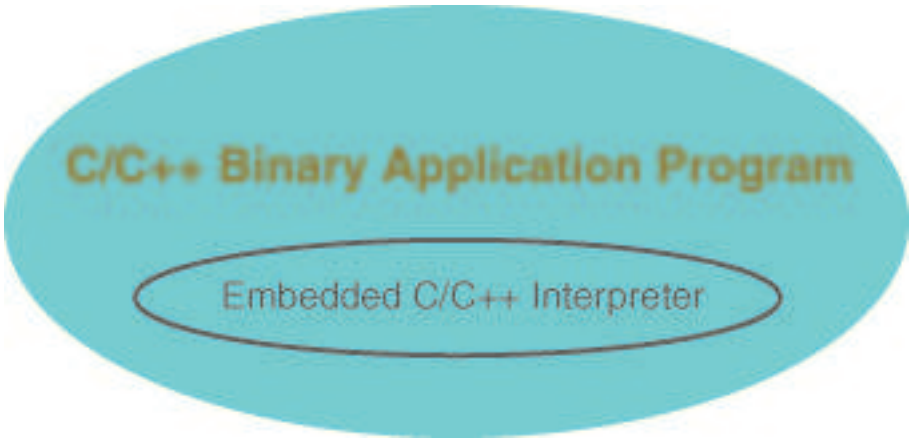
SoftIntegration®

The Ch Language Environment

Version 6.1

Embedded Ch 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 6.1, September 2008

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.

Embedded Ch includes distributable Embedded Ch Standard or Professional Edition. It requires a regular Ch Standard or Professional Edition and Ch SDK installed first. In most platforms, Ch SDK is bundled with both Ch Standard and Professional Editions.

By default, Embedded Ch includes distributable Embedded Ch Standard Edition. *Embedded Ch Standard Edition* contains all features of regular Ch Standard Edition, except that it cannot be started standalone. All capabilities in Ch Standard Edition can be invoked from your application program with Ch embedded. However, the size of *Embedded Ch Standard Edition* can be significantly reduced by removing the documentations, demo programs, Unix utilities in Windows, and toolkits such as OpenGL, Win32, GTK+. The size of such a slim *Embedded Ch Standard Edition* with all ISO C standard libraries is less than 3 Mb.

Applications developed using Embedded Ch Evaluation Edition can only run in a machine where Embedded Ch Evaluation Edition is installed.

This documentation contains detailed information about the use of Embedded Ch SDK. For information about Ch and Ch SDK, please check the corresponding documentation.

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->Ch6.1->Ch** indicates that you first select **Start**. Then select submenu **Programs** by pointing the mouse on **Programs**, followed by **Ch6.1**. 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 6.1, SoftIntegration, Inc., 2008.

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

- *The Ch Language Environment, — User's Guide*, version 6.1, SoftIntegration, Inc., 2008.

  This document presents language features of Ch for various applications.

- *The Ch Language Environment, — Reference Guide*, version 6.1, SoftIntegration, Inc., 2008.

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

- *The Ch Language Environment, — SDK User's Guide*, version 6.1, SoftIntegration, Inc., 2008.

  This document presents Software Development Kit for interfacing with C/C++ functions in static or dynamical libraries.

- *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

Embedding Ch Programs in C Space

Note: The source code and templates for all examples described in this document are available in CHHOME/toolkit/demos/embedch/chapters and CHHOME/toolkit/demos/embedch/appendix, where CHHOME is the directory where Ch is installed such as `C:\Ch` for Windows and `/usr/local/ch` for Unix. For Windows, some additional examples are available at and CHHOME/toolkit/demos/embedch for different compilers and options. It is recommended that you try these examples while reading this document.

1.1 Getting Started

In *Ch SDK (Software Development Kit) User's Guide*, how to call C/C++ functions from Ch scripts has been described. Ch SDK can be used to build interface, including wrapper functions and dynamically loaded library in .dl file, between Ch and C/C++ when a Ch program runs first. The Ch program then calls C/C++ functions in binary library. In this case, a Dynamically Loaded Libraries (DLLs) with binary C/C++ functions is loaded first by function **dlopen**(). Function **dlopen**() follows the Unix POSIX standard. It is a standard C function in Linux. It is equivalent to LoadLibrary() in Windows.

Then, the address of a function is obtained by function **dlsym**(). Finally, the function is invoked by function **dlrunfun**().

With Embedded Ch, typically, a binary application program is launched first. The program invokes Ch scritps (C/C++ scripts) through an Embedded Ch scripting engine. When a Ch scripting engine is embedded in a C/C++ executable program, the binary library functions and Ch scripting engine are part of the same application. The addresses of variables in both C/C++ and Ch spaces can be shared each other. Therefore, it is much simpler and easier to call C/C++ functions from Ch space or call Ch script functions from C/C++ space.

Embedded Ch provides macros and core APIs **Ch_Initialize**(), **Ch_RunScript**() and **Ch_End**() in header file **embedch.h** located in the directory `CHHOME/extern/include`. In this document, these data type, macros and APIs will be described in detail with examples.

Program 1.1 is an example of C programs with embedded Ch. The embedded Ch executes a Ch program, named `embedch1.ch`, with only one statement

```
printf("hello from embedch1.ch\n");
```

After compiling and linking file `embedch1.c` with a makefile such as one shown in Program 1.2 for Unix, the binary executable file `embedch1.exe` is generated.

The output of executing application program `embedch1.exe` is as follows.

```
/*****************************************************************
* File Name: embedch1.c
* executing Ch program in C program with embedded Ch
* (with default options)
*****************************************************************/
#include <embedch.h>
#include <stdio.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"embedch1.ch", NULL};

  /* initialize embedded Ch */
  status = Ch_Initialize(&interp, NULL);
  if(status)
    printf("Error: initialization of Embedded Ch failed\n");

  /* run an embedded Ch program indicated by argvv */
  status = Ch_RunScript(interp,  argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch1.ch failed\n");

  /* release memory and terminate the shell */
  Ch_End(interp);
}
```

Program 1.1: A C program with embedded Ch (embedch1.c).

```
> embedch1.exe
hello from embedch1.ch
```

The Ch program `embedch1.ch` has been executed to generate the output.

Let's take a close look at Program 1.1 to understand the procedure of embedding Ch in C/C++ programs. First, the header file **embedch.h** in which APIs are specially defined for embedded Ch must be included.

Next, APIs are used to involve Embedded Ch. Variable `argvv` is an array of pointers to char which is used to keep the file name of the Ch program to be executed, as well as arguments passed to the Ch program. The initialization

```
char *argvv[]={"embedch1.ch", NULL};
```

indicates that Ch program `embedch1.ch` will be called without argument. The last element of `argvv` should be a NULL pointer.

Figure 1.1 illustrates how a Ch program is executed by these three core APIs of embedded Ch. The function call

```
Ch_Initialize(&interp, NULL)
```

initializes a Ch interpreter. with default options. The startup file CHHOME/config/chrc will be executed in this initialization function. The first argument is a pointer to **ChInterp_t**, a Ch interpreter. The data type **ChInterp_t** is defined in header file **ch.h**. All macros and functions defined in header file **ch.h** are available in Embedded Ch. The variable `interp` for an instance of Ch interpreter is used in all subsequent calls. The data type of the first argument of all Ch APIs is **ChInterp_t**, except for function **Ch_Initialize**() with the data type of its first argument of a pointer to **ChInterp_t**. The second NULL in the argument list indicates default options. User-defined options of Ch shell will be described later. Function call

```
#assume CHHOME is /usr/local/ch
#If not, change /usr/local/ch to CHHOME such as /usr/ch
CC= cc
INC=-I/usr/local/ch/extern/include
CFLAG=
LFLAG=/usr/local/ch/extern/lib/libchsdk.a \
      /usr/local/ch/extern/lib/libembedch.a -lm -lsocket -ldl

target: embedch1.exe embedch1b.exe embedch2.exe reopen.exe \
        embedch.exe chhome.exe embedchprograms.exe parseprograms.exe \
        thread.exe threads.exe append.exe

embedch.exe: embedch.c
        $(CC) embedch.c $(INC) $(LFLAG) $(CFLAG) -o embedch.exe
embedch1.exe: embedch1.c
        $(CC) embedch1.c $(INC) $(LFLAG) $(CFLAG) -o embedch1.exe
embedch1b.exe: embedch1b.c
        $(CC) embedch1b.c $(INC) $(LFLAG) $(CFLAG) -o embedch1b.exe
embedch2.exe: embedch2.c
        $(CC) embedch2.c $(INC) $(LFLAG) $(CFLAG) -o embedch2.exe
reopen.exe: reopen.c
        $(CC) reopen.c $(INC) $(LFLAG) $(CFLAG) -o reopen.exe
chhome.exe: chhome.c
        $(CC) chhome.c $(INC) $(LFLAG) $(CFLAG) -o chhome.exe
embedchprograms.exe: embedchprograms.c
        $(CC) embedchprograms.c $(INC) $(LFLAG) $(CFLAG) -o embedchprograms.exe
parseprograms.exe: parseprograms.c
        $(CC) parseprograms.c $(INC) $(LFLAG) $(CFLAG) -o parseprograms.exe
thread.exe: thread.c
        $(CC) thread.c $(INC) $(LFLAG) $(CFLAG) -o thread.exe
threads.exe: threads.c
        $(CC) threads.c $(INC) $(LFLAG) $(CFLAG) -o threads.exe -lpthread
append.exe: append.c
        $(CC) append.c $(INC) $(LFLAG) $(CFLAG) -o append.exe

clean:
        rm -f *.o *.obj *.exe *.dl *.lib *.exp
```

Program 1.2: Makefile in Unix for all examples in this chapter.

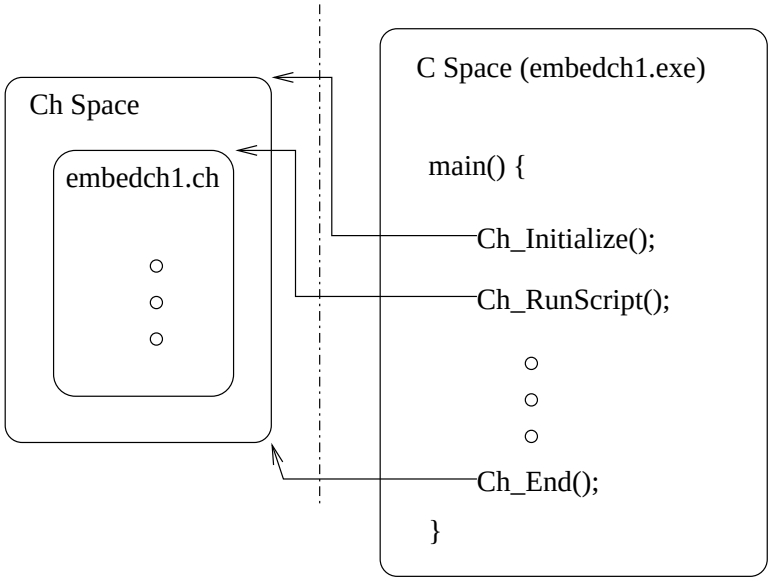


Figure 1.1: Core APIs of embedded Ch.

```
status = Ch_RunScript(interp, argvv);
```

runs the Ch program indicated in argument `argvv[0]` in the Ch interpreter initialized by function **Ch_Initialize**(). The return value `status` is **CH_OK** on successful execution, **CH_ERROR** on failure. These two macros for return status are defined in header file **ch.h**. Before the program exits, function **Ch_End()** needs to be called to end the embedded Ch initialized by function **Ch_Initialize()**. Function calls **Ch_Initialize()** at the beginning and **Ch_End()** at the end are required for embedding Ch into C/C++ applications. Function **Ch_RunScript()** can be replaced by other APIs in the Embedded Ch.

Finally, The libraries `libchsdk.a` and `libembedch.a` for Unix or `chsdk.lib` and `embedch.lib` for Windows located in the directory `CHHOME/extern/lib` are linked with compiled objects to create a binary executable application program. The sample Makfile file in `CHHOME/toolkit/demos/embedch/chapters/chapter1/Makefile` contains options for compilation and linking application programs with Embedded Ch for specific platforms. Users can still use a graphical Integrated Development Environment to develop applications with Embedded Ch. Additional information about how to setup paths for header files and libraries for different platforms for portable compilation and linking using Ch SDK can be found in Chapter 1 in *Ch SDK User's Guide.*

1.2 Building Executables in Windows

Executables with Embedded Ch in Windows can be developed using different compilers either in command shell or in an Integrated Development Environment (IDE).

1.2.1 Building Executables in Windows Using Command Shell

The makefile shown in Program 1.3 can be used to build all executables in Ch command shell for Windows for all examples described in this document. Assume C:/Ch is the home directory for Ch, the header files **ch.h** and **embedch.h** are located in the directory `C:/Ch/extern/include`, which should be added to the search path for header files in compilation. By default, the option */MD* shall be added in compilation for using the multi-thread dynamically linked system libraries. The libraries `chsdk.lib` and `embedch.lib`

```
#assume CHHOME is C:/Ch
#If not, change C:/Ch to CHHOME such as D:/Ch
CC= cl
INC=-IC:/Ch/extern/include
CFLAG=/MD
LFLAG=C:/Ch/extern/lib/chsdk.lib \
      C:/Ch/extern/lib/embedch.lib advapi32.lib

target: embedch1.exe embedch1b.exe embedch2.exe reopen.exe \
        embedch.exe chhome.exe embedchprograms.exe parseprograms.exe \
        thread.exe threadswin.exe append.exe

embedch.exe: embedch.c
        $(CC) embedch.c $(INC) $(LFLAG) $(CFLAG) /Feembedch.exe
embedch1.exe: embedch1.c
        $(CC) embedch1.c $(INC) $(LFLAG) $(CFLAG) /Feembedch1.exe
embedch1b.exe: embedch1b.c
        $(CC) embedch1b.c $(INC) $(LFLAG) $(CFLAG) /Feembedch1b.exe
embedch2.exe: embedch2.c
        $(CC) embedch2.c $(INC) $(LFLAG) $(CFLAG) /Feembedch2.exe
reopen.exe: reopen.c
        $(CC) reopen.c $(INC) $(LFLAG) $(CFLAG) /Fereopen.exe
chhome.exe: chhome.c
        $(CC) chhome.c $(INC) $(LFLAG) $(CFLAG) /Fechhome.exe
embedchprograms.exe: embedchprograms.c
        $(CC) embedchprograms.c $(INC) $(LFLAG) $(CFLAG) /Feembedchprograms.exe
parseprograms.exe: parseprograms.c
        $(CC) parseprograms.c $(INC) $(LFLAG) $(CFLAG) /Feparseprograms.exe
thread.exe: thread.c
        $(CC) thread.c $(INC) $(LFLAG) $(CFLAG) /Fethread.exe
threadswin.exe: threadswin.c
        $(CC) threadswin.c $(INC) $(LFLAG) $(CFLAG) /Fethreadswin.exe
append.exe: append.c
        $(CC) append.c $(INC) $(LFLAG) $(CFLAG) /Feappend.exe

clean:
        rm -f *.o *.obj *.exe *.dl *.lib *.exp
```

Program 1.3: Makefile in Windows for all examples in this chapter.

for applications compiled with option */MD* are located in the directory `C:/Ch/extern/lib`, which should be added to the search path for libraries in linking the executable program. If an application is compiled with option */MDd* for debug multi-thread dynamically linked system libraries. libraries `chsdk_mdd.lib` and `embedch_mdd.lib` should be used. If an application is compiled with option */MT* for multi-thread static linked system libraries. libraries `chsdk_mt.lib` and `embedch_mt.lib` should be used. If an application is compiled with option */MTd* for debug multi-thread static linked system libraries. libraries `chsdk_mtd.lib` and `embedch_mtd.lib` should be used. For some applications, the system library `advapi32.lib` may also need to be linked to create an executable application program.

