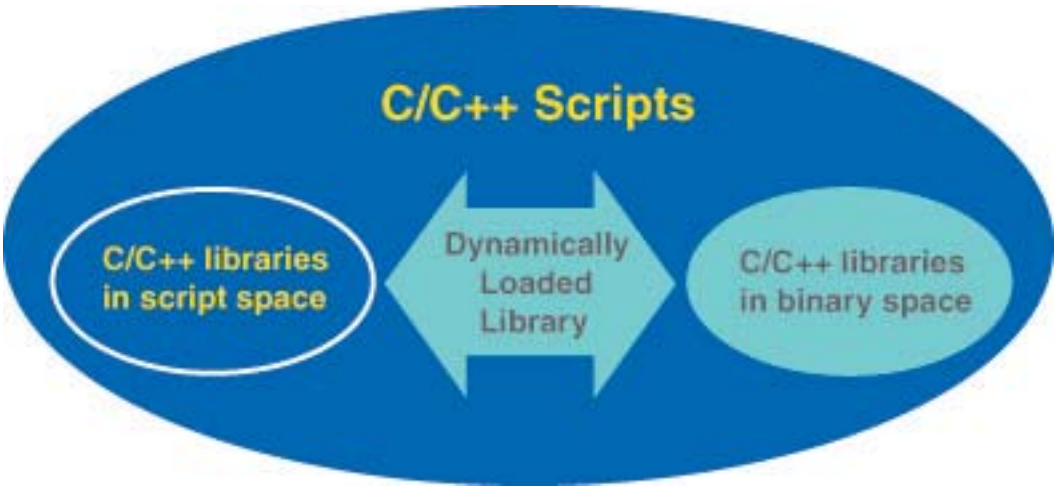
SoftIntegration®

The Ch Language Environment

Version 5.1

SDK User's 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, May 2006

Preface

Ch is an embeddable C/C++ interpreter. Ch supports all features in the ISO 1990 C standard, most new features in the latest ISO C99 standard including complex numbers and variable length arrays, classes in POSIX, C++, Win32, X/Motif, OpenGL, GTK+, ODBC, WinSock, very high-level shell programming, cross-platform internet computing in safe Ch, computational arrays for linear algebra and matrix computations, high-level 2D/3D plotting and numerical computing such as differential equation solving, integration, Fourier analysis. Ch can also be used as a login Unix command shell and for high-level scripting such as shell programming to automate tasks and common gateway interface in a Web server in both Unix and Windows.

Ch Software Development Kit (SDK) can be used to develop dynamically loaded libraries for C/C++ scripts (Ch scripts) to interface C/C++ binary libraries. It allows Ch scripts to access global variables or call functions in the compiled C/C++ libraries such as static library, shared library or Dynamically Linked Library (DLL). Ch scripts can callback Ch functions from C/C++ libraries functions. There is no distinction between the interpreted and compiled code. As illusrated in Figure1, function `func()` in a binary C library can be called from a Ch script using a wrapper functoin in function file `func.chf` in the Ch space through a dynamically loaded library with function `func_chdl()` in the C space developed using Ch SDK.

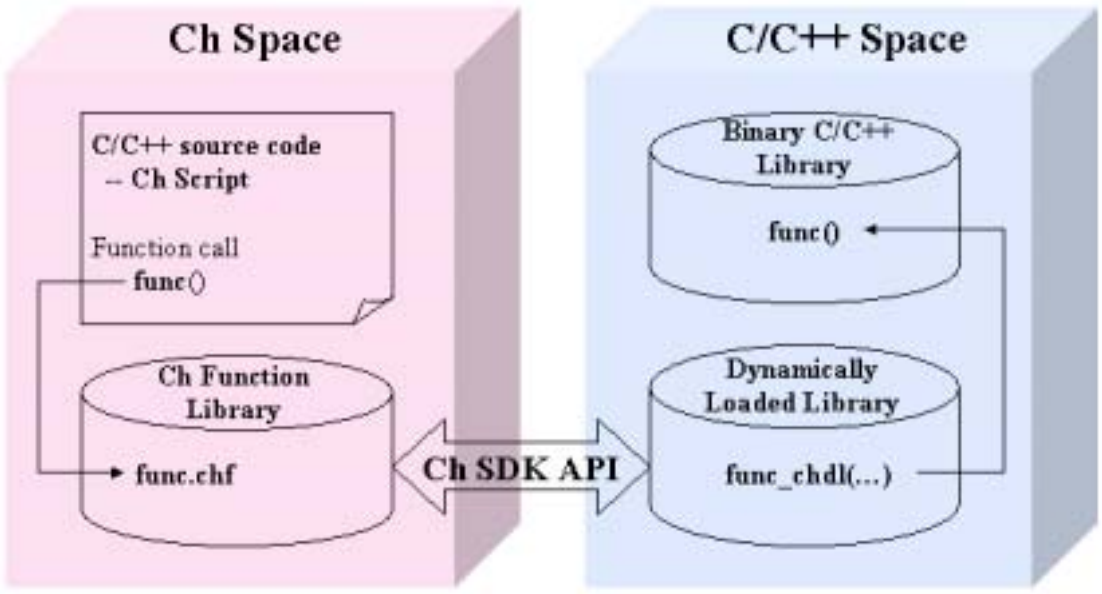


Figure 1: Using SDK to build the interface between C/C++ scripts and C/C++ libraries.

Details of interface Ch scripts with C/C++ libraries are described in this Ch SDK User's Guide. Features not explained in this document 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 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

Portable Compilation and Linking of C/C++ Programs in Different Platforms

Note: The source code and templates for all examples described in this document are available in CHHOME/toolkit/demos/SDK. CHHOME is the directory where Ch is installed. It is recommended that you try these examples while reading this document.

There are many different ways to compile a C/C++ program. Different platforms and compilers have different options and settings for compiling a C/C++ program. To interface Ch script with C/C++ libraries, a simple and consistent way for compiling a library across different platforms is introduced in this chapter. The compiling method in this chapter will be applied in the following chapters to compile libraries for Ch script to interface with binary modules.

1.1 How to Setup Ch for Compilation

Ch scripts are used to automate the task of compilation. Thus, you need to have Ch Standard Edition or Ch Professional Edition installed first.

1.1.1 Tested C/C++ Compilers

The tested C/C++ compilers for Ch SDK are Microsoft Visual C++ (.NET 2003) in Windows, Borland C/C++ compler v5.5 and Borland CBuilder v6.0 for Windows, MingW C/C++ compiler through gcc v3.4.4 for Windows. **gcc** and **g++** (Version 2.91 or above) in Linux, **cc** and **c++** (Version 3.3 or above) from Apple Computer in Mac OS X, **cc** and **CC** (Version 5.0 or above) from Sun Microsystem in Solaris, **cc** and **CC** from Hewlett-Packard (Version A.10.32.03 or above) in HP-UX, **cc** and **c++** for FreeBSD 5.1, and **cc (qcc)** and **CC (QCC)** from QNX Software Systems for QNX 6.2.1,

To compile the sample C/C++ programs discussed in the following sections, the search paths have to be set up properly first.

1.1.2 Setting Up Ch in Unix and Linux

The default path for the C and C++ compilers in Unix systems is `/bin`. You can check and see if it has already been set in Ch shell by typing the command either `echo $PATH` or `_path`. However, if the compilers are located in other directories or there is no default path settings, you must add the paths with the commands for the C and C++ compilers to the system variable **_path** either in the system startup file

CHHOME/config/chrc or user's startup file ~/.chrc, where CHHOME is the directory Ch in which is installed. Thus, whenever Ch is started, the path will be included. The user can start a Ch shell with option -d as follows

```
ch -d
```

to copy a sample startup file from directory CHHOME/config/ to the user's home directory if there is no startup file in the home directory yet. Please refer *Ch User's Guide* for more information about _path and chrc.

Below is an example to set PATH for Sun CC compiler by adding directory /opt/SUNWspro/bin" for _path:

```
_path = stradd(_path, "/opt/SUNWspro/bin;");
```

The advantage of using Ch SDK for building interface with C/C++ binary is that no change is needed for your current building of C/C++ libraries. It mimimizes the changes for your existing code. Once you have created the library either in Windows IDE or Unix, you can use solutions below to create the interface automatically.

1.1.3 Setting Up Ch in Windows

The startup file ~/_chrc in the user's home directory can be configured to setup the Visual .NET, Visual (VC++), or Borland C++ compiler in Windows for automatic compilation in a Ch shell.

1.1.3.1 Using Microsoft Visual .NET and Visual C++

If a VC++ in .NET 2003 is installed at C:/Program Files/Microsoft Visual Studio .NET 2003, the code below needs to be added to the startup file _chrc in the user's home directory.

```
_path = stradd(_path, "C:/Program Files/Microsoft Visual Studio .NET 2003/VC7/bin;");
_path = stradd(_path, "C:/Program Files/Microsoft Visual Studio .NET 2003/COMMON7/IDE;");
putenv(stradd("LIB=C:/Program Files/Microsoft Visual Studio .NET 2003/VC7/lib;",
       "C:/Program Files/Microsoft Visual Studio .NET 2003/VC7/PlatformSDK/lib;",
       getenv("LIB")));
putenv(stradd("INCLUDE=C:/Program Files/Microsoft Visual Studio .NET 2003/VC7/include;",
       "C:/Program Files/Microsoft Visual Studio .NET 2003/VC7/PlatformSDK/Include",
       ".;", getenv("INCLUDE")));
```

If a VC++ in .NET is installed at C:/Program Files/Microsoft Visual Studio .NET, the code below needs to be added to the startup file _chrc in the user's home directory.

```
_path = stradd(_path, "C:/Program Files/Microsoft Visual Studio .NET/VC7/bin;");
_path = stradd(_path, "C:/Program Files/Microsoft Visual Studio .NET/COMMON7/IDE;");
putenv(stradd("LIB=C:/Program Files/Microsoft Visual Studio .NET/VC7/lib;",
       "C:/Program Files/Microsoft Visual Studio .NET/VC7/PlatformSDK/lib;",
       getenv("LIB")));
putenv(stradd("INCLUDE=C:/Program Files/Microsoft Visual Studio .NET/VC7/include;",
       "C:/Program Files/Microsoft Visual Studio .NET/VC7/PlatformSDK/Include",
       ".;", getenv("INCLUDE")));
```

For VC++ 6.0 installed at C:/Program Files/Microsoft Visual Studio/, the code below needs to be added to the startup file ~/_chrc in the user's home directory.

```
string_t tmp_;
_path = stradd(_path, "C:/Program Files/Microsoft Visual Studio/VC98/Bin;",
            "C:/Program Files/Microsoft Visual Studio/Common/MSDev98/Bin;");
putenc(stradd("LIB=", "C:/Program Files/Microsoft Visual Studio//VC98/Lib;",
              "C:/Program Files/Microsoft Visual Studio/VC98/MFC/Lib;",
              getenv("LIB")));
putenv(stradd("INCLUDE=",
      "C:/Program Files/Microsoft Visual Studio/VC98/include;",
      "C:/Program Files/Microsoft Visual Studio/VC98/MFC/include;",
      ".;", getenv("INCLUDE")));
```

The code above defines the search paths for all the programs that will be run in the VC++ environment. Ch will look through **_path** to find the `cl.exe` command that is used to compile programs. The `LIB` path will be searched for libraries. Similarly, the `INCLUDE` paths will be searched for the header files.

1.1.3.2 Using Borland C++ Compiler

Executable binary programs and dynamical loaded libraries in Windows can also be built using Borland C++ compiler. Examples using Borland C++ compiler. are distributed in Embedded Ch SDK in the directory `CHHOME/toolkit/demos/embedch/Borland`. Assume Ch is installed in the directory `C:\Ch`. To compile the code using Borland C++ compiler, follow the procedures below.

- The command `C:/Ch/bin/make.exe` does not work for Makefile for compiling and linking code in Borland C++ compiler. Make sure the command `make.exe` distributed and bundled with Borland compiler is used in Ch shell. you can type command

  ```
  which make
  ```

  to check which `make.ex` is invoked when in Ch shell. Assume Borland compiler version 5.5 is used. the Ch statement below sets up the command search paths so that `C:/borland/bcc55/bin/make.exe` from Borland will be used in Ch shell.

  ```
  _path = stradd("C:/borland/bcc55/bin;", _path);
  ```

  This statement can be placed in the startup file _chrc in the user's home directory. For the convenience of users, command

  ```
  ch -d
  ```

  copies a startup file from the directory `CHHOME/config` to the user's home directory.

- Header file **ch.h** located in the directory `CHHOME/extern/include`, is needed to compile code using Embedded Ch.

- Ch libraries **ch.lib** and **chsdk.lib** are used to create dynamically loaded libraries using Visual .NET or VC++. Using a Borland compiler, libraries **ch_bc.lib** and **chsdk_bc.lib**, instead of **ch.lib** and **chsdk.lib**, shall be used. These libraries are located in the directory `C:/Ch/extern/lib`.

1.2 Set Paths for Shared Libraries

An executable program has to know where to search for the shared libraries that has suffix `.so` in Unix or dynamically linked libraries `.dll` in Windows. It can only find libraries in the default path. The libraries not located in the default path will not be found. Different operating systems use different macros for library paths. Solaris, Linux, and QNX use the environment variable `LD_LIBRARY_PATH`, Windows uses `PATH` or `_path`, and HP-UX uses `SHLIB_PATH`. Paths are separated by colons. The paths specified by those variables are searched before the default paths.

1.2.1 Dynamically Linked Library Paths for Windows

`PATH` is the environment variable that Windows uses. However, the user can add the library paths to `_path` and those paths will be shown up in both `_path` and `PATH`.

1.2.2 Shared Library Paths for Solaris, Linux, and QNX

The environment variable `LD_LIBRARY_PATH` is used to store the library paths on Solaris, Linux, and QNX. The default value for this environment variable is `/usr/ch/extern/lib`.

1.2.3 Shared Library Paths for HP-UX

HP_UX uses environment variable `SHLIB_PATH`. Examples of how these environment variables are setup can be found in the system startup file `CHHOME/config/chrc`.

1.3 Compile and Link C Programs

Program 1.1, `sample.h`, is the header file of a C program named `func.c`.

```
#ifndef SAMPLE_H
#define SAMPLE_H

/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>
#include <stdio.h>

extern double func1(double x);
extern double func2(double x, double y);

#endif /* SAMPLE_H */
```

Program 1.1: Header file sample.h.

Since `func1()` and `func2()` call functions that are defined in header file **math.h**, **math.h** is included in `sample.h`. There are two function declarations in the header file. `func1()` is defined as an external function that takes an argument of type double and returns a value of type double. Function `func2()` is defined similarly, except that it takes two arguments of type double instead of one. Program1.2, `func.c`, contains the definitions of functions `func1()` and `func2()`.

```
#include "sample.h"

double func1(double x) {
    return 2*sin(x);
}

double func2(double x, double y) {
   return 2*hypot(x, y);
}
```

Program 1.2: Function definitions `func.c`.

The function declarations will be searched in `sample.h` with an `#include` preprocessing directive at the very beginning of Program 1.2. For functions `func1()` and `func2()`, each performs a mathematical computation and returns the result to the calling function. Program 1.3 is an application program that uses `func1()` and `func2()`.

```
#include "sample.h"

int main () {
   double d1, d2;

   d1 = sin(3.0);
   d2 = hypot(3, 4);
   printf("sin(3) = %f\n", d1);
   printf("hypot(3,4) = %f\n", d2);

   d1 = func1(3.0);
   d2 = func2(3, 4);
   printf("func1(3) = %f\n", d1);
   printf("func2(3,4) = %f\n", d2);
}
```

Program 1.3: Testing program (program.c).

To compile the C programs `func.c`, `program.c`, and `sample.h` in Unix, the following makefile, `Makefile`, can be used. A Makefile is a textual file which is used with the make utility to build an executable file or libraries. The commonly used make utilities are the command **make** in Unix, and **nmake** in Windows. The commands **dlcomp** in section C.2 and **dllink** in section C.3 are used in Makefile as default compiler and link-editor. Details of these commands will discussed below. `Makefile, make` and `nmake` are easy to use even for the beginners in our Ch SDK.

```
# Make command program

target: program

# -lm for using functions in libm.so
program: program.o func.o
        ch dllink program program.o func.o  -lm
program.o:  program.c
        ch dlcomp program.dl program.c
func.o:  func.c
        ch dlcomp program.dl func.c
clean:
        rm -f *.o program
```

Program 1.4: Makefile for Unix.

The makefile in Program 1.4 can be used to generate an executable file named as `program`.

The command **dlcomp** and **dllink** are created to hide the different compiler and platforms specifications from the user, and make the cross platform compilation easier and convenient. The command **dlcomp** compiles a C or C++ file and produces a binary object file for a dynamically loaded library. Additional compiler options such as `-DMacro` to define a macro `Macro` can be added at the end of the command **dlcomp**.

Using VC++ in .NET in Windows, the libname following the command **dlcomp** must have a surfix .dl so that the source code will be compiled to use the multi-thread dynamically loaded system library.

Command **dllink** links the object files to create an executable binary program or a dynamically loaded library. It links the object files `program.o` and `func.o` together to generate `program`. Please check more information regarding the commands **dlcomp** and **dllink** in Appendix C.

Additional link options can be added at the end of the command **dllink**. For example, additional libraries to be linked by command **dllink** can be added by options starting with `-l`, The `-lm` option will link the program with functions in the library `libm.so` library, although the mathematical library is linked by command **dllink** by default.

To compile them in Windows, the makefile, `Makefile.win` shown in Program 1.5 can be used.

```
# Make command program

target: program.exe

program.exe: program.obj func.obj
        ch dllink program.exe program.obj func.obj
program.obj:  program.c
        ch dlcomp program.dl program.c
func.obj:  func.c
        ch dlcomp program.dl func.c
clean:
        del  *.obj
        del  program.exe
```

Program 1.5: Makefile for Windows (Makefile.win).

The makefiles for Windows and Unix are written similarly. However, `Makefile.win` generates an object file with file extension `.obj` and an executable program `program.exe` with file extension `.exe`. In Unix, command

```
make
```

will generate `program`, whereas in Windows, command

```
nmake -f makefile
```

or

```
make -f makefile.win
```

will generate `program.exe`. To run the program, simply type the command `program` at the Unix prompt or at the Windows prompt. The output from executing the compiled program is shown below.

```
sin(3) = 0.141120
hypot(3,4) = 5.000000
func1(3) = 0.282240
func2(3,4) = 10.000000
```

1.4 Compile and Link C++ Programs

In above section, we described how to compile and link a C program. This section describes how to compile C++ programs across different platforms. Program 1.6 is the header file for a sample C++ program `program.cpp`.

```
#ifndef HELLOWORLD_H
#define HELLOWORLD_H

void HelloWorld();

#endif
```

Program 1.6: Header file for C++ program (HelloWorld.h).

Program 1.7 contains the definition of the function `HelloWorld()`. This function neither takes argument nor returns anything. `HelloWorld()` simply prints out `Hello World from Ch!`.

```
#include <iostream>
#include "HelloWorld.h"
using std::cout;
using std::endl;

void HelloWorld() {
    cout << "Hello World from Ch!" << endl;
}
```

Program 1.7: Function definition (func.cpp).

The application program is shown in Program 1.8. The header file `HelloWorld.h` is included because it contains the function prototype. The function `main()` makes a call to function `HelloWorld()` and exits by returning a zero.

```
#include <iostream>
#include "HelloWorld.h"

int main()
{
    HelloWorld();
    return 0;
}
```

Program 1.8: Application program (program.cpp).

```
# Make command program

target: program

# -lm for using functions in libm.so
program: program.o func.o
       ch dllink program cplusplus program.o func.o -lm
program.o:  program.cpp
       ch dlcomp program.dl cplusplus program.cpp
func.o:  func.cpp
       ch dlcomp program.dl cplusplus func.cpp
clean:
       rm -f *.o program
```

Program 1.9: Unix Makefile (Makefile).

The makefile for compiling and linking in Unix is shown in Program 1.9. The executable, `program`, that the makefile generates is obtained by compiling `program.cpp` and `func.cpp` and then linking `program.o` and `func.o` together. This Unix makefile and one introduced in the previous section are almost the same, except that this one contains the keyword `cplusplus` and C++ programs have file extension `.cpp`. Adding the keyword `cplusplus` after the name of the executable instructs command **dlcomp** to compile the files with the C++ compiler. To run this makefile, type `make` at the Unix command prompt.

The makefile for Windows is shown in Program 1.10. Type `nmake -f Makefile.win` at the Windows command prompt, the executable `program.exe` will be generated.

```
# Make command program

target: program.exe

program.exe: program.obj func.obj
        ch dllink program.exe cplusplus program.obj func.obj
program.obj:  program.cpp
        ch dlcomp program.dl cplusplus program.cpp
func.obj:  func.cpp
        ch dlcomp program.dl cplusplus func.cpp
clean:
        del  *.obj
        del  program.exe
```

Program 1.10: Windows Makefile (Makefile.win).

The following output will be printed on the screen when `program` is executed.

```
Hello World from Ch!
```

1.5 Use Ch Programs Inside Makefiles

Sometimes, we may want to use a Ch program in a makefile. The makefiles use the programs **dllink** and **dlcomp** in Unix, and **dllink.exe** and **dlcomp.exe** in Windows. There are three ways to run a Ch program in a Makefile, if we assume that `chcmd` is a ch command and its first line is `#!/bin/ch`. (Note that the line `#!/bin/ch` indicates that the file is a Ch program.)

```
1. ch chcmd
2. ch chcmd.ch
3. chcmd.ch
```

The first two options will run in all environments under different operating systems and shells such as C Shell and Korn Shell in Windows. The last format works in Unix platforms only.

Chapter 2

Interfacing Binary Modules Using Dynamically Loaded Libraries

Several special file extensions and types are used in Ch. It is very helpful to know in advance the file extensions in Ch space and C space.

2.1 File Extensions and File Types in Ch Space

File with extension .ch indicates a Ch application, Ch script or Ch program. It is a program written in Ch language, and can be executed in the Ch shell without compiling and linking. The extension of a Ch program is `.ch` by default, it is specified by system variable **_pathext**. For a Ch program named `sample.ch`, the user can type either `sample.ch` or `sample` at the shell prompt to run this program. Ch will search for this program in the paths specified by system variable **_path**. Functions invoked in a Ch program are either generic functions or functions that are defined in the program or function files.

File with extension .h indicates the Ch header files, similar to header files in C. Global variables, function prototypes, macros and data types used in Ch programs and Ch functions are defined in the header files. Ch will search for Ch header files in paths specified by system variable **_ipath**.

File with extension .chf indicates a Ch function file, or **chf file**. It is an ASCII file in which the corresponding Ch function is defined. Its file name shall be the same as the function name, and its extension is specified by system variable **_fpathext**. By default, it is `.chf`. Assume a Ch function named `func1()` is invoked in a Ch program, the corresponding function file can be named as `func1.chf`. Ch will search for this function file in paths specified by system variable **_fpath**. Because a Ch function can load a Dynamically Loaded Library (DLL) discussed below, and then call the binary function in the DLL to implement the function, it is regarded as a dynamic link interface between Ch programs and binary functions. Ch functions can be defined within Ch programs.

All variables and functions in a Ch program, relevant Ch function files (chf files) and Ch header files make up a **Ch Space**.

2.2 File Extensions and File Types in C Space

C/C++ file extensions and file types will be introduced here.

File with extension .c or .cpp indicates a C or C++ function file. It is an ASCII file in which the functions in C or C++ are defined. It usually has the extension of .c for C function file or .cpp for C++ function file.

File with extension .c or .cpp indicates a **chdl file**. It is an ASCII file that includes dynamic link export interface functions in C or C++. It has file extension `.c` for C programs and `.cpp` for C++ programs. If your `chdl` file is not big, C or C++ function files can be merged into `chdl` functions since they have the same file extensions. As an export interface, a chdl function is used to call a C or C++ function in library from Ch script. According to the Ch program convention, a chdl function is named after the corresponding Ch function with the suffix `_chdl`, such as function name `sample_chdl`() in chdl file. The user needs to write this file to build a DLL as discussed below.

File with extension .h indicates a header file for chdl files, C and C++ function files. It has the extension of `.h`. Some global variables, function prototypes and data types are usually defined in header files.

File with extension .so, .sl, .dll, .o, .obj, .lib indicates a shared or static library in which functions exist in binary mode. The library files can be either provided by software vendors or built by the user. The file extension depends on the platform for which the library is designed for, as well as the type of the library – shared or static. Typically it is `.so` for shared library in Solaris, Linux, FreeBSD, QNX, `.sl` for HP-UX, `.dll` for Dynamically Linked Library in Windows, `.o` for relocatable object and `.a` for archive library in Unix, `.obj` for COFF or OMF object in Windows. `.lib` is for library used for linking in Windows.

In most cases, the user can only have shared library or static library instead of the C and C++ source files to interface with Ch. The C/C++ source files do not require modifications to work with Ch. For the integrality of the examples throughout this document, we will provide source code of all component files to run the program.

File with extension .dl indicates a Dynamically Loaded LibraryDLL . It has a different meaning in Ch from the definition in other places. It is a binary file which is built from chdl files, C/C++ files and library files. By convention, a DLL is designated by prefix lib and suffix .dl in Ch, such as libsample.dl. When a DLL is loaded by function **dlopen()** within a Ch program or function, Ch will search for it in paths specified by system variable **_lpath**. Ch script will load DLL (Dynamically Loaded Library) to call the C/C++ library functions.

All symbols in a DLL , including variables and functions in C/C++ header files, C/C++ function files and chdl files, make up a C space.

Of the above file types, chdl file and chf file, which are interfaces between **Ch space** and **C space**, are the main focuses.

Ch space and **C space** have their own name spaces. As a result, a function in C space cannot communicate with a function that is in Ch space directly. However by binding the DLL , a Ch program is able to extend its address space from Ch address space to C address space during execution, and call functions in C space. The memory dynamically allocated in C space by functions such as malloc() can be freed in Ch space using function free(), and vice versa.

2.3 Interfacing C Libraries From Ch Space

2.3.1 A Simple Example

The procedure of calling functions in C/C++ libraries from Ch will be discussed briefly in this section.

To call a C function from Ch, it is necessary to create a wrapper function that can link the Ch and the underlying C function. A wrapper function must be able to do three things:

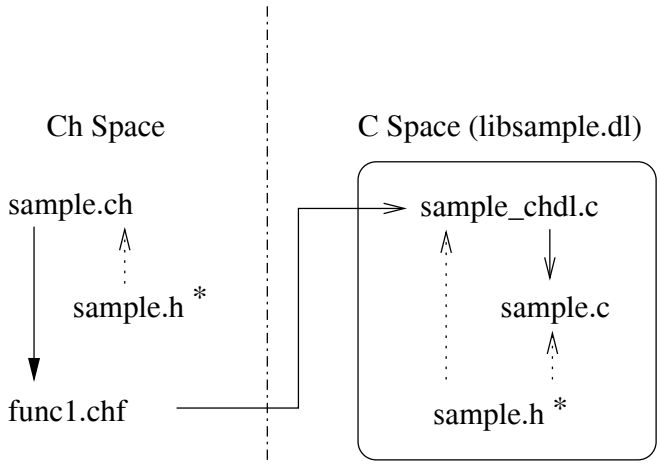
1. It can be called from Ch, pass argument to C function.
2. It can call the C function.
3. It can return value from C function to Ch.

In Ch, the wrapper functions consists of chf functions in Ch space and chdl functions in C space.

Here is how it works. When a Ch program is running directly in Ch space, the Ch header files are loaded first. As it has been mentioned, Ch searches for these header files in the paths specified by system variable **_ipath**. Once a Ch function in the program is invoked, the corresponding chf file which includes the definition of the Ch function is loaded. Ch searches for the chf file in the paths specified by system variable **_fpath**. The Ch function can be completely implemented with the Ch space, or by calling a chdl function in a DLL file. If the chf file needs to load a DLL file by calling function **dlopen()** which is declared in the header file **dlfcn.h**. Ch searches for the DLL file in the paths specified by system variable **_lpath**. Then the chf file can call the corresponding chdl function by its address in the loaded DLL file. Through the chdl function, the C function is called. Most libraries and software packages can be added to the Ch language environment by this mechanism.

The C/C++ libraries are provided by either source code, object file, static library, shared library or dynamically linked library.

The following example illustrates how Ch programs call functions in library(compiled with sample.c).



* In most cases, these two header files are different from each other.

Figure 2.1: Files in Ch and C Spaces.

Figure 2.1 illustrates the relation of Ch and C space. In Ch space, the Ch application `sample.ch` calls the Ch function `func1`() in the file `func1.chf`. The chf file `func1.chf` in Ch space loads the DLL file `libsample.dl` in C space and calls the chdl function `sample_chdl`() by the address in chdl file `sample_chdl.c`. `sample_chdl`() then calls the C function `func1`() in file `sample.c` and finishes the interfacing.

2.3.1.1 Files in Ch Space

Listing 1 — a Ch program `sample.ch`.

```
#include "sample.h"

int main() {
    double d1;

    d1 = func1(3.0);
    printf("func1(3.0) = %f\n", d1);
    return 0;
}
```

At the command line, the user can type command `./sample.ch` or `./sample` in the current working directory to run this program. If the current working directory is included in **_path** and it has the highest searching priority, typing `sample.ch` or `sample` in any directory will run this program. This Ch program includes a header file `sample.h` which defines the prototype of function `func1()`. In the **main()** function, the function `func1()` is invoked to implement some calculations.

Listing 2 — header file `sample.h`.

```
#ifndef SAMPLE_H
#define SAMPLE_H

extern double func1(double);

#endif /* SAMPLE_H */
```

This header file includes the prototype of the function `func1()`. "extern" indicates that this function is defined in C/C++ libraries. This header file is used in both C space and Ch space. Header files will be searched for in the directories specified by **_ipath in Ch space**.

Note that although header files `sample.h` in Ch and C spaces are the same in this simple example, they are different from each other in most cases that we will discuss in the remaining chapters.

Listing 3 — Ch function file `func1.chf`.

```
#include <dlfcn.h>
double func1(double x) {
    void *chSample_handle;
    void *fptr;
    double retval;

    chSample_handle = dlopen("libsample.dl", RTLD_LAZY);
    if(chSample_handle == NULL) {
        fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
        fprintf(_stderr, "cannot get chSample_handle in sample.h\n");
        exit(-1);
    }
```

```
    fptr = dlsym(chSample_handle, "func1_chdl");
    if(fptr == NULL) {
        fprintf(_stderr,
                  "Error: %s(): dlsym(): %s\n", __func__, dlerror());
        return NaN;
    }

    dlrunfun(fptr, &retval, func1, x);

    dlclose(chSample_handle);
    return retval;
}
```

In this program, the function call

```
chSample_handle = dlopen("libsample.dl", RTLD_LAZY);
```

locates and adds the Dynamically Loaded Library(DLL) `libsample.dl` to the address space of the running process. How to build `libsample.dl` will be discussed below. Functions **dlopen()**, **dlsym()**, **dlclose()**, **dlerror()**, and **dlrunfun()** are declared in the head file **dlfcn.h** The first argument of **dlopen**() indicates the library name and the second argument is the flag **RTLD_LAZY**. **RTLD_LAZY** instructs Ch to resolve undefined symbols from the dynamic library when the program is executed. Note that the function **dlopen()** used to load a Ch DLL should use **RTLD_LAZY** as its second argument. The function returns to the process a handle `chSample_handle` which the process uses on subsequent calls to **dlsym**() and **dlclose**(). If the attempt to load the library fails, **dlopen()** returns NULL, and a string describing the most recent error that occurred from any of the dynamical loading functions **dlopen()**, **dlsym()**, and **dlclose()** can be returned by function **dlerror()**. The function call

```
fptr = dlsym(chSample_handle, "func1_chdl");
```

locates the symbol `func1_chdl` within the dynamically loaded library pointed to by handler `chSample_handle`. The application can then reference the data or call the function defined by the symbol using the function **dlrunfun**().

```
dlrunfun(fptr, &retval, func1, x);
```

runs the function in the dynamically loaded object through the address pointed to by `fptr` which is returned by function **dlsym**(). The second argument `&retval` is the address of return value. If the function doesn't have a return value, i.e. its return type is void, NULL should be used as the second argument. If the third argument is the function name itself, in this case, it is `func1`, Ch will check the number and type of the rest of the arguments according to the function prototype. In this case, the function `func1()` takes one argument of type double. If the third argument is NULL, the argument check is ignored. In the case of argument list of variable length, the third argument shall be NULL. The fourth argument `x` is the argument of function `func1()`, which is passed from Ch program `sample.ch` and will be passed to the chdl function `func1_chdl()`. If `func1()` took no argument, the function **dlrunfun()** would take only the first three arguments. If `func1()` took more than one argument, they would be listed after `x` in the argument list of **dlrunfun()**. Before the function returns, the function

```
dlclose(chSample_handle);
```

shall be called to free the dynamically loaded library pointed to by the argument `chSample_handle`. Ch function files will be searched in the directories specified by the system variable **_fpath**.

2.3.1.2 Files in C Space

This section explores the C space. Some mandatory APIs that are used for handling argument list passed from Ch space will be introduced in this section.

As mentioned earlier, Ch function files are used to link the application program to dynamically loaded library during runtime. In other words, the Ch program is able to extend its address space during its execution by binding to the DLL through the dynamic link interface. In the C space, the following chdl file is used for building DLL `libsample.dl`.

Listing 4 — chdl file `sample_chdl.c`.

```
#include <ch.h>
#include <sample.h>
EXPORTCH double func1_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double retval;

    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    retval = func1(x);
    Ch_VaEnd(interp, ap);
    return retval;
}
```

`sample.h` is the same as `sample.h` in Ch space which can be found in Listing 2. The function `func1_chdl()` takes one argument `varg`. If no argument is passed to the C function from the Ch function file, the chdl function takes no argument in the argument list. Otherwise, it always takes one argument of type `void *` even if there are more than one argument to be passed. The macro

```
Ch_VaStart(interp, ap, varg);
```

initializes a Ch interpreter *interp* of type **ChInterp_t** and an object *ap* of type **ChVaList_t**, to represent in the C space for a variable number of arguments in the Ch space, for subsequent use by macro **Ch_VaArg**() and function **Ch_VaEnd**(). These macros and functions are defined in the header file `ch.h`. The **Ch_VaArg** macro expands an expression that has the specified type and the value of the argument in the call. The first invocation of the **Ch_VaArg** macro after the **Ch_VaStart** macro, i.e.

```
x = Ch_VaArg(interp, ap, double);
```

returns the value of the first argument passed from Ch function file `func1.chf`. If more than one argument is passed, successive invocations return the values of the remaining arguments in succession. In other words, the macro **Ch_VaArg**(*interp, ap, double*) returns an argument of double type, from the argument list of a Ch interpreter *interp* and steps *ap*, which is a pointer, to the next one. The expression

```
retval = func1(x);
```

calls the function `func1()` in the C space to calculate the return value. The function `func1()` is listed in the file `sample.c` below.

Listing 5 — C function file `sample.c`.

```
#include <sample.h>
#include <math.h>
double func1(double x) {
    return 2*sin(x);
}
```

In function `func1(x)`, the standard C function **sin()**, defined in header file **math.h**, is called. For the sake of simplity, we can use

```
retval = 2 * sin(x);
```

to replace the function call

```
retval = func1(x);
```

in function `func1_chdl()`.

Listing 6 — Makefile (to create dl file) .

```
target: libsample.dl
libsample.dl: sample.o sample_chdl.o
        ch dllink libsample.dl sample_chdl.o sample.o -lm
sample_chdl.o:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I/dir1/include_dir
sample.o:  sample.c
        ch dlcomp libsample.dl sample.c -I/dir1/include_dir
clean:
        rm -f *.o *.dl
```

The command

```
ch dlcomp libsample.dl sample.c -I/dir1/include_dir
```

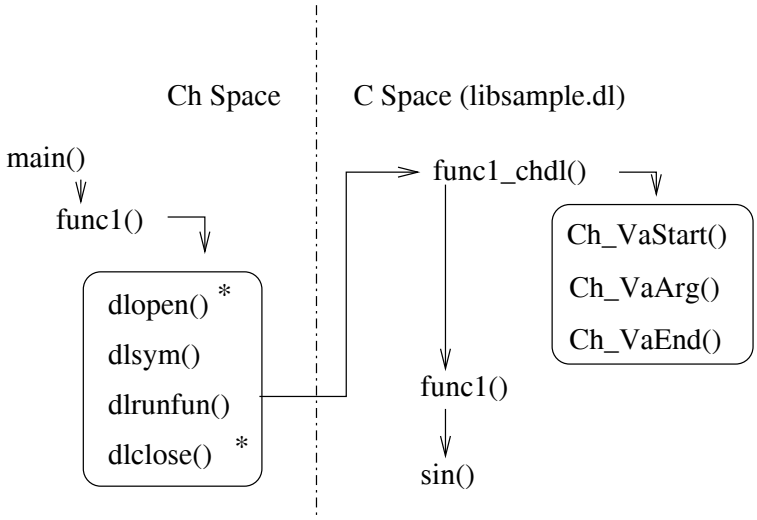
makes the object file `sample.o` from `sample.c` with command **dlcomp**. The argument `libsample` indicates that the generated object file will be used to build the dynamically loaded library `libsample.dl`. The option `-I/dir1/include_dir` gives the additional path /dir1/include_dir where the header file `sample.h` is located. The command

```
ch dllink libsample.dl sample.o -lm
```

builds the dynamically loaded library `libsample.dl` from the object file `sample.o` with command **dllink**. The option `-lm` is for linking function **sin()** in C library **libm.so**. Ch will search for the dynamically loaded libraries in the directories specified in system variable **_lpath** when they are loaded by the function **dlopen()** in Ch function files.

Output from executing `sample.ch` at the command line is

```
> sample.ch
0.282240
>
```



* In mose cases, these two functions are called in the header files

Figure 2.2: Functions and APIs in Ch and C Spaces.

Figure 2.2 illustrates the relation of Ch and C space from a viewpoint of functions and APIs. In the main() function of Ch application `sample.ch`, the Ch function `func1`() is called. Within Ch function `func1`(), four commonly used APIs, **dlopen**(), **dlsym**(), **dlrunfun**() and **dlclose**(), are called to load the DLL file and invoke chdl function `func1_chdl`() by the address. In C space, once the chdl function `func1_chdl`() is called, three argument-handling APIs, **Ch_VaStart**(), **Ch_VaArg**() and **Ch_VaEnd**(), are called to retrieve arguments passed from Ch space. Then the C function `func1`() and C Standard function **sin**() are called to work with these arguments.

Note that we load and release DLL file within the Ch function `func1()` here, since it has only one Ch function in this example. However in cases in which multiple Ch functions exist, it is not efficient and error prone to load and release DLL in different Ch functions. It is strongly recommended to invoke APIs **dlopen**() and **dlclose**() in the header file `sample.h` in Ch space to load the DLL file at the beginning of execution of the program and release the DLL file when the program exits. That is why in most cases, the header files in C space and Ch space are different. Examples with header files of this kind will be given later.

2.3.2 APIs in Header File dlfcn.h in Ch Space

As mentioned in the above example, prototypes for APIs **dlopen()**, **dlsym()**, **dlclose()**, **dlerror()**, and **dlrunfun()** are declared in header **dlfcn.h** file which is located in the directory `CHHOME/include`. These four functions are mandatory for Ch functions which need to load DLL files.

2.3.3 Mandatory APIs in Header File ch.h in C Space

To handle the argument passed from the Ch space, the chdl function in C space needs to call some APIs declared in header file **ch.h** which is located in the directory `CHHOME/extern/include`. These Mandatory APIs include macros **Ch_VaStart**(), **Ch_VaArg**(), and function **Ch_VaEnd**().

2.3.4 APIs Getting Information about Argument List

Besides three mandatory APIs, other APIs for different purposes are also declared in the `ch.h`, such as **Ch_VaCount()** and **Ch_VaDataType()** for getting information about argument list, **Ch_VaArrayDim()** and **Ch_VaArrayExtent()** for handling argument of array type, **Ch_VaFuncArgDataType()** and **Ch_VaFuncArgNum()** for handling argument of pointer to function, **Ch_VaVarArgsCreate()**, **Ch_VaVarArgsDelete()** and **Ch_VaArrayType()** for handling variable argument list, and **Ch_VaUserDefinedAddr()**, **Ch_VaUserDefinedName()**, and **Ch_VaUserDefinedSize()** for handling argument of struct type.

The macro **Ch_VaArg**(*interp, ap, type*) returns one argument from the argument list and steps *ap* to the next one. To handle the current argument, function calls **Ch_VaArrayType**(*interp, ap*), **Ch_VaArrayDim**(*interp, ap*), **Ch_VaArrayExtent**(*interp, ap, i*), **Ch_VaIsFunc**(*interp, ap*), **Ch_VaIsFuncVarArg**(*interp, ap*), **Ch_VaDataType**(*interp, ap*), **Ch_VaFuncArgDataType**(*interp, ap, argnum*), and **Ch_VaFuncArgNum**(*interp, ap*) must be called before **Ch_VaArg**(*interp, ap, type*) is called to step *ap* to the next argument for a Ch interpreter *interp*. Similarly, the function call **Ch_VaCount**(*interp, ap*) calculates the number of the remaining arguments in the argument list, so it should be called before the first calling of **Ch_VaArg**(*interp, ap, type*) to get the total number of arguments passed from the Ch space. The next four sections will describe these APIs one by one.

2.3.4.1 Arguments Number and Type

Function **Ch_VaCount()** can be used in chdl function to determine how many arguments has been actually passed from Ch space. For example,

```
EXPORTCH int func1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int arg_num;
  Ch_VaStart(interp, ap, varg);

  arg_num = Ch_VaCount(interp, ap)
  printf("%d argument(s) has been passed from Ch function.\n",
         arg_num);
  ...
}
```

This function is useful in the case in which the Ch function passes arguments of variable argument list to the chdl function. This kind of case will be discussed in Section 5.5.3.

Function **Ch_VaDataType()** can get the type of an argument or element type of an array. For example,

```
EXPORTCH int func2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int elem_type;

  Ch_VaStart(interp, ap, varg);
  if(Ch_VaDataType(interp, ap) ==  CH_INTTYPE) {
    printf("The type of the element is int\n");
  }
  ...
```

```
}
```

All element types for Ch, such as **CH_INTTYPE** for type int, are defined in header file **ch.h**.

2.3.4.2 Arguments of Pointer To Function

Functions **Ch_VaFuncArgDataType()** and **Ch_VaFuncArgNum()** are designed for handling arguments of pointers to functions. It gets the number of the arguments of the function, which is pointed to by the argument of function pointer. For example,

```
EXPORTCH int func3_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int num;
  void *FUNC_chdl_funptr;
  ChType_t dtype;

  Ch_VaStart(interp, ap, varg);
  num = Ch_VaFuncArgNum(interp, ap);
  for(argnum = 0; argnum < num; argnum++) {
    dtype = Ch_VaFuncArgDataType(interp, ap, argnum);
    if(dtype == CH_INTTYE)
      printf("arg %d is int\n", argnum+1);
    else if(dtype == CH_INTPTRTYPE)
      printf("arg %d is pointer to int \n", argnum+1);
    else if(dtype == CH_DOUBLETYPE)
      printf("arg %d is double \n", argnum+1);
    else
      printf("data type for arg %d is not processed\n", argnum+1);
  }
  FUNC_chdl_funptr = Ch_VaArg(interp, ap, void *);
  ...
}
```

If the first argument passed from a Ch function is a pointer to function, say `funptr()`, after **Ch_VaFuncArgNum()** is called, the variable `num` stores the number of arguments of `funptr()` according to its definition. The data type of each argument of the function is obtained by function **Ch_VaFuncArgDataType()** using a loop. Then, the variable `FUNC_chdl_funptr` is pointed to function `funptr()` after the function call **Ch_VaArg**(*interp, ap, void* *). Note that the function **Ch_VaCount()** is used to get the number of the arguments passed from the Ch function to chdl function itself. This function also plays an important role in handling cases of functions with arguments of pointer to function taking different number of arguments, which will be discussed in section 6.1.8.

2.3.4.3 Arguments of Array Type

Functions **Ch_VaArrayType**(*ap*), **Ch_VaArrayDim(***ap***)** and **Ch_VaArrayExtent(***ap, i***)** are specially designed for getting information of an variable length array argument. They can determine if an argument is an array and then retrieve dimension and extent of an array. For example, in the code fragment below

```
EXPORTCH int func4_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int dim, ext;

  Ch_VaStart(interp, ap, varg);
  if(Ch_VaArrayType(inerp, ap)) {
    dim = Ch_VaArrayDim(interp, ap);
    for(i = 0; i < dim; i++) {
      ext = Ch_VaArrayExtent(interp, ap, i);
      printf("The extent of the %d dimension is %d.\n", i+1, ext);
    }
    ...
  }
```

the for-loop prints out the number of elements in every dimension of the array if the current argument is of array type. The example using these APIs is available in Section 5.7.2.1.

2.3.4.4 Variable Argument Lists

Functions **Ch_VaVarArgsCreate**() and **Ch_VaVarArgsDelete**() are designed specially for handling variable argument lists. Since argument lists in Ch space, called Ch argument lists, are different from those in C space, argument lists passed from Ch space to C space by **dlrunfun**() can be handled only by the functions declared in the header file **ch.h**, such as **Ch_VaStart**() and **Ch_VaArg**(), rather than by the C standard functions **va_start**() and **va_arg**() declared in header file **stdarg.h**. For the same reason, before an argument, say `ap_ch`, which is of type **ChVaList_t** from Ch space is going to be passed to a C function, it should be converted to the corresponding C variable argument list called `ap_c`. The function **Ch_VaVarArgsCreate**() can create a C style variable argument list `ap_c` from a Ch variable argument list `ap`. The C style variable argument list `ap_c` can then be passed to a C function. For example, the statement below

```
ap_c = Ch_VaVarArgsCreate(interp, ap_ch, &memhandle);
```

creates a C variable argument list named `ap_c` from the Ch variable argument list named `ap_ch` for a Ch interpreter `interp`. The variable `memhandle` points to a linked list. It is used to manage the memory allocated in function **Ch_VaVarArgsCreate**().

To release memory allocated by function **Ch_VaVarArgsCreate**(), the function **Ch_VaVarArgsDelete**() should be called when the generated C variable argument list `ap_c` is no longer needed. For example,

```
Ch_VaVarArgsDelete(interp, memhandle);
```

More information and examples about converting Ch variable argument list into C variable argument list are available in section 5.5.1 and Appendix A on page 322.

2.4 Callback Ch Functions From C Space

In the previous sections, we described how to call a C function located in a DLL from a Ch function by loading the DLL and getting the address of the C function. On the other hand, in some cases, we need to call back an existing Ch function (Ch space) from a C function (C space) in a running Ch program (Ch space). It can be illustrated in Figure 2.3. Please note that you cannot call a Ch function from a C program if this

```
/*********************************
* Filename - callch_chdl.c
* Call Ch function chfun() through three methods
*********************************/

#include<ch.h>
#include<stdio.h>

EXPORTCH void callch_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  void *chfunhandle;
  int funret;

  Ch_VaStart(interp, ap, varg);
  /* Method 1: get address of ch function and call it */
  chfunhandle = Ch_SymbolAddrByName(interp, "chfun");
  Ch_CallFuncByAddr(interp, chfunhandle, &funret, 10);
  printf("after Method 1, return value is %d\n\n", funret);

  /* Method 2: call ch function by its name */
  Ch_CallFuncByName(interp, "chfun", &funret, 20);
  printf("after Method 2, return value is %d\n\n", funret);

  Ch_VaEnd(interp, ap);
  return;
}
```

Program 2.1: Example of Calling Ch function from C function (callch_chdl.c).

C program is not called from Ch unless you have an `embedded Ch` from SoftIntegration. Interfacing Ch modules from C space is for call back only.

In this section, through an example, we will explain how to call a Ch function or access variables in the Ch space from a C function in a Ch program.

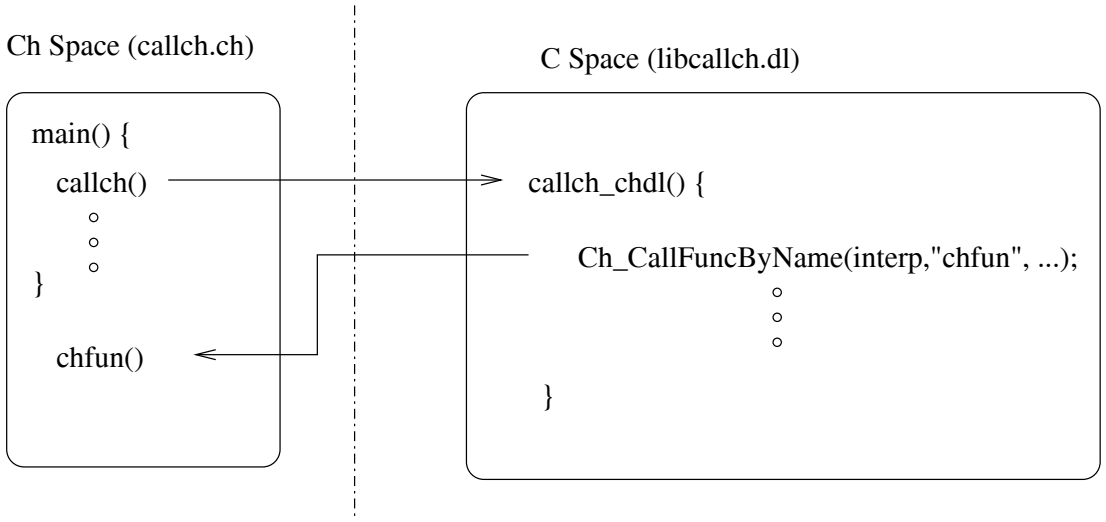


Figure 2.3: Interface Ch modules from C Spaces.

```
CC=cc
INC1=-I/usr/include
INC2=-I/usr/ch/extern/include
LIB=-L/usr/ch/extern/lib
RLOAD=-R/usr/ch/extern/lib

target: libcallch.dl Makefile

libcallch.dl: callch_chdl.o
        ch dllink libcallch.dl callch_chdl.o

callch_chdl.o: callch_chdl.c
        ch dlcomp libcallch.dl callch_chdl.c $(INC1) $(INC2)

clear:
        rm -f *.o *.dl
```

Program 2.2: Makefile for building DLL (Makefile).

Assume that the Ch function to be called in a C function is `chfun()`. The function `chfun()` in Program 2.3 is the Ch function to be called in `callch_chdl()` in Program 2.1 using three different methods described below.

Method 1: The function **Ch_SymbolAddrByName()** is used to get the address of `chfun()`. The first argument is a Ch interpreter. The second argument of **Ch_SymbolAddrByName()** is the function name. The function **Ch_CallFuncByAddr()** is used to call `chfun()` by the address. The second argument of **Ch_CallFuncByAddr()** is the address of `chfun()`, the third is the address of the return value of `chfun()` and the fourth is the argument to be passed to the function `chfun()`.

Method 2: The function **Ch_CallFuncByName()** is used to call `chfun()` by its name. The function call

```
Ch_CallFuncByName(interp,name,retval,arg)
```

is equivalent to

```
Ch_CallFuncByAddr(interp,Ch_SymbolAddrByName(interp,name),retval,arg)
```

The second argument is the name of the function to be called, the third is the address of the return value of `chfun()` and the fourth is the argument to be passed to the function `chfun()`.

Program 2.2 is the Makefile to build the dynamically loaded library `libcallch.dl` using Program 2.1. This DLL is used in Program 2.3.

The output from executing Program 2.3 is displayed in Figure 2.4.

Note: For advanced application using above APIs, you can check Chapters 6 and 8 for more details. Chapter 6 is about handling the pointer to function. Chapter 8 is about how to callback Ch functions with argument of VLAs from C space.

```
/************************************************
*   Filename - callch.ch
*   The function chfun() will be called from c space
************************************************/

#include<dlfcn.h>

void callch() {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libcallch.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                     __func__, dlerror());
    return;
  }

  fptr = dlsym(dlhandle, "callch_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  dlrunfun(fptr, NULL, callch);

  if(dlclose(dlhandle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
                     __func__, dlerror());
    return;
  }

  return;
}

/* this function will be called from a C function */
int chfun(int arg1) {
  printf("in the Ch function, arg1 = %d\n", arg1);
  return arg1 * arg1;
}

int main() {
  callch();
  return 0;
}
```

Program 2.3: Example of calling Ch function from C function (callch.ch).

```
in the Ch function, arg1 = 10
after Method 1, return value is 100

in the Ch function, arg1 = 20
after Method 2, return value is 400
```

Figure 2.4: Output of calling Ch function from C function.

Chapter 3

Calling Multiple Functions From a Library

In the previous chapter, we described how to call a function in the C space from the Ch space. In many applications, there are many functions in a library which will be called from Ch program. We do not want to load and close a DLL time for every function call. In this chapter, we will describe how to build a Dynamically Loaded Library (DLL), chf and chdl files. Often times, the functions in a library are prototyped in a header file, we introduce the command **c2chf** with details described in section C.1 that generates chf and chdl files automatically based on a header file. It can relieve a use from tedious coding.

3.1 Preparing to Build a Dynamically Loaded Library

To interface C/C++ libraries from Ch, Ch does not require any modifications of your C/C++ source code. However, if you do not want to access every function or variables in your C/C++ library, you can eliminate these functions and variables in your Ch header. It can save lots of your effort in coding.

The purpose of this section is to guide users toward building a library. As we walk through this process, the users will gain a better understanding of how Ch interface the C functions in a library. We will use a C program as an example.

Program 3.1 is a C header file. It includes two function prototypes `func1` and `func2`.

```
#ifndef SAMPLE_H
#define SAMPLE_H

/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>
#include <stdio.h>

extern double func1(double x);
extern double func2(double x, double y);

#endif /* SAMPLE_H */
```

Program 3.1: C header file (sample.h).

Program 3.2 contains the definitions of functions `func1()` and `func2()`. Each of these functions takes an argument of type double and returns a value to its corresponding calling function after performing a mathematical computation.

```
#include "sample.h"

double func1(double x) {
    return 2*sin(x);
}

double func2(double x, double y) {
   return 2*hypot(x, y);
}
```

Program 3.2: C Function definition (func.c).

Program 3.1 and Program 3.2 can be compiled as a library in C.

Program 3.3 is a Ch program that wants to call two functions `func1()` and `func2()`. In addition, it calls two mathematical functions `sin()` and `hypot()` declared in the standard C library header file **math.h**.

```
#include "sample.h"

int main () {
   double d1, d2;

   d1 = sin(3.0);
   d2 = hypot(3, 4);
   printf("sin(3) = %f\n", d1);
   printf("hypot(3,4) = %f\n", d2);

   d1 = func1(3.0);
   d2 = func2(3, 4);
   printf("func1(3) = %f\n", d1);
   printf("func2(3,4) = %f\n", d2);
}
```

Program 3.3: Ch Program (program.ch).

When the Ch program `program.ch` is executed in the Ch space, it does not call the C header file as shown in Program 3.1. Instead, it calls its Ch header file. Program 3.4 is the Ch header file `sample.h` that Program 3.3 will use at execution time. You may notice that `sample.h` is quite different from the declaration in Chapter 1.

```
#ifndef SAMPLE_H
#define SAMPLE_H

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

#ifdef _CH_
#include <dlfcn.h>
void *_Chsample_handle;
_Chsample_handle = dlopen("libsample.dl", RTLD_LAZY);
if(_Chsample_handle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get _Chsample_handle in sample.h\n");
   exit(-1);
}
void _dlclose_sample(void) {
   dlclose(_Chsample_handle);
}
atexit(_dlclose_sample);
#endif

extern double func1(double);
extern double func2(double, double);

#endif /* SAMPLE_H */
```

Program 3.4: Ch header file (sample.h).

Note that the header file in Program 3.4 in the Ch space contains not only the same function prototypes as the Program 3.1 in the C space, but also codes that are used to access the dynamically loaded library `libsample.dl`. The statement

```
_Chsample_handle = dlopen("libsample.dl", RTLD_LAZY);
```

opens the library `libsample.dl` using the function **dlopen()** to load the binary module that contain functions `func1()` and `func2()` in the C space. `libsample.dl` will be discussed in the next section. The statement

```
atexit(_dlclose_sample);
```

makes the user defined function `_dlclose_sample`() be called at the end of execution of program `program.ch` to close the dynamically loaded library.

```
#ifndef SAMPLE_H
#define SAMPLE_H

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

#ifdef _CH_
#include <chdl.h>
LOAD_CHDL(sample)
#endif

extern double func1(double);
extern double func2(double, double);

#endif /* SAMPLE_H */
```

Program 3.5: An alternative Ch header file (sample.h).

An alternative form of Ch header file `sample.h` is shown in Program 3.5. The header file `sample.h` in Program 3.5 is the same as that in Program 3.4. The macro `LOAD_CHDL()` is defined in the header file `chdl.h`.

Since the application program makes calls to two different functions, two function files will have to be written. Programs 3.6 and 3.7 are the two Ch function files for functions `func1()` and `func2()`, respectively. They will be called from Program 3.3.

```
double func1(double x) {
    void *fptr;
    double retval;

    fptr = dlsym(_Chsample_handle, "func1_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return NaN;
    }
    dlrunfun(fptr, &retval,func1, x);
    return retval;
}
```

Program 3.6: Ch function file (func1.chf).

```
double func2(double x, double y) {
    void *fptr;
    double retval;

    fptr = dlsym(_Chsample_handle, "func2_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return NaN;
    }
    dlrunfun(fptr, &retval,func2, x, y);
    return retval;
}
```

Program 3.7: Ch function file (func2.chf).

Programs 3.6 and 3.7 each makes a call to its corresponding chdl function using **dlrunfun()**. Then the chdl file shown in Program 3.8 in turn call the binary C functions.

```
#include <ch.h>
#include <sample.h>

EXPORTCH double func1_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double retval;

    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    retval = func1(x);
    Ch_VaEnd(interp, ap);
    return retval;
}

EXPORTCH double func2_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double y;
    double retval;

    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    y = Ch_VaArg(interp, ap, double);
    retval = func2(x, y);
    Ch_VaEnd(interp, ap);
    return retval;
}
```

Program 3.8: chdl file (sample_chdl.c).

The chdl functions in Program 3.8 are written according to the definitions of the two C functions in Program 3.2. Functions `func1_chdl()` and `func2_chdl()` both have arguments and return values.

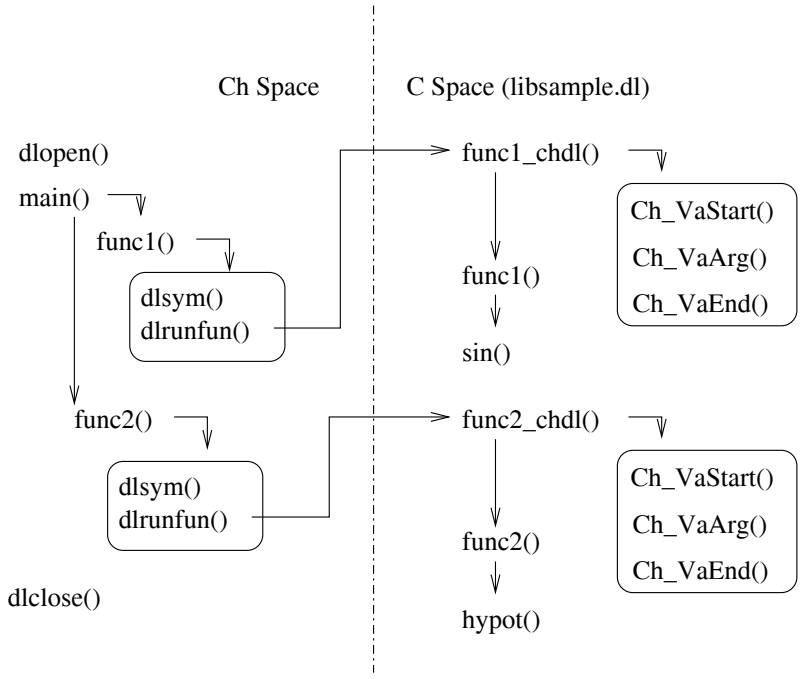


Figure 3.1: Functions and APIs in Ch and C Spaces.

They obtain the arguments passed from the Ch space using **Ch_VaArg()** and then call the C functions with those arguments.

The relations of functions and APIs used in this example are illustrated by Figures 3.1. The function **dlopen**() is involked in the header file `sample.h` before the main() function is called, and the function **dlclose**() is called before the program exits. In Ch space, the function `func1`() calls chdl function `func1_chdl`() in DLL by APIs **dlsym**() and **dlrunfun**(). In C space, the chdl function `func1_chdl`() calls **func1**(), and then `func1`() calls **sin**(). After the Ch function `func1`() returns, the next Ch function, `func2()`, is called in the same procedure.

3.2 Building a Dynamically Loaded Library

A dynamically loaded library (DLL) is a library of functions in binary mode. It is built from the chdl files and bound to the running programs at runtime. Program 3.9 is the Unix Makefile for building a DLL named `libsample.dl`.

```
# build dynamically loaded  lib libsample.dl

target: libsample.dl

# -lm for func1() and func2() using functions in lib libm.so
libsample.dl: sample_chdl.o func.o
        ch dllink libsample.dl sample_chdl.o func.o -lm
sample_chdl.o:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I../C
func.o:  ../C/func.c
        ch dlcomp libsample.dl ../C/func.c -I../C
clean:
        rm -f *.o *.dl
```

Program 3.9: Makefile for Unix (Makefile).

The Makefile shown in Program 3.9 creates the `libsample.dl` in Unix, assuming that `sample_chdl.c` is stored in the current directory and that `func.c` is stored in the parent's `C` directory. The `-I` option directs the C compiler inside program **dlcomp** to find the header file `sample.h` in the parent's `C` directory. The `-lm` option tells the linker inside program **dllink** to look in the library `libm.so` to find functions `sin()` and `hypot()` used in functions `func 1()` and `func2()`.

```
# build dynamically loaded  lib libsample.dl

target: libsample.dl

libsample.dl: sample_chdl.obj func.obj
        ch dllink libsample.dl sample_chdl.obj func.obj
sample_chdl.obj:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I../C
func.obj:  ../C/func.c
        ch dlcomp libsample.dl ../C/func.c -I../C
clean:
        del *.obj
        del *.exp
        del *.lib
        del *.dl
```

Program 3.10: Makefile for Windows (Makefile.win).

Program 3.10 is the makefile used in Windows. The windows makefile builds `libsample.dl` in a similar manner described in the Unix makefile section. The object file has file extension `.obj` in Windows, instead of `.o` in Unix.

We assume that the files are located in their proper directories specified in the makefiles. To run the Makefile, simply type

```
make
```

at the Unix command prompt and

```
nmake -f Makefile.win
```

at the Windows command prompt.

In makefiles shown in Programs 3.9, and 3.10, We compile program `func.c` to create an object file, which is then linked with the object file from program `sample_chdl.c` to create dynamically loadable file `libsample.dl`. Instead, the dynamically loadable file `libsample.dl` can be created with an object file from program `sample_chdl.c` and dynamical or static library. In this example, the static library `libfunc.a` is created using commands ar and `ranlib` from the object file of program `func.c` which contains functions `func1()` and `func2()` as shown in the makefile in Program 3.11 for Unix.

```
# build dynamically loaded lib libsample.dl using libfunc.a

target: libsample.dl

# func1() and func2() is located in libfunc.a
# -lm for func1() and func2() using functions in lib libm.so
libsample.dl: sample_chdl.o libfunc.a
        ch dllink libsample.dl sample_chdl.o libfunc.a -lm
sample_chdl.o:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I../C
libfunc.a: ../C/func.c
        cc -c ../C/func.c  -I../C
        ar -r libfunc.a func.o
        ranlib libfunc.a
clean:
        rm -f *.o *.dl libfunc.a
```

Program 3.11: Makefile for Unix (Makefile_lib).

The makefile in Program 3.12 illustrates how to create dynamically loadable library `libsample.dl` in Windows using functions in a dynamically linked library. The dynamically linked library `func.dll` with symbols in `func.lib` is created from Program 3.13 by Visual C++ compiler `cl` and linker `link`. Unlike functions in Program 3.2, functions `func1()` and `func2()` in. program `func_lib.c` are exported through the type qualifier `__declspec(dllexport)`. When the object file from program `func_lib.c` is linked to create the runtime dynamical library `func.dll`, file `func.lib` will also be generated. Library file `func.lib` is used to resolve the external symbols when the dynamically loadable library `libsample.dl` is created.

```
# build dynamically loaded lib libsample.dl using func.lib

target: libsample.dl

libsample.dl: sample_chdl.obj func.obj
        ch dllink libsample.dl sample_chdl.obj func.lib
sample_chdl.obj:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I../C
# create func.lib and func.dll
func.obj:  func_lib.c
        cl /c /Fofunc.obj func_lib.c
        link /DLL /OUT:func.dll func.obj
clean:
        del *.obj
        del *.exp
        del *.lib
        del *.dll
        del *.dl
```

Program 3.12: Makefile for Windows (Makefile_lib.win).

```
/* This file is used for creating func.lib and func.dll in Windows only.
   Move func.dll in a directory pointed by the enviornment variable PATH */

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

__declspec(dllexport)
double func1(double x) {
    return 2*sin(x);
}

__declspec(dllexport)
double func2(double x, double y) {
   return 2*hypot(x, y);
}
```

Program 3.13: C Function definition (func_lib.c) for Windows.

We assume that the files are located in their proper directories specified in the makefiles. To run the makefile, simply type

```
make -f Makefile_lib
```

at the Unix command prompt and

```
nmake -f Makefile_lib.win
```

at the Windows command prompt to create the dynamically loadable library `libsample.dl`.

