(*wc***, wctype("space")) // iswspace(***wc***)**
iswctype(*wc***, wctype("upper")) // iswupper(***wc***)**
iswctype(*wc***, wctype("xdigit")) // iswxdigit(***wc***)**

Example

```
/* This is an example for iswctype(wint_t, wctype_t).  This test the wint_t
   parameter for the specifed property.  If the wide character has the
   specified property, than a nonzero value will be returned.  If it does not,
   then zero will be returned.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
```

```
    wint_t wc;
    char *prop;

    prop = "upper";
    wc = 'I';
    fputs("The wide character '", stdout);
    fputwc(wc, stdout);
    if(iswctype(wc, wctype(prop)) > 0)
        printf("' has the property: %s\n", prop);
    else
        printf("' does not have the property: %s\n", prop);
}
```

Output

```
The wide character 'I' has the property: upper
```

See Also

iswdigit

Synopsis
#include <wctype.h>
int iswdigit(wint_t *wc***);**

Purpose
Tests for decimal-digit wide characters.

Return Value
The **iswdigit**() function returns a boolean value which is true when the character is a decimal-digit character.

Parameters
wc Integer argument.

Description
The **iswdigit**() function tests for any wide character that corresponds to a decimal-digit character.

Example

```
/* This is an example for iswdigit(wint_t).  This function will return a
   non-zero value if the parameter is digit.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = '1';
    fputwc(wc, stdout);
    if(iswdigit(wc) > 0)
        printf(": is a digit.\n");
    else
        printf(": is not a digit.\n");
}
```

Output

```
1: is a digit.
```

See Also

iswgraph

Synopsis
#include <wctype.h>
int iswgraph(wint_t *wc***);**

Purpose
Tests for printing wide characters.

Return Value
The **iswgraph**() function returns a boolean value which is true when the character is a printing wide character.

Parameters
wc Integer argument.

Description
The **iswgraph**() function tests for any wide character for which **iswprint**() is true and **iswspace**() is false.

Example

```
/* This is an example for iswgraph(wint_t).  This function will return a
   non-zero value if the parameter is a printable character except space.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswgraph(wc) != 0)
printf(": is a printable character.\n");
    else
 printf(": is not a printable character.\n");
}
```

Output

```
i: is a printable character.
```

See Also

iswlower

Synopsis
#include <wctype.h>
int iswlower(wint_t *wc***);**

Purpose
Tests for lowercase wide characters.

Return Value
The **iswlower**() function returns a boolean value which is true when the argument is a lower case letter.

Parameters
wc Integer argument.

Description
The **iswlower**() function tests for any wide character that correspnds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**(), **iswdigit**(), **iswpunct**(), or **iswspace**() is true.

Example

```
/* This is an example for iswlower(wint_t).  This function will return a
   non-zero value if the parameter is a lowercase letter.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswlower(wc) != 0)
         printf(": is a lowercase letter.\n");
    else
          printf(": is not a lowercase letter.\n");
}
```

Output

```
i: is a lowercase letter.
```

See Also

iswprint

Synopsis
#include <wctype.h>
int iswprint(wint_t *wc***);**

Purpose
Tests for printing wide characters.

Return Value
The **iswprint**() function returns a boolean value which is true when the argument is a printing wide character.

Parameters
wc Integer argument.

Description
The **iswprint**() function tests for any printing wide character.

Example

```
/* This is an example for iswprint(wint_t).  This function will return a
   non-zero value if the parameter is a printable character including space
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'i';
    fputwc(wc, stdout);
    if(iswprint(wc) != 0)
        printf(": is a printable character.\n");
    else
        printf(": is not a printable character.\n");
}
```

Output

```
i: is a printable character.
```

See Also

iswpunct

Synopsis
#include <wctype.h>
int iswpunct(wint_t *wc***);**

Purpose
Tests for punctuation wide characters.

Return Value
The **iswpunct**() function returns a boolean value which is true when the argument is a punctuation wide character.

Parameters
wc Integer argument.

Description
The **iswpunct**() function tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither **iswspace**() nor **iswalnum**() is true.

Example

```
/* This is an example for iswpunct(wint_t).  This function will return a
   non-zero value if the parameter is x punctuation character.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = '.';
    fputwc(wc, stdout);
    if(iswpunct(wc) != 0)
        printf(": is a punctuation mark.\n");
    else
        printf(": is not a punctuation mark.\n");
}
```

Output

```
.: is a punctuation mark.
```

See Also

iswspace

Synopsis
#include <wctype.h>
int iswspace(wint_t *wc***);**

Purpose
Tests for white-space wide characters.

Return Value
The **iswspace**() function returns a boolean value which is true when the argument is a white-space wide characters.

Parameters
wc Integer argument.

Description
The **iswspace**() function tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of **iswalnum**(), **iswgraph**(), or **iswpunct**() is true.

Example

```
/* This is an example for iswspace(wint_t).  This function will return a
   non-zero value if the parameter is a white space character.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = ' ';
    fputwc(wc, stdout);
    if(iswspace(wc) != 0)
        printf(": is a white space character.\n");
    else
        printf(": is not a white space character.\n");
}
```

Output

```
 : is a white space character.
```

See Also

iswupper

Synopsis
#include <wctype.h>
int iswupper(wint_t *wc***);**

Purpose
Tests for uppercase letter wide characters.

Return Value
The **iswupper**() function returns a boolean value which is true when the argument is an uppercase letter wide character.

Parameters
wc Integer argument.

Description
The **iswupper**() function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**(), **iswdigit**(), **iswpunct**(), or **iswspace**() is true.

Example

```
/* This is an example for the function iswupper(wint_t).  This function returns
   a non negative number if the input is an uppercase letter.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'I';
    fputwc(wc, stdout);
    if(iswupper(wc) != 0)
        printf(": is an uppercase letter.\n");
    else
        printf(": is not an uppercase letter.\n");
}
```

Output

```
I: is an uppercase letter.
```

See Also

iswxdigit

Synopsis
#include <wctype.h>
int iswxdigit(wint_t *wc***);**

Purpose
Tests for hexadecimal-digit wide characters.