The startup file `~/_chrc` in the user's home directory can be configured to setup the Visual C++ (VC++) in Windows for automatic compilation using a Makefile in a Ch shell. This startup file can be created by running the following command in a command shell.

```
ch -d
```

If a VC++ in .NET 2005 is installed at `C:/Program Files/Microsoft Visual Studio 8`, 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 8/VC/bin;");
_path = stradd(_path, "C:/Program Files/Microsoft Visual Studio 8/COMMON7/IDE;");
putenv(stradd("LIB=C:/Program Files/Microsoft Visual Studio 8/VC/lib;",
       "C:/Program Files/Microsoft Visual Studio 8/VC/PlatformSDK/lib;",
       "C:/Program Files/Microsoft Visual Studio 8/SDK/v2.0/lib;", getenv("LIB")));
putenv(stradd("INCLUDE=C:/Program Files/Microsoft Visual Studio 8/VC/include;",
       "C:/Program Files/Microsoft Visual Studio 8/VC/PlatformSDK/Include;",
       "C:/Program Files/Microsoft Visual Studio 8/VC/atlmfc/include;.;",
        getenv("INCLUDE")));
```

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",
       "C:/Program Files/Microsoft Visual Studio .NET 2003/VC7/atlmfc/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;");
putenv(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.

In Windows, command

```
nmake -f makefile
```

or

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

will generate executable programs defined in the makefile. To run a generated program, simply type the command such as `embedch1.exe` at the Windows prompt.

1.2.2 Building Executables in Windows Using Visual .NET

Assume Ch is installed in the directory `C:\Ch`. To compile the code using Embedded Ch SDK in Visual .NET, follow the procedures below.

- Create a Win32 project or Win32 console project.
- Header files **ch.h** and **embedch.h** are located in the directory `CHHOME/extern/include`. Some sample programs described in this document use header files in the current working directory. Add the directory `CHHOME/extern/include` and the current working directory. to the header file search path. Click `Project => Properties ...`, it will pop up an Properties pages window. From left panel, click `Configuration Properties => C/C++ => General`, select `Additional Include Directories` from right panel, Add the include path `C:\Ch\extern\include;..`
- Take one of the following actions:

  (a) Add compilation option `/MD`. Click `Command Line` under `C/C++`, then select `Additional Options`, Add `/MD`.

  The application program should be linked with Embedded Ch libraries `chsdk.lib` and `embedch.lib` located in the directory `CHHOME/extern/lib`, To add this directory, in the library search path as follows. Under the same Configuration Properties Window, select `Linker`, select `Additional Library Directories` from right panel, Add `C:\Ch\extern\lib`, then click `Apply` to finalize the settings in Property Window. Click `Command Line` under `Linker`, then select `Additional Options`, Add `chsdk.lib, embedch.lib`. Finally, click `OK` to finalize the settings in Project Settings.

  (b) Add compilation option `/MDd` and use Embedded Ch libraries `chsdk_mdd.lib` and `embedch_mdd.lib`.

  (c) Add compilation option `/MT` and use Embedded Ch libraries `chsdk_mt.lib` and `embedch_mt.lib`.

  (c) Add compilation option `/MTd` and use Embedded Ch libraries `chsdk_mtd.lib` and `embedch_mtd.lib`.

The source code for Visual .NET project files `embedfuncmain.vcproj` and `embedclassmain.vcproj` for examples described in sections 5.1 and 5.5 can be found in the distribution of Embedded Ch for Windows in the directories `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain` and `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedclassmain`, respectively. They can be opened from Visual .NET directly. Please check `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain/readme` and `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedclassmain/readme` for more details on how to run the program.

1.2.3 Building Executables in Windows Using Visual C++ 6.0

Assume Ch is installed in the directory `C:\Ch`. To compile the code using Embedded Ch SDK in VC++ 6.0, follow the procedures below.

- Create a Win32 project or Win32 console project.
- Header files **ch.h** and **embedch.h** are located in the directory `CHHOME/extern/include`. Some sample programs described in this document use header files in the current working directory. Add the directory `CHHOME/extern/include` and the current working directory. to the header file search path. Click `Tools => Options`, it will pop up an Option window. Click `Directories`, select `Win32` under `platform` and select `Include files` under `Show directories for:`. Finally add `C:\Ch\extern\include;.` under `Directories`.
- The application program should be linked with Embedded Ch libraries `chsdk.lib` and `embedch.lib` located in the directory `CHHOME/extern/lib`, To add this directory, in the library search path as follows. Under the same Option Window, select `Library files` under `Show directories for:`, add `C:\Ch\extern\lib` under `Directories`, then click `OK` to finalize the settings in Option Window. Finally, click `Projects => Settings ...`, and select `Link`, add `chsdk.lib`, `embedch.lib` under `Project Options`. Also add `/NODEFAULTLIB:"LIBCD.LIB"` under `Project Options`. Then, click `OK` to finalize the settings in Project Settings.

The source code for Visual C++ 6.0 workspace and project files for examples described in sections 5.1 and 5.5 can be found in the distribution of Embedded Ch for Windows in the directories `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain` and `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedclassmain`, respectively. You will find VC++ project files `embedfuncmain.dsw`, `embedfuncmain.dsp`, `embedclassmain.dsw`, and `embedclassmain.dsp` in these two directories. Please check `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain/readme` and `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedclassmain/readme` for more details on how to run the program.

1.2.4 Building Executables in Windows Using Intel C++ Compiler

To use Intel C++ compiler, you need to install Visual .NET first. If Intel C++ compiler is installed at `"C:/Program Files/Intel/Compiler/C++/10.1.021/IA32"`, the code below needs to be added to the startup file `_chrc` in the user's home directory.

```
_path = stradd(_path, "C:/Program Files/Intel/Compiler/C++/10.1.021/IA32/Bin;");
putenv(stradd("LIB=C:/Program Files/Intel/Compiler/C++/10.1.021/IA32/lib;",
              getenv("LIB")));
putenv(stradd("INCLUDE=C:/Program Files/Intel/Compiler/C++/10.1.021/IA32/include;",
              ".;", getenv("INCLUDE"));
```

You also need to use the Intel C++ compiler and linker command **icl.exe** by changing `"CC = cl"` to `"CC = icl"` in the distributed make files such as `Makefile`.

1.2.5 Building Executables in Windows Using Borland C++ Compiler

Executable binary programs and dynamic 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 files **ch.h** and **embedch.h**, located in the directory `CHHOME/extern/include`, are needed to compile code using Embedded Ch.

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

1.2.6 Building Executables in Windows Using Borland CBuilder

Executable binary programs and dynamic loaded libraries in Windows can also be built using Borland CBuilder version 6 or above. An example using Borland CBuilder is are distributed in Embedded Ch SDK in the directory `CHHOME/toolkit/demos/embedch/Borland/CBuilder`. You may click or type `CHHOME/toolkit/demos/embedch/Borland/CBuilder/embedfuncmain.bpr` to start the project for compilation and linking. More information can be found in `CHHOME/toolkit/demos/embedch/Borland/CBuilder/readme.txt`.

1.3 Other Essential APIs and Options for Embedded Ch

When an embedded Ch program is executed, it first will be parsed, then executed. **Ch_RunScript**() performs these two steps together. Often time it is desirable to check the syntax of the embedded Ch program before it is executed. This can be accomplished by functions **Ch_ParseScript**() and **Ch_ExecScript**() as shown in Program 1.4. Function **Ch_ParseScript**() parses a Ch program only. Later, the parsed program can be executed through function **Ch_ExecScript**(). The arguments of function **Ch_ParseScript**() is the same as those of function **Ch_RunScript**(). The function returns 0 if the program is parsed successfully. Otherwise, it returns non-zero. Like other functions, the first argument of function **Ch_ExecScript**() is Ch interperter. The second argument is the first element of the second argument of array of pointer to char in function **Ch_ParseScript**(). The function returns 0 on successful and -1 on failure.

Binary executable C/C++ programs can call Ch functions which are subject to change at the user's preference. In Program 1.1, the second argument of function `Ch_Initialize`() is NULL to indicate the

```
/*************************************************************
* File Name: embedch1b.c
* executing Ch program in C program with embedded Ch
* (using separate parse and execution functions)
*************************************************************/
#include <embedch.h>
#include <stdio.h>
#include <string.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"embedch1.ch", NULL};

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* parse embedded Ch program indicated by argvv */
  status = Ch_ParseScript(interp,  argvv);
  if(status == CH_ERROR)
    printf("Error: parse program embedch1.ch failed\n");
  else {
    /* execute embedded Ch program embedch1.ch */
    status = Ch_ExecScript(interp,  argvv[0]);
    if(status)
      printf("Error: execute script embedch1.ch failed\n");
  }

  /* release memory and terminate the shell */
  Ch_End(interp);
}
```

Program 1.4: A C program with embedded Ch with separate parse and execution (embedch1b.c).

default options of Ch shell are used. In Program 1.5, we will describe how to use user-defined options with the data type **ChOptions_t**. The variable `option` is declared as a structure of type **ChOptions_t** with member fields `shelltype` and `chhome` as follows.

```
typedef struct ChOptions{
    int shelltype;  // shell type
    char *chhome;   // Embedded Ch home directory
} ChOptions_t;
```

- Field `shelltype` indicates the type of the Ch shell. It could be **CH_REGULARCH** for a regular Ch shell or **CH_SAFECH** for a safe Ch shell. By default, its value is **CH_REGULARCH**.
- Field `chhome` contains the home directory for the Embedded Ch. If the value of field `chhome` is NULL, the Ch home directory for the regular Standard or Professional Edition is used for the Embedded Ch. In most applications, the Embedded Ch home directory is different from that for a regular Ch. In this example, the Embedded Ch home directory is specified as `../../../embedch`. The startup file `config/chrc` in the Embedded Ch home directory will be used. More information about Ch startup file and safe Ch shell is available in *Ch User's Guide*.

The values for arguments `argc` and `argv` of function `main()` inside an embedded Ch program (`embedch2.ch`) shown in Program 1.6 are obtained from the second argument of function **Ch_RunScript()**. **Ch_RunScript(interp, argvv)** is similar to invoke program `embedch2.ch` in a command shell as follows.

```
embedch2.ch opt1 opt2
```

The output of executing file `embedch2.exe` is shown as follows.

```
> embedch2.exe
argc = 3
argv[0] = embedch2.ch
argv[1] = opt1
argv[2] = opt2
i in fun() = 5
i in fun() = 10
>
```

The Ch program `embedch2.ch` is executed to generate the above output when the function **Ch_RunScript()** is called. At that point, the Embedded Ch maintains the states of the interpreted program. Functions such as **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**(), **Ch_CallFuncByAddr**(), a **Ch_ExprCalc**(), and **Ch_ExprEval**() can be used to access the states. In Program 1.5, **Ch_CallFuncByName**() is used to call function `fun()` in the embedded program `embedch2.ch` explicitly.

If an embedded Ch program has multiple functions, but without function main() as shown below.

```
/* embedded Ch program */
/* header files */
int func1() {
  /* ... */
}
void func2() {
  /* ... */
}
```

```
/*************************************************************
* File Name: embedch2.c
* executing Ch program in C program with embedded Ch
* (with user-defined options)
*************************************************************/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"embedch2.ch", "opt1", "opt2", NULL};
  ChOptions_t option;
  int retval;

  /* initialize embedded Ch */
  option.shelltype = CH_REGULARCH;
  option.chhome = strdup("../../../../embedch");  /* Embedded Ch Home */
  Ch_Initialize(&interp, &option);

  /* run an Embedded Ch program indicated by argvv */
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch2.ch failed\n");
  Ch_CallFuncByName(interp, "fun", &retval, 10);

  /* release memory and terminate the shell */
  free(option.chhome);
  Ch_End(interp);
}
```

Program 1.5: A C program with embedded Ch with user-defined options (embedch2.c).

```
#include <stdio.h>

int fun(int i) {
  printf("i in fun() = %d\n", i);
  return 0;
}
int main(int argc, char **argv){
  int i = 0;
  printf("argc = %d\n", argc);
  while(argv[i] != NULL) {
    printf("argv[%d] = %s\n", i, argv[i]);
    i++;
  }
  fun(5);
  return 0;
}
```

Program 1.6: An embedded Ch program (embedch2.ch).

After **Ch_RunScript**() is called, **Ch_CallFuncByName**() can be used to invoke different functions inside the embedded Ch program separately as follows.

```
/* ... */
int status, retval;
status = Ch_RunScript(interp, argvv);
Ch_CallFuncByName(interp, "func1", &retval);
Ch_CallFuncByName(interp, "func2", NULL);
/* ... */
```

1.4 Incremental Execution by Appending Code to an Embedded Ch Program

For some applications, it is desirable to append a piece of code to an embedded program after it has been executed. Embedded Ch provides this capability through functions **Ch_AppendRunScript**() and **Ch_AppendParseScript**(). These functions append a piece of code to an embedded Ch program executed by functions **Ch_RunScript**() or **Ch_ExecScript**(). Like other APIs, the first arguments of functions **Ch_AppendRunScript**() and **Ch_AppendParseScript**() are Ch interpreter. The second argument of these two functions contains the code to be appended to the existing embedded Ch program inside the interpreter. The function **Ch_AppendRunScript**() both parses and executes the code, whereas function **Ch_AppendParseScript**() only parses the code to check its syntax. The code is not executed. Functions **Ch_ExprEval**(), **Ch_CallFuncByAddr**(), or **Ch_CallFuncByName**() shall be called to execute the code. This implies that functions shall be appended by function **Ch_AppendParseScript**(). These two appending functions return 0 on successful and -1 on failure.

Application of these two functions is illustrated in Program 1.7. Program 1.7 is modified based on Program 1.5. Different from Program 1.5, Program 1.7 execute two fragments of Ch code after the embedded Ch program Program 1.6 has been executed by function **Ch_RunScript**(). The code fragment `code1` which calls function `fun()` in Program 1.6 and its own function `fun2()` in Program 1.7 is executed by **Ch_AppendRunScript**(). The code fragment `code2` is parsed by **Ch_AppendParseScript**() first. Then, function **Ch_ExprParse**() is called to check if the expression `fun3(30)` is valid. Finally, function **Ch_ExprEval**() is invoked to call function `fun3()` in the code fragment `code2`. The output of executing Program 1.7 is shown as follows.

```
> append.exe
argc = 3
argv[0] = embedch2.ch
argv[1] = opt1
argv[2] = opt2
i in fun() = 5
i in fun() = 10
i in fun2() = 20
i in fun() = 100
i in fun3() = 30
```

The code that can be appended to an embedded Ch program has some restrictions. For example, it shall not contain preprecessing directives and comments delimited by '//'. To overcome these restrictions, Functions **Ch_AppendRunScriptFile**() and

```
/* File Name: append.c
* Executing Ch program in C proram with embedded Ch
* (Append code after embedded Ch program has executed) */
#include <embedch.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *code1 = "\
fun(10);\
void fun2(int i) {\
  printf(\"i in fun2() = %d\\n\", i);\
}\
fun2(20);";

char *code2 = "\
void fun3(int i) {\
  fun(100);\
  printf(\"i in fun3() = %d\\n\", i);\
}";

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"embedch2.ch", "opt1", "opt2", NULL};
  ChOptions_t option;
  int retval;

  /* initialize embedded Ch */
  option.shelltype = CH_REGULARCH;
  option.chhome = strdup("../../../../embedch");  /* Embedded Ch Home */
  Ch_Initialize(&interp, &option);

  /* run an Embedded Ch program indicated by argvv */
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch2.ch failed\n");
  status = Ch_AppendRunScript(interp, code1);
  if(status == CH_ERROR)
    printf("Error: Ch_AppendRunScript(interp, code1) failed\n");
  /*** Ch_AppendParseScript() will parse, but not execute code ***/
  status = Ch_AppendParseScript(interp, code2);
  if(status == CH_ERROR)
    printf("Error: Ch_AppendParseScript(interp, code2) failed\n");
  else {
    status = Ch_ExprParse(interp, "fun3(30)");
    if(status == CH_ERROR)
      printf("Error: Ch_ExprParse(interp, \"fun3(30)\") failed\n");
    else
      Ch_ExprEval(interp, "fun3(30)");
  }
  /* release memory and terminate the shell */
  free(option.chhome);
  Ch_End(interp);
}
```

Program 1.7: Append code after embedded Ch program has executed.

Ch_AppendParseScriptFile() can be used to append the code located in a file to an embedded Ch program. More information about using functions **Ch_AppendRunScript**() and **Ch_AppendRunScriptFile**() can be found in section 3.4.

1.5 Embedding Ch in Multi-Thread Applications

In some applications, multiple Ch programs need to be executed in a single Ch interpreter embedded in an application. Function **Ch_RunScript**() or **Ch_ExecScript**() cannot be invoked multiple times through a single interpreter. To execute multiple embedded Ch programs sequentially inside an application program with a single instance of Ch interpreter, functions **Ch_RunScriptM**(), or **Ch_ParseScript**() and **Ch_ExecScriptM**() shall be used. For example, only one instance of Ch interpreter is created by function **Ch_Initialize()** in Program 1.8. After the start-up file CHHOME/config/chrc is executed through **Ch_Initialize()**, embedded Ch programs `embedch1.ch` and `embedch2.ch` are executed by calling **Ch_RunScriptM**() twice. Then, Ch program `embedch2.ch` again is parsed by **Ch_ParseScript**() and executed by **Ch_ExecScriptM**(). Finally, the Ch interpreter is terminated by function **Ch_End()**. After a Ch program is executed by either **Ch_RunScriptM**() or **Ch_ExecScriptM**(), its global variables and functions shall not be accessed by other APIs.

```
/*************************************************************
* File Name: embedchprograms.c
* executing multiple Ch programs in a C program with embedded Ch
*************************************************************/
#include <embedch.h>
#include <stdio.h>
#include <string.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argv1[]={"embedch1.ch", NULL};
  char *argv2[]={"embedch2.ch", "opt1", "opt2",NULL};

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* execute embedch1.ch */
  status = Ch_RunScriptM(interp, argv1);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch1.ch failed\n");
  /* execute embedch2.ch */
  status = Ch_RunScriptM(interp, argv2);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch2.ch failed\n");
  /* execute embedch2.ch */
  status = Ch_ParseScript(interp, argv2);
  if(status == CH_ERROR)
    printf("Error: parsing program embedch2.ch failed\n");
  status = Ch_ExecScriptM(interp, argv2[0]);
  if(status == CH_ERROR)
    printf("Error: executing program embedch2.ch failed\n");


  /* release memory and terminate the interpreter */
  Ch_End(interp);
}
```

Program 1.8: A program executing multiple embedded Ch programs (embedchprograms.c).