In many applications, the source code for functions are not available. Only dynamical or static library such as `libfunc.a` for Unix and `func.lib` for Windows and relevant header files are provided by software vendors. Ch can also interface such binary functions located in a static or dynamically loaded library as shown in makefiles in Programs 3.11 and 3.12 for Unix and Windows, respectively.

3.3 Setting Up the Paths and Running Ch Programs

To run the above program, we have to include ch header file, chf file and DLL in the same directory as Ch application program. There might be too many files in a same directory once the program gets bigger.

Certain paths can be set up for a Ch program to run correctly. When the program runs, it needs to locate and use the specific libraries, Ch function files, and header files that it depends on.

Ch searchs the directories specified in system variables **_fpath**, **_ipath**, **_lpath**, and **_path** for function files, header files, dynamically loaded library, and commands, respectively.

To check the current values of these system variables, the user can type variable names in the command mode directly. For example, the user can check if the directories in which the function files are located are included in system variable **_fpath** by the following command.

```
> _fpath
```

To add a directory for the system variables, the user can use the command `stradd`. For example, if the directory for functions files, say **/mydir/lib**, is not included, the user can add it by using the generic function **stradd()** as follows.

```
> _fpath = stradd(_fpath, "/mydir/lib;")
```

3.3.1 Setting Up Paths for a Toolkit

Files in a toolkit are typicall copied to directories specified by the system paths. Toolkits are mainly distributed by SoftIntegration.

The user can take advantage of existing toolkit path set up by default. You can simply copy the Ch program header files to the directory `CHHOME/toolkit/include`, copy the program Dynamically Loaded Files (.dl) to the directory `CHHOME/toolkit/dl`. A Ch program header file need a slight modification as described below. Binary programs and dynamically loaded library (.DLL) in Windows are copied into `CHHOME/toolkit/bin` or `CHHOME/toolkit/sbin`. Programs in the directory `CHHOME/toolkit/sbin` can be accessed by Safe Ch. By default, `CHHOME/toolkit/include`, `CHHOME/toolkit/dl`, and `CHHOME/toolkit/bin` and `CHHOME/toolkit/sbin` have been included in the system variables **_ipath**, **_lpath**, and **_path**, respectively.

We use different method to handle the the function files since the Ch program usually contains too many Ch functions files. The solution in Ch is to add preprocessing directives `#pragma _fpath` into the corresponding Ch program header files.

For example, if the header file `gtk.h` for the GTK toolkit contains

```
#ifndef __GTK_H__
#define __GTK_H__
#pragma _fpath <GTK/gtk>
...
#endif /* __GTK_H__ */
```

The Ch program containing `gtk.h` will add the directory `CHHOME/toolkit/lib/fpathname` into the system variable **_fpath** of the subshell in which the Ch program is running The Ch program will search the directory `CHHOME/toolkit/lib/GTK/gtk` for function files.

The other form of the pragma directive,

```
#pragma _fpath /dir1/dir2/fpathname
```

can add the absolute path name `/dir1/dir2/fpathname` into the system variable **_fpath** of the subshell.

3.3.2 Setting Up Paths for a Package

Different ways of setting up paths for a package are described below.

1. Modify system paths by system variables **_path**, **_fpath**, **_ipath**, and **_lpath**. Assuming that the executable file `program.ch` is in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/bin`,
`libsample.dl` is in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/dl`,
`func1.chf` and `func2.chf` are in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/lib`,
and `sample.h` is in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/include`.

The following code need to be added to either `CHHOME/config/chrc` or `~/_chrc`. Thus, these path settings will be effective whenever Ch is started.

```
_path = stradd(_path, "/usr/local/ch/toolkit/demos/SDK/command/c2chf/bin;");
_lpath = stradd(_lpath, "/usr/local/ch/toolkit/demos/SDK/command/c2chf/dl;");
_fpath = stradd(_fpath, "/usr/local/ch/toolkit/demos/SDK/command/c2chf/lib;");
_ipath = stradd(_ipath, "/usr/local/ch/toolkit/demos/SDK/command/c2chf/include;");
```

2. Use **_ppath** in conjunction with `#pragma package <packagename>`. **_ppath** contains directories for packages.

Assuming that the executable file `program.ch` is in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/bin`,
`libsample.dl` is in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/dl`,
`func1.chf` and `func2.chf` are in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/lib`,
and `sample.h` is in
`/usr/local/ch/toolkit/demos/SDK/command/c2chf/include`.

The following code need to be added to the startup file `CHHOME/config/chrc` or `~/_chrc`.

```
_ppath = stradd(_ppath, "/usr/local/ch/toolkit/demos/SDK/command;");
_ipath = stradd(_ipath, "/usr/local/ch/toolkit/demos/SDK/command/c2chf/include;");
```

Program `pkginstall.ch` described in section C.2 can be executed for the above setup during the installation of the package `c2chf` as shown below:

```
> pkginstall.ch -d /usr/local/ch/toolkit/demos/SDK/command c2chf
```

In addition, the preprocessing directive

```
#pragma package <c2chf>
```

has to be included at the beginning of the Ch program or its header file. `#pramga package <c2chf>` adds `_ppath/c2chf/bin` to **_path**, `_ppath/c2chf/lib` to **_fpath**, `_ppath/c2chf/include` to **_ipath**, and `_ppath/c2chf/dl` to **_lpath**. Program 3.14 is a Ch program based on the application program shown in Program 3.3.

```
#pragma package <c2chf>

#include <sample.h>

int main () {
   double d1, d2;

   d1 = sin(3.0);
   d2 = hypot(3, 4);
   printf("sin(3) = %f\n", d1);
   printf("hypot(3,4) = %f\n", d2);

   d1 = func1(3.0);
   d2 = func2(3, 4);
   printf("func1(3) = %f\n", d1);
   printf("func2(3,4) = %f\n", d2);
}
```

Program 3.14: Testing program (program.ch).

There is another option to run the Program 3.14 without setting the **_ppath**. Just change preprocessing directive from

```
#pragma package <c2chf>
```

to

```
#pragma package "/usr/local/ch/toolkit/demos/SDK/command/c2chf"
```

at the beginning of the Program 3.14. By specifying the absolute path, Ch can find all the file components in the subdirectories of the directory `/usr/local/ch/toolkit/demos/SDK/command/c2chf` without setting any system path.

3. The simplest way is to use program `pkginstall.ch`, described in section C.2, to install a package into the directory CHHOME/package. For example, a package `chsample` can be installed by

```
> pkginstall.ch chsample
```

and uninstalled by

```
> pkginstall.ch -u chsample
```

as described in section 3.5.

3.4 Generating chf and chdl Files Using Command c2chf

It is usually not necessary to access every function in a library. In most cases, you might only need a relatively small subset of library. You may need to eliminate the unnecessary functions and variables in your Ch header first to interface C library from Ch space. However, it is inconvenient to write the chf and chdl files for every function by hand. To speed up application development, the command **c2chf** with details described in section C.1 can be used. Command **c2chf** generates the chf and chdl files automatically based on a header file that the user provides.

You can either use the existing C library header file or modify an existing header file. This header file should only contain all the prototypes of those functions that will be used in the application program without macros and typedef statements.

The header file, `sample.h`, with the contents below is an example of what a typical header file would look like.

```
extern double func1(double x);
extern double func2(double x, double y);
```

Type qualifier `extern` will be ignored by command `c2chf`. The above header readily for processing by command `c2chf` can be obtained by function `processhfile()`. For example, for the two commands below typed at the command prompt,

```
> processhfile("extern", 0, ";", "C/sample.h", "sample.h", NULL);
> c2chf sample.h
```

function `processhfile()` with details described in section D.2 will create a new header file `sample.h` in the current directory based on the original C header file in the C directory first. Command `c2chf` then creates a directory with the name of the header file's prefix. Inside the directory there would be the chf files and a chdl file for creating a dynamically loaded library. The number of chf files in the directory will match the number of function prototypes written in the header file.

The command **c2chf** has limitations as described in section C.1. It is able to handle most C functions. But, it can not handle functions with argument of pointer to function. You need to write the code manually for this case.

In this example, the directory being created by **c2chf** is `sample` and its contents are `func1.chf`, `func2.chf`, and `sample_chdl.c`. We have used these three generated files in the previous sections.

More information about function `processhfile()` and command **c2chf** can be found in sections D.2 and C.1, respectively.

3.5 Creating Interface to C Library Using a Script Program

To make the process of creating and maintaining a dynamically loaded library and function files simpler, we will be using a script program. In order to demonstrate the use of a script program, we modify the programs used at the begin of the chapter by adding a new function. The new function `func3()` has an argument of pointer to a function which has one argument and returns an integer. A function with pointer to function cannot be handled automatically by the program `c2chf` , so it needs to be handled manually. Program 3.15 is the new C header file which includes three function prototypes. Program 3.16 contains the definitions of functions `func1()`,`func2()`, and `func3()`. Program 3.17 is a C program that calls these three functions and prints their results on the screen.

3.5.1 Creating a Package Using `pkgcreate.ch`

```
#ifndef SAMPLE_H
#define SAMPLE_H

/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>
#include <stdio.h>

#if defined(_WIN32)
#define EXPORT _declspec(dllexport)
#else
#define EXPORT
#endif

typedef int (*FUNPTR)(int n);

EXPORT extern double func1(double x);
EXPORT extern double func2(double x, double y);
EXPORT extern int func3(FUNPTR fp, int n);  /* arg is pointer to func */

#endif /* SAMPLE_H */
```

Program 3.15: C header file (sample.h).

```
#include "sample.h"

EXPORT
double func1(double x) {
    return 2*sin(x);
}

EXPORT
double func2(double x, double y) {
   return 2*hypot(x, y);
}

EXPORT
int func3(FUNPTR fp, int n) {
   fp(n);
   return 2*n;
}
```

Program 3.16: C Function definition (func.c).

```
#include "sample.h"

int func(int n) {
   printf("n in func() = %d\n", n);
   return 0;
}

int main () {
   double d1, d2;
   int ret;

   d1 = sin(3.0);
   d2 = hypot(3, 4);
   printf("sin(3) = %f\n", d1);
   printf("hypot(3,4) = %f\n", d2);

   d1 = func1(3.0);
   d2 = func2(3, 4);
   printf("func1(3) = %f\n", d1);
   printf("func2(3,4) = %f\n", d2);

   ret = func3(func, 10);
   printf("ret = %d\n", ret);
}
```

Program 3.17: C application program (program.c).

```
#!/bin/ch
#include <unistd.h>   // for access()
#include <dlfcn.h>    // for dlopen()
#include <chshell.h>  // for chinfo()

int addChdlHeader(char *headerfile, char *macro, char *chdlheader);

string_t pkgname="chsample";// define package name
chinfo_t info;              // for Ch version number
string_t cwd = _cwd;        // the current working directory
string_t debugFile;         // compilation debug information in Windows
string_t debug;
string_t makecmd;
#if defined(_WIN32_)
    debug=">nul 2>nul";         // surpress messages during cleaning  in Windows
    debugFile =">logfile 2>&1"; // compilation debug information in 'logfile'
    makecmd = "nmake -f Makefile.win";
#else
    makecmd = "make -f Makefile";
#endif

//make sure pgkcreate.ch is run from the current working directory
if(access("pkgcreate.ch", R_OK)) {
    echo Run ./pkgcreate.ch in the current directory.
    exit(-1);
}
// run this script in proper Ch version
chinfo(&info);
if ((info.vermajor*100+ info.verminor*10 +  info.vermicro) < 501) {
   echo "To run this script, you need to install Ch version 5.0.1.12201 or higher"
   echo "You can download the latest version from http://www.softintegration.com/download"
   exit(-1);
}

//echo clean up existing directory and create new ones
if (!access(pkgname, F_OK))
  rm -rf $pkgname
mkdir $pkgname
mkdir $pkgname/dl $pkgname/lib $pkgname/include $pkgname/demos $pkgname/bin

//echo copying demo programs ...
cp -rf demos/* $pkgname/demos

//echo copying and modify header files ...
cp include/sample.h $pkgname/include
char *macro ="#define SAMPLE_H";
char *chdlheader =
   "\n\n"
   "#ifdef _CH_\n"
   "#pragma package <chsample> \n"
   "#include <chdl.h> \n"
   "LOAD_CHDL(sample);\n"
   "#endif \n\n";
addChdlHeader(stradd(pkgname, "/include/sample.h"), macro, chdlheader);


//echo extracting function prototypes ...
processhfile("EXPORT", 0,  ";",  stradd(pkgname, "/include/sample.h"), "chfcreate/sample.h", NULL);
//echo removing special functions
removeFuncProto("chfcreate/sample.h", "func3", 0);
//echo generating sample_chdl.c in c/sample and *.chf in $pkgname/lib
c2chf chfcreate/sample.h -h _Chsample_handle -r chfcreate/sample_err -o c c -o chf $pkgname/lib
//echo patching special .chf files
cp -f chfhandmade/*.chf $pkgname/lib
chmod 644 $pkgname/lib/*.chf
```

Program 3.18: Script file (pkgcreate.ch).

```
//echo making .dl files
cd c
$makecmd clean $debug
$makecmd $debugFile

cd $cwd
//echo .DLL path and testing .dl file
//#pragma exec _path=stradd(_path, "chsample/bin");
if (dlopen("./c/libsample.dl", RTLD_LAZY) == NULL) {
    printf("Error: test of loading libsample.dl: %s\n", dlerror());
    exit(-1);
}
cd c
mv -f libsample.dl $cwd/$pkgname/dl
$makecmd clean $debug
echo package $pkgname created successfully!

/*****************************************************************
 * * This function will add the required preprocessing directives
 * * needed by Ch to a header file.
 * * headerfile:   header file to be changed
 * * macro:        token to be recoganized for insertion of chdlheader
 * * chdlheader:   preprocessing directives to be inserted
 * ***************************************************************/
int addChdlHeader(char *headerfile, char *macro, char *chdlheader)
{
   char *stop_ptr;
   string_t tempfile, token;
   string_t result;
   FILE *stream;

   tempfile = ‘‘cat $headerfile‘‘;
   token = strstr( tempfile, macro);
   stop_ptr = strstr(token, "\n");
   strncpy(result, tempfile, strlen(tempfile)-strlen(stop_ptr));
   result=stradd(result, chdlheader);
   result=stradd(result, stop_ptr);
   if(!(stream = fopen("_tmpfile", "w")))
   {
      perror("_tmpfile");
      return -1;
   }
   fprintf(stream, "%s", result);
   fclose(stream);
   mv -f _tmpfile $(headerfile)
   return 0;
}
```

Program 3.18: Script file (pkgcreate.ch) (cont.).

Program 3.18 is the Ch script which creates the dynamically loaded library. Before the script can be executed, the proper directory structure must be in place. Figure3.2 shows how the directory structure is set up. Start by creating this directory structure and inserting your files as shown. After running `pkgcreate.ch` in the current directory, the pakage `chsample` in the current directory is created as shown in Figure3.2 Under the directory `chsample`, the directory `demos` contains demo programs in C that are ready to run in Ch. The directory `dl` contains the dynamically loaded library. The directory `lib` contains the function files. The directory `include` contains the header files used in the program. The contents in this directory are copied to CHHOME/toolkit/include when the package is installed using program `pkginstall.ch`.

Now let's take a closer look at the script program to see exactly what is being done. The script begins with the lines

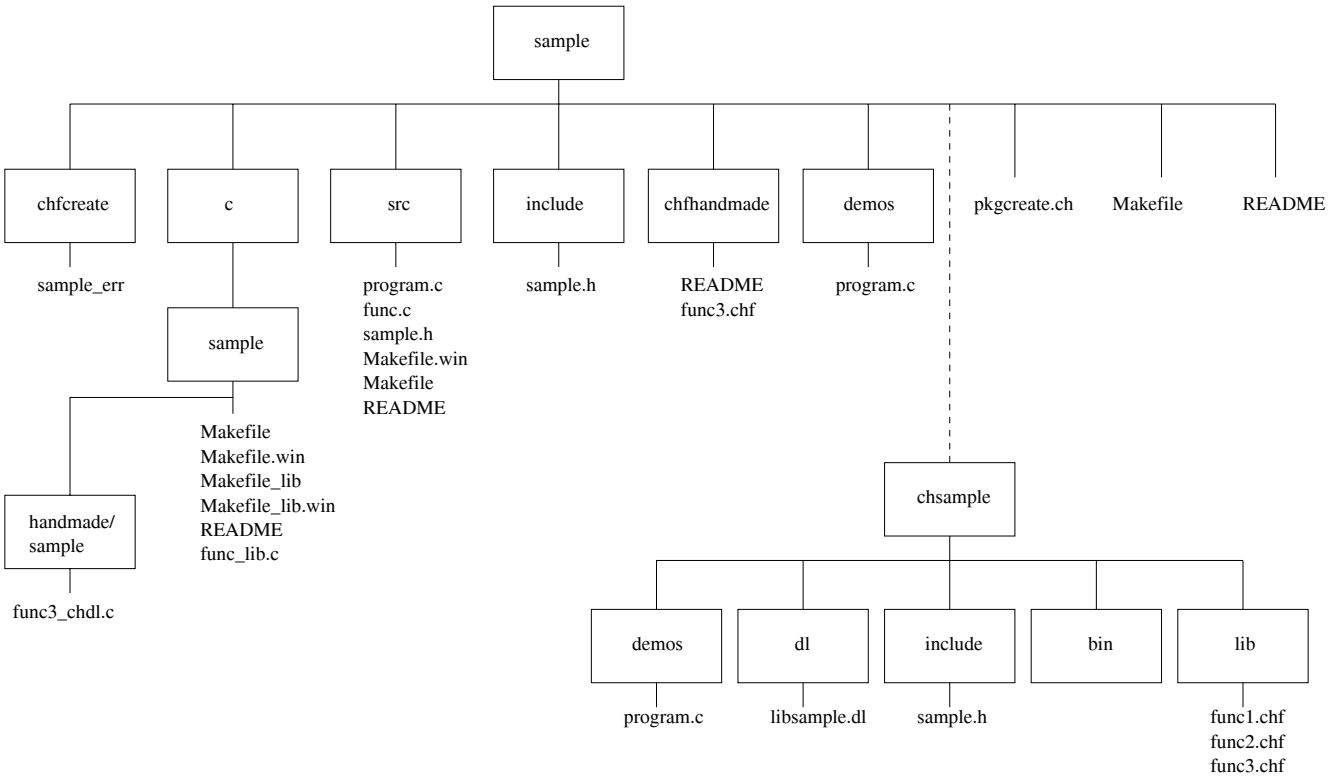


Figure 3.2: Directory structure creating package `chsample` using the script pkgcreate.ch.

```
#!/bin/ch
#include <unistd.h>   // for access()
#include <dlfcn.h>    // for dlopen()
#include <chshell.h>  // for chinfo()
```

Starting the program with `#!/bin/ch`, rather than `main()` lets the system know that this is a Ch script. This script can be executed in Ch and other shells. Proper header files are included in the program, because it contains system and process functions that will be called in the script program.

After some variables are declared and initialized, the next few lines contain an `if` statement.

```
if(access("pkgcreate.ch", R_OK)) {
    echo Run ./pkgcreate.ch in the current directory.
    exit(-1);
}
```

The function `access()` checks the accessibility of the file named by the pathname pointed to by the first argument. The second argument `R_OK` tells the function to check for read permission. If `pkgcreate.ch` exists and the user has read permission, `access()` will return zero. If either the file does not exist, or the user does not have read permission, the function will return 1 and a message will be displayed and the script will exit.

Function `chinfo()` obtained the information about Ch. The script can only run in Ch version 5.0.1.12201 or higher.

If the directory `chsample` exists, it will be removed first. This directory for the package and its subdirectories as shown in Figure 3.2. will be created then. Afterwards, demo programs are copied to the package `chsample` by the command.

```
cp -rf demos/* $pkgname/demos
```

Then, proper header files are copied to the package `chsample` by the commands below.

```
cp include/sample.h  $pkgname/include
char *macro ="#define SAMPLE_H";
char *chdlheader =
   "\n\n"
   "#ifdef _CH_\n"
   "#pragma package <chsample> \n"
   "#include <chdl.h> \n"
   "LOAD_CHDL(sample);\n"
   "#endif \n\n";
addChdlHeader(stradd(pkgname, "/include/sample.h"), macro, chdlheader);
```

copies heade file `include/sample.h` to `chsample/include/sample.h`. The header file `chsample/include/sample.h` for Ch is then modified using function `addChdlHeader()`. The first argument is the header file to be modified. The second argument is the symbol, `"#define SAMPLE_H"`, to be recoganized for insertion of the preprocessing directives in the third argument. The difference between the original header file `include/sample.h` in Program 3.15 and modified header file `chsample/include/sample.h` in Program 3.19 is that `chsample/include/sample.h` contains the directive in the third argument of function `addChdlHeader()`.

```
#ifdef _CH_
#pragma package <chsample>
#include <chdl.h>
LOAD_CHDL(sample)
#endif
```

The Ch PNG package, available at `http://www.softintegration.com/products/thirdparty`, also contains a code that can change a function prototype from

```
extern PNG_EXPORT(rettype ,func) PNGARG((type arg));
```

to

```
extern rettype func(type arg);
```

so that it can be processed by command **c2chf**. The Ch OpenCV package has a sample code to change

```
CVAPI(data_type) func(arg_list);
```

to

```
data_type func(arg_list);
```

Function call

```
processhfile("EXPORT", 0,  ";", "../include/sample.h",
             "chfcreate/sample.h", NULL);
```

```
#ifndef SAMPLE_H
#define SAMPLE_H

#ifdef _CH_
#pragma package <chsample>
#include <chdl.h>
LOAD_CHDL(sample);
#endif



/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>
#include <stdio.h>

#if defined(_WIN32)
#define EXPORT _declspec(dllexport)
#else
#define EXPORT
#endif

typedef int (*FUNPTR)(int n);

EXPORT extern double func1(double x);
EXPORT extern double func2(double x, double y);
EXPORT extern int func3(FUNPTR fp, int n);  /* arg is pointer to func */

#endif /* SAMPLE_H */
```

Program 3.19: Ch header file (sample.h).

```
 extern double func1(double x);
 extern double func2(double x, double y);
```

Program 3.20: Processed C header file (sample.h).

creates a new header file in the directory `./chfcreate`, based on the delimiters `"EXPORT"` and `";"` in the header file `include/sample.h`. The first delimiter `"EXPORT"` is the first common symbol for function prototypes in the header file. The new header file `sample.h` only contains the prototype of the functions that will be used in the application program without macros and typedef statements. Details about the function `processhfile()` can be found in section D.2. Before the header file `sample.h` is processed by `c2chf`, special functions such as `func3()` with an argument of pointer to functions are removed by function removeFuncProto().

```
removeFuncProto("chfcreate/sample.h", "func3", 0);
```

Details about function `removeFuncProto()` can be found in section D.3. The new `sample.h` in the directory `chfcreate` is shown in Program 3.20.

Command

```
c2chf chfcreate/sample.h -h _Chsample_handle -r chfcreate/sample_err
      -o c c -o chf $pkgname/lib
```

generates a C wrapper file `sample_chdl.c` in the directory `c` and function files with file extension .chf in the direectory `chpack/lib`. The number of chf files in the directory will match the number of function prototypes written in the header file. The option `-h` allows the user to specify handle for chf files. In this case the handle `_Chsample_handle` is used.

```
func1 NaN
func2 NaN
```

Program 3.21: Return values for function files (sample_err)

The option `-r` allows the user to provide a file which contains a list of function names and their error return values used inside their chf function files to over write the default value NULL for a function returning pointer type and -1 for other types. In this case the error return values are specified in the file `sample_err` in the directory `chfcreate`. Program 3.21 shows the file `sample_err`.

The next block of code is

```
cp -f chfhandmade/*.chf $pkgname/lib
chmod 644 $pkgname/lib/*.chf
```

Function files for special cases were prepared before the script `pkgcreate.ch` is executed. These function files are copied to the function file directory of the package. Program3.22, show function file for `func3()` which is created manually for function `func3()` with an argument of pointer to function.

```
int func3(FUNPTR fp, int n) {
    void *fptr;
    int retval;

    fptr = dlsym(_Chsample_handle, "func3_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return -1;
    }
    dlrunfun(fptr, &retval, func3, fp, n);
    return retval;
}
```

Program 3.22: chf file (func3.chf).

The command `chmod` changes the permissions for a file. The number `644` changes all the permissions of the files in the directory `sample` to give the owner read and write permissions as well as read permission to the group and everyone else.

The directory `c` contains a chdl C program `sample_chdl.c` generated automatically by command `c2chf`. Program 3.24 shows the `sample_chdl.c` for functions `func1()` and `func2()`. Program `func3_chdl.c` for special function `func3()` in the directory `c/handmade/sample` is manually created. Program `func3_chdl.c` contains is the chdl file for `func3()`.

```
#include <sample.h>
#include <ch.h>

static ChInterp_t interp;
static int fp_funarg(int n);
static void *fp_funptr;

EXPORTCH int func3_chdl(void *varg) {
    ChVaList_t ap;
    FUNPTR fp_ch, fp_c;
    int n;
    int retval;

    Ch_VaStart(interp, ap, varg);
    fp_ch = Ch_VaArg(interp, ap, FUNPTR);
    n = Ch_VaArg(interp, ap, int);
    fp_funptr = (void *)fp_ch;
    if (fp_ch != NULL)  {
        fp_c = (FUNPTR)fp_funarg;
    }
    retval = func3(fp_c, n);
    Ch_VaEnd(interp, ap);
    return retval;
}

static int fp_funarg(int n) {
     int retval;
     Ch_CallFuncByAddr(interp, fp_funptr, &retval, n);
     return retval;
}
```

Program 3.23: chdl file (func3_chdl.c).

```
#include <sample.h>
#include <ch.h>

EXPORTCH double func1_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double retval;

    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    retval = func1(x);
    Ch_VaEnd(interp, ap);
    return retval;
}

EXPORTCH double func2_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double y;
    double retval;

    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    y = Ch_VaArg(interp, ap, double);
    retval = func2(x, y);
    Ch_VaEnd(interp, ap);
    return retval;
}
```

Program 3.24: Cteated C file (sample_chdl.c).

The code below compiles chdl files to create a dynamically loaded library `libsample.dl` in the directory `c`.

```
cd c
$makecmd clean $debug
$makecmd $debugFile
```

The above `makecmd` commands are equivalent to

```
#ifdef _WIN32_
  nmake -f Makefile.win clean >nul 2>nul
  nmake -f Makefile.win >logfile 2>&1
#else
  make -f Makefile clean
  make -f Makefile
#endif
```

In Windows, the makefile `Makefile.win` will be used. The program `nmake` first cleans up the unwanted files specified by `Makefile.win` and then runs `Makefile.win`. The command suffix `>nul 2> nul` from the variable `debug` redirects the standard output and standard error streams of the `nmake` command.

The command suffix `>logfile 2>& 1` from the variable `debugFile` redirects the standard output and standard error streams of the `nmake` command to file `logfile`. Program 3.25 displays Makefile.win used for Windows. In other operating systems, program `make` and Makefile will be used. Program 3.26 displays Makefile for Unix.

```
# build dynamically loaded  lib libsample.dl

target: libsample.dl

libsample.dl: sample_chdl.obj func3_chdl.obj func.o
        ch dllink libsample.dl sample_chdl.obj \
                  func3_chdl.obj func.obj
sample_chdl.obj:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I../src
func3_chdl.obj:  handmade/sample/func3_chdl.c
        ch dlcomp libsample.dl handmade/sample/func3_chdl.c -I../src
func.obj:  ../src/func.c
        ch dlcomp libsample.dl ../src/func.c -I../src
clean:
        del *.obj
        del *.exp
        del *.lib
        del *.dl
```

Program 3.25: Windows Makefile (Makefile.win).

```
# build dynamically loaded  lib libsample.dl

target: libsample.dl

# -lm for func1() and func2() using functions in lib libm.so
libsample.dl: sample_chdl.o func3_chdl.o func.o
        ch dllink libsample.dl sample_chdl.o \
                  func3_chdl.o func.o -lm
sample_chdl.o:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I../src
func3_chdl.o:  handmade/sample/func3_chdl.c
        ch dlcomp libsample.dl handmade/sample/func3_chdl.c -I../src
func.o:  ../src/func.c
        ch dlcomp libsample.dl ../src/func.c -I../src
clean:
        rm -f *.o *.dl *.a *.dll *.lib *.exp *.obj
```

Program 3.26: Unix Makefile (Makefile).

The generated dynamically loaded interface library `libsample.dl` is tested using function `dlopen()`. The code segment below

```
cd c
mv -f c/libsample.dl $cwd/$pkgname/dl
$makecmd clean $debug
```

moves the file `libsample.dl` into the dl directory of the package. The script then does a final cleaning of unwanted files in the directory `c`.

3.5.2 Installing a Package Using `pkginstall.ch`

After the package `chsample` is created by command `pkgcreate.ch`, it can be installed by the command `pkginstall.ch`. By default, pkginstall installs a package in the current directory into the directory `CHHOME/package` and copies required header files in `package/include` into the directory `CHHOME/toolkit/include`.

If you want to install Ch package into your preferred directory, you can specify it in the command line. During the installation, it will modify `_ipath` and `_ppath` in `.chrc` in Unix or `_chrc` in Windows in the user home directory. During installation, an installation file is created under the directory `CHHOME/package/installed` with a list of the installed directories and files.

This program can also uninstall a Ch Package by removing header files installed into the directory `CHHOME/toolkit/include` and the package in the `CHHOME/package` directory based on the corresponding package file in `CHHOME/package/installed` directory.

The package `chsample` can be installed by the command

```
> pkginstall.ch chsample
```

It can be uninstalled by the command

```
> pkginstall.ch -u chsample
```

```
PACKAGE=chsample

create:
        ch ./pkgcreate.ch
install:
        ch pkginstall.ch $(PACKAGE)
uninstall:
        ch pkginstall.ch -u $(PACKAGE)
rmpkg:
        rm -rf $(PACKAGE)
clean:
        rm -f  c/*.c
        rm -f  chfcreate/*.h
        cd c;  make clean
```

Program 3.27: Makefile for creating, installing, and uninstalling a package.

A Makefile in Program 3.27 allows creation, installation, and uninstallation of a package to be handled by commands

```
make
make install
make uninstall
```

respectively.

Chapter 4

Accessing Global Variables in C Space

In the previous chapter, handling functions with prototypes declared in a header file is described. Besides prototypes of functions, global variables may also be declared in header files. In this chapter, we will describe how to access global variables in C Space from Ch space.

In the example below, the global variable `const int i1` in C space will be accessed in Ch space.

Example

Listing 1 — header file in C space (expvar_c.h)

```
#ifndef _EXPVAR_H_
#define _EXPVAR_H_

/* global variables to be exported to the Ch space */
const int i1 = 100;

#endif
```

Listing 2 — C function (expvar.c)

```
#include <ch.h>
#include <stdio.h>
#include "expvar_c.h"

EXPORTCH void expvar_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int *ii1;

  printf("The variables in the C space are %d\n", i1);

  Ch_VaStart(interp, ap, varg);
  ii1 = Ch_VaArg(interp, ap, int*);
  *ii1 = i1; /* pass the value to the Ch space */

  Ch_VaEnd(interp, ap);
  return;
}
```

Listing 3 — header file in Ch space (expvar.h)

```
#ifndef _EXPVAR_H_
#define _EXPVAR_H_
```

```
int i1;

/* load DLL */
#include <dlfcn.h>
void *_ChExpvar_handle = dlopen("libexpvar.dl", RTLD_LAZY);
if(_ChExpvar_handle == NULL) {
     fprintf(_stderr, "Error: %s(): dlopen(): %s\n", __func__, dlerror());
     fprintf(_stderr, "       cannot get _ChExpvar_handle in expvar.h\n");
     exit(-1);
  }

/* release DLL when program exits */
void _dlclose_expvar(void) {
    dlclose(_ChExpvar_handle);
}
atexit(_dlclose_expvar);

/* Get addresses of global variables in C space */
void *fptr_expvar_h = dlsym(_ChExpvar_handle , "expvar_chdl");
if(fptr_expvar_h == NULL) {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
}
dlrunfun(fptr_expvar_h, NULL, NULL, &i1);

#endif
```

Listing 4 — Ch application (expvar.ch)

```
#include <expvar.h>

int main() {
  printf("Value of i1 in Ch is %d\n", i1);
  return 0;
}

/*** Result from excuting this program
The variables in the C space are 100
Value of i1 in Ch is 100
***/
```

Listing 5 — Makefile (Makefile)

```
target: Makefile libexpvar.dl

libexpvar.dl: Makefile expvar.o
        ch dllink libexpvar.dl expvar.o

expvar.o:  expvar.c
        ch dlcomp libexpvar.dl expvar.c

clean:
        rm -f *.o *.dl
```

In the next example, the value for global variable `int i2` in C space will be changed in Ch space.

Program 4.1 is a C header file containing three global variables `i1`, `i2`, and `a`. We assume the variable `i1` remains unchanged after the original value is assigned, whereas variable `i2` will be changed during the execution of the C program. Variable `a` is an array of int type. Three methods of handling global variables

```
#ifndef _EXPVAR_H_
#define _EXPVAR_H_

/* global variables to be exported to the Ch space */
const int i1 = 100;
int i2 = 200; /* i2 will be changed during execution */
int a[10] = {1,2,3,4,5,6,7,8,9,10};

/* function prototypes */
/* ... */

#endif
```

Program 4.1: C header file containing global variables (`expvar_c.h`).

will be introduced by this example. In Ch header file shown in Program 4.2, code for loading and releasing DLL module has been described in Chapter 3. The global variable `i1` in Ch is declared with the same data type as the corresponding global variable `i1` in C. It is used to keep the original value of the C variable. Since the value of `i2` in C space will be changed during the execution of the program, not only its value, but also its address is needed in Ch space. A temporary variable `i2_` is declared as pointer to int to keep the address of C space variable `i2` in Program 4.2, so that we can get the C space value of `i2` from Ch space using macro `i2` defined as follows

```
#define i2 *i2_
```

To use the memory for array `a` in the C space, variable `a` in the Ch space is declared as a pointer to int. This pointer in the Ch space will point to the memory for the array `a` in the C space. The chdl function `expvar_chdl`() defined in Program 4.3 is called by statements

```
void *fptr_expvar_h = dlsym(_ChExpvar_handle, "expvar_chdl");
dlrunfun(fptr_expvar_h, NULL, NULL, &i1, &i2_, &a);
```

where addresses of variables `i1`, `i2_`, and `a` in Ch space are passed to C space as arguments of **dlrunfun**().

In the chdl function `expvar_chdl`() shown in Program 4.3, statements

```
ii1 = Ch_VaArg(interp, ap, int*);
*ii1 = i1; /* pass the value to the Ch space */
```

assign the value of `i1` to the address of variable `i1` in Ch space. Therefore, the variable ı1 in Ch has the same value as the one in the C space. But, if the variable `i1` in C space is changed afterward, the one in Ch will keep the original value. For variable `i2`, statements

```
ii2 = Ch_VaArg(interp, ap, int**);
*ii2 = &i2; /* pass the address to the Ch space */
```

assign the address of `i2` to the address of the temporary variable `i2_` in Ch, so that the variable in C can be accessed from Ch space by its address. If the variable `i2` in C is changed, the one in Ch will change correspondingly. For array `a`, statements

```
aptr = Ch_VaArg(interp, ap, int**);
*aptr = a; /* pass the address to the Ch space */
```

```
#ifndef _EXPVAR_H_
#define _EXPVAR_H_

int i1;
int *i2_;
#define i2 *i2_
int *a;

/* load DLL */
#include <dlfcn.h>
void *_ChExpvar_handle = dlopen("libexpvar.dl", RTLD_LAZY);
if(_ChExpvar_handle == NULL) {
     fprintf(_stderr, "Error: %s(): dlopen(): %s\n", __func__, dlerror());
     fprintf(_stderr, "       cannot get _ChExpvar_handle in expvar.h\n");
     exit(-1);
  }

/* release DLL when program exits */
void _dlclose_expvar(void) {
    dlclose(_ChExpvar_handle);
}
atexit(_dlclose_expvar);

/* Get addresses of global variables in C space */
void *fptr_expvar_h = dlsym(_ChExpvar_handle , "expvar_chdl");
if(fptr_expvar_h == NULL) {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
}
dlrunfun(fptr_expvar_h, NULL, NULL, &i1, &i2_, &a);

#endif
```

Program 4.2: Ch header file for handling global variables (`expvar.h`).

assign the address for array `a` in the C space to the pointer `a` in the Ch sapce.

The function `changeCVar_chdl`() changes the values of `i1`, `i2` and element `a[3]` in the C space. The Ch application shown in Program 4.4 prints out the corresponding values of `i1` and `i2` in Ch space before and after `changeCVar`() is called as well as values for the array `a` . The `i2` is changed from `200` to `400`, whereas `i1` keeps the original value `100`. The value for element `a[3]` has been changed. The result from executing Program 4.4 is appended at the end of the file.

```
#include <ch.h>
#include <stdio.h>
#include "expvar_c.h"

EXPORTCH void expvar_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int *ii1;
  int **ii2;
  int **aptr;

  printf("The variables in the C space are %d and %d\n", i1, i2);

  Ch_VaStart(interp, ap, varg);
  ii1 = Ch_VaArg(interp, ap, int*);
  *ii1 = i1; /* pass the value to the Ch space */
  ii2 = Ch_VaArg(interp, ap, int**);
  *ii2 = &i2; /* pass the address to the Ch space */
  aptr = Ch_VaArg(interp, ap, int**);
  *aptr = a;    /* pass the address to the Ch space */

  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH void changeCVar_chdl() {
  i2 = 400; /* change value of C variable */
  printf("The variables in the C space are %d and %d\n", i1, i2);
  a[3] = 300; /* change value of an element of C array*/

  return;
}
```

Program 4.3: chdl functions for handling global variables (`expvar.c`).

```
#include "expvar.h"

void changeCVar() {
  void *fptr;

  fptr = dlsym(_ChExpvar_handle , "changeCVar_chdl");
  if(fptr == NULL) {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  /* Pass the address of retval to C space */
  dlrunfun(fptr, NULL, NULL);
  return;
}

int main() {
  int i;
  printf("Original values of i1 and i2 in Ch are %d and %d\n", i1, i2);
  changeCVar(); /* change variables in C space */
  printf("After calling changeCVar(), i1 and i2 are %d and %d\n", i1, i2);
  for(i=0; i<10; i++) {
    printf("a[%d] = %d\n", i, a[i]);
  }
  return 0;
}

/*** Result from excuting this program
The variables in the C space are 100 and 200
Original values of i1 and i2 in Ch are 100 and 200
The variables in the C space are 100 and 400
After calling changeCVar(), i1 and i2 are 100 and 400
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 300
a[4] = 5
a[5] = 6
a[6] = 7
a[7] = 8
a[8] = 9
a[9] = 10
***/
```

Program 4.4: expvar.ch: Ch application accessing global variables in C space (Note: chf file is included)

Chapter 5

Templates for Calling Regular C Functions

Previous chapters described how to write chf and chdl files and build dynamically loaded libraries. After reading the first three chapters, the users should have gained a basic understanding of what chdl, chf, and dynamically loaded libraries are, and how the C binary modules interface with the Ch space through APIs. We still need to explain topics that deal with specific types of function. This chapter will describe how a simple function with arguments or return value of simple data types in C space can be called in Ch space. The Makefile in the first example is used for all examples in this chapter.

5.1 Functions without Return Value or Argument

The section is for handling functions without a return value and argument. Assume the function to be handled in C space is:

```
void functionName() {
  ...
}
```

where `functionName` can be any valid function name in C. It has no argument and return value.

In the chf file, the Ch function is given in Program 5.1.

```
#include<dlfcn.h>
  ...

void functionName()
{
  void *dlhandle, *fptr;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                     __func__, dlerror());
    return;
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address */
  dlrunfun(fptr, NULL, functionName);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
            __func__, dlerror());
    return;
  }
}
```

Program 5.1: Regular functions without return value and argument (chf function).

Here, we assume that `libproject.dl` is the DLL (Dynamically Loaded Library) file that the user builds from the library files and the chdl files. Functions introduced in Chapter 2 such as **dlopen()**, **dlsym()**, **dlrunfun()** and **dlclose()**, defined in **dlfcn.h**, are used to access the DLL . Appendix B contains detailed reference of these functions. The **dlrunfun()** in the Ch function calls the chdl function (Program5.2) by its address in the C address space. The third argument of **dlrunfun()** can be NULL or `functionName`. If `functionName` is used, Ch will perform the prototype checking. Ch skips the prototype check otherwise.

Program 5.2 is the chdl file that has the chdl function `functionName_chdl`().

```
#include<ch.h>
  ...

EXPORTCH void functionName_chdl() {
  functionName();
}
```

Program 5.2: Regular functions without return value and argument (chdl function).

The macro **EXPORTCH**, defined in **ch.h**, indicates that this function is exported from a dynamically loaded library and suffix `_chdl` in the function name `functionName_chdl` indicates that it is a chdl function. Since the original C function has no argument and return value, `functionName_chdl`'s argument list is empty. The function `functionName`(), which is in binary module, is invoked within `functionName_chdl`().

Example

A complete example including the original C function file, C header file, chf , chdl , Makefile, and application program is given in this section. The header `header1.h` in both C space and Ch space is the same here for easy of illustration. If you have multiple functions to interface, you might need to follow the example in Chapter 3 to construct different header file.

Listing 1 — header file (hello1.h)

```
void hello1();
```

Listing 2 — C function (hello1.c)

```
#include <stdio.h>
#include "hello1.h"

void hello1(){
  printf("hello world !\n");
}
```

Listing 3 — chf file (hello1.chf)

```
#include<dlfcn.h>
void hello1() {
  void *handle, *fptr;

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return;
  }

  fptr = dlsym(handle, "hello1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, hello1);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return;
  }

  return;
}
```

Listing 4 — Makefile (Makefile)

```
# This Makefile is for all examples

All : libhello.dl Makefile
```

```
libhello.dl : hello1.o hello1_chdl.o \
               hello2.o hello2_chdl.o \
               hello3.o hello3_chdl.o \
               testarray.o testarray_chdl.o \
               sum1.o sum1_chdl.o \
               sum2.o sum2_chdl.o \
               retstruct.o retstruct_chdl.o \
               ret_comp_arr_chdl.o comp_array.o
ch dllink libhello.dl hello1.o hello1_chdl.o \
                              hello2.o hello2_chdl.o \
                              hello3.o hello3_chdl.o \
                              testarray.o testarray_chdl.o \
                              sum1.o sum1_chdl.o \
                              sum2.o sum2_chdl.o \
                              retstruct.o retstruct_chdl.o \
                              ret_comp_arr_chdl.o comp_array.o

hello1.o : hello1.c
ch dlcomp libhello.dl hello1.c

hello1_chdl.o : hello1_chdl.c
ch dlcomp libhello.dl hello1_chdl.c

hello2.o : hello2.c
ch dlcomp libhello.dl hello2.c

hello2_chdl.o : hello2_chdl.c
ch dlcomp libhello.dl hello2_chdl.c

hello3.o : hello3.c
ch dlcomp libhello.dl hello3.c

hello3_chdl.o : hello3_chdl.c
ch dlcomp libhello.dl hello3_chdl.c

testarray.o : testarray.c
ch dlcomp libhello.dl testarray.c

testarray_chdl.o : testarray_chdl.c
ch dlcomp libhello.dl testarray_chdl.c

sum1.o : sum1.c
ch dlcomp libhello.dl sum1.c

sum1_chdl.o : sum1_chdl.c
ch dlcomp libhello.dl sum1_chdl.c

sum2.o : sum2.c
ch dlcomp libhello.dl sum2.c

sum2_chdl.o : sum2_chdl.c
ch dlcomp libhello.dl sum2_chdl.c

retstruct.o : retstruct.c
ch dlcomp libhello.dl retstruct.c

retstruct_chdl.o : retstruct_chdl.c
ch dlcomp libhello.dl retstruct_chdl.c
```

```
comp_array.o : comp_array.c
ch dlcomp libhello.dl comp_array.c

ret_comp_arr_chdl.o : ret_comp_arr_chdl.c
ch dlcomp libhello.dl ret_comp_arr_chdl.c

clean :
rm -f *.dl *.o
```

Listing 5 — chdl file (hello1_chdl.c)

```
#include <ch.h>
#include "hello1.h"

EXPORTCH void hello1_chdl() {
  hello1();
  return;
}
```

Listing 6 — Ch application (hello1.ch)

```
#include "hello1.h"

int main() {
  hello1();
  return 0;
}
```

Output

```
hello world !
```

5.2 Functions with Arguments of Simple Type

After learning how to deal with the most simple function, let's deal with functions that are a little more complicated. This section describes functions that have arguments of simple data types but no return value.

Let's assume that the function has the following form:

```
void functionName(data_type1 arg1, data_type2 arg2) {
  ...
}
```

`data_type1` and `data_type2` could be any simple data type and `arg1` and `arg2` could be any valid variable names in C / Ch. The simple data type `data_type1` can be any one of the following type: `char, short, int, float, double, complex, double complex, char*, shor t*, int*, float*, double*, complex double*, char**, int**, union, struct, pointer to struct, etc`. The complex data types such as class, the pointer to function, VLA etc do not belong to the simple data type. When a struct or union type is involved, the same structure or union needs to be defined in both Ch and C spaces.

Program 5.3 contains the Ch function `functionname`() that communicates with the C space.

```
#include<dlfcn.h>
  ...

void functionName(data_type1 arg1, data_type2 arg2)
{
  void *dlhandle, *fptr;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
            __func__, dlerror());
    return;
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library
     by address. arguments are passed to chdl function here */
  dlrunfun(fptr, NULL, functionName, arg1, arg2);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
                     __func__, dlerror());
    return;
  }
}
```

Program 5.3: Regular functions with arguments of simple type (chf function).

This function is essentially the same as the one in the first section, except that this one has two arguments. This difference is reflected in the argument list of the Ch function and **dlrunfun()**. The value `arg1` and `arg2`, the fourth and fifth arguments of the **dlrunfun()**, respectively, are passed to the chdl function following the chdl handle, `NULL`, and `functionName`. Because `functionName`() has no return value, the second argument of `dlrunfun`() is **NULL**.

In the chdl file, the chdl function `functionName_chdl`() is shown in Program 5.4,

```
#include<ch.h>
  ...

EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  data_type1 arg1;
  data_type2 arg2;

  /* get arguments passed from the Ch address space*/
  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  arg2 = Ch_VaArg(interp, ap, data_type2);   /* get 2nd argument  */
  functionName(arg1, arg2);
  Ch_VaEnd(interp, ap);
}
```

Program 5.4: Regular functions with arguments of simple type (chdl function).

where macros **va_list**, **Ch_VaStart()**, **Ch_VaArg()** and **Ch_VaEnd()** are defined in header file **ch.h** to handle the argument list of exported functions for a dynamically loaded library presented in AppendixA. The pointer `varg` is used to keep the variable argument list in this function. After all the arguments are received, they are passed to the C function `functionName`().

Example

In this example, function `hello2`() is used to illustrate the interface of a function with one argument.

Listing 1 — header file (hello2.h)

```
void hello2(char *name);
```

Listing 2 — C function (hello2.c)

```
#include <stdio.h>
#include "hello2.h"

void hello2(char *name){
  printf("hello world from %s !\n", name);
}
```

Listing 3 — chf file (hello2.chf)

```
#include<dlfcn.h>

void hello2(char *name) {
  void *handle, *fptr;

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return;
  }

  fptr = dlsym(handle, "hello2_chdl");
```

```
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, hello2, name);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return;
  }

  return;
}
```

Listing 4 — chdl file (hello2_chdl.c)

```
#include <ch.h>
#include "hello2.h"

EXPORTCH void hello2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  char *name;

  Ch_VaStart(interp, ap, varg);
  name = Ch_VaArg(interp, ap, char *);
  hello2(name);
  Ch_VaEnd(interp, ap);
  return;
}
```

Listing 5 — Ch application (hello2.ch)

```
#include "hello2.h"

int main() {
  hello2("Ch");
  return 0;
}
```

Output

```
hello world from Ch !
```

5.3 Functions with Return Value of Simple Type

This case illustrates how to interface C functions with return values of simple type with Ch space. Assume that the C function to be handled is:

```
return_type functionName(data_type1 arg1, data_type2 arg2) {
  return_type val_of_return_type;

  ...

  return val_of_return_type;
}
```

It returns a value of simple data type `return_type`. The simple data type is illustrated in section 5.2. The variable `val_of_return_type` simply indicates that it is a variable for storing the return value. In the chf file, the Ch function is shown in Program 5.5.

```
#include<dlfcn.h>
  ...

return_type functionName(data_type1 arg1, data_type2 arg2)
{
  void *dlhandle, *fptr;
  return_type retval;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                      __func__, dlerror());
     return FAIL_VALUE;
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }

  /* call the chdl function in dynamically loaded library */
  /* arguments arg1 and arg2 are passed to chdl function here */
  /* return value will passed from chdl function */
  dlrunfun(fptr, &retval, functionName, arg1, arg2);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
            __func__, dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 5.5: Regular functions with return value of simple type (chf function).

where `retval` is of data type `return_type`. To get the return value from C address space, its address `&retval` is passed to `functionName_chdl`() in Program 5.6 by **dlrunfun()**, as its second argument. If the DLL can not be loaded, or the chdl function can not be found in DLL , the value `FAIL_VALUE` will be returned. `FAIL_VALUE` is a value of type `return_type3`, such as NULL for point type and -1 for integral type, to indicate the failure of the function calling.

In the chdl file, the chdl function is shown in Program 5.6.

```
#include<ch.h>
  ...

EXPORTCH return_type functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  data_type1 arg1;
  data_type2 arg2;
  return_type retval;

  /* get arguments passed from the Ch function */
  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  arg2 = Ch_VaArg(interp, ap, data_type2);   /* get 2nd argument  */

  /* call the function in Source Library files */
  retval = functionName(arg1, arg2);

  Ch_VaEnd(interp, ap);
  return retval; /* pass the return value to Ch function */
}
```

Program 5.6: Regular functions with return value of simple type (chdl function).

where `retval`'s data type is `return_type`. It is used to keep the return value of the C function `functionName`(), and then return to the Ch function in Program 5.5.

Example

In this example, function `hello3`() is used to illustrate the case of a function with return value and arguments.

Listing 1 — header file (hello3.h)

```
int hello3(char *name);
```

Listing 2 — C function (hello3.c)

```
#include <stdio.h>
#include <string.h>
#include "hello3.h"

int hello3(char *name){
  printf("hello world from %s !\n", name);
  return strlen(name);
}
```

Listing 3 — chf file (hello3.chf)

```
#include<dlfcn.h>

int hello3(char *name) {
  void *handle, *fptr;
  int retval;
```

```
  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }

  fptr = dlsym(handle, "hello3_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, hello3, name);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }

  return retval;
}
```

Listing 4 — chdl file (hello3_chdl.c)

```
#include <ch.h>
#include "hello3.h"

EXPORTCH int hello3_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  char *name;
  int retval;

  Ch_VaStart(interp, ap, varg);
  name = Ch_VaArg(interp, ap, char *);
  retval = hello3(name);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 5 — Ch application (hello3.ch)

```
#include "hello3.h"

int main() {
  int strl;

  strl = hello3("Ch");
  printf("strlen = %d\n", strl);
  return 0;
}
```

Output

```
hello world from Ch !
strlen = 2
```

5.4 Functions with Arguments of Array

This section has the sample chf and chdl files for functions with arguments of array of simple types.

Assume that we have the following function:

```
return_type functionName(data_type1 arg1[num1],
                         data_type2 arg2[num2][num3],
                         data_type3 arg3[]) {
  return_type val_of_return_type;

  ...

  return val_of_return_type;
}
```

`num1`, `num2` and `num3` are integers indicating the size of each dimension of arrays. This example is slightly different from the previous one, since arguments of array type, instead of simple data type, are passed to the function.

In the chf file, the Ch function is shown in Program 5.7.

```
#include<dlfcn.h>
  ...

return_type functionName(data_type1 arg1[num1],
                         data_type2 arg2[num2][num3])
                         data_type3 arg3[])
{
  void *dlhandle, *fptr;
  return_type retval;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                      __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }

  /* call the chdl function in dynamically loaded library */
  /* arguments are passed to chdl function */
  dlrunfun(fptr, &retval, functionName, arg1, arg2, arg3);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
                     __func__, dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 5.7: Regular functions with argument of array (chf function).

where names (actually addresses) of arrays are passed to the chdl function as the fourth, fifth and sixth arguments of the function **dlrunfun()**, just like what was done for the previous example.

In the chdl file, the chdl function is shown in Program 5.8.

```
#include<ch.h>
    ...

EXPORTCH return_type functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  /* use pointers to access arrays */
  data_type1 *arg1;
  data_type2 *arg2;
  data_type3 *arg3;
  return_type retval;

  /* get arguments passed from the Ch function */
  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1*);   /* get 1st argument  */
  arg2 = Ch_VaArg(interp, ap, data_type2*);   /* get 2nd argument  */
  arg3 = Ch_VaArg(interp, ap, data_type3*);   /* get 3rd argument  */

  retval = functionName(arg1, arg2, arg3);

  Ch_VaEnd(interp, ap);
  return retval; /* pass the return value to Ch function */
}
```

Program 5.8: Regular functions with argument of array (chdl function).

Three pointers, `arg1`, `arg2` and `arg3` are created to store the three arrays that are passed in from the Ch space. Note that we used pointers instead of arrays. Arguments `arg1` and `arg2` are then passed to the C `functionName`().

Example

In this example, function `testarray`() is used to illustrate the case of the function with arguments of array.

Listing 1 — header file (testarray.h)

```
int testarray(int arri[5]);
```

Listing 2 — C function (testarray.c)

```
#include <stdio.h>
#include "testarray.h"

int testarray(int arri[5]){
  int i;

  for(i=0; i<5; i++)
    printf("i[%d] = %d\n", i, arri[i]);

  return 0;
}
```

Listing 3 — chf file (testarray.chf)

```
#include<dlfcn.h>

int testarray(int arri[5]) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }

  fptr = dlsym(handle, "testarray_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, testarray, arri);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }

  return retval;
}
```

Listing 4 — chdl file (testarray_chdl.c)

```
#include <ch.h>
#include "testarray.h"

EXPORTCH int testarray_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int *arri;
  int retval;

  Ch_VaStart(interp, ap, varg);
  arri = Ch_VaArg(interp, ap, int *);
  retval = testarray(arri);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 5 — Ch application (testarray.ch)

```
#include "testarray.h"

int main() {
  int arri[5];
  int i;

  for(i=0; i<5; i++)
    arri[i] = i;

  testarray(arri);

  return 0;
}
```

Output

```
i[0] = 0
i[1] = 1
i[2] = 2
i[3] = 3
i[4] = 4
```

5.5 Functions with an Argument List of Variable Length

This section illustrates how to interface a function with a variable number of arguments in the C space such as one shown below.

```
void functionName(data_type1 param, ... ) {
  ...
}
```

Normally, functions of this kind take at least one argument, say `param` with type of `data_type1`. The ellipsis `...` in the argument list indicates that the function can take any number of arguments following the first one. Three methods described below can be used to handle this function depending on different ways it works in.
Section 5.5.1 is a general method for handling a function with variable number of arguments. Sections5.5.2 and 5.5.3 can be used for hanlding special cases.

5.5.1 Using Functions with Argument of va_list Type

If you have a C function with variable number of arguments in a library, such as the one shown below

```
void functionName(data_type1 param, ...);
```

the function has to be implemented using a corresponding function, typically named `vfunctionName`(), taking an argument of `va_list`, instead of argument list `"..."`, i.e.

```
void vfunctionName(data_type1 param, va_list ap);
```

Then, the function `functionName()` can be implemented as follows:

```
#include <stdarg.h>
void functionName(data_type1 param, ...) {
     va_list ap;
     va_start(ap, param);
     vfunctionName(param, ap);
     va_end(ap);
}
```

The function `vfunctionName`() will typically have the following format.

```
void vfunctionName(data_type1 param, va_list ap) {
     /* can be anything here */
     datatype2 d = va_arg(ap, datatype2);
     ...
}
```

When a Ch program calls function void functionName(datatype param, ...) in the Ch space, Ch cannot pass parametter `"..."` from the Ch space to the C space directly. We have to wrap them as an argument of type `va_list ap`, which can be any type and any number of arguements in the Ch space. The argument `va_list ap` can be easily passed from the Ch space to the C space. If your C library function does not have a corresponding function, similar to `vfunctionName()`, you need to modify the C source code to make it suitable for interfacing with Ch.

If the function has a return value, it can be handled similarly. For example, a C function that returns a value of double type can be implemented as follows:

```
#include <stdarg.h>
double functionName2(data_type1 param, ...) {
     va_list ap;
     double retval;

     va_start(ap, param);
     retval = vfunctionName2(param, ap);
     va_end(ap);
     return retval;
}
```

The source code for C function `fuunctionName()` can be readily used as a function file in the Ch space. To interface a C function with an argument of type `va_list`, such as `vfunctionName()`, Ch provides two APIs, **Ch_VaVarArgsCreate()** and **Ch_VaVarArgsDelete()**, to handle variable argument lists.

Example

Example shown below illustrates how to handle variable argument list with APIs Ch_VaVarArgsCreate() and Ch_VaVarArgsDelete(). Variables of different data types, including char, int, double, struct, pointer, pointer to pointer and array in main() function of Program 5.9, are passed to function `func`(). Function `func`() can take variable arguments after its first argument. Within this function, a variable argument list `ap` with type va_list is generated by statement

```
va_start(ap, ii);
```

where `ii` is the first argument. Here `ap` is a pointer to the Ch variable argument list which includes all arguments following `ii`. Then the Ch function `vfunc`(), which takes two arguments, `ii` and `ap`, will be invoked. Typically, for a function like `func`() in Program 5.10, which takes an variable argument list, there is a corresponding function like `vfunc`(), which takes an argument with type va_list, to handle the variable argument list of the former. Through the function file in Program 5.11, the chdl function `vfunc_chdl`() and C function `vfunc`() in Program 5.12 will be called sequentially. The C function `vfunc`() takes a C variable argument list as the second argument. As mentioned above, a Ch variable argument list is different from the corresponding C variable argument in handling arguments inside. Therefore, Ch variable argument list `ap` in function `vfunc_chdl`() can't be passed directly to `vfunc`(). Statement

```
ap_c = Ch_VaVarArgsCreate(interp, ap_ch, &memhandle);
```

creates a C variable argument list named `ap_c` from the Ch variable argument list named `ap_ch` for a Ch interpreter *interp*. The variable `memhandle` is used only for management of memory allocated in function **Ch_VaVarArgsCreate**(). All memory allocated for `ap_c` will be released later by function call

```
Ch_VaVarArgsDelete(interp, memhandle);
```

The output from executing Program 5.9 is appended at the end of the file.

```
#!/bin/ch
#include <stdio.h>
#include <stdarg.h>

typedef struct {int i1; double d1; char c1; float f1;} SS1;
void func(int, ...);
void vfunc(int, va_list ap);

int main() {
    char    c1 = 'c';
    int     i1 = 10;
    short   s1 = 20;
    float   f1 = 30;
    double  d1 = 65536;
    double  *pd1 = &d1;       /* pointer to double */
    double  **ppd1 = &pd1;    /* pointer to double */
    float   af1[3][4];         /* array */
    SS1     ss1;              /* struct */
    double  d2 = 50;

    ss1.i1 = 10;
    ss1.d1 = 65536;
    ss1.c1 = 'c';
    ss1.f1 = 65536;

    af1[1][1] = 456;

    func(0, c1, i1, s1, f1, d1, pd1, ppd1, af1, ss1, d2);
    return 0;
}

/** output from executing this program **
The first argument is 0

c1 = c
i1 = 10
s1 = 20
f1 = 30.000000
d1 = 65536.000000

*pd1 = 65536.000000
**ppd1 = 65536.000000
af1[1][1] = 456.000000

ss1.i1 = 10
ss1.d1 = 65536.000000
ss1.c1 = c
ss1.f1 = 65536.000000

d2 = 50.00000

*******/
```

Program 5.9: Example for handling variable arguments – Ch program `vararg.ch`.

```
#include <stdarg.h>

void func(int ii, ...) {
    va_list ap;

    va_start(ap, ii); /* create an Ch va_list */
    vfunc(ii, ap);
    va_end(ap);
    return;
}
```

Program 5.10: Example for handling variable arguments – function file `func.chf`.

```
/* File name vararg.chf
   function file for sample function vararg()
*/
#include<dlfcn.h>
#include<stdarg.h>
#include<stdio.h>

void vfunc(int ii, va_list ap) {
    void *fptr, *handle;

    handle = dlopen("libvararg.dl", RTLD_LAZY);
    if(handle == NULL) {
      printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
      return;
    }

    fptr = dlsym(handle, "vfunc_chdl");
    if(fptr == NULL)  {
       printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
       return;
    }

    dlrunfun(fptr, NULL, NULL, ii, ap);

    if(dlclose(handle)!=0) {
       fprintf(stderr, "Error: %s(): dlclose(): %s\n", __func__, dlerror());
       return;
    }
    return;
}
```

Program 5.11: Example for handling variable arguments – function file `vfunc.chf`.

```
#include <ch.h>
#include <math.h>
#include <stdarg.h>
#include <stdio.h>

typedef struct {int i1; double d1; char c1; float f1;} SS1;
void func(int n, ...);
void vfunc(int n, ChVaList_t ap_c);

EXPORTCH void vfunc_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    void *ap_c; /* equivelent to va_list ap */
    ChVaList_t ap_ch;
    int ii;
    void *memhandle;

    Ch_VaStart(interp, ap, varg);
    ii = Ch_VaArg(interp, ap, int);
    ap_ch = Ch_VaArg(interp, ap, ChVaList_t);

    ap_c = Ch_VaVarArgsCreate(interp, ap_ch, &memhandle);
    vfunc(ii, ap_c);
    Ch_VaVarArgsDelete(interp, memhandle);

    Ch_VaEnd(interp, ap);
    return;
}

void vfunc(int n, va_list ap_c) {
    char c1;
    int i1;
    short s1;
    float f1;
    double d1;
    double  *pd1 = &d1;       /* pointer to double */
    double  **ppd1 = &pd1;    /* pointer to double */
    float  *af1;              /* for array */
    SS1    ss1;               /* struct */
    double d2;

    printf("The first argument is %d\n", n);

    c1 = va_arg(ap_c, char);
    printf("\nc1 = %c\n", c1);
    i1 = va_arg(ap_c, int);
    printf("i1 = %d\n", i1);
    s1 = va_arg(ap_c, short);
    printf("s1 = %d\n", s1);
    f1 = va_arg(ap_c, float);
    printf("f1 = %f\n", f1);
    d1 = va_arg(ap_c, double);
    printf("d1 = %f\n", d1);
```

Program 5.12: Example for handling variable arguments – chdl file `vararg.c`.

```
    pd1= va_arg(ap_c, double*);
    printf("\n*pd1 = %f\n", *pd1);
    ppd1= va_arg(ap_c, double**);
    printf("**ppd1 = %f\n", **ppd1);
    af1 = va_arg(ap_c, float*);
    printf("af1[1][1] = %f\n", *(af1+5));

    ss1 = va_arg(ap_c, SS1);
    printf("\nss1.i1 = %d\n", ss1.i1);
    printf("ss1.d1 = %f\n", ss1.d1);
    printf("ss1.c1 = %c\n", ss1.c1);
    printf("ss1.f1 = %f\n", ss1.f1);

    printf("d2 = %f\n", d2);

    return;
}


void func(int n, ...) {
    va_list ap;

    va_start(ap, n);
    vfunc(n, ap);
    va_end(ap);
    return;
}
```

Program 5.12: Example for handling variable arguments – chdl file `vararg.c` (Contd.).

Functions with variable number of arguments can be handled automatically by command **c2chf** with an option `-l func vfunc` as described in section C.1.

5.5.2 Calling Function Multiple Times

The method described in this section can be used if the function satisfies the following conditions:

1. The data types of the remaining arguments in the variable argument list of function `functionName`() are the same, say `data_type2`.
2. Calling function `functionName`() with multiple arguments is equivalent to calling the same function multiple times each time with two or three arguments. That is, function call of

```
functionName(param, arg1, arg2, arg3, arg4) {
```

is equivalent to

```
functionName(param, arg1);
functionName(param, arg2);
functionName(param, arg3);
functionName(param, arg4);
```

we have following chdl function and Ch function.

In the chf file, the Ch function is shown in Program 5.13.

```
#include<dlfcn.h>
  ...
void functionName(data_type1 param, ...)
{
  void *dlhandle, *fptr;
  int vacount, i;
  ChVaList_t ap;
  data_typy2 arg;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  va_start(ap, param);
  vacount = va_count(ap);

  if(vacount == 0)
    dlrunfun(fptr, NULL, functionName, param);
  else
    for(i = 1; i <= vacount; i++) {
      arg = va_arg(ap, data_type2);
      dlrunfun(fptr, NULL, functionName, param, arg);
    }

  va_end(ap);
  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

Program 5.13: Regular functions with arguments of variable length (chf function).

where macro **va_count()** is used to determine the number of arguments. We use a for-loop to call the function multiple times to simulate the calling of function with multiple arguments.

In the chdl file, the chdl function is shown in Program 5.14.

```
#include<ch.h>
    ...

EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  data_type1 param;
  data_type2 arg;

  /* get the arguments from the Ch function */
  Ch_VaStart(interp, ap, varg);
  param = Ch_VaArg(interp, ap, data_type1);
  arg = Ch_VaArg(interp, ap, data_type2);

  functionName(param, arg);
  Ch_VaEnd(interp, ap);
}
```

Program 5.14: Regular functions with arguments of variable length (chdl function).

where function `functionName`() is called with only two argument.

Example

In this example, function `sum1`() is used to describe the case of the function with variable length argument list.

Listing 1 — header file (sum1.h)

```
double sum1(int count, ...);
```

Listing 2 — C function (sum1.c)

```
#include <stdarg.h>
#include <stdio.h>
#include "sum1.h"

double sum1(int count, ...) {
  int i;
  double  retval = 0, d1;
  va_list ap;

  va_start(ap, count);
  if(count < 0) {
    printf("Wrong number of argument\n");
    exit(-1);
  }
  for(i = 0; i < count; i++) {
    d1 = va_arg(ap, double);
    retval += d1;
  }

  return retval;
}
```

This is the C function which can take variable number of arguments.

Listing 3 — chf file (sum1.chf)

```
#include<dlfcn.h>
#include<stdarg.h>

double sum1(int count, ...) {
  void *handle, *fptr;
  int vacount, i;
  va_list ap;
  double d1, d2, retval = 0;

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }

  fptr = dlsym(handle, "sum1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  va_start(ap, count);
  vacount = va_count(ap);
  // for-loop is used to call the chdl function multiple times
  for(i = 0; i < vacount; i++) {
    d1 = retval;
    d2 = va_arg(ap, double);
    dlrunfun(fptr, &retval, sum1, 2, d1, d2);
  }
  va_end(ap);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}
```

In the Ch space, a for-loop is used to implement multiple calling of the chdl function with three argument of `2`, `d1` and `d2`. The value of `2` stands for the number of the arguments in the variable argument list.

Listing 4 — chdl file (sum1_chdl.c)

```
#include <ch.h>
#include "sum1.h"

EXPORTCH double sum1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int count;
  double d1, d2, retval;

  Ch_VaStart(interp, ap, varg);
  count = Ch_VaArg(interp, ap, int);
  d1 = Ch_VaArg(interp, ap, double);
  d2 = Ch_VaArg(interp, ap, double);
  retval = sum1(count, d1, d2);
```

```
  Ch_VaEnd(interp, ap);
  return retval;
}
```

The chdl function calls the C function `sum1`() with three arguments.

Listing 5 — Ch application (sum1.ch)

```
#include "sum1.h"

int main() {

  printf("sum = %f\n", sum1(2.0, 1.1, 2.2));
  printf("sum = %f\n", sum1(3.0, 1.1, 2.2, 3.3));
  printf("sum = %f\n", sum1(4.0, 1.1, 2.2, 3.3, 4.4));
  printf("sum = %f\n", sum1(5.0, 1.1, 2.2, 3.3, 4.4, 5.5));

  return 0;
}
```

With the for-loop in the chf file, applications in Ch space can call `sum1`() with different number of arguments.

Output

```
sum = 3.300000
sum = 6.600000
sum = 11.000000
sum = 16.500000
```

5.5.3 Limited Number of Arguments

If the two conditions in the previous section are not satisfied and the function has a limited number of arguments with different data types, such as the possible callings of function `functionName`() below:

```
functionName(param);
functionName(param, arg1);
functionName(param, arg1, arg2);
functionName(param, arg1, arg2, arg3);
```

where `param`, `arg1`, `arg2` and `arg3` are data types of `data_type`, `data_type1`, `data_type2` and `data_type3`, respectively, the following chdl function and Ch function can be used.

In the chf file, the Ch function is shown in Program 5.15.

```
#include<dlfcn.h>
#include<stdarg.h>
void functionName(data_type param, ...) {
  void *dlhandle, *fptr;
  int vacount;
  va_list ap;
  data_type1 arg1;
  data_type2 arg2;
  data_type3 arg3;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  va_start(ap, param);
  vacount = va_count(ap);
  if(vacount == 0)
    dlrunfun(fptr, NULL, NULL, param);

  if(vacount >= 1) {
    arg1 = va_arg(ap, data_type1);
    if(vacount == 1)
      dlrunfun(fptr, NULL, NULL, param, arg1);
  }

  if(vacount >= 2) {
    arg2 = va_arg(ap, data_type2);
    if(vacount == 2)
      dlrunfun(fptr, NULL, NULL, param, arg1, arg2);
  }

  if(vacount == 3) {
    arg3 = va_arg(ap, data_type3);
    dlrunfun(fptr, NULL, NULL, param, arg1, arg2, arg3);
   }

  va_end(ap);
  if(dlclose(handle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

Program 5.15: Regular functions with a limited number of arguments (chf function).

Here, the API **va_count**() which is only available in Ch space, is used to determine the number of arguments after the first one. In Ch, a function can begin with the variable length argument list without any arguments, for example

```
void functionName(...)
```

If this is the case, **VA_NOARG** is used to replace the first argument in the following statement

```
va_start(ap, VA_NOARG);
```

to start the processing of variable number arguments. More information about this can be found in Ch User's Guide.

In the chdl file, the chdl function is shown in Program 5.16.

```
#include<ch.h>
    ...

EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  int count;
  data_type  param;
  data_type1 arg1;
  data_type2 arg2;
  data_type3 arg3;

  Ch_VaStart(interp, ap, varg);
  /* get the firat argument */
  param = Ch_VaArg(interp, ap, data_type);

  /* get num of arguments from the Ch function */
  count = Ch_VaCount(interp, ap);

  if(count == 0)
    functionName(param);

  if(count >= 1) {
    arg1 = Ch_VaArg(interp, ap, data_type1);
    if(count == 1)
      functionName(param, arg1);
  }

  if(count >= 2) {
    arg2 = Ch_VaArg(interp, ap, data_type2);
    if(count == 2)
      functionName(param, arg1, arg2);
  }

  if(count == 3) {
    arg3 = Ch_VaArg(interp, ap, data_type3);
    functionName(param, arg1, arg2, arg3);
  }

  Ch_VaEnd(interp, ap);
}
```

Program 5.16: Regular functions with a limited number of arguments (chdl function).

where the API **Ch_VaCount**() is used to obtain the number of the rest of arguments in the argument list after the first one `param` is taken.

Example

In this example, function `sum2()` is used to describe the case of the function with a variable-length argument list.

Listing 1 — header file (sum2.h)

```
double sum2(int param, ...);
```

Listing 2 — C function (sum2.c)

```
#include <stdarg.h>

double sum2(int param, ...) {
  va_list ap;

  int ii, ii2;
  float ff, ff2;
  double dd, dd2, retval = 0;

  if (param <= 0) return retval;
  va_start(ap, param);
  if (param >= 1) {
     ii = va_arg(ap, int);
     retval += ii;
   }
   if (param >= 2) {
     ii2 = va_arg(ap, int);
     retval += ii2;
   }
   if (param >= 3) {
     ff = va_arg(ap, float);
     retval += ff;
   }
   if (param >= 4) {
     ff2 = va_arg(ap, float);
     retval += ff2;
   }
   if (param >= 5) {
     dd = va_arg(ap, double);
     retval += dd;
   }
   if (param >= 6) {
     dd2 = va_arg(ap, double);
     retval += dd2;
   }
  va_end(ap);
  return retval;
}
```

Listing 3 — chf file (sum2.chf)

```
#include<dlfcn.h>
#include<stdarg.h>

double sum2(int param, ...) {
  void *dlhandle, *fptr;
```

```
va_list ap;
int count, ii, ii2;
float ff, ff2;
double dd, dd2, retval = 0;

/* load the dynamically loaded library */
dlhandle = dlopen("libhello.dl", RTLD_LAZY);
if(dlhandle == NULL) {
  fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                   __func__, dlerror());
  return -1;
}

/* get the address by function name */
fptr = dlsym(dlhandle, "sum2_chdl");
if(fptr == NULL) {
  printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
  return -1;
}

va_start(ap, param);
count = va_count(ap);
if(count == 0)
    dlrunfun(fptr, &retval, sum2, param);

if(count >= 1) {
    ii = va_arg(ap, int);
    if(count == 1)
      dlrunfun(fptr, &retval, sum2, param, ii);
}

if(count >= 2) {
    ii2 = va_arg(ap, int);
    if(count == 2)
      dlrunfun(fptr, &retval, sum2, param, ii, ii2);
}

if(count >= 3) {
    ff = va_arg(ap, float);
    if(count == 3)
      dlrunfun(fptr, &retval, sum2, param, ii, ii2, ff);
}

if(count >= 4) {
    ff2 = va_arg(ap, float);
    if(count == 4)
      dlrunfun(fptr, &retval, sum2, param, ii, ii2, ff, ff2);
}

if(count >= 5) {
    dd = va_arg(ap, double);
    if(count == 5)
      dlrunfun(fptr, &retval, sum2, param, ii, ii2, ff, ff2, dd);
}

if(count >= 6) {
    dd2 = va_arg(ap, double);
    if(count == 6)
      dlrunfun(fptr, &retval, sum2, param, ii, ii2, ff, ff2, dd, dd2);
```

```
  }

  va_end(ap);
  return retval;
}
```

Listing 4 — chdl file (sum2_chdl.c)

```
#include <ch.h>
#include "sum2.h"

EXPORTCH double sum2_chdl(void *varg){
  ChInterp_t interp;
  ChVaList_t ap;

  int count, param, ii, ii2;
  float ff, ff2;
  double dd, dd2, retval = 0;

  Ch_VaStart(interp, ap, varg);
  param = Ch_VaArg(interp, ap, int);

  count = Ch_VaCount(interp, ap);
  if(count == 0)
     retval = sum2(param);

  if(count >= 1) {
      ii = Ch_VaArg(interp, ap, int);
      if(count == 1)
        retval = sum2(param, ii);
  }

  if(count >= 2) {
      ii2 = Ch_VaArg(interp, ap, int);
      if(count == 2)
        retval = sum2(param, ii, ii2);
  }

  if(count >= 3) {
      ff = Ch_VaArg(interp, ap, float);
      if(count == 3)
        retval = sum2(param, ii, ii2, ff);
  }

  if(count >= 4) {
      ff2 = Ch_VaArg(interp, ap, float);
      if(count == 4)
        retval = sum2(param, ii, ii2, ff, ff2);
  }

  if(count >= 5) {
      dd = Ch_VaArg(interp, ap, double);
      if(count == 5)
        retval = sum2(param, ii, ii2, ff, ff2, dd);
  }

  if(count >= 6) {
      dd2 = Ch_VaArg(interp, ap, double);
      if(count == 6)
        retval = sum2(param, ii, ii2, ff, ff2, dd, dd2);
```

```
  }

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 5 — Ch application (sum2.ch)

```
#include "sum2.h"

int main() {

  printf("sum1 = %f\n", sum2(2, 11, 22));
  printf("sum2 = %f\n", sum2(3, 11, 22, 3.3));
  printf("sum3 = %f\n", sum2(4, 11, 22, 3.3, 4.4));
  printf("sum4 = %f\n", sum2(5, 11, 22, 3.3, 4.4, 5.5));
  printf("sum5 = %f\n", sum2(6, 11, 22, 3.3, 4.4, 5.5, 6.6));


  return 0;
}
```

Output

```
sum1 = 33.000000
sum2 = 36.300000
sum3 = 40.700000
sum4 = 46.200000
sum5 = 52.800000
```

5.6 Functions with Return Value of Struct Type

This case illustrates how to handle a function with a return value of struct type.
Assume that the function to be handled is:

```
/* define a struct named tag_t, better put struct
   definition in a header file */
struct tag_t {...};

struct tag_t functionName(data_type1 arg1, data_type2 arg2) {
  struct tag_t retval;
  ...
  return retval;
}
```

where `retval` is a variable of data type `struct tag_t`.

In the chf file, the Ch function is shown in Program 5.17,

```
#include<dlfcn.h>
...
struct tag_t functionName(data_type1 arg1, data_type2 arg2) {
  void *dlhandle, *fptr;
  struct tag_t retval;

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }

  /* Address of retval is passed as an argument to C address space.
     To avoid checking prototype, the third argument should be NULL */
  dlrunfun(fptr, NULL, NULL, &retval, arg1, arg2);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
            __func__, dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 5.17: Regular functions with return value of struct (chf function).

where `retval` is declared as a variable of type of struct `tag_t`. Its address `&retval` is passed to C space as the fourth argument, instead of the second argument, of function **dlrunfun**(). The second and third arguments are NULLs. This is the key point for this case.

In the chdl file, the chdl function is shown in Program 5.18.

```
#include<ch.h>
...
EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  /* use pointers to access arrays */
  data_type1 arg1;
  data_type2 arg2;
  struct tag_t *pretval;

  /* get arguments passed from the Ch function */
  Ch_VaStart(interp, ap, varg);
  pretval = Ch_VaArg(interp, ap, struct tag_t *); /* address of struct in Ch space */
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  arg2 = Ch_VaArg(interp, ap, data_type2);   /* get 2nd argument  */

  *pretval = functionName(arg1, arg2);

  Ch_VaEnd(interp, ap);
}
```

Program 5.18: Regular functions with return value of struct (chdl function).

where chdl function `functionName_chdl` has no return value. Variable `pretval` is defined as a pointer to struct `tag_t` and it will take the address of the variable `retval` in Ch space. In this function, the content of the return value, which is a struct, of C function `functionName`() is copied over to the memory of the variable `retval` in Ch space by statement

```
*pretval = functionName(arg1, arg2);
```

Example

In this example, function `retstruct`() is used to describe the case of the function returning a struct.

Listing 1 — header file (retstruct.h)

```
struct tag_t {int i; short s; double d;};
struct tag_t retstruct(int i, short *sptr, double d, struct tag_t *ps);
```

Listing 2 — C function (retstruct.c)

```
#include "retstruct.h"

struct tag_t retstruct(int i, short *sptr, double d, struct tag_t *ps) {
  ps->i = i;
  ps->s = *sptr;
  ps->d = d;
  i = 10 * i;
  d = 10 * d;
  *sptr = 10 * (*sptr);
  return *ps;
}
```

Listing 3 — chf file (retstruct.chf)

```
#include<dlfcn.h>

struct tag_t retstruct(int i, short *sptr, double d, struct tag_t *ps) {
  void *handle, *fptr;
  struct tag_t retval;

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return retval;
  }

  fptr = dlsym(handle, "retstruct_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return retval;
  }

  dlrunfun(fptr, NULL, NULL, &retval, i, sptr, d, ps);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return retval;
  }

  return retval;
}
```

Listing 4 — chdl file (retstruct_chdl.c)

```
#include <ch.h>
#include <stdlib.h>
#include "retstruct.h"

EXPORTCH void retstruct_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  int i;
  short *sptr;
  double d;
  struct tag_t *ps;
  struct tag_t *pretval;

  Ch_VaStart(interp, ap, varg);
  pretval = Ch_VaArg(interp, ap, struct tag_t *);
  i = Ch_VaArg(interp, ap, int);
  sptr = Ch_VaArg(interp, ap, short *);
  d = Ch_VaArg(interp, ap, double);
  ps = Ch_VaArg(interp, ap, struct tag_t *);

  *pretval = retstruct(i, sptr, d, ps);

  Ch_VaEnd(interp, ap);
}
```

Listing 5 — Ch application (retstruct.ch)

```
#include "retstruct.h"
```

```
int main() {

  char c;
  short h=2;
  struct tag_t s, p;

  s.i = 10;
  s.s = 20;
  s.d = 30;

  p = retstruct(1, &h, 3, &s);
  printf("h = %d\n", h);
  printf("s.i in main() = %d\n", s.i);
  printf("s.s in main() = %d\n", s.s);
  printf("s.d in main() = %f\n", s.d);
  printf("p.i in main() = %d\n", p.i);
  printf("p.s in main() = %d\n", p.s);
  printf("p.d in main() = %f\n", p.d);

  return 0;
}
```

Output

```
h = 20
s.i in main() = 1
s.s in main() = 2
s.d in main() = 3.000000
p.i in main() = 1
p.s in main() = 2
p.d in main() = 3.000000
```

Functions return structure type can also be handled automatically by command **c2chf** as described in section C.1. However, when the return type is represented by a typedefed struct variable, an option **-s** needs to be used for command **c2chf**.

5.7 Functions with Arguments of Special Data Type

There are some data types in Ch that have to be handled in slightly different ways, such as `string_t` in Ch.

5.7.1 Functions with Arguments of Type string_t

This section discusses how to call a function with ”char *” as an argument in C space from a function in Ch space that have arguments with type string_t.

Assume that the function that we have to deal with has the following prototype:

```
void functionName(string_t arg1) {
  ...
}
```

Program 5.19 is the chf function that would be loaded when the application program runs in Ch.

```
#include<dlfcn.h>

void functionName(string_t s) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libt.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return;
  }

  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, functionName, s);  /* string is valid data type for dlrunfun */

  if(dlclose(dlhandle) != 0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
                        __func__, dlerror());
    return;
  }
}
```

Program 5.19: Regular functions with Arguments of type string_t (chf function).

Notice that the chf function above is very similar to the ones displayed earlier. But unlike other chf functions, this one takes a `string_t` argument from the application program and passes the same argument to the chdl function using **dlrunfun()**.

Let's take a look of chdl file that is shown in Program 5.20. Remember that the chdl function is called by **dlrunfun()**.

```
#include<ch.h>

EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  char *s;

  Ch_VaStart(interp, ap, varg);
  s = Ch_VaArg(interp, ap, char*);       /* get arg from Ch space, s is a string_t  */

  functionName(s);
  free(s);  /* s cannot be saved and registered  in functionName and used
            later */
            /* do not use string_t for callback function etc. Without this,
            memory will be leaked */

  Ch_VaEnd(interp, ap);
}
```

Program 5.20: Regular functions with argument of type string_t chdl function).

Since `string_t` is a data type that is specifically designed for Ch, it will not be recognized in the C space. As a result, we have to declare the string variable type `char *` in the chdl file to store a copy of the `string_t` argument that is passed in. The `string_t` variable in the Ch space will be deallocated automatically by the Ch kernel when the function exists. However, the memory allocated for the string in the C space has to be freed by function `free()` to avoid a memory leak. Do not use functions with argument of `string_t` as callback functions.

5.7.2 Functions with Arguments of Variable Length Arrays (VLAs)

5.7.2.1 Assumed-Shape Array

In section 5.4, we have introduced how to handle functions with argument of arrays which have fixed dimensions and extents. The array names which represent the addresses of arrays have been passed to the C space. Because numbers of dimensions and extents are fixed, no information other than addresses of arrays need to be passed to the C space in that case. If a C function handles arrays with different extents, it typically takes information of extents from the argument list. For example, a C function which performs array multiplication of `C = A * B` would have the prototype of

```
int arrmul(double **pa, double **pb, double **pc,
           int n, int m, int r);
```

where arguments of double pointers, `pa`, `pb` and `pc`, represent two-dimensional arrays with shape of $(n \times m)$, $(m \times r)$ and $(n \times r)$. Due to different `n`, `m` and `r`, the extents of arrays in the multiplication operations can be different. There are many different ways to build the corresponding Ch function and chdl function to call a C function like `arrmul`(). One of them is to use a Ch function with the same prototype as the corresponding C function. If this is the case, the Ch function and chdl function can be coded according to the templates of cases of regular functions. Another way is that the Ch function takes only arrays, either C array or Ch computational array, as arguments without any other information. If this is the case, the prototype of the corresponding Ch function `arrmul`() can be

```
int arrmul(array double a[:][:], array double b[:][:],
           array double c[:][:]);
```

where arguments a, b and c are assumed-shape computational arrays instead of double pointers. Please refer to *Ch User's Guide* for more information about Ch computational arrays. With assumed-shape arrays, a Ch function can take arrays with different extents. A function with this prototype is more convenient to use for Ch users. To make this Ch function work with the C function which needs more information of arrays, we need extract the information, including numbers of dimensions and extents, in the Ch function or the chdl function.
Programs 5.21 along with 5.22 is a complete example of array multiplication. In this example (Program 5.21), either C arrays or Ch computational arrays are passed to the Ch function. In the chdl function arrmul_chdl() in Program 5.22, we extract the information about these arrays with functions Ch_VaArrayType(), Ch_VaArrayExtent() and Ch_VaArrayDim(). For example, the code below

```
if(!Ch_VaArrayType(interp, ap)) {
  printf("The 1st argument should be an array.\n");
  exit(-1);
}
```

print out an error message if the first argument is not an array. The code

```
if(Ch_VaArrayDim(interp, ap) != 2) {
    ...
}
```

determines if the dimension of the array is 2, and

```
ext_a_0 = Ch_VaArrayExtent(interp, ap, 0);
```

retrieves the extent of the first dimension of the array. To match the prototype of the C function, we also need to make three double pointers pa, pb and pc to represent these passed-in arrays. Then the statement

```
retval = arrmul(pa, pb, pc, ext_a_0, ext_a_1, ext_b_1);
```

calls the C function with all necessary information.

5.7.2.2 Array of Reference

Here is another case. Assume the C function

```
double summary(double *aa, int dim, int *ext);
```

calculates the summary of all elements of an array. The first argument aa is a pointer which represents a one-dimensional array. As we have known, with enough information, aa can also represents a multi-dimensional array. The second argument, dim, contains the number of the dimension, and the third one, ext, contains extents of all dimensions. The Ch function with the function prototype

```
double summary(array double &a);
```

can be used, where a is array of reference. Please refer to *Ch User's Guide* for more information about Ch computational array of reference. With arrays of reference, a Ch function can take arrays with different dimensions, extents, and even different data types. Program 5.23 along with 5.24 is a complete example of calculating the summary of each element of an array. Unlike the previous example, in which the information of array is extracted in the chdl function, here we extract the information in the Ch function with the function shape(). Given an n-dimensional array a, the function call

```
/* File name: arrmul.ch */
#include<dlfcn.h>
#include<array.h>
int arrmul(array double a[:][:], array double b[:][:], array double c[:][:]) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libvlaarg.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }
  fptr = dlsym(handle, "arrmul_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }
  dlrunfun(fptr, &retval, arrmul, a, b, c);
  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}

int main() {
  array double a1[2][3] = { 1, 5, 3,
                            5, 6, 7};
  double b1[3][2] = { 1, 5,
                      5, 6,
                      3, 7};
  array double c1[2][2];
  array double a2[3][4] = { 1, 5, 3, 7,
                            5, 6, 7, 3,
                            1, 2, 6, 8};
  double b2[4][3] = { 1, 5, 4,
                      5, 6, 9,
                      3, 7, 5,
                      2, 3, 4};
  array double c2[3][3];

  arrmul(a1, b1, c1);
  printf("c1 = \n%f\n", c1);
  arrmul(a2, b2, c2);
  printf("c2 = \n%f", c2);

  return 0;
}
```

Program 5.21: arrmul.ch: Example of array multiplication with arguments of VLAs (Ch and chf file).

```
/* arrmul.c
   this case is for the C function which takes arguments of
   double ** (two dimensions), rather than double * (one dimension)
*/
#include <ch.h>
#include <stdio.h>

int arrmul(double **pa, double **pb, double **pc, int n, int m, int r);

EXPORTCH int arrmul_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int i;
  double *a, *b, *c;
  double **pa, **pb, **pc; /* pointer to temporary memory */
  int ext_a_0, ext_a_1;
  int ext_b_0, ext_b_1;
  int ext_c_0, ext_c_1;
  int retval;

  Ch_VaStart(interp, ap, varg);

  if(!Ch_VaArrayType(interp, ap)) {
      printf("The 1st argument should be an array.\n");
      exit(-1);
  }
  if(Ch_VaArrayDim(interp, ap) != 2) {
      printf("Wrong dimension of the 1st argument.\n");
      exit(-1);
  }
  ext_a_0 = Ch_VaArrayExtent(interp, ap, 0);
  ext_a_1 = Ch_VaArrayExtent(interp, ap, 1);
  a = Ch_VaArg(interp, ap, double*);

  if(!Ch_VaArrayType(interp, ap)) {
      printf("The 2nd argument should be an array.\n");
      exit(-1);
  }
  if(Ch_VaArrayDim(interp, ap) != 2) {
      printf("Wrong dimension of the 2nd argument.\n");
      exit(-1);
  }
  ext_b_0 = Ch_VaArrayExtent(interp, ap, 0);
  ext_b_1 = Ch_VaArrayExtent(interp, ap, 1);
  b = Ch_VaArg(interp, ap, double*);

  if(!Ch_VaArrayType(interp, ap)) {
      printf("The 3rd argument should be an array.\n");
      exit(-1);
  }
  if(Ch_VaArrayDim(interp, ap) != 2) {
      printf("Wrong dimension of the 3rd argument.\n");
      exit(-1);
  }
  ext_c_0 = Ch_VaArrayExtent(interp, ap, 0);
  ext_c_1 = Ch_VaArrayExtent(interp, ap, 1);
  c = Ch_VaArg(interp, ap, double*);
```

Program 5.22: arrmul.c: Example of array multiplication with arguments of VLAs (C and chdl file).

```
  /* check if extents of these three arrays match for array multiplication */
  if(ext_a_1 == ext_b_0 && ext_a_0 == ext_c_0 && ext_b_1 == ext_c_1) {
      if( !(pa = (double **)malloc(ext_a_0 * sizeof(double*))) ||
          !(pb = (double **)malloc(ext_b_0 * sizeof(double*))) ||
          !(pc = (double **)malloc(ext_c_0 * sizeof(double*))) ) {
         printf("Can't allocate memory.\n");
         exit(-1);
      }
      for(i = 0; i <= ext_a_0-1; i++) {
          pa[i] = &a[i*ext_a_1];
          pc[i] = &c[i*ext_c_1];
      }
      for(i = 0; i <= ext_b_0-1; i++) {
          pb[i] = &b[i*ext_b_1];
      }
      retval = arrmul(pa, pb, pc, ext_a_0, ext_a_1, ext_b_1);
  }
  else {
      printf("The extents of arrays don't match.\n");
      exit(-1);
  }
  free(pa); free(pb); free(pc);
  Ch_VaEnd(interp, ap);
  return retval;
}

/* c[n][r] = a[n][m] * b[m][r] */
int arrmul(double **pa, double **pb, double **pc, int n, int m, int r) {
  int i, j, k;

  for(i = 0; i <= n-1; i++) {
      for(j = 0; j <= r-1; j++) {
          pc[i][j] = 0;
          for(k = 0; k <= m-1; k++)
              pc[i][j] += pa[i][k] * pb[k][j];
      }
  }
  return 0;
}
```

Program 5.22: Example of array multiplication with arguments of VLAs (arrmul.c).

```
ext = shape(a)
```

returns a one-dimensional array `ext` with n elements which contain all extents of array `a`. Therefore, the function call

```
dim = shape(shape(a))
```

returns the number of the dimensions of `a`. The statement

```
aa = (array double [totnum])a;
```

casts computational array `a`, which can be different dimension and different data types, to `aa` which is an one-dimensional computational array with type double. After that, all necessary information with array `aa` will be passed to the C space with the statement below.

```
dlrunfun(fptr, &retval, NULL, aa, dim, ext);
```

```
/* File name: summary.ch */
#include<dlfcn.h>
#include<array.h>
double summary(array double &a);

double summary(array double &a) {
  void *handle, *fptr;
  double retval;
  int dim = (int)shape(shape(a));
  array int ext[dim];
  int i, totnum = 1;
  double sums;

  ext = shape(a);
  for(i = 0; i < dim; i++) {
     totnum = totnum * ext[i];
  }
  array double aa[totnum];
  aa = (array double [totnum])a; /* cast all data types to double */

  handle = dlopen("libvlaarg.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return NaN;
  }

  fptr = dlsym(handle, "summary_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NaN;
  }
  dlrunfun(fptr, &retval, NULL, aa, dim, ext);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return NaN;
  }
  return retval;
}

int main() {
  array double a1[2][3] = { 1, 5, 3,
                            5, 6, 7};
  int b1[3] = { 1, 5, 3};

  printf("summary = %f\n", summary(a1));
  printf("summary = %f\n", summary(b1));
  return 0;
}
```

Program 5.23: Example of calculating summary with arguments of VLAs (summary.ch).

```
/* summary.c
   this case is for the C function which takes
   arguments of double * (one dimensions) to represent arrays
   of different dimension, as well as information about the
   array including dimensions and extents.
*/
#include <ch.h>
#include <stdio.h>

double summary(double *aa, int dim, int *ext);

EXPORTCH double summary_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  double *a;
  int dim, *ext;
  double retval = 0;

  Ch_VaStart(interp, ap, varg);

  a = Ch_VaArg(interp, ap, double *);
  dim = Ch_VaArg(interp, ap, int);
  ext = Ch_VaArg(interp, ap, int*);

  retval = summary(a, dim, ext);

  Ch_VaEnd(interp, ap);
  return retval;
}

double summary(double *aa, int dim, int *ext) {
    int i, totnum = 1;
    double sums = 0;

    for(i = 0; i < dim; i++) {
      totnum = totnum * ext[i];
    }
    for(i = 0; i < totnum; i++) {
      sums = sums + aa[i];
    }
    return sums;
}
```

Program 5.24: Example of calculating summary with arguments of VLAs (summary.c).

5.7.3 Functions with Return Value of Ch Computational Array

A lot of existing Ch functions designed for numeric computations are taking the advantage of the feature of Ch computation arrays. Sometimes, it is more efficient to make some computational jobs done in C space and return the result in the form of Ch computational array to Ch. Unlike previous sections where cases are focused on how to port C code to Ch, cases in this section will illustrate how to make up Ch functions, chdl functions, as well as C function if necessary, to work with Ch computational arrays.

Let's get started with writing a Ch function which returns a Ch computational array. It is shown in Program 5.25,

```
#include<dlfcn.h>
...
array return_type functionName(data_type1 arg1, data_type2 arg2)[ARRAY_DIM] {
  void *dlhandle, *fptr;
  array return_type retval[ARRAY_DIM];

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                      __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }

  /* Address of retval is passed as an argument to C address space.
     To avoid checking prototype, the third argument should be NULL */
  dlrunfun(fptr, NULL, NULL, &retval[0], arg1, arg2);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
            __func__, dlerror());
    return FAIL_VALUE;
  }
  return retval;
}
```

Program 5.25: Regular functions with return value of Ch computational array (chf function).

where `retval` is defined as a **Ch Computational Array**, and `ARRAY_DIM` is a predefined macro that indicates the fixed size of the array. The address of the first element of `retval` `&retval[0]` is passed to C space as the fourth argument of function **dlrunfun**(), and the second and the third arguments are NULLs.

In the chdl file, the chdl function is shown in Program 5.26,

```
#include<ch.h>
...
EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  return_type *pretval;
  data_type1 arg1;
  data_type2 arg2;

  /* get arguments passed from the Ch address space */
  Ch_VaStart(interp, ap, varg);
  pretval = Ch_VaArg(interp, ap, retrun_type*); /* address of the array in Ch space */
  arg1 = Ch_VaArg(interp, ap, data_type1);     /* get 1st argument  */
  arg2 = Ch_VaArg(interp, ap, data_type2);     /* get 2nd argument  */
  /*
      ...
      here is code to do computational jobs
      and copy result into memory pointed to by pretval
  */
  Ch_VaEnd(interp, ap);
}
```

Program 5.26: Regular functions with return value of Ch computational array (chdl function).

where `pretval` is defined as a pointer to `return_type` to hold the address of the computational array passed from the Ch space. Then, we can do some computational jobs in C space and put the result to the computational array by its address.

Example

In this example, the Ch function `ret_comp_arr`() returns a Ch computational array with extent of 5.

Listing 1 — header file (ret_comp_arr.h)

```
#define ARRAY_DIM 5  /* the extent of the computational array */
```

In this header file, the extent of the computational array is defined as a macro. Both C and Ch files will use this file.

Listing 2 — Ch application (ret_comp_arr.ch)

```
#include <array.h>
#include "ret_comp_arr.h"

array double ret_comp_arr() [ARRAY_DIM];

int main() {
  array double a[ARRAY_DIM];

  a = ret_comp_arr();
  printf("a in main() = %f\n", a);
}
```

Being different from the template shown in Program 5.25, the Ch function `ret_comp_arr()` does not take any argument.

Listing 3 — chf file (ret_comp_arr.chf)

```
#include <dlfcn.h>
#include <array.h>
#include "ret_comp_arr.h"

array double ret_comp_arr()[ARRAY_DIM] {
  void *handle, *fptr;
  array double retval[ARRAY_DIM];

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return retval;
  }

  fptr = dlsym(handle, "ret_comp_arr_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return retval;
  }

  /* pass the address of the first element of retval
     to the C space */
  dlrunfun(fptr, NULL, NULL, &retval[0]);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return retval;
  }

  return retval;
}
```

Listing 4 — chdl file (ret_comp_arr_chdl.c)

```
#include <stdio.h>
#include <ch.h>
#include "ret_comp_arr.h"

EXPORTCH void ret_comp_arr_chdl(void *varg) {
  ChInterp_t interp;
  int i;
  ChVaList_t ap;

  double *pa;

  Ch_VaStart(interp, ap, varg);
  pa  = Ch_VaArg(interp, ap, double *); /* get address of array in Ch space */

  /* do some computational jobs as follows */
  for(i=0; i<ARRAY_DIM; i++) {
    *pa = i+1;
    pa++;
  }
```

```
  Ch_VaEnd(interp, ap);
}
```

Here, the for-loop stands for some complicated computational jobs in practical cases.

Output

```
a in main() = 1.000000 2.000000 3.000000 4.000000 5.000000
```

5.7.4 Functions with Return Value of Variable Length Computational Array

Based on the previous section, here we will continue with the discussion of functions with return value of variable length computation arrays (VLCA).

The corresponding chf file of the template is shown in Program 5.27,

```
#include<dlfcn.h>
...
array return_type functionName(int array_dim1, int array_dim2,
                          data_type1 arg1, data_type2 arg2) [:][:] {
  void *dlhandle, *fptr;
  array return_type retval[array_dim1][array_dim2];

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
            __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "functionName_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }

  /* Address of the computational array is passed as an argument
     to C address space.
     To avoid checking prototype, the third argument should be NULL */
  dlrunfun(fptr, NULL, NULL, &retval[0][0],
           array_dim1, array_dim2, arg1, arg2);

  /* close the dynamically loaded library */
  if(dlclose(handle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
           __func__, dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 5.27: Regular functions with return value of VLCA (chf function).

where `[:][:]` in the function header indicates that the dimension of the returned Ch computational array is 2, but the extents of each dimension are variable. This is the main difference between Program5.27 and Program 5.25 where extents of the returned array are fixed. The address of the first element of the computational array, i.e. `&retval[0][0]`, as well as its extents which are determined at runtime are passed to the C space as the arguments of function **dlrunfun**.

The chdl function is shown in Program 5.28,

```
#include<ch.h>
  ...
EXPORTCH void functionName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  int array_dim1, array_dim2;
  data_type1 arg1;
  data_type2 arg2;
  return_type *pretval;

  /* get arguments including address and dimentions of
     array passed from the Ch address space */
  Ch_VaStart(interp, ap, varg);
  /* get address of the returned array in Ch */
  pretval = Ch_VaArg(interp, ap, return_type*);
  array_dim1 = Ch_VaArg(interp, ap, int); /* get 1st dimention */
  array_dim2 = Ch_VaArg(interp, ap, int);  /* get 2nd dimention */
  arg1 = Ch_VaArg(interp, ap, data_type1); /* get 1st other argument */
  arg2 = Ch_VaArg(interp, ap, data_type2); /* get 2nd other argument */

  /*
      ...
      here is code to do computational jobs
      and put result into memory pointed to by pretval
  */

  Ch_VaEnd(interp, ap);
}
```

Program 5.28: Regular functions with return value of VLCA (chdl function).

It is similar to the chdl function shown in Program 5.26 except that it has two arguments specifying the size of each dimension of the array.

Example

This example shows how to handle a function which returns a variable length computational array (VLCA).

Ch function – comp_array.chf is shown in Program 5.29,

```
#include<dlfcn.h>
#include<array.h>

array double comp_array(int i) [:] {
  void *handle, *fptr;
  array double retval[i];

  handle = dlopen("libhello.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return retval;
  }

  fptr = dlsym(handle, "comp_array_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return retval;
  }

  /* pass the first element of retval to C space */
  dlrunfun(fptr, NULL, NULL, &retval[0], i);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return retval;
  }

  return retval;
}
```

Program 5.29: Example of functions with return value of VLCA (chf file).

where `[:]` in the function header indicates the returned array has one dimension with variable extent. The argument `i` will contain the extent of the array at runtime.

C and chdl file – The chdl function and corresponding C function `comp_array()` is shown in Program 5.30,

```
#include<stdio.h>
#include<ch.h>

void comp_array(double *pa, int n) {
  int i;
  double  *pa_tmp;

  /* take the address of Ch computational array */
  pa_tmp = pa;

  for(i=0; i<n; i++) {
    *pa_tmp = i+1;
     pa_tmp++;
  }
}

EXPORTCH void comp_array_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;

  int nn;
  double *pa;

  Ch_VaStart(interp, ap, varg);
  pa  = Ch_VaArg(interp, ap, double *); /* get address of array in Ch space */
  nn = Ch_VaArg(interp, ap, int); /* get 1st arg  */

  comp_array(pa, nn);

  Ch_VaEnd(interp, ap);
}
```

Program 5.30: Example of functions with return value of VLCA (chdl file).

Here we use a separated C function `comp_array`() to handle the computational jobs in C space.

Application – comp_array.ch is shown in Program 5.31. Two arrays of one dimension, but different extents, are tested in this program.

```
#include <array.h>

array double comp_array(int i) [:];

int main() {
  array double a[5];
  array double b[10];

  a = comp_array(5);
  printf("a in main() = %f\n", a);
  b = comp_array(10);
  printf("b in main() = %f\n", b);
}
```

Program 5.31: Example of functions with return value of VLCA (Application).

Output is shown in Program 5.32,

```
a in main() = 1.000000 2.000000 3.000000 4.000000 5.000000

b in main() = 1.000000 2.000000 3.000000 4.000000 5.000000
6.000000 7.000000 8.000000 9.000000 10.000000
```

Program 5.32: Example of functions with return value of VLCA (Result).

Chapter 6

Templates for Functions with Pointer to Function

Pointers are powerful in C. Pointer to function allows C programmers to do sophisticated manipulations with functions. In this chapter, we will discuss functions that have arguments or return values of pointer to function. This chapter is more difficult to comprehend compared to the previous chapters due to the subject matter. Callback Ch functions from C space in Section 2.3 is required to understand first. It would be very helpful to read *Ch User's Guide* and learn about pointer to function first if you are new to the concept of pointer to function. Most examples presented in this chapter are commonly used to set callback functions and then get callback functions. This chapter is organized in a way that users will learn how to register/set function pointer in C space and get function pointer from the C space. Typically, setting function pointer and getting function pointer are used in conjunction, the functions are defined in Ch space. Some complete examples that invoke functions for both setting and getting function pointer will be given. It is crucial to read the Section 6.1 thoroughly, because it has detailed explanations for the codes that will not be repeated for other sections.

6.1 Functions with Arguments of Pointer to Function in C space

6.1.1 Pointer to Function without Return Value and Argument in C space

Let's look at a C space program with pointer to function before the general templates are given. The function `setFunction1`() in Program 6.1 takes two arguments. The first one is a normal argument with type of `int`. The second one is a pointer to function which has no argument and return value. The function call using the pointer to function is the only statement in the definition of function `setFunction1`().

```
#include <stdio.h>
void setFunction1(int a, void (*f)()) { // int a is not used here
    if ( f != NULL)
    {
        f();
    }
}
```

Program 6.1: Function taking argument of pointer to function.

In the Ch space, the application `setfuncptr1.ch` shown in Program 6.2, the function `f`() is passed to `setFunction1()` as a pointer.

```
#include <stdio.h>

void f() {
  printf("This is printed from f()\n");
}

int main() {
  setFunction1(10, f);

  return 0;
}
```

Program 6.2: setfuncptr1.ch: Ch application calling function setFunction1()

the expected result from execution of the application `setfuncptr1.ch` is

```
This is printed from f()
```

The C function `setFunction1`() mentioned above can be generalized as the following function,

```
void setFunction1(data_type1 arg1, void (*funptr)()) {
  ...
}
```

where the first argument `arg1` is simple data type `data_type1`. The simple data type `data_type1` can be any one of the following type: `char, short, int, float, double, complex, double complex, char*, short*, int*, float*, double*, complex double*, char**, int**, union, pointer to struct, etc.` The complex data types such as class, the pointer to function and VLA do not belong to the simple data type. The second argument `funptr` is a pointer to the function which has no return value and argument. In circumstances of callback, function `setFunction1`() can be considered as an API to register a user-defined Ch callback function pointed to by `funptr`. This user-defined function `funptr` can be used to handle certain system event, for example, the left mouse button being pushed or the menu item being selected. After the function pointer `funptr` is registered, every time when the left mouse button is pushed, the user-defined function will be invoked through function pointer `funptr`. The first argument `int` is optional in this case.

The corresponding Ch function `setFunction1`() is shown in Program 6.3. Two arguments of function `setFunction1`(), `arg1` and `funptr`, are passed to `setFunction1_chdl`() in the C space as the fourth and fifth arguments of function **dlrunfun()**. `setFunction1_chdl`() can be found in Program 6.4. Note that the argument `funptr` in Program 6.3 has to be NULL or a pointer which points to a valid local or global function in Ch space. If it is a pointer pointing to a C function or other improper memory address, the function `dlrunfun`() will not work properly. In function `setFunction1_chdl()`, `funptr` will be saved and replaced by a function which is defined in C space. The statements opening and closing the DLL file, say `libproject.dl`, should be moved to a Ch header file as illustrated in Chapter 3 if there are multiple functions to be handled.

The chdl function `setFunction1_chdl`() which runs in C space is shown in Program 6.4. One data type is defined and two static variables are declared before the definition of the chdl function. Type `funcHandle` is typedefed as a pointer to a function that has no return value and argument. This definition

```
#include<dlfcn.h>

void setFunction1(data_type1 arg1, void (*funptr)()) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "setFunction1_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address,
     arguments including function pointer are passed to C space */
  dlrunfun(fptr, NULL, setFunction1, arg1, funptr);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

Program 6.3: Argument of pointer to function without return value and argument (chf file).

should match the second argument of `setFunction1`(). The variable `setFunction1_chdl_funptr` is defined as a static pointer which is used to save the function pointer passed from Ch space. Since function `setFunction1_chdl_funarg`() is a static C function which is used to replace the Ch function pointer, it is very critical to make the prototype of this C function consistent with that of the Ch function `setFunction1()`. This rule applies to all remaining cases of setting and getting function pointers.

In the definition of function `setFunction1_chdl`(), the Ch function pointer that is passed from the Ch function in Program 6.3 is obtained and saved with statements

```
handle_ch = Ch_VaArg(ap, funcHandle);
setFunction1_chdl_funptr = (void *)handle_ch;
```

`handle_ch` is cast to be compatible with `setFunction1_chdl_funptr`. The C function `setFunction1`() is then called with two arguments as follows

```
setFunction1(arg1, handle_c);
```

If the Ch function pointer `handle_ch` is not a NULL pointer, the second argument `handle_c` points to the static C function `setFunction1_chdl_funarg`(), which is also defined in this chdl file, otherwise, NULL will passed as the function pointer. Since a function in the C address space cannot call a function defined in the Ch space directly, the call to the Ch function pointed to by `setFunction1_chdl_funptr` are directed to the C function `setFunction_chdl_funarg`(). This is the key point of cases of setting function pointers. In other words, the C function `setFunction1_chdl_funarg`() is called in C space instead of the Ch function. In turn, `setFunction1_chdl_funarg`() calls the Ch function pointed to by `setFunction1_chdl_funptr` with statement

```
Ch_CallFuncByAddr(interp, setFunction1_chdl_funptr, NULL);
```

```
#include<ch.h>
#include<stdio.h>
/* ... */

typedef void (*funcHandle)(); /* function pointer type */

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static void setFunction1_chdl_funarg();

/* save the function pointer from the Ch space */
static void *setFunction1_chdl_funptr;

EXPORTCH void setFunction1_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument */
  /* get and save Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction1_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction1_chdl_funarg;
  }

  /* set the C function pointer instead of the Ch one,
     the NULL pointer will not be replaced */
  setFunction1(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static void setFunction1_chdl_funarg() {
  /* Call Ch function by its address */
  Ch_CallFuncByAddr(interp, setFunction1_chdl_funptr, NULL);
}
```

Program 6.4: Argument of pointer to function without return value and argument (chdl file).

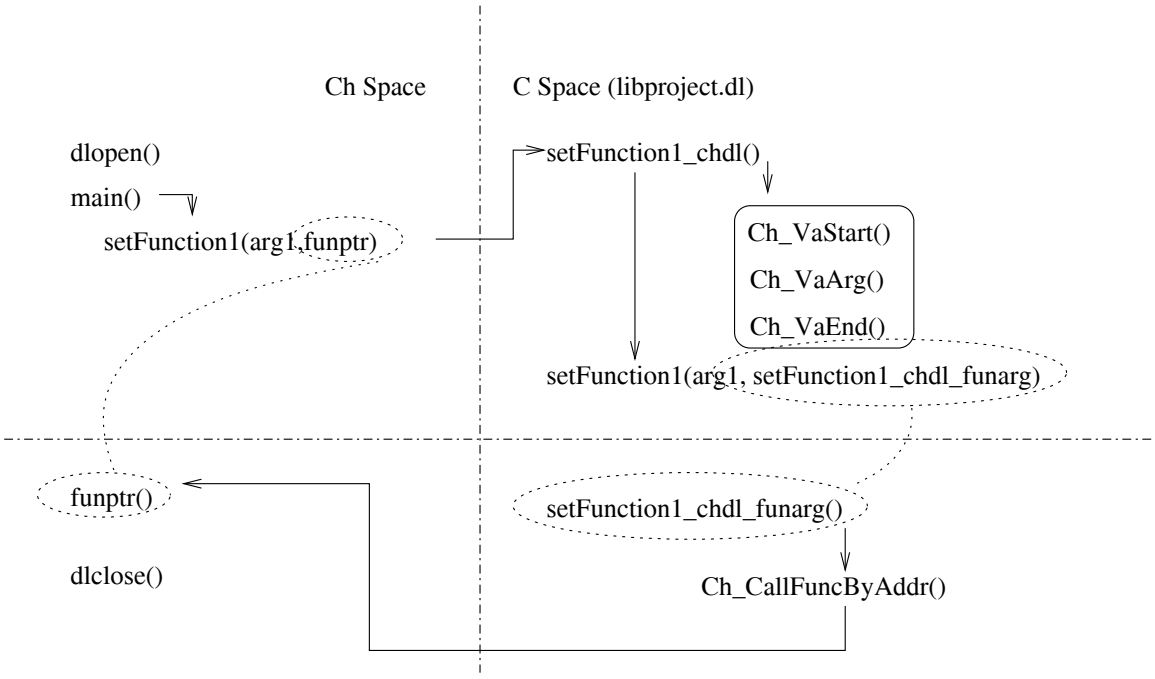


Figure 6.1: Functions and APIs in Ch and C spaces for setting a function.

More information about interfacing Ch function from C space with the API **Ch_CallFuncByAddr**() is available in section 2.4. In this case, the first argument `interp` of static duration was assigned a Ch interpreter by function **Ch_VaStart**() inside function `setFunction1_chdl()`. The second argument of **Ch_CallFuncByAddr**() is the Ch function pointer. The third argument is `NULL`, because the Ch function has no return value. If the function does return a value, then its second argument would be `&retval`. The case of the pointer to function with return value will be introduced in the next section.

If function `setFunction1_chdl`() is called for multiple times to set different Ch function pointers, the last pointer will replace the previous one, because the static variable `setFunction1_chdl_funptr` only holds the last Ch function pointer.

Figures 6.1 illustrates the relation between the functions setting function pointers, and functions being set in both Ch and C spaces. In Ch space, the function `setFunction1`() is called to set the Ch function `funptr`(). The function name `funptr` is passed as an argument to the chdl function `setFunction1_chdl`() in C space. In C space, the chdl function calls C function `setFunction1`() to set a function pointer. The function to be set in C space is the C function `setFunction1_chdl_funarg`() instead of the Ch function `funptr`(), because in C space, Ch functions such as `funptr`(), can only be called by the Ch APIs **Ch_CallFuncByAddr**() and **Ch_CallFuncByName**(). If a Ch function is set in C space, and called back later from C space directly, the program will not work properly.

Example

This is a complete example that illustrates the case of setting pointer to function that has no return value and argument.

Listing 1 — Ch function file (`setFunction1.chf`)

```
#include<dlfcn.h>
```

```
void setFunction1(int arg1, void (*funptr)()) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libsetfuncptr.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "setFunction1_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address,
     arguments including function pointer are passed to C space */
  dlrunfun(fptr, NULL, setFunction1, arg1, funptr);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

The above Chf function is almost the same as the template shown in Program 6.3, except for the DLL file `libsetfuncptr.dl`.

Listing 2 — chdl file (`setFunction1_chdl.c`)

```
#include <stdio.h>
#include <ch.h>

typedef void (*funcHandle)(); /* function pointer type */

/* C functions to replace the Ch one */
static ChInterp_t interp;
static void setFunction1_chdl_funarg();

/* function pointers to save Ch function pointers */
static void *setFunction1_chdl_funptr;

EXPORTCH void setFunction1_chdl(void *varg){
  ChVaList_t ap;
  int arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, int);
  handle_ch = Ch_VaArg(interp, ap, funcHandle); /* get ch function pointer */
  setFunction1_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction1_chdl_funarg;
  }

  setFunction1(arg1, handle_c);
  Ch_VaEnd(interp, ap);
```

```
}

static void setFunction1_chdl_funarg() {
  Ch_CallFuncByAddr(interp, setFunction1_chdl_funptr, NULL);
}
```

It is actually the template shown in Program 6.4.

Listing 3 — C function to be handled (`setFunction1.c`)

```
#include <stdio.h>
void setFunction1(int a, void (*f)()) { // int a is not used here
    if ( f != NULL)
    {
        f();
    }
}
```

Listing 4 — Makefile to build DLL (`Makefile`)

```
# for cases pointer to function

INC1=-I/usr/include
INC2=-I/usr/ch/extern/include

target: libsetfuncptr.dl

libsetfuncptr.dl: setFunction1.o setFunction1_chdl.o
        ch dllink libsetfuncptr.dl \
                  setFunction1.o setFunction1_chdl.o

setFunction1.o: setFunction1.c
        ch dlcomp libsetfuncptr.dl setFunction1.c $(INC1) $(INC2)

setFunction1_chdl.o: setFunction1_chdl.c
        ch dlcomp libsetfuncptr.dl setFunction1_chdl.c $(INC1) $(INC2)

clear:
        rm -f  *.o
clean:
        rm -f  *.o
        rm -f  lib*.dl
```

Listing 5 — Ch Application (`setfuncptr1.ch`)

```
#include <stdio.h>

void f() {
  printf("This is printed from f()\n");
}

int main() {
  setFunction1(10, f);

  return 0;
}
```

After making the DLL file, the result from direct execution of application `setfuncptr1.c` is

```
This is printed from f()
```

```
#include<dlfcn.h>

void setFunction2(data_type1 arg1, retrun_type (*funptr)()) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "setFunction2_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address,
     arguments including function pointer are passed C space */
  dlrunfun(fptr, NULL, setFunction2, arg1, funptr);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

Program 6.5: Argument of pointer to function with return value (chf file).

6.1.2 Pointer to Function with Return Value

This section discuss functions with arguments of pointers to functions that have return values. Assume the function to be called in C space is:

```
void setFunction2(data_type1 arg1, return_type (*funptr)()) {
  ...
}
```

This function would be the same as the one in the first section if its second argument did not have a return value.

The Ch function `setFunction2`() is given in Program 6.5. The only difference between this chf file from the one shown in Program 6.3 is the type of the pointer in the argument list. It is defined as a pointer to function with a return value of data type `return_type`.

In the chdl file, the chdl function `setFunction2_chdl`() is shown in Program 6.6. Note that the typedef statement declares `funcHandle` in accordance to the second argument of the Ch function `setFunction2`() in the beginning of this section. To be compatible with the Ch function, the C replacement function `setFunction2_chdl_funarg`() has return value too. The return value is obtained from the Ch function with the statement

```
Ch_CallFuncByAddr(interp, setFunction2_chdl_funptr, &retval);
```

where the third argument of **Ch_CallFuncByAddr()** is the address of the return value.

```
#include<ch.h>
#include<stdio.h>
/* ... */

typedef return_type (*funcHandle)(); /* function pointer type */

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static return_type setFunction2_chdl_funarg();

/* for saving the function pointer from Ch space */
static void *setFunction2_chdl_funptr;

EXPORTCH void setFunction2_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle  handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get and save Ch function pointer  */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction2_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one */
  if(handle_ch != NULL) { /* the NULL pointer can't be replaced */
    handle_c = (funcHandle)setFunction2_chdl_funarg;
  }

  setFunction2(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static return_type setFunction2_chdl_funarg() {
  return_type retval;

  /* Call Ch address space function by its address */
  Ch_CallFuncByAddr(interp, setFunction2_chdl_funptr, &retval);
  return retval;
}
```

Program 6.6: Argument of pointer to function with return value (chdl file).

```
#include<dlfcn.h>

void setFunction3(data_type1 arg1, void(*funptr)(data_type2)) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "setFunction3_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address,
     arguments including function pointer are passed to C space */
  dlrunfun(fptr, NULL, setFunction3, arg1, funptr);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

Program 6.7: Argument of pointer to function with argument (chf file).

6.1.3 Pointer to a Function with Arguments

This section gives a template including chf and chdl files that are used for functions with arguments of pointers to functions that have arguments. Let's say we have the following function:

```
void setFunction3(data_type1 arg1,
                  void (*funptr)(data_type2 arg2)) {
  ...
}
```

where `setFunction3`() has two arguments and no return value. The first argument `arg1` is of simple data type `data_type1`. The second argument `funptr` is a pointer to function which has one argument `arg2` of simple data type `data_type2` and has no return value.

The chf function is shown in Program 6.7. It is different from Program 6.5 in function argument only.

The chdl function `setFunction3_chdl`() is shown in Program 6.8. The typedef statement in the chdl file should be identical to the one in the chf file. Now, let's compare this file with the previous two chdl files. This one is different from the chdl file shown in Program 6.6 in a way that this function does not return value. As a result, **Ch_CallFuncByAddr**() calls the function that `funcHandle` points to by passing `NULL` in as the third argument instead of `&retval`. This is also the reason why the line

```
return retval;
```

in function `setFunction2_chdl_funarg`() of Program 6.6 is not included in here. Notice that **Ch_CallFuncByAddr**() has an extra argument, `arg2`, that the function in **Ch_CallFuncByAddr**() in Programs 6.4 and Programs 6.6 do not have. This is because the Ch function pointer here points to a function that has an argument, and `arg2` has to be passed to the Ch function when it is called by **Ch_ CallFuncByAddr**().

```
#include<ch.h>
#include<stdio.h>
/* ... */

/* function pointer type which points the function with argument */
typedef void (*funcHandle)(data_type2);

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static void setFunction3_chdl_funarg(data_type2 arg2);

/* save the function pointer from the Ch space */
static void *setFunction3_chdl_funptr;

EXPORTCH void setFunction3_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get and save Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction3_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction3_chdl_funarg;
  }

  setFunction3(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static void setFunction3_chdl_funarg(data_type2 arg2) {
  /* Call Ch callback function by its address */
  Ch_CallFuncByAddr(interp, setFunction3_chdl_funptr, NULL, arg2);
}
```

Program 6.8: Argument of pointer to function with argument (chdl function).

6.1.4 Pointer to Function with Both Return Value and Arguments

This section is a summary of previous two sections. Assume that the function to be handled in this section is:

```
void setFunction4(data_type1 arg1,
                  return_type (*funptr)(data_type2 arg2)) {
 ...
}
```

Function `setFunction4()` has two arguments and no return value. The first argument `arg1` is of simple data type `data_type1`. The second argument `funptr` is a pointer to function that has both argument and return value. The argument `arg2` is of simple data type `data_type2`. The return value has data type `return_type`. In fact, `setFunction4()` is a combination of `setFunction2()` and `setFunction3()`. Variable `funptr` points to a function that has a return value and an argument. Therefore, we can find some methods that have been used earlier in the chdl and chf functions.

```
#include<dlfcn.h>

void setFunction4(data_type1 arg1, return_type(* funptr)(data_type2)) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "setFunction4_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* to call the chdl function in dynamically loaded library by address;
      arguments including function handle are passed C space */
  dlrunfun(fptr, NULL, setFunction4, arg1, funptr);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

Program 6.9: Argument of pointer to function with return value and argument (chf file).

Program 6.9 is the chf file for this example. The return value of function is of data type `return_type` and the argument is of data type `data_type2`. This is the only difference between Program 6.9 and Ch functions in other examples.

```
#include<ch.h>
#include<stdio.h>
/* ... */

/* define function pointer type with argument of data_type2
   and return value of return_type */
typedef return_type (*funcHandle)(data_type2);

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static return_type setFunction4_chdl_funarg(data_type2 arg2);

/* save the function pointer from the Ch space */
static void *setFunction4_chdl_funptr;

EXPORTCH void setFunction4_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get and save function pointer from Ch space  */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction4_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction4_chdl_funarg;
  }

  setFunction4(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static return_type setFunction4_chdl_funarg(data_type2 arg2) {
  return_type retval;

  /* Call Ch function by its address which has argument arg2,
     and return a value */
  Ch_CallFuncByAddr(interp, setFunction4_chdl_funptr, &retval, arg2);

  return retval;
}
```

Program 6.10: Argument of pointer to function with return value and argument (chdl function).

The chdl function `setFunction4_chdl`() is shown as Program 6.10. The typedef statement in this file should be identical to the one that is in the chf file. In Program 6.6, `funcHandle` has a return value of data type `return_type`. In Program 6.8, `funcHandle` has an argument of data type `data_type2`. Here, `funcHandle` has both an argument and a return value.

In the function `setFunction4_chdl_funarg`(), the third and fourth arguments in the function call below

```
Ch_CallFuncByAddr(interp, setFunction3_chdl_funptr, &retval, arg2);
```

are `&retval` and `arg2`, since the function that Ch function pointer pointing to has a return value and an argument. As it has been mentioned before, prototype of C function `setFunction4_chdl_funarg()` should be consistent with that of the Ch function `setFunction4()`.

Example

This is a complete example that illustrates the case of setting pointer to function that has return value and two arguments.

Listing 1 — Ch function file (`setFunction4.chf`)

```
#include<dlfcn.h>
void setFunction4(int arg1, int (*funptr)(float, int)) {
  void *dlhandle, *fptr;

  dlhandle = dlopen("libsetfuncptr4.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "setFunction4_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address,
     arguments including function pointer are passed to C space */
  dlrunfun(fptr, NULL, setFunction4, arg1, funptr);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }
}
```

The above Chf function is almost the same as the template shown in Program 6.9, except for the DLL file `libsetfuncptr4.dl`.

Listing 2 — chdl file (`setFunction4_chdl.c`)

```
#include <stdio.h>
#include <ch.h>

typedef int (*funcHandle)(float, int); /* function pointer type */

/* C functions to replace the Ch one */
static ChInterp_t interp;
static int setFunction4_chdl_funarg(float, int);

/* function pointers to save Ch function pointers */
static void *setFunction4_chdl_funptr;

EXPORTCH void setFunction4_chdl(void *varg){
  ChVaList_t ap;
```

```
  int arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, int);
  handle_ch = Ch_VaArg(interp, ap, funcHandle); /* get ch function pointer */
  setFunction4_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction4_chdl_funarg;
  }

  setFunction4(arg1, handle_c);
  Ch_VaEnd(interp, ap);
}

static int setFunction4_chdl_funarg(float f, int i) {
  int retval;
  Ch_CallFuncByAddr(interp, setFunction4_chdl_funptr, &retval, f, i);
  return retval;
}
```

Listing 3 — C function to be handled (`setFunction4.c`)

```
#include <stdio.h>
void setFunction4(int i, int (*f)(float, int)) {
    int retval;
    if ( f != NULL) {
        retval = f(1.1, i);
        printf("in C function setFunction4(), retval = %d\n", retval);
    }
}
```

Listing 4 — Makefile to build DLL (`Makefile`)

```
# for cases pointer to function

INC1=-I/usr/include
INC2=-I/usr/ch/extern/include

target: libsetfuncptr4.dl

libsetfuncptr4.dl: setFunction4.o setFunction4_chdl.o
        ch dllink libsetfuncptr4.dl \
                  setFunction4.o setFunction4_chdl.o

setFunction4.o: setFunction4.c
        ch dlcomp libsetfuncptr4.dl setFunction4.c $(INC1) $(INC2)

setFunction4_chdl.o: setFunction4_chdl.c
        ch dlcomp libsetfuncptr4.dl setFunction4_chdl.c $(INC1) $(INC2)

clear:
        rm -f  *.o
clean:
        rm -f  *.o
        rm -f  lib*.dl
```

Listing 5 — Ch Application (`setfuncptr4.ch`)

```
#include <stdio.h>

int func(float f, int i) {
  printf("In Ch function func(), f = %f, i = %d\n", f, i);
  return 100;
}

int main() {
  setFunction4(10, func);
  return 0;
}
```

After making the DLL file, the result from direct execution of application `setfuncptr4.c` is

```
In Ch function func(), f = 1.100000, i = 10
in C function setFunction4(), retval = 100
```

6.1.5 Pointer to Struct with a Field of Pointer to Function

Functions with return value of struct type has been discussed in Section 5.6. Functions with the argument of struct or pointer to struct type without the field of pointer to function can be handled as a simple type, it has been discussed in Section 5.3. If a structure has member fields of pointer to functions and functions pointed by these member fieldds are in binary space, then the pointer to structure can be treated the same as in Section 5.3. In this case, functions pointed by pointer to functions might be set by other functions.

In this section, we will discuss functions with arguments of structs that have fields of pointers to functions and these functions are in the Ch script space. Assume that we have the following function in C space:

```
struct s_t {
   data_type mem;
   return_type1 (*funptr1)(data_type1 arg1);
   return_type2 (*funptr2)(data_type2 arg2);
};
return_type3 setFunction6(data_type3 arg3, struct s_t *arg4) {
  ...
}
```

struct `s_t` has three members. Member `mem` is of simple data type, whereas members `funptr1` and `funptr2` are of pointers to functions which have arguments and return values.

Function `setFunction6`() takes two arguments and returns a value of type `return_type3`. The argument `arg3` is of date type `data_type3`, and `arg4` is a pointer to struct `s_t`.

The Ch function `setFunction6`() is given in Program 6.11. Since `setFunction6` returns a value, the method introduced in section 5.3 can also be applied to this case. Function **dlrunfun()**'s second argument is `&retval` instead of `NULL`. It is shown as follows.

```
dlrunfun(fptr, &retval, setFunction6, arg3, arg4);
```

At the same time, the variable `retval` is returned at the end of the chf function. The argument `arg4`, a pointer to struct `s_t`, is passed as the fifth argument of the function **dlrunfun()**. If the DLL cannot be loaded, or the chdl function can not be found in DLL , the value `FAIL_VALUE` will be returned. `FAIL_VALUE` is a value of type `return_type3`, such as NULL for point type and -1 for integral type, to indicate the failure of the function calling.

```
#include<dlfcn.h>

return_type3 setFunction6(data_type3 arg3, struct s_t *arg4) {
  void *dlhandle, *fptr;
  return_type3 retval;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }
  fptr = dlsym(dlhandle, "setFunction6_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }

  /* call the chdl function in dynamically loaded library by address,
     arguments including pointer to the struct are passed */
  dlrunfun(fptr, &retval, setFunction6, arg3, arg4);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }
  return retval;
}
```

Program 6.11: Argument of struct with members of pointer to function (chf file).

```
#include<ch.h>
#include<stdio.h>

static ChInterp_t interp;
/* C functions to replace the Ch function pointers */
static return_type1 setFunction6_chdl_funarg1(data_type1 arg1);
static return_type2 setFunction6_chdl_funarg2(data_type2 arg2);

/* save the Ch function pointers */
static void *setFunction6_chdl_funptr1, *setFunction6_chdl_funptr2;

EXPORTCH return_type3 setFunction6_chdl(void *varg) {
  ChVaList_t ap;
  data_type3 arg3;
  struct s_t *arg4;
  return_type3 retval;

  Ch_VaStart(interp, ap, varg);
  arg3 = Ch_VaArg(interp, ap, data_type3);   /* get 1st argument  */

  arg4 = Ch_VaArg(interp, ap, struct s_t *); /* get pointer to a struct from Ch space */
  if(arg4 != NULL) { /* save the Ch function pointer */
    setFunction6_chdl_funptr1 = (void*)arg4->funptr1; // used for callback
    if( arg4->funptr1 != NULL) {
      arg4->funptr1 = setFunction6_chdl_funarg1;
    }

    setFunction6_chdl_funptr2 = (void*)arg4->funptr2; // used for callback
    if( arg4->funptr2 != NULL) {
       arg4->funptr2 = setFunction6_chdl_funarg2;
    }
  }
  retval = setFunction6(arg3, arg4);

  if(arg4 != NULL) { /* restore the original pointers */
    arg4->funptr1 = setFunction6_chdl_funptr1;
    arg4->funptr2 = setFunction6_chdl_funptr2;
  }
  Ch_VaEnd(interp, ap);
  return retval;
}

/* C function to replace Ch function pointers as struct members */
static return_type1 setFunction6_chdl_funarg1(data_type1 arg1) {
  return_type1 retval;
  Ch_CallFuncByAddr(interp, setFunction6_chdl_funptr1, &retval, arg1);
  return retval;
}
static return_type2 setFunction6_chdl_funarg2(data_type2 arg2) {
  return_type2 retval;
  Ch_CallFuncByAddr(interp, setFunction6_chdl_funptr2, &retval, arg2);
  return retval;
}
```

Program 6.12: Argument of struct with members of pointer to functions (chdl function).