Return Value
The **iswxdigit**() function returns a boolean value which is true when the argument is a hexadecimal-digit character.

Parameters
wc Integer argument.

Description
The **iswxdigit**() function tests for any wide character that corresponds to a hexadecimal-digit character.

Example

```
/* This is an example for iswxdigit(wint_t).  This function will return a
   non-zero value if the parameter is a hexadecimal digit.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;

    wc = 'A';
    fputwc(wc, stdout);
    if(iswxdigit(wc) != 0)
        printf(": is a hexadecimal digit.\n");
    else
        printf(": is not a hexadecimal digit.\n");
}
```

Output

```
A: is a hexadecimal digit.
```

See Also

towctrans

Synopsis
#include <towctrans.h>
wint_t towctrans(wint_t *wc***);**

Purpose
Maps wide characters.

Return Value
The **towctrans**() function returns the mapped value of *wc* using the mapping described by *desc*.

Parameters
wc Integer argument.
desc Scalar argument.

Description
The **towctrans**() function maps the wide character *wc* using the mapping described by *desc*. The current setting of the **LC_TYPE** category shall be the same as during the call to **wctrans**() that returned the value *desc*.

Each of the following expressions behaves the same as the call to the wide-character case- mapping function in the comment that follows the expression:

towctrans(*wc***, wctrans("tolower")) /* towlower(***wc***) */**
towctrans(*wc***, wctrans("toupper")) /* towupper(***wc***) */**

Example

```
/* This is an example for towctrans(wint_t, wctrans_t).  This test the wint_t
   parameter for the specifed property.  If the wide character has the
   specified property, than a nonzero value will be returned.  If it does not,
   then zero will be returned.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    char *prop;

    prop = "toupper";
    wc = 'i';
    fputs("The wide character '", stdout);
    fputwc(wc, stdout);
    printf("' has been changed to: '");
    fputwc(towctrans(wc, wctrans(prop)), stdout);
    printf("'. \nBy the property of %s.\n", prop);
}
```

Output

```
The wide character 'i' has been changed to: 'I'.
By the property of toupper.
```

See Also

towlower

Synopsis
#include <towlower.h>
wint_t towlower(wint_t *wc***);**

Purpose
Converts uppercase letter to corresponding lowercase letter.

Return Value
If the argument is a wide character for which **iswupper**() is true and there are one or more corresponding wide characters, as specified by the current locale , for which **iswlower**() is true, the **towlower**() funciton returns one of the correspoding wide characters; otherwise, the argument is returned unchanged.

Parameters
wc Integer argument.

Description
The **towlower**() function converts an uppercase letter to a corresponding lowercase letter.

Example

```
/* This is an example of the function towlower(wint_t).  This will convert the
   wide character into its lowercase representation.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    wc ='A';

    printf("Before: ");
    fputwc(wc, stdout);
    printf("\n");
    printf("After: ");
    fputwc(towlower(wc), stdout);
}
```

Output

```
Before: A
After: a
```

See Also

towupper

Synopsis
#include <towupper.h>
wint_t towupper(wint_t *wc***);**

Purpose
Converts lowercase letter to corresponding uppercase letter.

Return Value
If the argument is a wide character for which **iswlower**() is true and there are one or more corresponding wide characters, as specified by the current locale , for which **iswupper**() is true, the **towupper**() function returns one of the correspoding characters; otherwise, the argument is returned unchanged.

Parameters
wc Integer argument.

Description
The **towupper**() function converts a lowercase letter to a corresponding uppercase letter.

Example

```
/* This is an example of the function towupper(wint_t).  This function converts
   lowercase letters to uppercase.
*/
#include <wctype.h>
#include <wchar.h>

int main() {
    wint_t wc;
    wc = 'i';

    printf("Before: ");
    fputwc( wc, stdout);
    printf("\n");
    printf("After: ");
    fputwc( towupper(wc), stdout);
    printf("\n");
}
```

Output

```
Before: i
After: I
```

See Also

wctrans

Synopsis
#include <wctype.h>
wctrans_t wctrans(const char ***property*****);**

Purpose
Describes mapping between wide characters.

Return Value
If *property* identifies a valid mapping of wide characters according to the **LC_CTYPE** category of the current locale, the **wctrans**() function returns a nonzero value that is valid as the second argument to the **towctrans**() function; otherwise, it returns zero.

Parameters
**property* Character argument.

Description
The **wctype**() function constructs a value with type **wctrans_t**() that describes a mapping between wide characters identified by the string argument **property**.

The strings listed in the description of the **towctrans**() function shall be valid in all locales as *property* arguments to the **wctrans**() function.

Example

```
/* This is an example of the function wctrans(const char*).  This returns a
   non-negative number if the string is a valid wctrans.
*/
#include <wctype.h>

int main() {
    char *prop;
    prop ="tolower";
    if(wctrans(prop) > 0)
printf("'%s' is a valid wctrans.\n", prop);
    else
        printf("'%s' is not a valid wctrans.\n", prop);
}
```

Output

```
'tolower' is a valid wctrans.
```

See Also

wctype

Synopsis
#include <**wctype.h**>
wctype_t wctype(const char **property*);

Purpose
Describes a class of wide characters.

Returns
If *property* identifies a valid class of wide characters according to the **LC_CTYPE** category of the current locale, the **wctype**() function returns a nonzero value that is valid as the second argument to the **iswctype**() function; otherwise, it returns zero.

Parameters
**property* Character argument.

Description
The **wctype**() function constructs a value with type **wctype_t**() that describes a class of wide characters identified by the string argument *property*.

The strings listed in the description of the **iswctype**() function shall be valid in all locales as *property* arguments to the **wctype**() function.

Example

```
/* This is an example of the function wctype(const char*).  This returns a
   non-negative number if the string is a valid wctype.
*/
#include <wctype.h>

int main() {
    char *prop;
    prop ="xdigit";
    if(wctype(prop) > 0)
        printf("'%s' is a valid wctype.\n", prop);
    else
        printf("'%s' is not a valid wctype.\n", prop);
}
```

Output

```
'xdigit' is a valid wctype.
```

See Also
iswctype()

Appendix A

Functions Not Supported in Specific Platforms

Most functions in Ch are supported across different platforms. Functions which are not supported under different operating systems are listed below.