```
/*********************************************************************
* File Name: thread.c
* executing multiple Ch programs using multiple instances of Ch interpreter.
*********************************************************************/
#include <embedch.h>
#include <stdio.h>
#include <string.h>

int main() {
  ChInterp_t interp1, interp2;
  int status, retval;
  char *argv1[]={"embedch1.ch", NULL};
  char *argv2[]={"embedch2.ch", "opt1", "opt2",NULL};

  /* initialize an embedded Ch interp1 */
  Ch_Initialize(&interp1, NULL);
  /* execute embedch1.ch */
  status = Ch_RunScriptM(interp1, argv1);
  if(status == CH_ERROR)
    printf("Error: execution of embedch1.ch in interp1 failed\n");
  /* execute embedch2.ch */
  status = Ch_RunScriptM(interp1, argv2);
  if(status == CH_ERROR)
    printf("Error: execution of embedch2.ch in interp1 failed\n");

  /* initialize another embedded Ch interp2 */
  Ch_Initialize(&interp2, NULL);
  /* execute embedch2.ch */
  status = Ch_RunScript(interp2, argv2);
  if(status == CH_ERROR)
    printf("Error: execution of embedch2.ch in interp2 failed\n");

  /* use variables and functions in embedch2.ch in interp2 */
  Ch_CallFuncByName(interp2, "fun", &retval, 10);

  /* release memory and terminate interperters */
  Ch_End(interp1);
  Ch_End(interp2);
}
```

Program 1.9: A program executing multiple embedded Ch programs using multiple instances of Ch interpreter (thread.c).

Program 1.9 illustrates how to invoke multiple Ch interpreters embedded in an application with a single thread. In this example, two function calls **Ch_Initialize()** initialize two instances `interp1` and `interp2` of Ch interpreter. The same Ch interpreter `interp1` executes both Ch programs `embedch1.ch` and `embedch2.ch` by calling function **Ch_RunScriptM**() twice. Global variables in embedded Ch programs `embedch1.ch` and `embedch2.ch` are in separate namespaces. There is no memory collisions between `embedch1.ch` and `embedch2.ch`. The global variables in these Ch programs executed by **Ch_RunScriptM**(), however, cannot be accessed later in the host program. Global variables for the Ch program `embedch2.ch` loaded by different interpreters are also in different namespaces. The instance of Ch interpreter `interp2`, executes the Ch program `embedch2.ch` using **Ch_RunScript**(). The program `embedch2.ch` executed by **Ch_RunScript**() remains inside the memory so that other Embedded Ch APIs, such as **Ch_CallFuncByName**(), can be used to access global variables and functions before the interpreter

is terminated by **Ch_End()**.

Typical scenarios for running embedded Ch programs in a single-thread application are as follows:

- ```
  Ch_Initialize(&interp, option);
  while(1) {
      Ch_RunScriptM(interp, argv);
  }
  Ch_End(interp);
  ```

- ```
  Ch_Initialize(&interp, option);
  while(1) {
      Ch_ParseScript(interp, argv);
      Ch_ExecScriptM(interp, argv[0]);
  }
  Ch_End(interp);
  ```

- ```
  Ch_Initialize(&interp, option);
  while(1) {
      Ch_RunScriptM(interp, argv1);
      Ch_ParseScript(interp, argv2);
      Ch_ExecScriptM(interp, argv2[0]);
  }
  Ch_End(interp);
  ```

- ```
  Ch_Initialize(&interp, option);
  while(1) {
      Ch_ParseScript(interp, argv);
      /* other API such as Ch_CallFuncByName() to access
         variables and functions in script argv[0] */
  }
  Ch_End(interp);
  ```

- ```
  Ch_Initialize(&interp, option);
  while(1) {
      Ch_ParseScript(interp, argv);
      Ch_AppendParseScript(interp, code);
      Ch_AppendParseScriptFile(interp, filename);
      /* other API such as Ch_CallFuncByName() to access variables
         and functions in script argv[0], code, and filename */
  }
  Ch_End(interp);
  ```

- ```
  Ch_Initialize(&interp, option);
  while(1) {
      Ch_AppendParseScript(interp, code);
      Ch_AppendParseScriptFile(interp, filename);
      /* other API such as Ch_CallFuncByName() to access
         variables and functions in script code and filename */
  }
  Ch_End(interp);
  ```

- ```
  while(1) {
      Ch_Initialize(&interp, option);
      Ch_RunScript(interp, argv);
      /* other API such as Ch_CallFuncByName() to access
         variables and functions in script argv[0] */
      Ch_End(interp);
  }
  ```

- ```
  while(1) {
      Ch_Initialize(&interp, option);
      Ch_ParseScript(interp, argv);
      Ch_ExecScript(interp, argv[0]);
      /* other API such as Ch_CallFuncByName() to access
         variables and functions in script argv[0] */
      Ch_End(interp);
  }
  ```

In an application with multi-threads, each thread can have a separate instance of Ch interpreter. In this case, multiple Ch programs can be executed simultaneously by multiple Ch interpreters embedded in the application. Each Ch interpreter maintains its own namespace and can also be used to process multiple Ch programs as shown in Program 1.8. The maximum number of active Ch interpreters, which do not include interpreters terminated by **Ch_End**(), is decided by the number of dynamically loaded embedded Ch library **chmt#.dl** in the Windows system directory in Windows and in the directory `/usr/lib` in Unix.

Program 1.10 illustrates how to invoke multiple Ch interpreters embedded in a multi-thread application using POSIX `pthread`. In this example, three threads are created using POSIX function **pthread_create**(). The first argument of function **pthread_create**() is the thread id, the second one the attributes of the thread, the third one the thread function, and the last one the argument to the thread function. In the thread function `thread_function`(), **Ch_Initialize()** initializes an instance of Ch interpreter. The Ch code in the memory is loaded using **Ch_AppendRunScript**(). Function `func()` in the Ch space is called by **Ch_CallFuncByName**() with the value of its argument passed from the fourth argument of function **pthread_create**(). Each Ch interpreter in different threads has to be terminated by **Ch_End**() when it is no longer needed. The process waits for all three threads to complete using the function **pthread_join**(). Because these threads are executed simultaneously without a particular order, a possible output from the application is as follows.

```
1: hello
2: hello
3: hello
```

Program 1.11 is an example of multi-thread application with Embedded Ch in Windows. The logic of Program 1.11 is the same as that of Program 1.10. Instead of **pthread_create**(), Windows API **CreateThread()** is used to create three threads. These three threads are joined in the process using **WaitForSingleObject()**, instead of **pthread_join()**. Although the prototype of the thread function `thread_function()` in Program 1.11 is different from that in in Program 1.10, the contents of the function as to how to embed Ch are the same.

Note that Embedded Ch SDK APIs for the same embedded Ch can be invoked across different threads with the same first argument of `interp`. A running thread of a Ch script can be aborted in a thread by the function **Ch_Abort**() in another thread. Check Appendix A on **Ch_Abort**() and **Ch_SetGlobalUserData**() for more examples about embedding Ch in multi-thread applications across different platforms in Windows and Unix.

```
/*********************************************************************
* File Name: threads.c
* Embeddeding Ch in a multi-thread application using POSIX pthread
*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <embedch.h>
#include <pthread.h>

#define N 3
char *code = "int func(int i) { printf(\"%d: hello\\n\", i); return 0;}";

void *thread_function(void *arg){
    ChInterp_t interp;
    int val, retval;

    val = *((int*)arg);
    Ch_Initialize(&interp, NULL);
    Ch_AppendRunScript(interp, code);
    Ch_CallFuncByName(interp, "func",  &retval, val);
    Ch_End(interp);
    return NULL;
}

int main (){
    pthread_t threadId[N];
    int i, arg[N], status;

    /* Create three new threads. */
    for(i = 0; i < N; i++){
       arg[i] = i+1;
       status = pthread_create(&threadId[i], NULL, thread_function, arg+i);
       if(status != 0) {
          perror("Thread creation failed");
          exit(EXIT_FAILURE);
       }
    }

    /* Waiting for threads to finish */
    for(i = 0; i < N; i++){
       status != pthread_join(threadId[i], NULL);
       if(status != 0) {
          perror("Thread join failed");
          exit(EXIT_FAILURE);
       }
    }
    exit(EXIT_SUCCESS);
}
```

Program 1.10: A multi-thread program using POSIX pthread executing multiple embedded Ch programs using multiple instances of Ch interpreter (threads.c).

```
/*********************************************************************
* File Name: threadswin.c
* Embeddeding Ch in a multi-thread application in Windows
*********************************************************************/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <embedch.h>

#define N 3
char *code = "int func(int i) { printf(\"%d: hello\\n\", i); return 0;}";

DWORD WINAPI thread_function(PVOID arg) {
    ChInterp_t interp;
    int val, retval;

    val = *((int*)arg);
    Ch_Initialize(&interp, NULL);
    Ch_AppendRunScript(interp, code);
    Ch_CallFuncByName(interp, "func",  &retval, val);
    Ch_End(interp);
    return 0;
}

int main (){
    HANDLE thread[N];
    DWORD threadId[N];
    int i, arg[N];

    /* Create three new threads. */
    for(i = 0; i < N; i++)  {
      arg[i] = i+1;
      thread[i] = CreateThread(NULL, 0, thread_function, (PVOID)(arg+i),
                               0, &threadId[i]);
      if(thread[i] == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
      }
    }

    /* Waiting for threads to finish */
    for(i = 0; i < N; i++)  {
      if(WaitForSingleObject(thread[i], INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
      }
    }
    exit(EXIT_SUCCESS);
}
```

Program 1.11: A multi-thread program using Windows API executing multiple embedded Ch programs using multiple instances of Ch interpreter (threadswin.c).

1.6 Redirecting Standard Output and Error Stream from Embedded Ch

Sometimes, it is desirable to display error messages in pop-up Windows in some GUI applications. The simplest method to display a message from a Ch script is to use Windows API `MessageBox()` as shown below.

```
/* This Ch script will display a pop-up message in Windows */
#include <windows.h>
#define APP_NAME "MessageBox Title"

int main() {
    PCHAR string="This is a message generated from MessageBox().\n";
    MessageBox( NULL, string, APP_NAME, MB_OK | MB_SYSTEMMODAL | MB_NOFOCUS);
    return 0;
 }
```

More demo programs for Ch Windows can be found in the directory `CHHOME/toolkit/demos/Windows`.

The output from standard streams of a script can also be passed to the host application first, then displayed. In this case, functions **Ch_Reopen**(), **Ch_Close**(), and **Ch_Flush**() with the following function prototypes can be used to handle redirection of **stdin, stdout**, and **stderr** streams.

```
int      Ch_Close(ChInterp_t interp, ChFile_t fildes);
ChFile_t Ch_Reopen(ChInterp_t interp, const char *filename,
                   const char *mode, int fildes);
int      Ch_Flush(ChInterp_t interp, ChFile_t fildes);
```

where data type `ChFile_t` for file descriptor is defined in header file **embedch.h**. File descriptors **STDIN_FILENO**, **STDOUT_FILENO**, **STDERR_FILENO** corresponding to **stdin, stdout**, and **stderr** streams, respectively, defined in the header file **embedch.h**, can be used as the fourth argument of function **Ch_Reopen**(). In some platforms, function **Ch_Reopen**() need to be called before any I/O happens. It is best to call function **Ch_Reopen**() right after **Ch_Initialize**() is called as shown in in Program 1.12.

In Program 1.12, the argument `argvv` of function **Ch_ParseScript**() for program name `reopen.ch` without command line option is created dynamically. Before script `reopen.ch` in Program 1.13, is parsed by function **Ch_ParseScript**() in Embedded Ch, the **stderr** stream is redirected to a temporary file `stderrfile` by function call

```
fildes_stderr = Ch_Reopen(interp, stderrfile, "w", STDERR_FILENO);
```

which is equivalent to

```
FILE* stream = freopen(stderrfile, "w", stderr);
```

Before the error message generated by function call of **Ch_ParseScript**() can be retrieved from file `stderrfile`, the file descriptor for the redirected **stderr** stream has to be closed by function call

```
Ch_Close(interp, fildes_stderr);
```

which is equivalent to

```
fclose(stream);
```

The argument of function printf() in script `reopen.ch` shown in Program 1.13 is missing a closing double quote. When Program 1.12 is executed in a Unix machine, it will display the following messages in the console.

```
  Error: parse program reopen.ch failed.
  ERROR: missing "
  ERROR: argument 1 of the function is undefined or not a valid expression
  ERROR: syntax error before or at line 6 in file 'reopen.ch'
    ==>:   return 0;
    BUG:   return<== ???
```

The contents of the temporary file `/var/tmp/aaax1aa6M` are displayed by function `displayContentsInFile()` in Program 1.12.

```
/* File Name: reopen.c
*  redirect stderr from Embedded Ch to different file. */
#include <embedch.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Display contents in a file */
void displayContentsInFile(char *filename) {
   FILE *stream;
   char buffer[BUFSIZ];
   int length;

   stream = fopen(filename, "r");
   if(stream != NULL) {
     length = fread(buffer, 1, BUFSIZ, stream);
     buffer[length] = '\0';
     if(length > 0)
        printf("%s\n", buffer);
        /* use MessageBox(NULL, buffer, "Message title", 0 ); for GUI in Windows */
     fclose(stream);
   }
}

int main() {
  ChInterp_t interp;
  int status;
  char **argvv, *program = "reopen.ch", stderrfile[L_tmpnam];
  ChFile_t fildes_stderr;

  Ch_Initialize(&interp, NULL);
  tmpnam(stderrfile);  /* create a temporary file name */
  fildes_stderr = Ch_Reopen(interp, stderrfile, "w", STDERR_FILENO);
  if(fildes_stderr == CH_ERROR)
     printf("Error: cannot redirect stderr error stream\n");
  /* create char *argvv[] = {"reopen.ch", NULL} dynamically */
  argvv = (char **)malloc(2*sizeof(char *));
  argvv[0] = strdup(program);
  argvv[1] = NULL;
  status = Ch_ParseScript(interp, argvv);
  if(status == CH_ERROR) {
     printf("Error: parse program %s failed.\n", program);
     Ch_Close(interp, fildes_stderr);
     displayContentsInFile(stderrfile);
  }
  else {
    status = Ch_ExecScript(interp, program); /* execute reopen.ch */
    Ch_Close(interp, fildes_stderr);
    if(status == CH_ERROR) {
      printf("Error: execute script %s failed\n", program);
      displayContentsInFile(stderrfile);
    }
  }
  remove(stderrfile); /* remove temporary file for stderr */
  free(argvv[0]);     /* release memory and terminate the shell */
  free(argvv);
  Ch_End(interp);
}
```

Program 1.12: Redirecting stderr to a file.

```
#include <stdio.h>

int main() {
  //printf("hello, good\n");
  printf("hello, missing quote\n);
  return 0;
}
```

Program 1.13: Script with an error for testing stderr redirection.

Note that you can capture the error messages from Ch Embedded Ch API's by redirecting them using **Ch_Reopen**(), except for the error message from the API **Ch_Initialize**(). In this case, you can create a console application based on one of demos in the directory CHHOME/toolkit/demos/embedch. guide. Then, you should see the error message from **Ch_Initialize**().

1.7 Distributing Embedded Ch in Your Applications

If the second argument of function call ChInitialize() is NULL, such as

```
Ch_Initialize(interp, NULL);
```

The `CHHOME/toolkit/embedch` is used as the home directory of Embedded Ch. This is the case for the development of applications with Embedded Ch. In this case, the startup file `CHHOME/toolkit/embedch/config/chrc` will be executed in this initialization function. Embedded Ch searches for the dynamically loaded libraries such as `chmt1.dl` and `chmt2.dl` in the directory CHHOME/toolkit/embedch.

To distribute your application with Embedded Ch, after obtaining the authorization, the Embedded Ch located in `CHHOME/toolkit/embedch` can be bundled with your application program for distribution. This directory contains the Ch run time environment. Typically, it will be placed under the home directory of your application such as `C:/myApplication/embedch`. Your application program should set the Embedded Ch home directory properly as shown in example in Program 1.5. A sample code for distribution of Embedded Ch without regular Ch is given in Program 1.14.

To distribute your application with Embedded Ch, but without regular Ch, follow the steps below.

1. If your application is distributed in the directory `C:/myApplication` in a target machine. Copy and distribute the contents in the directory `CHHOME/toolkit/embedch` of the Embedded Ch in your development machine as `C:/myApplication/embedch` in the target machine.

2. In Windows, make sure applications will be able to access file `ch.dll`. In general, the dynamically linked library `ch.dll` for Embedded Ch shall be located at the same directory which contains the hosting application with Embedded Ch. For example, if the binary application is located in `C:/myApplication/bin`, copy `CHHOME/toolkit/embedch/bin/ch.dll` to `C:/myApplication/bin/ch.dll`. If the binary hosting application with Embedded Ch is located in the Windows system directory such as `C:/Windows/System32`, it is not recommended to put ch.dll over there. Instead, the environment variable PATH can be set with the directory `C:/myApplication/bin` which contains `ch.dll`. This will reduce the chance of the conflict since the installation of regular standardalone Ch puts ch.dll in the Windows system directory.

3. The second argument of function call ChInitialize() should not be NULL, such as

```
/* We assume that your application is distributed
   in /usr/local/companyname in Unix or C:/myApplication in Windows.
   The Embedded Ch runtime environment should be copied from
   CHHOME/toolkit/embedch in your development machine and distributed
   at /usr/local/companyname/embedch in Unix and
   C:/myApplication/embedch in Windows in a target machine. Then
   Embedded Ch home directory will be /usr/local/companyname/embedch
   in Unix and C:/myApplication/embedch in Windows. */

#include <embedch.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
  int status, len;
  ChInterp_t interp;
  ChOptions_t option;
  char *proghome, *p;

  /* get the home dir for your application program from
     an environment variable or registry value PROG_HOME which
     was setup during the installation of your application in a
     target machine. The Embedded Ch home directory will be
     PROG_HOME/embedch which needs be setup using option.chhome
     as follows. */

  /* set option.shelltype */
  option.shelltype = CH_REGULARCH;

  /* set option.chhome */
  proghome = getenv("PROG_HOME"); /* from an environment variable */
  /* or obtain the value for proghome from a registry value in Windows,
     using Windows API RegOpenKeyEx() and RegQueryValueEx(). */
  len = strlen(proghome)+ strlen("/embedch")+1;
  option.chhome = (char *)malloc(len);
  strcpy(option.chhome, proghome);
  strcat(option.chhome, "/embedch");

  Ch_Initialize(&interp, &option);
  /* ... */
  Ch_End(interp);
  free(option.chhome);
}
```

Program 1.14: A sample code for distribution of Embedded Ch without regular Ch.