The chdl function `setFunction6_chdl`() for this example is shown in Program 6.12. Argument `arg4` is a pointer to struct `s_t`, in which members `funptr1` and `funptr2` are pointers to functions in Ch address space. After the statements

```
setFunction6_chdl_funptr1 = (void*)arg4->funptr1;
setFunction6_chdl_funptr2 = (void*)arg4->funptr2;
```

save these two members into static pointers `setFunction6_chdl_funptr1` and `setFunction6_chdl_funptr2`, we assign two C function pointers to members of `arg4` as follows to replace Ch function pointers if they are not NULL pointers.

```
setFunction6_chdl_funptr1 = (void*)arg4->funptr1;
if(arg4->funptr1 != NULL) {
  arg4->funptr1 = setFunction6_chdl_funarg1;
}
...
```

Then the C function `setFunction6`() is called with arguments `arg3` and `arg4` with the statement

```
retval = setFunction6(arg3, arg4);
```

where the members of argument `arg4` is C function pointers instead of Ch function pointers. This is a key point of this case.

Since it is the address of the struct rather than the copy of content of the struct that is passed from Ch space, The assignments of struct member in C space also change the content of the struct in Ch space. Therefore, restoring the original content of the struct with the following two statements before the function is returned is also one of the key points of this case.

```
arg4->funptr1 = setFunction6_chdl_funptr1;
arg4->funptr2 = setFunction6_chdl_funptr2;
```

6.1.6 Arguments of Array of Function

In this section, we will discuss functions with arguments of array of function. Assume that we have a C function with following prototype,

```
return_type setarrayfunc(int n, return_type2 (*func[])(data_type arg));
```

The second argument `func` is an array of function which takes one argument `arg` and returns a value with type `return_type2`. The length of this array is variable and can be determined at runtime. Typically, the length can be passed as another argument, for example, `n` in this prototype.

The way we handle this function is similar to the one described in the previous section. The corresponding Ch function is shown in Program 6.13.

```
#include <dlfcn.h>
return_type setarrayfunc(int n, return_type2 (*func[])(data_type arg)) {
  void *dlhandle, *fptr;
  return_type retval;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }
  fptr = dlsym(dlhandle, "setarrayfunc_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return FAIL_VALUE;
  }

  dlrunfun(fptr, &retval, setarrayfunc, n, func);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }
  return retval;
}
```

Program 6.13: Arguments of array of function (chf file).

There is nothing need to be specially handled in this function. The array of function `func` is passed to the C space like a simple data type.

The chdl function is shown in Program 6.14.

```
#include <ch.h>
#include <stdio.h>

#define MAXNUM 100 /* maximum length of the array of function */

/* type of pointer to function array */
typedef return_type2 (**pfuncArr_t)(data_type arg);

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static return_type2 (*setarrayfunc_chdl_funarg[MAXNUM])(data_type arg);

/* save the function pointer from the Ch space */
static return_type2 (*setarrayfunc_chdl_funptr[MAXNUM])(data_type arg);

static return_type2 setarrayfunc_chdl_funarg0(data_type arg);
static return_type2 setarrayfunc_chdl_funarg1(data_type arg);
static return_type2 setarrayfunc_chdl_funarg2(data_type arg);
/* ... up to MAXNUM */

EXPORTCH return_type setarrayfunc_chdl(void *varg) {
  ChVaList_t ap;
  return_type retval;
  int i, n;
  pfuncArr_t parr_ch, parr_c = NULL;

  Ch_VaStart(interp, ap, varg);
  n = Ch_VaArg(interp, ap, int);

  /* get Ch array */
  parr_ch = Ch_VaArg(interp, ap, pfuncArr);
  if(parr_ch != NULL) {
    /* save Ch array */
    for(i = 0; i < n; i++) {
      setarrayfunc_chdl_funptr[i] = (return_type2 (*)(data_type))parr_ch[i];
    }

    /* assign C function pointer to the C array */
    setarrayfunc_chdl_funarg[0] = setarrayfunc_chdl_funarg0;
    setarrayfunc_chdl_funarg[1] = setarrayfunc_chdl_funarg1;
    setarrayfunc_chdl_funarg[2] = setarrayfunc_chdl_funarg2;
    /* ... up to MAXNUM */

    /* replace the Ch array with the C one,
       the NULL pointer can't be replaced */
    parr_c = (pfuncArr)setarrayfunc_chdl_funarg;
  }

  /* pass the C array with members of pointers to C functions
     instead of the Ch one, the NULL pointer will not be replaced */
  retval = setarrayfunc(n, parr_c);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 6.14: Arguments of array of function (chdl file).

```
static return_type2 setarrayfunc_chdl_funarg0(data_type arg) {
  return_type2 retval;

  Ch_CallFuncByAddr(interp, (void*)setarrayfunc_chdl_funptr[0], &retval, arg);
  return retval;
}

static return_type2 setarrayfunc_chdl_funarg1(data_type arg) {
  return_type2 retval;

  Ch_CallFuncByAddr(interp, (void*)setarrayfunc_chdl_funptr[1], &retval, arg);
  return retval;
}

static return_type2 setarrayfunc_chdl_funarg2(data_type arg) {
  return_type2 retval;

  Ch_CallFuncByAddr(interp, (void*)setarrayfunc_chdl_funptr[2], &retval, arg);
  return retval;
}

/* ... up to MAXNUM */
```

Program 6.15: Arguments of array of function (chdl file) (Contd.).

The macro `MAXNUM` indicates the maximum length of the array of function. Data type `pfuncArr_t` is defined as a pointer to pointer to function. Static variable `setarrayfunc_chdl_funarg`, which is declared as an array of function, will be used later to replace the Ch array. C functions `setarrayfunc_chdl_funarg1`(), `setarrayfunc_chdl_funarg2`(), and so on, have the same prototype as members of the Ch array of function. After the Ch array is retrieved from the argument list, we will assign these C functions to the C array `setarrayfunc_chdl_funarg`. Then, this C array will be passed to the C function `setarrayfunc`() instead of the Ch array.

Example

In main() function in application `prog.ch` of this example, the array of function `f` with three elements, which are user-defined Ch functions `f0`(), `f1`(), and `f2`(), is going to be passed to the C space by the function `setarrayfunc`().

Listing 1 — Application (`prog.ch`)

```
#include <stdarg.h>

int setarrayfunc(int n, int (*f[])(int num));

int f0(int num) {
  printf("f0() is called, num = %d\n", num);
  return 0;
}

int f1(int num) {
  printf("f1() is called, num = %d\n", num);
  return 0;
}
```

```
int f2(int num) {
  printf("f2() is called, num = %d\n", num);
  return 0;
}

int main() {
  int (*f[3])(int num);
  f[0] = f0;
  f[1] = f1;
  f[2] = f2;

  setarrayfunc(3, f);
  return 0;
}
```

Listing 2 — Ch function file (`setarrayfunc.chf`)

```
#include <dlfcn.h>
int setarrayfunc(int n, int(*f[])(int num)) {
  void *dlhandle, *fptr;
  int retval;

  dlhandle = dlopen("libarrayfunc.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return -1;
  }

  fptr = dlsym(dlhandle, "setarrayfunc_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, setarrayfunc, n, f);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return -1;
  }
  return retval;
}
```

Listing 3 — chdl file (`setarrayfunc_chdl.c`)

```
#include <ch.h>
#include <stdio.h>

#define MAXNUM 3 /* maximum length of the array of function */

typedef int (**pfuncArr_t)(int num); /* function pointer type */
static ChInterp_t interp;
static int (*setarrayfunc_chdl_funarg[MAXNUM])(int num);
static int (*setarrayfunc_chdl_funptr[MAXNUM])(int num);

static int setarrayfunc_chdl_funarg0(int num);
static int setarrayfunc_chdl_funarg1(int num);
static int setarrayfunc_chdl_funarg2(int num);
```

```
EXPORTCH int setarrayfunc_chdl(void *varg) {
  ChVaList_t ap;
  int retval;
  int i, n;
  pfuncArr_t parr_ch, parr_c = NULL;

  Ch_VaStart(interp, ap, varg);
  n = Ch_VaArg(interp, ap, int);

  /* get Ch array */
  parr_ch = Ch_VaArg(interp, ap, pfuncArr_t);
  if(parr_ch != NULL) {
    /* save Ch array */
    for(i = 0; i < n; i++) {
      setarrayfunc_chdl_funptr[i] = (int (*)(int))parr_ch[i];
    }

    /* assign C function to the C array */
    setarrayfunc_chdl_funarg[0] = setarrayfunc_chdl_funarg0;
    setarrayfunc_chdl_funarg[1] = setarrayfunc_chdl_funarg1;
    setarrayfunc_chdl_funarg[2] = setarrayfunc_chdl_funarg2;
    /* ... up to MAXNUM */

    /* replace the Ch array with the C one,
       the NULL pointer can't be replaced */
    parr_c = (pfuncArr_t)setarrayfunc_chdl_funarg;
  }

  /* pass the C array with members of pointers to C functions
     instead of the Ch one, the NULL pointer will not be replaced */
  retval = setarrayfunc(n, parr_c);

  Ch_VaEnd(interp, ap);
  return retval;
}

static int setarrayfunc_chdl_funarg0(int num) {
  int retval;

  Ch_CallFuncByAddr(interp, (void*)setarrayfunc_chdl_funptr[0], &retval, num);
  return retval;
}

static int setarrayfunc_chdl_funarg1(int num) {
  int retval;

  Ch_CallFuncByAddr(interp, (void*)setarrayfunc_chdl_funptr[1], &retval, num);
  return retval;
}

static int setarrayfunc_chdl_funarg2(int num) {
  int retval;

  Ch_CallFuncByAddr(interp, (void*)setarrayfunc_chdl_funptr[2], &retval, num);
  return retval;
}
```

Listing 4 — C functions to be handled (`setarrayfunc.c`)

```
int setarrayfunc(int n, int (*f[])(int num)) {
```

```
  int i;

  for(i=0; i<n; i++) { /* call functions in the array */
    f[i](i);
  }

  return 0;
}
```

Listing 5 — Makefile (`Makefile`)

```
target: libarrayfunc.dl
libarrayfunc.dl: setarrayfunc_chdl.o setarrayfunc.o
        ch dllink libarrayfunc.dl setarrayfunc_chdl.o setarrayfunc.o

setarrayfunc_chdl.o: setarrayfunc_chdl.c
        ch dlcomp libarrayfunc.dl setarrayfunc_chdl.c

setarrayfunc.o: setarrayfunc.c
        ch dlcomp libarrayfunc.dl setarrayfunc.c

clear:
        rm -f *.o *.dl
```

Listing 6 — Output

```
f0() is called, num = 0
f1() is called, num = 1
f2() is called, num = 2
```

6.1.7 Pointer to Function with Variable Number of Arguments

In this section, functions with argument of pointer which points to function with variable number of arguments will be discussed. To give the template, assume we have the general function prototype as follows

```
return_type2 (*FUNC)(int num, ...);
return_type setFuncVNA(FUNC pf);
```

where function `setFuncVNA` has an argument of pointer with type `FUNC`, and `FUNC` is type defined as a pointer to function which has variable number of arguments. Typically, the first argument `num` of `FUNC` is the information of number of the following argument.

The corresponding Ch function is shown in Program 6.16.

```
#include <dlfcn.h>
return_type setFuncVNA(return_type2 (*pf)(int num, ...)) {
  void *dlhandle, *fptr;
  return_type retval;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  fptr = dlsym(dlhandle, "setFuncVNA_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return FAIL_VALUE;
  }

  dlrunfun(fptr, &retval, setFuncVNA, pf);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return FAIL_VALUE;
  }
  return retval;
}
```

Program 6.16: Argument of function with variable number of arguments (chf file).

There is nothing need to be specially handled in this function. The function pointer `pf` is passed to the C space like a simple data type.

The chdl function is shown in Program 6.17.

```
#include <ch.h>
#include <stdio.h>
#include <limits.h>

typedef return_type2 (*FUNC)(int num, ...); /* function pointer type */
static ChInterp_t interp;
static void *func_chdl_funptr;
static return_type2 func_chdl_funarg(int num, ...) {
    return_type2 retval;
    data_type1 arg1;
    data_type2 arg2;
    /* ... other possible arguments in the variable length argument list */
    ChVaList_t ap_ch, ap;

    va_start(ap, num);
    ap_ch = Ch_VarArgsCreate(interp);
    if(num>=1) {
      arg1 = va_arg(ap, data_type1);
      Ch_VarArgsAddArg(interp, &ap_ch, CH_TYPE_1, arg1); /* CH_TYPE_1 is one of macros
                                     corresponding to data_type1 and
                                     defined in ch.h,
                                     such as CH_INTTYPE to int */
    }
    if(num>=2) {
      arg2 = va_arg(ap, data_type2);
      Ch_VarArgsAddArg(interp, &ap_ch, CH_TYPE_2, arg2);
    }
    /* ... for other possible number of argument */

    Ch_CallFuncByAddr(interp, func_chdl_funptr, &retval, num, ap_ch);
    Ch_VarArgsDelete(interp, ap_ch);
    return retval;
}

EXPORTCH  return_type setFuncVNA_chdl(void *varg)
{
   ChVaList_t ap;
   return_type retval;

   Ch_VaStart(interp, ap, varg);
   func_chdl_funptr = Ch_VaArg(interp, ap, void *);
   if(func_chdl_funptr != NULL) {
      retval = setFuncVNA(func_chdl_funarg);
   }
   else {
      retval = setFuncVNA(NULL);
   }

   Ch_VaEnd(interp, ap);
   return retval;
}
```

Program 6.17: Argument of function with variable number of arguments (chdl file).

APIs **Ch_VarArgsCreate**(), **Ch_VarArgsAddArg**() and **Ch_VarArgsDelete**() are used to create a Ch argument list. For each argument in the C argument list `ap` in every possible case, we need to write a statement shown below to retrieve it from the C argument list and add to the Ch argument list `ap_ch`.

```
if(num>=1) {
  arg1 = va_arg(ap, data_type1);
  Ch_VarArgsAddArg(interp, &ap_ch, CH_TYPE_1, j);
             /* CH_TYPE_1 is one of macros corresponding to data_type1
                and defined in ch.h, such as CH_INTTYPE to int */
}
```

Example

In this example, user-defined Ch functions `func`() is going to be passed to the C space by the function `func1`(), and then this Ch function is called from C space with 0, 1, 2 arguments, respectively.

Listing 1 — Application and Ch function (`prog1.ch`)

```
#include <dlfcn.h>
#include <stdarg.h>

typedef int (*FUNC)(int num, ...);

int f(int num, ...) { // function to be set by func1()
  int vacount, val_i, i;
  float val_f;
  double val_d;
  va_list ap;

  va_start(ap, num);
  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);
  printf("num = %d\n", num);
  for(i=0; i<vacount; i++) {
     if(va_elementtype(ap) == elementtype(int)) {
       val_i = va_arg(ap, int);
       printf("val_i = %d\n", val_i);
     }
     else if (va_elementtype(ap) == elementtype(float)) {
       val_f = va_arg(ap, float);
       printf("val_f = %f\n", val_f);
     }
     else if (va_elementtype(ap) == elementtype(double)) {
       val_d = va_arg(ap, double);
       printf("val_f = %f\n", val_d);
     }
  }
  printf("\n");
  return 0;
}

int func1(FUNC f) {
   void *handle, *fptr;
   int retval;

   handle = dlopen("libprog.dl", RTLD_LAZY);
   if(handle == NULL) {
```

```
      printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
      return -1;
   }
   fptr = dlsym(handle, "func1_chdl");
   if(fptr == NULL) {
      printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
      return -1;
   }
   dlrunfun(fptr, &retval, func1, f);
   return retval;
}

int main() {
   func1(f);
   return 0;
}
```

Listing 2 — chdl file (`prog1_chdl.c`)

```
#include <ch.h>
#include <stdio.h>
#include <limits.h>

typedef int (*FUNC)(int num, ...);
extern int func1(FUNC);

static ChInterp_t interp;
static void *func_chdl_funptr;
static int func_chdl_funarg(int num, ...) {
    int retval;
    int j;
    float f;
    double d;
    ChVaList_t ap_ch, ap;

    va_start(ap, num);
    ap_ch = Ch_VarArgsCreate(interp);
    if(num>=1) {
      j = va_arg(ap, int);
      Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, j);
    }
    if(num>=2) {
      d = va_arg(ap, double);
      Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
    }
    Ch_CallFuncByAddr(interp, func_chdl_funptr, &retval, num, ap_ch);
    Ch_VarArgsDelete(interp, ap_ch);
    return retval;
}

EXPORTCH int func1_chdl(void *varg)
{
   ChVaList_t ap;
   int count, num;
   int retval;

   Ch_VaStart(interp, ap, varg);
   count = Ch_VaCount(interp, ap);
   printf("Total args passed to func1_chdl() in C code = %d\n", count);
```

```
   num = Ch_VaFuncArgNum(interp, ap);
   if(num == INT_MAX)
     printf("Passed pointer to func in func1_chdl has variable number arg\n");
   else
     printf("Passed pointer to func in func1_chdl has %d arg\n", num);
   func_chdl_funptr = Ch_VaArg(interp, ap, void *);

   if(func_chdl_funptr != NULL) {
      retval = func1(func_chdl_funarg);
   }
   else {
      printf("passed fun argument to func1_chdl() is NULL\n");
   }

   Ch_VaEnd(interp, ap);
   return retval;
}
```

Listing 3 — C functions to be handled (`prog1.c`)

```
#include <stdio.h>

typedef int (*FUNC)(int num, ...);
int func1(FUNC fp) {
   int retval;
   int j = 10;
   double d=30;

   retval = fp(0);
   retval = fp(1, j);
   retval = fp(2, j, d);
}
```

Listing 4 — Makefile (`Makefile`)

```
target: libprog.dl

libprog.dl: prog1.o prog1_chdl.o \
            prog2.o prog2_chdl.o \
            prog3_chdl.o
ch dllink libprog.dl prog1.o prog1_chdl.o \
                             prog2.o prog2_chdl.o \
                             prog3_chdl.o

prog1.o: prog1.c
ch dlcomp libprog.dl prog1.c
prog1_chdl.o: prog1_chdl.c
ch dlcomp libprog.dl prog1_chdl.c
prog2.o: prog2.c
ch dlcomp libprog.dl prog2.c
prog2_chdl.o: prog2_chdl.c
ch dlcomp libprog.dl prog2_chdl.c
prog3_chdl.o: prog3_chdl.c
ch dlcomp libprog.dl prog3_chdl.c

clean:
rm -f *.o *.obj libprog.dl libprog.exp libprog.lib
```

Listing 5 — Output

```
Total args passed to func1_chdl() in C code = 1
Passed pointer to func in func1_chdl has varaible number arg
vacount = 0
num = 0

vacount = 1
num = 1
val_i = 10

vacount = 2
num = 2
val_i = 10
val_f = 30.000000
```

Example

In this example, the user-defined Ch functions `vf`(), which takes an argument `ap` with type **ap_list**, is going to be passed to the C space by the function `func2`(), and then this Ch function is called from C space with argument of type `ap_list` including 0, 1, 2 arguments, respectively.

Listing 1 — Application and Ch function (`prog2.ch`)

```
#include <dlfcn.h>
#include <stdarg.h>

typedef int (*FUNC)(int num, va_list ap);
int f(int num, ...) {
  int vacount, val_i, i;
  float val_f;
  double val_d;
  va_list ap;

  va_start(ap, num);
  vf(num, ap);
  va_end(ap);
  return 0;
}

int vf(int num, va_list ap) {
  int vacount, val_i, i;
  float val_f;
  double val_d;

  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);
  printf("num = %d\n", num);
  for(i=0; i< vacount; i++) {
     if(va_elementtype(ap) == elementtype(int)) {
       val_i = va_arg(ap, int);
       printf("val_i = %d\n", val_i);
     }
     else if (va_elementtype(ap) == elementtype(float)) {
       val_f = va_arg(ap, float);
       printf("val_f = %f\n", val_f);
     }
     else if (va_elementtype(ap) == elementtype(double)) {
       val_d = va_arg(ap, double);
```

```
      printf("val_d = %f\n", val_d);
     }
     else {
       printf("warning: unknown data type\n");
    }
  }
  printf("\n");
  return 0;
}

int func2(FUNC f) {
   void *handle, *fptr;
   int retval;

   handle = dlopen("libprog.dl", RTLD_LAZY);
   if(handle == NULL) {
      printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
      return -1;
   }
   fptr = dlsym(handle, "func2_chdl");
   if(fptr == NULL) {
      printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
      return -1;
   }
   dlrunfun(fptr, &retval, func2, f);
   return retval;
}

int main() {
   func2(vf);
   return 0;
}
```

Listing 2 — chdl file (`prog2_chdl.c`)

```
#include <ch.h>
#include <stdio.h>
#include <stdarg.h>
#include <limits.h>

typedef int (*FUNC)(int num, ChVaList_t ap);
extern int func2(FUNC, ...);

static ChInterp_t interp;
static void *vfunc_chdl_funptr;
static int vfunc_chdl_funarg(int num, ChVaList_t ap) {
    int retval;
    int j;
    double d;
    ChVaList_t ap_ch;

    ap_ch = Ch_VarArgsCreate(interp);
    if(num >= 1) {
      j  = va_arg(ap, int);
      Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, j);
    }
    if(num >= 2) {
      d  = va_arg(ap, double);
      Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
    }
```

```
    Ch_CallFuncByAddr(interp, vfunc_chdl_funptr, &retval, num, ap_ch);
    Ch_VarArgsDelete(interp, ap_ch);
    return retval;
}

EXPORTCH int func2_chdl(void *varg)
{
   ChVaList_t ap;
   int count, num;
   int retval;

   Ch_VaStart(interp, ap, varg);
   count = Ch_VaCount(interp, ap);
   printf("Total args passed to func2_chdl() in C code = %d\n", count);
   num = Ch_VaFuncArgNum(interp, ap);
   if(num == INT_MAX)
     printf("Passed pointer to func in func2_chdl has varaible number arg\n");
   else
     printf("Passed pointer to func in func2_chdl has %d arg\n", num);
   vfunc_chdl_funptr = Ch_VaArg(interp, ap, void *);

   if(vfunc_chdl_funptr != NULL) {
      int num;
      int j =10;
      double d = 20;
      num = 0;
      retval = func2(vfunc_chdl_funarg, num);
      num = 1;
      retval = func2(vfunc_chdl_funarg, num, j);
      num = 2;
      retval = func2(vfunc_chdl_funarg, num, j, d);
   }
   else {
      printf("passed fun argument to func2_chdl() is NULL\n");
   }

   Ch_VaEnd(interp, ap);
   return retval;
}
```

Listing 3 — C functions to be handled (`prog2.c`)

```
#include <stdio.h>
#include <stdarg.h>

typedef int (*FUNC)(int num, va_list ap);
int func2(FUNC fp, ...) {
   va_list ap;
   int retval;
   int num;

   va_start(ap, fp);
   num = va_arg(ap, int);
   retval = fp(num, ap);
   va_end(ap);
   return 0;
}
```

Listing 4 — Output

```
Total args passed to func2_chdl() in C code = 1
Passed pointer to func in func2_chdl has 2 arg
vacount = 0
num = 0

vacount = 1
num = 1
val_i = 10

vacount = 2
num = 2
val_i = 10
val_d = 20.000000
```

Example

Unlike in C, Ch functions with variable number of arguments don't have to take at least one argument. In this example, the user-defined Ch functions `f`(), which only takes variable number of arguments is going to be passed to the C space by the function `func3`(), and then this Ch function is called from C space with 3 arguments. Note that C function `func3_tmp` in Listing 2 `prog3_chdl.c` is not the corresponding C function of Ch function `func3`(). They have different prototypes. So the goal of this example is only to illustrate how to invoke Ch functions with variable number of arguments from C space.

Listing 1 — Application and Ch function (`prog3.ch`)

```
#include <dlfcn.h>
#include <stdarg.h>

typedef int (*FUNC)(...);

int f(...) {
  int vacount, val_i, i;
  float val_f;
  double val_d;
  va_list ap;

  va_start(ap, VA_NOARG);
  vf(ap);
  va_end(ap);
  return 0;
}

int vf(va_list ap) {
  int vacount, val_i, i;
  float val_f;
  double val_d;

  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);
  for(i=0; i< vacount; i++) {
     if(va_elementtype(ap) == elementtype(int)) {
       val_i = va_arg(ap, int);
       printf("val_i = %d\n", val_i);
     }
     else if (va_elementtype(ap) == elementtype(float)) {
       val_f = va_arg(ap, float);
```

```
      printf("val_f = %f\n", val_f);
    }
    else if (va_elementtype(ap) == elementtype(double)) {
      val_d = va_arg(ap, double);
      printf("val_d = %f\n", val_d);
    }
  }
  return 0;
}

int func3(FUNC f) {
   void *handle, *fptr;
   int retval;

   handle = dlopen("libprog.dl", RTLD_LAZY);
   if(handle == NULL) {
      printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
      return -1;
   }
   fptr = dlsym(handle, "func3_chdl");
   if(fptr == NULL) {
      printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
      return -1;
   }
   dlrunfun(fptr, &retval, func3, f);
   return retval;
}

int main() {
   func3(f);
   return 0;
}
```

Listing 2 — chdl file (`prog3_chdl.c`)

```
#include <ch.h>
#include <stdio.h>
#include <limits.h>

static ChInterp_t interp;
static void *func_chdl_funptr;
static int func3_tmp() {
    int retval;
    int j = 10;
    float f=20;
    double d=40;
    ChVaList_t ap_ch;

    ap_ch = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, j);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_FLOATTYPE, f);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
    Ch_CallFuncByAddr(interp, func_chdl_funptr, &retval, ap_ch);
    /* Ch_CallFuncByName(interp, "func", &retval, ap_ch);   */
    Ch_VarArgsDelete(interp, ap_ch);
    return retval;
}

EXPORTCH int func3_chdl(void *varg)
```

```
{
   ChVaList_t ap;
   int i;
   int j;
   int count, num;
   int retval;

   Ch_VaStart(interp, ap, varg);
   num = Ch_VaFuncArgNum(interp, ap);
   if(num == INT_MAX)
     printf("Passed pointer to func in func2_chdl has varaible number arg\n");
   else
     printf("Passed pointer to func in func2_chdl has %d arg\n", num);
   func_chdl_funptr = Ch_VaArg(interp, ap, void *);

   if(func_chdl_funptr != NULL) {
      retval = func3_tmp();
   }
   else {
      printf("passed fun argument to func3_chdl() is NULL\n");
   }

   Ch_VaEnd(interp, ap);
   return retval;
}
```

Listing 3 — Output

```
Passed pointer to func in func2_chdl has varaible number arg
vacount = 3
val_i = 10
val_f = 20.000000
val_d = 40.000000
```

6.1.8 Pointer to Function Having Different Number of Arguments

As mentioned in Chapter 2, API **Ch_VaFuncArgNum()** gets the number of the arguments of the function which is passed as an argument. For example, if we have three C functions with following prototypes,

```
int (*f1)(int i);
int (*f2)(int i1, int i2);
int (*f3)(int i, ...);
int setDiffFunc(int num, int(*pf)())
```

and assume that the function `setDiffFunc`() can pass `f1`(), `f2`() or `f3`() as the second argument to the C space, the API **Ch_VaFuncArgNum**() then can be applied in C space to determine which function pointer is passed. As a special case, we assume the first argument of `f3`() is an integer which indicates the number of the following arguments. Up to two arguments, one is integer and another is double, might be passed after the first argument to `f3`(). To give the template, we have the general prototype as follows

```
return_type2 (*FUNC)();
return_type setDiffFunc(data_type arg1, FUNC pf);
```

where `arg1` is an argument with simple data type, and `pf` is a pointer to function. Functions returning data of type `retrun_type2` and taking any number of arguments could be the second argument in this case.

The corresponding Ch function is shown in Program 6.18.

```
#include <dlfcn.h>
return_type setDiffFunc(data_type arg1, int (*pf)()) {
  void *dlhandle, *fptr;
  int retval;

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return -1;
  }

  fptr = dlsym(dlhandle, "setDiffFunc_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, setDiffFunc, arg1, pf);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return -1;
  }
  return retval;
}
```

Program 6.18: Argument of functions taking different number of arguments (chf file).

There is nothing need to be specially handled in this function. The function pointer `pf` is passed to the C space like a simple data type.

The chdl function is shown in Program 6.19.

```
#include <ch.h>
#include <stdio.h>

typedef return_type2 (*FUNC)(); /* function pointer type */
static ChInterp_t interp;
static void *setDiffFunc_chdl_funptr;

static return_type2 setDiffFunc_chdl_funarg0() {
  return_type2 retval;

  Ch_CallFuncByAddr(interp, setDiffFunc_chdl_funptr, &retval);
  return retval;
}

static return_type2 setDiffFunc_chdl_funarg1(data_type11 num) {
  return_type2 retval;

  Ch_CallFuncByAddr(interp, setDiffFunc_chdl_funptr, &retval, num);
  return retval;
}

static return_type2 setDiffFunc_chdl_funarg2(data_type21 num1, data_type22 num2) {
  return_type2 retval;

  Ch_CallFuncByAddr(interp, setDiffFunc_chdl_funptr, &retval, num1, num2);
  return retval;
}

static return_type2 diff_arg_chdl_funarg3(int num, ...) {
  return_type2 retval;
  ChVaList_t ap, ap_ch;
  int x;
  double d;

  va_start(ap, num1);
  ap_ch = Ch_VarArgsCreate(interp);
  if(num1 > 1) {
    x = va_arg(ap, int);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, x);
  }
  if(num1 == 2) {
    d = va_arg(ap, double);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
  }
  Ch_CallFuncByAddr(interp, diff_arg_chdl_funptr, &retval, num1, ap_ch);
  Ch_VarArgsDelete(interp, ap_ch);
  return retval;
}

/* ... for other possible prototypes ... */
```

Program 6.19: Argument of functions taking different number of arguments (chdl file).

```
EXPORTCH return_type setDiffFunc_chdl(void *varg) {
  ChVaList_t ap;
  return_type retval;
  data_type arg1;
  int argnum;
  FUNC handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type);

  /* get argument number of the Ch function */
  argnum = Ch_VaFuncArgNum(interp, ap);

  /* get the Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, FUNC);

  if(handle_ch != NULL) {
    /* replace the Ch array with the proper C one,
       the NULL pointer can't be replaced */
    if(argnum == 0) {
      handle_c = setDiffFunc_chdl_funarg0;
    }
    else if(argnum == 1) {
      handle_c = setDiffFunc_chdl_funarg1;
    }
    else if(argnum == 2) {
      handle_c = setDiffFunc_chdl_funarg2;
    }
    else if(argnum == INT_MAX) {
      handle_c = (pf_t)diff_arg_chdl_funarg3;
    }
    /* ... */
    setDiffFunc_chdl_funptr = (void *)handle_ch;
  }

  /* pass the proper C function pointer
     instead of the Ch one,
     the NULL pointer will not be replaced.
     If the Ch function has more than 2 argument,
     except for INT_MAX, NULL will be passed. */
  retval = setDiffFunc(arg1, handle_c);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 6.20: Argument of functions taking different number of arguments (chdl file) (Contd.).

For each prototype of Ch function to be passed, we should write a corresponding C function with the same prototype, for example, `setDiffFunc_chdl_funarg0`(), `setDiffFunc_chdl_funarg1`(), `setDiffFunc_chdl_funarg2`(), `setDiffFunc_chdl_funarg3`(), and so on. In chdl function `setDiffFunc_chdl`(), the API **Ch_VaFuncArgNum**() is applied to obtain the number of its argument before the pointer of Ch function is taken from the argument list. Then, a C function with the proper prototype is chosen to be pass to `setDiffFunc`(). If the Ch function pointed to by `handle_ch` takes variable number of arguments, the return value of `Ch_VaFuncArgNum(ap)` is **INT_MAX**. In func-

tion `setDiffFunc_chdl_funarg3`(), we use APIs **Ch_VarArgsCreate**(), **Ch_VarArgsAddArg**() and **Ch_VarArgsDelete**() to create a Ch variable-length argument list. These APIs have been described in the previous section.

Example

In this example, user-defined Ch functions `f0`(), `f1`(), `f2`() `f3`(), as well as NULL pointer are going to be passed to the C space by the function `diff_arg`(). These three Ch functions take no argument, one argument, two argument and varible number arguments, respectively.

Listing 1 — Application (`prog.ch`)

```
#include <stdarg.h>
#include <limits.h>

typedef int (*FUNC)();
int diff_arg(int, FUNC);

int f0() {
   printf("f0() is called\n");
   return 0;
}

int f1(int arg1) {
  printf("f1() is called, argl = %d\n", arg1);
  return 0;
}

int f2(int arg1, int arg2) {
  printf("f2() is called, arg1 = %d, arg2 = %d\n", arg1, arg2);
  return 0;
}

int f3(int arg1, ...) {
   int i, vacount;
   va_list ap;

   printf("f3() is called, \narg1 = %d\n", arg1);
   va_start(ap, b);
   vacount = va_count(ap);
   for(i=0; i<vacount; i++) {
      if(va_elementtype(ap)  == elementtype(int)) {
         printf("arg%d = %d\n", i+2, va_arg(ap, int));
      }
      else if(va_elementtype(ap)  == elementtype(float)) {
         printf("arg%d = %f\n", i+2, va_arg(ap, float));
      }
      else if(va_elementtype(ap)  == elementtype(double)) {
         printf("arg%d = %f\n", i+2, va_arg(ap, double));
      }
   }
   return 0;
}



int main() {
```

```
  diff_arg(0, f0);
  diff_arg(0, NULL);
  diff_arg(1, (FUNC)f1);
  diff_arg(2, (FUNC)f2);
  diff_arg(INT_MAX, (FUNC)f3);
  return 0;
}
```

Listing 2 — Ch function file (`diff_arg.chf`)

```
#include <dlfcn.h>
int diff_arg(int arg1, int (*pf)()) {
  void *dlhandle, *fptr;
  int retval;

  dlhandle = dlopen("libdiffarg.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return -1;
  }

  fptr = dlsym(dlhandle, "diff_arg_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, diff_arg, arg1, pf);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return -1;
  }
  return retval;
}
```

Listing 3 — chdl file (`diff_arg_chdl.c`)

```
#include <ch.h>
#include <stdio.h>
#include <limits.h>
#include <stdarg.h>

typedef int (*pf_t)(); /* function pointer type */
static ChInterp_t interp;
static void *diff_arg_chdl_funptr;

static int diff_arg_chdl_funarg0() {
  int retval;

  Ch_CallFuncByAddr(interp, diff_arg_chdl_funptr, &retval);
  return retval;
}

static int diff_arg_chdl_funarg1(int num) {
  int retval;

  Ch_CallFuncByAddr(interp, diff_arg_chdl_funptr, &retval, num);
  return retval;
}
```

```
static int diff_arg_chdl_funarg2(int num1, int num2) {
  int retval;

  Ch_CallFuncByAddr(interp, diff_arg_chdl_funptr, &retval, num1, num2);
  return retval;
}

static int diff_arg_chdl_funarg3(int num1, ...) {
  int retval;
  va_list ap;
  ChVaList_t ap_ch;
  int x;
  double d;

  va_start(ap, num1);
  ap_ch = Ch_VarArgsCreate(interp);
  if(num1 >= 1) {
    x = va_arg(ap, int);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, x);
  }
  if(num1 == 2) {
    d = va_arg(ap, double);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
  }
  Ch_CallFuncByAddr(interp, diff_arg_chdl_funptr, &retval, num1, ap_ch);
  Ch_VarArgsDelete(interp, ap_ch);
  return retval;
}

EXPORTCH int diff_arg_chdl(void *varg) {
  ChVaList_t ap;
  int retval;
  int arg1, argnum;
  pf_t handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, int);

  /* get argument number of the Ch function */
  argnum = Ch_VaFuncArgNum(interp, ap);
  if(argnum == INT_MAX)
     printf("\npassed varaible number arg\n");
  else
     printf("\nargnum = %d\n", argnum);
  /* get the Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, pf_t);

  if(handle_ch != NULL) {
    /* replace the Ch array with the proper C one,
       the NULL pointer can't be replaced */
    if(argnum == 0) {
      handle_c = diff_arg_chdl_funarg0;
    }
    else if(argnum == 1) {
      handle_c = diff_arg_chdl_funarg1;
    }
    else if(argnum == 2) {
      handle_c = diff_arg_chdl_funarg2;
```

```
    }
    else if(argnum == INT_MAX) {
      handle_c = (pf_t)diff_arg_chdl_funarg3;
    }
    diff_arg_chdl_funptr = (void *)handle_ch;
  }

  /* pass the proper C function pointer
     instead of the Ch one,
     the NULL pointer will not be replaced.
     If the Ch function has more than 2 argument,
     except for INT_MAX, NULL will be passed. */
  retval = diff_arg(arg1, handle_c);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 4 — C functions to be handled (`diff_arg.c`)

```
#include <stdio.h>
#include <limits.h>

int diff_arg(int num, int (*pf)()) {
  if(pf == NULL) {
     printf("NULL for pf is passed to diff_arg() in C\n");
     return 0;
  }
  switch (num) {
    case 0:
      pf();
      break;
    case 1:
      pf(10);
      break;
    case 2:
      pf(10, 20);
      break;
    case INT_MAX:
      pf(0);
      pf(1, 20);
      pf(2, 20, 30.0);
      break;
  }

  return 0;
}
```

Listing 5 — Makefile (`Makefile`)

```
target: libdiffarg.dl
libdiffarg.dl: diff_arg_chdl.o diff_arg.o
        ch dllink libdiffarg.dl diff_arg_chdl.o diff_arg.o

diff_arg_chdl.o: diff_arg_chdl.c
        ch dlcomp libdiffarg.dl diff_arg_chdl.c

diff_arg.o: diff_arg.c
        ch dlcomp libdiffarg.dl diff_arg.c
```

```
clear:
        rm -f *.o *.dl
```

Listing 6 — Output

```
argnum = 0
f0() is called

argnum = 0
NULL for pf is passed to diff_arg() in C

argnum = 1
f1() is called, argl = 10

argnum = 2
f2() is called, arg1 = 10, arg2 = 20

passed varaible number arg
f3() is called,
arg1 = 0
f3() is called,
arg1 = 1
arg2 = 20
f3() is called,
arg1 = 2
arg2 = 20
arg3 = 30.000000
```

6.2 Functions with Return Value of Pointer to Function

To get some ideas of how functions with return value of pointer to function work, let's take a look at the following simple example.

Listing 1 — header file for Ch functions (`ptfun.h`)

```
#ifndef _PTFUN_H_
#define _PTFUN_H_

typedef void(*funcHandle)();
void ptfun_setHandle(funcHandle funptr);
funcHandle ptfun_getHandle();

/* below is added only for Ch */
#include <dlfcn.h>
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
}

/* user-defined function which is called to close DLL
   when the program exits */
void _dlclose_ptfun(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun);
```

```
#pragma importf <ptfun.chf>
#endif /* _PTFUN_H_ */
```

Listing 2 — Ch function file (`ptfun.chf`)

```
#include <dlfcn.h>

void ptfun_setHandle(funcHandle funptr)  {
  void *fptr;

  fptr = dlsym(_Chptfun_handle, "ptfun_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, ptfun_setHandle, funptr);
  return;
}

funcHandle ptfun_getHandle()  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, ptfun_getHandle);
  return retval;
}
```

Listing 3 — header file for C functions (`ptfun_c.h`)

```
#ifndef _PTFUN_H_
#define _PTFUN_H_

typedef void (*funcHandle)();
void ptfun_setHandle(funcHandle funptr);
funcHandle ptfun_getHandle();

#endif /* _PTFUN_H_ */
```

Listing 4 — chdl file (`ptfun_chdl.c`)

```
#include <stdio.h>
#include "ptfun_c.h"
#include <ch.h>

static ChInterp_t interp;
static void ptfun_chdl_funarg();
static void *ptfun_chdl_funptr;

EXPORTCH void ptfun_setHandle_chdl(void *varg){
  ChVaList_t ap;
  funcHandle handle_ch;
```

```
  Ch_VaStart(interp, ap, varg);
  /* get the Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  ptfun_setHandle(handle_ch);
  Ch_VaEnd(interp, ap);
}

EXPORTCH funcHandle ptfun_getHandle_chdl(void *varg){
  ChVaList_t ap;
  funcHandle retval_ch;

  Ch_VaStart(interp, ap, varg);
  retval_ch = ptfun_getHandle();
  Ch_VaEnd(interp, ap);

  return retval_ch;
}
```

Listing 5 — C functions to be handled (`ptfun.c`)

```
#include <stdio.h>
#include "ptfun_c.h"

/* keep function handles in c space */
funcHandle handle;

void ptfun_setHandle(funcHandle funptr)  {
  handle = funptr;
}

funcHandle ptfun_getHandle()  {
  return handle;
}
```

Listing 6 — Makefile to build DLL (`Makefile`)

```
# for cases pointer to function

INC1=-I/usr/include
INC2=-I/usr/ch/extern/include

target: libptfun.dl Makefile

libptfun.dl: ptfun1_1.o ptfun1_1_chdl.o \
              ptfun1_2.o ptfun1_2_chdl.o \
              ptfun2.o ptfun2_chdl.o \
              ptfun3.o ptfun3_chdl.o \
              ptfun4.o ptfun4_chdl.o \
              ptfun5.o ptfun5_chdl.o \
              ptfun6.o ptfun6_chdl.o \
              ptfun.o ptfun_chdl.o
        ch dllink libptfun.dl \
                  ptfun1_1.o ptfun1_1_chdl.o \
                  ptfun1_2.o ptfun1_2_chdl.o \
                  ptfun2.o ptfun2_chdl.o \
                  ptfun3.o ptfun3_chdl.o \
                  ptfun4.o ptfun4_chdl.o \
                  ptfun5.o ptfun5_chdl.o \
                  ptfun6.o ptfun6_chdl.o \
                  ptfun.o ptfun_chdl.o
```

```
ptfun.o: ptfun.c
        ch dlcomp libptfun.dl ptfun.c $(INC1) $(INC2)
ptfun_chdl.o: ptfun_chdl.c
        ch dlcomp libptfun.dl ptfun_chdl.c $(INC1) $(INC2)

ptfun1_1.o: ptfun1_1.c
        ch dlcomp libptfun.dl ptfun1_1.c $(INC1) $(INC2)
ptfun1_1_chdl.o: ptfun1_1_chdl.c
        ch dlcomp libptfun.dl ptfun1_1_chdl.c $(INC1) $(INC2)

ptfun1_2.o: ptfun1_2.c
        ch dlcomp libptfun.dl ptfun1_2.c $(INC1) $(INC2)
ptfun1_2_chdl.o: ptfun1_2_chdl.c
        ch dlcomp libptfun.dl ptfun1_2_chdl.c $(INC1) $(INC2)

ptfun2.o: ptfun2.c
        ch dlcomp libptfun.dl ptfun2.c $(INC1) $(INC2)
ptfun2_chdl.o: ptfun2_chdl.c
        ch dlcomp libptfun.dl ptfun2_chdl.c $(INC1) $(INC2)

ptfun3.o: ptfun3.c
        ch dlcomp libptfun.dl ptfun3.c $(INC1) $(INC2)
ptfun3_chdl.o: ptfun3_chdl.c
        ch dlcomp libptfun.dl ptfun3_chdl.c $(INC1) $(INC2)

ptfun4.o: ptfun4.c
        ch dlcomp libptfun.dl ptfun4.c $(INC1) $(INC2)
ptfun4_chdl.o: ptfun4_chdl.c
        ch dlcomp libptfun.dl ptfun4_chdl.c $(INC1) $(INC2)

ptfun5.o: ptfun5.c
        ch dlcomp libptfun.dl ptfun5.c $(INC1) $(INC2)
ptfun5_chdl.o: ptfun5_chdl.c
        ch dlcomp libptfun.dl ptfun5_chdl.c $(INC1) $(INC2)

ptfun6.o: ptfun6.c
        ch dlcomp libptfun.dl ptfun6.c $(INC1) $(INC2)
ptfun6_chdl.o: ptfun6_chdl.c
        ch dlcomp libptfun.dl ptfun6_chdl.c $(INC1) $(INC2)

clear:
        rm -f  *.o
clean:
        rm -f  *.o
        rm -f  lib*.dl
```

This makefile is also for subsequent examples in this chapter.

Listing 7 — Application (`ptfun.ch`)

```
/***************************************
 * function handle with no arguments and return value
 ***************************************/
#include <math.h>
#include "ptfun.h"

void func1() {
  printf("# Ch function func1() is called \n");
```

```
  return;
}

void func2() {
  printf("# Ch function func2() is called\n");
  return;
}

int main() {
  funcHandle  handle;

  ptfun_setHandle(func1);   /* set function pointer */
  handle = ptfun_getHandle();   /* get Ch function pointer */
  handle();   /* call function by function pointer */

  ptfun_setHandle(func2);   /* set another function pointer */
  ptfun_getHandle()(); /* get function pointer and call the function */

  return 0;
}
```

Output

```
# Ch function func1() is called
# Ch function func2() is called
```

In this example , the function `ptfun_setHandle()` is called twice to set the Ch function pointers `func1` and `func2`, respectively. After every time of a Ch function pointer being set, the function `ptfun_getHandle()` is called to retrieve it back. Then the corresponding Ch function is invoked in Ch space through the retrieved function pointer. Unlike cases discussed in the previous section, Ch functions in this simple example are not called from C space. Therefore, we can handle these pointers of function as normal pointers. But, in most cases of interfacing C modules in the libraries provided by other software vendors, users don't know exactly how the passed Ch function pointers are handled in the C space. In other words, it is not guaranteed that the passed Ch functions will not be called in C space. Once a Ch function is called from C space, the program will crash. The users are strongly recommended to use the templates and examples presented in the subsequent sections.

6.2.1 Pointer to Function without Return Value and Argument

We already described how to set (or register) functions that have arguments of pointer to functions in the previous section. These Ch functions can be called from C space. This subsection will discuss how to return the pointer to functions set by functions discussed in the previous section.

We will use the following template to describe how to add a function that returns a pointer to a function without return value and argument. To return a pointer to funciton from C space to Ch space, we need to set (or register) a pointer to function first. We assume that the function pointer returned by the function `getFunction1()` below is set by function `setFunction1()` which is presented in section 6.1.1.

```
typedef void(*funcHandle)();
void setFunction1(data_type1 arg1, funcHandle funptr) {
  ...
}
funcHandle getFunction1(data_type1 arg1) {
  ...
}
```

```
#include<dlfcn.h>

void (*getFunction1(data_type1 arg1))() {
  void *dlhandle, *fptr;
  void (*retval)();

  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return NULL;
  }
  fptr = dlsym(dlhandle, "getFunction1_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return NULL;
  }

  /* call the chdl function in dynamically loaded library by address */
  dlrunfun(fptr, &retval, getFunction1, arg1);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return NULL;
  }

  return retval;
}
```

Program 6.21: Returning pointer to function without return value and argument (chf file).

`funcHandle` is typedefed as a pointer to function without return value and argument. `setFunction1`() has the same prototype in section 6.1.1. Function `getFunction1`() has an argument of simple data type `arg1` and a return value of pointer to function **funcHandle**.

The Ch function `getFunction1`() is shown in Program 6.21. Its return type `retval` is of a pointer to a function. The address of this pointer is passed to C space as the second argument of function **dlrunfun()**.

Program 6.22 shows the chdl functions getFunction1_chdl() and setFunction1_chdl(). setFunction1_chdl() is the same as shown in Program 6.4. The main difference between Program 6.4 and Program 6.22 is the added chdl function `getFunction1_chdl`() in Program 6.22.

In `getFunction1_chdl`(), after getting all arguments by function **Ch_VaArg()**, `getFunction1`() is called with an argument `arg1` of data type `data_type1`.

```
retval_c = getFunction1(arg1);
```

It returns a pointer which points to a function in C space. Typically this pointer is set by C function `setFunction1`() in `setFunction1_chdl`(). `getFunction1_chdl`() cannot return this pointer to Ch address space directly, because it is a pointer to a function in C space. In other words, only the corresponding Ch function pointer can be returned to Ch space. The consistency will be checked first with statements below

```
if(retval_c == setFunction1_chdl_funarg) {
  retval_ch = (funcHandle)setFunction1_chdl_funptr;
}
```

```
#include<ch.h>
#include<stdio.h>
/* ... */

typedef void (*funcHandle)(); /* function pointer type */

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static void setFunction1_chdl_funarg();

/* save the function pointer from the Ch space */
static void *setFunction1_chdl_funptr;

EXPORTCH void setFunction1_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument */
  /* get and save Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction1_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction1_chdl_funarg;
  }

  /* set the C function pointer instead of the Ch one,
     the NULL pointer will not be replaced */
  setFunction1(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static void setFunction1_chdl_funarg() {
  /* Call Ch function by its address */
  Ch_CallFuncByAddr(interp, setFunction1_chdl_funptr, NULL);
}
```

Program 6.22: Returning pointer to function without return value and argument (chdl file).

```
/*******************************************************/
/* following part is added for getting function handle */

EXPORTCH funcHandle getFunction1_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle  retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */

  /* get the C function pointer which is set by setFunction1_chdl() */
  retval_c = getFunction1(arg1);

  /* replace the C pointer with the Ch one */
  if(retval_c == setFunction1_chdl_funarg) { /* check the consistency */
    retval_ch = (funcHandle)setFunction1_chdl_funptr;
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}
```

Program 6.22: Returning pointer to function without return value and argument (chdl file) (Contd.).

If the function returned by C function `getFunction1`() equals the C function pointer `setFunction1_chdl_funarg`, we can say that the returned pointer `retval_c` is the right pointer that was set in function `setFunction1_chdl`(). Therefore, the pointer kept in `setFunction1_chdl_funptr` must be the corresponding Ch function pointer which should be returned instead of the C function pointer `retval_c`. This is the key point of this case. If function `setFunction1_chdl`() is called for multiple times to set different Ch function pointers, the last one will replace the previous one, Therefore, function `ptfun1_getHandle_chdl`() returns the last Ch function pointers set by function `setFunction1_chdl`().

If the equation doesn't hold, the pointer `retval_c` is not the last pointer set by function `setFunction1()`. It may be a wrong result or a default system function. The default system function is the callback function which defined by system rather than the users. Typically it is set by default when the software is started. If this is the case, the NULL pointer will be returned. The case about the function which returns a pointer to default system function will be discussed in section6.3.1

Figures 6.2 illustrates the relations between the functions setting or getting function pointers, and functions to be set in both Ch and C spaces. In Ch space, the Ch function `funptr`() is set and passed to C space. In C space, the temporary C function `setFunction1_chdl_funarg`() is practically set by `setfunction`(). If this C function is called back later, it will invoke the corresponding Ch function `funptr`() using API **Ch_CallFuncByAddr**().

After that, if the Ch function `getFunction1`() is called to get the function pointer which has been set before, the C functions `getFunction1_chdl`() and `getFunction1`() are invoked. Set by `setfunction`(), the function pointer `setFunction1_chdl_funarg`() will be returned by the C function `getFunction1`(). In function `getFunction1_chdl`(), this C function pointer is replaced with the corresponding Ch function pointer `funptr`, which has been set by Ch function `setFunction1`(). Finally, it is Ch function pointer that will be returned by the Ch function `getFunction1`().

Example

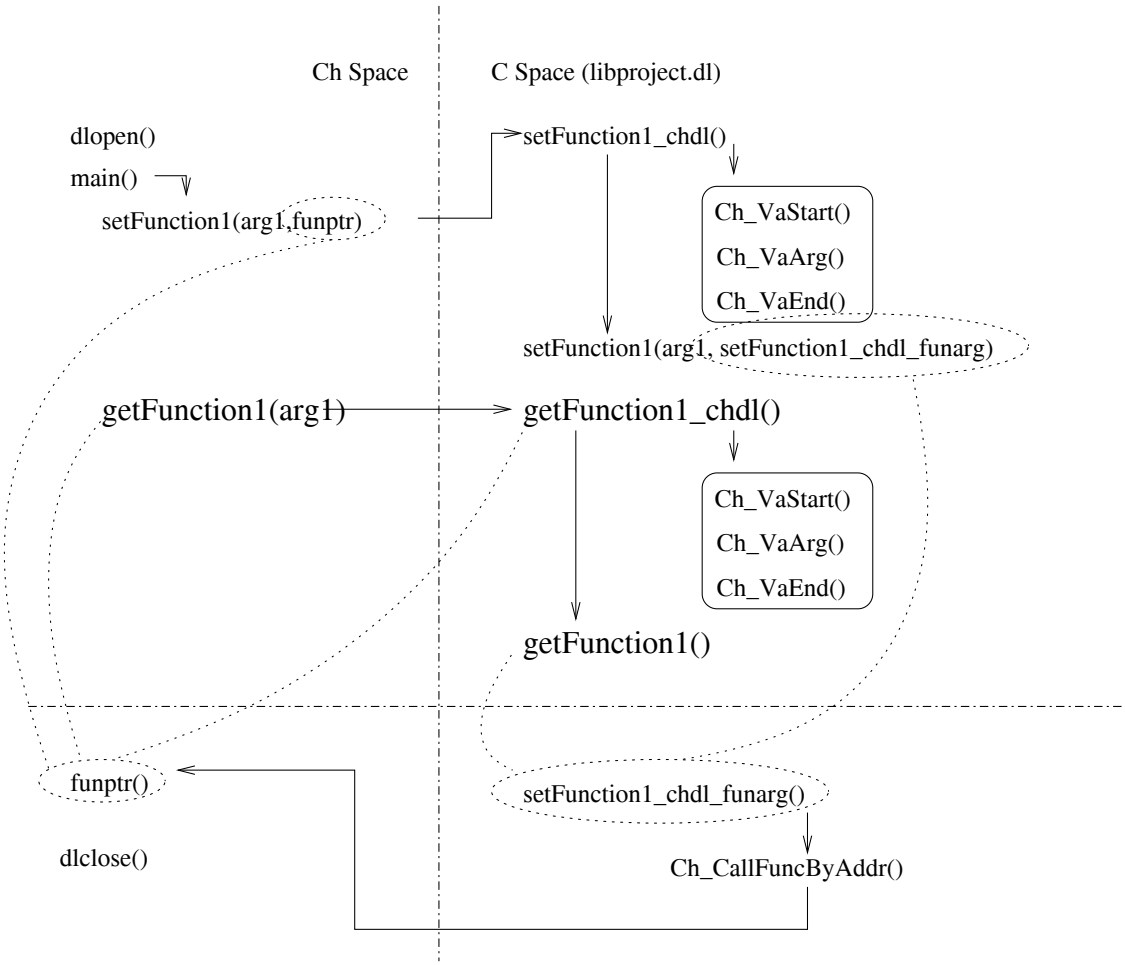


Figure 6.2: Functions and APIs in Ch and C Spaces for getting a function.

This is a complete example which illustrates the case of setting and getting pointer to function without return value and argument.

The functions `ptfun1_1_setHandle`() is used to set a pointer to function, and function `ptfun1_1_getHandle`() can be called to get the function pointer set by function `ptfun1_1_setHandle`(). If the function `ptfun1_1_setHandle`() is called for multiple times with different function pointers, the new Ch function pointer will replace the old one.

Listing 1 — header file for Ch functions (`ptfun1_1.h`)

```
#ifndef _PTFUN1_1_H_
#define _PTFUN1_1_H_

typedef void(*funcHandle)();
void ptfun1_1_setHandle(funcHandle funptr);
funcHandle ptfun1_1_getHandle();

/* below is added only for Ch */
#include <dlfcn.h>
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
}

/* user-defined function which is called to close DLL
   when the program exits */
void _dlclose_ptfun1_1(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun1_1);

#pragma importf <ptfun1_1.chf>
#endif /* _PTFUN1_1_H_ */
```

Besides the content of C header file `ptfun1_1_c.h`, some particular statements for Ch are added in this file. The DLL file `libptfun.dl` is loaded with statement

```
void *_Chptfun1_handle = dlopen("libptfun.dl", RTLD_LAZY);
```

To close this DLL file, the statement

```
atexit(_dlclose_ptfun1);
```

instructs the system to invoke the user-defined function `_dlclose_ptfun1` when the program exits. In the previous templates, the DLL file is handled in the chf functions.

The statement

```
#pragma importf <ptfun1_1.chf>
```

includes function file `ptfun1_1.chf`. Typically, we defined only one Ch function in a function file, and the file name is named after the corresponding Ch function. Therefore, Ch can search for the proper Ch function file by the function name. In this case, for the sake of convenience, we put two Ch functions, `ptfun1_1_setHandle`() and ptfun1_1_getHandle(), into the same function file. The `#pragma importf` directive is used to include definitions of these two functions in the header file, thereby in the Ch application `ptfun1_1.ch` in Listing 7. More information about `#pragma` directive is available in *Ch*

User's Guide.

Listing 2 — Ch function file (`ptfun1_1.chf`)

```
#include <dlfcn.h>

void ptfun1_1_setHandle(funcHandle funptr)  {
  void *fptr;

  fptr = dlsym(_Chptfun_handle, "ptfun1_1_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, ptfun1_1_setHandle, funptr);
  return;
}

funcHandle ptfun1_1_getHandle()  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun1_1_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, ptfun1_1_getHandle);
  return retval;
}
```

Two Ch functions, `ptfun1_1_setHandle`() and ptfun1_1_getHandle(), are defined in this Ch function file.

Listing 3 — header file for C functions (`ptfun1_1_c.h`)

```
#ifndef _PTFUN1_1_H_
#define _PTFUN1_1_H_

typedef void (*funcHandle)();
void ptfun1_1_setHandle(funcHandle funptr);
funcHandle ptfun1_1_getHandle();

#endif /* _PTFUN1_1_H_ */
```

This is the header file used for compiling C files.

Listing 4 — chdl file (`ptfun1_1_chdl.c`)

```
#include <stdio.h>
#include "ptfun1_1_c.h"
#include <ch.h>

static ChInterp_t interp;
static void ptfun1_1_chdl_funarg();
static void *ptfun1_1_chdl_funptr;

EXPORTCH void ptfun1_1_setHandle_chdl(void *varg){
```

```
  ChVaList_t ap;
  funcHandle handle_ch, handle_c;

  Ch_VaStart(interp, ap, varg);
  /* get and save the Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  ptfun1_1_chdl_funptr = (void *)handle_ch;
  printf("\nin ptfun1_1_setHandle_chdl(), Ch function pointer is %p\n",
          handle_ch);

  /* replace the Ch function pointer with the C one */
  if(handle_ch == NULL) { /* NULL pointer will not replaced */
    handle_c = NULL;
  }
  else {
    handle_c = (funcHandle)ptfun1_1_chdl_funarg;
  }

  printf("in ptfun1_1_setHandle_chdl(), \
the corresponding C function pointer is %p\n\n", handle_c);
  ptfun1_1_setHandle(handle_c);

  Ch_VaEnd(interp, ap);
}

EXPORTCH funcHandle ptfun1_1_getHandle_chdl(void *varg){
  ChVaList_t ap;
  funcHandle retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);

  retval_c = ptfun1_1_getHandle();
  if(retval_c == ptfun1_1_chdl_funarg) { /* check the consistency */
    retval_ch = (funcHandle)ptfun1_1_chdl_funptr;
  }

  printf("in ptfun1_1_getHandle_chdl(), \
the returned C function pointer %p\n", retval_c);
  printf("in ptfun1_1_getHandle_chdl(), \
Ch function pointer to be returned is %p\n\n", retval_ch);

  Ch_VaEnd(interp, ap);
  return retval_ch;
}

/**** C functions to replace Ch callback functions ****/
static void ptfun1_1_chdl_funarg() {
  Ch_CallFuncByAddr(interp, ptfun1_1_chdl_funptr, NULL);
}
```

Listing 5 — C functions to be handled (`ptfun1_1.c`)

```
#include <stdio.h>
#include "ptfun1_1_c.h"

/* keep function handles in c space */
funcHandle handle;

void ptfun1_1_setHandle(funcHandle funptr)  {
  printf("Call Ch function in C space : ");
```

```
  funptr(); /* call Ch function in C space */
  handle = funptr;
}

funcHandle ptfun1_1_getHandle()  {
  return handle;
}
```

In this file, two C functions, `ptfun1_1_setHandle`() and `ptfun1_1_getHandle`(), are defined to save and return function pointers set by the user. Mostly, APIs of this kind are available in binary libraries. What users need to do is to link their chdl functions with the binary libraries. As it is mentioned before, this document places great emphasis on coding chf and chdl files. We give the source code of C file `ptfun1_1.c` only for the completeness of the example. In practic, the user can only get shared library or static library instead of the C and C++ function files. It is recommended to skip this file, because it may distract the users' attention from the chdl and chf files.
To make difference from the example of `ptfun.ch`, the Ch function passed in is called from C space.

Listing 6 — Application (`ptfun1_1.ch`)

```
/*************************************
 * function handle with no arguments and return value
 *************************************/
#include <math.h>
#include "ptfun1_1.h"

void ptfun1_1() {
  printf("# Ch function ptfun1_1() in ptfun1.ch is called \n\n");
  return;
}

void ptfun1_2() {
  printf("# Ch funtion ptfun1_2() in ptfun1.ch is called\n");
  return;
}

int main() {
  funcHandle  handle;

  ptfun1_1_setHandle(ptfun1_1);   /* set function pointer */
  handle = ptfun1_1_getHandle();   /* get Ch function pointer */
  handle();   /* call function by function pointer */

  ptfun1_1_setHandle(ptfun1_2);   /* set another function pointer */
  ptfun1_1_getHandle()(); /* get function pointer and call the function */

  return 0;
}
```

In this application, the function `ptfun1_1_setHandlel`() is called twice to set the Ch function pointers `ptfun1_1` and `ptfun1_2`, respectively. After every time of a Ch function pointer being set, the function `ptfun1_1_getHandle`() is called to retrieve it back. Then the corresponding Ch function is invoked through the retrieved function pointer.

In this example, the pointer `ptfun1_2` overwrites the pointer `ptfun1_1` after the second calling `ptfun1_1_setHandlel`().

Output

```
in ptfun1_1_setHandle_chdl(), Ch function pointer is 1bd348
in ptfun1_1_setHandle_chdl(), the corresponding C function pointer is ef572308

Call Ch function in C space : # Ch function ptfun1_1() in ptfun1.ch is called

in ptfun1_1_getHandle_chdl(), the returned C function pointer ef572308
in ptfun1_1_getHandle_chdl(), Ch function pointer to be returned is 1bd348

# Ch function ptfun1_1() in ptfun1.ch is called


in ptfun1_1_setHandle_chdl(), Ch function pointer is 1bd670
in ptfun1_1_setHandle_chdl(), the corresponding C function pointer is ef572308

Call Ch function in C space : # Ch funtion ptfun1_2() in ptfun1.ch is called
in ptfun1_1_getHandle_chdl(), the returned C function pointer ef572308
in ptfun1_1_getHandle_chdl(), Ch function pointer to be returned is 1bd670

# Ch funtion ptfun1_2() in ptfun1.ch is called
```

By the output above, we can understand better the whole procedure of the pointer replacements. In Ch space, the Ch function pointer `ptfun1_1` with value `0x1cb260` is passed to the chdl function `ptfun1_1_setHandle_chdl`(). In the chdl function, it is replaced with the C function pointer `ptfun1_1_chdl_funarg` with value `0xef572264`. It is pointer `0xef572264` that is set in C space by function `ptfun1_1_setHandle`().

While Ch function `ptfun1_1_getHandle`() is called in Ch space, functions `ptfun1_1_getHandle_chdl`() and `ptfun1_1_getHandle`() are called subsequently in C space. The C function `ptfun1_1_getHandle`() returns the last C pointer set by C function `ptfun1_1_setHandle`(). In function `ptfun1_1_getHandle_chdl`(), the returned pointer `0xef572264` is replaced with the corresponding Ch function pointer `0x1cb260`. The pointer `0x1cb260` which stands for Ch function `ptfun1_1`() is finally returned by function `ptfun1_1_getHandle_chdl`() to Ch space.

Example

In the previous example of setting function pointer, the function `ptfun1_1_setHandle`() can set and keep only one Ch function pointer. In other words, the new Ch function pointer will overwrite the old one if function `ptfun1_1_setHandle`() is called for multiple times. However, the same API can be used to set different callback functions for different events in some applications. For example, with the same API, user can set a callback function to handle the event of clicking on the right button of the mouse, the event of clicking on the left button and the event of clicking on the middle button. The similar method can also be applied for handling menus.

For the example listed below, the functions `ptfun1_2_setHandle`() can set a function pointer with an event ID, whereas the function `ptfun1_2_getHandle`() can retrieve the function pointer by the associated event ID. Different Ch callback functions can be set with different event ID using the same function `ptfun1_2_setHandle`().

Listing 1 — header file for Ch functions (`ptfun1_2.h`)

```
#ifndef _PTFUN1_2_H_
#define _PTFUN1_2_H_

typedef void(*funcHandle)();
```

```
int ptfun1_2_setHandle(int num, funcHandle funptr);
funcHandle ptfun1_2_getHandle(int num);

/* below is added only for Ch */
#include <dlfcn.h>
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
}

/* initialize and set static function pointer array */
void *_fptr = dlsym(_Chptfun_handle, "ptfun1_2_setgetHandles_init");
if(_fptr == NULL)  {
   printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
   exit(-1);
}
dlrunfun(_fptr, NULL, NULL);

/* usr-defined function which is called to close DLL
   when the program exits */
void _dlclose_ptfun1_2(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun1_2);

#pragma importf <ptfun1_2.chf>
#endif /* _PTFUN1_2_H_ */
```

After the DLL is loaded, the C function `ptfun1_2_setgetHandles_init`(), which is defined in file ptfun1_2_chdl.c in Listing 4, is called to initialize a static array in C space. With this static array, named ptfun1_2_chdl_funarg, all C function pointers can be accessed by the index.

Listing 2 — Ch function file (`ptfun1_2.chf`)

```
#include <dlfcn.h>

int ptfun1_2_setHandle(int i, funcHandle funptr)  {
  void *fptr;
  int retval;

  fptr = dlsym(_Chptfun_handle, "ptfun1_2_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, ptfun1_2_setHandle, i, funptr);
  return retval;
}

funcHandle ptfun1_2_getHandle(int i)  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun1_2_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
```

```
  }

  dlrunfun(fptr, &retval, ptfun1_2_getHandle, i);
  return retval;
}
```

Listing 3 — header file for C functions (`ptfun1_2_c.h`)

```
#ifndef _PTFUN1_2_H_
#define _PTFUN1_2_H_

typedef void (*funcHandle)();
int ptfun1_2_setHandle(int num, funcHandle funptr);
funcHandle ptfun1_2_getHandle(int num);

#endif /* _PTFUN1_2_H_ */
```

Listing 4 — chdl file (`ptfun1_2_chdl.c`)

```
#include <stdio.h>
#include "ptfun1_2_c.h"
#include <ch.h>

/* The maximum number of ptfun1_2_setHandle()
   can be called */
#define MAXNUM   3

/* C functions to replace the Ch one */
static ChInterp_t interp;
static void (*ptfun1_2_chdl_funarg[MAXNUM])();

/* function pointers to save Ch function pointers */
static void *ptfun1_2_chdl_funptr[MAXNUM];

/* index of pointer pairs, from 0 to MAXNUM-1 */
static int index = 0;

EXPORTCH int ptfun1_2_setHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle handle_ch, handle_c;
  int retval;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);
  handle_ch = Ch_VaArg(interp, ap, funcHandle);

  if(handle_ch == NULL) {
    handle_c = NULL;
  }
  else {
    if(index == MAXNUM) {
      printf("Error: Only %d function pointers can be set\n", MAXNUM);
      return -1;
    }
    ptfun1_2_chdl_funptr[index] = (void *)handle_ch;
    handle_c = ptfun1_2_chdl_funarg[index];
    index++;
  }
```

```
  printf("in ptfun1_2_setHandle_chdl():\nCh function pointer %p is \
replaced by C function pointer %p\n\n", handle_ch, handle_c);
  retval = ptfun1_2_setHandle(id, handle_c);

  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH funcHandle ptfun1_2_getHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);

  retval_c = ptfun1_2_getHandle(id);

  /* replace C pointer with corresponding Ch function pointer */
  if(retval_c == NULL)  {
    retval_ch = NULL;
  }
  else {
    for(i = 0; i < index; i++) {
      if(ptfun1_2_chdl_funarg[i] == retval_c) {
        retval_ch = (funcHandle)ptfun1_2_chdl_funptr[i];
      }
    }
  }
  Ch_VaEnd(interp, ap);
  printf("in ptfun1_2_getHandle_chdl():\nC function pointer %p is \
replaced by Ch function pointer %p\n", retval_c, retval_ch);
  return retval_ch;
}

/**** C functions to replace Ch callback functions ****/

static void ptfun1_2_chdl_funarg0() {
  Ch_CallFuncByAddr(interp, ptfun1_2_chdl_funptr[0], NULL);
}

static void ptfun1_2_chdl_funarg1() {
  Ch_CallFuncByAddr(interp, ptfun1_2_chdl_funptr[1], NULL);
}

static void ptfun1_2_chdl_funarg2() {
  Ch_CallFuncByAddr(interp, ptfun1_2_chdl_funptr[2], NULL);
}

/********** initialization routine called in ptfun1_2.h **********/
EXPORTCH int ptfun1_2_setgetHandles_init() {
  /* from 0 to MAXNUM -1 */

   ptfun1_2_chdl_funarg[0] = ptfun1_2_chdl_funarg0;
   ptfun1_2_chdl_funarg[1] = ptfun1_2_chdl_funarg1;
   ptfun1_2_chdl_funarg[2] = ptfun1_2_chdl_funarg2;
```

```
   return 0;
}
```

Listing 5 — C functions to be handled (`ptfun1_2.c`)

```
#include <stdio.h>
#include "ptfun1_2_c.h"

#define MAXNUM 500

/* keep function handles in C space */
struct FuncList{
        int num; /* id of the function handle */
        funcHandle handle;
} funcList[MAXNUM];

static int index_c = 0;   /* index of funList */
int ptfun1_2_setHandle(int num, funcHandle funptr)  {
  int i;

  if(index_c == MAXNUM) {
    printf("Error: too many functions");
    return -1;
    }

  /* make sure that one id is mapped to only one handle */
  for(i=0; i<index_c; i++)
    if(funcList[i].num == num) {
      funcList[i].num = num;
      funcList[i].handle = funptr;
      return num;
    }
  funcList[index_c].num = num;
  funcList[index_c++].handle = funptr;

  /* Call Ch function from C space */
  printf("Calling Ch function from C space:");
  funptr();

  return num;
}

funcHandle ptfun1_2_getHandle(int num)  {
  int i;

  for(i=0; i<index_c; i++)
    if(funcList[i].num == num)
      return funcList[i].handle;
  return NULL;
}
```

The macro `MAXNUM` defines the maximum number of different Ch function pointers to be set and saved. The static function pointer array `ptfun1_2_chdl_funarg` is used to access all C function pointers by an index. When the program is started, this array is initialized by function `ptfun1_2_setgetHandles_init`() which is invoked in the header file `ptfun1_2.h`. The static pointer array `ptfun1_2_chdl_funptr` make it possible to save multiple Ch function pointers at the same time. The static variable `index` can index both these two arrays. A C function pointer in array `ptfun1_2_chdl_funarg` is supposed to match the Ch one in array `ptfun1_2_chdl_funptr` with the same index.

In function ptfun1_2_setHandle_chdl(), if the Ch function pointer retval_ch is not a NULL pointer, it is saved in array ptfun1_2_chdl_funptr, at the same time a C function is assigned to replace it as follows

```
ptfun1_2_chdl_funptr[index] = (void *)handle_ch;
handle_c = ptfun1_2_chdl_funarg[index];
```

Otherwise, a NULL pointer is used.

In function ptfun1_2_getHandle_chdl(), if the C function pointer retval_c returned by function ptfun1_2_getHandle() can be found in array ptfun1_2_chdl_funarg, it is replaced with the corresponding Ch function pointer in array ptfun1_2_chdl_funptr with the same index as follows.