A.1 HPUX

The following functions are implemented with return error code only under the HPUX operating system.

Function	Header File: windows/winsock.h	Return Value
HANDLE WSAAPI WSAAsyncGetHostByAddr(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **addr*, **int** *len*, **int** *type*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetHostByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetProtoByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetProtoByNumber(**HWND** *hwnd*, **unsigned int** *wMsg*, **int** *number*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetServByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **const char FAR** **proto*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetServByPort(**HWND** *hwnd*, **unsigned int** *wMsg*, **int** *port*, **const char FAR** **proto*, **char FAR** **buf*, **int** *buflen*)		0
int WSAAPI WSAAsyncSelect(**SOCKET** *s*, **HWND** *hwnd*, **unsigned int** *wMsg*, **long** *lEvent*)		– 1
int WSAAPI WSACancelAsyncRequest(**HANDLE** *AsyncTaskHandle*)		– 1
int WSAAPI WSACancelBlockingCall(**void**)		– 1
int WSAAPI WSACleanup(**void**)		0
void WSAAPI WSAGetLastError(**void**)		N/A
BOOL WSAAPI WSAIsblocking(**void**)		False

BOOL WSAAPI WSASetBlockingHook(**FARPROC** *lpBlockFunc*)	NULL
void WSAAPI WSASetLastError(**int** *iError*)	N/A
int WSAAPI WSAStartup(**WORD** *wVersionRequested*, **LPWSADATA** *lpWSAData*)	0
int WSAAPI WSAUnhookBlockingHook(**void**)	– 1
int WSAAPI ioctlsocket(**SOCKET** *s*, **long** *cmd*, **u_long** **argp*)	– 1

Function	Header File: netdb.h	Return Value
struct hostent *__**gethostbyname_r**(**const char** *__*name*, **struct hostent** *__*result*, **char** *__*buffer*, **int** *__*buflen*, **int** *__*h_errnop*)		NULL
struct hostent *__**gethostbyaddr_r**(**const char** *__*addr*, **int** *__*length*, **int** *__*type*, **struct hosteni** *__*result*, **char** *__*buffer*, **int** *__*buflen*, **int** *__*h_errnop*)		NULL
struct hostent *__**gethostent_r**(**struct hostent** *__*result*, **char** *__*buffer*, **int** *__*buflen*, **int** *__*h_errnop*)		NULL
struct servent *__**getservbyname_r**(**const char** *__*name*, **const char** *__*proto*, **struct servent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct servent *__**getservbyport_r**(**int** *__*port*, **const char** *__*proto*, **struct servent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct servent *__**getservent_r**(**struct hosten** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct netent *__**getnetbyname_r**(**const char** *__*name*, **struct netent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct netent *__**getnetbyaddr_r**(**long** *__*net*, **int** *__*type*, **struct netent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct netent *__**getnetent_r**(**struct netent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct protoent *__**getprotobyname_r**(**const char** *__*name*, **struct protent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct protoent *__**getprotobynumber_r**(**int** *__*proto*, **struct protent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL
struct protoent *__**getprotoent_r**(**struct protent** *__*result*, **char** *__*buffer*, **int** *__*buflen*)		NULL

Function	Header File: math.h	Return Value
double gamma_r(**double** *x*, **int** **signagamp*)		– 1
double scalbn(**double** *x*, **int** **n*)		– 1
int isnormal(*real-floating x*)		0
int signbit(*real-floating x*)		– 1

Function	Header File: unistd.h	Return Value
char getlogin_r(**char** *__*name*, **int** *__*namelen*)		NULL
char *__**ttyname_r**(**int** *__*fildes*, **char** *__*buf*, **int** *__*len*)		NULL

int setegid(**uid_t** *egid*)	0
int seteuid(**uid_t** *egid*)	0

Function	Header File: grp.h	Return Value
struct group *__**getgrnam_r**(**const char** *__*name*, **struct group** *__*result*, **char** *__*buffer*, **int** *buflen*)		NULL
struct group *__**getgrgid_r**(**gid_t** *gid*, **struct group** *__*result*, **char** *__*buffer*, **int** *buflen*)		NULL
struct group *__**fgetgrent_r**(**FILE** *__*f*, **struct group** *__*result*, **char** *__*buffer*, **int** *buflen*)		NULL
struct group *__**fgetgrent**(**FILE** *__*f*)		NULL

Function	Header File: mqueue.h	Return Value
ssize_t mq_receive(**mqd_t** *mqdes*, **char** *__*msg_ptr*, **size_t** *msq_len*, **unsigned int** *__*msg_prio*)		0

Function	Header File: sys/procset.h	Return Value
int sigsendset(**procset_t** *__*psp*, **int** *sig*)		– 1

Function	Header File: libintl.h	Return Value
char *__**gettext**(**const char** *__*msgid*)		NULL
char *__**dgettext**(**const char** *__*domaninname*, **const char** *__*msgid*)		NULL
char *__**textdomain**(**const char** *__*domainname*)		NULL
char *__**bindtextdomain**(**const char** *__*domainname*)		NULL

Function	Header File: time.h	Return Value
int ascftime(**char** *__*s*, **const char** *__*format*, **const struct tm** *__*timeptr*)		0
int cftime(**char** *__*s*, **const char** *__*format*, **const time_t** *clock*)		0

Function	Header File: wchar.h	Return Value
wint_t btowc(**int** *c*)		WEOF

int fwide(**FILE** **stream*, **int** *c*)	0
int mbsinit(**const mbstate_t** **ps*)	0
size_t mbrlen(**const char** **str*, **size_t** *length*. **mbstate_t** *ps*)	0
size_t mbrtowc(**wchar_t** **str1*, **const char** **str2*, **size_t** *length*, **mbstate_t** *ps*)	– 1
size_t mbsrtowcs(**wchar_t** **str1*, **const char** ***str2*, **size_t** *length*, **mbstate_t** **ps*)	– 1
size_t wcrtomb(**char** **str1*, **wchar_t** *s*, **mbstate_t** **ap*)	– 1
size_t wcsrtombs(**char** **str1*, **const char** ***str2*, **size_t** *length*, **mbstate_t** **ps*)	– 1
wchar_t wcsstr(**const wchar_t** **str1*, **const wchar_t** **str2*)	NULL
int wctob(**wint_t** *wc*)	EOP
wchar_t wmemchr(**const wchar_t** **str1*, **wchar_t** *c*, **size_t** *count*)	NULL
int wmemcmp(**const wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)	NULL
wchar_t *wmemmove(**wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)	NULL
wchar_t *wmemset(**wchar_t** **str1*, **wchar_t** *str2*, **size_t** *count*)	NULL
wchar_t *wmemcpy(**wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)	NULL