```
Ch_Initialize(interp, &option);
```

as shown in Program 1.14. The startup file `option.chhome/config/chrc` or `C:/myApplication/embedch/config/chrc` will be executed in this initialization function. Embedded Ch searches for dynamically loaded libraries such as `chmt1.dl` and `chmt2.dl` in the directory `option.chhome/extern/lib`.

Multiple Ch interpreters can be embedded in an application. Each instance of Ch interpreter is dynamically loaded. The dynamically loaded library `chmt1.dl` is for the first instance of Ch interpreter, `chmt2.dl` for the second one, and `chmt`n`.dl` for the nth one. Although these dynamically loaded libraries are exactly the same, different copies are needed if your application has multi-threads. For a multi-thread application which needs n instances of Ch interpreter, copy `CHHOME/toolkit/embedch/extern/lib/chmt1.dl` to `chmt2.dl`, `chmt3.dl`, and `chmt`n`.dl` in `option.chhome/extern/lib`. The maximum number of simultaneous instances of Ch interpreter is 500.

4. Like regular Ch shell, a Ch script invoked by Embedded Ch has its own search pathes for commands, include header files, and function files specified by the systems variables **_path, _ipath,** and _fpath, respectively. The value for the system variable **_path** is also copied to the environment variable `PATH`. The pathes specified by the environment variable `PATH` are searched for dynamically loaded libraries in Windows. If your application needs some dynamically loaded libraries which are not located in the default directories specified by the system variable **_path**, make sure they are added to **_path** in the system startup file EMBEDCHHOME/config/chrc or individual user's startup file _chrc in the user home directory. If some dynamically loaded libraries are located in `C:/myApplication/bin`, add the statement below in the startup file.

   ```
   _path = stradd(_fpath, "C:/myApplication/bin;");
   ```

5. Function files with extension `.chf`, used in Ch scripts, can be distributed along with your application. You may put these function files in a separate directory under your application home directory such as `C:/myApplication/ch/lib.` Then add the directory in the system variable **_fpath** in file `config/chrc` under the home directory of Embedded Ch as follows.

   ```
   _fpath = stradd(_fpath, "C:/myApplication/ch/lib;");
   ```

   Similarly, the search paths `C:/myApplication/ch/include` and `C:/myApplication/ch/dl` for header files and dynamically loaded libraries with file extension .dl in Ch, respectively, can be added as follows.

   ```
   _ipath = stradd(_ipath, "C:/myApplication/ch/include;");
   _lpath = stradd(_lpath, "C:/myApplication/ch/dl;");
   ```

6. Finally, your application must include the following language in the copyright section or in other suitable place such as the `"About"` box of the product:

   Contains Ch$^{(R)}$, an embeddable C/C++ interpreter developed by SoftIntegration, Inc., http://www.softintegration.com. All Rights Reserved.

1.7.1 Minimizing the Distribution of Runtime Components

For many applications, only a subset of Ch is needed. In this case, some of components in Embedded Ch can be removed in your distribution. More specifically, the following modules are removable.

- If your application is single thread and needs only one instance of the interpreter, remove all `chmt#.dl` files except `chmt1.dl` in the directory `CHHOME/toolkit/embedch/extern/lib`.
- Unix utilities such as ls.exe, grep.exe, sed.exe, etc. for Windows in the directory `option.home/toolkit/embedch/bin.`
- OpenGL, GTK+, Win32, X11/Motif toolkits in the directory `option.home/toolkit/embedch/toolkit`.

For applications using only expressions and built-in mathematical functions listed in the header file **math.h** in Embedded Ch, files in `option.home/toolkit/embedch` are not needed. In this case, however, scripts processed by Embedded Ch shall not contain function files and header files. The member field `option.chhome` for the embedded Ch home directory can be set to the current working directory `"."`. In Unix, no additional file is needed.

Chapter 2

Accessing Ch Variables from C Space

In the previous chapter, we introduced the basic procedure and core APIs for embedding Ch. After a Ch program is executed by a C/C++ program, users may want to access and modify the value of variables, or evaluate expressions in the Ch program. We will describe APIs which can be used to access variables, assign values of variables, and evaluate expressions in the Ch space from a C/C++ program in the C space.

2.1 Accessing Variables by Addresses

The basic method to access a variable in Ch program typically consists of two steps:

1. Getting the address of a variable in Ch space within its scope by API **Ch_SymbolAddrByName**() with its function prototype in header file **ch.h** as follows.

```
void*  Ch_SymbolAddrByName(ChInterp_t interp, char *name);
```

The first argument of function **Ch_SymbolAddrByName**() is an instance of Ch interpreter. The second argument is the name of the symbol whose address in Ch space is obtained. The function returns the address obtained on successful execution or a NULL pointer on failure. Function **Ch_SymbolAddrByName**() can obtain the address of both local and global variables. Function **Ch_GlobalSymbolAddrByName**() can be used to obtain the address of a global variable when the interpreter is executing a function as shown in an example on page 278. More information about **Ch_GlobalSymbolAddrByName**() can be found in *Ch SDK User's Guide*. Other functions with the prefix **Ch_GlobalSymbol** can be found in the reference described Appendix A.

2. Accessing the variable by its address.

C program `accesschvar.c` in Program 2.1 is an example of accessing global variables in a Ch program. After function **Ch_RunScript**() is called, only global variables in the Ch space can be accessed. The Ch program `accesschvar.ch` shown in Program 2.2 has two global variables, `ii` with type int and `dd` with type double. The data type of a variable in the Ch space can be obtained by the API **Ch_DataType**(), which returns a value for a valid Ch type defined in the header file **ch.h**. Accordingly, two variables `pii` with pointer to int and `pdd` with pointer to double are declared in `accesschvar.c` for keeping addresses of those two Ch variables. The size of a variable or expression can be obtained by the function **Ch_DataSize**(). After the embedded Ch is initialized, Program 2.2 is loaded and executed. Statements below

```
pii = Ch_SymbolAddrByName(interp,"ii");
pdd = Ch_SymbolAddrByName(interp,"dd");
```

obtain addresses of variables `ii` and `dd` in Ch space. Then the statement

```
/*************************************************************
* File Name: accesschvar.c
* Accessing global variables in the Ch program
*************************************************************/
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"accesschvar.ch", NULL};
  int *pii;
  double *pdd;

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program accesschvar.ch failed\n");

  if(Ch_DataType(interp, "ii") == CH_INTTYPE)
    printf("ii is int\n");
  else
    printf("ii is not int\n");


  if(Ch_DataType(interp, "dd") == CH_DOUBLETYPE)
    printf("dd is double\n");
  else
    printf("dd is not double\n");

  printf("Ch_DataSize(interp, \"dd\") = %d\n", Ch_DataSize(interp, "dd"));

  pii = Ch_SymbolAddrByName(interp,"ii");
  pdd = Ch_SymbolAddrByName(interp,"dd");
  printf("In C, before assignments, ii = %d, dd = %f\n", *pii, *pdd);
  *pii = 20;
  *pdd = 2.2;
  printf("In C, after assignments, ii = %d, dd = %f\n", *pii, *pdd);

  Ch_End(interp);
}
```

Program 2.1: Accessing variables in a Ch program (accesschvar.c).

```
int ii = 10;
double dd = 1.1;
printf("printed from accesschvar.ch, ii = %d, dd = %f\n", ii, dd);
```

Program 2.2: Accessing variables in a Ch program (accesschvar.ch).

```
printed from accesschvar.ch, ii = 10, dd = 1.100000
ii is int
dd is double
Ch_DataSize(interp, "dd") = 8
In C, before assignments, ii = 10, dd = 1.100000
In C, after assignments, ii = 20, dd = 2.200000
```

Figure 2.1: Output of executable compiling from `accesschvar.c`.

```
int i = 10;
double d = 1.1;
int prototype();
extern int e;
struct tag {
  int j;
} s;
int func() {
    int localvariable =20;
    printf("printed from symbol.ch, i = %d, d = %f\n", i, d);
    return 0;
}
int main() {
    func();
    return 0;
}
```

Program 2.3: A Ch program for testing its symbol table (symbol.ch).

```
printf("In C, before assignments, ii = %d, dd = %f\n", *pii, *pdd);
```

prints out values in these two addresses. Afterwards, assignment statements below

```
*pii = 20;
*pdd = 2.2;
```

change values of `ii` and `dd` in the Ch program. The output from executing the program built from Program 2.1 is shown in Figure 2.1.

Symbols in header files including system header files are treated the same as in a user's program. All symbols in the symbol table for global variables in an embedded Ch program can be accessible by the hosting program after the Ch program is loaded by **Ch_RunScript**(), **Ch_ParseScript**(), or other functions. Program 2.3 has seven symbols `i, d, prototype, e, s, func`, and `main`. The function prototype `prototype`, and user defined functions `func` and `main` are included in the symbol table. The symbol table does not contain generic functions `printf()`, system functions, tag name for a structure/class/union definition and local variables of automatic duration not in its scope. The information about Program 2.3 is processed by Program 2.4. The output of Program 2.4 is displayed in Figure 2.2. The total number eight of all symbols in the symbol table is obtained by function **Ch_SymbolTotalNum**(). The variable `d` is the second symbol in the symbol table as shown in the return value from function **Ch_SymbolIndexByName**(). Note that the index number for a symbol table starts with 0. The address of a variable can be obtained by function **Ch_SymbolAddrByName**() or **Ch_SymbolAddrByIndex**() based on its name or index number in the symbol table, respectively. The address for a function prototype without function definition or an extern variable without definition such as `prototype` and `e`, respectively, in Program 2.3 is also NULL. The symbol name corresponding to its index number in the symbol table can be obtained by function **Ch_SymbolNameByIndex**().

```
/*************************************************************
* File Name: symbol.c
*************************************************************/
#include<stdio.h>
#include<embedch.h>

int numOfVar(ChInterp_t interp) {
  int index, totalnum, varnum=0;
  char *name;

  totalnum = Ch_SymbolTotalNum(interp);
  for(index=0; index<totalnum;index++){
    name = Ch_SymbolNameByIndex(interp, index);
    if(!Ch_FuncType(interp, name))
      varnum++;
  }
  if(!Ch_VarType(interp, "localvariable"))
    printf("localvariable is not CH_LOCALVARTYPE or CH_GLOBALVARTYPE\n");
  if(Ch_VarType(interp, "func") == CH_GLOBALVARTYPE)
    printf("func is CH_GLOBALVARTYPE\n");
  return varnum;
}

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"symbol.ch", NULL};
  int index, totalnum;

  Ch_Initialize(&interp, NULL);
  status = Ch_RunScript(interp,argvv);
  /* or status = Ch_ParseScript(interp,argvv); */
  if(status == CH_ERROR)
    printf("Error: execution of program symbol.ch failed\n");

  index = Ch_SymbolIndexByName(interp, "d");
  printf("symbol index for 'd' is %d\n", index);
  index = Ch_SymbolIndexByName(interp, "unknown");
  printf("symbol index for 'unknown' is %d\n", index);
  totalnum = Ch_SymbolTotalNum(interp);
  printf("The total number of symbols is %d\n", totalnum);
  for(index=0; index<totalnum;index++){
    printf("symbol index %d, name %s, address %p\n", index,
            Ch_SymbolNameByIndex(interp, index),
            Ch_SymbolAddrByIndex(interp, index));
  }
  printf("Ch_SymbolAddrByName(interp, \"main\") = %p\n",
          Ch_SymbolAddrByName(interp, "main"));
  printf("Number of global variables of not function type is %d\n", numOfVar(interp));

  Ch_End(interp);
}
```

Program 2.4: Accessing symbols in the symbol table in a Ch program (symbol.c).

```
printed from symbol.ch, i = 10, d = 1.100000
symbol index for 'd' is 1
symbol index for 'unknown' is -1
The total number of symbols is 7
symbol index 0, name i, address 40a60
symbol index 1, name d, address 57b00
symbol index 2, name prototype, address 0
symbol index 3, name e, address 0
symbol index 4, name s, address 53628
symbol index 5, name func, address 54cd0
symbol index 6, name main, address 55180
Ch_SymbolAddrByName(interp, "main") = 55180
localvariable is not CH_LOCALVARTYPE or CH_GLOBALVARTYPE
func is CH_GLOBALVARTYPE
Number of global variables of not function type is 4
```

Figure 2.2: Output of the executable program from `symbol.c`.

The information about variables can be obtained by other functions. The data type, array element data type, and function return type of a variable can be obtained by function **Ch_DataType**(). If a variable is an array, its array type, dimension, and extent for each dimension can be obtained by functions **Ch_ArrayDim**(), **Ch_ArrayExtent**(), and **Ch_ArrayType**(), respectively. Whether a variable is function, function with variable number of arguments, and the number of arguments can be obtained by functions **Ch_FuncType**(), **Ch_IsFuncVarArg**(), and **Ch_FuncArgNum**(), respectively. **Ch_FuncType**() can be used to test whether a symbol is function with definition, function prototype, pointer to function, member function, constructor, destructor of a class. The symbol table contains global and local variables. Function **Ch_VarType**() can be used to check if a symbol is a variable, global variable or local variable as shown in Program 2.4. Function `numOfVar()` returns the number of variables of not function type. Program 2.3 has four variables `i, d, e`, and `s` of not function type. The tag name, size, and total number of members of a user defined class, structure, or union can be obtained by functions **Ch_UserDefinedInfo**(). The information for a member of of a user defined class, structure, or union can be obtained by functions **Ch_UserDefinedMemInfoByIndex**() or **Ch_UserDefinedMemInfoByIndex**().

2.2 Assigning Values in C Space to Variables in Ch Space

Values in the C space can be assigned to variables in the Ch space by function **Ch_SetVar**() with the following function prototype.

```
int  Ch_SetVar(ChInterp_t interp, const char *name, ChType_t atype, ...);
```

For a simple data type, the function will be called as if it was prototyped as follows.

```
int  Ch_SetVar(ChInterp_t interp, const char *name, ChType_t atype, type value);
```

where `name` is an identifier in the Ch space, `atype` is the data type **ChType_t** of the value in the C space defined in header file **CHHOME/extern/include/ch.h**, the last argument `value` is an expression or a variable in the C space. It can be any valid data type in C such as int, double, pointer to int, array, structure, etc. By assigning the address of a variable or data in the C space to a variable of pointer type in the Ch space, the memory in the C space can be shared by the variables in the Ch space.

For example, Program 2.5 declares three variables `ch_i` of int, `ch_p` of pointer to int, and `ch_data` of pointer to int in the Ch space by function **Ch_AppendRunScript**(). The value of the variable `c_i` in the C space and its address are assigned to variables `ch_i` and `ch_p`, respectively, by function calls

```
/* File Name: setvar.c */
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    int status;
    int c_i, c_data[10] = {1,2,3,4,5,6,7,8,9,10};

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);
    /*  declare variables i, p, ch_data in Ch space */
    Ch_AppendRunScript(interp,"int ch_i, *ch_p, *ch_data;");
    c_i = 10;
    /* assign c_i in C space to ch_i in Ch space */
    Ch_SetVar(interp, "ch_i", CH_INTTYPE, c_i);
    Ch_ExprEval(interp, "printf(\"ch_i = %d\n\",ch_i)");
    /* assign &c_i in C space to ch_p in Ch space */
    Ch_SetVar(interp, "ch_p", CH_INTPTRTYPE, &c_i);
    c_i = 20;
    Ch_ExprEval(interp, "printf(\"ch_i = %d\n\",ch_i)");
    Ch_ExprEval(interp, "printf(\"*ch_p = %d\n\",*ch_p)");
    Ch_ExprEval(interp, "*ch_p = 30");
    printf("c_i = %d\n",c_i);
    /* assign &c_data in C space to ch_data in Ch space */
    Ch_SetVar(interp, "ch_data", CH_INTPTRTYPE, &c_data);
    Ch_ExprEval(interp, "printf(\"ch_data[9] = %d\n\",ch_data[9])");
    Ch_End(interp);
}
```

Program 2.5: Assigning values in C space to variables in Ch space (setvar.c).

```
Ch_SetVar(interp, "ch_i", CH_INTTYPE, c_i);
Ch_SetVar(interp, "ch_p", CH_INTPTRTYPE, &c_i);
```

The value for `c_i` in the C space is later changed to 30 indirectly through the indirection operator in the Ch space by function call

```
Ch_ExprEval(interp, "*ch_p = 30");
```

The array `c_data` in the C space is shared in the Ch space by the variable `ch_data`. The output from executing Program 2.5 is shown in Figure 2.3.

Variables in the Ch space in Program 2.5 are created dynamically by function **Ch_AppendRunScript**(). Variables in a Ch script can also access values in the C space as shown in Program 2.6. Program 2.6 is similar to Program 2.5. Program 2.6 invokes embedded Ch program `setvar.ch` in Program 2.7. Variable `c_i` and its address are assigned to variables `ch_i` and `ch_p` in Program 2.7. Array `c_data` in the C space is shared by variable `ch_data` in Program 2.7. Function `chfun()` in the Ch space is called by

```
Ch_CallFuncByName(interp, "chfun", NULL);
```

in the C space, which changes the values for elements `c_data[0]` and `c_data[9]` in the C space. Details about calling a Ch function using the API **Ch_CallFuncByName**() will be described in detail Chapter 3. The output from executing Program 2.6 is shown in Figure 2.4.

```
ch_i = 10
ch_i = 10
*ch_p = 20
c_i = 30
ch_data[9] = 10
```

Figure 2.3: Output from executing `setvar.c`.

2.3 Obtaining information for a Variable of Structure, Class, and Union

Ch provides the following four APIs to find information for user defined variables of structure, class, and union types.

```
ChUserDefinedTag_t Ch_UserDefinedTag(ChInterp_t, const char *name);
int Ch_UserDefinedInfo(ChInterp_t interp, ChUserDefinedTag_t udtag,
                       ChUserDefinedInfo_t *udinfo);
int Ch_UserDefinedMemInfoByName(ChInterp_t interp,
ChUserDefinedTag_t udtag, const char *memname, ChMemInfo_t *meminfo);
int Ch_UserDefinedMemInfoByIndex(ChInterp_t interp,
    ChUserDefinedTag_t udtag, int index, ChMemInfo_t *meminfo);
```

A user defined structure, class and union has a tag. This tag can be obtained by the function **Ch_UserDefinedTag**() based on the name of variable or tag name. The returned tag from function **Ch_UserDefinedTag**() is passed as the second argument of function **Ch_UserDefinedInfo**(). The information for the user defined data type is passed back to the calling function in the third argument of function **Ch_UserDefinedInfo**(). The **ChUserDefinedInfo_t** structure contains three members for the tag name, size and total number of members of the user defined type. The information for each member of a variable of the user defined type can be obtained by functions **Ch_UserDefinedMemInfoByName**() and **Ch_UserDefinedMemInfoByIndex**() based on the member name and index of the member in the symbol table for the user defined type, respectively.

As an example, Program 2.8 contains a user defined structure `tag` with four members `i, d, p`, and `a`. Three variables `s, sp`, and `sa` of structure, pointer to structure, and array of structure are declared. For comparison, a global variable `i` int type using the same name of a member of structure is also declared in the program.

Program 2.9 first executes the script in Program 2.8 by function **Ch_RunScript**(). It then obtain the information for structure `tag` and its members by function `processUserDefined()`. The dimension of array for variable `sa` is obtained by **Ch_ArrayExtent**(). Like variables of other type, the address for a variable of structure `tag` can be obtained by function **Ch_SymbolAddrByName**(). The tag for structure is assigned to the variable `udtag`. The information for the variable `s` is kept in the variable `udinfo`, whereas the variable `meminfo` contains the information for a member of variable `s`. Function `printMemInfo()` prints out the information for a member. Using the offset for each member of structure `tag`, the address for each member of a variable of structure `tag` can be obtained by the expression

```
addr + meminfo.offset
```

The output from executing Program 2.9 is shown in Figure 2.5.