```
if(ptfun1_2_chdl_funarg[i] == retval_c) {
   retval_ch = (funcHandle)ptfun1_2_chdl_funptr[i];
}
```

Listing 6 — Application ptfun1_2.ch

```
#include "ptfun1_2.h"

void ptfun1_1() {
  printf("\nptfun1_1() in Ch space is called \n\n");
  return;
}

void ptfun1_2() {
  printf("\nptfun1_2() in Ch space is called \n\n");
  return;
}

int main() {
  int id;
  funcHandle  handle;

  /* set function pointer */
  ptfun1_2_setHandle(1, ptfun1_1);
  ptfun1_2_setHandle(2, ptfun1_2);

  handle = ptfun1_2_getHandle(1); /* get function pointer with event 1 */
  handle(); /* call function by function pointer */
  /* get function pointer and call function with event 2 */
  ptfun1_2_getHandle(2)();

  ptfun1_2_setHandle(2, ptfun1_1);
  ptfun1_2_getHandle(2)();

  return 0;
}
```

Two Ch function pointers, ptfun1_1 and ptfun1_2, are set with different event ID by function ptfun1_2_setHandle(). The function pointer ptfun1_2 set with event ID 2 will not overwrite the previous function pointer set with event ID 1. Both of them can be retrieved by function ptfun1_2_getHandle() using associated event ID.

Output

```
in ptfun1_2_setHandle_chdl():
Ch function pointer 1d9078 is replaced by C function pointer ef5a25b8

Calling Ch function from C space:
ptfun1_1() in Ch space is called

in ptfun1_2_setHandle_chdl():
Ch function pointer 1d28a8 is replaced by C function pointer ef5a25d4

Calling Ch function from C space:
ptfun1_2() in Ch space is called

in ptfun1_2_getHandle_chdl():
C function pointer ef5a25b8 is replaced by Ch function pointer 1d9078

ptfun1_1() in Ch space is called

in ptfun1_2_getHandle_chdl():
C function pointer ef5a25d4 is replaced by Ch function pointer 1d28a8

ptfun1_2() in Ch space is called

in ptfun1_2_setHandle_chdl():
Ch function pointer 1d9078 is replaced by C function pointer ef5a25f4

in ptfun1_2_getHandle_chdl():
C function pointer ef5a25f4 is replaced by Ch function pointer 1d9078

ptfun1_1() in Ch space is called
```

The function `ptfun1_2_setHandle`() is called three times to set Ch function pointers. In chdl function, three differents C function pointers are assigned to replace them.

6.2.2 Pointer to Function with Return Value

This section describes functions that return pointers to functions with return values. We assume that the returned function pointer of the function `getFunction2`() below is already set by `setFunction2`() which is presented in section 6.1.2.

```
typedef return_type (*funcHandle)();
funcHandle getFunction2(data_type1 arg1) {
  ...
}
```

`funcHandle` has the same typedef statement as the one in `setFunction2`(). `funcHandle` is defined as a pointer to function that has a return value but without argument. `getFunction2`() has both argument and return value. The argument `arg1` is of simple data type `data_type1` and the return value is a pointer of data type `funcHandle`.

The Ch function `getFunction2`() is shown in Program 6.23.

```
#include<dlfcn.h>

return_type(*getFunction2(data_type1 arg1))() {
  void *dlhandle, *fptr;
  return_type (*retval)();

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n", __func__,
                     dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "getFunction2_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address
   arguments are passed to chdl function here */
  dlrunfun(fptr, &retval, getFunction2, arg1);

  /* close the dynamically loaded library */
  if(dlclose(dlhandle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n", __func__,
                     dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 6.23: Returning pointer to function with return value (chf file).

```
#include<ch.h>
#include<stdio.h>
/* ... */

typedef return_type (*funcHandle)(); /* function pointer type */

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static return_type setFunction2_chdl_funarg();

/* for saving the function pointer from Ch space */
static void *setFunction2_chdl_funptr;

EXPORTCH void setFunction2_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle  handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get and save Ch function pointer  */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction2_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one */
  if(handle_ch != NULL) { /* the NULL pointer can't be replaced */
    handle_c = (funcHandle)setFunction2_chdl_funarg;
  }

  setFunction2(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static return_type setFunction2_chdl_funarg() {
  return_type retval;

  /* Call Ch address space function by its address */
  Ch_CallFuncByAddr(interp, setFunction2_chdl_funptr, &retval);
  return retval;
}
```

Program 6.24: Returning pointer to function with return value (chdl file).

```
/*****************************************************/
/* following part is added for getting function handle */

EXPORTCH funcHandle getFunction2_chdl(void *varg) {
  ChVaList_t ap;

  data_type1 arg1;
  funcHandle  retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */

  /* get the C function pointer which is set by setFunction2_chdl() */
  retval_c = getFunction2(arg1);

  /* replace the C pointer with the Ch one */
  if(retval_c == setFunction2_chdl_funarg) { /* check the consistency */
    retval_ch = (funcHandle)setFunction2_chdl_funptr;
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}
```

Program 6.24: Returning pointer to function with return value (chdl file) (Contd.).

The chdl function `getFunction2_chdl`() is shown in Program 6.24. It is created based on Program 6.22. The only difference between Programs 6.24 and 6.22 is the typedef statement. Therefore, function `setFunction2_chdl_funarg`() has a different prototype. Variables `handle_ch` in function `setFunction2_chdl`() and `retval_c` and `retval_ch` in function `getFunction2_chdl`() have different data types. The return value of function `getFunction2_chdl`() is also different.

In Program 6.24, only one function pointer can be set and kept. If function `setFunction2_chdl_funarg`() is called for multiple times, the function `getFunction2_chdl_funarg`() retrieves the pointer set in the last time. However, function `ptfun2_setHandle`() in the example below can set up to 3 different Ch functions, and function `ptfun2_getHandle`() can retrieve them all by different event ID.

Example

In this example, functions `ptfun2_setHandle`() and `ptfun2_getHandle` illustrate how to set and get function pointers which point to functions with returned values. If you already understand the previous section, it will be easy to follow this example.

Listing 1 — header file for Ch functions (`ptfun2.h`)

```
#ifndef _PTFUN2_H_
#define _PTFUN2_H_

typedef int (*funcHandle)(); /* return a int */
int ptfun2_setHandle(int num, funcHandle funptr);
funcHandle ptfun2_getHandle(int num);

/* added for Ch */
#include <dlfcn.h>
```

```
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
}

/* initialize and set static function pointer array */
void *_fptr = dlsym(_Chptfun_handle, "ptfun2_setgetHandles_init");
if(_fptr == NULL)  {
   printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
   exit(-1);
}
dlrunfun(_fptr, NULL, NULL);

/* to close dl */
void _dlclose_ptfun2(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun2);

#pragma importf <ptfun2.chf>
#endif /* _PTFUN2_H_ */
```

Listing 2 — Ch function file (`ptfun2.chf`)

```
#include <dlfcn.h>

int ptfun2_setHandle(int i, funcHandle funptr)  {
  void *fptr;
  int retval;

  fptr = dlsym(_Chptfun_handle, "ptfun2_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, ptfun2_setHandle, i, funptr);
  return retval;
}

funcHandle ptfun2_getHandle(int i)  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun2_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, ptfun2_getHandle, i);
  return retval;
}
```

Listing 3 — header file for C functions (`ptfun2.h`)

```
#ifndef _PTFUN2_H_
#define _PTFUN2_H_
```

```
typedef int (*funcHandle)();
int ptfun2_setHandle(int num, funcHandle funptr);
funcHandle ptfun2_getHandle(int num);

#endif /* _PTFUN2_H_ */
```

Listing 4 — chdl (`ptfun2_chdl.c`)

```
#include <stdio.h>
#include "ptfun2_c.h"
#include <ch.h>

/* The maximum number of ptfun2_setHandle()
   can be called */
#define MAXNUM   3

/* C functions to replace the Ch one */
static ChInterp_t interp;
static int (*ptfun2_chdl_funarg[MAXNUM])();

/* function pointers to save Ch function pointers */
static void *ptfun2_chdl_funptr[MAXNUM];

/* index of pointer pairs, from 0 to MAXNUM-1 */
static int index = 0;

EXPORTCH int ptfun2_setHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle handle_ch, handle_c;
  int retval;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);
  handle_ch = Ch_VaArg(interp, ap, funcHandle);

  if(handle_ch == NULL) {
    handle_c = NULL;
  }
  else {
    if(index == MAXNUM) {
      printf("Error: Only %d function pointers can be set\n", MAXNUM);
      return -1;
    }
    ptfun2_chdl_funptr[index] = (void *)handle_ch;
    handle_c = ptfun2_chdl_funarg[index];
    index++;
  }

  retval = ptfun2_setHandle(id, handle_c);

  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH funcHandle ptfun2_getHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
```

```
  funcHandle retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);

  retval_c = ptfun2_getHandle(id);

  /* replace C pointer with corresponding Ch function pointer */
  if(retval_c == NULL)  {
    retval_ch = NULL;
  }
  else {
    for(i = 0; i < index; i++) {
      if(ptfun2_chdl_funarg[i] == retval_c) {
        retval_ch = (funcHandle)ptfun2_chdl_funptr[i];
      }
    }
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}

/**** hundreds of functions can be here ****/

static int ptfun2_chdl_funarg0() {
  int retval;

  Ch_CallFuncByAddr(interp, ptfun2_chdl_funptr[0], &retval);
  return retval;
}

static int ptfun2_chdl_funarg1() {
  int retval;

  Ch_CallFuncByAddr(interp, ptfun2_chdl_funptr[1], &retval);
  return retval;
}

static int ptfun2_chdl_funarg2() {
  int retval;

  Ch_CallFuncByAddr(interp, ptfun2_chdl_funptr[2], &retval);
  return retval;
}

/********* initialization routine called in ptfun2.h **********/
EXPORTCH int ptfun2_setgetHandles_init() {
  /* from 0 to MAXNUM -1 */

   ptfun2_chdl_funarg[0] = ptfun2_chdl_funarg0;
   ptfun2_chdl_funarg[1] = ptfun2_chdl_funarg1;
   ptfun2_chdl_funarg[2] = ptfun2_chdl_funarg2;

   return 0;
}
```

Listing 5 — C functions to be added (`ptfun2.c`)

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

```
#include "ptfun2_c.h"
/* #include <ptfun2.h> */

#define MAXNUM 500

struct FuncList{
        int num; /* id of the function handle */
        funcHandle handle;
};

/* to keep function handles in c space */
struct FuncList funcList[MAXNUM];
static int index_c = 0;   /* index of funList */

int ptfun2_setHandle(int num, funcHandle funptr)  {
  int i;

  if(index_c == MAXNUM) {
    printf("Error: too many functions");
    return -1;
    }

  /* make sure that one id is mapped to only one handle */
  for(i=0; i<index_c; i++)
    if(funcList[i].num == num) {
      funcList[i].num = num;
      funcList[i].handle = funptr;
      return num;
    }
  funcList[index_c].num = num;
  funcList[index_c++].handle = funptr;

  /* Call Ch function from C space */
  printf("Calling Ch function from C space: return value is %d\n",
          funptr());

  return num;
}

funcHandle ptfun2_getHandle(int num)  {
  int i;

  for(i=0; i<index_c; i++)
    if(funcList[i].num == num)
      return funcList[i].handle;
  return NULL;
}
```

Listing 6 — Application (`ptfun2.ch`)

```
#include "ptfun2.h"

int ptfun2_1() {
  printf("ptfun2_1() in ptfun2.ch is called \n");
  return 10;
}

int ptfun2_2() {
  printf("ptfun2_2() in ptfun2.ch is called \n");
  return 20;
```

```
}

int main() {
  int id;
  funcHandle  handle;

  /* set function handle */
  ptfun2_setHandle(1, ptfun2_1);
  ptfun2_setHandle(2, ptfun2_2);

  /* get function handle */
  handle = ptfun2_getHandle(1);
  printf("retval of id1 = %d\n", handle());
  handle  = ptfun2_getHandle(2);
  printf("retval of id2 = %d\n", handle());

  ptfun2_setHandle(2, ptfun2_1);
  printf("retval of id2 = %d\n", ptfun2_getHandle(2)());

  return 0;
}
```

Output

```
ptfun2_1() in ptfun2.ch is called
Calling Ch function from C space: return value is 10
ptfun2_2() in ptfun2.ch is called
Calling Ch function from C space: return value is 20
ptfun2_1() in ptfun2.ch is called
retval of id1 = 10
ptfun2_2() in ptfun2.ch is called
retval of id2 = 20
ptfun2_1() in ptfun2.ch is called
retval of id2 = 10
```

6.2.3 Pointer to Function with Arguments

This section discusses functions that return pointer to functions with arguments. Here, we assume that the returned function pointer of the function `getFunction3`() below is already set by `setFunction3`() which is presented in section 6.1.3.

```
typedef void (*funcHandle)(data_type2);
void setFunction3(data_type1 arg1, funcHandle){
 ...
}
funcHandle getFunction3(data_type1 arg1) {
  ...
}
```

`funcHandle` is typedefed as a pointer to a function that has an argument of data type `data_type2`. Function `getFunction3`() has both argument and return value. The argument `arg1` is of simple data type `data_type1` and the return value is a pointer of data type `funcHandle`.

The Ch function `getFunction3`() is shown in Program 6.25. The return value of `getFunction3`() is defined as `funcHandle` which is typedefed as a pointer to function that has an argument of data type `data_type2`. Therefore, the arguments of function **dlrunfun()** are different. This is the only difference between Program 6.25 and other function files such as Program 5.5 and Program 6.21.

```
#include<dlfcn.h>

void (*getFunction3(data_type1 arg1))(data_type2) {
  void *dlhandle, *fptr;
  void (*retval)(data_type2);

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                     __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "getFunction3_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded
     library by address arguments to chdl function */
  dlrunfun(fptr, &retval, getFunction3, arg1);

  /* close the dynamically loaded library */
  if(dlclose(dlhandle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
                     __func__, dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 6.25: Returning pointer to function with arguments (chf file).

```
#include<ch.h>
#include<stdio.h>
/* ... */

/* function pointer type which points the function with argument */
typedef void (*funcHandle)(data_type2);

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static void setFunction3_chdl_funarg(data_type2 arg2);

/* save the function pointer from the Ch space */
static void *setFunction3_chdl_funptr;

EXPORTCH void setFunction3_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get and save Ch function pointer */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction3_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction3_chdl_funarg;
  }

  setFunction3(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static void setFunction3_chdl_funarg(data_type2 arg2) {
  /* Call Ch callback function by its address */
  Ch_CallFuncByAddr(interp, setFunction3_chdl_funptr, NULL, arg2);
}
```

Program 6.26: Returning pointer to function with arguments (chdl file).

```
/*******************************************************/
/* following part is added for getting function handle */

EXPORTCH funcHandle getFunction3_chdl(void *varg) {
  ChVaList_t ap;

  data_type1 arg1;
  funcHandle  retval_c = NULL, retval_ch = NULL;

  /* get arguments passed from the Ch function */
  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */

  /* get the C function pointer which is set by setFunction3_chdl() */
  retval_c = getFunction3(arg1);

  /* replace the C pointer with the Ch one */
  if(retval_c == setFunction3_chdl_funarg) { /* check the consistency */
    retval_ch = (funcHandle)setFunction3_chdl_funptr;
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}
```

Program 6.26: Returning pointer to function with arguments (chdl file) (Contd.).

The chdl function `getFunction3_chdl`() is shown in Program 6.26, Program 6.26 is based on Program 6.8 and Program 6.22. The only difference between Program 6.26 and Program 6.22 is that `funcHandle` is typedefed as a pointer to function that has an argument of data type `data_type2`. To replace Ch functions of different prototypes, prototypes of functions `setFunction3_chdl_funarg` and `setFunction2_chdl_funarg` are different. This is also true for the variables such as `handle_c`, `handle_ch`, `retval_ch`, and `retval_c`. These variables have data types that are defined specifically by `funcHandle`.

Example

Functions `ptfun3_setHandle`() and `ptfun3_getHandle` in this example illustrates the case of setting and getting pointers to functions that have arguments.

Listing 1 — header file for Ch functions (`ptfun3.h`)

```
#ifndef _PTFUN3_H_
#define _PTFUN3_H_

typedef void (*funcHandle)(float, int);
int ptfun3_setHandle(int num, funcHandle funptr);
funcHandle ptfun3_getHandle(int num);

/* below is added for Ch */
#include <dlfcn.h>
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
```

```
}

/* initialize and set static function pointer array */
void *_fptr = dlsym(_Chptfun_handle, "ptfun3_setgetHandles_init");
if(_fptr == NULL)  {
   printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
   exit(-1);
}
dlrunfun(_fptr, NULL, NULL);

/* to close dl */
void _dlclose_ptfun3(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun3);

#pragma importf <ptfun3.chf>
#endif /* _PTFUN3_H_ */
```

Listing 2 — chf file (`ptfun3.chf`)

```
#include <dlfcn.h>

int ptfun3_setHandle(int i, funcHandle funptr)  {
  void *fptr;
  int retval;

  fptr = dlsym(_Chptfun_handle, "ptfun3_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, ptfun3_setHandle, i, funptr);
  return retval;
}

funcHandle ptfun3_getHandle(int i)  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun3_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, ptfun3_getHandle, i);
  return retval;
}
```

Listing 3 — header file for C functions (`ptfun3.h`)

```
#ifndef _PTFUN3_H_
#define _PTFUN3_H_

typedef void (*funcHandle)(float, int);
int ptfun3_setHandle(int num, funcHandle funptr);
funcHandle ptfun3_getHandle(int num);

#endif /* _PTFUN3_H_ */
```

Listing 4 — chdl file (`ptfun3_chdl.c`)

```
#include <stdio.h>
#include "ptfun3_c.h"
#include <ch.h>

/* The maximum number of ptfun3_setHandle()
   can be called */
#define MAXNUM   3

/* C functions to replace the Ch one */
static ChInterp_t interp;
static void (*ptfun3_chdl_funarg[MAXNUM])(float, int);

/* function pointers to save Ch function pointers */
static void *ptfun3_chdl_funptr[MAXNUM];

/* index of pointer pairs, from 0 to MAXNUM-1 */
static int index = 0;

EXPORTCH int ptfun3_setHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle handle_ch, handle_c;
  int retval;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);
  handle_ch = Ch_VaArg(interp, ap, funcHandle);

  if(handle_ch == NULL) {
    handle_c = NULL;
  }
  else {
    if(index == MAXNUM) {
      printf("Error: Only %d function pointers can be set\n", MAXNUM);
      return -1;
    }
    ptfun3_chdl_funptr[index] = (void *)handle_ch;
    handle_c = ptfun3_chdl_funarg[index];
    index++;
  }

  retval = ptfun3_setHandle(id, handle_c);

  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH funcHandle ptfun3_getHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);
```

```
  retval_c = ptfun3_getHandle(id);

  /* replace C pointer with corresponding Ch function pointer */
  if(retval_c == NULL)  {
    retval_ch = NULL;
  }
  else {
    for(i = 0; i < index; i++) {
      if(ptfun3_chdl_funarg[i] == retval_c) {
        retval_ch = (funcHandle)ptfun3_chdl_funptr[i];
      }
    }
  }
  Ch_VaEnd(interp, ap);
  return retval_ch;
}

/**** hundreds of functions can be here ****/

static void ptfun3_chdl_funarg0(float f, int i) {
  Ch_CallFuncByAddr(interp, ptfun3_chdl_funptr[0], NULL, f, i);
}

static void ptfun3_chdl_funarg1(float f, int i) {
  Ch_CallFuncByAddr(interp, ptfun3_chdl_funptr[1], NULL, f, i);
}

static void ptfun3_chdl_funarg2(float f, int i) {
  Ch_CallFuncByAddr(interp, ptfun3_chdl_funptr[2], NULL, f, i);
}

/********* initialization routine called in ptfun3.h **********/
EXPORTCH int ptfun3_setgetHandles_init() {
  /* from 0 to MAXNUM -1 */

   ptfun3_chdl_funarg[0] = ptfun3_chdl_funarg0;
   ptfun3_chdl_funarg[1] = ptfun3_chdl_funarg1;
   ptfun3_chdl_funarg[2] = ptfun3_chdl_funarg2;

   return 0;
}
```

Listing 5 — C function to be handled (`ptfun3.c`)

```
#include <stdio.h>
#include "ptfun3_c.h"

#define MAXNUM 500

/* keep function handles in C space */
struct FuncList{
        int num; /* id of the function handle */
        funcHandle handle;
} funcList[MAXNUM];

static int index_c = 0;   /* index of funList */
int ptfun3_setHandle(int num, funcHandle funptr)  {
  int i;

  if(index_c == MAXNUM) {
```

```
    printf("Error: too many functions");
    return -1;
    }

  /* make sure that one id is mapped to only one handle */
  for(i=0; i<index_c; i++)
    if(funcList[i].num == num) {
      funcList[i].num = num;
      funcList[i].handle = funptr;
      return num;
    }
  funcList[index_c].num = num;
  funcList[index_c++].handle = funptr;

  /* Call Ch function from C space */
  printf("Calling Ch function from C space\n");
  funptr(1.1, 1);

  return num;
}

funcHandle ptfun3_getHandle(int num)  {
  int i;

  for(i=0; i<index_c; i++)
    if(funcList[i].num == num)
      return funcList[i].handle;
  return NULL;
}
```

Listing 6 — Application functions (`ptfun3.ch`)

```
#include "ptfun3.h"

void ptfun3_1(float f, int i) {
  printf("ptfun3_1() in ptfun3.ch is called, f = %f, i = %d\n", f, i);
  return;
}

void ptfun3_2(float f, int i) {
  printf("ptfun3_2() in ptfun3.ch is called, f = %f, i = %d\n", f, i);
  return;
}

int main() {
  int id;
  funcHandle handle;

  /* set function handle */
  ptfun3_setHandle(1, ptfun3_1);
  ptfun3_setHandle(2, ptfun3_2);

  /* get function handle */
  handle = ptfun3_getHandle(1);
  handle(1.1, 1);
  handle = ptfun3_getHandle(2);
  handle(2.2, 2);

  ptfun3_setHandle(2, ptfun3_1);
  ptfun3_getHandle(2)(1.1, 1);
```

```
  return 0;
}
```

Output

```
Calling Ch function from C space
ptfun3_1() in ptfun3.ch is called, f = 1.100000, i = 1
Calling Ch function from C space
ptfun3_2() in ptfun3.ch is called, f = 1.100000, i = 1
ptfun3_1() in ptfun3.ch is called, f = 1.100000, i = 1
ptfun3_2() in ptfun3.ch is called, f = 2.200000, i = 2
ptfun3_1() in ptfun3.ch is called, f = 1.100000, i = 1
```

6.2.4 Pointer to Function with Return Value and Arguments

This section describes how to deal with functions that return pointers to functions that have return values and arguments. We assume that the function pointer returned by the function `getFunction4`() below is set by function `setFunction4`() which is presented in section 6.1.4.

```
typedef return_type (*funcHandle)(data_type2)
funcHandle getFunction4(data_type1 arg1) {
  ...
}
```

Notice that `funcHandle` is a pointer to a function that has both return value and argument. So, the definition of `funcHandle` in this example is a combination of `getFunction2`() and `getFunction3`().

Program 6.27 has the Ch function `getFunction4`(), `funcHandle` in this example is typedefed as a pointer to function which has both argument and return value. This is the only difference between Program 6.27 and other Ch function files such as Programs 5.5, 6.23 and 6.25.

Program 6.28 has the chdl function. It is written according to Program 6.10, 6.24 and 6.26. The only difference among these files is that `funcHandle` in this template is typedefed as a pointer to function which has both an argument and a return value.

Example

In this example, functions `ptfun4_setHandle`() and `ptfun4_getHandle` illustrates the case of setting and getting pointers to functions that have both return value and argument.

Listing 1 — header file for Ch functions (`ptfun4.h`)

```
#ifndef _PTFUN4_H_
#define _PTFUN4_H_

typedef int(*funcHandle)(float, int);
int ptfun4_setHandle(int i, funcHandle);
funcHandle ptfun4_getHandle(int i);

/* below is added only for Ch */
#include <dlfcn.h>
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
}
```

```
#include<dlfcn.h>

return_type (* getFunction4(data_type1 arg1))(data_type2) {
  void *dlhandle, *fptr;
  return_type (*retval)(data_type2);

  /* load the dynamically loaded library */
  dlhandle = dlopen("libproject.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    fprintf(_stderr, "Error: %s(): dlopen(): %s\n",
                     __func__, dlerror());
    return FAIL_VALUE; /* FAIL_VALUE is typically NULL for point
                          and negative value for integral type */
  }

  /* get the address by function name */
  fptr = dlsym(dlhandle, "getFunction4_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  /* call the chdl function in dynamically loaded library by address
     and pass arguments to C address space */
  dlrunfun(fptr, &retval, getFunction4, arg1);

  /* close the dynamically loaded library */
  if(dlclose(dlhandle)!=0) {
    fprintf(_stderr, "Error: %s(): dlclose(): %s\n",
                     __func__, dlerror());
    return FAIL_VALUE;
  }

  return retval;
}
```

Program 6.27: Returning pointer to function with return value and arguments (chf file).

```
#include<ch.h>
#include<stdio.h>
/* ... */

/* define function pointer type with argument of data_type2
   and return value of return_type */
typedef return_type (*funcHandle)(data_type2);

static ChInterp_t interp;
/* C function to replace the Ch function pointer */
static return_type setFunction4_chdl_funarg(data_type2 arg2);

/* save the function pointer from the Ch space */
static void *setFunction4_chdl_funptr;

EXPORTCH void setFunction4_chdl(void *varg) {
  ChVaList_t ap;
  data_type1 arg1;
  funcHandle handle_ch, handle_c = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get and save function pointer from Ch space  */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  setFunction4_chdl_funptr = (void *)handle_ch;

  /* replace the Ch function pointer with the C one,
     the NULL pointer can't be replaced */
  if(handle_ch != NULL) {
    handle_c = (funcHandle)setFunction4_chdl_funarg;
  }

  setFunction4(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

static return_type setFunction4_chdl_funarg(data_type2 arg2) {
  return_type retval;

  /* Call Ch function by its address which has argument arg2,
     and return a value */
  Ch_CallFuncByAddr(interp, setFunction4_chdl_funptr, &retval, arg2);

  return retval;
}
```

Program 6.28: Returning pointer to function with return value and arguments (chdl file).

```
EXPORTCH funcHandle getFunction4_chdl(void *varg) {
  ChVaList_t ap;

  data_type1 arg1;
  funcHandle  retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */

  /* get the C function pointer which is set by setFunction4_chdl() */
  retval_c = getFunction4(arg1);

  /* replace the C pointer with the Ch one */
  if(retval_c == setFunction4_chdl_funarg) { /* check the consistency */
    retval_ch = (funcHandle)setFunction4_chdl_funptr;
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}
```

Program 6.28: Returning pointer to function with return value and arguments (chdl file) (Contd.).

```
/* initialize and set static function pointer array */
void *_fptr = dlsym(_Chptfun_handle,
                    "ptfun4_setgetHandles_init");
if(_fptr == NULL)  {
   printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
   return -1;
}
dlrunfun(_fptr, NULL, NULL);

/* to close dl */
void _dlclose_ptfun4(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun4);

#pragma importf <ptfun4.chf>
#endif /* _PTFUN4_H_ */
```

Listing 2 — Ch function file (`ptfun4.chf`)

```
#include <dlfcn.h>

int ptfun4_setHandle(int i, funcHandle funptr)  {
  void *fptr;
  int retval;

  /* ldd is loaded in ptfun4.h */
  fptr = dlsym(_Chptfun_handle, "ptfun4_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }
```

```
  dlrunfun(fptr, &retval, ptfun4_setHandle, i, funptr);
  return retval;
}

funcHandle ptfun4_getHandle(int i)  {
//int(*)(float, int) ptfun4_getHandle(int i)  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun4_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, ptfun4_getHandle, i);
  return retval;
}
```

Listing 3 — header file for C functions (`ptfun4_c.h`)

```
#ifndef _PTFUN4_H_
#define _PTFUN4_H_

typedef int(*funcHandle)(float, int);
int ptfun4_setHandle(int i, funcHandle);
funcHandle ptfun4_getHandle(int i);

#endif /* _PTFUN4_H_ */
```

Listing 5 — chdl file (`ptfun4_chdl.c`)

```
#include <stdio.h>
#include "ptfun4_c.h"
#include <ch.h>

/* The maximum number of ptfun4_setHandle()
   can be called */
#define MAXNUM   3

/* C functions to replace the Ch one */
static ChInterp_t interp;
static int (*ptfun4_chdl_funarg[MAXNUM])(float, int );

/* function pointers to save Ch function pointers */
static void *ptfun4_chdl_funptr[MAXNUM];

/* index of pointer pairs, from 0 to MAXNUM-1 */
static int index = 0;

EXPORTCH int ptfun4_setHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle handle_ch, handle_c;
  int retval;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);      /* get id of function handle */
```

```
  handle_ch = Ch_VaArg(interp, ap, funcHandle); /* get ch function pointer */

  if(handle_ch == NULL) {
    handle_c = NULL;
  }
  else {
    if(index == MAXNUM) {
      printf("Error: Only %d function pointers can be set\n", MAXNUM);
      return -1;
    }
    ptfun4_chdl_funptr[index] = (void *)handle_ch;
    handle_c = (funcHandle)ptfun4_chdl_funarg[index];
    index++;
  }

  retval = ptfun4_setHandle(id, handle_c);

  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH funcHandle ptfun4_getHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle retval_c = NULL, retval_ch = NULL;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);         /* get 1st arg  */

  retval_c = ptfun4_getHandle(id);

  /* replace C pointer with corresponding Ch function pointer */
  if(retval_c == NULL)  {
    retval_ch = NULL;
  }
  else {
    for(i = 0; i < index; i++) {
      if(ptfun4_chdl_funarg[i] == retval_c) {
        retval_ch = (funcHandle)ptfun4_chdl_funptr[i];
      }
    }
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}

/**** hundreds of functions can be here ****/

static int ptfun4_chdl_funarg0(float f, int j) {
  int retval;

  Ch_CallFuncByAddr(interp, ptfun4_chdl_funptr[0], &retval, f, j);
  return retval;
}

static int ptfun4_chdl_funarg1(float f, int j) {
  int retval;
```

```
  Ch_CallFuncByAddr(interp, ptfun4_chdl_funptr[1], &retval, f, j);
  return retval;
}

static int ptfun4_chdl_funarg2(float f, int j) {
  int retval;

  Ch_CallFuncByAddr(interp, ptfun4_chdl_funptr[2], &retval, f, j);
  return retval;
}

/****initialization routine called in ptfun4.h **********/
EXPORTCH int ptfun4_setgetHandles_init() {
  /* from 0 to MAXNUM -1 */

   ptfun4_chdl_funarg[0] = ptfun4_chdl_funarg0;
   ptfun4_chdl_funarg[1] = ptfun4_chdl_funarg1;
   ptfun4_chdl_funarg[2] = ptfun4_chdl_funarg2;

   return 0;
}
```

Listing 4 — C functions to be handled (`ptfun4.c`)

```
#include <stdio.h>
#include "ptfun4_c.h"

#define MAXNUM 500

/* keep function handles in c space */
struct FuncList{
        int num; /* id of the function handle */
        funcHandle handle;
} funcList[MAXNUM];

static int index_c = 0;   /* index of funList */
int ptfun4_setHandle(int num, funcHandle funptr)  {
  int i;

  if(index_c == MAXNUM) {
    printf("Error: too many functions");
    exit(1);
    }

  /* make sure that one id is mapped to only one handle */
  for(i=0; i<index_c; i++)
    if(funcList[i].num == num) {
      funcList[i].num = num;
      funcList[i++].handle = funptr;
      return num;
    }
  funcList[index_c].num = num;
  funcList[index_c++].handle = funptr;

  /* Call Ch function from C space */
  printf("Calling Ch function from C space: return value is %d\n",
          funptr(1.1, 1));

  return num;
```

```
}

funcHandle ptfun4_getHandle(int num)  {
  int i;

  for(i=0; i<index_c; i++)
    if(funcList[i].num == num)
      return funcList[i++].handle;
  return NULL;
}
```

Listing 6 — Application (`ptfun4.ch`)

```
#include "ptfun4.h"

int ptfun4_1(float f, int j) {
  printf("f in ptfun4_1() in ptfun4.ch  = %f\n", f);
  printf("j in ptfun4_1() in ptfun4.ch  = %d\n", j);
  printf("ptfun4_1() in ptfun4.ch is called \n");
  return 10;
}

int ptfun4_2(float f, int j) {
  printf("f in ptfun4_2() in ptfun4.ch  = %f\n", f);
  printf("j in ptfun4_2() in ptfun4.ch  = %d\n", j);
  printf("ptfun4_2() in ptfun4.ch is called \n");
  return 20;
}

int main() {
  int id;
  funcHandle handle;

  /* set function handle */
  ptfun4_setHandle(1, ptfun4_1);
  ptfun4_setHandle(2, ptfun4_2);

  /* get function handle */
  handle = ptfun4_getHandle(1);
  printf("retval of id1 = %d\n", handle(1.1, 1));
  handle = ptfun4_getHandle(2);
  printf("retval of id2 = %d\n", handle(2.2, 2));

  ptfun4_setHandle(2, ptfun4_1);
  printf("retval of id2 = %d\n", ptfun4_getHandle(2)(1.1, 1));

  return 0;
}
```

Output

```
f in ptfun4_1() in ptfun4.ch  = 1.100000
j in ptfun4_1() in ptfun4.ch  = 1
ptfun4_1() in ptfun4.ch is called
Calling Ch function from C space: return value is 10
f in ptfun4_2() in ptfun4.ch  = 1.100000
j in ptfun4_2() in ptfun4.ch  = 1
ptfun4_2() in ptfun4.ch is called
Calling Ch function from C space: return value is 20
f in ptfun4_1() in ptfun4.ch  = 1.100000
```

```
/* Header file for cases of returning a system default function */

/* This is an empty function. Only its address is used */
return_type _Ch_PROC_Default(data_type2) {
  return_type retval;
  return retval;
}

/* if the system default function has no return value,
   the fake Ch function should look like
void _Ch_PROC_Default(data_type2) {
}
*/

/* ... */
```

Program 6.29: Header File for Returning Default Function.

```
j in ptfun4_1() in ptfun4.ch  = 1
ptfun4_1() in ptfun4.ch is called
retval of id1 = 10
f in ptfun4_2() in ptfun4.ch  = 2.200000
j in ptfun4_2() in ptfun4.ch  = 2
ptfun4_2() in ptfun4.ch is called
retval of id2 = 20
f in ptfun4_1() in ptfun4.ch  = 1.100000
j in ptfun4_1() in ptfun4.ch  = 1
ptfun4_1() in ptfun4.ch is called
retval of id2 = 10
```

6.3 Special Cases

6.3.1 Function with Return Value of Pointer to Default System Function

In the previous section, we described how to deal with the functions setting a function pointer or getting a function pointer. These function pointers are supposed to be defined in Ch space. In other words, the functions to be set are user-defined functions in Ch space when an application runs. Often times, one may encounter situations that may need to get the function pointer in Ch space from the C space, pass back the function pointer to C space from Ch space later, and then have it invoked in C space. Usually these pointer functions are system default functions in C space rather than user-defined functions in Ch space. They need to be handled specifically in chdl functions.

Based on Program 6.28, we add a function `getFunction5` which returns a pointer to a default system function. In the function `getFunction1_chdl`() in Program 6.22, if a pointer to a Ch function is not found to map the C function pointer returned by function `getFunction1`(), **NULL** will be returned to Ch space. This is because we assume that the pointer returned by function `getFunction1`() has been set by function `setFunction1`() in chdl function `setFunction1_chdl`() and should be identical to the pointer `setFunction1_chdl_funarg`. However, the C function returned is assumed to be a system function in this case, and it couldn't be identical to any user-defined C pointer, such as `setFunction1_chdl_funarg`. Therefore, we can't find an existing Ch function to correspond to it. For getting and setting a function pointer of this kind in Ch space, the fake Ch function `_Ch_PROC_Default`, which has the same prototype as the system function, is defined in the Ch header file Program 6.29. This

```
/* keep Ch and C function pointer pair */
struct HandlePairs{
        funcHandle handle_c;
        funcHandle handle_ch;
} handlePairs;

typedef return_type (*funcHandle)(data_type2);
static ChInterp_t interp;
static return_type setFunction5_chdl_funarg(data_type2 arg2);
EXPORTCH funcHandle getFunction5_chdl(void *varg) {
  ChVaList_t ap;

  datatype1 arg1;
  funcHandle handle_c = NULL, handle_ch = NULL;

  /* get arguments passed from the Ch function */
  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, int);

  handle_c = getFunction5(arg1);

  /* a NULL pointer is returned if handle_c is NULL*/
  if(handle_c == NULL)  {
    handle_ch = NULL;
  }

  /* this c function has a ch function to map */
  else if(handlePairs.handle_c == handle_c);
    handle_ch = handlePairs.handle_ch;

  /* replace the C pointer with the pointer to Ch fake function */
  else {
    handlePairs.handle_c = handle_c;
    handle_ch = handlePairs.handle_ch
              = (funcHandle *)Ch_SymbolAddrByName(interp, "_Ch_PROC_Default");
  }

  Ch_VaEnd(interp, ap);
  return handle_ch;
}
```

Program 6.30: Returning pointer to system default function (chdl file).

```
EXPORTCH void setFunction5_chdl(void *varg) {
  ChVaList_t ap;

  data_type1 arg1;
  funcHandle  handle_ch = NULL, handle_c = NULL;

  /* get arguments passed from the Ch function */
  Ch_VaStart(interp, ap, varg);
  arg1 = Ch_VaArg(interp, ap, data_type1);   /* get 1st argument  */
  /* get function handle of Ch space  */
  handle_ch = Ch_VaArg(interp, ap, funcHandle);

  if(handle_ch == NULL)  {
    handle_c = NULL;
  }

  else if(handlePairs.handle_ch == handle_ch) {
    handle_c = handlePairs.handle_c;
  }

  else {
    setFunction5_chdl_funptr = (void *)handle_ch;
    handlePairs.handle_ch = handle_ch;
    handle_c = handlePairs.handle_c = setFunction5_chdl_funarg;
  }

  setFunction5(arg1, handle_c);

  Ch_VaEnd(interp, ap);
}

/* C function to replace the Ch function pointer */
static return_type setFunction5_chdl_funarg(data_type2 arg2) {
  return_type retval;
  Ch_CallFuncByAddr(interp, setFunction5_chdl_funptr, &retval, arg2);
  return retval;
}
```

Program 6.30: Returning pointer to system default function (chdl file) (Contd.).

function is an empty function except for the return statement if the system function has return value. When the header file is included by an application, this function is also loaded into Ch space. In chdl function `getFunction5_chdl`(), function **Ch_SymbolAddrByName**() can be used to get the address of the fake Ch function. More information about function **Ch_SymbolAddrByName**() can be found in Appendix A.

In Program 6.30, a static variable `handlePairs` of struct `HandlePairs` can save the Ch and C function pointer pair. In function `getFunction5_chdl`(), if the C function returned by `getFunction5`() has a corresponding Ch function pointer in `handlePairs`, the Ch function pointer is returned. This corresponding Ch function pointer could point to either a fake Ch function, or a user-defined Ch function. Otherwise, the pointer to fake Ch function `_Ch_PROC_Default` is assigned to this C function and returned. At the same time, this function pointer pair is saved in `handlePairs`.

Similarly, in function `setFunction5_chdl`(), if the Ch function pointer passed from Ch space has a corresponding C function pointer in `handlePairs`, the C function pointer is used. This corresponding C function pointer could point to either a default system function, or function `setFunction5_chdl_funarg`(). Otherwise, the pointer to function `setFunction5_chdl_funarg`() is assigned to replace this Ch function. This function pair is saved in `handlePairs`.

Example

In this example, functions `ptfun5_setHandle`() and `ptfun5_getHandle`() are used to illustrate the case of setting and getting system default functions.

Listing 1 — header file for Ch functions (`ptfun5.h`)

```
#ifndef _PTFUN5_H_
#define _PTFUN5_H_

typedef int(*funcHandle)(float, int);

int def_func1(float f, int i);
int def_func2(float f, int i);
int def_func3(float f, int i);

int ptfun5_setHandle(funcHandle);
funcHandle ptfun5_getHandle(int i);
void ptfun5_call_it();


/* added for Ch */

#include <dlfcn.h>
void *_Chptfun_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chptfun_handle== NULL) {
   printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
   exit(-1);
}

/* initialize and set static function pointer array */
void *_fptr = dlsym(_Chptfun_handle, "ptfun5_setgetHandles_init");
if(_fptr == NULL)  {
   printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
   exit(-1);
}
dlrunfun(_fptr, NULL, NULL);
```

```
/* to close dl */
void _dlclose_ptfun5(void) {
   dlclose(_Chptfun_handle);
}
atexit(_dlclose_ptfun5);

#pragma importf "ChProcDefault.chf"
#pragma importf "ptfun5.chf"

#endif /* _PTFUN5_H_ */
```

For multiple default functions, we define `Ch_PROC_Default` as an array of pointers to function. Function `ptfun5_call_it`() can be used to invoke the the default system function.

Listing 2 — definitions of fake Ch functions (`ChProcDefault.chf`)

```
#define MAXNUM_FUN 3

int (*_Ch_PROC_Default[MAXNUM_FUN])(float, int);

int _Ch_PROC_Default0(float f, int i) {return 0;}
int _Ch_PROC_Default1(float f, int i) {return 0;}
int _Ch_PROC_Default2(float f, int i) {return 0;}

_Ch_PROC_Default[0] = _Ch_PROC_Default0;
_Ch_PROC_Default[1] = _Ch_PROC_Default1;
_Ch_PROC_Default[2] = _Ch_PROC_Default2;
```

This file is included in the header file `ptfun5.h` with code `#pragma importf "ChProcDefault.chf"`.

Listing 3 — function file (`ptfun5.chf`)

```
#include <dlfcn.h>

int ptfun5_setHandle(funcHandle funptr)  {
  void *fptr;
  int retval;

  /* dl is loaded in ptfun5.h */
  fptr = dlsym(_Chptfun_handle, "ptfun5_setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, ptfun5_setHandle, funptr);
  return retval;
}

funcHandle ptfun5_getHandle(int i)  {
  void *fptr;
  funcHandle retval;

  fptr = dlsym(_Chptfun_handle, "ptfun5_getHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return NULL;
  }
```

```
  dlrunfun(fptr, &retval, ptfun5_getHandle, i);
  return retval;
}

void ptfun5_call_it()  {
  void *fptr;

  fptr = dlsym(_Chptfun_handle, "ptfun5_call_it_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, ptfun5_call_it);
  return;
}
```

Listing 4 — header file for C functions (`ptfun5.h`)

```
#ifndef _PTFUN5_H_
#define _PTFUN5_H_

typedef int(*funcHandle)(float, int);

int def_func1(float f, int i);
int def_func2(float f, int i);
int def_func3(float f, int i);

int ptfun5_setHandle(funcHandle);
funcHandle ptfun5_getHandle(int i);
void ptfun5_call_it();

#endif /* _PTFUN5_H_ */
```

Listing 5 — chdl file (`ptfun5_chdl.c`)

```
#include <stdio.h>
#include "ptfun5_c.h"
#include <ch.h>

/* The maximum number of pointers can be get */
#define MAXNUM   3

/* keep handle pairs in C and Ch space
   can use dynamic memory allocation */
struct HandlePairs{
        funcHandle handle_c;
        funcHandle handle_ch;
} handlePairs[MAXNUM];

/* C functions to replace the Ch one */
static ChInterp_t interp;
static int (*ptfun5_chdl_funarg[MAXNUM])(float, int);

/* function pointers to save Ch function pointers */
static void *ptfun5_chdl_funptr[MAXNUM];

/* index of funarg and funptr, from 0 to MAXNUM-1 */
```

```
static int index = 0;

EXPORTCH int ptfun5_setHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  funcHandle handle_ch, handle_c;
  int retval;

  Ch_VaStart(interp, ap, varg);
  handle_ch = Ch_VaArg(interp, ap, funcHandle);

  if(handle_ch == NULL) {
    handle_c = NULL;
  }
  else {
    /* make sure that one ch handle is mapped to only one c handle */
    for(i = 0; i < index; i++) /* this ch function has been set */
      if(handlePairs[i].handle_ch == handle_ch) {
        handle_c = handlePairs[i].handle_c;
        break;
      }
    if(i == index) { /* has never been set */
      handlePairs[index].handle_ch = handle_ch;
      ptfun5_chdl_funptr[index] = (void *)handle_ch;
      handle_c = handlePairs[index].handle_c = ptfun5_chdl_funarg[index];
      index++;
    }
  }

  retval = ptfun5_setHandle(handle_c);
  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH funcHandle ptfun5_getHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int id;
  funcHandle retval_c = NULL, retval_ch = NULL;
  funcHandle *p = NULL;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);

  retval_c = ptfun5_getHandle(id);

  if(retval_c == NULL)  {
    Ch_VaEnd(interp, ap);
    return NULL;
  }

  for(i=0; i<index; i++)
    if(handlePairs[i].handle_c == retval_c) {
      retval_ch = handlePairs[i].handle_ch;
      break;
    }

  if(i == index) {
    handlePairs[index].handle_c = retval_c;
```

```
    p = (funcHandle *)Ch_SymbolAddrByName(interp, "_Ch_PROC_Default");
    retval_ch = handlePairs[index].handle_ch = p[index];
    index++;
  }

  Ch_VaEnd(interp, ap);
  return retval_ch;
}

EXPORTCH void ptfun5_call_it_chdl(){

  ptfun5_call_it();
  return;
}

/**** C functions to replace Ch callback functions ****/

static int ptfun5_chdl_funarg0(float f, int i) {
  int retval;
  Ch_CallFuncByAddr(interp, ptfun5_chdl_funptr[0], &retval, f, i);
  return retval;
}

static int ptfun5_chdl_funarg1(float f, int i) {
  int retval;
  Ch_CallFuncByAddr(interp, ptfun5_chdl_funptr[1], &retval, f, i);
  return retval;
}

static int ptfun5_chdl_funarg2(float f, int i) {
  int retval;
  Ch_CallFuncByAddr(interp, ptfun5_chdl_funptr[2], &retval, f, i);
  return retval;
}

/********* initialization routine called in ptfun5.h **********/
EXPORTCH int ptfun5_setgetHandles_init() {
  /* from 0 to MAXNUM -1 */

   ptfun5_chdl_funarg[0] = ptfun5_chdl_funarg0;
   ptfun5_chdl_funarg[1] = ptfun5_chdl_funarg1;
   ptfun5_chdl_funarg[2] = ptfun5_chdl_funarg2;

   return 0;
}
```

Listing 6 — Application (`ptfun5.ch`)

```
#include "ptfun5.h"

int main() {
  int id;
  int retval;
  funcHandle  def_handle;

  def_handle =  ptfun5_getHandle(1);
  ptfun5_setHandle(def_handle);
  ptfun5_call_it();

  def_handle =  ptfun5_getHandle(2);
```

```
  ptfun5_setHandle(def_handle);
  ptfun5_call_it();

  return 0;
}
```

Listing 7 — C functions to be handled (`ptfun5.c`)

```
#include <stdio.h>
#include "ptfun5_c.h"

funcHandle def_handle  = NULL;

int ptfun5_setHandle(funcHandle funptr)  {
  def_handle = funptr;
  /* Call Ch function from C space */
  printf("Calling function from C space: return value is %d\n\n",
         funptr(1.1, 1));
  return 0;
}

funcHandle ptfun5_getHandle(int num)  {
  switch (num) {
    case 1 :
      return def_func1;
    case 2 :
      return def_func2;
    case 3 :
      return def_func3;
    default:
      return NULL;
  }
}

void ptfun5_call_it() {
  if(def_handle != NULL)
    (*def_handle)(1.1, 2);
  else {
    printf("No default C function has been set\n");
  }
}

/* the default callback functions in c space */
int def_func1(float f, int i) {
  printf("in def_func1, f = %f, i = %i\n\n", f, i);
  return 0;
}

int def_func2(float f, int i) {
  printf("in def_func2, f = %f, i = %i\n\n", f, i);
  return 0;
}

int def_func3(float f, int i) {
  printf("in def_func3, f = %f, i = %i\n", f, i);
  return 0;
}
```

Output

```
in def_func1, f = 1.100000, i = 1

Calling function from C space: return value is 0

in def_func1, f = 1.100000, i = 2

in def_func2, f = 1.100000, i = 1

Calling function from C space: return value is 0

in def_func2, f = 1.100000, i = 2
```

6.3.2 Functions with Pointer to Function containing Argument of Pointer to Void

In Section 6.2.3, we have discussed how to handle the functions with pointer to function containing argument. It can be used to handle functions with pointer to function containing argument of pointer to void. However, we provide a different way for handling functions with an argument of pointer to function and the pointer to function has the argument of pointer to void. Consider the function `setHandle()` in Program 6.32 which has prototype of It has the following two characteristics: 1. It takes a function pointer `handle` of type `funcHandle` as an argument, where `funcHandle` is typedefed as a pointer to function which takes a pointer to **void** as the argument.

2. One of arguments of the function `setHandle`(), such as `usrMsg` is a pointer to void which will be passed to the function pointed by function pointer `funptr` when it is called.

The method introduced here only applies to the case that satisfies these two conditions. While in Section 6.2.3, pointer to void won't be passed as an argument in function `setHandle`(). The method introduced here are commonly used to set event or error handlers as callback functions. For example, it can be used to set function pointer `funptr` to deal with the error with error number of `index` which is the first argument of the function. When the system detects an error with number of `index`, the function pointed to by `funptr` is called and the user-defined message `usrMsg` is displayed. The first argument is only an index for different events, and it does not have to be of type int. If only one event to be handled, the first argument is not necessary.

Function `callFun`() in Program 6.32 calls the handler according to argument `index`. The user-defined message is passed to the handler. An array of struct `FuncList` is used internally to save the information of handlers and messages.

Note In most cases, the user can only get shared library or static library for interfacing with Ch. The chdl file and chf file are the main focuses in this document. The C/C++ function files we provide in Program6.32 are just for demos to display the running result even though they can be combined with chdl file for code optimization.

```
#ifndef _PTFUN6_H_
#define _PTFUN6_H_

typedef int(*funcHandle)(void *usrMsg);
int setHandle(int index, funcHandle handle, void * usrMsg);
void callFun(int index);

#endif /* _PTFUN6_H_ */
```

Program 6.31: Setting function with argument of pointer to void (header file ptfun6_c.h in C space).

The Ch functions `setHandle`() and `callFun`() are shown in Program 6.33 where `_Chsethandle_handle` is set in Program 6.34. To handle functions `setHandle`() and `callFun`(), we have chdl functions `setHandle_chdl`() and `callFun_chdl`() in Program 6.35

Comparing Program 6.35 with other chdl function that can set multiple Ch function pointers, you can find there is no array of pointer needed to save the Ch function pointers in the Program6.35. Only one C function `chdl_funarg`() is used to replace the Ch function pointers. This code is more compact.

Inside the function `setHandle_chdl`(), we use `chdl_funarg` , which is a C function, to replace `handle_ch`. Finally, `chdl_funarg` is pass to function `setHandle`(). This is one of the critical points mentioned before. Then the address of the struct `handlePairs[i]` is passed to the function to replace the pointer `usrMsg`. Because `usrMsg` is a pointer to **void**, we need to cast the type of the address to the pointer to **void**.

The struct `handlePairs[i]`, which is passed to replace the pointer `usrMsg`, has three fields. `num` makes sure that each event has only one handler and message. `handle_ch` keeps the function pointer in Ch address space. `usrMsg` is set to the pointer `usrMsg` passed from Ch.

When some errors or events occur, the function `chdl_funarg`() is called with the argument `pHandlePairs`, which is actually a pointer to `handlePairs[i]` set by the function `setHandle`(). Therefore, besides the information of user-defined message, the function pointer in Ch address space is passed. No array of pointers to save the Ch function pointers is needed here.

The application is in Program 6.36. First we set different handlers with different messages to different events and then call them. Then we set the same handlers with different messages to different events. The results is shown in Program 6.37.

```
#include<stdio.h>
#include "ptfun6_c.h"

#define MAXNUM 500
struct FuncList{
        /* id of the function handle, treated as the event id */
        int num;
        /* function handle, regarded as a pointer to the callback function*/
        funcHandle handle;
        /* assume usrMsg is a pointer used by callback function */
        void* usrMsg;
};

/* internal data to keep function handles */
struct FuncList functList[MAXNUM];
static int index_c = 0;   /* index of funList */

int setHandle(int index, funcHandle funptr, void * usrMsg)  {
  int i;

  if(index_c == MAXNUM) {
    printf("Error: too many functions");
    exit(1);
    }

  /* make sure that one id is mapped to only one handle */
  for(i=0; i<index_c; i++)
    if(functList[i].num == index) {
      functList[i].handle = funptr;
      functList[i].usrMsg = usrMsg;
      return index;
    }
  functList[index_c].num = index;
  functList[index_c].handle = funptr;
  functList[index_c++].usrMsg = usrMsg;

  return index;
}

void callFun(int index)  {
  int i;

  for(i=0; i<index_c; i++) {
    if(functList[i].num == index)
       functList[i].handle(functList[i].usrMsg);
  }
  return;
}
```

Program 6.32: Setting function with argument of pointer to void (C functions).

```
#include <dlfcn.h>

int setHandle(int i, funcHandle funptr, void *usrMsg)  {
  void *fptr;
  int retval;

  /* ldd is loaded in .h */
  fptr = dlsym(_Chsethandle_handle, "setHandle_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, setHandle, i, funptr, usrMsg);
  return retval;
}

void callFun(int i)  {
  void *fptr;

  fptr = dlsym(_Chsethandle_handle, "callFun_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, callFun, i);
  return;
}
```

Program 6.33: Setting function with argument of pointer to void (chf functions).

```
#ifndef _PTFUN6_H_
#define _PTFUN6_H_

typedef int(*funcHandle)(void *usrMsg);
int setHandle(int index, funcHandle handle, void * usrMsg);
void callFun(int index);

#include <dlfcn.h>
void *_Chsethandle_handle = dlopen("libptfun.dl", RTLD_LAZY);
if(_Chsethandle_handle== NULL) {
   printf("Error: dlopenn(): %s\n", dlerror());
   exit(-1);
}

/* to close dl */
void _dlclose_sethandle(void) {
   dlclose(_Chsethandle_handle);
}
atexit(_dlclose_sethandle);

#pragma importf "ptfun6.chf"
#endif /* _PTFUN6_H_ */
```

Program 6.34: Setting function with argument of pointer to void (header file ptfun6.h in Ch space).

```
#include <stdio.h>
#include <ch.h>

typedef int(*funcHandle)(void *);
int setHandle(int index, funcHandle funptr, void *usrMsg);
void callFun(int index);
static ChInterp_t interp;
static int chdl_funarg(void *usrMsg);

struct HandlePairs{
        int num;
        funcHandle handle_ch;
        void *usrMsg;
};

/* keep pairs of index, handler and message */
#define MAXNUM_P   3
static struct HandlePairs handlePairs[MAXNUM_P];
static int index_p = 0;   /* index of handlePairs, from 0 to MAXNUM-1 */

EXPORTCH int setHandle_chdl(void *varg){
  int i;
  ChVaList_t ap;
  int index;
  funcHandle handle_ch;
  void *usrMsg;
  int retval;

  Ch_VaStart(interp, ap, varg);
  index = Ch_VaArg(interp, ap, int);
  handle_ch = Ch_VaArg(interp, ap, funcHandle);
  usrMsg = Ch_VaArg(interp, ap, void *);

  if(index_p == MAXNUM_P) {
    printf("Error: too many functions to set, the limit is %d", MAXNUM_P);
    return -1;
  }

  /* make sure that the index is mapped to only one ch handle */
  for(i=0; i<index_p; i++) /* this ch function has been set */
  {
      if(handlePairs[i].num == index) {
          handlePairs[i].handle_ch = handle_ch;
          handlePairs[i].usrMsg = usrMsg;
          break;
       }
  }

  if(i == index_p) { /* has never been set */
    handlePairs[index_p].handle_ch = handle_ch;
    handlePairs[index_p].usrMsg = usrMsg;
    index_p++;
  }

  retval = setHandle(index, chdl_funarg, (void *)&handlePairs[i]);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 6.35: Setting function with argument of pointer to void (chdl functions).

```
EXPORTCH void callFun_chdl(void *varg){
  ChVaList_t ap;
  int id;

  Ch_VaStart(interp, ap, varg);
  id = Ch_VaArg(interp, ap, int);         /* get 1st arg  */

  callFun(id);

  Ch_VaEnd(interp, ap);
  return;
}


static ChInterp_t interp;
static int chdl_funarg(void* pHandlePairs) {
  int retval;

  if (pHandlePairs == NULL) {
    printf("Error: The pHandlePairs pointer is NULL\n");
    retval =  -1;
  }
  else if(((struct HandlePairs *)pHandlePairs)->handle_ch) {
         Ch_CallFuncByAddr(interp,
         (void *)((struct HandlePairs *)pHandlePairs)->handle_ch, &retval,
         ((struct HandlePairs *)pHandlePairs)->usrMsg);
       }

  return retval;
}
```

Program 6.35: Setting function with argument of pointer to void (chdl functions) (Contd.).

```
#include "ptfun6.h"

int setHandle(int i, funcHandle funptr, void *usrMsg);
void callFun(int i);

int fun1(void *usrMsg) {
  printf("in function1() usrMsg is \"%s\"\n", usrMsg);
  return 0;
}

int fun2(void *usrMsg) {
  printf("in function2() usrMsg is \"%s\"\n", usrMsg);
  return 0;
}


int main() {
  int id;
  int retval;

  /* set function handle */
  setHandle(1, fun1, "This is event 1");
  setHandle(2, fun2, "This is event 2");

  /* id1 -> fun: 1, msg: "This is event 1") */
  /* id2 -> fun: 2, msg: "This is event 2") */
  callFun(1);
  callFun(2);

  printf("\n\n");

  setHandle(1, fun1, "This is event 2");
  setHandle(2, fun1, "This is event 2");

  /* id1 -> fun: 1, msg: "This is event 1") */
  /* id2 -> fun: 1, msg: "This is event 2") */
  callFun(1);
  callFun(2);

  return 0;
}
```

Program 6.36: Setting function with argument of pointer to void (Ch Application).

```
in function1() usrMsg is "This is event 1"
in function2() usrMsg is "This is event 2"


in function1() usrMsg is "This is event 1"
in function1() usrMsg is "This is event 2"
```

Program 6.37: Setting function with argument of pointer to void (Output).

Chapter 7

Interfacing Classes and Member Functions in C++

This chapter describes how to interface Ch with classes in C++. The method described in this chapter can only wrap C++ classes whose data members are declared private.

7.1 Class Definition, Constructor and Destructor

Classes in C++ are supported in Ch. It makes the interface to class simple and easy. This section describes how to build the constructor and destructor of a class in the Ch space to interface the corresponding ones in a C++ class in the C++ space. Interface for member functions will be described in the subsequent sections.

```
class Class1 {
  private:
    data_type1  m_1;
  public:
    Class1();
    ~Class1();
};
```

Program 7.1: The class Class1 in the C++ space.

As an example, a C++ class `Class1` in the C++ space with its declaration is shown in Program 7.1. Let's take a look at `Class1` definition in details. `Class1` has a private member `m_1` whose `data_type` is a simple data type as discussed in the previous section. Like any other conventional C++ classes, it has a constructor and a destructor that creates and destroys the instance, respectively.