Function	Header File: wctype.h	Return Value
wint_t towctrans(**win_t** *wc*, **wctrans_t** *desc*)		– 1
wctrans_t wctrans(**const char** **prop*)		0

A.2 Linux

The following functions are implemented with return error code only under the Linux operating system.

Function	Header File: windows/winsock.h	Return Value
HANDLE WSAAPI WSAAsyncGetHostByAddr(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **addr*, **int** *len*, **int** *type*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetHostByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetProtoByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetProtoByNumber(**HWND** *hwnd*, **unsigned int** *wMsg*, **int** *number*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetServByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **const char FAR** **proto*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetServByPort(**HWND** *hwnd*, **unsigned int** *wMsg*, **int** *port*, **const char FAR** **proto*, **char FAR** **buf*, **int** *buflen*)		0
int WSAAPI WSAAsyncSelect(**SOCKET** *s*, **HWND** *hwnd*, **unsigned int** *wMsg*, **long** *lEvent*)		– 1

int WSAAPI WSACancelAsyncRequest(**HANDLE** *AsyncTaskHandle*)	– 1
int WSAAPI WSACancelBlockingCall(**void**)	– 1
int WSAAPI WSACleanup(**void**)	0
void WSAAPI WSAGetLastError(**void**)	N/A
BOOL WSAAPI WSAIsblocking(**void**)	False
BOOL WSAAPI WSASetBlockingHook(**FARPROC** *lpBlockFunc*)	NULL
void WSAAPI WSASetLastError(**int** *iError*)	N/A
int WSAAPI WSAStartup(**WORD** *wVersionRequested*, **LPWSADATA** *lpWSAData*)	0
int WSAAPI WSAUnhookBlockingHook(**void**)	– 1
int WSAAPI ioctlsocket(**SOCKET** *s*, **long** *cmd*, **u_long** **argp*)	– 1

Function	Header File: aio.h	Return Value
int aio_read(**struct aiocb** **aiocbp*)		– 1
int aio_write(**struct aiocb** **aiocbp*)		– 1
int lio_listio(**int** *mode*, **struct aiocb** **list []*, **int** *nent*, **struct sigevent** **sig*)		– 1
int aio_error(**const struct aiocb** **aiocbp*)		– 1
int aio_return(**struct aiocb** **aiocbp*)		– 1
int aio_cancel(**int** *fildes*, **struct aiocb** **aiocbp*)		– 1
int aio_suspend(**const struct aiocb** **aiocbp[]*, **int** *nent*, **const struct timespec** **timeout*)		– 1
int aio_fsync(**int** *op*, **struct aiocb** **aiocbp*)		– 1

Function	Header File: time.h	Return Value
char ***ctime_r**(**const time_t** **clock*, **char** **buf*, **int** *buflen*)		NULL
char ***cftime**(**char** **s*, **char** **format*, **const time_t** **clock*)		0
struct tm ***localtime_r**(**const time_t** **clock*, **struct tm** **res*)		NULL
struct tm ***gmtime_r**(**const time_t** **clock*, **struct tm** **res*)		NULL
char ***asctime_r**(**const struct** *tm* **tm*, **char** **buf*, **int** *buflen*)		NULL
int clock_settime(**clockid_t** *clock_id*, **const struct timespec** **tp*)		– 1
int clock_gettime(**clockid_t** *clock_id*, **const struct timespec** **tp*)		– 1
int clock_getres(**clockid_t** *clock_id*, **const struct timespec** **res*)		– 1
int timer_create(**clockid_t** *clock_id*, **struct sigevent** **evp*, **timer_t** **timerid*)		– 1
int timer_delete(**timer_t** *timerid*)		– 1
int timer_settime(**timer_t** *timerid*, **int** *flags*, **const struct itimerspec** **value*, **struct itimerspec** **ovalue*)		– 1
int timer_gettime(**timer_t** *timerid*, **struct timespec** **value*)		– 1
int timer_getoverrun(**timer_t** *timerid*)		– 1
int nanosleep(**const struct timespec** **rqtp*, **struct timespec** **rmtp*)		– 1
char ascftime(**char** **s*, **const char** **format*, **const struct tm** **timeptr*)		0

Function	Header File: libgen.h	Return Value
char* **regcmp**(**const char** **string1*, /* **char***string2* */, ... /* (**char***)0 */)		NULL
char* **regex**(**const char** **re*, **const char** **subject*, /* **char***ret0* */ ...)		NULL

Function	header File: math.h	Return Value
int fpclassify(*real-floating x*)		– 1
int isnormal(*real-floating x*)		0
double gamma_r(**double** *x*, **int** **signgamp*)		NaN
int signbit(*real-floating x*)		– 1

Function	Header File: signal.h	Return Value
int sighold(**int** *sig*)		– 1
int sigrelse(**int** *sig*)		– 1
int sigignore(**int** *sig*)		– 1
int sigaltstack(**const stack_t** **ss*, **stack_t** *oss*)		– 1

Function	Header File: stropts.h	Return Value
int fattach(**int** *fildes*, **const char** **path*)		– 1
int fdetach(**const char** **path*)		– 1
int getmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **long** **flagsp*)		– 1
int getpmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **int** **band*, **long** **flagsp*)		– 1
int isastream(**int** *fildes*)		– 1
int putmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **long** *flags*)		– 1
int putpmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **int** *band*, **long** *flags*)		– 1