```
/* File Name: setvar2.c */
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    int status;
    int c_i, c_data[10] = {1,2,3,4,5,6,7,8,9,10};
    char *argvv[]={"setvar.ch", NULL};

    /* initialize embedded Ch */
     Ch_Initialize(&interp, NULL);
    /* run a Ch function file */
    status = Ch_RunScript(interp,argvv);
    if(status == CH_ERROR)
      printf("Error: execution of program setvar.ch.ch failed\n");

    c_i = 10;
    Ch_SetVar(interp, "ch_i", CH_INTTYPE, c_i);
    Ch_ExprEval(interp, "printf(\"ch_i = %d\n\",ch_i)");
    Ch_SetVar(interp, "ch_p", CH_INTPTRTYPE, &c_i);
    c_i = 20;
    Ch_ExprEval(interp, "printf(\"ch_i = %d\n\",ch_i)");
    Ch_ExprEval(interp, "printf(\"*ch_p = %d\n\",*ch_p)");
    Ch_ExprEval(interp, "*ch_p = 30");
    printf("c_i = %d\n",c_i);
    Ch_SetVar(interp, "ch_data", CH_INTPTRTYPE, &c_data);
    Ch_ExprEval(interp, "printf(\"ch_data[9] = %d\n\",ch_data[9])");
    Ch_CallFuncByName(interp, "chfun", NULL);
    Ch_ExprEval(interp, "printf(\"ch_data[9] = %d\n\",ch_data[9])");
    Ch_End(interp);
}
```

Program 2.6: Assigning values in C space to variables in a script in Ch space (setvar2.c).

```
#include <stdio.h>

int ch_i, *ch_p, *ch_data;
void  chfun() {
  printf("in the Ch function chfun(), ch_data[0] = %d, ch_data[9] = %d\n",
          ch_data[0], ch_data[9]);
  ch_data[0]++;
  ch_data[9]++;
}
```

Program 2.7: Ch script with values assigned in C space (setvar.ch).

```
ch_i = 10
ch_i = 10
*ch_p = 20
c_i = 30
ch_data[9] = 10
in the Ch function chfun(), ch_data[0] = 1, ch_data[9] = 10
ch_data[9] = 11
```

Figure 2.4: Output from executing `setvar2.c`.

```
struct tag {int i; double d; int *p; int a[2][3];};
struct tag s = {10, 20, NULL, 30}, *sp, sa[2]={10, 20, NULL, 30};
int i;
int main () {
   s.p = &i;
   sp = &s;
   printf("&s = %p\n", &s);
   printf("&s.i = %p\n", &s.i);
   printf("&sp = %p\n", &sp);
   printf("&sp->i = %p\n", &sp->i);
   return 0;
}
```

Program 2.8: Ch program with user defined data type (userdefinedsymbol.ch).

2.4 Evaluating Expressions

Ch provides API **Ch_ExprEval**() to evaluate expressions in a Ch program. This function prototype is given in header file **embedch.h** as follows.

```
int  Ch_ExprEval(ChInterp_t interp, char *expr);
```

The first argument of this function is an instance of Ch interpreter. The second argument is the string of the expression to be evaluated in Ch space. The function returns zero on successful execution, non-zero on failure.

C program `expreval.c` shown in Program 2.10 is an example of evaluating expressions in a Ch program. The Ch program executed by embedded Ch is shown in Program 2.11. Two variables, `ii` with type int and `dd` with type double, are assigned with original values of `10` and `1.1`, respectively. The pointer `p` is pointed to the address of the variable `ii`. The array `a` and structure `s` are also declared and initialized. Figure 2.7 illustrates how to evaluate expressions in the Ch program with embedded Ch. In C program `expreval.c`, after the embedded Ch shell is initialized, the Ch program is loaded and executed. Statements below

```
pii = Ch_SymbolAddrByName(interp, "ii");
pdd = Ch_SymbolAddrByName(interp, "dd");
p2 = Ch_SymbolAddrByName(interp, "p");
pa = Ch_SymbolAddrByName(interp, "a");
sp = Ch_SymbolAddrByName(interp, "s");
```

obtain the addresses of the Ch global variables `ii, dd, p, a`, and `s` of int, double, pointer to int, array of int, and structure types, respectively. The function calls

```
Ch_ExprEval(interp, "ii++");
Ch_ExprEval(interp, "dd = dd + ii + 2*(1+pow(2.0,3))");
Ch_ExprEval(interp, "func()");
```

evaluate expressions

```
ii++
dd = dd + ii + 2*(1+pow(2.0,3))
func()
```

```
/*****************************************************************
* File Name: userdefinedsymbol.c
*****************************************************************/
#include<stdio.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"userdefinedsymbol.ch", NULL};

  Ch_Initialize(&interp, NULL);
  status = Ch_RunScript(interp,argvv);
  processUserDefined(interp, "s");
/*
  processUserDefined(interp, "sp");
  processUserDefined(interp, "sa");
*/
  Ch_End(interp);
  return 0;
}

int processUserDefined(ChInterp_t interp, const char *name) {
  int i, n = 1, index, totalnum;
  char *addr;
  ChUserDefinedTag_t udtag;
  ChUserDefinedInfo_t udinfo;
  ChMemInfo_t meminfo;

  if(Ch_ArrayType(interp, name) &&
     (Ch_DataType(interp, name) == CH_STRUCTTYPE ||
     Ch_DataType(interp, name) == CH_CLASSTYPE))
  {
      n = Ch_ArrayExtent(interp, name, 0); /* assume it is 1-dim array */
  }

  addr = Ch_SymbolAddrByName(interp, name);
  if(Ch_DataType(interp, name) == CH_STRUCTPTRTYPE ||
     Ch_DataType(interp, name) == CH_CLASSPTRTYPE)
  {
     addr = *(void **)addr;
  }
  printf("The address of the variable %p\n", addr);


  udtag = Ch_UserDefinedTag(interp, name);
  printf("The user defined tag is %p\n", udtag);
  Ch_UserDefinedInfo(interp, udtag, &udinfo);
  printf("udinfo.dtype == CH_STRUCTTYPE = %d\n", udinfo.dtype == CH_STRUCTTYPE);
  printf("The size of the struct/class/union is %d\n", udinfo.size);
  printf("The tag name of the struct/class/union is %s\n", udinfo.tagname);
  printf("The total number of members is %d\n", udinfo.totnum);
  for(i=0; i<n; i++) {
    for(index=0; index<udinfo.totnum;index++){
      Ch_UserDefinedMemInfoByIndex(interp, udtag, index, &meminfo);
      printMemInfo(&meminfo, addr+i*udinfo.size);
    }
  }
```

Program 2.9: C program for obtaining information for variables of user defined type (userdefinedsymbol.c).

```
  for(index=0; index<udinfo.totnum;index++){
      Ch_UserDefinedMemInfoByIndex(interp, udtag, index, &meminfo);
      printf("meminfo.offset = %d\n", meminfo.offset);
  }

  Ch_UserDefinedMemInfoByName(interp, udtag, "i", &meminfo);
  printf("i = %d\n", *(int *)(addr + meminfo.offset));
  Ch_UserDefinedMemInfoByName(interp, udtag, "d", &meminfo);
  printf("d = %f\n", *(double*)(addr + meminfo.offset));
  Ch_UserDefinedMemInfoByName(interp, udtag, "p", &meminfo);
  printf("p = %p\n", (int*)(addr + meminfo.offset));
  Ch_UserDefinedMemInfoByName(interp, udtag, "a", &meminfo);
  printf("a[0] = %d\n", *(int*)(addr + meminfo.offset));
  return 0;
}

int printMemInfo(ChMemInfo_t *meminfo, void *addr) {
    printf("meminfo->index = %d\n", meminfo->index);
    printf("meminfo->memname = %s\n", meminfo->memname);
    printf("meminfo->offset = %d\n", meminfo->offset);
    if(meminfo->dtype == CH_INTTYPE)
      printf("meminfo->dtype == CH_INTTYPE = 1\n");
    else if(meminfo->dtype == CH_DOUBLETYPE)
      printf("meminfo->dtype == CH_DOUBLETYPE = 1\n");
    else if(meminfo->dtype == CH_INTPTRTYPE)
      printf("meminfo->dtype == CH_INTPTRTYPE = 1\n");
    printf("meminfo->ispublic = %d\n", meminfo->ispublic);
    if(meminfo->isfunc)
       printf("meminfo->isfunc= %d\n", meminfo->isfunc);
    if(meminfo->isconstructor)
       printf("meminfo->isconstructor = %d\n", meminfo->isconstructor);
    if(meminfo->isdestructor)
       printf("meminfo->isdestructor = %d\n", meminfo->isdestructor);
    if(meminfo->isvararg)
       printf("meminfo->isvararg = %d\n", meminfo->isvararg);
    if(meminfo->arraytype) {
       printf("meminfo->arraytype = %d\n", meminfo->arraytype);
       printf("meminfo->dim = %d\n", meminfo->dim);
       printf("meminfo->externt[0] = %d\n", meminfo->extent[0]);
       printf("meminfo->externt[1] = %d\n", meminfo->extent[1]);
    }
    if(meminfo->isbitfield) {
       printf("meminfo->isbitfield = %d\n", meminfo->isbitfield);
       printf("meminfo->fieldsize = %d\n", meminfo->fieldsize);
       printf("meminfo->fieldoffset= %d\n", meminfo->fieldoffset);
    }
    if(meminfo->dtype == CH_STRUCTTYPE ||
       meminfo->dtype == CH_CLASSTYPE ||
       meminfo->dtype == CH_UNIONTYPE)
       printf("meminfo->udtag= %p\n", meminfo->udtag);
    printf("\n");
    return 0;
}
```

Program 2.9: C program for obtaining information for variables of user defined type (userdefinedsymbol.c) (Contd.).

```
&s = 50090
&s.i = 50090
&sp = 50110
&sp->i = 50090
The address of the variable 50090
The user defined tag is 59940
udinfo.dtype == CH_STRUCTTYPE = 1
The size of the struct/class/union is 48
The tag name of the struct/class/union is tag
The total number of members is 4
meminfo->index = 0
meminfo->memname = i
meminfo->offset = 0
meminfo->dtype == CH_INTTYPE = 1
meminfo->ispublic = 1

meminfo->index = 1
meminfo->memname = d
meminfo->offset = 8
meminfo->dtype == CH_DOUBLETYPE = 1
meminfo->ispublic = 1

meminfo->index = 2
meminfo->memname = p
meminfo->offset = 16
meminfo->dtype == CH_INTPTRTYPE = 1
meminfo->ispublic = 1

meminfo->index = 3
meminfo->memname = a
meminfo->offset = 20
meminfo->dtype == CH_INTTYPE = 1
meminfo->ispublic = 1
meminfo->arraytype = 80
meminfo->dim = 2
meminfo->externt[0] = 2
meminfo->externt[1] = 3

meminfo.offset = 0
meminfo.offset = 8
meminfo.offset = 16
meminfo.offset = 20
i = 10
d = 20.000000
p = 500a0
a[0] = 30
```

Figure 2.5: Output from executing `userdefinedsymbol.c`.

```
/*****************************************************************
* File Name: expreval.c
* Evaluating expression in Ch space
*****************************************************************/
#include<stdio.h>
#include<string.h>
#include<embedch.h>

struct tag {
  int i;
  double f;
} *sp;
int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"expreval.ch", NULL};
  int *pii, **p2, *pa;
  double *pdd;

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp,argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program expreval.ch failed\n");

  pii = Ch_SymbolAddrByName(interp,"ii");
  pdd = Ch_SymbolAddrByName(interp,"dd");
  p2 = Ch_SymbolAddrByName(interp,"p");
  pa = (int *)Ch_SymbolAddrByName(interp,"a");
  sp = Ch_SymbolAddrByName(interp,"s");
  printf("In C, before expression evaluations:\n");
  printf("ii = %d, dd = %f, *p = %d\n",*pii, *pdd, **p2);
  printf("a[0] = %d, a[1] = %d, a[2] =%d\n", *pa, *(pa+1), pa[2]);
  printf("s.i = %d, s.f = %f\n", sp->i, sp->f);

  Ch_ExprEval(interp,"ii++");
  Ch_ExprEval(interp,"dd = dd + ii + 2*(1+pow(2.0,3))");
  Ch_ExprEval(interp,"func()");

  printf("In C, after expression evaluations:\n");
  printf("ii = %d, dd = %f, *p = %d\n",*pii, *pdd, **p2);

  Ch_End(interp);
}
```

Program 2.10: Evaluating expressions in a Ch program (expreval.c).

```
int ii = 10;
double dd = 1.1;
int *p = &ii;
int a[3] = {1,2,3};
struct tag {
  int i;
  double f;
} s = {10,20};

printf("printed from expreval.ch, ii = %d, dd = %f\n", ii, dd);
int func() {
  printf("func() in expreval.ch called, ii = %d, dd = %f\n", ii, dd);
  return 0;
}
```

Program 2.11: Evaluating expressions in a Ch program (expreval.ch).

```
printed from expreval.ch, ii = 10, dd = 1.100000
In C, before expression evaluations:
ii = 10, dd = 1.100000, *p = 10
a[0] = 1, a[1] = 2, a[2] =3
s.i = 10, s.f = 20.000000
func() in expreval.ch called, ii = 11, dd = 30.100000
In C, after expression evaluations:
ii = 11, dd = 30.100000, *p = 11
```

Figure 2.6: Output from executing `expreval.c`.

in the Ch space based on the current values of two variables `ii` and `dd`.

The output from executing the file built from `expreval.c` is shown in Figure 2.6.

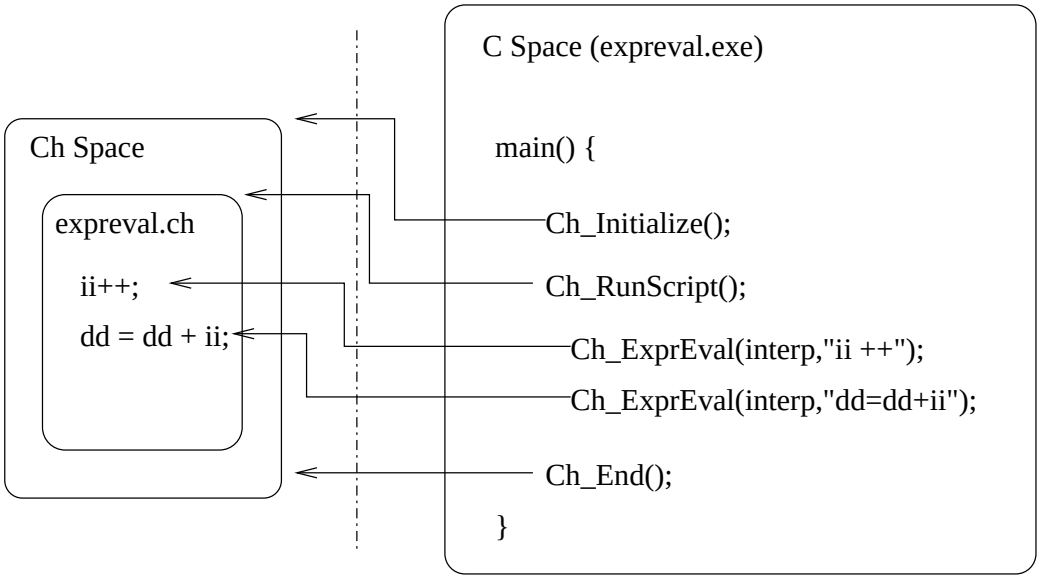


Figure 2.7: Evaluating expressions in Ch program.

Before an expression in Ch space is evaluated by function **Ch_ExprEval**(), it can be parsed to test if it is syntactically correct. For example, the expression evaluation statement

```
Ch_ExprEval(interp, "ii++");
```

can be replaced by the following statements.

```
if(!Ch_ExprParse(interp, "ii++") {
   Ch_ExprEval(interp, "ii++");
}
else {
   printf("Error: Ch expression ii++ is invalid\n");
}
```

Ch_ExprEval() has no way to return a value, so a global variable, such as `ii` or `dd`, in the Ch space is used to hold the result of the expression first. It is then accessed in the C space through its address using function **Ch_GlobalSymbolAddrByName**() or **Ch_SymbolAddrByName**(). Function **Ch_ExprCalc()** can obtain the result of an expression in the Ch space directly without through such an intermediate variable in the Ch space. For example, the statements

```
strcpy(expr,"2*(1+pow(2.0,3))+cos(0.3)");
 ...
Ch_ExprCalc(interp, expr, CH_DOUBLETYPE, &result);
```

in Program 2.12 evaluate the expression `expr` of `2*(1+pow(2.0,3))+cos(0.3)` in the Ch space and pass its result through the fourth argument. The data type of the fourth argument, specified by the third argument of the function, shall be compatible with the data type of the expression. The output from executing the program built from in Program 2.12 is shown below

```
/*****************************************************************
* File Name: exprcalc.c
* Evaluating expression in Ch space
*****************************************************************/
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  double result;
  char expr[512];

  strcpy(expr,"2*(1+pow(2.0,3))+cos(0.3)");
  Ch_Initialize(&interp, NULL); /* initialize embedded Ch */
  status = Ch_ExprParse(interp,expr);
  if(status==CH_ERROR) {
    printf("%s is invalid.\n", expr);
  }
  else {
    Ch_ExprCalc(interp, expr, CH_DOUBLETYPE, &result);
    printf("%s = %f\n", expr, result);
  }
  Ch_End(interp);
}
```

Program 2.12: Evaluating an expression in the Ch space (exprcalc.c).

```
2*(1+pow(2.0,3))+cos(0.3) = 18.955336
```

The expression evaluated by functions **Ch_ExprEval()** and **Ch_ExprCalc()** may contain functions located in function files. If the Embedded Ch is used to evaluate an expression, which may contain functions, no Ch program needs to be loaded explicitly by functions such as **Ch_RunScript**() as shown in Program 2.12. Variables and functions in the Ch space can also be added by APIs **Ch_AppendParseScript**() and **Ch_AppendRunScript**(), without calling **Ch_RunScript**() first, then used by APIs **Ch_ExprParse**(), **Ch_ExprEval**(), and **Ch_ExprCalc**() described in section 3.4.

2.5 Removing and Redeclaring Variables

Variables in the Ch space can declared inside a Ch program loaded by **Ch_RunScript**(), **Ch_RunScriptM**(), and **Ch_ParseScript**(). After the program is loaded and executed, additional global variables can be added by the APIs **Ch_AppendParseScript**(), **Ch_AppendParseScriptFile**(), **Ch_AppendRunScript**(), **Ch_AppendRunScriptFile**(). For example, a global variable `s` of int type can be declared by

```
Ch_AppendParseScript(interp, "int s;");
```

Unlike global variables, system variables remain in an instance of an Embedded Ch engine when a new program is loaded. A system variable is declared in Ch with the type qualifier `__declspec(global)` as shown below.

```
__declspec(global) int s;
```

A system variable can be declared using the API **Ch_DeclareVar**() as shown below.