We have to modify a header file in the C++ space for use in the Ch space. Private variables and often time private member functions need to be removed. The corresponding declaration of `Class1` in the Ch space is given in Program 7.2. The static private member `g_sample_dlhandle` is a handle to load a dynamically loaded library (DLL). The static private member `g_sample_dlcount` is the total number of class instances in the Ch space. It is used to make sure the loaded DLL will be released if there is no class instance in use.

If there is only a single class in a program, in order to avoid name space conflict, these two static private members can be used to handle the dynamically loaded library for interface between Ch and C++. Using these two variables, the program will manage the DLL actively by loading the DLL in a constructor

and releasing it in a destructor. An alternative general method for handling DLL with multiple classes or combination of classes and functions will be presented later in an example in this section and further illustrated in section 7.5.

```
class Class1 {
  private:
    static void *g_sample_dlhandle;
    static int g_sample_dlcount;
  public:
    Class1();
    ~Class1();
};
void * Class1::g_sample_dlhandle = NULL;
int Class1::g_sample_dlcount = 0;
```

Program 7.2: Declaration of Class1 in the Ch space.

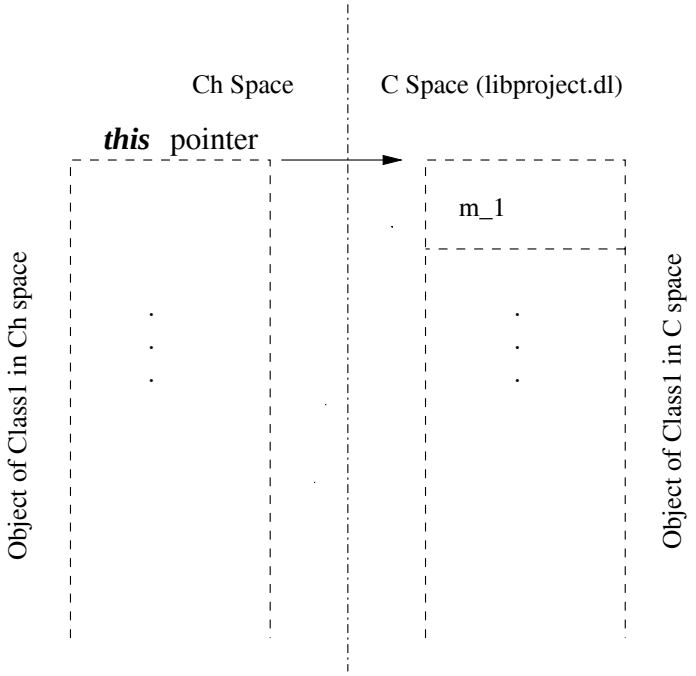


Figure 7.1: Declaration of Class1 in Ch and C Spaces.

The relation of class `Class1` in both Ch and C++ spaces are illustrated in Figure 7.1. An object in the Ch space shares the memory corresponding to the object in the C++ space. Member functions in both C++ and Ch do not occupy additional memory in an instance of class. Static members in a class also do not occupy the memory in each instance of class. Therefore, member functions in the Ch space can be different from the ones in the C++ space.

We will try to make the constructor and destructor work in the Ch space first. To add the constructor `Class1::Class1`() in the Ch space, we need to write both the chf function and chdl function like what we did for the regular functions in previous chapters.

```
#include<dlfcn.h>
#include "class1_ch.h"

Class1::Class1() {
  void *fptr;

  /* Here to load the dynamically loaded library if necessary.
     g_sample_dlhandle is a global pointer pointing to the loaded DLL
     g_sample_dlcount  is a global int counting instances for all classes
     They are declared in the header file class1_ch.h for Ch space.
  */
  if(g_sample_dlhandle == NULL || g_sample_dlcount == 0) {
    g_sample_dlhandle = dlopen("libproject.dl", RTLD_LAZY);
    if(g_sample_dlhandle == NULL) {
       printf("Error: %s(): dlopen(): %s\n", __class_func__, dlerror());
       return;
    }
  }

  /* to get the address by function name */
  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }

  /* to call the chdl function in dynamically loaded
     library by address */
  dlrunfun(fptr, NULL, NULL);
  g_sample_dlcount++; // to increase count of instance
}
```

Program 7.3: Constructor of Class1 (chf file).

The chf function of constructor `Class1::Class1`() is shown in Program 7.3. The variable `g_sample_dlhandle` points to the loaded DLL, and `g_sample_dlcount` counts instances for all classes, so that the loaded DLL will be released if there is no class instance in use. If `g_sample_dlhandle` is NULL or `g_sample_dlcount` is zero, the DLL file `libproject.dl` is loaded and its handle is kept by the variable `g_sample_dlhandle`. The DLL file is closed in the chf function of destructor.

Function **dlrunfun()** will call the interface function `Class1_Class1_chdl()` defined in Program 7.4. It will get the handle of a class instance in the C++ space. The second argument is the address for holding the returned value from calling the binary chdl function. It shall be NULL for constructor of a class. The third argument of function **dlrunfun()** is the name of the member function that is called. If the third argument of function **dlrunfun()** is `Class1`, Ch will check the compatibility of the passed arguments based on the prototype of class `Class1`.

```
#include<ch.h>
EXPORTCH void Class1_Class1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c=new Class1();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
}
```

Program 7.4: Constructor of Class1 (chdl file).

Program 7.4 has the chdl function for the constructor `Class1_Class1_chdl`. This function initializes a new instance of `Class1` in the C++ space and keeps it in the Ch space. The function **Ch_CppChangeThisPointer**() changes the *this* pointer of an instance of a class in the Ch space as shown below.

```
Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
```

The *this* pointer of an instance of a class in the Ch space will point to an instance of class in the C++ space passed as the second argument `c`. The size of the class in the Ch space will also be changed to the size of the class in the C++ space passed as the third argument.

```
#include<dlfcn.h>
#include "class1_ch.h"

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }

  /* call the chdl function in DLL by address */
  dlrunfun(fptr, NULL, NULL, this);
  g_sample_dlcount--; // to decrease count of instance

  if(g_sample_dlcount <= 0 && g_sample_dlhandle != NULL) {
    if(dlclose(g_sample_dlhandle)!=0)
       printf("Error: %s(): dlclose(): %s\n", __class_func__, dlerror());
  }
}
```

Program 7.5: Destructor of Class1 (chf file).

The chf function of destructor `Class1::~Class1`() is shown in Program 7.5. The DLL file loaded in chf function of constructor `Class1::Class1`() is closed in the destructor `Class1::~Class1`(). By calling the chdl function of destructor `Class1_dClass1_chdl`() using function **dlrunfun()**, it deletes the instance in the C++ space by passing the *this* pointer in the C++ space. If there is no more instance in use, the DLL will be closed.

```
#include<ch.h>
EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class1();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}
```

Program 7.6: Destructor of Class1 (chdl file).

The chdl function of destructor `Class1_dClass1_chdl`() is shown in Program 7.6. Contrary to the constructor, this function deletes the instance of `Class1` in the C++ space which is created in chdl function of the constructor `Class1_Class1_chdl`() as shown below.

```
if(Ch_CppIsArrayElement(interp))
    c->~Class1();
else
   delete c;
```

The function **Ch_CppIsArrayElement**() test if the destructor of a class in the Ch space is called by an element of an array of class. If the destructor s called by an element of an array of class in the Ch space, only the destructor in the C++ space will be called. In this case, the memory allocated in the constructor has been released in the Ch kernel beforehand.

Example: A class with constructor and destructor

This is a simple example in which the `Class1` has only constructor and destructor.

For the convenience of development and maintenance, the same header file `sampclass.h` in Listing 1 can be used for both Ch and C++. C++ program `sampclass.cpp` in Listing 2 contains constructor and destructor. In its constructor in the C++ space, the value of member `m_d` is printed out. The application program `script.cpp` in Listing 3 will print out the size of the class in the Ch space. Programs `sampclass.cpp` and `script.cpp` can be compiled using a C++ compiler to create a binary executable program `script.exe` with the output shown in Listing 5 using a Makefile shown in Listing 4. The size of the class `Class1` is the same in both Ch and C++ spaces as shown in Listing 5.

The interface program `sampclass_chdl.cpp` in Listing 6 can be compiled and linked against the C++ program `sampclass.cpp` to create the dynamically loaded library `libsampclass.dl`. Function files and member function definitions in Ch are placed in a separate file `sampclass.chf` in Listing 7. This function file is included in header file `sampclass.h` by a preprocessing directive

```
#pragma importf <sampclass.chf>
```

which will search the program `sampclass.chf` based on the function path `_fpath`.

Listing 1 — the header file `sampclass.h` in the Ch and C++ spaces.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

/********* for Ch space **********/
#ifdef _CH_
#include <dlfcn.h>

class Class1 {
  private:
    static void *g_sample_dlhandle;
    static int g_sample_dlcount;
  public:
    Class1();
    ~Class1();
};
void * Class1::g_sample_dlhandle = NULL;
int Class1::g_sample_dlcount = 0;

#pragma importf <sampclass.chf>
#else
/********* for C++ space **********/
class Class1 {
  private:
    double m_d;
  public:
    Class1();
    ~Class1();
};
#endif

#endif /* _SAMPCLASS_H_ */
```

Listing 2 — the C++ program `sampclass.cpp`

```
#include <stdio.h>
#include "sampclass.h"

Class1::Class1() {
  m_d = 1;
  printf("in Constructor in C++ space, m_d = %f\n", m_d);
}

Class1::~Class1() {
}
```

Listing 3 — The application program `script.cpp` in both Ch and C++ spaces.

```
#include <stdio.h>
#include "sampclass.h"

int main() {
  class Class1 c1 = Class1();
  class Class1 *c2 = new Class1();

  printf("sizeof(class Class1) = %d\n", sizeof(class Class1));
  printf("sizeof(c1) = %d\n", sizeof(c1));
  delete c2;
  return 0;
}
```

Listing 4 — Makefile

```
target: libsampclass.dl script.exe

libsampclass.dl: sampclass.o sampclass_chdl.o
        ch dllink libsampclass.dl cplusplus sampclass.o sampclass_chdl.o

sampclass.o: sampclass.cpp
        ch dlcomp libsampclass.dl cplusplus sampclass.cpp

sampclass_chdl.o: sampclass_chdl.cpp
        ch dlcomp libsampclass.dl cplusplus sampclass_chdl.cpp

script.exe: script.o sampclass.o
        ch dllink script.exe cplusplus script.o sampclass.o

script.o: script.cpp
        ch dlcomp libsampclass.dl cplusplus script.cpp

test:
        script.exe    > test1
        ch script.cpp > test2
        test1 test2

clean:
        rm -f *.o *.dl *.obj *.exp *.lib *.exe test1 test2
```

Listing 5 — Output from executing program `script.cpp`.

```
in Constructor in C++ space, m_d = 1.000000
in Constructor in C++ space, m_d = 1.000000
sizeof(class Class1) = 8
sizeof(c1) = 8
```

Listing 6 — the chdl program `sampclass_chdl.cpp`

```
#include <stdio.h>
#include <ch.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

EXPORTCH void Class1_Class1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  c = new Class1();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
```

```
    c->~Class1();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}
```

Listing 7 — the Ch function file `sampclass.chf`

```
/****** member functions of Class1 ******/
Class1::Class1(){
  void *fptr;

  if(g_sample_dlhandle == NULL || g_sample_dlcount == 0) {
    g_sample_dlhandle = dlopen("libsampclass.dl", RTLD_LAZY);
    if(g_sample_dlhandle == NULL) {
       printf("Error: %s(): dlopen(): %s\n", __class_func__, dlerror());
       return;
    }
  }
  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }

  dlrunfun(fptr, NULL, NULL);
  g_sample_dlcount++; // to increase count of instance
  return;
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, NULL, this);
  g_sample_dlcount--; // to decrease count of instance

  if(g_sample_dlcount <= 0 && g_sample_dlhandle != NULL) {
    if(dlclose(g_sample_dlhandle)!=0)
       printf("Error: %s(): dlclose(): %s\n", __class_func__, dlerror());
  }
  return;
}
```

If the DLL is opened in a constructor and closed in a destructor, a count is needed for keeping the number of instance of the class. Using two static private members to handle DLL as illustrated in the above example works only for a single class. If there are multiple classes in a dynamically loaded library, a global variable should be used for the DLL handle, which will be illustrated in section 7.5.

If a dynamically loaded library `libsampclass.dl` contains both C++ classes and C functions, it can be loaded when the program is executed and released when the program exits. This can be accomplished in a header file. In this case, the DLL will be loaded once at the beginning of the program execution by calling **dlopen()** using the global variable `g_sampl_dlhandle`. It is closed when the program exits. Therefore, the variable `g_sample_dlcount` for counting the number of instantiation of the class can be eliminated.

The updated header file `sampclass.h` is given in Listing 8 below. The DLL is opened at the program execution and closed by function `_dlclose_sampclass()` which is registered to run at the exit of the program. The corresponding function file `sampclass.chf` in Listing 9 is similar to that in Listing 7, except that constructor and destructor have been changed.

Listing 8 — the Ch/C++ header file `sampclass.h`.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

/********* for Ch space **********/
#ifdef _CH_
#include <dlfcn.h>
void *g_sample_dlhandle = dlopen("libsampclass.dl", RTLD_LAZY);
if(g_sample_dlhandle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get g_sample_dlhandle in sampclass.h\n");
   exit(-1);
}
void _dlclose_sampclass(void) {
   dlclose(g_sample_dlhandle);
}
atexit(_dlclose_sampclass);
#endif

class Class1 {
#ifndef _CH_
/********* for C++ space **********/
  private:
    double m_d;
#endif
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};

#pragma importf <sampclass.chf>

#endif /* _SAMPCLASS_H_ */
```

Listing 9 — the Ch function file `sampclass.chf`.

```
/****** member functions of Class1 ******/

Class1::Class1(){
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, _dlerror());
    return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
```

```
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }

  dlrunfun(fptr, NULL, NULL, this);
  return;
}
```

7.2 Member Functions without Return Value and Argument

This section discusses interface class member functions without return value and argument.

As an example, a C++ class `Class1` with member functions in the C++ space with its declaration is shown in Program 7.7. An alternative form without constructor and destructor is shown in Program 7.8. Its public member functions are `memfun1()`, `memfun2()`, and `memfun3()`. Note that `memfun1()`, `memfun2()`, and `memfun3()` have the same forms as the ones we already discussed in Chapter 5. However, they will be handled differently since they are member functions of class instead of regular functions. The corresponding class declaration in the Ch space for the class either in Program 7.7 or Program 7.8 is given in Program 7.9. In this section, we'll present sample chdl and chf function files for member function `memfun1()` without return value and argument The other two member functions will be handled in the subsequent sections.

```
class Class1 {
  private:
    data_type1  m_1;
  public:
    Class1();
    ~Class1();
    void memfun1();
    void memfun2(data_type2 arg2);
    return_type1 memfun3();
};
```

Program 7.7: The class Class1 with member functions in the C++ space.

```
class Class1 {
  private:
    data_type1  m_1;
  public:
    void memfun1();
    void memfun2(data_type2 arg2);
    return_type1 memfun3();
};
```

Program 7.8: The class Class1 with member functions in the C++ space. It has no constructor and destructor.

```
void *g_sample_dlhandle;
int g_sample_dlcount;

class Class1 {
  public:
    Class1();
    ~Class1();
    void memfun1();
    void memfun2(datatype2 arg2);
    return_type1 memfun3();
};
```

Program 7.9: The class Class1 with member functions in the Ch space.

The definition of member function `Class1::memfun1`() in the Ch space is shown in Program 7.10, This function file is very similar to the ones in the previous two chapters. Note that the *this* pointer of the instance of the class is passed to the C++ space as an argument of function **dlrunfun()** even though `memfun1()` has no argument and return value. This is because we are trying to call a member function, `memfun1()`, of a specific class instance. In order to do this, we have to invoke this function through the pointer that points to that instance in the C++ space. The *this* pointer gets its value when the constructor is called.

```
void Class1::memfun1() {
  void *fptr;

  /* to get the address by function name */
  fptr = dlsym(g_sample_dlhandle, "Class1_memfun1_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }

  /* to call the chdl function in dynamically loaded library by address */
  dlrunfun(fptr, NULL, memfun1, this);
  return;
}
```

Program 7.10: Member function without return value or argument (chf file).

```
EXPORTCH void Class1_memfun1_chdl(void *varg){
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);

  c->memfun1();

  Ch_VaEnd(interp, ap);
  return;
}
```

Program 7.11: Member function without return value or argument (chdl file).

In Program 7.11, `Class1_memfun1_chdl`() is a chdl function, for `Class1::memfun1`(). `Class1_memfun1_chdl`() is actually not a member function of a class. It retrieves one argument from the Ch space, namely *this* pointer, and stores it in `c`. Then it calls the binary C++ member function `memfun1()` through that pointer.

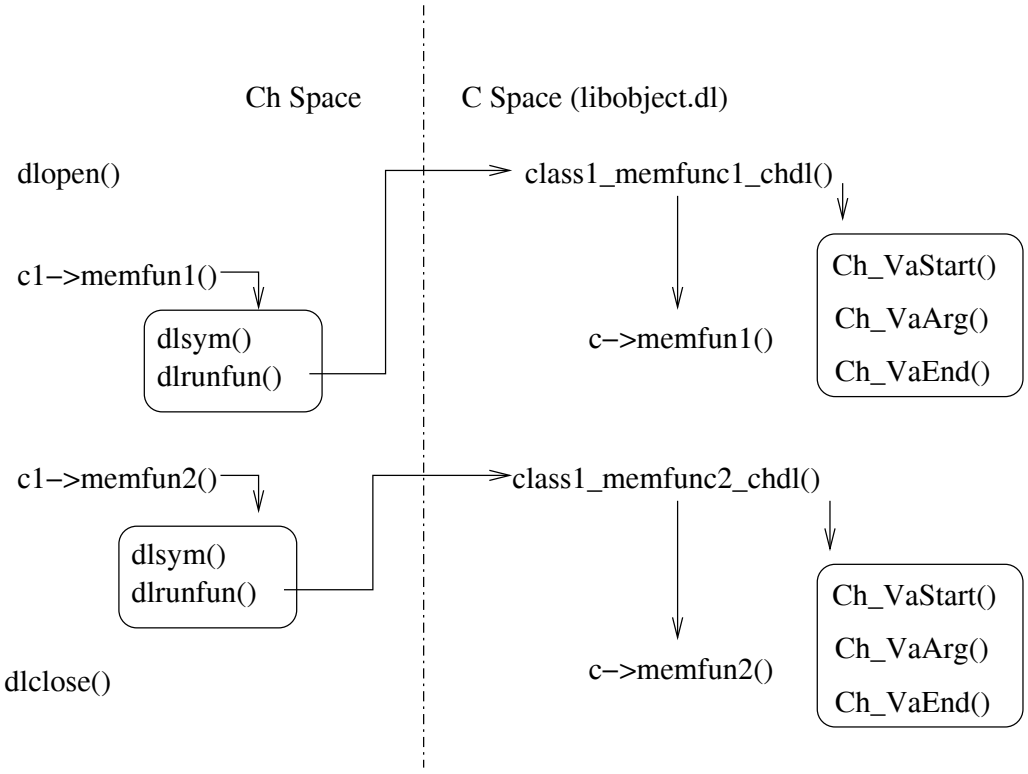


Figure 7.2: Relation of member function of object of Class1 in Ch and C Spaces.

Figure 7.2 illustrates how to deal with calling a Ch member function in the Ch space. If the member function `memfun1`() of the object `c1` is called, the chdl function `class1_memfunc1_chdl`(), which is in the DLL, will be invoked by **dlrunfun**(). In the meantime, the *this* pointer in the C++ space maintained in the Ch space is passed to the C++ space as an argument. In the C++ space, the corresponding member

function `memfun1()` of C++ object `c`, which is pointed to by the *this* pointer, will be finally called. The object *c* in the C++ space and object *c1* in the Ch space share the same memory.

7.3 Member Functions with Arguments of Simple Types

This section describes about class member functions that have arguments of simple data types but no return value. The example that we use is the member function `Class1::memfun2()` in Program 7.7.

```
void Class1::memfun2(data_type2 arg2) {
{
  void *fptr;

  /* to get the address by function name */
  fptr = dlsym(g_sample_dlhandle, "Class1_memfun2_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }

  /* to call the chdl function in dynamically loaded
   library by address */
  dlrunfun(fptr, NULL, memfun2, this, arg2);
  return;
}
```

Program 7.12: Member function with argument of simple data type (chf file).

The member function `Class1::memfun2()` in the C++ space will be called by the member function `Class1::memfun2()` in the Ch space as shown in Program 7.12, Just like the chf function file in Program 7.10, the *this* pointer has to be passed to the C++ space. At the same time, `arg2` is sent as the fifth argument since function `memfun2()` takes that argument.

```
EXPORTCH void Class1_memfun2_chdl(void *varg){
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  data_type2 arg2;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  arg2 = Ch_VaArg(interp, ap, data_type2);

  c->memfun2(arg2);

  Ch_VaEnd(interp, ap);
  return;
}
```

Program 7.13: Member function with argument of simple data type (chdl file).

Program 7.13 is the chdl file for chdl function `Class1_memfun2_chdl()`. This chdl file is pretty

similar to `Class1_memfun1_chdl`() in Program 7.11. However, this function gets two arguments from the Ch space. One is the *this* pointer and the other is the argument `arg2`.

7.4 Member Functions with Return Values of Simple Types

This section illustrates how to add a function with return value of simple data type shown in member function `Class1::memfun3()` in Program 7.7.

```
return_type1 Class1::memfun3() {
{
  void *fptr;
  return_type1 retval;

  /* to get the address by function name */
  fptr = dlsym(g_sample_dlhandle, "Class1_memfun3_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }

  /* to call the chdl function in dynamically
    loaded library by address */
  dlrunfun(fptr, &retval, memfun3, this);

  return retval;
}
```

Program 7.14: Member function with return value of simple data type (chf file).

The member function `Class1::memfun3`() in the Ch space is shown as Program 7.14, This chf is similar to the one shown in Program 7.10, except that the second and third arguments are different in the argument list of function **dlrunfun**(). The address of retval, `&retval`, is passed to the C++ space instead of NULL because it needs to retrieve the return value. The third argument is the name of the member function that is called.

```
EXPORTCH return_type1 Class1_memfun3_chdl(void *varg){
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  return_type1 retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);

  retval = c->memfun3();

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 7.15: Member function with return value of simple data type (chdl file).

Program 7.15 is the chdl function `Class1_memfun3_chdl`(). Besides the pointer to the instance of `Class1` in the C++ space, there is no other argument passed from the Ch space. On the other hand, function `Class1_memfun3_chdl`() returns a value of type `return_type1`.

7.4.1 Two Complete Examples

Several complete examples for interface Ch with C++ classes are presented in this section.

7.4.1.1 Example of Class with a Regular Member function

The complete source code presented below is an example for interface Ch with a C++ class which consists of constructor, destructor, and a member function. Programs `sampclass.cpp` and `script.cpp` can be compiled using a C++ compiler to create a binary executable program `script.exe` with the same Makefile in Listing 4 in section 7.1. The output from the program `script.cpp` is shown in Listing 5. The same application program `script.cpp` can also be executed in Ch without compilation.

Listing 1 — the header file `sampclass.h` in both Ch and C++ spaces.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

/********* for Ch space ***********/
#ifdef _CH_
#include <dlfcn.h>
void *g_sample_dlhandle = dlopen("libsampclass.dl", RTLD_LAZY);
if(g_sample_dlhandle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get g_sample_dlhandle in sampclass.h\n");
   exit(-1);
}
void _dlclose_sampclass(void) {
   dlclose(g_sample_dlhandle);
}
atexit(_dlclose_sampclass);
#endif

class Class1 {
#ifndef _CH_
/********* for C++ space **********/
  private:
    int m_i1;
#endif
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};

#pragma importf <sampclass.chf>

#endif /* _SAMPCLASS_H_ */
```

Listing 2 — the C++ program `sampclass.cpp`.

```
#include <stdio.h>
#include "sampclass.h"
```

```
Class1::Class1() {
  m_i1 = 1;
  printf("m_i1 in Class1::Class1() = %d\n", m_i1);
}

Class1::~Class1() {
  printf("m_i1 in Class1::~Class1() = %d\n", m_i1);
}

int Class1::memfun1(int i) {
  m_i1 += i;
  printf("m_i1 = %d\n", m_i1);
  return m_i1;
}
```

Listing 3 — The application program `script.cpp` in both Ch and C++ spaces.

```
#include "sampclass.h"

int main() {
    class Class1 c1;
    class Class1 *c2 = new Class1;
    class Class1 c3[2];

    c1.memfun1(100);
    c2->memfun1(200);
    c3[0].memfun1(300);
    c3[1].memfun1(300);
    delete c2;
}
```

Listing 4 — Output from executing program `script.cpp`.

```
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
m_i1 = 101
m_i1 = 201
m_i1 = 301
m_i1 = 301
m_i1 in Class1::~Class1() = 201
m_i1 in Class1::~Class1() = 301
m_i1 in Class1::~Class1() = 301
m_i1 in Class1::~Class1() = 101
```

Listing 5 — the chdl program `sampclass_chdl.cpp`.

```
#include <stdio.h>
#include <ch.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

EXPORTCH void Class1_Class1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c = new Class1();
```

```
  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class1();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH int Class1_memfun1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  int i;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1*);
  i = Ch_VaArg(interp, ap, int);         /* get 1st arg  */

  retval = c->memfun1(i);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 6 — the Ch function file `sampclass.chf`.

```
/****** member functions of Class1 ******/

Class1::Class1(){
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, _dlerror());
    return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, NULL, this);
```

```
  return;
}

int Class1::memfun1(int i) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class1_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, memfun1, this, i);
  return retval;
}
```

7.4.1.2 Linked List

The example in this section illustrates how to handle member functions of linked list with classes. Three files `list.h`, `List.chf`, and `liblist.dl` are needed to run the Ch application `script.cpp`. The class `List` defined in header file `list.h` contains a list of students' names and ID numbers. The static fields of the class and member functions are located in file `List.chf`. The public member functions can prepend, insert, append, remove a student in the list, and print out the list. The dynamically loaded library `liblist.dl` is built by two C++ programs `list.cpp` and `list_chdl.cpp` , and header file `list.h` using a makefile `makefile_cpp`. Program `list.cpp` contains the C++ source code for member functions of class List. Program `list_chdl.cpp` is an interface between Ch and C++. Programs in both Ch and C++ spaces share the same header file `list.h`.

Listing 1 — An example Ch program `script.cpp` in C and C++ spaces using member functions in the dynamically loaded library.

```
#include "list.h"

int main() {
  class List students;

  int i;
  struct Data data1, data2, data3, data4;

  /* prepare the data */
  createData(&data1, 456, "Mary");
  createData(&data2, 789, "John");
  createData(&data3, 234, "Peter");
  createData(&data4, 567, "Hanson");
  students.prepend(data1);
  students.prepend(data2);
  students.append(data3);
  students.insert(data4, 2);

  students.printList();
  students.printName("Peter");

  i = students.printName("Bob");
  cout << "There is/are " << i << " person(s) named Bob" << endl;
```

```
  students.removeElement(789);
  students.printList();
  students.printName("Peter");

  students.clear();
  students.printList();

  freeMem(&data1);
  freeMem(&data2);
  freeMem(&data3);
  freeMem(&data4);
  return 0;
}
```

Listing 2 — The Ch/C++ header file `list.h` defining the class.

```
/* list.h */
#ifndef LIST_H_
#define LIST_H_

#include <iostream>
#include <stdlib.h>
using namespace std;

struct Data {
  int id;             /* id number */
  char *name;         /* name */
};

struct Element {
  struct Data data;
  struct Element *next;
};

void createData(struct Data *newData, int id, char *name);
void freeMem(struct Data *elem);

#ifdef _CH_
#include <dlfcn.h>
void *g_sample_dlhandle = dlopen("liblist.dl", RTLD_LAZY);
if(g_sample_dlhandle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get g_sample_dlhandle in sampclass.h\n");
   exit(-1);
}
void _dlclose_sampclass(void) {
   dlclose(g_sample_dlhandle);
}
atexit(_dlclose_sampclass);

class List {
  public:
    List();
    ~List();                //remove memory
    void prepend(struct Data data);   //prepend to the list at the beginning
    void append(struct Data data);   //append to the list at end
    void insert(struct Data data, int pos);   //insert to the list at pos
    void printList();   //print out the list
    int printName(char *name);   //print nodes with the same lname
    void removeElement(int id);   //delete first node with id = id
```

```
    void clear();   //delete all nodes attached to the list
};

/* get member functions of class List*/
#pragma importf <List.chf>
#else
class List {
  private:
    static int totnum;
    Element *head;
  public:
    List();
    ~List();   //remove memory
    void prepend(struct Data data);   //prepend to the list at the beginning
    void append(struct Data data);   //append to the list at end
    void insert(struct Data data, int pos);   //insert to the list at pos
    void printList();   //print out the list
    int printName(char *name);   //print nodes with the same lname
    void removeElement(int id);   //delete first node with id = id
    void clear();   //delete all nodes attached to the list
};
#endif

#endif /* LIST_H_ */
```

Listing 3 — The chf function file `List.chf` contains the definitions of member functions which invoke the dynamically loaded library.

```
/*************************************************************
* File name: List.chf
*            member functions of class List
*************************************************************/

/******************* Public Functions *******************/
List::List(){
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_List_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, NULL);
}

List::~List(){
    void *fptr;
    fptr = dlsym(g_sample_dlhandle, "List_dList_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, NULL, this);
}


void List::prepend (struct Data data) {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_prepend_chdl");
```

```
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    };
    dlrunfun(fptr, NULL, prepend, this, data);
}

void List::append(struct Data data) {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_append_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    };
    dlrunfun(fptr, NULL, append, this, data);
}

void List::insert(struct Data data, int pos) {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_insert_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    };
    dlrunfun(fptr, NULL, insert, this, data, pos);
}

void List::printList() {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_printList_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, printList, this);
}

int List::printName(char *name) {
    void *fptr;
    int retval;

    fptr = dlsym(g_sample_dlhandle, "List_printName_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
        return -1;
    };
    dlrunfun(fptr, &retval, printName, this, name);
    return retval;
}

void List::removeElement (int id) {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_removeElement_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, removeElement, this, id);
```

```
    return;
}

void List::clear() {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "List_clear_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, clear, this);
    return;
}

void createData(struct Data *newData, int id, char *name) {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "createData_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, createData, newData, id, name);
}

void freeMem(struct Data *elem) {
    void *fptr;

    fptr = dlsym(g_sample_dlhandle, "freeMem_chdl");
    if(fptr == NULL)  {
        printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
        return;
    };
    dlrunfun(fptr, NULL, freeMem, elem);
}
```

Listing 4 — The C++ program `list.cpp` contains the C++ source code for member functions of class List.

```
/*****************************************
 *  list_cpp.cpp
 *  Definition of member functions of class List for linked list
 *****************************************/
#include "list.h"
#include "string.h"
#include "stdio.h"

/* static member to maintain the number
   of the students in the list */
int List::totnum = 0;

List::List() {
  head = NULL;
}

List::~List() {

   clear();    // release all memory allocated
}
```

```
/* append to the list at end */
void List::append(struct Data data)
{
  struct Element *temp = head;
  struct Element *e = new Element;
  e->data = data;
  e->next = NULL;

  if(head == NULL) /* no element in linked list */
    head = e;
  else
  {
    /* get the tail of the linked list */
    while(temp->next)
      temp = temp->next;
    /* add the new one to the end of the list */
    temp->next = e;
  }

  totnum++;
  printf("\nelement with id %d is appended\n", e->data.id);
  //cout << endl << "element with id " <<  e->data.id << " is appended" << endl;
}

/* delete all nodes attached to the list */
void List::clear() {
  struct Element *e = NULL;

  while(head) {
    e = head;
    head = head->next;
    delete e;
    totnum--;
  }

  printf("\nAll elements are cleared.\n");
  //cout << endl << "All elements are cleared." << endl;
  return;
}

/* insert to the list at pos */
void List::insert(struct Data data, int pos)
{
  int i = 0;
  struct Element *temp = head;

  struct Element *e = new Element;
  e->data = data;
  e->next= NULL;   /* can be replaced later */

  /* insert to the first position */
  if(pos == 1) {
    head = e;
    head->next = temp;
    totnum++;
    printf("\nAn element with id %d is inserted at the beginning\n", e->data.id);
    return;
  }
```

```
  /* get position */
  while(temp) {
    if(++i >= pos-1) break;
    temp = temp->next;
  }
  if(i != pos-1) { /* can not get that position */
    printf("\nWrong position for insert\n");
    //cout << endl << "Wrong position for insert " << endl;
    delete e;
    return;
  }

  e->next = temp->next;
  temp->next = e;
  totnum++;

  printf("\nAn element with id %d is inserted in the position %d\n", e->data.id, pos);

  return;
}

/* prepend to the list at the beginning */
void List::prepend(struct Data data)
{
  struct Element *e = new Element;

  e->data = data;
  e->next = head;
  head = e;

  totnum++;
  printf("\nAn element with id %d is prepended\n", e->data.id);
  //cout << endl << "An element with id " <<  e->data.id << " is prepended" << endl;

  return;
}

/* print out the list */
void List::printList() {
  struct Element *temp = head;

  if(temp == NULL) {
    printf("\nNo element is in the list\n");
    //cout << endl << "No element is in the list" << endl;
    return;
  }
  else
    printf("\n %d element is in the list\n", totnum);
    //cout << endl << totnum << " element is in the list" << endl;

  while(temp) {
    printf("id  = %d, neame = %s\n", temp->data.id, temp->data.name);
    //cout << "id  = " << temp->data.id << " name = " << temp->data.name << endl;
    temp = temp->next;
  }

  return;
}
```

```
/* print nodes with the same lname */
int List::printName(char * name) {
  struct Element *temp = head;
  int count = 0;

  if(temp == NULL) {
    printf("\nNo element with the name %s\n", name);
    //cout << endl << "No element with the name " << name << endl;
    return 0;
  }

  printf("\nFollowing is/are element(s) with name %s\n", name);
  //cout << endl << "Following is/are element(s) with name " << name << endl;
  while(temp) {
    if(!strcmp(temp->data.name, name))
    {
      printf("id  = %d, name = %s\n", temp->data.id, temp->data.name);
      //cout << "id  = " << temp->data.id << " name = " << temp->data.name << endl;
      count++;
    }
    temp = temp->next;
  }

  return count;
}

/* delete first node with id = id */
void List::removeElement(int id) {
  struct Element *tmp1 = head, *tmp2 = NULL;

  if (head == NULL)
    return;

  else if (head->data.id == id)
  {
    head = head->next;
    printf("\nAn element with id %d is moved\n", tmp1->data.id);
    //cout << endl << "An element with id " << tmp1->data.id << " is moved. " << endl;
    delete tmp1;
    totnum--;
    return;
  }

  else
    while (tmp1->next)
    {
      if (tmp1->next->data.id == id)
      {
        tmp2 = tmp1->next;
        tmp1->next = tmp1->next->next;
        printf("\nAn element with id %d is moved\n", tmp2->data.id);
        //cout << endl << "An element with id " << tmp2->data.id << " is moved." << endl;
        delete tmp2;
        totnum--;
        return;
      }
      tmp1 = tmp1->next;
    }
```

```
  return;
}

/* create a new item of data */
void createData(struct Data *newData, int id, char *name) {
  newData->id = id;
  newData->name = strdup(name);
  return;
}

/* release the memory */
void freeMem(struct Data *elem) {
  free(elem->name);
  return;
}
```

Listing 5 — The chdl program `list_chdl.cpp` that interfaces between programs `List.chf` in Ch and `list.cpp` in C++ through a dynamically loaded library `liblist.dl`.

```
/* list_chdl.cpp */
#include <ch.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
#include "list.h"


EXPORTCH void List_List_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c = new List();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(List));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void List_dList_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  if(Ch_CppIsArrayElement(interp))
    c->~List();
  else
    delete c;
  Ch_VaEnd(interp, ap);
}

EXPORTCH void List_append_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;
  struct Data data;

  Ch_VaStart(interp, ap, varg);
```

```
  c = Ch_VaArg(interp, ap, class List*);
  data = Ch_VaArg(interp, ap, struct Data);
  c->append(data);

  Ch_VaEnd(interp, ap);
}

EXPORTCH void List_prepend_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;
  struct Data data;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  data = Ch_VaArg(interp, ap, struct Data);
  c->prepend(data);

  Ch_VaEnd(interp, ap);
}

EXPORTCH void List_insert_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;
  struct Data data;
  int pos;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  data = Ch_VaArg(interp, ap, struct Data);
  pos = Ch_VaArg(interp, ap, int);
  c->insert(data, pos);

  Ch_VaEnd(interp, ap);
}

EXPORTCH void List_printList_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  c->printList();
  Ch_VaEnd(interp, ap);
}

EXPORTCH int List_printName_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;
  char *name;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  name  = Ch_VaArg(interp, ap, char *);
  retval = c->printName(name);
```

```
  Ch_VaEnd(interp, ap);

  return retval;
}

EXPORTCH void List_removeElement_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int id ;
  class List *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  id = Ch_VaArg(interp, ap, int);
  c->removeElement(id);
  Ch_VaEnd(interp, ap);
}

EXPORTCH void List_clear_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class List *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class List*);
  c->clear();
  Ch_VaEnd(interp, ap);
}

EXPORTCH void createData_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  struct Data *newData;
  int id;
  char *name;

  Ch_VaStart(interp, ap, varg);
  newData = Ch_VaArg(interp, ap, struct Data *);
  id = Ch_VaArg(interp, ap, int);
  name = Ch_VaArg(interp, ap, char *);
  createData(newData, id, name);
  Ch_VaEnd(interp, ap);
}

EXPORTCH void freeMem_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  struct Data *elem;

  Ch_VaStart(interp, ap, varg);
  elem = Ch_VaArg(interp, ap, struct Data *);
  freeMem(elem);
  Ch_VaEnd(interp, ap);
}
```

Listing 6 — Makefile `makefile_cpp`.

```
# Make dynamic linked lib liblist.dl for demo lib
```

```
target: liblist.dl script.exe

liblist.dl: list.o list_chdl.o
        ch dllink liblist.dl cplusplus list.o list_chdl.o

list.o: list.cpp
        ch dlcomp liblist.dl cplusplus list.cpp

list_chdl.o: list_chdl.cpp
        ch dlcomp liblist.dl cplusplus list_chdl.cpp

script.exe: script.o list.o
        ch dllink script.exe cplusplus script.o list.o

script.o: script.cpp
        ch dlcomp liblist.dl cplusplus script.cpp

test:
        script.exe    > test1
        ch script.cpp > test2
        diff test1 test2

clean:
        rm -f *.o *.obj a.out liblist.dl *.lib *.exp *.exe test1 test2
```

Listing 7 — Output from executing program `script.cpp`.

```
An element with id 456 is prepended

An element with id 789 is prepended

element with id 234 is appended

An element with id 567 is inserted in the position 2

 4 element is in the list
id  = 789, neame = John
id  = 567, neame = Hanson
id  = 456, neame = Mary
id  = 234, neame = Peter

Following is/are element(s) with name Peter
id  = 234, name = Peter

Following is/are element(s) with name Bob
There is/are 0 person(s) named Bob

An element with id 789 is moved

 3 element is in the list
id  = 567, neame = Hanson
id  = 456, neame = Mary
id  = 234, neame = Peter

Following is/are element(s) with name Peter
id  = 234, name = Peter

All elements are cleared.
```

```
No element is in the list

All elements are cleared.
```

7.5 Multiple Classes and Functions with Class of Pointer or Array Type

In section 7.1, we have described how to load and unload a DLL in a constructor and deconstructor. However, if we have many classes in a DLL, it is inefficient or even impossible to load a DLL in a constructor for each class.

In this section, we will give an example on how to interface multiple classes in C++ using a single dynamically loaded library. We will also illustrate how to handle member functions with arguments or return value of pointer to classes and pointer to pointer to classes. as well as functions with arguments of class of array type. The interface from Ch to member functions of pointer and array type of classes is the same as interface with other simple data types. This will be illustrated by an example below.

A second class named `Class2` will be used in conjunction with `Class1` for the illustrative purpose. The definitions of both `Class1` and `Class2` in both Ch and C++ spaces are shown in Listing 1. Member function `Class2::memfun1`(). has an argument of pointer to `Class1`. Member function `Class2::memfun2`() returns a pointer to `Class1`. Member function `Class2::memfun3`() returns a pointer to `Class1` and has an argument of pointer to pointer to `Class1`. Member function `Class2::memfun4`() has an argument of array of pointer to `Class1`. Member function `Class2::memfun5`() has an argument of array of `Class1`.

The source code for definitions of classes `Class1` and `Class2` in the C++ space is listed in Listing 2. It can be compiled and linked along with the application program `script.cpp` in Listing 3 using a C++ compiler to create a binary executable program with the output shown in Listing 4. The same Makefile in Listing 4 in section 7.1 can be used. As illustrated in the example, the interface for multiple classes can be handled in the same manner as the one with a single class.

Listing 1 — the header file `sampclass.h` in both Ch and C++ spaces.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

#define NUM 2

#ifdef _CH_
#include <dlfcn.h>
void *g_sample_dlhandle = dlopen("libsampclass.dl", RTLD_LAZY);
if(g_sample_dlhandle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get g_sample_dlhandle in sampclass.h\n");
   exit(-1);
}
void _dlclose_sampclass(void) {
   dlclose(g_sample_dlhandle);
}
atexit(_dlclose_sampclass);

class Class1 {
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};
```

```
class Class2 {
  public:
    Class2();
    ~Class2();
    void memfun1(class Class1 *pc);
    class Class1 *memfun2(int i);
    class Class1 *memfun3(int i, class Class1 **ppc);
    int memfun4(class Class1 *apc[NUM]);
    int memfun5(int num, class Class1 c[NUM]);
};
#pragma importf <sampclass.chf>
#else /************************ for C++ space *****************/
class Class1 {
  private:
    int m_i1;
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};

class Class2 {
  private:
    int m_i2;
  public:
    Class2();
    ~Class2();
    void memfun1(class Class1 *pc);
    class Class1 *memfun2(int i);
    class Class1 *memfun3(int i, class Class1 **ppc);
    int memfun4(class Class1 *apc[NUM]);
    int memfun5(int num, class Class1 c[NUM]);
};
#endif


#endif _SAMPCLASS_H_
```

Listing 2 — the C++ program `sampclass.cpp`.

```
#include <stdio.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

Class1::Class1() {
  m_i1 = 1;
  printf("m_i1 in Class1::Class1() = %d\n", m_i1);
}

Class1::~Class1() {
  printf("m_i1 in Class1::~Class1() = %d\n", m_i1);
}

int Class1::memfun1(int i) {
  m_i1 += i;
  printf("m_i1 = %d\n", m_i1);
  return m_i1;
}
```

```
/****** member functions of Class2 ******/

Class2::Class2() {
  printf("Class2::Class2() called\n");
}

Class2::~Class2() {
  printf("Class2::~Class2() called\n");
}

void Class2::memfun1(Class1 *pc) {

  pc->memfun1(10); //set m_i1 to 10
}

Class1 *Class2::memfun2(int i) {
  Class1 *retval;

  retval = new Class1();
  retval->memfun1(i);
  return retval;
}

Class1 *Class2::memfun3(int i, Class1 **ppc) {
  Class1 *pc;
  pc = *ppc;
  return &pc[i];
}

int Class2::memfun4(Class1 *apc[NUM]) {
  int i;
  for (i = 0; i < NUM; i++) {
     if(apc[i]) {
       apc[i]->memfun1(i);
     }
  }
  return 0;
}

int Class2::memfun5(int num, Class1 *pc) {
  int i;

  for(i=0; i<num; i++) {
    pc[i].memfun1(1000+i);
  }
  return 0;
}
```

Listing 3 — the main program `script.cpp` in both Ch and C++ spaces.

```
#include "sampclass.h"

int main() {
  class Class1 c1;
  class Class1 *pc1, *pc2, *pc3;
  class Class1 ac[NUM];
  class Class1 *apc[NUM];
  class Class2 b;
```

```
  int i;
  b.memfun1(&c1);
  pc1 = b.memfun2(20);
  pc2 = new Class1[2];
  pc2[0].memfun1(10);
  pc2[1].memfun1(20);
  pc3 = b.memfun3(1, &pc2);
  pc3->memfun1(100);

  for (i = 0; i < NUM; i++) {
     apc[i] = new Class1;
  }
  b.memfun4(apc);
  b.memfun5(NUM, ac);


  delete pc1;
#ifdef _CH_
  delete [2] pc2;
#else
  delete []  pc2;
#endif
  for (i = 0; i < NUM; i++) {
     delete apc[i];
  }
  return 0;
}
```

Listing 4 — Output from executing program `script.cpp`.

```
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
Class2::Class2() called
m_i1 = 11
m_i1 in Class1::Class1() = 1
m_i1 = 21
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
m_i1 = 11
m_i1 = 21
m_i1 = 121
m_i1 in Class1::Class1() = 1
m_i1 in Class1::Class1() = 1
m_i1 = 1
m_i1 = 2
m_i1 = 1001
m_i1 = 1002
m_i1 in Class1::~Class1() = 21
m_i1 in Class1::~Class1() = 121
m_i1 in Class1::~Class1() = 11
m_i1 in Class1::~Class1() = 1
m_i1 in Class1::~Class1() = 2
Class2::~Class2() called
m_i1 in Class1::~Class1() = 1002
m_i1 in Class1::~Class1() = 1001
m_i1 in Class1::~Class1() = 11
```

Through the interface programs in Listings 5 to 7 we shall be able to execute program `script.cpp` interpretively in Ch. The makefile in Listing 7 can be used to build the dynamically loaded library

`libsampclass.dl` and binary executable program `script.exe`.

Listing 5 — the chdl program `sampclass_chdl.cpp`.

```
#include <stdio.h>
#include <ch.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

EXPORTCH void Class1_Class1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c = new Class1();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class1();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH int Class1_memfun1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  int i;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1*);
  i = Ch_VaArg(interp, ap, int);         /* get 1st arg  */
  retval = c->memfun1(i);
  Ch_VaEnd(interp, ap);
  return retval;
}

/****** member functions of Class2 ******/

EXPORTCH void Class2_Class2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c = new Class2();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class2));
```

```
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class2_dClass2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class2();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH void Class2_memfun1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  class Class1 *pc;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2*);
  pc = Ch_VaArg(interp, ap, class Class1*);
  c->memfun1(pc);
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH class Class1 *Class2_memfun2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  int i;
  class Class1 *retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2*);
  i = Ch_VaArg(interp, ap, int);
  retval = c->memfun2(i);
  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH class Class1 *Class2_memfun3_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  int i;
  class Class1 **ppc;
  class Class1 *retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2*);
  i = Ch_VaArg(interp, ap, int);
  ppc = Ch_VaArg(interp, ap, class Class1**);
```

```
  retval = c->memfun3(i, ppc);
  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH int Class2_memfun4_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  class Class1 **apc;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2*);
  apc = Ch_VaArg(interp, ap, class Class1**);
  retval = c->memfun4(apc);
  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH int Class2_memfun5_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  class Class1 *pc;
  int num;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2*);
  num = Ch_VaArg(interp, ap, int);
  pc = Ch_VaArg(interp, ap, class Class1*);
  retval = c->memfun5(num, pc);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 6 — the Ch function file `sampclass.chf`.

```
/****** member functions of Class1 ******/
Class1::Class1(){
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
```

```
  dlrunfun(fptr, NULL, NULL, this);
}

int Class1::memfun1(int i) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class1_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, memfun1, this, i);
  return retval;
}

/****** member functions of Class2 ******/

Class2::Class2(){
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class2_Class2_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class2::~Class2() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class2_dClass2_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL, this);
}

void Class2::memfun1(Class1 *pc) {
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, memfun1, this, pc);
  return;
}

Class1 *Class2::memfun2(int i) {
  void *fptr;
  Class1 *retval;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun2_chdl");
  if(fptr == NULL)  {
```

```
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, memfun2, this, i);
  return retval;
}

Class1 *Class2::memfun3(int i, Class1 **ppc) {
  void *fptr;
  Class1 *retval;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun3_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, memfun3, this, i, ppc);
  return retval;
}

int Class2::memfun4(Class1 *apc[NUM]) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun4_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, NULL, this, apc);
  return retval;
}

int Class2::memfun5(int num, Class1 c[NUM]) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun5_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, NULL, this, num, c);
  return retval;
}
```

7.6 Classes with Multiple Constructors

This section will use an example to illustrate how to handle C++ classes with multiple constructors. The declarations of the class `Class1` with multiple constructors in both Ch and C++ spaces are given in the header file `sampclass.h` in Listing 1. The definitions of ts constructors in the C++ space are shown program `sampclass.cpp` in Listing 2. The constructor of `Class1` can take up to two arguments. If it takes two

arguments, the first one has to be an `integer` and the second one `double`. Programs `sampclass.cpp` and `script.cpp` can be compiled using the same Makefile in Listing 4 in section 7.1 to create a binary executable program `script.exe`. The application program `script.cpp` in Listing 3 will produce the output in Listing 4.

The corresponding constructor in the Ch space is given in Listing 5 We use `class1::class1(...)` to handle the different number of arguments. With APIs `va_start`(), `va_count`() and `va_elementtype`(), cases with different number of arguments will take different `if-then` paths.

The chdl interface function of the constructor of `Class1` is shown in Listing 6. The same Makefile in Listing 4 in section 7.1 can be used to create a dynamically loaded library `libsampclass.dl`.

Example: A class with multiple constructors
This is an example of handling multiple constructors. Three objects will be initialized by calling constructor with different arguments. The output from executing program `script.cpp` in Listing 3 in both Ch and C++ spaces is given in Listing 4.

Listing 1 — the header file `sampclass.h` in both Ch and C++ spaces.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_


/********* for Ch space **************/
#ifdef _CH_
#include<dlfcn.h>
#include<stdarg.h>
class Class1 {
  private:
    static void *g_sample_dlhandle; // make sure to load dl once
    static int   g_sample_dlcount;  // count the instance of class
  public:
    Class1(...);
    ~Class1();
};
void * Class1::g_sample_dlhandle = NULL;
int Class1::g_sample_dlcount = 0;

#pragma importf <sampclass.chf>
#else
/********* for C++ space **************/
class Class1 {
  private:
    double m_d;
  public:
    Class1();
    Class1(int i);
    Class1(int i, double d);
    ~Class1();
};
#endif

#endif /* _SAMPCLASS_H_ */
```

Listing 2 — the C++ program `sampclass.cpp`

```
#include <stdio.h>
#include "sampclass.h"
```

```
Class1::Class1() {
  m_d = 1;
  printf("in Constructor in C++ space, m_d = %f\n", m_d);
}

Class1::Class1(int i) {
  m_d = i;
  printf("in Constructor in C++ space, m_d = %f\n", m_d);
}

Class1::Class1(int i, double d) {
  m_d = i+d;
  printf("in Constructor in C++ space, m_d = %f\n", m_d);
}

Class1::~Class1() {
}
```

Listing 3 — The application program `script.cpp` in both Ch and C++ spaces.

```
#include "sampclass.h"

int main() {
  class Class1 c1=Class1();
  class Class1 c2=Class1(5);
  class Class1 c3=Class1(5, 10.0);

  return 0;
}
```

Listing 4 — Output from executing program `script.cpp` in the Ch space.

```
m_d = 1.000000
m_d = 5.000000
m_d = 15.000000
```

Listing 5 — the Ch function file `sampclass.chf`.

```
/****** member functions of Class1 ******/

Class1::Class1(){
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, _dlerror());
    return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, NULL, this);
```

```
  return;
}

int Class1::memfun1(int i) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class1_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, memfun1, this, i);
  return retval;
}
```

Listing 6 — the chdl program `sampclass_chdl.cpp`

```
#include <stdio.h>
#include <ch.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

EXPORTCH void Class1_Class1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  int vacount;
  int i;
  double d;

  Ch_VaStart(interp, ap, varg);
  vacount = Ch_VaCount(interp, ap);
  if(vacount ==0)
      c = new Class1();
  else if(vacount ==1) {
      i = Ch_VaArg(interp, ap, int);
      c = new Class1(i);
  }
  else if(vacount ==2) {
      i = Ch_VaArg(interp, ap, int);
      d = Ch_VaArg(interp, ap, double);
      c = new Class1(i, d);
  }
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class1();
  else
```

```
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}
```

7.7 Member Functions with Return Values and Arguments of Class Type

Member functions with return values and arguments of class type are discussed in this section. They are handled differently from other data types.

Let's take a look at the following member function prototypes in Program 7.16.

```
Class1 memfun1();
inv memfun2(Class1 c);
```

The member function `Class2::memfun1()` returns class `Class1` whereas the member function `Class2::memfun2()` has an argument of class `Class1`. The complete definitions of classes `Class1` and `Class2` in both Ch and C++ spaces are given in Program 7.16. The definitions of member functions in the C++ space is shown in Program 7.17. Instances of `Class1` is initialized and returned by member function `Class2::memfun1()`. Note that, in practice, the user typically may only get the binary library of these member functions rather than the source code.

An application uses these member functions is shown in Program 7.18. The output of executing Program 7.18 is shown in Figure 7.3. The same Makefile in Listing 4 in section 7.1 can be used.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

#ifdef _CH_
#include <dlfcn.h>
void *g_sample_dlhandle = dlopen("libsampclass.dl", RTLD_LAZY);
if(g_sample_dlhandle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get g_sample_dlhandle in sampclass.h\n");
   exit(-1);
}
void _dlclose_sampclass(void) {
   dlclose(g_sample_dlhandle);
}
atexit(_dlclose_sampclass);

class Class1 {
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};

class Class2 {
  public:
    Class2();
    ~Class2();
    Class1 memfun1();      // return Class1
    int memfun2(Class1 c); // arg of Class1
};

#pragma importf "sampclass.chf"
#else
class Class1 {
  private:
    int m_i1;
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};

class Class2 {
  private:
    int m_i2;
  public:
    Class2();
    ~Class2();
    Class1 memfun1();            // return Class1
    int memfun2(class Class1 c); // arg of Class1
};
#endif

#endif /* _SAMPCLASS_H_ */
```

Program 7.16: The header file for definitions of classes in Ch/C++ space (sampclass.h).

```
#include <stdio.h>
#include <stdlib.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

Class1::Class1() {
  m_i1 = 1;
}

Class1::~Class1() {
}

int Class1::memfun1(int i) {
  m_i1 += i;
  printf("In Class1::memfun1(), m_i1 = %d\n", m_i1);
  return m_i1;
}

/****** member functions of Class2 ******/

Class2::Class2() {
  m_i2 = 2;
}

Class2::~Class2() {
}

Class1 Class2::memfun1() {
  Class1 c1;
  return c1;
}

int Class2::memfun2(Class1 c) {
  c.memfun1(1000);
  return 0;
}
```

Program 7.17: Definitions of member functions in the C++ space (sampclass.cpp).

```
#include <stdio.h>
#include "sampclass.h"

int main() {
  int i;
  class Class1 c1_1, c1_2;
  class Class2 c2_1;
  class Class2 *c2_2;

  c1_1 = c2_1.memfun1();
  c1_1.memfun1(10);
  c2_2 = new Class2;
  c1_2 = c2_2->memfun1();
  c1_2.memfun1(10);

  c2_1.memfun2(c1_1);

  delete c2_2;
  return 0;
}
```

Program 7.18: Application program in C and C++ spaces(script.cpp).

```
In Class1::memfun1(), m_i1 = 11
In Class1::memfun1(), m_i1 = 11
In Class1::memfun1(), m_i1 = 1011
```

Figure 7.3: The output of executing Program 7.18

.

Ch functions of these member functions are shown in Program 7.19. To avoid calling constructor multiple times, for member function `Class2::memfun1`() returning a class, the memory for the returning class is allocated in the Ch space by function `malloc()`. The address is passed to function `Class2_memfun1_chdl()` in the C++ space to receive the returned value. The memory will be deallocated in the Ch kernel.

Similarly, for member function `Class2::memfun2`() with an argument of class type, the address of the argument is passed to function `Class2_memfun2_chdl()` in the C++ space, which in turn will pass the class using an indirection operator to member function `Class2::memfun2`() in the C++ space.

7.8 Static Member Functions

The memory in both Ch and C++ spaces can be shared as described in the previous sections. Regular and static member functions as well as static data members do not occupy the memory in an instance of class. Therefore, it is possible that additional member functions, including static and private member functions as well static data members, in a class in the Ch space can use different names, which can call the same member function in the C++ space.

In this section, we will demonstrate how to create static member functions in the Ch space that will call regular member functions in the C++ space. The definitions of class `Class1` in both Ch and C++ spaces

```
/****** member functions of Class1 ******/
Class1::Class1() {
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL, this);
}

int Class1::memfun1(int i) {
  void *fptr;
  int retval;
  fptr = dlsym(g_sample_dlhandle, "Class1_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return -1;
  }
  dlrunfun(fptr, &retval, memfun1, this, i);
  return retval;
}
```

Program 7.19: The function files of member functions (sampclass.chf).

```
/****** member functions of Class2 ******/
Class2::Class2() {
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class2_Class2_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class2::~Class2() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class2_dClass2_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL, this);
}

Class1 Class2::memfun1() {
  void *fptr;
  Class1 *retval;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     exit -1;
  }
  retval = (Class1 *)malloc(sizeof(Class1));

  dlrunfun(fptr, NULL, NULL, retval, this);
  return *retval;
}


int Class2::memfun2(Class1 c) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class2_memfun2_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return NULL;
  }

  dlrunfun(fptr, &retval, NULL, this, &c);
  return retval;
}
```

Program 7.19: The function files of member functions (sampclass.chf) (Contd.).

```
#include <stdio.h>
#include <ch.h>
#include <string.h>
#include "sampclass.h"

EXPORTCH void Class1_Class1_chdl(void) {
  ChInterp_t interp;
  class Class1 *c=new Class1();
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
}

EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class1();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH int Class1_memfun1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  int i;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1*);
  i = Ch_VaArg(interp, ap, int);         /* get 1st arg  */
  retval = c->memfun1(i);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 7.20: The chdl functions of member functions (sampclass_chdl.cpp).

```
EXPORTCH void Class2_Class2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c = new Class2();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class2));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class2_dClass2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class2();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH void Class2_memfun1_chdl(void *varg) { /* returning Class1 */
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  Class1 *retval;

  Ch_VaStart(interp, ap, varg);
  retval = Ch_VaArg(interp, ap, class Class1*);
  c = Ch_VaArg(interp, ap, class Class2*);
  *retval = c->memfun1(); /* copy over */
  Ch_VaEnd(interp, ap);
}

EXPORTCH int Class2_memfun2_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class2 *c;
  class Class1 *pc;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class2*);
  pc = Ch_VaArg(interp, ap, class Class1*);
  retval = c->memfun2(*pc);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 7.20: The chdl functions of member functions (sampclass_chdl.cpp) (Contd.).

are given in Program 7.21. The constructor, destructor and member function `Class1::memfun1()` are the same as in the previous sections as shown in Program 7.22. But, the class `Class1` in the Ch space has two extra static member functions `Class1::Create()` and `Class1::Delete()` in comparison with one in the C++ space. They can be used to create and delete an instance of class.

The member function `Class1::Create()` in the Ch space is defined in Program 7.23. It calls the binary interface function `Class1_Create_chdl` in Program 7.24 to instantiate an instance of the class, similar to the interface function. `Class1_Class1_chdl` for constructor. Unlike function `Class1_Class1_chdl()` which does not return a value, function `Class1_Create_chdl()` returns a pointer to the created instance of class. This instance of class shall be deleted by the memeber function `Class1::Delete()` in the Ch space, which in turn calls the interface function `Class1_dClass1_chdl()` for the destructor.

A Ch application program and its output are shown in Program 7.25 and Figure 7.4, respectively.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

#ifdef _CH_
#include <dlfcn.h>
void *g_sample_dlhandle = dlopen("libsampclass.dl", RTLD_LAZY);
if(g_sample_dlhandle == NULL) {
   fprintf(_stderr, "Error: dlopen(): %s\n", dlerror());
   fprintf(_stderr, "       cannot get g_sample_dlhandle in sampclass.h\n");
   exit(-1);
}
void _dlclose_sampclass(void) {
   dlclose(g_sample_dlhandle);
}
atexit(_dlclose_sampclass);

class Class1 {
  public:
    static Class1 *Create();
    static void Delete(Class1 *);
    Class1();
    ~Class1();
    int memfun1(int i);
};

#pragma importf <sampclass.chf>
#else /************************ for C++ space *****************/
class Class1 {
  private:
    int m_i1;
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
};
#endif


#endif _SAMPCLASS_H_
```

Program 7.21: The header file for definitions of classes in Ch/C++ space (sampclass.h).

```
#include <stdio.h>
#include "sampclass.h"


Class1::Class1() {
  m_i1 = 1;
  printf("in Constructor in C++ space, m_i1 = %d\n", m_i1);
}

Class1::~Class1() {
  printf("in Destructor in C++ space, m_i1 = %d\n", m_i1);
}

int Class1::memfun1(int i) {
  m_i1 += i;
  printf("m_i1 = %d\n", m_i1);
  return m_i1;
}
```

Program 7.22: Definitions of member functions in the C++ space (sampclass.cpp).

```
#include "sampclass.h"

int main() {
  class Class1 c1;
  class Class1 *c2 = new Class1();
  class Class1 *c3 = Class1::Create();

  c1.memfun1(100);
  c2->memfun1(200);
  c3->memfun1(300);
  delete c2;
  Class1::Delete(c3);
  return 0;
}
```

Program 7.25: Ch application program (script.ch).

```
in Constructor in C++ space, m_i1 = 1
in Constructor in C++ space, m_i1 = 1
in Constructor in C++ space, m_i1 = 1
m_i1 = 101
m_i1 = 201
m_i1 = 301
in Destructor in C++ space, m_i1 = 201
in Destructor in C++ space, m_i1 = 301
in Destructor in C++ space, m_i1 = 101
```

Figure 7.4: The output of executing Program 7.25

.

```
/****** member functions of Class1 ******/

Class1::Class1(){
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class1_Class1_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return;
  }
  dlrunfun(fptr, NULL, NULL);
}

Class1::~Class1() {
  void *fptr;
  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }

  dlrunfun(fptr, NULL, NULL, this);
}

int Class1::memfun1(int i) {
  void *fptr;
  int retval;

  fptr = dlsym(g_sample_dlhandle, "Class1_memfun1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, memfun1, this, i);
  return retval;
}

Class1 *Class1::Create(){
  void *fptr;
  Class1 *cptr;

  fptr = dlsym(g_sample_dlhandle, "Class1_Create_chdl");
  if(fptr == NULL)  {
    printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
    return NULL;
  }
  dlrunfun(fptr, &cptr, NULL);
  return cptr;
}

void Class1::Delete(Class1 *cptr) {
  void *fptr;

  fptr = dlsym(g_sample_dlhandle, "Class1_dClass1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __class_func__, dlerror());
     return;
  }
  dlrunfun(fptr, NULL, NULL, cptr);
}
```


Program 7.23: The function files of member functions (sampclass.chf).

```
#include <stdio.h>
#include <ch.h>
#include "sampclass.h"

/****** member functions of Class1 ******/

EXPORTCH void Class1_Class1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c = new Class1();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
}

EXPORTCH void Class1_dClass1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1 *);
  if(Ch_CppIsArrayElement(interp))
    c->~Class1();
  else
    delete c;
  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH int Class1_memfun1_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  int i;
  int retval;

  Ch_VaStart(interp, ap, varg);
  c = Ch_VaArg(interp, ap, class Class1*);
  i = Ch_VaArg(interp, ap, int);         /* get 1st arg  */

  retval = c->memfun1(i);

  Ch_VaEnd(interp, ap);
  return retval;
}

EXPORTCH class Class1* Class1_Create_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  class Class1 *c;
  c = new Class1();

  Ch_VaStart(interp, ap, varg);
  Ch_CppChangeThisPointer(interp, c, sizeof(Class1));
  Ch_VaEnd(interp, ap);
  return c;
}
```

Program 7.24: The chdl functions of member functions (sampclass_chdl.cpp).

7.9 C++ Functions with Arguments of Data Type boolean

The type `boolean` in C++ cannot interface with Ch in the same manner as other data types. The following chdl code fragment shows the correct way to handle this data type.

```
EXPORTCH void functionName_chdl(void *varg) {
   ChInterp_t interp;
   ChVaList_t ap;
   class class1 *c;
   bool value;

   Ch_VaStart(interp, ap, varg);
   c = Ch_VaArg(interp, ap, class class1*);
   value = Ch_VaArg(interp, ap, char);
   /* buffering = 1; */
   c->functionName(value);
   Ch_VaEnd(interp, ap);
}
```

The argument value of boolean type in C++ should be obtained as char to interface. That is why we used

```
value = Ch_VaArg(interp, ap, char);
```

instead of

```
value = Ch_VaArg(interp, ap, bool);
```

Chapter 8

Calling Ch Functions with Arguments of VLAs from C Space

A large number of existing Ch code, especially on numerical computation, are very useful to users. Some users may want to call these Ch functions from both Ch space and C space. In section2.4, we have discussed three typical methods of calling Ch functions or access Ch variables from C space. Some commonly used functions, including **Ch_CallFuncByAddr()**, **Ch_ExprEval()**, and **Ch_ExprCalc()**, have been introduced with some simple samples. Although those simple samples can help users deal with most cases they may encounter, some special cases about functions with arguments of VLAs have not been covered yet.

In C space, an array is passed to a function as a pointer only. For the C function with arguments of fixed length arrays, such as

```
int funct1(int i, int a[2], int b[2][3]);
```

the information of number of dimensions and extents is obtained from the prototype of the function. For the C function with arguments of variable length arrays, such as

```
int funct2(int i, int a[], int b[][3], int m, int n);
```

the information of dimensions and extents is mostly obtained from the additional arguments, for example, arguments `m` and `n` for arrays `a[m]` and `b[n][3]` in the function prototype `funct2`().