Function	Header File: sys/mman.h	Retrun Value
int msync(**void** * *addr*, **size_t** *len*, **int** *flags*)		– 1
int mlock(**void** **addr*, **size_t** *len*)		– 1

int munlock(**void** **addr*, **size_t** *len*)	– 1
int shm_open(**const char** **name*, **int** *oflag*, **mode_t** *mode*)	– 1
int shm_unlink(**const char** **name*)	– 1

Function	Header File: sys/procset.h	Return Value
int sigsend(**idtype_t** *idtype*, **id_t** *id*, **int** *sig*)		– 1
int sigsendset(**procset_t** **psp*, **int** *sig*)		– 1

Function	Header File: sys/wait.h	Return Value
int waitid(**idtype_t** *idtype*, **id_t** *id*, **siginfo_t** **infop*, **int** *options*)		– 1

Function	Header File: poll.h	Return Value
int poll(**struct pollfd** **fds*, **unsigned int** *nfds*, **int** *timeout*)		– 1

Function	Header File: unistd.h	Return Value
int fchroot(**int** *fildes*)		– 1
char ****gettxt**(**const char** **msgid*, **const char** **dflt_str*)		NULL
pid_t getsid(**pid_t** *pid*)		– 1
int mincore(**caddr_t** *addr*, **size_t** *len*, **char** **vec*) **unsigned int** *scale*)		0

A.3 Solaris

The following functions are implemented with return error code only under the Solaros operating system.

Function	Header File: windows/winsock.h	Return Value
HANDLE WSAAPI WSAAsyncGetHostByAddr(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **addr*, **int** *len*, **int** *type*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetHostByName(**HWND** *hwnd*, **unsigned int** *wMsg*, **const char FAR** **name*, **char FAR** **buf*, **int** *buflen*)		0
HANDLE WSAAPI WSAAsyncGetProtoByName(**HWND** *hwnd*, **unsigned**		0

int *wMsg*, **const char FAR** **name*, **char FAR** **buf*, **int** *buflen*)	
HANDLE WSAAPI WSAAsyncGetProtoByNumber(**HWND** *hwnd*, **unsigned**	0
int *wMsg*, **int** *number*, **char FAR** **buf*, **int** *buflen*)	
HANDLE WSAAPI WSAAsyncGetServByName(**HWND** *hwnd*, **unsigned int** *wMsg*, 0	
char FAR **buf*, **int** *buflen*)	
HANDLE WSAAPI WSAAsyncGetServByPort(**HWND** *hwnd*, **unsigned int** *wMsg*,	0
int *port*, **const char FAR** **proto*, **char FAR** **buf*, **int** *buflen*)	
int WSAAPI WSAAsyncSelect(**SOCKET** *s*, **HWND** *hwnd*, **unsigned int** *wMsg*,	– 1
long *lEvent*)	
int WSAAPI WSACancelAsyncRequest(**HANDLE** *AsyncTaskHandle*)	– 1
int WSAAPI WSACancelBlockingCall(**void**)	– 1
int WSAAPI WSACleanup(**void**)	0
void WSAAPI WSAGetLastError(**void**)	N/A
BOOL WSAAPI WSAIsblocking(**void**)	False
BOOL WSAAPI WSASetBlockingHook(**FARPROC** *lpBlockFunc*)	NULL
void WSAPPI WSASetLastError(**int** *iError*)	N/A
int WSAAPI WSAStartup(**WORD** *wVersionRequested*, **LPWSADATA** *lpWSAData*)	0
int WSAAPI WSAUnhookBlockingHook(**void**)	– 1
int WSAAPI ioctlsocket(**SOCKET** *s*, **long** *cmd*, **u_long** **argp*)	– 1

Function	Header File: math.h	Return Value
int fpclassify(*real-floating x*)		– 1
int isnormal(**double** *x*)		0
int signbit(**double** *x*)		0

Function	Header File: wchar.h	Return Value
wint_t btowc(**int** *c*)		WEOF
int fwide(**FILE** **stream*, **int** *c*)		0
int mbsinit(**const mbstate_t** **ps*)		0
size_t mbrlen(**const char** **str*, **size_t** *length*. **mbstate_t** *ps*)		0
size_t mbrtowc(**wchar_t** **str1*, **const char** **str2*, **size_t** *length*, **mbstate_t** *ps*)		– 1
size_t mbsrtowcs(**wchar_t** **str1*, **const char** ***str2*, **size_t** *length*,		– 1
mbstate_t **ps*)		
size_t wcrtomb(**char** **str1*, **wchar_t** *s*, **mbstate_t** **ap*)		– 1
size_t wcsrtombs(**char** **str1*, **const char** ***str2*, **size_t** *length*, **mbstate_t** **ps*)		– 1
wchar_t ***wcsstr**(**const wchar_t** **str1*, **const wchar_t** **str2*)		NULL
int wctob(**wint_t** *wc*)		EOP
wchar_t wmemchr(**const wchar_t** **str1*, **wchar_t** *c*, **size_t** *count*)		NULL
int wmemcmp(**const wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)		NULL
wchar_t wmemmove(**wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)		NULL

wchar_t wmemset(**wchar_t** **str1*, **wchar_t** *str2*, **size_t** *count*)	NULL
wchar_t wmemcpy(**wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)	NULL

A.4 Windows

The following functions are implemented with return error code only under the Windows 95/98/NT operating systems.