```
/* File Name: removevar.c */
#include<stdio.h>
#include<embedch.h>

int main() {
    ChInterp_t interp;
    int status;
    char *argvv[]={"removevar.ch", NULL};

    Ch_Initialize(&interp, NULL);
    status = Ch_RunScript(interp,argvv);
    if(status == CH_ERROR) {
      printf("Error: execution of program removevar.ch failed\n");
    }
    Ch_ExprEval(interp, "printf(\"var = %d\n\",var)");
    if(Ch_SymbolAddrByName(interp, "var")) {
        Ch_AppendParseScript(interp,"#pragma remvar(var)");
    }
    Ch_AppendParseScript(interp,"float var = 100.0;");
    Ch_ExprEval(interp, "printf(\"var = %f\n\",var)");
    Ch_AppendParseScript(interp,"#pragma remvar(var)");
    Ch_AppendParseScript(interp,"#include <array.h>");
    Ch_AppendParseScript(interp,"array double var[2][3]={1,2,3,4,5,6};");
    Ch_ExprEval(interp, "printf(\"var = \n%f\",var)");
    Ch_End(interp);
}
```

Program 2.13: Removing and declaring variables in the Ch space (removevar.c).

```
int var = 10, b = 20;
printf("var in removevar.ch = %d\n", var);
```

Program 2.14: A Ch program to test removing and declaring variables (removevar.ch).

```
Ch_DeclareVar(interp, "int s;");
```

A system variable can be changed to a new data type specifier using the API **Ch_DeclareTypedef**(). For example, the typedefed specifier `newType_t` of int type is used to declare a system variable `sysvar`.

```
Ch_DeclareVar(interp, "int newType_t;");
Ch_DeclareTypedef(interp, "newType_t");
Ch_DeclareVar(interp, "newType_t sysvar;");
```

In some applications, it may be desirable to reuse some variables in the Ch space which are already declared. Global and system variables, including variables of function type, in the Ch space can be removed by preprocessing directive `#pragma`. For example, the following Ch statement

```
#pragma remvar(var)
```

removes the variable `var` of the global scope or system scope declared with the type specifier `__declspec(global)` in a Ch program. After a variable is removed from the symbol table, its name can be re-used and redeclared as any other data type.

For example, when the Ch script Program 2.13, loads and executes Program 2.14 using the API **Ch_RunScript**(). The variable `var` of int type is declared and initialized with value 10. Its value is then

```
var in removevar.ch = 10
var = 10
var = 100.000000
var =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
```

Figure 2.8: Output of the executable program from `removevar.c`.

printed out. API **Ch_SymbolAddrByName**() returns the address of a global variable by name in the Ch Space. If the name is not a global variable, it returns NULL. In Program 2.13, it is used to check whether the variable `var` exists in program `removevar.ch`. If it exists, the variable is removed by the function call

```
Ch_AppendParseScript(interp, "#pragma remvar(var)")
```

Then, the name `var` is redeclared as a floating-point variable of float type and initialized with the value 100.0. After it is printed out, the variable is again removed. It is then redeclared as a computational array of double type and initialized with values for each elment of the array by the following statements.

```
Ch_AppendParseScript(interp, "#include <array.h>");
Ch_AppendParseScript(interp, "array double var[2][3]={1,2,3,4,5,6};");
```

The entire array is then printed out. The output from in Program 2.13 is shown in Figure 2.8.

Chapter 3

Calling Ch Functions from C Space

In the previous chapters, we introduced methods of accessing variables and evaluating expressions in Ch programs from C/C++ programs with embedded Ch. Furthermore, users can call Ch functions in C/C++ programs (C space). There are four methods to call Ch functions from the C space. In this chapter, we will describe these methods in detail.

3.1 Calling Ch Functions by Expression Evaluations

In addition to regular expressions, Ch function calls that appear in the form of expressions can also be handled by functions **Ch_ExprEval**() and **Ch_ExprCalc**(). Using **Ch_ExprCalc**() to evaluate an expression which may contain functions is quite simple as illustrated in Program 2.12.

C program `funcall.c` in Program 3.1 is an example of calling Ch functions by expression evaluations. The Ch program `funcall.ch` shown in Program 3.2 has three global variables, `ii` with type int, `dd` with type double and `retval_g` with type int, as well as a Ch function `func()`. Function `func()` takes an argument `arg1` of type int and returns an integer with value of `2 * arg1`. It also prints out the current values of variables `ii` and `dd`. To get the return value of the Ch function in C space, the variables `pretval_g` is declared as a pointer to int in C program `funcall.c`. After the Ch program 3.2 is loaded and executed by `funcall.c` with embedded Ch, the statement below

```
pretval_g = Ch_SymbolAddrByName(interp,"retval_g");
```

obtain addresses of variable `retval_g` declared in Ch program and `pretval_g` is used to keep this address. The statement

```
Ch_ExprEval(interp,"retval_g = func(10)");
```

evaluates the expression

```
retval_g = func(10);
```

which actually is a function call in Ch space. The return value is saved in Ch variable `retval_g` which is pointed to by the C pointer `pretval_g`. After Ch variables `ii` and `dd` are changed by the following expression evaluations

```
Ch_ExprEval(interp,"ii++");
Ch_ExprEval(interp,"dd = dd + ii");
```

the Ch function `func`() is invoked again by

```
/*****************************************************************
* File Name: funcall.c
* Calling Ch functions by evaluating expressions
*****************************************************************/
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"funcall.ch", NULL};
  int *pretval_g;

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* parse embedded Ch program indicated by argvv */
  status = Ch_ParseScript(interp,  argvv);
  if(status == CH_ERROR)
    printf("Error: parse program funcall.ch failed\n");
  else {
    /* execute embedded Ch program funcall.ch */
    status = Ch_ExecScript(interp,  argvv[0]);
    if(status)
      printf("Error: execute script funcall.ch failed\n");
  }

  pretval_g = Ch_SymbolAddrByName(interp,"retval_g");
  printf("In C, the original value of *pretval_g:\n");
  printf("*pretval_g = %d\n", *pretval_g);

  Ch_ExprEval(interp,"retval_g = func(10)"); /* call Ch function func() */
  printf("In C, after the first function call:\n");
  printf("*pretval_g = %d\n", *pretval_g);

  Ch_ExprEval(interp,"ii++");
  Ch_ExprEval(interp,"dd = dd + ii");

  Ch_ExprEval(interp,"retval_g = func(ii)"); /* call Ch function func() */
  printf("In C, after expression evaluations and the second function call:\n");
  printf("*pretval_g = %d\n", *pretval_g);

  Ch_End(interp);
}
```

Program 3.1: Calling Ch function by expression evaluations (funcall.c).

```
int ii = 10;
double dd = 1.1;
int retval_g;

int func(int arg1) {
  printf("\nprinted from Ch function func(), ii = %d, dd = %f\n", ii, dd);
  return 2 * arg1;
}
```

Program 3.2: Calling Ch function by expression evaluations (funcall.ch).

```
In C, the original value of *pretval_g:
*pretval_g = 0

printed from Ch function func(), ii = 10, dd = 1.100000
In C, after the first function call:
*pretval_g = 20

printed from Ch function func(), ii = 11, dd = 12.100000
In C, after expression evaluations and the second function call:
*pretval_g = 22
```

Figure 3.1: Output from executing `funcall.c`.

```
Ch_ExprEval(interp,"retval_g = func(ii)");
```

to print out the new values of these two Ch variables.

To get the return value of the function call, a global variable, for example, `retval_g`, is used as the lvalue of an assignment operation to save the return value of the Ch function. Then it can be retrieved by function **Ch_GlobalSymbolAddrByName**() or **Ch_SymbolAddrByName**() when there is no local variable in its scope.

The output from executing the file built from `funcall.c` is shown in Figure 3.1.

Function **Ch_ExprCalc**() can also be used to calculate an expression with function calls.

The function definition in an expression can be located in a function file 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 an operand of an expression.

Variables and functions in the Ch space can also be added by APIs **Ch_AppendParseScript**() and **Ch_AppendRunScript**(), without calling **Ch_RunScript**() first, then used by APIs **Ch_ExprParse**(), **Ch_ExprEval**(), and **Ch_ExprCalc**() described in section 3.4.

3.2 Calling Ch Functions by Address Using Ch_CallFuncByAddr() and Ch_CallFuncByA

A Ch function in a Ch program can also be called by its address with API **Ch_CallFuncByAddr**(). The prototype of API **Ch_CallFuncByAddr**() is given in header file **ch.h** as follows.

```
int Ch_CallFuncByAddr(ChInterp_t interp, void *fptr, void *retval, ...);
```

```
/*****************************************************************
* File Name: funcall2.c
* Calling Ch functions by evaluating expressions
*****************************************************************/
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"funcall2.ch", NULL};
  void *fptr;
  int retval_g;


  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program funcall.ch failed\n");

  fptr = Ch_SymbolAddrByName(interp,"func"); /* obtain the address of the function */
  Ch_CallFuncByAddr(interp, fptr, &retval_g, 10); /* call Ch function func() */
  printf("In C, after the first function call:\n");
  printf("retval_g = %d\n", retval_g);

  Ch_ExprEval(interp,"ii++");
  Ch_ExprEval(interp,"dd = dd + ii");

  Ch_CallFuncByAddr(interp, fptr, &retval_g, 20); /* call Ch function func() */
  printf("In C, after expression evaluations and the second function call:\n");
  printf("retval_g = %d\n", retval_g);

  Ch_End(interp);
}
```

Program 3.3: Calling Ch function with API **Ch_CallFuncByAddr**() (funcall2.c).

The first argument is an instance of Ch interpreter. The second argument is the pointer containing the address of the function, and the third one is the pointer containing the address of the return value of the called function. If the Ch function to be called takes arguments, they will be listed following the third argument `retval`. The function returns zero on successful execution or non-zero on failure.

C program `funcall2.c` in Program 3.3 is an example of calling Ch functions with API **Ch_CallFuncByAddr**(). The Ch program `funcall2.ch` shown in Program 3.4 is similar to the one in Program 3.2. The difference is that the global Ch variable `retval_g` is not used in this case. To get the return value of the Ch function, the `retval_g` is declared as an int in C space instead. After the Ch program 3.4 is executed, the statement below

```
fptr = Ch_SymbolAddrByName(interp,"func");
```

obtain the address of Ch function `func`(). Then function call

```
Ch_CallFuncByAddr(interp, fptr, &retval_g, 10);
```

```
int ii = 10;
double dd = 1.1;

int func(int arg1) {
  printf("\nprinted from Ch function func(), ii = %d, dd = %f\n", ii, dd);
  return 2 * arg1;
}
```

Program 3.4: Calling Ch function with API **Ch_CallFuncByAddr**() (funcall2.ch).

```
printed from Ch function func(), ii = 10, dd = 1.100000
In C, after the first function call:
retval_g = 20

printed from Ch function func(), ii = 11, dd = 12.100000
In C, after expression evaluations and the second function call:
retval_g = 40
```

Figure 3.2: Output from executing `funcall2.c`.

invokes this Ch function by its address pointed to by `fptr`. The return value is saved in variable `retval_g`, and the integer `10` is passed as the argument. In the same way, the function call below

```
Ch_CallFuncByAddr(interp, fptr, &retval_g, 20);
```

calls the same Ch function with argument of `20` at the second time.

The output from executing the file built from `funcall2.c` is shown in Figure 3.2.

A Ch function in a Ch program can also be called by its address with API **Ch_CallFuncByAddrv**(). The prototype of API **Ch_CallFuncByAddrv**() is given in header file **ch.h** as follows.

```
int Ch_CallFuncByAddrv(ChInterp_t interp, void *fptr, void *retval, va_list ap);
```

where *ap* is the variable argument list of a function obtained by va_start(). Details about **Ch_CallFuncByAddrv**() can be found in *Ch SDK User's Guide*.

3.3 Calling Ch Functions by Name Using Ch_CallFuncByName() and Ch_CallFuncByNa

In addition to **Ch_CallFuncByAddr**(), API **Ch_CallFuncByName**() can also be used to call a Ch function in the Ch program. In this case, the Ch function is called by its name instead of the address. The prototype of API **Ch_CallFuncByName**() is given in header file **ch.h** as follows.

```
int Ch_CallFuncByName(ChInterp_t interp, char *name, void *retval, ...);
```

The first argument is an instance of Ch interpreter. The second argument is a string containing the name of the function. The function `name` can be a function located in a function file 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 third argument is the pointer containing the address of the return value of the called function. If the Ch function takes arguments, they

```
/*****************************************************************
* File Name: funcall3.c
* Calling Ch functions by evaluating expressions
*****************************************************************/
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"funcall2.ch", NULL};
  void *fptr;
  int retval_g;


  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp,argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program funcall.ch failed\n");

  Ch_CallFuncByName(interp, "func", &retval_g, 10); /* call Ch function func() */
  printf("In C, after the first function call:\n");
  printf("retval_g = %d\n", retval_g);

  Ch_ExprEval(interp,"ii++");
  Ch_ExprEval(interp,"dd = dd + ii");

  Ch_CallFuncByName(interp, "func", &retval_g, 20); /* call Ch function func() */
  printf("In C, after expression evaluations and the second function call:\n");
  printf("retval_g = %d\n", retval_g);

  Ch_End(interp);
}
```

Program 3.5: Calling Ch function with API **Ch_CallFuncByName**() (funcall3.c).

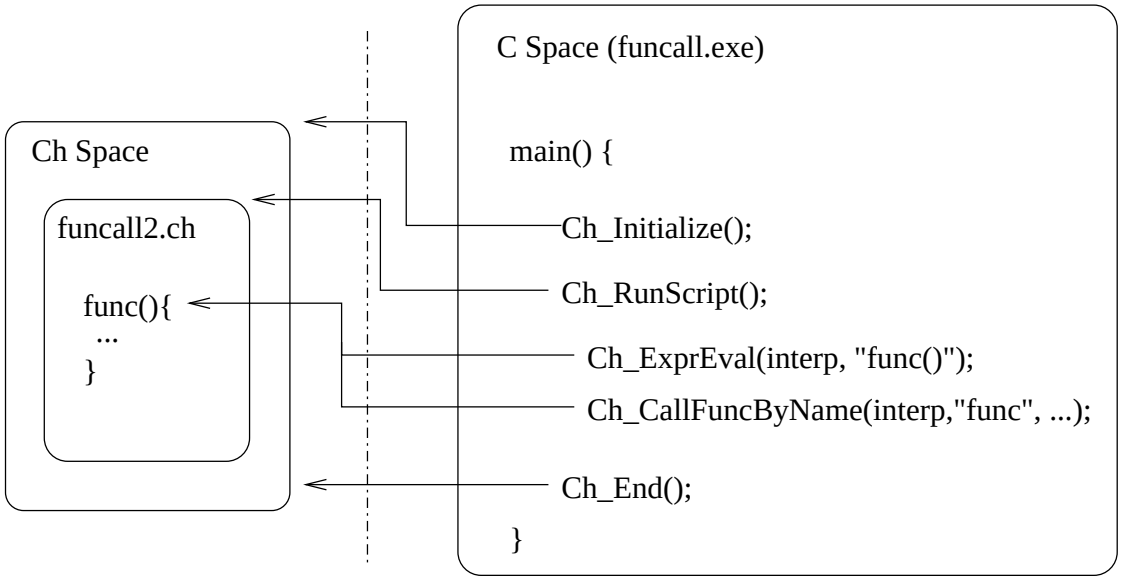


Figure 3.3: Calling functions in Ch program.

will be listed following the third argument `retval`. The function returns zero on successful execution or non-zero on failure.

C program `funcall3.c` in Program 3.5 is an example of calling Ch functions with API **Ch_CallFuncByName**(). Program 3.4 is used as the Ch program in this example. Figure 3.3 illustrates how to call a Ch function `func`() by expression evaluations and API **Ch_CallFuncByName**(). After the Ch program is loaded and executed, the statement below

```
Ch_CallFuncByName(interp, "func", &retval_g, 10);
```

invokes Ch function `func`() by its name. It is equivalent to statements

```
fptr = Ch_SymbolAddrByName(interp,"func");
Ch_CallFuncByAddr(interp, fptr, &retval_g, 10);
```

in Program 3.3. The output from executing the file built from `funcall3.c` is the same as Figure 3.2.

The argument of function `func()` in Program 3.4 is int type. If an argument of the Ch function is a pointer type, the data in the C space can be shared by the variable in the Ch space. The details about sharing C space data by variables in the Ch space through function arguments are described in detail in section 4.1.

A Ch function in a Ch program can also be called by its name with API **Ch_CallFuncByNamev**(). The prototype of API **Ch_CallFuncByNamev**() is given in header file **ch.h** as follows.

```
int Ch_CallFuncByNamev(ChInterp_t interp, const char *name*fptr,
                       void *retval, va_list ap);
```

where *ap* is the variable argument list of a function obtained by va_start(). This API can be used to implement user's own API for calling functions in the C/C++ script space as shown Program 3.6. The output from Program 3.6. is given as follows.

```
retval = 5.000000
retval = 20.000000
```

Details about **Ch_CallFuncByNamev**() can be found in *Ch SDK User's Guide*.

```
/* File: callfuncbynamev.c */
#include <stdio.h>
#include <embedch.h>

ChInterp_t interp;

/* this function Some_API() is exported and documented for user */
int Some_API(char *funcname, void *retval, ...) {
  va_list ap;

  va_start(ap, retval);
  /* call Ch function by its name */
  Ch_CallFuncByNamev(interp, funcname, retval, ap);
  va_end(ap);
}

int main () {
    char *code = "double func(int n) {int retval;\
                  retval = 2*n; return retval;} \
                  int main() {printf(\"hello\n\"); return 0;}";
    int c_i = 10;
    double retval;


    Ch_Initialize(&interp, NULL);
    /* call Ch function by its name */
    Some_API("hypot", &retval, 3.0, 4.0);
    printf("retval = %f\n", retval);

    Ch_AppendRunScript(interp,code);
    Some_API("func", &retval, c_i);
    printf("retval = %f\n", retval);
    Ch_End(interp);
    return 0;
}
```

Program 3.6: Calling Ch functions using user's own API through function **Ch_CallFuncByNamev**().