However, in the argument list of a Ch function, an array, which can be either C array or Ch computational array, is passed assumed-shape array. Other information about this array, which is invisible to users, is also passed through the argument list inexplicitly. So, for a Ch function, users don't need to pass extra arguments explicitly except for the array name. For example, the prototype of Ch function with argument of VLA could be

```
int funct3(int i, int a[:], int b[:][:]);
```

More information about the arrays in Ch can be found in *Ch User's Guide*. Different methods of calling Ch functions with arguments of VLAs in the Ch program using APIs **Ch_CallFuncByName**() and **Ch_CallFuncByAddr**() will be described in this chapter. It is equivelent to build an argument list using API **Ch_VarArgsAddArg**() for function call of **Ch_CallFuncByNameVar**() .

8.1 Calling Ch Functions with Arguments of Assumed-Shape Arrays

The methods described in this section can apply to Ch functions with arguments of both C arrays and Ch computational arrays of assumed-shape. Assumed that the Ch function below takes a two-dimensional array `a` as the argument.

```
int func1(array int a[:][:]) {
...
}
```

According to the discussion in section 6.1.4, we can pass a pointer to this Ch function to C space. In C space, the C function `cfun1`() in which the Ch function `func1`() is invoked can be defined as follows.

```
int cfun1(void *chfunptr){
   int dim, ext1, ext2;
   int retval;
   int aa1[2][3] = {1, 2, 3,
                    4, 5, 6};
   dim =2, ext1 = 2; ext2 = 3; /* extents of aa1 */
   Ch_CallFuncByAddr(interp, chfunptr, &retval, aa1, dim, ext1, ext2);
return 0;
}
```

Inside function `cfun1`(), the function **Ch_CallFuncByAddr**() is used to call the Ch function `func1`(), which is pointed to by the second argument, `chfunptr`. The third argument gets the return value from the Ch function. The argument `aa1` is the array in C space to be passed to the Ch function as the argument. In most cases, such as the one in section 6.1.4, the number of the arguments after the third one matchs exactly with the number of the arguments of the Ch function to be called. This rule doesn't apply to the case of calling a Ch function with arguments of VLAs, because VLAs in Ch argument lists can provide more information in addition to addresses whereas those in C argument list can't. So, the users need to provide extra arguments about extents of array explicitly when a Ch function with arguments of VLAs is called from C space. In the above function definition, although the Ch function takes only one argument, two extra arguments, `ext1` and `ext2`, are also passed to the Ch function by **Ch_CallFuncByAddr**(). The argument `aa1` is only an address in C space, while `dim`, `ext1`, and `ext2` provide the information about the dimension and extents of array `aa1` to the Ch function. Because the Ch function `func1`() only takes the argument of assumed-shape array of fixed dimension, the users don't need to provide the information of dimensions from C space. For the same reason, if a Ch function `func1`() takes the argument of an X-dimensional assumed-shape array, where X can be any integer, the users should add X extra arguments in **Ch_CallFuncByAddr**() to provide the information of extents of all dimensions of the array.

Example 1

The complete sample of this case is shown in Programs 8.1 and 8.2. The Ch function `sum2d`() calculates the sum of each element of the two-dimensional array `a`. The function **shape**() called in `sum2d`() can get the information of shape of an array in Ch space. In the function `main`() in Ch space, the function `sum2d`() is called first to calculate the sum of the array `a` as follows,

```
sum_ret = sum2d(a);
```

Then the function pointer to `sum2d`() is passed to C space by function call

```
callchvla_2d(sum2d);
```

In C space (Program 8.2), the function pointer passed from Ch space is assigned to `sum2d_ch_fun` by

```
sum2d_ch_fun = Ch_VaArg(interp, ap, void *);
```

and passed to function `cfun1`() by function call

```
retval = cfun1(sum2d_ch_fun);
```

In the C function `cfun1`(), the Ch function `sum2d`(), which is pointed to by `chfunptr`, is called twice to calculate sum of two arrays `aa1` and `aa2` of different shapes. Array `aa1` has shape of (2×3). In the code below,

```
dim = 2, ext1 = 2; ext2 = 3; /* dim and extents of aa1 */
Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa1, dim, ext1, ext2);
```

two extra arguments `ext1` and `ext2`, which contain extents of `aa1`, are passed to Ch function. For array `aa2`, its extents are also passed as extra arguments in the function call below.

```
dim = 2, ext1 = 3; ext2 = 4; /* dim and extents of aa2 */
Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa2, dim, ext1, ext2);
```

Note that the dimensions of arrays in C space, such as `aa1` and `aa2`, should be the same as the dimensions of the array argument the Ch function `sum2d`(). In this example, `sum2d` takes a 2-dimensional array as the argument, arrays in C space should also be of two dimensions. The output from executing this example is shown in Figure 8.1.

If the Ch function takes the argument of an array of reference without a subscript, arrays with different dimensions can be passed to it. This special case will be discussed in the next section.

Example 2

Another sample is shown in Programs 8.3 and 8.4. In this example the called Ch function `func`() has two arguments of array of assumed-shape. Every element of these two arrays is doubled by `func`(). When the function is called in the dynamically loaded object to handle C arrays `a` and `b`, every element of these two arrays is also doubled.

8.2 Calling Ch Functions with Arguments of Arrays of Reference

In this section we will describe how to call Ch function with argument of arrays of reference from C space. Assume that the Ch function `func2`() shown below takes an array of reference with one dimension as argument.

```
int func2(array int a[&]) {
...
}
```

In C space, the C function `cfun2`() in which the Ch function `func2`() is invoked can be defined as follows.

```
int cfun2(void *chfunptr){
   int dim, ext1, ext2;
   int retval;
   int aa1[2][3] = { 1, 2, 3,
                     4, 5, 6};
   dim = 2, ext1 = 2; ext2 = 3; /* dim and extents of aa1 */
   Ch_CallFuncByAddr(interp, chfunptr, &retval, aa1, ext1, ext2);
return 0;
}
```

```
#include<dlfcn.h>
#include<array.h>

int callchvla_2d(int (*fun)(int a[:][:])) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libcallchvla.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }
  fptr = dlsym(handle, "callchvla_2d_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, callchvla_2d, fun);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}

int sum2d(int a[:][:]) { /* function to be called from C space */
  int i, j, retval = 0;
  array int ext[2] = shape(a);

  printf("ext[0] = %d\n", ext[0]);
  printf("ext[1] = %d\n", ext[1]);
  for(i = 0; i < ext[0]; i++)
    for(j = 0; j < ext[1]; j++)
       retval += a[i][j];

  return retval;
}

int main() {
  int sum_ret;
  int a[3][4] = { 1, 2, 3, 4,
                  5, 6, 7, 8,
                  9, 10, 11, 12 };

  sum_ret = sum2d(a);
  printf("sum2d has been called from Ch space and returned %d\n\n", sum_ret);

  callchvla_2d(sum2d);
  return 0;
}
```

Program 8.1: Example of calling Ch function with arguments of assumed-shape arrays from C function (callchvla2d.ch).

```
#include<stdio.h>
#include<ch.h>

int cfun1(ChInterp_t interp, void *chfunptr){
   int dim, ext1, ext2;
   int sum_ret;
   int aa1[2][3] = { 1, 2, 3,
                     4, 5, 6};

   int aa2[3][4] = { 1, 2, 3, 4,
                     5, 6, 7, 8,
                     9, 10, 11, 12};

   dim = 2, ext1 = 2; ext2 = 3; /* dim and extents of aa1 */
   Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa1, dim, ext1, ext2);
   printf("sum2d has been called from C space and returned %d\n\n", sum_ret);

   dim = 2, ext1 = 3; ext2 = 4; /* dim and extents of aa2 */
   Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa2, dim, ext1, ext2);
   printf("sum2d has been called from C space and returned %d\n", sum_ret);

   return 0;
}

EXPORTCH int callchvla_2d_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int retval;
  void *sum2d_ch_fun;

  Ch_VaStart(interp, ap, varg);
  sum2d_ch_fun = Ch_VaArg(interp, ap, void *);

  retval = cfun1(interp, sum2d_ch_fun);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 8.2: Example of calling Ch function with arguments of assumed-shape arrays from C function (callchvla2d.c).

```
ext[0] = 3
ext[1] = 4
sum2d has been called from Ch space and returned 78

ext[0] = 2
ext[1] = 3
sum2d has been called from C space and returned 21

ext[0] = 3
ext[1] = 4
sum2d has been called from C space and returned 78
```

Figure 8.1: Output from executing callchvla2d.ch.

```
#include <dlfcn.h>
#include <array.h>

int chcallfuncbyaddr_1(int (*func)(int a[:], int b[:][:])) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libch.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }

  fptr = dlsym(handle, "chcallfuncbyaddr_1_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, chcallfuncbyaddr_1, func);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }

  return retval;
}

int func(int a[:], int b[:][:]) {  /* a and b are arrays of assumed-shape*/
  int j, k;
  int n = (int) shape(a);
  array int dim[2] = shape(b);
  int n1 = dim[0], m1 = dim[1];

  printf("The dimensions, numbers of elements and values of array a and b \
in func is:\n");
  printf("n  = %d\n", n);
  printf("n1 = %d\n", n1);
  printf("m1 = %d\n", m1);
  for(j=0; j<n; j++) {
    printf("a[%i] = %d\n", j, a[j]);
    a[j] *= 2;
  }
  for(j=0; j<n1; j++) {
    for(k=0; k<m1; k++) {
      printf("b[%i][%i] = %d\n", j, k, b[j][k]);
      b[j][k] *= 2;
    }
  }
  return 0;
}

int main() {
  chcallfuncbyaddr_1(func);
}
```

Program 8.3: Example 2 of calling Ch function with arguments of assumed-shape arrays from C function (Ch application).

```
#include<ch.h>
#include<stdio.h>

int chcallfuncbyaddr_1(ChInterp_t interp, void * func_chdl_fptr) {
  int retval, j, k;
  int a[3] = {1,2,3,};
  int b[3][4] = {1,2,3,4,
                 5,6,7,8,
                 9, 10, 11, 12};

  Ch_CallFuncByAddr(interp, func_chdl_fptr, &retval, a, 1, 3, b, 2, 3, 4);

  printf("\nIn the dynamically loaded object:\n");
  for(j=0; j<3; j++)
    printf("a[%i] = %d\n", j, a[j]);
  for(j=0; j<3; j++) {
    for(k=0; k<4; k++) {
      printf("b[%i][%i] = %d\n", j, k, b[j][k]);
    }
  }
  return 0;
}

EXPORTCH int chcallfuncbyaddr_1_chdl(void *varg) {
  ChInterp_t  interp;
  void *func_chdl_fptr;
  ChVaList_t ap;
  int retval;
  Ch_VaStart(interp, ap, varg);
  func_chdl_fptr = Ch_VaArg(interp, ap, void *);   /* get pointer of func  */
  retval = chcallfuncbyaddr_1(interp, func_chdl_fptr);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 8.4: Example 2 of calling Ch function with arguments of assumed-shape arrays from C function (chdl and C functions).

```
The dimensions, numbers of elements and values of array a and b in func is:
n  = 3
n1 = 3
m1 = 4
a[0] = 1
a[1] = 2
a[2] = 3
b[0][0] = 1
b[0][1] = 2
b[0][2] = 3
b[0][3] = 4
b[1][0] = 5
b[1][1] = 6
b[1][2] = 7
b[1][3] = 8
b[2][0] = 9
b[2][1] = 10
b[2][2] = 11
b[2][3] = 12

In the dynamically loaded object:
a[0] = 2
a[1] = 4
a[2] = 6
b[0][0] = 2
b[0][1] = 4
b[0][2] = 6
b[0][3] = 8
b[1][0] = 10
b[1][1] = 12
b[1][2] = 14
b[1][3] = 16
b[2][0] = 18
b[2][1] = 20
b[2][2] = 22
b[2][3] = 24
```

Figure 8.2: Output from executing Program 8.3.

The function **Ch_CallFuncByAddr**() is used to call the Ch function `func2`(), which is pointed to by the second argument, `chfunptr`. The way of calling Ch function with arguments of array of reference with fixed dimension is exactly the same as calling Ch function with arguments of assumed-shape array which has been described in the previous section. The extra arguments providing information of extents of arrays need to be passed to the Ch function. In *Ch User's Guide*, it is described that arrays with different data type can be handled by the same function with argument of array of reference. This feature cannot apply to embedded Ch. The array in C space should be the same data type as the argument of the Ch function to be called. In other words, only arrays of type `int` in C space can be passed to Ch function `func2`().

Program 8.5 is another version of Ch function `sum2d`() which takes an argument of array of reference with the fixed dimension. The C program in Program 8.2 still works with this Ch function. The output is the same as Figure 8.1.

If the Ch function `func3`() shown below takes array of reference without constraint of dimension as the argument, arrays with different dimension can be passed to this function.

```
int func3(array int &a) {
...
}
```

To call this Ch function from C space, the users need to provide not only information of extents, like what we have done in the previous section, but also the number of dimensions. If the array in C space has 2 dimension, the C function `cfun3`(), in which the Ch function `func3`() is called by address, can defined as follows.

```
int cfun3(void *chfunptr){
   int dim, ext1, ext2;
   int retval;
   int aa[3][4] = { 1, 2, 3, 4,
                    5, 6, 7, 8,
                    9, 10, 11, 12};
  /* number of dimensions of aa are 2, extents are 3 and 4 */
   dim = 2; ext1 = 3; ext2 = 4;
   Ch_CallFuncByAddr(interp, chfunptr, &retval, aa, dim, ext1, ext2);
   return 0;
}
```

where `chfunptr` is the pointer to the Ch function to be called. The array `aa` is a two-dimension array with extents of 3 and 4. In the argument list of **Ch_CallFuncByAddr**(), three extra arguments, `dim`, `ext1` and `ext2`, which contains number of dimensions and extents of `aa`, have been added and passed to the Ch function `func3`().

The complete sample of this case is shown in Programs 8.6 and 8.7. The Ch function `sumNd`() can calculate the sum of each element of array `a` with different dimension. The statement

```
aa = (array int [totnum])a;
```

casts array `a` which can have any dimensions to the one-dimensional array `aa` with `totnum` elements. In function `main`(), the function `sumNd`() is called to calculate the sum of array `a`, which is a two-dimensional array. Then the function pointer to `sumNd`() is passed to C space by the function call

```
callchvla_Nd(sumNd);
```

In C space (Program 8.7), the function pointer passed from Ch space is obtained by

```
#include<dlfcn.h>
#include<array.h>

int callchvla_2d(int (*fun)(int a[&][&])) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libcallchvla.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }
  fptr = dlsym(handle, "callchvla_2d_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, callchvla_2d, fun);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}

int sum2d(array int a[&][&]) { /* function to be called from C space */
  int i, j, retval = 0;

  array int ext[2] = shape(a);
  int n = ext[0], m = ext[1];
  array int aa[n][m];
  aa = a;

  printf("ext[0] = %d\n", ext[0]);
  printf("ext[1] = %d\n", ext[1]);
  for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
       retval += aa[i][j];

  return retval;
}

int main() {
  int sum_ret;
  int a[3][4] = { 1, 2, 3, 4,
                  5, 6, 7, 8,
                  9, 10, 11, 12 };

  sum_ret = sum2d(a);
  printf("sum2d has been called from Ch space and returned %d\n\n", sum_ret);

  callchvla_2d(sum2d);
  return 0;
}
```

Program 8.5: Example of calling Ch functions with arguments of array of reference with fixed dimension (callchvla2d.ch).

```
sumNd_ch_fun = Ch_VaArg(interp, ap, void *);
```

Then the C functions `cfun3`() calls this Ch function twice to calculate sum of two arrays of different dimensions. For array `aa1` with the shape of (3×4), three extra arguments, `dim`, `ext1` and `ext2`, which represent dimensions and extents, are added following the argument `aa1`. It is shown as follows.

```
dim = 2; ext1 = 3; ext2 = 4; /* dimension and extents of aa1 */
Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa1, dim, ext1, ext2);
```

For the array `aa2` which has the shape of $(2 \times 3 \times 4)$, four extra arguments, the dimension `dim` and extents `ext1`, `ext2` and `ext3`, are added after the argument `aa2`. It is shown as follows.

```
dim=3; ext1=2; ext2=3; ext3=4; /* dimension and extents of aa2 */
Ch_CallFuncByAddr(interp, chfunptr,&sum_ret,aa2,dim,ext1,ext2,ext3);
```

The output from executing this example is shown in Figure 8.3.

```
#include<dlfcn.h>
#include<array.h>

int callchvla_Nd(int (*fun)(array int &a)) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libcallchvla.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }
  fptr = dlsym(handle, "callchvla_Nd_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }
  dlrunfun(fptr, &retval, callchvla_Nd, fun);
  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}

int sumNd(array int &a) { /* function to be called from C space */
  int i, totnum = 1, retval = 0;
  int dim = (int)shape(shape(a));
  array int ext[dim];

  ext = shape(a);
  for(i = 0; i < dim; i++) {
    printf("ext[%i] = %d\n", i, ext[i]);
    totnum *= ext[i];
  }
  array int aa [totnum];
  aa = (array int [totnum])a;
  for(i = 0; i < totnum; i++) {
     retval += aa[i];
  }
  return retval;
}

int main() {
  int sum_ret;
  int a[3][4] = { 1, 2, 3, 4,
                  5, 6, 7, 8,
                  9, 10, 11, 12};

  sum_ret = sumNd(a);
  printf("sumNd has been called from Ch space and returned %d\n\n", sum_ret);
  callchvla_Nd(sumNd);
  return 0;
}
```

Program 8.6: Example of calling Ch function with arguments of arrays of reference without constraint of dimension from C function (callchvlaNd.ch).

```
#include<stdio.h>
#include<ch.h>

int cfun3(ChInterp_t interp, void *chfunptr){
   int dim, ext1, ext2, ext3;
   int sum_ret;
   int aa1[3][4] = { 1, 2, 3, 4,
                     5, 6, 7, 8,
                     9, 10, 11, 12};

   int aa2[2][3][4] = { 1, 2, 3, 4,
                        5, 6, 7, 8,
                        9, 10, 11, 12,

                        1, 2, 3, 4,
                        5, 6, 7, 8,
                        9, 10, 11, 12};

   dim = 2; ext1 = 3; ext2 = 4; /* dimension and extents of aa1 */
   Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa1, dim, ext1, ext2);
   printf("sumNd has been called from C space and returned %d\n\n", sum_ret);

   dim = 3; ext1 = 2; ext2 = 3; ext3 = 4; /* dimension and extents of aa2 */
   Ch_CallFuncByAddr(interp, chfunptr, &sum_ret, aa2, dim, ext1, ext2, ext3);
   printf("sumNd has been called from C space and returned %d\n", sum_ret);

   return 0;
}

EXPORTCH int  callchvla_Nd_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int retval;
  void *sumNd_ch_fun;

  Ch_VaStart(interp, ap, varg);
  sumNd_ch_fun = Ch_VaArg(interp, ap, void *);

  retval = cfun3(interp, sumNd_ch_fun);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 8.7: Example of calling Ch function with arguments of arrays of reference from C function (callchvlaNd.c).

```
ext[0] = 3
ext[1] = 4
sumNd has been called from Ch space and returned 78

ext[0] = 3
ext[1] = 4
sumNd has been called from C space and returned 78

ext[0] = 2
ext[1] = 3
ext[2] = 4
sumNd has been called from C space and returned 156
```

Figure 8.3: Output from executing callchvlaNd.ch.

Appendix A

Functions for Dynamically Loaded Library —<ch.h>

The header file **ch.h** defines macros and function prototypes used to build Dynamically Loaded Libraries (DLL) to work with Ch.

Data Types

The following data type are declared in the header file **ch.h**.

Data Type	Description
ChInterp_t	A Ch interperer. The first argument for all Ch SDK APIs.
ChType_t	The internal Ch data type.
The internal Ch data type.	
ChVaList_t	To represent in the C space for a variable number of
¿ arguments in the Ch space.	

Functions

The following functions are defined inside the header file **ch.h**.

Function	Description
Ch_CallFuncByAddr	Call a Ch function by its address from the C address space.
Ch_CallFuncByName	Call a Ch function by its name from the C address space.
Ch_CallFuncByNameVar	Call a Ch function by its name from the C address space with a variable number of arguments.
Ch_CppChangeThisPointer	Change *this* pointer of an instance of a class.
Ch_CppIsArrayElement	Check if the destructor is called by an elment of an array of class.
Ch_GetSymbol	Get the address of a global variable in a Ch program. Deprecated, use **Ch_SymbolAddrByName**.
Ch_Home	Get the home directory in which the Ch is installed
Ch_SymbolAddrByName	Get the address of a global variable based on its name.

Ch_VaArg	Obtain a value of a function argument in Ch passed to a dynamically loaded library.
Ch_VaArrayDim	Obtain the dimension of argument of array.
Ch_VaArrayExtent	Obtain the number of elements in the specified dimension of
Ch_VaArrayType	Determine if the argument in the argument list is an array. the passed argument.
Ch_VaCount	Obtain the number of the arguments to be processed by **Ch_VaArg**().
Ch_VaDataType	Obtain the data type of an argument in the argument list.
Ch_VaElementtype	Obtain the data type of an argument in the argument list. Deprecated, use **Ch_VaDataType**.
Ch_VaEnd	For a normal return from a function.
Ch_VaFuncArgDataType	Obtain the data type of an argument of the function which is pointed to by an argument of function pointer.
Ch_VaFuncArgNum	Obtain the number of the arguments of the function which is pointed to by an argument of function pointer.
Ch_VaIsFunc	Determine if the argument in the argument list is function type.
Ch_VaIsFuncVarArg	Determine if the argument in the argument list is function with a variable number of arguments.
Ch_VarArgsAddArg	Add an argument into a Ch style variable argument list.
Ch_VarArgsAddArgExpr	Add an argument in expression into a Ch style variable argument list.
Ch_VarArgsCreate	Create a Ch style variable argument list initially.
Ch_VarArgsDelete	Delete a Ch style variable argument list and release its memory.
Ch_VaStart	Obtain an instance of Ch interpreter and initialize a *handle* for reference to arguments of a function in Ch.
Ch_VaStructAddr	Obtain the address of the argument of struct. Deprecated, use **Ch_VaUserDefinedAddr**.
Ch_VaStructSize	Obtain the size of the argument of struct. Deprecated, use **Ch_VaUserDefinedSize**.
Ch_VaUserDefinedAddr	Obtain the address of the user defined data type of the argument in the argument list.
Ch_VaUserDefinedName	Obtain the name of the user defined data type of the argument.
Ch_VaUserDefinedSize	Obtain the size of the user defined data type of the argument.
Ch_VaVarArgsCreate	Create a C style variable argument list according to the Ch style variable argument list.
Ch_VaVarArgsDelete	Delete memory allocated in function **Ch_VaVarArgsCreate**().

Macros

The data type **ChType_t** defined inside the header file **ch.h** has the following values.

Value	Description
CH_UNDEFINETYPE	Undefined type.
CH_CHARTYPE	char type.
CH_UCHARTYPE	unsigned char type.
CH_SHORTTYPE	short int type.

CH_USHORTTYPE	unsigned short int type.
CH_INTTYPE	int/long int type.
CH_UINTTYPE	unsigned int type.
CH_LLINTTYPE	long long int type.
CH_ULLINTTYPE	unsigned long long int type.
CH_FLOATTYPE	float type.
CH_DOUBLETYPE	double type.
CH_LDOUBLETYPE	long double type.
CH_COMPLEXTYPE	float complex type.
CH_LCOMPLEXTYPE	long complex type.
CH_STRINGTYPE	string_t type.
CH_FILETYPE	FILE type.
CH_VOIDTYPE	void type.
CH_PROCTYPE	function type.
CH_STRUCTTYPE	struct type.
CH_CLASSTYPE	class type.
CH_UNIONTYPE	union type.
CH_ENUMTYPE	enum type.
CH_CARRAYTYPE	C array type.
CH_CARRAYPTRTYPE	pointer to C array.
CH_CARRAYVLATYPE	C VLA array.
CH_CHARRAYTYPE	Ch array type.
CH_CHARRAYPTRTYPE	pointer to Ch array.
CH_CHARRAYVLATYPE	Ch VLA array.

CH_NULLTYPE	NULL pointer type.
CH_VOIDPTRTYPE	pointer to void type.
CH_CHARPTRTYPE	pointer to char type.
CH_UCHARPTRTYPE	pointer to unsigned char type.
CH_SHORTPTRTYPE	pointer to short int type.
CH_USHORTPTRTYPE	pointer to unsigned short int type.
CH_INTPTRTYPE	pointer to int/long int type.
CH_UINTPTRTYPE	pointer to unsigned int type.
CH_LLINTPTRTYPE	pointer to long long int type.
CH_ULLINTPTRTYPE	pointer to unsigned long long int type.
CH_FLOATPTRTYPE	pointer to float type.
CH_DOUBLEPTRTYPE	pointer to double type.
CH_LDOUBLEPTRTYPE	pointer to long double type.
CH_COMPLEXPTRTYPE	pointer to float complex type.
CH_LCOMPLEXPTRTYPE	pointer to long complex type.
CH_STRINGPTRTYPE	pointer to string_t type.
CH_PROCPTRTYPE	function pointer type.
CH_FILEPTRTYPE	pointer to FILE type.
CH_STRUCTPTRTYPE	pointer to struct type.
CH_CLASSPTRTYPE	pointer to class type.
CH_UNIONPTRTYPE	pointer to union type.
CH_ENUMPTRTYPE	pointer to enum type.

CH_VOIDPTR2TYPE	pointer to pointer to void type.
CH_CHARPTR2TYPE	pointer to pointer to char type.
CH_UCHARPTR2TYPE	pointer to pointer to unsigned char type.
CH_SHORTPTR2TYPE	pointer to pointer to short int type.
CH_USHORTPTR2TYPE	pointer to pointer to unsigned short int type.
CH_INTPTR2TYPE	pointer to pointer to int/long int type.
CH_UINTPTR2TYPE	pointer to pointer to unsigned int type.
CH_LLINTPTR2TYPE	pointer to pointer to long long int type.
CH_ULLINTPTR2TYPE	pointer to pointer to unsigned long long int type.
CH_FLOATPTR2TYPE	pointer to pointer to float type.
CH_DOUBLEPTR2TYPE	pointer to pointer to double type.
CH_LDOUBLEPTR2TYPE	pointer to pointer to long double type.
CH_COMPLEXPTR2TYPE	pointer to pointer to float complex type.
CH_LCOMPLEXPTR2TYPE	pointer to pointer to long complex type.
CH_STRINGPTR2TYPE	pointer to pointer to string_t type.
CH_FILEPTR2TYPE	pointer to pointer to FILE type.
CH_STRUCTPTR2TYPE	pointer to pointer to struct type.
CH_CLASSPTR2TYPE	pointer to pointer to class type.
CH_UNIONPTR2TYPE	pointer to pointer to union type.
CH_ENUMPTR2TYPE	pointer to pointer to enum type.

The return value for most Ch SDK APIs defined inside the header file **ch.h** has the following values.

Value	Description
CH_ERROR	-1 for failure of a Ch SDK API call.
CH_OK	0 for succeess of a Ch SDK API call.

Ch_CallFuncByAddr

Synopsis
#include <**ch.h**>
int Ch_CallFuncByAddr(**ChInterp_t** *interp*, **void** **fptr*, **void** **retval*, ...);

Purpose
Call a Ch function by its address from the C address space.

Return Value
This function returns **CH_OK** on success, **CH_ERROR** on failure, **CH_ABORT** when **Ch_Abort()** in Embedded Ch was called.

Parameters
interp A Ch interpreter.

fprt A pointer to the function in the Ch space.

retval The address to store the return value of the called function.

Description
The function **Ch_CallFuncByAddr**() calls a Ch function by its address from the C address space. The argument `retval` contains the address to store the returned value of the called function. In case that the function returns a computational array, it contains the address to store the elements of the returned array. The number of the arguments after *retval* in the argument list depends on that of the called function. The function **Ch_CallFuncByName**() calls a Ch function by its name.

Example 1
This example calculates the function values of an array. The function $y = x^2$ in Ch program is called from the dynamically loaded object. The Makefile for compiling and linking to create a dynamically loaded library **libch.dl** for all examples is shown in Example 1 of function **Ch_SymbolAddrByName**() on page293.

Listing 1 — A C program to create a dynamically loaded library

```
#include<ch.h>
#include<stdio.h>

int chcallfuncbyaddr(double x[], double y[], double (*func)(double), int n) {
  int i;
  double retval;
  for (i = 0; i < n; i++)
    y[i] = func(x[i]);
  return 0;
}

static ChInterp_t interp;
static void *func_chdl_funptr;
static double func_chdl_funarg(double xx) {
  double retval;
  Ch_CallFuncByAddr(interp, func_chdl_funptr, &retval, xx);
  return retval;
}
```

```
EXPORTCH int chcallfuncbyaddr_chdl(void *varg) {
  ChVaList_t ap;
  double *x, *y;
  int i, n, retval;

  Ch_VaStart(interp, ap, varg);
  n = Ch_VaArrayExtent(interp, ap, 0);
  x = Ch_VaArg(interp, ap, double*);
  y = Ch_VaArg(interp, ap, double*);
  func_chdl_funptr = Ch_VaArg(interp, ap, void *);
  retval = chcallfuncbyaddr(x, y, func_chdl_funarg, n);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Listing 2 — A Ch application program

```
#include <dlfcn.h>
#include <array.h>

int chcallfuncbyaddr(double x[:], double y[:], double (*func)(double)) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libch.dl", RTLD_LAZY);
  if(handle == NULL) {
       printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
       return -1;
  }
  fptr = dlsym(handle, "chcallfuncbyaddr_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }
  dlrunfun(fptr, &retval, chcallfuncbyaddr, x, y, func);
  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }

  return retval;
}

double  func(double a){
  return a*a;
}

int main() {
  array double x[4] = {1,2,3,4}, y[4];

  chcallfuncbyaddr(x, y, func);
  printf("x = %f\n", x);
  printf("y = %f\n", y);
}
```

Output

```
x = 1.000000 2.000000 3.000000 4.000000

y = 1.000000 4.000000 9.000000 16.000000
```

See Also
Ch_CallFuncByName(), **Ch_CallFuncByNameVar**(), **Ch_SymbolAddrByName**(), **Ch_VarArgsAddArg**(), **Ch_VarArgsAddArgExpr**(), **Ch_VarArgsCreate**(), **Ch_VarArgsDelete**().

Ch_CallFuncByName

Synopsis
#include <**ch.h**>
int Ch_CallFuncByName(**ChInterp_t** *interp*, **const char** **name*, **void** **retval*, ...);

Purpose
Call a Ch function by its name from the C address space.

Return Value
This function returns **CH_OK** on success, **CH_ERROR** on failure, **CH_ABORT** when **Ch_Abort()** in Embedded Ch was called.

Parameters
interp A Ch interpreter.

name The name of the called function.

retval The address to store the return value of the called function.

Description
The function **Ch_CallFuncByName**() calls a Ch function by its name from the C address space. The function `name` can be a function located in one of function files specified by the function file path_ `fpath`. For a generic function, its regular function version with a function prototype will be used. For example, for function $\sin()$, the regular function

```
double sin(double);
```

will be used if `sin` is the function name for **Ch_CallFuncByName**(). The argument `retval` contains the address to store the returned value of the called function. In case that the function returns a computational array, it contains the address to store the elements of the returned array. The number of the arguments after *retval* in the argument list depends on that of the called function. The function **Ch_CallFuncByAddr**() calls a Ch function by its address.

Example 1
In this example, the Chfunction `chfun()` is called by its name from the C function `chcallfuncbyname_chdl`(). The Makefile for compiling and linking to create a dynamically loaded library **libch.dl** for all examples is shown in Example 1 of function **Ch_SymbolAddrByName**() on page 293.

Listing 1 — A C program to create a dynamically loaded library

```
#include<ch.h>
#include<stdio.h>

EXPORTCH void chcallfuncbyname_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  void *chfunhandle;
  int funret;
  double d;

  Ch_VaStart(interp, ap, varg);
```

```
  /* call Ch function by its name */
  Ch_CallFuncByName(interp, "chfun", &funret, 20);
  printf("The return value in C space from Ch function chfun(20) is %d\n", funret);
  /* call Ch function in function file */
  Ch_CallFuncByName(interp, "hypot", &d, 3.0, 4.0);
  printf("The return value in C space from Ch function hypot(3,4) is %f\n", d);
  Ch_VaEnd(interp, ap);
  return;
}
```

Listing 2 — A Ch application program

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

void chcallfuncbyname() {

  void *dlhandle, *fptr;

  dlhandle = dlopen("libch.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }

  fptr = dlsym(dlhandle, "chcallfuncbyname_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  dlrunfun(fptr, NULL, chcallfuncbyname);

  if(dlclose(dlhandle)!=0) {
    printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
    return;
  }

  return;
}

/* this function will be called from a C function */
int chfun(int arg1) {

  printf("in the Ch function chfun(), arg1 = %d\n", arg1);
  return arg1 * arg1;
}

int ii1; // for return value of method 2
int main() {

  chcallfuncbyname();
  return 0;
}
```

Output 1

```
in the Ch function chfun(), arg1 = 20
The return value in C space from Ch function chfun(20) is 400
The return value in C space from Ch function hypot(3,4) is 5.000000
```

See Also
Ch_CallFuncByAddr(), **Ch_CallFuncByNameVar**(), **Ch_SymbolAddrByName**(), **Ch_VarArgsAddArg**(), **Ch_VarArgsAddArgExpr**(), **Ch_VarArgsCreate**(), **Ch_VarArgsDelete**().

Ch_CallFuncByNameVar

Synopsis
#include <**ch.h**>
int Ch_CallFuncByNameVar(**ChInterp_t** *interp*, **const char** **name*, **void** **retval*, **void** **arglist*);

Purpose
Call a Ch function by its name from the C address space.

Return Value
This function returns **CH_OK** on success, **CH_ERROR** on failure, **CH_ABORT** when **Ch_Abort()** in Embedded Ch was called.

Parameters
interp A Ch interpreter.

name The name of the called function.

retval The address to store the return value of the called function.

arglist The argument list for the function.

Description
The function **Ch_CallFuncByNameVar**() calls a Ch function by its `name` from the C address space. The function `name` can be a function located in one of function files specified by the function file path_ `fpath`. For a generic function, its regular function version with a function prototype will be used. For example, for function sin(), the regular function

```
double sin(double);
```

will be used if `sin` is the function name for **Ch_CallFuncByNameVar**(). The return value of the called function in the Ch space is passed back by argument `retval`. The argument `retval` contains the address to store the returned value of the called function. In case that the function returns a computational array, it contains the address to store the elements of the returned array. The arguments of the called function are passed in the fourth argument `arglis`. The argument `arglis` is built dynamically using functions **Ch_VarArgsAddArg**(), **Ch_VarArgsAddArgExpr**(), and **Ch_VarArgsCreate**(). Later, it shall be deleted by function **Ch_VarArgsDelete**().

Example
See examples in reference for **Ch_VarArgsAddArg**() and **Ch_VarArgsAddArgExpr**().

See Also
Ch_CallFuncByAddr(), **Ch_CallFuncByName**(), **Ch_SymbolAddrByName**(), **Ch_VarArgsAddArg**(), **Ch_VarArgsAddArgExpr**(), **Ch_VarArgsCreate**(), **Ch_VarArgsDelete**().

Ch_CppChangeThisPointer

Synopsis
#include <**ch.h**>
int Ch_CppChangeThisPointer(**ChInterp_t** *interp*, **void** **c*, **size_t** *size*);

Purpose
Change the *this* pointer of an instance of a class in the Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

c A pointer to an instance of a class in the C++ space.

size The size of the class in the C++ space.

Description
The function **Ch_CppChangeThisPointer**() changes the *this* pointer of an instance of a class in the Ch space. The *this* pointer of an instance of a class in the Ch space will point to an instance of class in the C++ space passed as the argument `c`. The size of the class in the Ch space will also be changed to the size of the class in the C++ space passed as the argument `size`. This function is required for interface a class in the C++ space.

Restriction
The function must be called by a constructor of a class in the Ch space.

Example
See chapter 7.

See Also
Ch_CppIsArrayElement().

Ch_CppIsArrayElement

Synopsis
#include <**ch.h**>
int Ch_CppIsArrayElement(**ChInterp_t** *interp*);

Purpose
Test if the destructor of a class in the Ch space is called by an element of an array of class.

Return Value
This function returns 1 if the destructor of a class in the Ch space is called by an element of an array of class. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

Description
The function **Ch_CppIsArrayElement**() test if the destructor of a class in the Ch space is called by an element of an array of class. This function is required for interface a class in the C++ space.

Restriction
The function must be called by a destructor of a class in the Ch space.

Example
See chapter 7.

See Also
Ch_CppChangeThisPointer().

Ch_Home

Synopsis
#include <**ch.h**>
void ***Ch_Home**(**ChInterp_t** *interp*);

Purpose
Get the home directory of Ch where Ch is installed, such as `/usr/ch` or `C:/Ch`.

Return Value
This function returns the home directory of Ch.

Parameters

interp A Ch interpreter.

Description
The function **Ch_Home**() gets the home directory of Ch where Ch is installed, such as `/usr/ch` or `C:/Ch`.

Example
In the C program shown below, **Ch_Home**() is called to get the home directory of Ch in chhome.c.

chhome.c —- A C program to get the home directory of Ch

```
/*********************************************
* Get home directory of Ch
*********************************************/
#include<stdio.h>
#include<ch.h>

EXPORTCH void chhome_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  char *path = NULL;

  Ch_VaStart(interp, ap, varg);
  path = Ch_Home(interp);
  printf("Home dir of Ch is %s\n", path);
  Ch_VaEnd(interp, ap);
  return;
}
```

Output

```
Home dir of CH is /usr/ch
```

See Also
Ch_SymbolAddrByName(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**().

Ch_SymbolAddrByName

Synopsis
#include <**ch.h**>
void *__**Ch_SymbolAddrByName**(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Get the address of a global variable in a Ch program.

Return Value
The address of a global variable in a Ch program. If `name` is not in the list of the symbol table the function returns NULL. If `name` is a tag name for class, structure, or union, the function returns NULL. For a variable of function prototype without function definition or an extern variable without definition, the function returns NULL. For a variable of C array or a pointer to C array, the returned value is the pointer to the address of the first element. For example, for variable of a and p declared in Ch

```
int a[3], *p;
p = a;
```

the returned values for both p and a are the same of pointer to pointer to int. For a variable of computational array type, the reurned value for **Ch_SymbolAddrByName**() is the address for the first element of the computational array.

Parameters
interp A Ch interpreter.

name The name of global variable in a Ch program.

Description
The function **Ch_SymbolAddrByName**() obtains the address of a global variable inside a dynamically loaded library or embedded Ch program. The global variable can be valid data type in Ch. The value of a global variable in Ch can be changed in a dynamically loaded object. This function can be used independently with other functions defined in header file **ch.h**.

Example
The addresses of global variables *ch_i* and *ch_f* are obtained in the dynamically loaded object by calling function **Ch_SymbolAddrByName**(). If the values pointed to by these two variables are changed in the object, the variables printed out in the application Ch program are also changed. Variable *ch_p* of pointer to int is pointed to variable *ch_i* in Ch space. The file `Makefile` building a dynamically loaded library `libch.dl` for samples of functions listed in header file **ch.h** is shown in **Listing 3** .

Listing 1 — A C program to create dynamically loaded library

```
#include<ch.h>
#include<stdio.h>

EXPORTCH int chsymboladdrbyname_chdl(void *varg) {
  ChInterp_t  interp;
```

```
ChVaList_t ap;
int *ch_i;
int **ch_p;
float *ch_f;
int *ch_a, **ch_aptr;
char **str;

Ch_VaStart(interp, ap, varg);
/* ch_i shall be an extern variable in the calling Ch program */
ch_i = (int *)Ch_SymbolAddrByName(interp, "ch_i");
if(ch_i!=NULL) {
   printf("ch_i value in dynamically loaded object passed from Ch program is: %d\n",
      *ch_i);
  *ch_i = 1000; /* global variable ch_i is assigned with another value */
}
else {
   printf("ch_i is not defined in Ch program\n");
}

ch_p = (int **)Ch_SymbolAddrByName(interp, "ch_p");
if(ch_p!=NULL) {
   if(*ch_p != NULL)  {
      printf("ch_p value in dynamically loaded object passed from Ch program is: %p\n",
      *ch_p);
      printf("*ch_p value in dynamically loaded object passed from Ch program is: %d\n",
      **ch_p);
   }
   else {
      printf("ch_p in Ch space is NULL\n");
   }
}
else {
   printf("ch_p is not defined in Ch program\n");
}

/* ch_f shall be an extern variable in the calling Ch program */
ch_f = (float *)Ch_SymbolAddrByName(interp, "ch_f");
printf("ch_f value in dynamically loaded object passed from Ch program is: %f\n",
      *ch_f);

*ch_f = 2000; /* global variable ch_f is assigned with another value */

/* ch_aptr is a pointer to pointer to int */
ch_aptr = (int **)Ch_SymbolAddrByName(interp, "ch_a");
ch_a = *ch_aptr;
/* or ch_a = *(int **)Ch_SymbolAddrByName(interp, "ch_a");  */
printf("ch_a[0] = %d ch_a[1] = %d, ch_a[2] = %d\n",
       ch_a[0], ch_a[1], ch_a[2]);

/* ch_aptr is a pointer to pointer to int */
ch_aptr = (int **)Ch_SymbolAddrByName(interp, "a");
ch_a = *ch_aptr;
/* or ch_a = *(int **)Ch_SymbolAddrByName(interp, "ch_a");  */
printf("ch_a[0] = %d ch_a[1] = %d, ch_a[2] = %d\n",
       ch_a[0], ch_a[1], ch_a[2]);

str = *(char***)Ch_SymbolAddrByName(interp, "str");
printf("str[0] = %s str[1] = %s\n", str[0], str[1]);
```

```
  Ch_VaEnd(interp, ap);
  return 0;
}
```

Listing 2 — A Ch application program **chsymboladdrbyname.ch**

```
#include<dlfcn.h>

int chsymboladdrbyname() {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libch.dl", RTLD_LAZY);
  if(handle == NULL) {
       printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
       return -1;
  }
  fptr = dlsym(handle, "chsymboladdrbyname_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }
  dlrunfun(fptr, &retval, chsymboladdrbyname);
  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}

int ch_i;      /* ch_i is an extern variable */
int *ch_p;     /* ch_p is an extern variable */
float ch_f;    /* ch_f is an extern variable */
int *ch_a;
int a[3] = {10, 20, 30};
char *str[] = {"abcd", "ABCD", NULL};

int main() {
  int retval;

  ch_i = 10;
  ch_p = &ch_i;
  ch_f = 20;
  ch_a = a;

  printf("&ch_i in Ch space: %p\n", &ch_i);
  chsymboladdrbyname();
  printf("ch_i value is re-assigned by dynamically loaded object to: %d\n", ch_i);
  printf("ch_f value is re-assigned by dynamically loaded object to: %f\n", ch_f);
}
```

Listing 3 — Makefile

```
# build dynamically loaded  lib libch.dl and libchcpp.dl

target: Makefile libch.dl libchcpp.dl

libch.dl: Makefile chvastart.o chsymboladdrbyname.o chcallfuncbyaddr.o \
          chcallfuncbyaddr_1.o chcallfuncbyaddr_2.o vararg.o
        ch dllink libch.dl chvastart.o chsymboladdrbyname.o chcallfuncbyaddr.o \
```

```
          chcallfuncbyaddr_1.o chcallfuncbyaddr_2.o chvaisfuncvararg.o vararg.o
libchcpp.dl: Makefile chuserdefinedname.o
        ch dllink libchcpp.dl cplusplus chuserdefinedname.o

chvastart.o:  chvastart.c
        ch dlcomp libch.dl chvastart.c
chcallfuncbyaddr.o:  chcallfuncbyaddr.c
        ch dlcomp libch.dl chcallfuncbyaddr.c
chsymboladdrbyname.o:  chsymboladdrbyname.c
        ch dlcomp libch.dl chsymboladdrbyname.c
chcallfuncbyaddr_1.o:  chcallfuncbyaddr_1.c
        ch dlcomp libch.dl chcallfuncbyaddr_1.c
chcallfuncbyaddr_2.o:  chcallfuncbyaddr_2.c
        ch dlcomp libch.dl chcallfuncbyaddr_2.c
chvaisfuncvararg.o:  chvaisfuncvararg.c
        ch dlcomp libch.dl chvaisfuncvararg.c
vararg.o:  vararg.c
        ch dlcomp libch.dl vararg.c
chuserdefinedname.o:  chuserdefinedname.cpp
        ch dlcomp libch.dl cplusplus chuserdefinedname.cpp
```

Output

```
&ch_i in Ch space: 2d7298
ch_i value in dynamically loaded object passed from Ch program is: 10
ch_p value in dynamically loaded object passed from Ch program is: 2d7298
*ch_p value in dynamically loaded object passed from Ch program is: 1000
ch_f value in dynamically loaded object passed from Ch program is: 20.000000
ch_a[0] = 10 ch_a[1] = 20, ch_a[2] = 30
ch_a[0] = 10 ch_a[1] = 20, ch_a[2] = 30
str[0] = abcd str[1] = ABCD
ch_i value is re-assigned by dynamically loaded object to: 1000
ch_f value is re-assigned by dynamically loaded object to: 2000.000000
```

See Also

Ch_CallFuncByAddr(), **Ch_CallFuncByName**().

Ch_VaArg

Synopsis
#include <**ch.h**>
type **Ch_VaArg**(**ChInterp_t** *interp*, **ChVaList_t** *ap*, *type*);

Purpose
Obtain a value of a function argument passed from a Ch function call in a dynamically loaded library.

Return Value
This macro returns the value of an argument.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t** declared in header file **ch.h**. It represents in the C space for a variable number of arguments in the Ch space.

type The data type of passed argument.

Description
The **Ch_VaArg** macro expands an expression that has the specified type and the value of the argument in the call. The first invocation of the **Ch_VaArg** macro after that of the **Ch_VaStart** macro returns the value of the first argument. Successive invocations return the values of the remaining arguments in succession. The parameter *ap* of the type **ChVaList_t** shall be the same as that initialized by **Ch_VaStart**(). The parameter *type* shall be a type name specified for the argument.

Example
See **Ch_VaStart**().

See Also
Ch_VaArrayDim(), **Ch_VaDataType**(), **Ch_VaArrayExtent**(), **Ch_VaStart**(), **Ch_VaEnd**(), **stdarg.h**.

Ch_VaArrayDim

Synopsis
#include <**ch.h**>
int Ch_VaArrayDim(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the dimension of passed array argument.

Return Value
If the argument is of array type, This function returns the dimension of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t** declared in header file **ch.h**. It represents in the C space for a variable number of arguments in the Ch space.

Description
The function **Ch_VaArrayDim**() returns the dimension of array argument passed from a Ch function to dynamically loaded object. The extent of each dimension of the array can be obtained by function **Ch_VaArrayExtent**(). This function shall be called before function **Ch_VaArg**() is called to obtain the value of the passed argument.

Example
See **Ch_VaStart**().

See Also
Ch_VaDataType(), **Ch_VaArrayExtent**(), **Ch_VaStart**(), **Ch_VaEnd**(), **Ch_VaArg**().

Ch_VaArrayExtent

Synopsis
#include <**ch.h**>
int Ch_VaArrayExtent(**ChInterp_t** *interp*, **ChVaList_t** *ap*, **int** *dim*);

Purpose
Obtain the number of elements in the specified dimension of the passed array argument.

Return Value
If the argument is of array type, This function returns the number of elements in a specified dimension of the passed array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

dim An integer specifying for which dimension the number of elements will be obtained, 0 for the first dimension.

Description
The function **Ch_VaArrayExtent**() returns the number of elements in the specified dimension of the passed argument. For array `a[n][m]`, If `0` is passed to this function as the second argument, the extent of the first dimension, i.e. `n`, will be returned. Otherwise, if `1` is passed to, `m` will be returned. This function shall be called before the argument is obtained by function **Ch_VaArg**().

Example
See **Ch_VaStart**().

See Also
Ch_VaArrayDim(), **Ch_VaDataType**(), **Ch_VaStart**(), **Ch_VaEnd**(), **Ch_VaArg**().

Ch_VaArrayType

Synopsis
#include <**ch.h**>
ChType_t Ch_VaArrayType(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Determine if the argument is an array.

Return Value
Based on its argument, this function returns one of the macros below for data type **ChType_t**, defined in header file **ch.h**.

Macro	Description	Example
CH_UNDEFINETYPE	not an array.	`int i`
CH_CARRAYTYPE	C array	`int a[3]`
CH_CARRAYPTRTYPE	pointer to C array	`int (*ap)[3]`
CH_CARRAYVLATYPE	C VLA array	`int a[n]` `int func(int a[n], int b[:], int c[&])`
CH_CHARRAYTYPE	Ch array	`array int a[3]`
CH_CHARRAYPTRTYPE	pointer to Ch array	`array int (*ap)[3]`
CH_CHARRAYVLATYPE	Ch VLA array	`array int a[n]; int fun(array int a[n],` `array int b[:], array int c[&])`

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaArrayType**() determines if the argument is an array and its type. It shall be call before **Ch_VaArg**() is called to step *ap* to the next argument.

Example
see **Ch_VaStart**().

See Also
Ch_VaDataType(), **Ch_VaArrayDim**(), **Ch_VaArrayExtent**(), **Ch_VaStart**(), **Ch_VaEnd**(), **Ch_VaArg**().

Ch_VaCount

Synopsis
#include <**ch.h**>
int Ch_VaCount(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the number of the arguments which have not been processed by **Ch_VaArg**() yet.

Return Value
This function returns the number of the remaining arguments in the argument list.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaCount**() returns the number of the remaining arguments in the argument list. To obtain total number of arguments passed from the Ch function, **Ch_VaCount**() shall be call before **Ch_VaArg**() is called, because each function call of **Ch_VaArg**() returns one argument from the argument list and steps *ap* to the next one.

Example
See **Ch_VaStart**().

See Also
Ch_VaStart(), **Ch_VaEnd**(), **Ch_VaArg**(), **Ch_VaFuncArgNum**().

Ch_VaDataType

Synopsis
#include <ch.h>
ChType_t Ch_VaDataType(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the data type of argument.

Return Value
The **Ch_VaDataType** returns the data type of the argument. The returned value is one of macros defined in **ch.h** and listed on page 280.
Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaDataType**() returns the data type of the argument which can be either a regular argument or an array argument. For an array argument, the data type of its elements is returned. For an argument of pointer to function, the data type of the returned value of the function is returned. This function shall be called before the argument is obtained by function **Ch_VaArg**().

Example
See **Ch_VaStart**().

See Also
Ch_VaDataType(), **Ch_VaStart**(), **Ch_VaEnd**(), **Ch_VaArrayDim**(), **Ch_VaArrayExtent**().

Ch_VaEnd

Synopsis
#include <**ch.h**>
void Ch_VaEnd(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
For normal return from a function.

Return Value
None.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t** declared in header file **ch.h**. It represents in the C space for a variable number of arguments in the Ch space. It is created in function **Ch_VaStart**().

Description
The function **Ch_VaEnd**() facilitates a normal return from the function whose variable argument list was referred to by the expansion of **Ch_VaStart**() that initialized `a[` of type **ChVaList_t**. The **Ch_VaEnd**() may modify *ap* so that it is no longer usable (without an intervening invocation of **Ch_VaStart**(). If there is no corresponding invocation of the **Ch_VaStart**() , or if the **Ch_VaEnd**() is not invoked before the return, the behavior is undefined.

Example
See **Ch_VaStart**().

See Also
Ch_VaStart(), **Ch_VaArrayDim**(), **Ch_VaDataType**(), **Ch_VaArrayExtent**().

Ch_VaFuncArgDataType

Synopsis
#include <**ch.h**>
int Ch_VaFuncArgDataType(**ChInterp_t** *interp*, **ChVaList_t** *ap*, **int** *argnum*);

Purpose
Get the data type for an argument of the function which is pointed to by the argument of function pointer.

Return Value
This function returns a macro for data type **ChType_t** defined in header file **ch.h**. For example, **CH_INTTYPE** and **CH_DOUBLETYPE** are two macros defined for data type of int and double in Ch.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
The function **Ch_VaFuncArgDataType**() is designed for handling arguments of pointer to functions. It gets the data type for an agument of the function which is pointed to by the argument of function pointer. If the function takes a variable number of arguments, it can only return the data type for the fixed number of arguments. In this case, function **Ch_VaIsFuncVarArg**() returns 1. **Ch_VaFuncArgDataType**() shall be call before **Ch_VaArg**() is called to step *ap* to the next argument.

Example
See **Ch_VaStart**().

See Also
Ch_VaStart(), **Ch_VaEnd**(), **Ch_VaArg**(), **Ch_VaCount**(), **Ch_VaFuncArgNum**().

Ch_VaFuncArgNum

Synopsis
#include <**ch.h**>
int Ch_VaFuncArgNum(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the number of the arguments of the function which is pointed to by the argument of function pointer.

Return Value
This function returns the number of the arguments of the function. If the argument is not function type, it returns -1.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaFuncArgNum**() is designed for handling arguments of pointer to functions. It gets the number of the arguments of the function which is pointed to by the argument of function pointer. If the function takes a variable number of arguments, it only returns the fixed number of arguments. In this case, function **Ch_VaIsFuncVarArg**() returns 1. **Ch_VaFuncArgNum**() shall be call before **Ch_VaArg**() is called to step *ap* to the next argument.

Example 1

Data type of the argument	Return value from **Ch_VaFuncArgNum**()
int func(int, int)	2
int func(void)	0
int func()	0
int func(int i, ...)	1
int func(int i, int j,...)	2
int i	-1
int (*fun)()	0

Example 2
See **Ch_VaStart**() and **Ch_VaIsFuncVarArg**().

See Also
Ch_VaStart(), **Ch_VaEnd**(), **Ch_VaArg**(), **Ch_VaCount**(), **Ch_VaFuncArgDataType**(). **Ch_VaIsFuncVarArg**().

Ch_VaIsFunc

Synopsis
#include <**ch.h**>
int Ch_VaIsFunc(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Determine if the argument is a function or pointer to function.

Return Value
If the argument is a function or pointer to function, this function returns 1. Otherwise, it returns 0.
Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaIsFunc**() determines if the argument is a function or pointer to function. It shall be call before **Ch_VaArg**() is called to step *ap* to the next argument.

Example
see **Ch_VaStart**().

See Also
Ch_VaDataType(), **Ch_VaArrayType**(), **Ch_VaFuncArgNum**(), **Ch_VaIsFuncVarArg**().

Ch_VaIsFuncVarArg

Synopsis
#include <**ch.h**>
int Ch_VaIsFuncVarArg(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Determine if the argument is a function type with variable number of arguments.
Return Value
This function returns 1 if the argument is a function type with a variable number of arguments. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaIsFuncVarArg**() is designed for handling arguments of pointer to functions. If the function takes a variable number of arguments, function **Ch_VaIsFuncVarArg**() returns 1. Otherwise, it returns 0. **Ch_VaIsFuncVarArg**() shall be call before **Ch_VaArg**() is called to step *ap* to the next argument.

Example 1
In this example, functions with different number of arguments are passed to the first argument of function `chvaisfuncvararg`(). Function **Ch_VaFuncArgNum**() is used to determine the number of arguments of the function, which does not include the aguments passed in the variable number argument list. File **Makefile** for compiling and linking to create a dynamically loaded library **libch.dl** for all examples is shown in the example for function **Ch_SymbolAddrByName**() on page 293.

Listing 1 — Ch application.

```
#include <dlfcn.h>
#include <stdarg.h>

typedef int (*FUNC)();
int func0(int a, int b, int c)
{
   int a1 = a;
   int b1 = b;
   int c1 = c;
   printf("func0() called\n");
   return a+b+c;
}

int func1(int a, int b)
{
   printf("func0() called\n");
   return a+b;
}

int func2() {
```

```
   printf("func2() called\n");
   return 10;
}

int func3(void) {
   printf("func3() called\n");
   return 10;
}

int func4(int a, int b, ...) {
   int i, vacount;
   va_list ap;
   int sum;
   int val;
   float valf;
   double vald;

   printf("func4() called\n");
   sum = a+b;
   va_start(ap, b);
   vacount = va_count(ap);
   for(i=0; i<vacount; i++) {
      if(va_elementtype(ap)  == elementtype(int)) {
         val = va_arg(ap, int);
         printf("arg %d = %d\n", i+3, val);
         sum += val;
      }
      else if(va_elementtype(ap)  == elementtype(float)) {
         valf = va_arg(ap, float);
         printf("arg %d = %f\n", i+3, valf);
         sum += valf;
      }
      else if(va_elementtype(ap)  == elementtype(double)) {
         vald = va_arg(ap, double);
         printf("arg %d = %f\n", i+3, vald);
         sum += vald;
      }
   }
   return sum;
}

int chvaisfuncvararg(FUNC f, int i, int j) {
   void *handle, *fptr;
   int retval;

   handle = dlopen("libch.dl", RTLD_LAZY);
   if(handle == NULL) {
      printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
      return -1;
   }
   fptr = dlsym(handle, "chvaisfuncvararg_chdl");
   if(fptr == NULL) {
      printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
      return -1;
   }
   dlrunfun(fptr, &retval, chvaisfuncvararg, f, i, j);
   return retval;
}
```

```
int main() {
   int x = 5;
   int y = 6;
   printf("x=%d, y=%d ==> x+y=%d\n", x, y, chvaisfuncvararg((FUNC)func0, x, y));
   printf("sum=%d\n", chvaisfuncvararg((FUNC)func1, x, y));
   printf("sum=%d\n", chvaisfuncvararg((FUNC)func2, x, y));
   printf("sum=%d\n", chvaisfuncvararg((FUNC)func3, x, y));
   printf("sum=%d\n", chvaisfuncvararg(NULL, x, y));
   printf("sum=%d\n", chvaisfuncvararg((FUNC)func4, x, y));
   return 0;
}
```

Listing 2 — chdl file.

```
#include<ch.h>
#include<stdio.h>
#include<limits.h>

typedef int (*FUNC)();
static ChInterp_t interp;
static int isvar = 0;
static void *FUNC_chdl_funptr;
static int FUNC_chdl_funarg(int i, int j)
{
   int retval;
   int k=1;
   int x=5;
   float f = 10;
   double d = 20;

   if(isvar) {
     ChVaList_t arglist;

     arglist = Ch_VarArgsCreate(interp);
     Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, x);
     Ch_VarArgsAddArg(interp, &arglist, CH_FLOATTYPE, f);
     Ch_VarArgsAddArg(interp, &arglist, CH_DOUBLETYPE, d);
     Ch_CallFuncByAddr(interp, FUNC_chdl_funptr, &retval, i, j, arglist);
     Ch_VarArgsDelete(interp, arglist);
   }
   else
     Ch_CallFuncByAddr(interp, FUNC_chdl_funptr, &retval, i, j, k);

   return retval;
}

int chvaisfuncvararg(FUNC f, int i, int j)
{
     return f(i, j);
}

EXPORTCH int chvaisfuncvararg_chdl(void *varg)
{
   ChVaList_t ap;
   int i;
   int j;
   int count, num;
   int retval=-1;
```

```
   Ch_VaStart(interp, ap, varg);
   count = Ch_VaCount(interp, ap);
   printf("Total Args passed to chvaisfuncvararg_chdl() in C code = %d\n", count);
   num = Ch_VaFuncArgNum(interp, ap);
   printf("Passed pointer to func in chvaisfuncvararg_chdl has %d arg\n", num);
   if(Ch_VaIsFuncVarArg(interp, ap)) {
     isvar = 1;
     printf("Passed pointer to func in chvaisfuncvararg_chdl has varaible number arg\n");
   }
   FUNC_chdl_funptr = Ch_VaArg(interp, ap, void *);
   i = Ch_VaArg(interp, ap, int);
   j = Ch_VaArg(interp, ap, int);

   if(FUNC_chdl_funptr != NULL) {
      retval = chvaisfuncvararg(FUNC_chdl_funarg, i, j);
   }
   else {
      printf("passed fun argument to chvaisfuncvararg_chdl() is NULL\n");
   }

   Ch_VaEnd(interp, ap);
   return retval;
}
```

Output

```
Total Args passed to chvaisfuncvararg_chdl() in C code = 3
Passed pointer to func in chvaisfuncvararg_chdl has 3 arg
func0() called
x=5, y=6 ==> x+y=12
Total Args passed to chvaisfuncvararg_chdl() in C code = 3
Passed pointer to func in chvaisfuncvararg_chdl has 2 arg
func0() called
sum=11
Total Args passed to chvaisfuncvararg_chdl() in C code = 3
Passed pointer to func in chvaisfuncvararg_chdl has 0 arg
func2() called
sum=10
Total Args passed to chvaisfuncvararg_chdl() in C code = 3
Passed pointer to func in chvaisfuncvararg_chdl has 0 arg
func3() called
sum=10
Total Args passed to chvaisfuncvararg_chdl() in C code = 3
Passed pointer to func in chvaisfuncvararg_chdl has 0 arg
passed fun argument to chvaisfuncvararg_chdl() is NULL
sum=-1
Total Args passed to chvaisfuncvararg_chdl() in C code = 3
Passed pointer to func in chvaisfuncvararg_chdl has 2 arg
Passed pointer to func in chvaisfuncvararg_chdl has varaible number arg
func4() called
arg 3 = 5
arg 4 = 10.000000
arg 5 = 20.000000
sum=46
```

See Also

Ch_VaStart(), **Ch_VaEnd**(), **Ch_VaArg**(), **Ch_VaCount**(), **Ch_VaFuncArgNum**().

Ch_VaStart

Synopsis
#include <**ch.h**>
void Ch_VaStart(**ChInterp_t** *interp*, **ChVaList_t** *ap*, **void** **varg*);

Purpose
Obtain an instance of Ch interpreter *interp* and initialize *ap* for reference to arguments of a function in Ch.

Return Value
None.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t** declared in header file **ch.h**. It represents in the C space for a variable number of arguments in the Ch space.

varg The variable argument list.

Description
The **Ch_VaStart** macro initializes *interp* and *ap* for subsequent use by **Ch_VaArg**() **Ch_VaArrayDim**(), **Ch_VaArrayExtent**(), **Ch_VaDataType**(), and **Ch_VaEnd**(). **Ch_VaStart**() shall not be invoked again for the same *ap* without an intervening invocation of **Ch_VaEnd**() for the same *ap*. The **Ch_VaStart** macro shall be invoked before any call of function with prefix **Ch_Va** defined by **ch.h** header file.

Example
In this example, arguments of array, float and pointer to function have been are passed into C space. Many macros and functions defined in header file **ch.h**, including **Ch_VaCount**, **Ch_VaFuncArgDataType**, **Ch_VaFuncArgNum**, **Ch_VaArrayExtent**, **Ch_VaArrayDim** and **Ch_VaDataType** are also used. File **Makefile** for compiling and linking to create a dynamically loaded library **libch.dl** for examples is shown in the example of function **Ch_SymbolAddrByName**() on page 293.

Listing 1 — Ch application program.

```
#include<dlfcn.h>
#include<array.h>

typedef int (*func1_t)();
typedef array double func2_t(int a, int b) [2][3];
int func1(int a, int b) {
   return a+b;
}
array double func2(int a, int b) [2][3]{
   double a[2][3];
   return a;
}


int chvastart(double a[3][4], double d1, func1_t f1, func2_t f2,
```

```
              array double b[30][40], int n, double c[n], double d[n]) {
  void *handle, *fptr;
  int retval;

  handle = dlopen("libch.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return -1;
  }

  fptr = dlsym(handle, "chvastart_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return -1;
  }

  dlrunfun(fptr, &retval, NULL, a, d1, f1, f2, b, n, c, d);

  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return -1;
  }
  return retval;
}

int main() {
  double a[3][4];
  array double b[30][40];
  int n = 5;
  double c[5], d[n];

  chvastart(a, 2.0, func1, func2, b, n, c, d);
  return 0;
}
```

Listing 2 — C program to create a dynamically loaded library.