Function	Header File: arpa/inet.h	Return Value
unsigned long inet_network(**const char** **cp*)		– 1
struct in_addr inet_makeaddr(**int** *net*, **int** *lna*)		– 1
unsigned long inet_lnaof(**struct in_addr** *in*)		– 1
unsigned long inet_netof(**struct in_addr** *in*)		– 1

Function	Header File: libintl.h	Return Value
char ***gettext**(**const char** **msgid*)		NULL
char ***dgettext**(**const char** **domaninname*, **const char** **msgid*)		NULL
char ***textdomain**(**const char** **domainname*)		NULL
char ***bindtextdomain**(**const char** **domainname*)		NULL

Function	Header File: aio.h	Return Value
int aio_read(**struct aiocb** **aiocbp*)		– 1
int aio_write(**struct aiocb** **aiocbp*)		– 1
int lio_listio(**int** *mode*, **struct aiocb** **list []*, **int** *nent*, **struct sigevent** **sig*)		– 1
int aio_error(**const struct aiocb** **aiocbp*)		– 1
int aio_return(**struct aiocb** **aiocbp*)		– 1
int aio_cancel(**int** *fildes*, **struct aiocb** **aiocbp*)		– 1
int aio_suspend(**const struct aiocb** **aiocbp[]*, **int** *nent*, **const struct timespec** **timeout*)		– 1
int aio_fsync(**int** *op*, **struct aiocb** **aiocbp*)		– 1

Function	Header File: dirent.h	Return Value
int closedir(**DIR** **dirp*)		– 1
DIR opendir(**const char** **filename*)		NULL

struct dirent readdir(**DIR** **dirp*)	NULL
int readdir_r(**DIR** **dirp*, **struct dirent** **entry*, **struct dirent** ***result*)	NULL
void rewinddir(**DIR** **dirp*)	N/A
void seekdir(**DIR** **dirp*, **long** *loc*)	N/A
long telldir(**DIR** **dirp*)	– 1

Function	Header File: glob.h	Return Value
int glob(**const char** **pattern*, **int** *flags*, **int** (**errfunc*(**const char** **epath*, **int** *eerrno*) **glob_t** **pglob*)		–1
void globfree(**glob_t** **pglob*)		N/A

Function	Header File: grp.h	Return Value
int initgroups(**const char** **name*, **gid_t** *basegid*)		– 1
void setgrent(**void**)		N/A
struct group ***getgrent**(**void**)		NULL
int getgrent_r(**struct group** **result*, **char** **buffer*, **int** *buflen*, **FILE** ***grfp*)		– 1

Function	Header File: libgen.h	Return Value
char* **regcmp**(**const char** **string1*, /* **char****string2* */, ... /* (**char***)0 */)		NULL
char* **regex**(**const char** **re*, **const char** **subject*, /* **char****ret0* */ ...)		NULL

Function	Header File: math.h	Return Value
double erf(**double** *x*)		NaN
double erfc(**double** *x*)		NaN
double cbrt(**double** *x*)		NaN
double rint(**double** *x*)		NaN
double scalbn(**double** *x*, **int** *n*)		NaN
double remainder(**float** *x*, **float** *x*)		NaN
double ilogb(**double** *x*)		NaN
double gamma(**double** *x*)		NaN
double lgamma(**double** *x*)		NaN
double gamma_r(**double** *x*, **int** **signgamp*)		NaN
double lgammar(**double** *x*, **int** **signgamp*)		NaN
int fpclassify(*real-floating x*)		– 1
int isnormal(*real-floating x*)		0
int signbit(*real-floating x*)		– 1

Function	Header File: netdb.h	Return Value
struct hostent *__**gethostent**__(**void**)		NULL
struct netent *__**getnetbyname**__(**const char** **name*)		NULL
struct netent *__**getnetbyaddr**__(**long** *net*, **int** *type*)		NULL
struct netent *__**getnetent**__(**void**)		NULL
struct servent *__**getservent**__(**void**)		NULL
struct protoent *__**getprotoent**__(**void**)		NULL
int sethostent(**int** *stayopen*)		– 1
int endhostent(**void**)		– 1
int endnetent (**void**)		– 1
int setnetent(**int** *stayopen*)		– 1
int setservent(**int** *stayopen*)		– 1
int endservent(**void**)		– 1
int setprotoent(**int** *stayopen*)		– 1
int endprotoent(**void**)		– 1
int rcmd(**char** ***ahost*, **unsigned short** *inport*, **char** **luser*, **char** **ruser*, **char** **cmd*, **int** **fd2p*)		– 1
int rresvport(**int** **port*)		– 1
int ruserok(**char** **rhost*, **int** *suser*, **char** **ruser*, **char** **luser*)		– 1
int rexec(**char** ***ahost*, **unsigned short** *inport*, **char** **user*, **char** **passwd*, **char** **cmd*, **int** **fd2p*)		– 1
struct hostent *__**gethostbyname_r**__(**const char** **name*, **struct hostent** **result*, **char** **buffer*, **int** **buflen*, **int** **h_errnop*)		NULL
struct hostent *__**gethostbyaddr_r**__(**const char** **addr*, **int** *length*, **int** *type*, **struct hostent** **result*, **char** **buffer*, **int** *buflen*, **int** **h_errnop*)		NULL
struct hostent *__**gethostent_r**__(**struct hostent** **result*, **char** **buffer*, **int** *buflen*, **int** **h_errnop*)		NULL
struct servent *__**getservbyname_r**__(**const char** **name*, **const char** **proto*, **struct servent** **result*, **char** **buffer*, **int** **buflen*)		NULL
struct servent *__**getservbyport_r**__(**int** *port*, **const char** **proto*, **struct servent** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct servent *__**getservent_r**__(**struct hosten** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct netent *__**getnetbyname_r**__(**const char** **name*, **struct netent** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct netent *__**getnetbyaddr_r**__(**long** *net*, **int** *type*, **struct netent** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct netent *__**getnetent_r**__(**struct netent** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct protoent *__**getprotobyname_r**__(**const char** **name*, **struct protent** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct protoent *__**getprotobynumber_r**__(**int** *proto*, **struct protent** **result*, **char** **buffer*, **int** *buflen*)		NULL
struct protoent *__**getprotoent_r**__(**struct protent** **result*, **char** **buffer*, **int** *buflen*)		NULL

Function	Header File: poll.h	Return Value

int poll(**struct pollfd** **fds*, **unsigned int** *nfds*, **int** *timeout*)	– 1

Function	Header File: pwd.h	Return Value
struct passwd ***getpwent**( )		NULL
struct passwd ***getpwnam**(**const char** **name*)		NULL
struct passwd ***getpwuid**(**uid_t** *uid*)		NULL
struct passwd ***putpwent**(**const struct passwd** **p*, **FILE** **f*)		– 1