```
/* File: funcall4.c */
#include <stdio.h>
#include <embedch.h>

int main () {
    ChInterp_t interp;
    char *code = "double func(int n) {int retval;\
                  retval = 2*n; return retval;} \
  int main() {printf(\"hello\n\"); return 0;}";
    int c_i = 10;
    double retval;

    Ch_Initialize(&interp, NULL);
    Ch_CallFuncByName(interp, "hypot", &retval, 3.0, 4.0);
    printf("retval = %f\n", retval);
    Ch_AppendRunScript(interp,code);
    Ch_CallFuncByName(interp, "main", &retval);
    Ch_CallFuncByName(interp, "func", &retval, c_i);
    printf("retval = %f\n", retval);
    Ch_AppendRunScript(interp,"int ch_i;");
    Ch_SetVar(interp,"ch_i", CH_INTTYPE, c_i);
    Ch_ExprEval(interp,"ch_i *= 5");
    Ch_ExprCalc(interp,"10*ch_i+hypot(3,4)",
                CH_DOUBLETYPE, &retval);
    printf("retval = %f\n", retval);
    Ch_End(interp);
    return 0;
}
```

Program 3.7: Calling Ch function and accessing Ch variable created in the C space.

3.4 Calling Ch Functions and Accessing Ch Variables Created in C

In some applications, the user provides mathematical expressions and formulas in C syntax. These mathematical expressions and formulas can be sent to an embedded Ch for processing directly. Functions in function files can also be used directly in embedded Ch. Variables and functions in the Ch space can be added by APIs **Ch_AppendParseScript**() and **Ch_AppendRunScript**(), without calling **Ch_RunScript**() first, then used by APIs **Ch_ExprParse**(), **Ch_ExprEval**(), **Ch_ExprCalc**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), and **Ch_CallFuncByNameVar**().

For example, Program 3.7 calls function `hypot()` in a function file directly by

```
Ch_CallFuncByName(interp, "hypot", &retval, 3.0, 4.0);
```

The third argument passes back the returned value in the called function. Functions `func()` and `main()` are first added to the Ch space by

```
Ch_AppendRunScript(interp, code);
```

then called by **Ch_CallFuncByName**(). Function `main()` is optional. Unlike using **Ch_RunScript**(), function `main()`, loaded by **Ch_AppendRunScript**() or **Ch_AppendRunScriptFile**(), has to be called explicitly using **Ch_CallFuncByName**() or other APIs. Next, function **Ch_AppendRunScript**() is used to declare variable `ch_i` in the Ch space. The variable `ch_i` of int type in the Ch space is assigned with the value for `c_i` in the Ch space by function call

```
Ch_SetVar(interp, "ch_i", CH_INTTYPE, c_i);
```

Details about using function **Ch_SetVar**() to set variables in the Ch space are described in section 2.2. The expression and formulas, including functions in function files, in the Ch space can be calculated by either functions **Ch_ExprEval**() or **Ch_ExprCalc**(). Function **Ch_ExprCalc**() can pass back the result of the calculated expression. Program 3.7 the expression contains function `hypot()` in a function file in function call.

```
Ch_ExprCalc(interp, "10*ch_i+hypot(3,4)", CH_DOUBLETYPE, &retval);
```

The output from Program 3.7 is given as follows.

```
retval = 5.000000
hello
retval = 20.000000
retval = 505.000000
```

3.5 Calling Ch Functions by Ch_CallFuncByNameVar()

Both functions **Ch_CallFuncByAddr**(), and **Ch_CallFuncByName**() can be used to call functions with different number of arguments. However, a function to be called by these two APIs has to be hardcoded. API **Ch_CallFuncByNameVar**() can be used to call different functions with different arguments dynamically. The prototype of API **Ch_CallFuncByNameVar**() is given in header file **ch.h** as follows.

```
int Ch_CallFuncByNameVar(ChInterp_t interp, char *name, void *retval,
                         void *arglist);
```

The first argument is an instance of Ch interpreter. The second argument is a string containing the name of the function. The function `name` can be a function located in a function file 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 third argument is the pointer containing the address of the return value of the called function. 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**(). The function returns zero on successful execution or non-zero on failure.

C program `callfuncbynamevar.c` in Program 3.8 is an example of calling Ch functions with API **Ch_CallFuncByNameVar**(). Program 3.9 is used as a Ch program in this example. The program illustrates how arguments of scalar, array, computational array, user defined data type such as structure, class, union are handled. Function `func()` in the Ch space is called by API

```
Ch_CallFuncByNameVar(interp, "func", &retval, arglist);
```

in C Program 3.8. The first call uses variables in the Ch space as arguments. It is equivalent to calling function `func()` by

```
func(n_ch, m_ch, f_ch, p_ch, a_ch, b_ch, c_ch, d_ch, _chs, sp_ch);
```

in the Ch space. When function `func()` is called the second time by API **Ch_CallFuncByNameVar**(), it uses values in the C space. It is equivalent to calling function `func()` by

```
func(n, m, f, p, a, b, c, d, s, sp);
```

```
/* File Name: callfuncbynamevar.c */
#include<stdio.h>
#include<embedch.h>
struct tag { int i; double f; } s = {10,20}, *sp=&s;

int main() {
    ChInterp_t interp;
    int n = 2, m = 3, *p = &n;
    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};
    int status, retval, dim = 2;
    char *argvv[]={"callfuncbynamevar.ch", NULL};
    ChVaList_t arglist;
    Ch_Initialize(&interp, NULL);
    status = Ch_RunScript(interp,argvv);
    if(status == CH_ERROR) {
      printf("Error: execution of program callfuncbynamevar.ch failed\n");
    }
    /* use variables and expressions in Ch space */
    arglist= Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArgExpr(interp, &arglist, "2");
    Ch_VarArgsAddArgExpr(interp, &arglist, "m_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "f_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "p_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "a_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "b_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "c_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "d_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "s_ch");
    Ch_VarArgsAddArgExpr(interp, &arglist, "sp_ch");
    /* equivalent of retval = func(n_ch, m_ch, f_ch, p_ch, a_ch,
                                   b_ch, c_ch, d_ch, _chs, sp_ch); */
    Ch_CallFuncByNameVar(interp, "func", &retval, arglist);
    Ch_VarArgsDelete(interp, arglist);

    /* use values in C space */
    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);
    /* equivalent of retval = func(n, m, f, p, a, b, c, d, s, sp); */
    Ch_CallFuncByNameVar(interp, "func", &retval, arglist);
    Ch_VarArgsDelete(interp, arglist);
    Ch_End(interp);
}
```


Program 3.8: Calling Ch function with API **Ch_CallFuncByNameVar**() (callfuncbynamevar.c).

```
#include<array.h>
#include<stdio.h>

int n_ch = 2, ch_m = 3, *p_ch = &n_ch;
double f_ch = 10.0;
int a_ch[2][3] = {1, 2, 3,
                  4, 5, 6};
int b_ch[2][3] = {1, 2, 3,
                  4, 5, 6};
array int c_ch[2][3] = {1, 2, 3,
                        4, 5, 6};
array int d_ch[2][3] = {1, 2, 3,
                        4, 5, 6};
struct tag {
  int i;
  double f;
} s_ch = {10,20}, *sp_ch=&s_ch;


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)
{
   printf("func() called--------\n");
   printf("n = %d\n", n);
   printf("m = %d\n", m);
   printf("f = %f\n", f);
   printf("*p = %d\n", *p);
   printf("a[1][1] = %d\n", a[1][1]);
   printf("b[1][2] = %d\n", a[1][2]);
   printf("c = \n%d", c);
   printf("d = \n%d", d);
   printf("s.i = %d\n", s.i);
   printf("sp->f = %f\n", sp->f);
   return 0;
}

/*
int main() {
  func(n_ch, m_ch, f_ch, p_ch, a_ch, b_ch, c_ch, d_ch, s_ch, sp_ch);
  return 0;
}
*/
```

Program 3.9: Ch program with function func() called by **Ch_CallFuncByNameVar**() (callfuncbynamevar.ch).

```
func() called--------
n = 2
m = 3
f = 10.000000
*p = 2
a[1][1] = 5
b[1][2] = 6
c =
1 2 3
4 5 6
d =
1 2 3
4 5 6
s.i = 10
sp->f = 20.000000
func() called--------
n = 2
m = 3
f = 10.000000
*p = 2
a[1][1] = 5
b[1][2] = 6
c =
1 2 3
4 5 6
d =
1 2 3
4 5 6
s.i = 10
sp->f = 20.000000
```

Figure 3.4: Output from executing `callfuncbynamevar.c`.

These two function calls in Program 3.8 generate the same output shown in Figure 3.4.

In Program 3.8, after a variable argument list in the Ch space is created by function **Ch_VarArgsCreate**(), arguments of the called function is added to the list by either function function **Ch_VarArgsAddArg**() or **Ch_VarArgsAddArgExpr**(). Then, the function `func()` in the Ch space is called by **Ch_CallFuncByNameVar**(). Finally, the variable argument list is deleted by **Ch_VarArgsDelete**().

Function **Ch_VarArgsAddArgExpr**() adds an expression in the Ch space as an augment to the argument list. Therefore, variables in an expression shall be declared in a Ch program before this **Ch_VarArgsAddArgExpr**() is called. Function **Ch_VarArgsAddArg**() adds an expression in the C space as an augment to the argument list. No variable in the Ch space needs to be explicitly declared. In a typical application, the argument list `arglist` is typically built dynamically through a linked list of values or expressions obtained from the user input. To build an argument list `arglist` dynamically using function **Ch_VarArgsAddArg**(), data type, array dimension, etc. may need to be obtained for the called function, such as `func()`, in the hosting program such as Program 3.8.

The data type, array element data type, and function return type of an argument of a function can be obtained by function **Ch_FuncArgDataType**(). If the argument of a function is array type, its array type, dimension, and extent for each dimension can be obtained by functions **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), and **Ch_FuncArgArrayType**(), respectively. Whether the argument of a function is a pointer to function, pointer to function with variable number of arguments, and the number of arguments of the pointer to function can be obtained by functions **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), and **Ch_FuncArgFuncArgNum**(), respectively. The tag name and size of the user defined data in terms of class, structure, and union for an argument of a function can be obtained by functions **Ch_FuncArgUserDefinedName**() and **Ch_FuncArgUserDefinedSize**(), respectively.

Arguments `c` and `d` for function `func()` are computational arrays which are also called Ch arrays. Either C or Ch array can be passed to an argument of Ch array type. Therefore, function calls

```
Ch_VarArgsAddArg(interp, &arglist, CH_CARRAYTYPE, CH_INTTYPE, c, dim, n, m);
Ch_VarArgsAddArg(interp, &arglist, CH_CARRAYTYPE, CH_INTTYPE, d, dim, n, m);
```

for building the argument list of the second call of API **Ch_CallFuncByNameVar**() in Program 3.8 can be replaced by

```
Ch_VarArgsAddArg(interp, &arglist, CH_CHARRAYTYPE, CH_INTTYPE, c, dim, n, m);
Ch_VarArgsAddArg(interp, &arglist, CH_CHARRAYTYPE, CH_INTTYPE, d, dim, n, m);
```

Note that Program 3.8 does not contain function `main()`. The code is processed first by API **Ch_RunScript**() as if it was an initialization for a Ch script. This kind of initialization can be useful for handling a large script code with many functions which are invoked later in the host program. For example, header file `windows.h` which contains a large number of lines of source code can be first processed in the API **Ch_RunScript**(). Then, Windows API in `windows.h` can can be called by Ch functions from the host program.

3.6 Calling Ch Functions with Variable Number of Arguments

Calling Ch functions with a variable number of arguments will be described in this section. In all previous examples, Ch functions take no argument or fixed number of arguments. In practice, some Ch functions can take variable number of arguments, for example

```
int func1(va_list ap);
int func2(int i, ...);
int func3(...);
```

These three typical prototypes can represents almost all Ch functions with variable number of arguments. The function `func1`() take a variable length argument list `ap` with type of `va_list`. The function `func2`() take variable number of arguments, but at least one argument, for example `i`, need to be passed in when this function is called. The function `func3`() take variable number of arguments, and it can even have no argument at all. The former two functions are also valid in C, whereas function `func3`() is only valid in Ch. This format allows a function to take any number of arguments without constraint on the first argument. More information on how this format for handling function with a variable number of arguments can be found in *Ch User's Guide*. The definitions of these three functions are shown in Program 3.10.

The APIs **va_start**(), **va_end**(), **va_count**() and **va_elementtype**() are used to handle the variable length argument list. These APIs are described in details in *The Ch Language Environment Reference Guide*.

To call these Ch functions from the C space with embedded Ch, we need to create Ch style variable length argument lists in the C space first, then pass these lists to Ch functions by calling **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**(), or **Ch_CallFuncByAddr**(). The C program with embedded Ch to call Ch programs taking variable number of arguments are shown in Program 3.11, where APIs **Ch_VarArgsCreate**(), **Ch_VarArgsAddArg**() **Ch_VarArgsAddArgVar**() and **Ch_VarArgsDelete**() are used to create Ch style variable length argument lists. The whole procedure is as follows: **Ch_VarArgsCreate**() is used first to initialize a Ch style variable length argument list firt. Then **Ch_VarArgsAddArg**() is used to add arguments with values in the C space into this list. Or **Ch_VarArgsAddArgVar**() is used to add arguments in the Ch space into the list. Finally **Ch_VarArgsDelete**() is used to delete this list. The result of running executable compiled from Program 3.11. is shown as Figure 3.5.

Function **Ch_CallFuncByName**() used in

```
Ch_CallFuncByName(interp, "func1", &retval, ap_ch);
```

and

```
Ch_CallFuncByName(interp, "func3", &retval, ap_ch);
```

in Program 3.11 can be replaced by function **Ch_CallFuncByNameVar**(). To use **Ch_CallFuncByNameVar**() to call function `func2()` in the Ch space, the first argument of function `func2()` should also be placed in the argument list. The code

```
ap_ch = Ch_VarArgsCreate(interp);
Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE,  j);
Ch_VarArgsAddArg(interp, &ap_ch, CH_FLOATTYPE, f);
Ch_CallFuncByName(interp, "func2", &retval, i, ap_ch);
Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
Ch_CallFuncByName(interp, "func2", &retval, i, ap_ch);
Ch_VarArgsDelete(ap_ch);
```

in Program 3.11 should be changed to

```
ap_ch = Ch_VarArgsCreate(interp);
Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE,  i);
Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE,  j);
Ch_VarArgsAddArg(interp, &ap_ch, CH_FLOATTYPE, f);
Ch_CallFuncByNameVar(interp, "func2", &retval, ap_ch);
Ch_VarArgsAddArg(&ap_ch, CH_DOUBLETYPE, d);
Ch_CallFuncByNameVar(interp, , "func2", &retval, ap_ch);
Ch_VarArgsDelete(interp, ap_ch);
```

```
#include <stdarg.h>
int func1(va_list ap) {
  int vacount, i, val_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 func2(int i, ...) {
  int vacount, val_i, i;
  float val_f;
  double val_d;
  va_list ap;

  va_start(ap, i);
  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);
     }
  }
  va_end(ap);
  return 0;
}

int func3(...) {
  int vacount, i;
  va_list ap;

  va_start(ap, VA_NOARG);
  func1(ap);
  va_end(ap);
  return 0;
}
```


Program 3.10: Calling Ch functions taking variable number of arguments (callchvna.ch).

```
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    char *argvv[]={"callchvna.ch", NULL};
    int retval, status;
    int i = 5;
    int j = 10;
    float f=20;
    double d=40;
    ChVaList_t ap_ch;

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch function file */
    status = Ch_RunScript(interp, argvv);
    if(status == CH_ERROR)
      printf("Error: execution of program callchvna.ch failed\n");

    /* int func1(va_list ap) */
    printf("calling func1() from C space:\n");
    ap_ch = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, j);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_FLOATTYPE, f);
    Ch_CallFuncByName(interp, "func1", &retval, ap_ch);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
    Ch_CallFuncByName(interp, "func1", &retval, ap_ch);
    Ch_VarArgsDelete(interp, ap_ch);

    /* int func2(int i, ...) */
    printf("\ncalling func2() from C space:\n");
    ap_ch = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE,  j);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_FLOATTYPE, f);
    Ch_CallFuncByName(interp, "func2", &retval, i, ap_ch);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
    Ch_CallFuncByName(interp, "func2", &retval, i, ap_ch);
    Ch_VarArgsDelete(interp, ap_ch);

    /* int func3(...) */
    printf("\ncalling func3() from C space:\n");
    ap_ch = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_INTTYPE, j);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_FLOATTYPE, f);
    Ch_CallFuncByName(interp, "func3", &retval, ap_ch);
    Ch_VarArgsAddArg(interp, &ap_ch, CH_DOUBLETYPE, d);
    Ch_CallFuncByName(interp, "func3", &retval, ap_ch);
    /*  or
     Ch_CallFuncByNameVar(interp, "func3", &retval, ap_ch);  */
    Ch_VarArgsDelete(interp, ap_ch);

    Ch_End(interp);
}
```

Program 3.11: Calling Ch functions taking variable number of arguments (callchvna.c).

```
calling func1() from C space:
vacount = 2
val_i = 10
val_f = 20.000000
vacount = 3
val_i = 10
val_f = 20.000000
val_d = 40.000000

calling func2() from C space:
vacount = 2
val_i = 10
val_f = 20.000000
vacount = 3
val_i = 10
val_f = 20.000000
val_d = 40.000000

calling func3() from C space:
vacount = 2
val_i = 10
val_f = 20.000000
vacount = 3
val_i = 10
val_f = 20.000000
val_d = 40.000000
```

Figure 3.5: Output of executable compiled from Program 3.11.

Besides using APIs **Ch_CallFuncByName**(), **Ch_CallFuncByNameVar**(), and **Ch_CallFuncByAddr**(), we can call Ch function by expression evaluations with APIs **Ch_ExprEval**() and **Ch_ExprCalc**(). It has been mentioned in the pervious chapter. This method can also be used to call Ch functions taking variable length argument lists. For example, functions `func2`() and `func3`() in Program 3.10 are called in Program 3.12 by expression evaluations. The result of running executable compiled from Program 3.12. is shown as Figure 3.6. Note that the function `func1`() which takes a variable length argument list cannot be called by this method.

```
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    char *argvv[]={"callchvna.ch", NULL};
    int retval, status;
    int i = 5;
    int j = 10;
    float f=20;
    double d=40;

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch function file */
    status = Ch_RunScript(interp, argvv);
    if(status == CH_ERROR)
      printf("Error: execution of program callchvna.ch failed\n");

    printf("calling func2() from C space:\n");
    /* int func2(int i, ...) */
    Ch_ExprEval(interp, "func2(5, 10, 20.0)");
    Ch_ExprEval(interp, "func2(5, 10, 20.0, 40.0)");

    printf("\ncalling func3() from C space:\n");
    /* int func3(...) */
    Ch_ExprEval(interp, "func3(10, 20.0)");
    Ch_ExprEval(interp, "func3(10, 20.0, 40.0)");

    Ch_End(interp);
}
```

Program 3.12: Calling Ch functions taking variable number of arguments by expression evaluations (callchvna2.c).