```
#include <ch.h>
#include <stdio.h>

static void *FUNC_chdl_funptr;
EXPORTCH int chvastart_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  double *a;
  ChType_t elementtype;
  int d_a, ext_a_1, ext_a_2;
  int arg_count, funarg_count, retval = 0;
  double d1;
  int n;

  Ch_VaStart(interp, ap, varg);

  /* Number of arguments passed from Ch space*/
  arg_count = Ch_VaCount(interp, ap);
  printf("Number of arguments before first calling of Ch_VaArg() is %d\n", arg_count);

  /* Data type of 1st argument */
  elementtype = Ch_VaDataType(interp, ap);
  if(elementtype == CH_DOUBLEPTRTYPE) {
```

```
  printf("Data type of the 1st argument is pointer to double\n");
}

if(Ch_VaArrayType(interp, ap) == CH_CARRAYTYPE) {
  printf("The 1st argument is C array\n");
}
else if(Ch_VaArrayType(interp, ap) == CH_CARRAYPTRTYPE) {
  printf("The 1st argument is pointer to C array\n");
}

/* Dimension of 1st argument */
d_a = Ch_VaArrayDim(interp, ap);
printf("Dimension of the 1st argument is = %d\n", d_a);

/* Number of elements in the 1st dimension of the 1st argument */
ext_a_1 = Ch_VaArrayExtent(interp, ap, 0);
printf("Extent of the 1st dimension of the 1st argument is = %d\n", ext_a_1);

/* Number of elements in the 2nd dimension of the 1st argument */
ext_a_2 = Ch_VaArrayExtent(interp, ap, 1);
printf("Extent of the 2nd dimension of the 1st argument is = %d\n", ext_a_2);

/* Return the 1st argument, and step to the next argument */
a = Ch_VaArg(interp, ap, double*);

/* Number of the rest of arguments */
arg_count = Ch_VaCount(interp, ap);
printf("\nNumber of the rest of arguments after the 1st calling of "
       "Ch_VaArg() is %d\n", arg_count);

/* Data type of 2nd argument */
elementtype = Ch_VaDataType(interp, ap);
if(elementtype == CH_DOUBLETYPE) {
  printf("Element data type of the 2nd argument is double\n");
}

/* Return the 2nd argument, and step to the next argument */
d1 = Ch_VaArg(interp, ap, double);

/* Number of the rest of arguments */
arg_count = Ch_VaCount(interp, ap);
printf("\nNumber of the rest of arguments after the 2nd calling of "
       "Ch_VaArg() is %d\n", arg_count);

/* Data type of 3nd argument */
elementtype = Ch_VaDataType(interp, ap);
if(elementtype == CH_INTTYPE) {
  printf("Return type of function of the 3nd argument of pointer to function is int\n");
}

if(Ch_VaIsFunc(interp, ap)) {
  printf("Element data type of the 3nd argument is pointer to function\n");
}
funarg_count = Ch_VaFuncArgNum(interp, ap);
printf("Number of arguments of the function pointed to by "
       "the 3nd argument is %d\n", funarg_count);
if(Ch_VaFuncArgDataType(interp, ap, 0) == CH_INTTYPE)
  printf("Arg 1 of the function pointed to by the 3nd argument is int\n");
if(Ch_VaFuncArgDataType(interp, ap, 1) == CH_INTTYPE)
```

```
    printf("Arg 2 of the function pointed to by the 3nd argument is int\n");

  /* Return the 3rd argument */
  FUNC_chdl_funptr = Ch_VaArg(interp, ap, void*);

  /* Data type of 4th argument */
  elementtype = Ch_VaDataType(interp, ap);
  if(elementtype == CH_DOUBLETYPE) {
    printf("Return type of function of the 4th argument of pointer to function is double\n");
  }

  if(Ch_VaIsFunc(interp, ap)) {
    printf("Element data type of the 4th argument is pointer to function\n");
  }

  if(Ch_VaArrayType(interp, ap) == CH_CHARRAYTYPE) {
    printf("Return type of function of the 4th argument of pointer to function "
           "is computational array\n");
  }
  funarg_count = Ch_VaFuncArgNum(interp, ap);
  printf("Number of arguments of the function pointed to by the 4th \
argument is %d\n", funarg_count);

  /* Return the 4th argument */
  FUNC_chdl_funptr = Ch_VaArg(interp, ap, void*);

  /* Data type of 5th argument */
  elementtype = Ch_VaDataType(interp, ap);
  if(elementtype == CH_DOUBLETYPE) {
    printf("Element data type of the 5th argument is double\n");
  }

  if(Ch_VaArrayType(interp, ap) == CH_CHARRAYTYPE) {
    printf("The 5th argument is computational array\n");
  }
  else if(Ch_VaArrayType(interp, ap) == CH_CHARRAYPTRTYPE) {
    printf("The 5th argument is pointer to computational array\n");
  }

  /* Dimension of 5th argument */
  d_a = Ch_VaArrayDim(interp, ap);
  printf("Dimension of the 5th argument is = %d\n", d_a);

  /* Number of elements in the 1st dimension of the 5th argument */
  ext_a_1 = Ch_VaArrayExtent(interp, ap, 0);
  printf("Extent of the 1st dimension of the 5th argument is = %d\n", ext_a_1);

  /* Number of elements in the 2nd dimension of the 5th argument */
  ext_a_2 = Ch_VaArrayExtent(interp, ap, 1);
  printf("Extent of the 2nd dimension of the 5th argument is = %d\n", ext_a_2);

  /* Return the 5th argument, and step to the next argument */
  a = Ch_VaArg(interp, ap, double*);

  /* Return the 6th argument, and step to the next argument */
  n = Ch_VaArg(interp, ap, int);
  printf("n from the 6th argument is = %d\n", n);

  /* Dimension of 7th argument */
```

```
  n = Ch_VaArrayDim(interp, ap);
  printf("Dimension of the 7th argument is = %d\n", n);

  /* Number of elements in the 1st dimension of the 7th argument */
  ext_a_1 = Ch_VaArrayExtent(interp, ap, 0);
  printf("Extent of the 1st dimension of the 7th argument is = %d\n", ext_a_1);

  /* Return the 7th argument, and step to the next argument */
  a = Ch_VaArg(interp, ap, double*);

  /* Dimension of 8th argument */
  n = Ch_VaArrayDim(interp, ap);
  printf("Dimension of the 8th argument is = %d\n", n);

  /* Number of elements in the 1st dimension of the 8th argument */
  ext_a_1 = Ch_VaArrayExtent(interp, ap, 0);
  printf("Extent of the 1st dimension of the 8th argument is = %d\n", ext_a_1);

  /* Return the 8th argument, and step to the next argument */
  a = Ch_VaArg(interp, ap, double*);

  Ch_VaEnd(interp, ap);
  return retval;
}
```

Output

```
Number of arguments before first calling of Ch_VaArg() is 8
Data type of the 1st argument is pointer to double
The 1st argument is pointer to C array
Dimension of the 1st argument is = 2
Extent of the 1st dimension of the 1st argument is = 3
Extent of the 2nd dimension of the 1st argument is = 4

Number of the rest of arguments after the 1st calling of Ch_VaArg() is 7
Element data type of the 2nd argument is double

Number of the rest of arguments after the 2nd calling of Ch_VaArg() is 6
Return type of function of the 3nd argument of pointer to function is int
Element data type of the 3nd argument is pointer to function
Number of arguments of the function pointed to by the 3nd argument is 2
Arg 1 of the function pointed to by the 3nd argument is int
Arg 2 of the function pointed to by the 3nd argument is int
Return type of function of the 4th argument of pointer to function is double
Element data type of the 4th argument is pointer to function
Return type of function of the 4th argument of pointer to function is computational array
Number of arguments of the function pointed to by the 4th argument is 2
Element data type of the 5th argument is double
The 5th argument is pointer to computational array
Dimension of the 5th argument is = 2
Extent of the 1st dimension of the 5th argument is = 30
Extent of the 2nd dimension of the 5th argument is = 40
n from the 6th argument is = 5
Dimension of the 7th argument is = 1
Extent of the 1st dimension of the 7th argument is = 5
Dimension of the 8th argument is = 1
Extent of the 1st dimension of the 8th argument is = 5
```

See Also

Ch_VaArg(). **Ch_VaEnd**(), **Ch_VaCount**(), **Ch_VaFuncArgDataType**(), **Ch_VaFuncArgNum**(), **Ch_VaArrayExtent**(), **Ch_VaArrayDim**() **Ch_VaDataType**().

Ch_VaUserDefinedAddr

Synopsis
#include <**ch.h**>
void *Ch_VaUserDefinedAddr(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the address of the argument of the user defined data type in terms of struct, class, or union and its pointer type.

Return Value
This function returns the address to the object or NULL if the argument is not a user defined data type.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaUserDefinedAddr**() returns the address of the argument of struct/class/union or its pointer type in the argument list. **Ch_VaUserDefinedAddr**() shall be call before **Ch_VaArg**() is called to step *ap* to the next one. This function and two relevant functions **Ch_VaUserDefinedName**() and **Ch_VaUserDefinedSize**() can be used to retrieve complete information about an argument of user defined type from the argument list. They can be useful for handling polymorphism of function overloading with arguments of user defined types of class or struct.

Example
See **Ch_VaUserDefinedName**().

See Also
Ch_VaArg(), **Ch_VaUserDefinedName**(), **Ch_VaUserDefinedSize**().

Ch_VaUserDefinedName

Synopsis
#include <**ch.h**>
void *Ch_VaUserDefinedName(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the tag name of the argument of the user defined data type in terms of struct, class, or union and its pointer type.

Return Value
This function returns the address to the tag name or NULL if the argument is not a user defined data type.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaUserDefinedName**() returns the address for the tag name of the argument of struct/class/union or its pointer type in the argument list. **Ch_VaUserDefinedName**() shall be call before **Ch_VaArg**() is called to step *ap* to the next one. This function and two relevant functions **Ch_VaUserDefinedName**() and **Ch_VaUserDefinedSize**() can be used to retrieve complete information about an argument of user defined type from the argument list. They can be useful for handling polymorphism of function overloading with arguments of user defined types of class or struct.

Example
In this example, arguments of structure, pointer to structure, pointer to class, and union type are passed to the function `chvauserdefinedname`(). functions **Ch_VaUserDefinedName**(), **Ch_VaUserDefinedName**(), are used to handle these arguments. File **Makefile** for compiling and linking to create a dynamically loaded library **libch.dl** for all examples is shown in the example for function **Ch_SymbolAddrByName**() on page 293.

Listing 1 — Ch application.

```
#include <dlfcn.h>
#include <stdarg.h>

typedef struct tags {
    int i;
    double d;
} tags_t;
typedef class tagc {
  public:
    int i;
    double d;
} tagc_t;
typedef union tagu {
    int i;
    double d;
```

```
} tagu_t;

void chuserdefinedname (struct tags s, struct tags *sp,
                        tagc_t *cp, union tagu u) {
    void *fptr, *handle;

    handle = dlopen("libchcpp.dl", RTLD_LAZY);
    if(handle == NULL) {
      printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
      return;
    }
    fptr = dlsym(handle, "chuserdefinedname_chdl");
    if(fptr == NULL)  {
       printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
       return;
    }
    dlrunfun(fptr, NULL, NULL, s, sp, cp, u);
    if(dlclose(handle)!=0) {
       printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
       return;
    }
    va_end(ap);
    return;
}

int main() {
   struct tags s = {10, 20.0};
   struct tagc c = {10, 20.0};
   struct tagu u = {10, 20.0};

   chuserdefinedname(s, &s, &c, u);
   return 0;
}
```

Listing 2 — chdl file.

```
/*  Testing functions Ch_VaUserDefined*() */
#include <ch.h>
#include <stdarg.h>
#include <stdio.h>

typedef struct tags {
    int i;
    double d;
} tags_t;
typedef class tagc {
  public:
    int i;
    double d;
} tagc_t;
typedef union tagu {
    int i;
    double d;
} tagu_t;
EXPORTCH void chuserdefinedname_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int i, vacount;
    void *addr;
    size_t  size;
```

```
    char *name;
    tags_t s, *sp;
    tagc_t *cp;
    tagu_t u;

    Ch_VaStart(interp, ap, varg);
    vacount = Ch_VaCount(interp, ap);
    printf("vacount = %d\n", vacount);
    for(i = 0; i < vacount; i++) {
      addr = Ch_VaUserDefinedAddr(interp, ap);
      name = Ch_VaUserDefinedName(interp, ap);
      size = Ch_VaUserDefinedSize(interp, ap);
      if(Ch_VaDataType(interp, ap) == CH_STRUCTTYPE) {
         printf("Structure:\n");
         printf("address is %p, the size is %d, tag name is %s\n", addr, size, name);
         s = Ch_VaArg(interp, ap, struct tags);
         printf("s.i = %d, s.d = %f\n", s.i, s.d);
      }
      else if(Ch_VaDataType(interp, ap) == CH_STRUCTPTRTYPE) {
         printf("Pointer to structure:\n");
         printf("address is %p, the size is %d, tag name is %s\n", addr, size, name);
         sp = Ch_VaArg(interp, ap, struct tags*);
         printf("sp->i = %d, sp->d = %f\n", sp->i, sp->d);
      }
      else if(Ch_VaDataType(interp, ap) == CH_CLASSPTRTYPE) {
         printf("Class:\n");
         printf("address is %p, the size is %d, tag name is %s\n", addr, size, name);
         cp = Ch_VaArg(interp, ap, tagc_t*);
         printf("cp->i = %d, cp->d = %f\n", cp->i, cp->d);
      }
      else if(Ch_VaDataType(interp, ap) == CH_UNIONTYPE) {
         printf("Union:\n");
         printf("address is %p, the size is %d, tag name is %s\n", addr, size, name);
         u = Ch_VaArg(interp, ap, tagu_t);
         printf("u.i = %d, u.d = %f\n", u.i, u.d);
      }
    }
    Ch_VaEnd(interp, ap);
    return;
}
```

Output

```
vacount = 4
Structure:
address is 2bff28, the size is 16, tag name is tags
s.i = 10, s.d = 20.000000
Pointer to structure:
address is 2bfef8, the size is 16, tag name is tags
sp->i = 10, sp->d = 20.000000
Class:
address is 2bff10, the size is 16, tag name is tagc
cp->i = 10, cp->d = 20.000000
Union:
address is 2b71e0, the size is 8, tag name is tagu
u.i = 1077149696, u.d = 20.000000
```

See Also

Ch_VaArg(), **Ch_VaUserDefinedAddr**(), **Ch_VaUserDefinedSize**().

Ch_VaUserDefinedSize

Synopsis
#include <**ch.h**>
int Ch_VaUserDefinedSize(**ChInterp_t** *interp*, **ChVaList_t** *ap*);

Purpose
Obtain the size of the user defined data type in terms of struct, class, or union and its pointer type for the argument.

Return Value
This function returns the size of the object or -1 if the argument is not a user defined data type.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t**.

Description
The function **Ch_VaUserDefinedSize**() returns the size of the argument of struct/class/union or its pointer type in the argument list. **Ch_VaUserDefinedSize**() shall be call before **Ch_VaArg**() is called to step *ap* to the next one. This function and two relevant functions **Ch_VaUserDefinedName**() and **Ch_VaUserDefinedSize**() can be used to retrieve complete information about an argument of user defined type from the argument list. They can be useful for handling polymorphism of function overloading with arguments of user defined types of class or struct.

Example
See **Ch_VaUserDefinedName**().

See Also
Ch_VaArg(), **Ch_VaUserDefinedAddr**(), **Ch_VaUserDefinedName**().

Ch_VaVarArgsCreate

Synopsis
#include <**ch.h**>
void * Ch_VaVarArgsCreate(**ChInterp_t** *interp*, **ChVaList_t** *ap*, **void ****_memhandle_);

Purpose
Create a C style variable argument list in _chdl.c interface function according to the Ch style variable length argument list for calling C functions of variable length agument list from Ch space.

Return Value
This function returns a C variable argument list, effectively the same as the argument of type **va_list** defined in header file **stdarg.h**.

Parameters
interp A Ch interpreter.

ap An object having type **ChVaList_t** declared in header file **ch.h**. It represents in the C space for a variable number of arguments in the Ch space.

memhandle The address of the pointer to an object for memory management with type **void ****.

Description
Since a variable argument list generated by a Ch function is different from the one generated by a C function, the arguments in a Ch style argument list can be handled only by functions defined in header file **ch.h**, such as **Ch_VaArg**(), and a Ch style argument list can't be passed to a C function directly. According to a Ch style variable argument list, the function **Ch_VaVarArgsCreate**() can create a C style variable argument list, which can be passed to a C function. To make sure that all arguments in the Ch style argument list will appear in the C style one, **Ch_VaVarArgsCreate**() shall be call before **Ch_VaArg**() is called.

The users need to declare a pointer to void in the calling function and then pass its address as the second argument of this function. This pointer will be used for deleting memory in function **Ch_VaVarArgsDelete**().

Example
In this example, function `func`() takes variable arguments, and function `vfunc`() is the corresponding function which takes the argument with type of va_list. Typically, for each function like `func`(), which takes an variable argument list, there will be a corresponding function like `vfunc`(), which takes an argument with type va_list, to handle the variable argument list of the former. File **Makefile** for compiling and linking to create the dynamically loaded library **libch.dl** for all examples in Appendix A is shown in Example 1 for function **Ch_SymbolAddrByName**() on page 293.

Listing 1 — Ch application (vararg.ch).

```
#include <stdio.h>

void func(int n, ...);
void vfunc(int n, va_list ap);

int main() {
    char    c1 = 'c';
```

```
    int     i1 = 10;
    double  d1 = 65536;

    func(0, c1, i1, d1);
    return 0;
}

void func(int n, ...) {
    va_list ap;

    va_start(ap, n); /* create a Ch va_list */
    vfunc(n, ap);
    va_end(ap);
}

void vfunc(int n, va_list ap) {
    void *fptr, *handle;

    handle = dlopen("libch.dl", RTLD_LAZY);
    if(handle == NULL) {
      printf("Error: dlopen(): %s\n", __func__, dlerror());
     return;
    }

    fptr = dlsym(handle, "vfunc_chdl");
    if(fptr == NULL)  {
       printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
       return;
    }
    dlrunfun(fptr, NULL, NULL, n, ap);

    if(dlclose(handle)!=0) {
       printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
       return;
    }
}
```

Listing 2 — chdl file (vararg.c).

```
/* example for testing functions Ch_VaVarargs*() */
#include <ch.h>
#include <math.h>
#include <stdarg.h>
#include <stdio.h>

void func(int n, ...);
void vfunc(int n, va_list ap_c);

EXPORTCH void vfunc_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    void * ap_c; /* equivalent to va_list ap_c */
    ChVaList_t ap_ch;
    int ii;
    void *memhandle;

    Ch_VaStart (interp, ap, varg);
    ii = Ch_VaArg (interp, ap, int);
    ap_ch = Ch_VaArg (interp, ap, ChVaList_t);
```

```
    ap_c = Ch_VaVarArgsCreate(interp, ap_ch, &memhandle);
    vfunc(ii, ap_c);
    Ch_VaVarArgsDelete(interp, memhandle);

    Ch_VaEnd(interp, ap);
    return;
}

void func(int n, ...) {
    va_list ap;

    va_start(ap, n);
    vfunc(n, ap);
    va_end(ap);
}

void vfunc(int n, va_list ap_c) {
    char c1;
    int i1;
    double d1;

    printf("The first argument is %d\n", n);

    c1 = va_arg(ap_c, char);
    printf("\nc1 = %c\n", c1);

    i1 = va_arg(ap_c, int);
    printf("i1 = %d\n", i1);

    d1 = va_arg(ap_c, double);
    printf("d1 = %f\n", d1);

    return;
}
```

Output

```
The first argument is 0

c1 = c
i1 = 10
d1 = 65536.000000
```

See Also
Ch_VaVarArgsDelete(), **Ch_VaStart**(), **Ch_VaEnd**(), **Ch_VaArg**().

Ch_VaVarArgsDelete

Synopsis
#include <**ch.h**>
void Ch_VaVarArgsDelete(**ChInterp_t** *interp*, **void ***memhandle*);

Purpose
Delete memory allocated in function **Ch_VaVarArgsDelete**().

Return Value
No return value.

Parameters
interp A Ch interpreter.

memhandle The pointer to an object for memory management with type **void ***.

Description
Since a variable argument list generated by a Ch function is different from the one generated by a C function, the arguments in a Ch style argument list can be handled only by functions defined in header file **ch.h**, such as **Ch_VaArg**(), and a Ch style argument list can't be passed to a C function directly. The function **Ch_VaVarArgsCreate**() creates a C style variable argument list, which can be passed to a C function, according to the Ch style variable argument list.

The users need to declare a pointer to void in the calling function and then pass its address as the second argument of function **Ch_VaVarArgsCreate**(). This pointer will be used for deleting memory in **Ch_VaVarArgsDelete**().

Example
See **Ch_VaVarArgsCreate**().

See Also
Ch_VaVarArgsCreate(), **Ch_VaStart**(), **Ch_VaEnd**(), **Ch_VaArg**().

Ch_VarArgsAddArg

Synopsis
#include <**ch.h**>
int Ch_VarArgsAddArg(**ChInterp_t** *interp*, **ChVaList_t** **arglistp*, **ChType_t** *atype*, ...);
int Ch_VarArgsAddArg(**ChInterp_t** *interp*, **ChVaList_t** **arglistp*, **ChType_t** *atype*, type value);
int Ch_VarArgsAddArg(**ChInterp_t** *interp*, **ChVaList_t** **arglistp*, **ChType_t** *atype*, const char *tagname, type value);
int Ch_VarArgsAddArg(**ChInterp_t** *interp*, **ChVaList_t** **arglistp*, **ChType_t** *atype*, **ChType_t** *etype*, type value,
int dim, int extent1, ...);

Syntax
Ch_VarArgsAddArg(*interp*, *arglistp*, *atype*, *value*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_INTTYPE*, *value*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_PROCPTRTYPE*, *NULL*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_PROCPTRTYPE*, *Ch_SymbolAddrByName(NULL, "fp")*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_STRUCTTYPE*, *tagname*, *structvalue*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_CLASSTYPE*, *tagname*, *classtvalue*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_UNIONTYPE*, *tagname*, *unionvalue*);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_CARRAYTYPE*, *etype*, *dim*, *extent1*, ...);
Ch_VarArgsAddArg(*interp*, *arglistp*, *CH_CHARRAYTYPE*, *etype*, *dim*, *extent1*, ...);

Purpose
Add an argument into the existing Ch style variable argument list.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

arglistp The address of Ch variable argument list.

atype The type of the argument to be added. The values for data types **ChType_t** are defined in header file **ch.h** on page 280.

... The argument to be added.

Description
This function is typically used in the case of calling a Ch function which takes a variable argument list. The Ch function will be called by one of functions **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), and **Ch_CallFuncByNameVar**(). Since a variable argument list for a Ch function is different from the one for C function, arguments can not be passed to a Ch style variable argument list through **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), or **Ch_CallFuncByNameVar**() directly. This function will add arguments, created in C space, into an existing Ch style variable argument list.

The function **Ch_VarArgsCreate**() will create a Ch style variable argument list with no argument inside. Ch style variable argument list needs to be deleted by **Ch_VarArgsDelete**() finally.

Example 1

```
    /* Function prototype in Ch space:
       int func(int n, int m,  double f, int *p,
           int a[2][3], int b[n][m], array int c[2][3], array int d[n][m],
           struct tag s,  struct tag *sp);
       int func2(int r, ... /* int n, int m,  double f, int *p,
           int a[2][3], int b[n][m], array int c[2][3], array int d[n][m],
            struct tag s,  struct tag *sp */);
       array int func3()[2][3];
    */

    int n = 2, m = 3, dim =2;
    double f = 10.0;
    int a[2][3] = {1, 2, 3,
                   4, 5, 6};
    int b[2][3] = {1, 2, 3,
                   4, 5, 6};
    int c[2][3] = {1, 2, 3,
                   4, 5, 6};
    int d[2][3] = {1, 2, 3,
                   4, 5, 6};
    struct tag {
       int i;
       double f;
    } s = {10,20}, *sp=&s;
    int r = 10;

    arglist = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, 2);
    Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, m);
    Ch_VarArgsAddArg(interp, &arglist, CH_DOUBLETYPE, f);
    Ch_VarArgsAddArg(interp, &arglist, CH_DOUBLEPTRTYPE, p);
    Ch_VarArgsAddArg(interp, &arglist, CH_CARRAYTYPE, CH_INTTYPE, a, dim, n, m);
    Ch_VarArgsAddArg(interp, &arglist, CH_CARRAYTYPE, CH_INTTYPE, b, dim, n, m);
    Ch_VarArgsAddArg(interp, &arglist, CH_CARRAYTYPE, CH_INTTYPE, c, dim, n, m);
    Ch_VarArgsAddArg(interp, &arglist, CH_CARRAYTYPE, CH_INTTYPE, d, dim, n, m);
/* or Ch_VarArgsAddArg(interp, &arglist, CH_CHARRAYTYPE, CH_INTTYPE, c, dim, n, m);
      Ch_VarArgsAddArg(interp, &arglist, CH_CHARRAYTYPE, CH_INTTYPE, d, dim, n, m); */
    Ch_VarArgsAddArg(interp, &arglist, CH_STRUCTTYPE, "tag", &s);
    Ch_VarArgsAddArg(interp, &arglist, CH_STRUCTPTRTYPE, sp);
    Ch_CallFuncByNameVar(interp, "func", &retval, arglist);
    Ch_CallFuncByName(interp, "func2", &retval, r, arglist);
    Ch_VarArgsDelete(interp, arglist);
    Ch_CallFuncByName(interp, "func3", a);// func3() returns computational array
```

Example 2
Program 6.17.

See Also
Ch_VarArgsAddArgExpr(), **Ch_VarArgsCreate**(), **Ch_VarArgsDelete**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**().

Ch_VarArgsAddArgExpr

Synopsis
#include <**ch.h**>
int Ch_VarArgsAddArgExpr(**ChInterp_t** *interp*, **ChVaList_t** **arglistp*, **char** *expr*);

Purpose
Add an argument in terms of an expression in the Ch space into the existing Ch style variable argument list.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

arglistp The address of Ch variable argument list.

expr The argument in the expression in the Ch space to be added.

Description
This function is typically used in the case of calling a Ch function. The Ch function will be called by one of functions **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), and **Ch_CallFuncByNameVar**(). This function will add arguments, created in the Ch space, into an existing Ch style variable argument list.

The function **Ch_VarArgsCreate**() will create a Ch style variable argument list with no argument inside. Ch style variable argument list needs to be deleted by **Ch_VarArgsDelete**() finally.

Example 1

```
/* Code in Ch space
   int func(int n, int m,  double f, int *p,
       int a[2][3], int b[n][m],
       array int c[2][3], array int d[n][m],
       struct tag s,  struct tag *sp);
   int n = 2, m = 3, dim =2;
   double f = 10.0;
   int a[2][3] = {1, 2, 3,
                  4, 5, 6};
   int b[2][3] = {1, 2, 3,
                  4, 5, 6};
   array int c[2][3] = {1, 2, 3,
                        4, 5, 6};
   array int d[2][3] = {1, 2, 3,
                        4, 5, 6};
   struct tag {
      int i;
      double f;
   } s = {10,20}, *sp=&s;
 */
```

```
arglist= Ch_VarArgsCreate(interp);
Ch_VarArgsAddArgExpr(interp, &arglist, "2");
Ch_VarArgsAddArgExpr(interp, &arglist, "m");
Ch_VarArgsAddArgExpr(interp, &arglist, "f");
Ch_VarArgsAddArgExpr(interp, &arglist, "p");
Ch_VarArgsAddArgExpr(interp, &arglist, "a");
Ch_VarArgsAddArgExpr(interp, &arglist, "b");
Ch_VarArgsAddArgExpr(interp, &arglist, "c");
Ch_VarArgsAddArgExpr(interp, &arglist, "d");
Ch_VarArgsAddArgExpr(interp, &arglist, "s");
Ch_VarArgsAddArgExpr(interp, &arglist, "sp");
Ch_CallFuncByNameVar(interp, "func", &retval, arglist);
Ch_VarArgsDelete(interp, arglist);
```

Example 2
See *Embedded Ch SDK User's Guide*.

See Also
Ch_VarArgsAddArg(), **Ch_VarArgsCreate**(), **Ch_VarArgsDelete**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**().

Ch_VarArgsCreate

Synopsis
#include <**ch.h**>
ChVaList_t Ch_VarArgsCreate(**ChInterp_t** *interp*);

Purpose
Create a Ch style variable argument list initially for calling Ch functions of variable agument list from C space.

Return Value
This function returns a Ch variable argument list with no argument inside.

Parameters

interp A Ch interpreter.

Description
This function is typically used in the case of calling a Ch function which takes a variable argument list. The Ch function will be called by one of functions **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), and **Ch_CallFuncByNameVar**(). Since a variable argument list for a Ch function is different from the one for C function, arguments can not be passed to a Ch style variable argument list through **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), or **Ch_CallFuncByNameVar**() directly. This function will create initially a Ch style variable argument list in C space.

After the Ch style variable argument list is created, the user can add arguments into it by function **Ch_VarArgsAddArg**() or **Ch_VarArgsAddArgExpr**(). Finally, this Ch style variable argument list needs to be deleted by **Ch_VarArgsDelete**().

Example
Program 6.17.

See Also
Ch_VarArgsAddArg(), **Ch_VarArgsAddArgExpr**(), **Ch_VarArgsDelete**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**().

Ch_VarArgsDelete

Synopsis
#include <**ch.h**>
void **Ch_VarArgsDelete**(**ChInterp_t** *interp*, **ChVaList_t** *arglist*);

Purpose
Delete a Ch style variable argument list.

Return Value
None.

Parameters
interp A Ch interpreter.

arglist The Ch variable argument list to be deleted.

Description
This function is typically used in the case of calling a Ch function which takes a variable argument list. The Ch function will be called by one of functions **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), and **Ch_CallFuncByNameVar**(). Since a variable argument list for a Ch function is different from the one for C function, arguments can not be passed to a Ch style variable argument list through **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), or **Ch_CallFuncByNameVar**() directly. This function will delete an existing Ch style variable argument list created by **Ch_VarArgsCreate**().

The function **Ch_VarArgsCreate**() will create a Ch style variable argument list with no argument inside. Then functions **Ch_VarArgsAddArg**() and **Ch_VarArgsAddArgExpr**() can add arguments into that Ch style variable argument afterwards.

Example
Program 6.17.

See Also
Ch_VarArgsAddArg(), **Ch_VarArgsAddArgExpr**(), **Ch_VarArgsDelete**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**().

Appendix B

Interface Functions with Dynamically Loaded Library — <dlfcn.h>

The header **dlfcn.h** defines macros and functions for interface of runtime functions in dynamically loaded libraries. Using these functions, an application will be able to extend its address space during its execution by binding to dynamically loaded object.

Functions

The following functions are defined by the header file **dlfcn.h**.

Function	Description
dlopen	Gain access to an executable object file.
dlsym	Get the address of a symbol in a dynamically loaded library.
dlclose	Free address space when application has finished with the shared object.
dlerror	Get diagnostic information of using these runtime interface routines.
dlrunfun	Call the function defined by the symbol in dynamically loaded library.

Macros

The following macros are defined by the header file **dlfcn.h**.

Macro	Description
RTLD_LAZY	Function call binding during process execution.
RTLD_NOW	Immediate function call binding when dynamically loaded object is added.

Portability

Function **dlerror**() may not return correct error message in Microsoft Windows.

Difference between C and Ch

Built-in function **dlrunfun()** is available only in Ch.

B.1 dlclose

Synopsis
#include <**dlfcn.h**>
int dlclose(**void** **handle*);

Purpose
Close a dynamically loaded library.

Return Value
This function returns 0 on success or non-zero value on failure. More detailed diagnostic information will be available through **dlerror**().

Parameters
handle Returned by routine **dlopen**() for symbol search.

Description
dlclose() disassociates a dynamically loaded object previously opened by **dlopen**() from the current process. Once an object has been closed using **dlclose**(), its symbols are no longer available to **dlsym**(). All objects loaded automatically as a result of invoking **dlopen**() on the referenced object are also closed.

Example
See **dlopen**().

See Also
dlsym(), **dlerror**(), **dlopen**(), **dlrunfun**().

B.2 dlerror

Synopsis
#include <**dlfcn.h**>
char ***dlerror**(**void**);

Purpose
Obtain diagnostic information during dynamic linking process.

Return Value
The function returns a null-terminated character string describing the error or NULL if there is no error.

Parameters
None.

Description
The function **dlerror**() returns a null-terminated character string with no trailing newline that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of **dlerror**(), **dlerror**() returns NULL. Thus, invoking **dlerror**() a second time immediately following a prior invocation will result in NULL being returned.

Example
See **dlopen**().

See Also
dlsym(), **dlclose**(), **dlopen**(), **dlrunfun**().

B.3 dlopen

Synopsis
#include <dlfcn.h>
void *__**dlopen**__(**const char** **filename*, **int** *mode*);

Purpose
Gain access to an executable object file.

Return Value
This routine returns a *handle* which can be used to locate symbol by application using **dlsym**() on success or NULL on failure.

Parameters
filename The name of the file which contains the dynamically loadable library. In Windows, Linux, Solaris, and Mac, if the value of *filename* is NULL, the function provides a hanlde for the objects in the original program.

mode The mode to access the executable object file. It can be set to one of following macros:

RTLD_LAZY Function call binding during process execution. Symbol in a symbol scope.

RTLD_NOW Immediate function call binding when shared object is added.

Description
The function **dlopen**() makes an executable object file available to a running process. The function returns to the process a handle which the process may use on subsequent calls to **dlsym**(), **dlrunfun**() and **dlclose**().

Example
In program **hypothetical.ch** shown in Listing 1, the function **hypothetical()** is called through the function file **hypothetical.chf** shown in Listing 2. The shared object is dynamically linked to application program through the file **hypothetical.c** shown in Listing 3. The file **hypothetical.c** is compiled and linked to create a dynamically loadable library **libhypothetical.dl**. The makefile for compiling and linking is shown in Listing 4. The function **hypothetical()** is defined as

$$\text{hypothetical}(x, y) = \sqrt{(x^2 + y^2)}$$

In this example, the binary object of the C function **hypot**() in program **hypothetical.c** is located in library `libc.a` and is automatically linked by command **dllink**. If C functions are located in some other binary library, say `mytest.a`, this library should be added as one of the arguments for command **dllink** as shown in below.

```
ch dllink libhypothetical.dl hypothetical.o mytest.a
```

Listing 1 — A Ch application program

```
/* File name: hypothetical.ch */

#include <stdlib.h>
#include <math.h>

int main() {
    double x,y;
    double z;

    x = 3;
    y = 4;
    z = hypothetical(x,y);
    printf("The value of sqrt(%1.0f*%1.0f+%1.0f*%1.0f) = %1.0f\n",x,x,y,y,z);
}
```

Listing 2 — A Ch function file that interfaces binary objects

```
/* File name hypothetical.chf */

#include<math.h>
#include<dlfcn.h>
#include<stdio.h>

double hypothetical(double x, double y) {
    void *fptr, *handle;
    double retval;

    if(isnan(y)) /* some native implementation give hypothetical(+/-Inf, NaN) == Inf */
      return NaN;
    handle = dlopen("libhypothetical.dl", RTLD_LAZY);
    if(handle == NULL) {
      fprintf(stderr, "Error: dlopen(): %s\n", dlerror());
      fprintf(stderr, "       cannot get handle in hypothetical.chf\n");
      return NaN;
    }
    fptr = dlsym(handle, "hypothetical_chdl");
    if(fptr == NULL)  {
       fprintf(stderr, "Error: %s(): dlsym(): %s\n", __func__, dlerror());
       return NaN;
    }
    dlrunfun(fptr, &retval, hypothetical, x, y);
    if(dlclose(handle)!=0) {
       fprintf(stderr, "Error: %s(): dlclose(): %s\n", __func__, dlerror());
       return NaN;
    }
    return retval;
}
```

Listing 3 — A C program to be compiled to create dynamically loaded library **libhypothetical.dl**.

```
/* file name: hypothetical.c */
#include<ch.h>
#include<math.h>

/* EXPORTCH must be used */
EXPORTCH double hypothetical_chdl(void *varg) {
    ChInterp_t interp;
    va_list ap;
```

```
    double x;
    double y;
    double retval;

    Ch_VaStart (interp, ap,varg);
    x = Ch_VaArg (interp, ap, double);
    y = Ch_VaArg (interp, ap, double);
    retval = hypot(x, y); /* hypot() is a standard function in libc.a */
    Ch_VaEnd(interp, ap);
    return retval;
}
```

Listing 4 — Makefile

```
# build dynamically loaded  lib libhypothetical.dl

target: Makefile libhypothetical.dl

libhypothetical.dl: Makefile hypothetical.o
        ch dllink libhypothetical.dl hypothetical.o
hypothetical.o:  hypothetical.c
        ch dlcomp libhypothetical.dl hypothetical.c
clean:
        rm -f *.o *.dl ../output/*
```

Output

```
The value of sqrt(3*3+4*4) = 5
```

See Also

dlclose(), **dlerror**(), **dlsym**(), **dlrunfun**(), **dlcomp**, **dllink**.

B.4 dlrunfun

Synopsis
#include <**dlfcn.h**>
int dlrunfun(**void** **fprt*, **void** **retval*, **type** (**func*)(), [**void** **m_class*], ...);

Purpose
Run a function in a dynamically loaded library.

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

Parameters
fptr The address of symbol in a dynamically loaded library returned from function **dlsym**().

retval A pointer to the return value of called function. NULL is used if the called function has return type of **void**.

func The name of the function to be called or NULL. Ch will perform the prototype checking if the function name is used.

m_class A pointer to an instant of class. It is used only when a member function of a class is called.

... The arguments of the called function. If third parameter `func` for **dlrunfun** is NULL, the arguments will not be checked against the function in definition. In other words, the number of arguments can be different from that in function definition.

Description
dlrunfun() runs a function in a dynamically loaed object through the address of the symbol returned by function **dlsym**(). The number of variable arguments depends on that of called function *func*.

Example
A regular function in a dynamically loaded object is called. See example for **dlopen**().

See Also
dlsym(), **dlerror**(), **dlopen**(), **dlclose**(), **dlcomp**, **dllink**.

B.5 dlsym

Synopsis
#include <**dlfcn.h**>
void *__dlsym__(**void** **handle*, **const char** **symbol*);

Purpose
Obtain the address of a symbol in a dynamically loaded library.

Return Value
This routine returns the address of symbol in a dynamically loaded library if *handle* returned by the function **dlopen**() is valid, otherwise returns NULL.

Parameters
handle The parameter returned by routine **dlopen**() for symbol search. In HP-UX, if handle is `NULL`, the symbol will be searched in the original program.

symbol The symbol to be searched in the dynamically loaded library. For an exported symbol, it shall be declared with type qualifier **EXPORTCH**.

Description
The function **dlsym**() locates symbol within the dynamically loaded library. The application can then reference the data or call the function defined by the symbol using function **dlrunfun**().

Example
See **dlopen**().

See Also
dlclose(), **dlerror**(), **dlopen**(), **dlrunfun**().

Appendix C

Commands

All native commands, such as **cp** in Unix and **dir** in Windows, are valid in the Ch language environment. The Ch specific commands are described in this documentation.

Commands

Command	Description
c2chf	Generate a C file for dynamical loaded library and Ch function files from a C header file
dlcomp	Compile a C/C++ file to generate an object file for dynamical loaded library
dllink	Link object files to create a dynamical loaded library
pkginstall.ch	Install a Ch package

C.1 c2chf

Synopsis
c2chf *header_file* [**-h** *handle*] [**-v**] [**-i c** *head1.h*] [**-i c** *head2.h*] ... [**-i chf** *header1.h*] [**-i chf** *header2.h*]...
[−**r** *return_value_file*] [**-o c** *directory_for_c_output_file*] [**-o chf** *directory_for_chf_output_files*]
[−**s** *cstructdefname*] [**-s** *structdefname2*]
[−**l** *func vfunc*] [**-l** *func2 vfunc2*]

Purpose
Generate a C file for dynamically loaded library and Ch function files from a C header file.

Description
This program will generate .c and .chf files from a C header file. The .chf files are function files to be used in the Ch language environment. They are used to create dynamic loaded libraries. **c2chf** will clean up the specified C header file by removing all comments. The program then takes the ”clean” version of the C header file and reads in one function prototype at a time. The program then proceeds to build up a single .c file for each individual C header file, and separate .chf files for each function prototype. The user can use the command line flags to specify which header files to include in the .c file or the .chf files. The program allows for multiple header files to be included. If the error return values are known for the function prototypes, the user can create a two column list consisting of a function name and its error return value. The program will then proceed to read from this file and store each pair of function name and return value into a struct. During the building process of the .chf files, the program will search for the current function name in the struct containing the error return values. If a match is made, the program will use the user specified error return value in place of the default values. This process will continue until the end of the C header file is reached.

Options
-h: Allows user to specify handle for .chf files. If no handle is specified, _Ch_*header_file*_handle will be used as the default handle.

-i Allow user to include header files in the .c and .chf files. **"-i c** *head1.h*" will place the specified header file *head1.h* in the .c file. **"-i chf** `header1.h`" will place the specified header file *header1.h* in all the .chf files. Option **-i** can be used multipel times to include multiple header files as shown in the above Synopsis for header files `head1.h` and `head2.h` for .c file and `header1.h` and `header2.h` for .chf files. If the user does not use the ”**-i**” flag, then the default include files will be ”ch.h” and ”header_file.h”.

-r Allow the user to include a file which contains a list of function names and their error return values. (For example: sin NaN) The default error values that will be used if this flag is not selected are NULL for functions returning a pointer type, and -1 for all other functions with return values. There is no return value for functions of type void.

-o Allow the user to specify a directory to place output files. Option **-o c** specifies the directory *directory_for_c_output_file* for a .c file. Option **-o chf** specifies the directory *directory_for_chf_output_files* for .chf function files. By default, both .c file and .chf files are placed in the directory *./header_file*.

-v Print out the processed functoins and statistics.

-s Allow the user to specify a typedefed structure name as return type for a function returning a structure. This option can be used multiple times for different typedefed structure names.

-l To handle functions with variable number of arguments. The function `func()` of a variable number of arguments and its corresponding function `vfunc()` with a fixed number of arguments are specified following option **-l**. This option can be used multiple times for multiple functions with variable number of arguments.

Notes

1. The header may need to be cleaned up first. Comments can remain but all preprocessor directives, such as #define and typedefs, must be removed manually or using function **processhfile()**.

2. If a directory already exists with the same name as the current header file or user specified directory, the directory will be used. The `.chf` files and `_chdl.c` file will be overwritten if they exist.

3. The command **c2chf** can handle functions with arguments of pointer to structure or or functions that return a pointer to structure. It can handle functions with arguments of structure type.

4. Although .chf and chdl functions for a function returning structure type are different from those for refular functions as described in section 5.6, the command **c2chf** also can handle a function that returns structure if the function return type is represented in the form of `struct tagname`. If the function return type is represented using a typedefed structure name such as `structdefname`, the option `-s structdefname` needs to be used.

5. Similar to handling functions returning structure type, the command **c2chf** can also handle function with variable number arguments such as `int func(int i, ...)`, if there is a corresponding function of `int vfunc(int i, va_list ap)` as described in 5.5. Otherwise, it needs to be handled manually.

6. The command **c2chf** cannot handle functions with argument type of pointer to function or functions that return pointer to function. These specicial functions need to be handled manually and they need to be removed from a header file using function or using function **removeFuncProto()**, before the header file can be processed by the command **c2chf**.

Example 1

How to include a header file in a .c and .chf file.

c2chf *example.h* **-i c** *header1.h* **-i chf** *header2.h*

Example 2

Multiple **-i** statements are acceptable. The following example shows how to include a user specified handle in the .chf files and specify directories for the output files:

c2chf *example.h* **-h** *_Chexample_handle* **-o c** *temp_dir_c* **-o chf** *temp_dir_chf*

Example 3

The commands below

```
processhfile("extern", 0, ";", "math.h", "chfcreate/math.h", NULL);
c2chf chfcreate/math.h -h _Chmath_handle -r math_err -o c c      \
       -o chf chf -s structdef_t -l func5 vfunc5
```

first extract function prototypes using function **processhfile()** from a header file `math.h` and saved the prototypes in `chfcreate/math.h`. Command **c2chf** then process function prototypes in file `chfcreate/math.h` to create the corresponding .c and .chf files in **c** and **chf** directories, respectively. Functions `func3()` and `func4()` return structure. Because the return type of function `func4()` is represented in a typedefed structure name `structdefname_t`, it is handled with option `-o structdefname_t` for command **c2chf**. Function

```
int func5(int a, ...);
```

has a variable number of arguments. If the correpsonding function with a fixed number of arguments

```
int vfunc5(int a, va_list ap);
```

exists, it can be handled by option `-l func5 vfunc5`.

Listing 1 – **math.h**

```
struct tag {int i;
            float f;
};
typedef struct tag2 {int i;
                     float f;
} structdef_t;
extern double erf(double x);
extern double hypot(double x, double y);
extern struct tag func3(int n);       /* return struct */
extern structdef_t func4(int n);      /* return struct */
extern int func5(int n, ...);         /* argument of ... */
extern int vfunc5(int n, va_list ap); /* argument of va_list */
```

Listing 2 – **chfcreate/math.h**

```
 double erf(double x);
 double hypot(double x, double y);
 struct tag func3(int n);
 structdef_t func4(int n);
 int func5(int n, ...);
 int vfunc5(int n, va_list ap);
```

Listing 3 – **math_err**

```
erf NaN
hypot NaN
```

Listing 4 – **c/math_chdl.c**

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

EXPORTCH double erf_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double retval;
```

```
    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    retval = erf(x);
    Ch_VaEnd(interp, ap);
    return retval;
}

EXPORTCH double hypot_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double y;
    double retval;

    Ch_VaStart(interp, ap, varg);
    x = Ch_VaArg(interp, ap, double);
    y = Ch_VaArg(interp, ap, double);
    retval = hypot(x, y);
    Ch_VaEnd(interp, ap);
    return retval;
}

EXPORTCH void func3_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int n;
    struct tag *retval;

    Ch_VaStart(interp, ap, varg);
    retval = Ch_VaArg(interp, ap, struct tag *);
    n = Ch_VaArg(interp, ap, int);
    *retval = func3(n);
    Ch_VaEnd(interp, ap);
}

EXPORTCH void func4_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int n;
    structdef_t *retval;

    Ch_VaStart(interp, ap, varg);
    retval = Ch_VaArg(interp, ap, structdef_t *);
    n = Ch_VaArg(interp, ap, int);
    *retval = func4(n);
    Ch_VaEnd(interp, ap);
}

EXPORTCH int func5_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int n;
    ChVaList_t ap_ch;
    void *ap_c;
    void *memhandle;
    int retval;

    Ch_VaStart(interp, ap, varg);
    n = Ch_VaArg(interp, ap, int);
```

```
    ap_ch = Ch_VaArg(interp, ap, ChVaList_t);
    ap_c = Ch_VaVarArgsCreate(interp, ap_ch, &memhandle);
    retval = vfunc5(n, ap_c);
    Ch_VaVarArgsDelete(interp, memhandle);
    Ch_VaEnd(interp, ap);
    return retval;
}

EXPORTCH int vfunc5_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int n;
    ChVaList_t ap_;
    void *ap_c;
    void *memhandle;
    int retval;

    Ch_VaStart(interp, ap, varg);
    n = Ch_VaArg(interp, ap, int);
    ap_ = Ch_VaArg(interp, ap, ChVaList_t);
    ap_c = Ch_VaVarArgsCreate(interp, ap_, &memhandle);
    retval = vfunc5(n, ap_c);
    Ch_VaVarArgsDelete(interp, memhandle);
    Ch_VaEnd(interp, ap);
    return retval;
}
```

Listing 5 – **chf/erf.chf**

```
double erf(double x) {
    void *fptr;
    double retval;

    fptr = dlsym(_Chmath_handle, "erf_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return NaN;
    }
    dlrunfun(fptr, &retval, erf, x);
    return retval;
}
```

Listing 6 – **chf/hypot.chf**

```
double hypot(double x, double y) {
    void *fptr;
    double retval;

    fptr = dlsym(_Chmath_handle, "hypot_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return NaN;
    }
    dlrunfun(fptr, &retval, hypot, x, y);
    return retval;
}
```

Listing 7 – **chf/func3.chf**

```
struct tag func3(int n) {
    void *fptr;
    struct tag retval;

    fptr = dlsym(_Chmath_handle, "func3_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return retval;
    }
    dlrunfun(fptr, NULL, NULL, &retval, n);
    return retval;
}
```

Listing 8 – **chf/func4.chf**

```
structdef_t func4(int n) {
    void *fptr;
    structdef_t retval;

    fptr = dlsym(_Chmath_handle, "func4_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return retval;
    }
    dlrunfun(fptr, NULL, NULL, &retval, n);
    return retval;
}
```

Listing 9 – **chf/func5.chf**

```
int func5(int n, ...) {
    void *fptr;
    va_list ap_ch;
    int retval;

    va_start(ap_ch, n);
    fptr = dlsym(_Chmath_handle, "func5_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return -1;
    }
    dlrunfun(fptr, &retval, func5, n, ap_ch);
    va_end(ap_ch);
    return retval;
}
```

Listing 10 – **chf/vfunc5.chf**

```
int vfunc5(int n, va_list ap_) {
    void *fptr;
    int retval;

    fptr = dlsym(_Chmath_handle, "vfunc5_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return -1;
    }
    dlrunfun(fptr, &retval, vfunc5, n, ap_);
    return retval;
}
```

See Also
processhfile(), removeFuncProto(), pkginstall.ch.

C.2 dlcomp

Synopsis
dlcomp *libname filename1.c* [*filename2.c* ...] [option] ...
dlcomp *libname* **[-]cplusplus** *filename1.cpp* [*filename2.cpp* ...] [option] ...

Purpose
To compile a C or C++ file to create an object file for building a dynamically loaded library in Ch.

Description
dlcomp creates an object file, with extensions .o in Unix and .obj in Windows, from the user inputed C and C++ files with extensions .c and .cpp, respectively. These object files will be used for building a dynamically loaded library in Ch. Command **dlcomp** can compile both C and C++ programs using the native C or C++ compiler provided by the operating system vendor. The object file will then be linked to form a dynamically loaded library specified by the user with file extension dl. Multiple C or C++ files can be specified at once on the command line. Each C/C++ file will be executed separately. The following are predefined macros that will be used during execution: "_REENTRANT", "NeedFunctionPrototypes". **dlcomp** also uses the following header file which are included by default: "/include/X11/Ch" **dlcomp** has been tested for C and C++ compilers on the following platforms: Windows, Linux, Mac OS X, Solaris, HPUX, FreeBSD, and QNX. If **dlcomp** is run on any other platform besides the ones listed above, the program will give the following error message: "Error: unknown OS for dlcomp".

In Windows with VC++, if the libname has surfix .dl, the command will compile the source code to use the multi-thread dynamically loaded system library.

In command **dlcomp**, one of the following macros: WINDOWS, LINUX, LINUXPPC, DARWIN, SOLARIS, HPUX, FREEBSD, and QNX, will be defined for the platform in use. With these macros, the users can write portable code which behaves differently in different platform, for example

```
#if defined(HPUX) /* code for HP-UX */
   size = 4;
#elif defined(SOLARIS) /* code for Solaris */
   size = 8;
#elif defined(LINUX) || defined(LINUXPPC) || defined(WINDOWS)
/* code for Linux and Windows */
   size = 16;
#else /* code for others */
   size = 32;
#endif
```

When the above source code fragment is compiled with **dlcomp**, only one statement is used according to different platforms. In HP-UX, the statement

```
size = 4;
```

will be used, whereas in Linux and Windows,

```
size = 16;
```

will be used.

Options

cplusplus: Command **dlcomp** uses C++ compile to compile the program.

Notes

Platform	Predefined Macros
Win32	WINDOWS, _POSIX_
Linus for Intel PC	LINUX
Linus for PowerPC	LINUXPPC
Mac OS X	DARWIN
Solaris	SOLARIS
HPUX	HPUX, _PROTOTYPES, _HPUX_SOURCE, HPUXOS
FreeBSD	FREEBSD
QNX	QNX

See Also

dlopen(), **dlerror**(), **dlsym**(), **dlrunfun**(), **dlclose**(), **dlfcn.h**, **ch.h**.

C.3 dllink

Synopsis
dllink *libname.dl* [**[-]cplusplus**] [**-embedded**] *objname1.o* [*objname2.o ...*] [*option*] ...
dllink *cmdname* [**[-]cplusplus**] [**-embedded**] *objname1.o* [*objname2.o ...*] [*option*] ...
dllink *cmdname.exe* [**[-]cplusplus**] [**-embedded**] *objname1.obj* [*objname2.obj ...*] [*option*] ...
dllink *libname.dl* [**[-]cplusplus**] [**-embedded**] *objname1.obj* [*objname2.obj ...*] [*option*] ...

Purpose
To build a dynamically loaded library in Ch.

Description
Command **dllink** dynamically links all the specified object files together to build dynamically loaded libary.

Command **dllink** has been tested on these platforms: Solaris, HPUX, Linux, Win32. If **dllink** is run on any other platform besides the ones listed above, the program will give the following error message: "Error: unknown OS for dllink".

If the first argument `libname.dl` following command **dllink** is ended with file extension .dl, a dynamically loaded library `libname.dl` will be created. If the first argument `cmdname` following command **dllink** is not ended with file extension .dl, **dllink** links the object files with file extension .o or .obj to create an executable binary command named `cmdname` or cmdname.exe in Unix and Windows, respectively. This can be used for testing purposes since all undefined external variables will be displayed.

Options
-cplusplus: This option will perform different tasks depending on which platform is used.

Platform	Action Taken
Solaris	-lCrun flag will be included.
Linux	g++ will be used instead of gcc
HPUX	-lC flag will be included.

-embedded: This option shall be added when building applications with Embedded Ch.

Notes
1. All the desired object files can be listed one after the other, but the total number of characters cannot exceed 4096 in a command line or a dllink error will occur.

See Also
dlopen(), **dlerror**(), **dlsym**(), **dlrunfun**(), **dlclose**(), **dlfcn.h**, **ch.h**.

C.4 pkginstall.ch

Synopsis
pkginstall.ch [-u] [-d directory] pkgname

Purpose
To install a Ch package.

Description
By default, **pkginstall**.ch installs a package `pkgname` in the current directory into the `CHHOME/package/pkgname` and copies required header files in `pkgname/include` into the `CHHOME/toolkit/include` directory. If the user has no permission to install `pkgname` in `CHHOME/package/pkgname`, it installs `pkgname` in `HOME/pkgname`.

If you want to install ch package into your preferred directory, you can specify it in the comand line. During the installation, it will modify `_ipath` and `_ppath` in `.chrc` in Unix or `_chrc` in Windows in the user home directory. During installation, an installation file is created under the `CHHOME/package/installed` directory with a list of the installed directories and files.

pkginstall.ch can also uninstall a Ch Package by removing header files installed into the `CHHOME/toolkit/include` directory and the package in the `CHHOME/package` directory based on the corresponding package file in `CHHOME/package/installed` directory.

Options
-u uninstallation.

-d install a package in a *directory*.

Example

```
(1) uninstall chpng:
          pkginstall.ch -u chpng
(2) install chpng in the default directory:
          pkginstall.ch chpng
(3) install chpng to a user specified directory:
          pkginstall.ch -d /your/preferred/directory chpng
```

See Also
processhfile(), removeFuncProto(), c2chf.

Appendix D

Functions for Building Dynamiaclly Loaded Library

The utility functions for automatically building dynamically loaded library with file extension .dl from header files are described in this appendix. A complete working example with source code for automatically building dynamically loaded library based on header files is available at the directory CHHOME/toolkit/demos/SDK/chapters/runc2chf_dll/sample in the distribution of Ch SDK.

Functions for Building Dynamiaclly Loaded Library

Function	Description
processcfile	Comment out special functions in a C program created by command `c2chf`.
processhfile	Process a header file readily for command `c2chf` to create function files and a C program for building a dynamically loaded library.
removeFuncProto	Remove function prototypes for special functions in a header file.

D.1 processcfile

Synopsis
int processcfile(**char** **dirname*, **char** * *filename*);

Purpose
Comments out special functions in *filename* based on functions in .c files under directory *dirname*.

Return Value
This function returns 0 on success and negative value on failure.

Parameters
dirname The directory where the chdl C program *filename* and other C programs with special functions are located.

filename A chdl C program created by command `c2chf`. It must located in the directory *dirname*.

Description
This function comment out special functions in *filename* based on functions in other C programs with file extension .c in the directory *dirname*. Special functions such as functions returning data type of struct or functions with argument or return type of pointer to functions have to be handled manually. The interface programs to special functions shall be located in the directory *dirname*. **This function is deplicated. Use functions removeFuncProto() instead.**

See Also
processhfile(), **removeFuncProto**().

D.2 processhfile

Synopsis
int processhfile(**char** **starttok*, **int** *flag*, **char** * *endtok*, **char** * *filefrom*, **char** * *fileto*, **char** * *deletefuncmacro*);

Purpose
Process a header file readily for program `c2chf` to create function files and a C file for building a dynamically loaded library.

Return Value
This function returns 0 on success and negative value on failure.

Parameters
starttok The start token for a function prototype such as `extern` starting at a new line.

flag A flag of 1 for keeping the start token, 0 for remove start token in the processed header file.

endtok The end token for a function prototype such as ";".

filefrom The original header file.

fileto The processed header file for program `c2chf`.

deletefuncmacro Function macro to be deleted.

Description
This function processes the orginal header file passed as the argument of `filefrom` and creates a new file passed back as the argument of `fileto`. Only function prototypes starting with the token `starttok` and ending token `endtok` will be kept in the file `fileto`. If the `flag` is 1, the starting token passed through `starttok` will be kept in the output file. Otherwise, the start token will be discarded in the output. If the string `starttoke` is "?", as shown below

```
processhfile("?", 1, ";", "input.h", "output.h",NULL);
```

all the function definitions in file `input.h` will be kept except comments, typedef, define, struct, union, enum, and so on. This is only used in the case in which you can't find out one `startrtok` to extract functions. It is not recommanded to use "?". If the first character of string `starttok` is '*', all characters starting from the beginning of a line till token in `starttok` will be kept. In this case, `flag` must be 1. If there is a space between first character '*' and token in `starttok`, the space is needed to make the token effective. The function will remove function calls defined as a macro if the macro to be deleted is passed through the argument `deletefuncmacro`. If the passed argument for `deletefuncmacro` is NULL, no macro will be deleted.

Example
Assume that an orignal header file `sample/chsample/include/sample.h` contains

```
#ifndef SAMPLE_H
#define SAMPLE_H

#ifdef _CH_
#pragma package <chsample>
#include <chdl.h>
LOAD_CHDL(sample);
#endif



/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>
#include <stdio.h>

#if defined(_WIN32)
#define EXPORT _declspec(dllexport)
#else
#define EXPORT
#endif

typedef int (*FUNPTR)(int n);

EXPORT extern double func1(double x);
EXPORT extern double func2(double x, double y);
EXPORT extern int func3(FUNPTR fp, int n);  /* arg is pointer to func */

#endif /* SAMPLE_H */
```

Function call

```
processhfile("EXPORT", 0, ";", "sample/chsample/include/sample.h",
             "chfcreate/sample.h", NULL);
```

will create the file `chfcreate/sample.h` with the following contents.

```
extern double func1(double x);
extern double func2(double x, double y);
extern int func3(FUNPTR fp, int n);
```

As another example, if the original header file `example.h` is given below

```
#include <ff.h>
#typedef int INTTYPE
struct{
  int a,
  void * l,
};

int CALLBACK func1();
CALLBACK double func2();
void CALLBACK func3();
int func4();
void func5();
double func7();
double func8( int i,
   char * text,
   float t
```

```
   );

void CALLBACK func9(
   int i,
   int j,
   char *k,
   double t);

CALLBACK func10(int a, int b,
    double p,
    char point,
    doubel dd
);
```

Execution of program below

```
int main(){
    processhfile("* CALLBACK", 1, ";", "example.h", "example1.h",NULL);
    processhfile("*CALLBACK",  1, ";", "example.h", "example2.h",NULL);
    return 0;
}
```

will create two files `example1.h` and `example2.h` as follows.
File `example1.h`

```
int CALLBACK func1();
void CALLBACK func3();
void CALLBACK func9(
   int i,
   int j,
   char *k,
   double t);
```

File `example2.h`

```
int CALLBACK func1();
CALLBACK double func2();
void CALLBACK func3();
void CALLBACK func9(
   int i,
   int j,
   char *k,
   double t);
CALLBACK func10(int a, int b,
    double p,
    char point,
    doubel dd
);
```

See Also
removeFuncProto(), **processcfile**().

D.3 removeFuncProto

Synopsis
int removeFuncProto(**char** **filename*, **char** * *funcname*, **int** *keepNum*);

Purpose
Remove a function prototype *funcname* for special functions in file *filename*.

Return Value
This function returns 0 on success and negative value on failure.

Parameters
filename A header file created by function `processhfile()`.

funcname The function name for which it will be removed from the file `filename`.

keepNum An integral value indicates how function prototypes are removed.

Description
This function removes function prototype `funcname` for special functions in *filename* Special functions such as functions returning data type of struct or functions with argument or return type of pointer to functions have to be handled manually. If `keepNum` is 0 or negative, remove all occurances of the function prototype. If `keepNum` is larger than 0, keep the keepNumth function prototype.

Example
Assume file `chfcreate/sample.h` contains

```
double func1(double x);
double func2(double x, double y);
TAG func3(int n);
int func4(int n);
int func4(float);
int func5(int n);
int func5(float);
int func5(double d);
```

After the following function calls

```
removeFuncProto("chfcreate/sample.h", "func3", 0);
removeFuncProto("chfcreate/sample.h", "func4", 0);
removeFuncProto("chfcreate/sample.h", "func5", 2);
```

file `chfcreate/sample.h` will contain

```
double func1(double x);
double func2(double x, double y);
int func5(float);
```

See Also
processhfile().

Appendix E

Porting Code with Ch SDK APIs to the Latest Version

E.1 Porting Code with Ch SDK APIs to Ch Version 5.0.0

Ch version 5.0 supports multi-threads. Like Embedded Ch SDK APIs, all Ch SDK APIs need a valid instance of Ch interpreter of type **ChInterp_t** as their first argument. Ch SDK APIs starting with a prefix **Ch_Va** shall add such an argument, i.e.,

```
Ch_XXX(NULL, ...);
```

should be changed to

```
Ch_XXX(interp, ...);
```

For example, the program `func_chdl.c` below

```
EXPORTCH double func_chdl(void *varg) {
    va_list ap;
    double x;
    double retval;

    Ch_VaStart(ap,varg);
    x = Ch_VaArg(ap, double);
    retval = func(x, y);
    Ch_VaEnd(ap);
    return retval;
 }
```

should be changed to

```
EXPORTCH double func_chdl(void *varg) {
    ChInterp_t interp;
    va_list ap;
    double x;
    double retval;
```

```
    Ch_VaStart(interp, ap,varg);
    x = Ch_VaArg(interp, ap, double);
    retval = func(x);
    Ch_VaEnd(interp, ap);
    return retval;
 }
```

Such modifications can be done automatically by running a Ch program `port47to50.ch`. For example, file `func_chdl.c` can be modified as `new_func_chdl.c` as follows.

```
ch port47to50.ch func_chdl.c new_func_chdl.c
```

Program `port47to50.ch` added

```
ChInterp_t interp;
```

and argument `interp` for functions **Ch_VaStart**(), **Ch_VaArg**(), and **Ch_VaEnd**().

To interface a C function with argument or return type of pointer to function, a valid Ch instance shall be used. A Ch interpreter instance obtained from function **Ch_VaStart**() can be assigned to a static variable or passed from function argument. For example, the program `funcptr_chdl.c` below

```
#include <ch.h>

typedef int (*FUNPTR)(int n);
static int fp_funarg(int n);
static void *fp_funptr;

EXPORTCH int func1_chdl(void *varg) {
   va_list ap;
   FUNPTR fp_ch, fp_c;
   int n;
   int retval;

   Ch_VaStart(ap, varg);
   fp_ch = Ch_VaArg(ap, FUNPTR);
   n = Ch_VaArg(ap, int);
   fp_funptr = (void *)fp_ch;
   if (fp_ch != NULL)  {
       fp_c = (FUNPTR)fp_funarg;
   }
   retval = func1(fp_c, n);
   Ch_VaEnd(ap);
   return retval;
}

static int fp_funarg(int n) {
    int retval;
    Ch_CallFuncByAddr(NULL, fp_funptr, &retval, n);
    return retval;
}
```

should be changed to

```
#include <ch.h>

typedef int (*FUNPTR)(int n);
static ChInterp_t interp;
static int fp_funarg(int n);
static void *fp_funptr;

EXPORTCH int func1_chdl(void *varg) {
   va_list ap;
   FUNPTR fp_ch, fp_c;
   int n;
   int retval;

   Ch_VaStart(interp, ap, varg);
   fp_ch = Ch_VaArg(interp, ap, FUNPTR);
   n = Ch_VaArg(interp, ap, int);
   fp_funptr = (void *)fp_ch;
   if (fp_ch != NULL)  {
       fp_c = (FUNPTR)fp_funarg;
   }
   retval = func1(fp_c, n);
   Ch_VaEnd(interp, ap);
   return retval;
}

static int fp_funarg(int n) {
     int retval;
     Ch_CallFuncByAddr(interp, fp_funptr, &retval, n);
     return retval;
}
```

Such modifications can be done automatically by a Ch program `port47to50.ch` with option `-s`. For example, file `funcptr_chdl.c` can be modified as `new_funcptr_chdl.c` as follows.

```
ch port47to50.ch -s funcptr_chdl.c new_funcptr_chdl.c
```

Program `port47to50.ch` added

```
static ChInterp_t interp;
```

and argument `interp` for functions **Ch_VaStart**(), **Ch_VaArg**(), and **Ch_VaEnd**(). It also changed

```
Ch_CallFuncByAddr(NULL, fp_funptr, &retval, n);
```

to

```
Ch_CallFuncByAddr(interp, fp_funptr, &retval, n);
```

E.2 Porting Code with Ch SDK APIs to Ch Version 5.1

Ch versions 5.0.3 and higher use data type **ChVaList_t** defined in header file **ch.h**, instead of **va_list** defined in header file **stdarg.h**, to represent in the C space for a variable number of arguments in the Ch space. Many APIs defined in header files **ch.h** and **embedch.h** for in Ch SDK and Embedded Ch, respectively, shall be changed to this new data type. For example, the program `func_chdl.c` below

```
EXPORTCH double func_chdl(void *varg) {
    ChInterp_t interp;
    va_list ap;
    double x;
    double retval;

    Ch_VaStart(interp, ap,varg);
    x = Ch_VaArg(interp, ap, double);
    retval = func(x);
    Ch_VaEnd(interp, ap);
    return retval;
 }
```

should be changed to

```
EXPORTCH double func_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x;
    double retval;

    Ch_VaStart(interp, ap,varg);
    x = Ch_VaArg(interp, ap, double);
    retval = func(x);
    Ch_VaEnd(interp, ap);
    return retval;
 }
```

Index