Function	Header File: regex.h	Return Value
char ***regcomp**(**regex_t** **preg*, **const char** **pattern*, **int** *cflags*)		– 1
int regexec(**const regex_t** **preg*, **const char** **string*, **size_t** *nmatch*, **regmatch_t** *pmatch* [])		– 1
regerror(**int** *errcode*, **const regex_t** **preg*, **char** **errbuf*, **size_t** *eflags*)		– 1
void regfree(**regex_t** **preg*)		N/A

Function	Header File: signal.h	Return Value
int kill(**pid_t** *pid*, **int** *sig*)		– 1
int sigaction(**int** *sig*, **const struct sigaction** **act*, **struct sigaction** **oact*)		– 1
int sigemptyset(**sigset_t** **set*)		– 1
int sigfillset(**sigset_t** **set*)		– 1
int sigaddset(**sigset_t** **set*, **int** *signo*)		– 1
int sigdelset(**sigset_t** **set*, **int** *signo*)		– 1
int sigismember(**const sigset_t** **set*, **int** *signo*)		– 1
int sigpending(**sigset_t** **set*)		– 1
int sigprocmask(**int** *how*, **const sigset_t** **set*, **sigset_t** **oset*)		– 1
int sigsuspend(**const sigset_t** **set*)		– 1
int gsignal(**int** *sig*)		– 1
int sighold(**int** *sig*)		– 1
int sigrelse(**int** *sig*)		– 1
int sigignore(**int** *sig*)		– 1
int sigpause(**int** *sig*)		– 1
int sigaltstack(**const stack_t** **ss*, **stack_t** **oss*)		– 1

Function	Header File: stdio.h	Return Value
FILE ***fdopen**(**int** fildes, **const char** **type*)		NULL

Function	Header File: stdlib.h	Return Value
char * **initstate**(**unsigned int** *seed*, **char** * *state*, **size_t** *size*)		NULL
long random()		0
char * **setstate**(**const char** *state*)		NULL
void srandom(**unsigned int** **seed*)		N/A
int mkstemp(**char** * tempplate)		-1

Function	Header File: stropts.h	Return Value
int fattach(**int** *fildes*, **const char** **path*)		– 1
int fdetach(**const char** **path*)		– 1
int getmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **long** **flagsp*)		– 1
int getpmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **int** **band*, **long** **flagsp*)		– 1
int isastream(**int** *fildes*)		– 1
int putmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **long** *flags*)		– 1
int putpmsg(**int** *fildes*, **struct strbuf** **ctptr*, **struct strbuf** **dataptr*, **int** *band*, **long** *flags*)		– 1

Function	Header File: sys/stat.h	Return Value
int mknod(**const char** **path*, **mode_t** *mode*, **dev_t** *dev*)		– 1
int fchmod(**int** *fildes*, **mode_t** *mode*)		– 1
int mkfifo(**const char** **path*, **mode_t** *mode*)		– 1

Function	Header File: sys/acct.h	Return Value
int acct(**const char** **path*)		– 1

Function	Header File: sys/fcntl.h	Return Value
int fcntl(**int** *fildes*, **int** *cmd*, .../* arg */)		– 1

Function	Header File: sys/msg.h	Return Value
int msgctl(**int** *msqid*, **int** *cmd*, **struct msqid_ds** ***buf**)		– 1

int msgget(**key_t** *key*, **int** *msgflg*)	– 1

Function	Header File: sys/mman.h	Return Value
void *__**mmap**(**void** * *addr*, **size_t** *len*, **int** *prot*, **int** *flags*, **int** *fildes*, **off_t** *off*)		(void *) – 1
int mprotect(**void** **addr*, **size_t** *len*, **int** *port*)		– 1
int munmap(**void** **addr*, **size_t** *len*)		– 1

Function	Header File: sys/procset.h	Return Value
int sigsend(**idtype_t** *idtype*, **id_t** *id*, **int** *sig*)		– 1
int sigsendset(**procset_t** **psp*, **int** *sig*)		– 1

Function	Header File: sys/resource.h	Return Value
int getpriority(**int** *which*, **int** *who*)		– 1
int setpriority(**int** *which*, **int** *who*, **int** *prio*)		– 1
int getrlimit(**int** *resource*, **struct rlimit** **rlp*)		– 1
int setrlimit(**int** *resource*, **const struct rlimit** **rlp*)		– 1

Function	Header File: sys/socket.h	Return Value
int recvmsg(**int** *s*, **struct msghdr** **msg*, **int** *flags*)		– 1
int sendmsg(**int** *s*, **struct msghdr** **msg*, **int** *flags*)		– 1
int socketpair(**int** *domain*, **int** *type*, **int** *protocol*, **int** *sv*)		– 1

Function	Header File: sys/stat.h	Return Value
int mknod(**const char** **path*, **mode_t** *mode*, **dev_t** *dev*)		– 1
int fchmod(**int** *fildes*,**mode_t** *mode*)		– 1
int mkfifo(**const char** **path*, **mode_t** *mode*)		– 1

Function	Header File: sys/times.h	Return Value
clock_t times (**struct tms** **buffer*)		– 1

Function	Header File: sys/uio.h	Return Value
int writev(**int** *fildes*, **const struct iovec** **iov*, **int** *iovcnt*)		– 1

Function	Header File: sys/utsname.h	Return Value
int uname(**struct utsname** **name*)		– 1

Function	Header File: sys/wait.h	Return Value
pid_t wait (**int** **stat_loc*)		0
pid_t waitpid(**pid_t** *pid*, **int** **stat_loc*, **int** *options*)		– 1

Function	Header File: syslog.h	Return Value
void syslog(**int** *priority*, **const char** **message*, ...)		N/A
void openlog(**const char** **ident*, **int** *logopt*, **int** *facility*)		N/A
void closelog(**void**)		N/A
int setlogmask(**int** *maskpri*)		– 1

Function	Header File: termios.h	Return Value
int cfgetispeed(**struct struct termios** **termios_p*)		NULL
spped_t cfgetospeed(**struct struct termios** **termios_p*)		NULL
int cfsetispeed(**struct termios** **termios_p*, **speed_t** *speed*)		– 1
int cfsetospeed(**struct termios** **termios_p*, **speed_t** *speed*)		– 1
int 1**tcdrain**(**int** *fildes*)		– 1
int tcflow(**int** *fildes*, **int** *action*)		– 1
int tcflush(**int** *fildes*, **int** *queue_selector*)		– 1
int tcgetattr(**int** *fildes*, **struct termios** **termios_p*)		– 1
int tcsendbreak(**int** *fildes*, **int** *duration*)		– 1
int tcsetattr(**int** *fildes*, **int** *optional_actions*, **const struct termios** **termios_p*)		– 1

Function	Header File: time.h	Return Value
int ascftime(**char** **s*, **const char** **format*, **const struct tm** **timeptr*)		0
int cftime(**char** **s*, **char** **format*, **const time_t** **clock*)		0
char ***ctime_r**(**const time_t** **clock*, **char** **buf*, **int** *buflen*)		NULL

struct tm *__**localtime_r**(**const time_t** *__clock*, **struct tm** *__res*)	NULL
struct tm *__**gmtime_r**(**const time_t** *__clock*, **struct tm** *__res*)	NULL
char *__**asctime_r**(**const struct tm** *__tm*, **char** *__buf*, **int** *__buflen*)	NULL
int clock_settime(**clockid_t** *clock_id*, **const struct timespec** *__tp*)	– 1
int clock_gettime(**clockid_t** *clock_id*, **const struct timespec** *__tp*)	– 1
int clock_getres(**clockid_t** *clock_id*, **const struct timespec** *__res*)	– 1
int timer_create(**clockid_t** *clock_id*, **struct sigevent** *__evp*, **timer_t** *__timerid*)	– 1
int timer_delete(**timer_t** *timerid*)	– 1
int timer_settime(**timer_t** *timerid*, **int** *flags*, **const struct itimerspec** *__value*, **struct itimerspec** *__ovalue*)	– 1
int timer_gettime(**timer_t** *timerid*, **struct timespec** *__value*)	– 1
int timer_getoverrun(**timer_t** *timerid*)	– 1
int nanosleep(**const struct timespec** *__rqtp*, **struct timespec** *__rmtp*)	– 1

Function	Header File: unistd.h	Return Value
int acct(**const char** *__path*)		– 1
unsigned alarm(**unsigned int** *sec*)		0
int chown(**const char** *__path*, **uid_t** *owner*, **gid_t** *group*)		– 1
int chroot(**const char** *__path*)		– 1
char *__**ctermid**(**char** *__s*)		NULL
char *__**cuserid**(**char** *__s*)		NULL
int fchdir(**int** *fildes*)		– 1
int fchown(**int** *fildes*, **uid_t** *owner*, **gid_t** *group*)		– 1
int fchroot(**int** *fildes*)		– 1
int fsync(**int** *fildes*)		– 1
int ftruncate(**int** *fildes*, **off_t** *length*)		– 1
uid_t getuid(**void**)		– 1
uid_t geteuid(**void**)		– 1
gid_t getgid(**void**)		– 1
gid_t getegid(**void**)		– 1
int getgroups(**int** *gidsetsize*, **gid_t** *__grouplist*)		– 1
gethostid(**void**)		0
char *__**getlogin**(**void**)		NULL
char *__**getlogin_r**(**char** *__name*, **int** *namelen*)		NULL
pid_t getpgrp(**void**)		NULL
pid_t getppid(**void**)		NULL
pid_t getpgid(**pid_t** *pid*)		– 1
char *__**gettxt**(**const char** *__msgid*, **const char** *__dflt_str*)		NULL
pid_t getsid(**pid_t** *pid*)		NULL
pid_t setsid(**void**)		NULL
int link(**const char** *__existing*, **const char** *__newfile*)		– 1
int lchown(**const char** *__path*, **uid_t** *owner*, **gid_t** *group*)		– 1
int mincore(**caddr_t** *addr*, **size_t** *len*, **char** *__vec*)		– 1
int nice(**int** *incr*)		– 1
long pathconf(**const char** *__path*, **int** *name*)		– 1

int pause(**void**)	– 1
void profil(**unsigned short** **buff*, **unsigned int** *bufsiz*, **unsigned int** *offset*, **unsigned int** *scale*)	N/A
int ptrace(**int** *request*, **pid_t** *pid*, **int** *addr*, **int** *data*)	– 1
int readlink(**const char** **path*, **void** **buf*, **size_t** *bufsiz*)	– 1
void *__**sbrk**(**int** *incr*)	(void *) – 1
int setuid(**uid_t** *uid*)	– 1
int setegid(**gid_t** *egid*)	– 1
int seteuid(**uid_t** *euid*)	– 1
int setgid(**gid_t** *gid*)	– 1
int setgroups(**int** *ngroups*, **const gid_t** **grouplist*)	– 1
int setpgid(**pid_t** *pid*, **pid_t** *pgid*)	– 1
pid_t setpgrp(**void**)	– 1
int stime(**const time_t** **tp*)	– 1
int symlink(**const char** **name1*, **const char** **name2*)	– 1
void sync(**void**)	N/A
long sysconf(**int** *name*)	– 1
pid_t tcgetpgrp(**int** *fildes*)	NULL
int tcsetpgrp(**int** *fildes*, **pid_t** *pgid*)	– 1
int truncate(**const char** **path*, **off_t** *length*)	– 1
char ttyname(**int** *fildes*)	NULL
char ttyname_r(**int** *fildes*, **char** **buf*, **int** *len*)	NULL
void vhangup(**void**)	N/A

Function	Header File: wchar.h	Return Value
int fwide(**FILE** **stream*, **int** *c*)		0
int mbsinit(**const mbstate_t** **ps*)		0
wchar_t wmemchr(**const wchar_t** **str1*, **wchar_t** *c*, **size_t** *count*)		NULL
int wmemcmp(**const wchar_t** **str1*, **const wchar_t** **str2*, **size_t** *count*)		NULL
int wcwidth(**const wchar_t** **c*)		– 1
int wcswidth(**const wchar_t** **str1*, **size_t** *width*)		– 1

Function	Header File: wctype.h	Return Value
wint_t towctrans(**win_t** *wc*, **wctrans_t** *desc*)		– 1
wctrans_t wctrans(**const char** **prop*)		0
wctype_t wctype(**const char** **prop*)		0

Index