```
calling func2() from C space:
vacount = 2
val_i = 10
val_d = 20.000000
vacount = 3
val_i = 10
val_d = 20.000000
val_d = 40.000000

calling func3() from C space:
vacount = 2
val_i = 10
val_d = 20.000000
vacount = 3
val_i = 10
val_d = 20.000000
val_d = 40.000000
```

Figure 3.6: Output of executable compiled from Program 3.12.

Chapter 4

Accessing C/C++ Variables from Ch Space

In Chapter 2, we discussed how to access variables in Ch space from C space with embedded Ch. Because both C and Ch use the same memory layout internally, accessing variables and data in C space from Ch space is much simpler than any other scripting languages.

Accessing global variables in C space, in static or dynamically loaded library declared in a header file, from Ch space has been described in Chapter 4 in *Ch SDK User's Guide*. The methods are readily applicable to embedded Ch programs. However, with Embedded Ch, interface between C and Ch is much simpler. This chapter describes the methods to access variables and share data in C space from Ch space in an embedded Ch program. Section 2.2 in Chapter 2 described how to use the API **Ch_SetVar**() to assign values in C space to variables in Ch space. Section 4.1 illustrates how to share the data in C space from Ch space using a function argument after the embedded Ch program has been executed. Section 4.2 describes the method to export or access variables in C space from an embedded Ch program. Section 4.3 describes the method to export or access variables in C space from an embedded Ch program using a dynamically loaded library.

4.1 Accessing C/C++ Variables from Ch Space Through Function Arguments

The memories of scalar variables, arrays, and structures in C space can be seamlessly shared in Ch space and can be achieved by just simply passing the address of the memory through function calls. The concept of sharing data in C and Ch spaces through a function argument will be described through an example in this section.

Program 4.1 in C space has variables `c_d` of double type and `c_data` of an int array of 10 elements. This C program invokes the Ch Program 4.2 with embedded Ch using function **Ch_RunScript**(), and then calls Ch function `chfun`() by its name using function **Ch_CallFuncByName**(). The memory addresses for variables `c_d` and `c_data` in C space are passed to function `chfun`() in Ch space as arguments. The function `chfun`() has no return value. The output from executing Program 4.1 is shown in Figure 4.1.

In this example, the data and memory in C space are shared in Ch space after the embedded Ch program has been executed. Using the same technique, the data and memory in Ch space can also be shared in C space.

```
#include<stdio.h>
#include<string.h>
#include<embedch.h>


int main () {
  ChInterp_t interp;
  int status;
  char *argvv[]={"embedch4.ch", NULL};
  double c_d = 10;
  int c_data[10] = {1,2,3,4,5,6,7,8,9,10};

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp,argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch4.ch failed\n");

  printf("in embedch4.c, c_d = %f, c_data[0] = %d, c_data[9] = %d\n",
         c_d, c_data[0], c_data[9]);
  Ch_CallFuncByName(interp, "chfun", NULL, &c_d, c_data);
  printf("in embedch4.c, c_d = %f, c_data[0] = %d, c_data[9] = %d\n",
         c_d, c_data[0], c_data[9]);
  c_d = 100;       /* change the variable value in C space */
  c_data[0] = 101; /* change the variable value in C space */
  c_data[9] = 110; /* change the variable value in C space */
  Ch_CallFuncByName(interp, "chfun", NULL, &c_d, c_data);
  printf("in embedch4.c, c_d = %f, c_data[0] = %d, c_data[9] = %d\n",
         c_d, c_data[0], c_data[9]);
  Ch_End(interp);
  return 0;
}
```

Program 4.1: Example of sharing data in C and Ch spaces (embedch4.c).

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

/* the main() function is not needed in this example */
int main() {
  printf("embedch4.ch invoked\n");
  return 0;
}
void  chfun(double *ch_d, int *ch_data) {
  printf("in the Ch function chfun(), *ch_d = %f, ch_data[0] = %d, ch_data[9] = %d\n",
          *ch_d, ch_data[0], ch_data[9]);
  *ch_d += 1;
  ch_data[0]++;
  ch_data[9]++;
}
```

Program 4.2: Example of sharing data in C and Ch spaces (embedch4.ch).

```
embedch4.ch invoked
in embedch4.c, c_d = 10.000000, c_data[0] = 1, c_data[9] = 10
in the Ch function chfun(), *ch_d = 10.000000, ch_data[0] = 1, ch_data[9] = 10
in embedch4.c, c_d = 11.000000, c_data[0] = 2, c_data[9] = 11
in the Ch function chfun(), *ch_d = 100.000000, ch_data[0] = 101, ch_data[9] = 110
in embedch4.c, c_d = 101.000000, c_data[0] = 102, c_data[9] = 111
```

Figure 4.1: Output from executing `embedch4.c`.

4.2 Accessing Variables in a Header File in C Space from Ch Space

Accessing global variables in C space, in a static or dynamically loaded library declared in a header file, from Ch space has been described in Chapter 4 in *Ch SDK User's Guide*. The methods are simple and readily applicable to embedded Ch programs. In this section, a same example presented there will be used to illustrate how to access global variables in a main application program in C space by embedded Ch scripts.

Program 4.3 is a C header file containing two global variables `i1` and `i2`. 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. These two variables in C space will be exported and shared by embedded Ch programs. They will be used to illustrate two methods of handling global variables. C header file `expvar_c.h` in Program 4.3, interface functions in Program 4.4, and application program 4.5 will be compiled and linked to create the executable program *prog.exe* using a makefile in Program 4.6 and Program 4.7 for Unix and Windows, respectively.

```
#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 */

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

#endif
```

Program 4.3: C header file containing global variables (`expvar_c.h`).

```
#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;

  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 */

  Ch_VaEnd(interp, ap);
  return;
}

EXPORTCH void changeCVar_chdl() {
  ChInterp_t interp;

  i2 = 400; /* change value of C variable */

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

  return;
}
```

Program 4.4: The chdl functions for handling global variables (`expvar_chdl.c`).

```
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    char *argvv[]={"expvar.ch", NULL};
    int status;

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch program */
    status = Ch_RunScript(interp, argvv);
    if(status)
      printf("Error: execution of program.ch failed\n");

    Ch_End(interp);
}
```

Program 4.5: C program invoking embedded Ch program (prog.c).

```
target: prog.exe

prog.exe: prog.o expvar_chdl.o
        ch dllink prog.exe -embedded prog.o expvar_chdl.o
prog.o: prog.c
        ch dlcomp libsample.dl prog.c
expvar_chdl.o:  expvar_chdl.c expvar_c.h
        ch dlcomp libsample.dl expvar_chdl.c -I./
clean:
        rm -f *.o *.obj *.exp *.lib *.exe
```

Program 4.6: Makefile for Unix (Makefile).

```
target: prog.exe

prog.exe: prog.obj expvar_chdl.obj
        ch dllink prog.exe -embedded prog.obj expvar_chdl.obj
prog.obj: prog.c
        ch dlcomp libsample.dl prog.c
expvar_chdl.obj:  expvar_chdl.c expvar_c.h
        ch dlcomp libsample.dl expvar_chdl.c -I./
clean:
        rm -f *.o *.obj *.exp *.lib *.exe
```

Program 4.7: Makefile for Windows (Makefile).

As described in the previous chapters, function **dlopen()** is used to load the application program itself for accessing its exported variables in the Ch header file shown in Program 4.8. 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 4.8, so that we can get the C space value of `i2` from Ch space using macro `i2` defined as follows

```
#define i2 *i2_
```

Then the chdl function `expvar_chdl`() defined in Program 4.4 is called by statements

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

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

In the chdl function `expvar_chdl`() shown in Program 4.4, 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 `i1` 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. The function `changeCVar_chdl`() changes the values of `i1` and i2 in C space. The embedded Ch script shown in Program 4.9 prints out the corresponding values of `i1` and `i2` in Ch space before and after `changeCVar`() is called. The `i2` is changed from `200` to `400`, whereas `i1` keeps the original value `100`. The result from executing embedded Ch Program 4.9 within the application Program 4.5 is appended at the end of the file. Note that the function `changeCVar()` in Program 4.9 calls the interface function `changeCVar_chdl()` Program 4.4 to access and change the values of the variables in C space.

```
#ifndef _EXPVAR_H_
#define _EXPVAR_H_

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

/* load DLL */
#include <dlfcn.h>
void *_ChExpvar_handle;
#ifndef _HPUX_
_ChExpvar_handle = dlopen(NULL, 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);
#endif

/* Get addresses of global variables in C space */
void *fptr_expvar_h;
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_);

#endif
```

Program 4.8: Ch header file for handling global variables (`expvar.h`).

```
#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() {
  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);
  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
***/
```

Program 4.9: expvar.ch: Ch application accessing global variables in C space.

4.3 Accessing Variables in a Header File in C Space from Ch Space with .dl File

Variables and data in C space can also be accessed from Ch space through a dynamically loaded library in a .dl file, which can be illustrated by Figure 4.2. Assume that we have a C program `embedch3.exe` in which there is a global variable `c_var`. This C program invokes a Ch program `embedch3.ch` with embedded Ch, and then calls Ch function `chfun`() by its name. Our goal here is to access the C global variable `c_var` from `chfun`(). The method we will use is to load a DLL file `libembedch3.dl` in which a chdl function returns the value of `c_var`. Note that the type specifier **IMPORTCH** defined in header file `ch.h` in declaration of `c_var` in `embedch3_chdl.c` indicates it is an external variable and the same memory of `c_var` in `embededch3.exe` is used.

The source code of this example is listed in Programs 4.10 to 4.12. Besides the value of `c_var`, the Ch function `chfun`() in Program 4.11 also obtains the address of C global variable `c_var` by passing the address of `ch_varptr`, which is declared as a pointer to int, to chdl function `embedch3_chdl`(). Therefore, the value of `c_var` can be changed in Ch space and be effective after returning back to C space.

Note: If the global variable in C space is a type of `const`, we can obtain the value of global variable instead of address in Ch space.

The output from executing the file built from `embedch3.c` is shown in Figure 4.3.

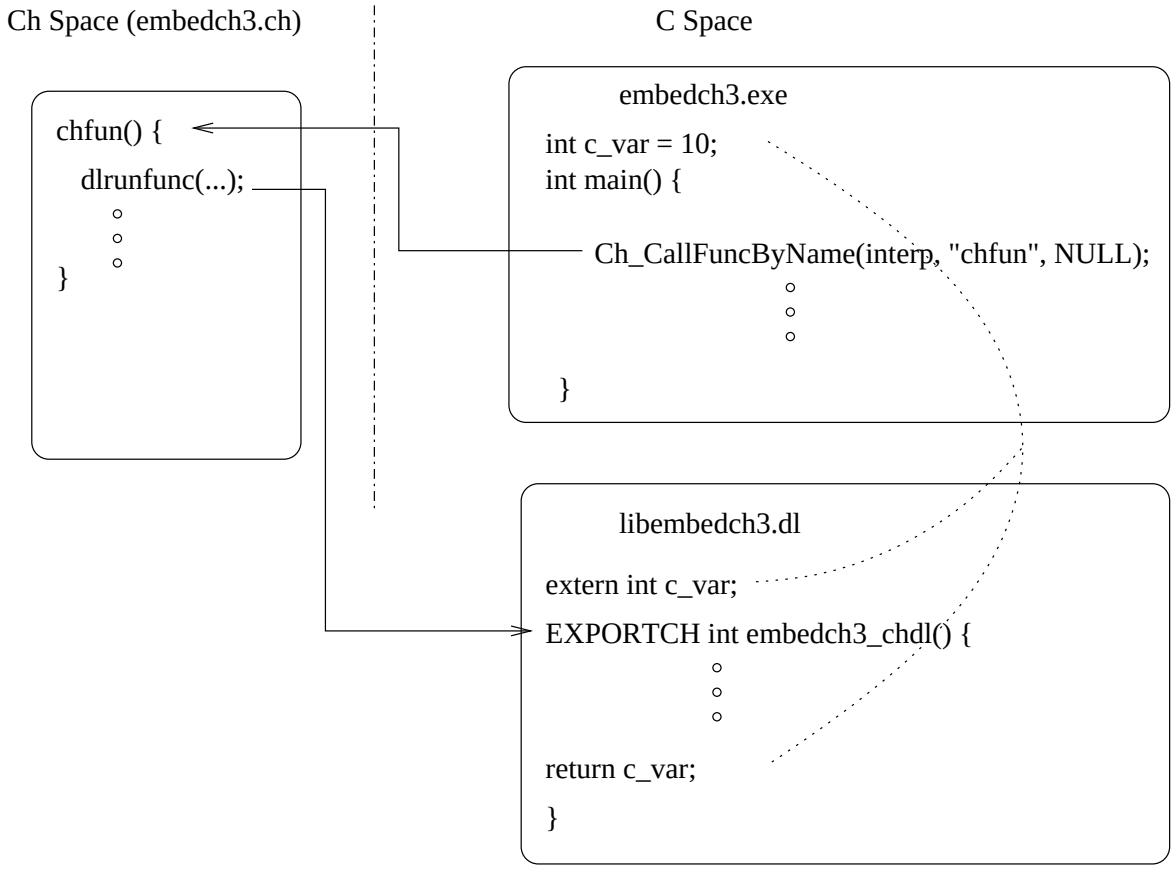


Figure 4.2: Accessing Variables in C Space from Ch Space.

```
#include<stdio.h>
#include<string.h>
#include<embedch.h>

EXPORTCH int c_var = 10;

int main () {
  ChInterp_t interp;
  int status;
  char *argvv[]={"embedch3.ch", NULL};

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp,argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program embedch3.ch failed\n");

  printf("in embedch3.c, &c_var = %p, c_var = %d\n", &c_var, c_var);
  Ch_CallFuncByName(interp, "chfun", NULL);
  printf("in embedch3.c, &c_var = %p, c_var = %d\n", &c_var, c_var);

  c_var = 100; /* change the variable value in C space */
  printf("\nin embedch3.c, &c_var = %p, c_var = %d\n", &c_var, c_var);
  Ch_CallFuncByName(interp, "chfun", NULL);
  printf("in embedch3.c, &c_var = %p, c_var = %d\n", &c_var, c_var);

  Ch_End(interp);

  return 0;
}
```

Program 4.10: Example of accessing variables in C Space from Ch space (embedch3.c).

```
#include<dlfcn.h>

void  chfun() {
  void *handle, *fptr;
  int ch_var, *ch_varptr;

  handle = dlopen("libembedch3.dl", RTLD_LAZY);
  if(handle == NULL) {
     printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
     return;
  }

  fptr = dlsym(handle, "embedch3_chdl");
  if(fptr == NULL)  {
     printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
     return;
  }
  dlrunfun(fptr, &ch_var, NULL, &ch_varptr);
  if(dlclose(handle)!=0) {
     printf("Error: %s(): dlclose(): %s\n", __func__, dlerror());
     return;
  }

  printf("in the Ch function, ch_var = %d\n", ch_var);
  *ch_varptr += 10;
}
```

Program 4.11: Example of accessing variables in C Space from Ch space (embedch3.ch).

```
#include<ch.h>

IMPORTCH int c_var;

EXPORTCH int embedch3_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int **c_varptr2;

  Ch_VaStart(interp, ap, varg);
  c_varptr2 = Ch_VaArg(interp, ap, int **);
  *c_varptr2 = &c_var;
  printf("in embedch3_chdl.c: &c_var = %p, c_var = %d\n", &c_var, c_var);
  Ch_VaEnd(interp, ap);
  return c_var;
}
```

Program 4.12: Example of accessing variables in C Space from Ch space (embedch3_chdl.c).

```
in embedch3.c, &c_var = 20d58, c_var = 10
in embedch3_chdl.c: &c_var = 20d58, c_var = 10
in the Ch function, ch_var = 10
in embedch3.c, &c_var = 20d58, c_var = 20

in embedch3.c, &c_var = 20d58, c_var = 100
in embedch3_chdl.c: &c_var = 20d58, c_var = 100
in the Ch function, ch_var = 100
in embedch3.c, &c_var = 20d58, c_var = 110
```

Figure 4.3: Output from executing `embedch3.c`.

Chapter 5

Calling C/C++ Functions from Ch Space

At the beginning of this document, Programs 1.1 illustrates how an embedded Ch program can be invoked in a C application program. The Ch program can in turn call functions in C space. This chapter describes methods of how to call C functions and invoke C++ class from a Ch program.

Ch SDK User's Guide described the details on how to interface C functions and C++ classes in static or dynamical libraries in C space from Ch space when the starting program is a Ch program. If the C/C++ libraries are not part of the main application program, please refer to *Ch SDK User's Guide*. You will need to create a separate dynamically loaded library such as `libsample.dl` for interfacing with C/C++ library.

However, no separate dynamically loaded library `.dl` file is needed for a Ch script to call C functions or C++ classes in the main application program. These C functions or C++ classes can be either statically linked to or dynamically loaded for the main application. Section 5.1 illustrates how to call C functions which are part of the main application program. The source code including Visual .NET project file `embedfuncmain.vcproj`, and VC++ project files `embedfuncmain.dsw` and `embedfuncmain.dsp` for the example described in section 5.1 can be found in in the directory `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain` File `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain/readme` has more detailed information about setup for compiling programs in Visual .NET and VC++. Treating a binary C function as a system function in the Ch space, section 5.2 illustrates how to call C functions even without using function files from a Ch program. The source code for this section is available in the directory `CHHOME/toolkit/demos/embedch/chapters/chapter5/declarefunc`. Section 5.3 shows how to call back a binary C function inside an application from the Ch space. The source code for this section is available in the directory `CHHOME/toolkit/demos/embedch/chapters/chapter5/callback`. Section 5.4 illustrates calling C functions from Ch space using a dynamically loaded library `libsample.dl` file.

Invoking C++ classes in the main application program from a Ch script without a separate `.dl` file is illustrated in section 5.5. The source code including Visual .NET and VC++ project files for the example described in section 5.5 are located in `CHHOME/toolkit/demos/embedch/chapters/chapter5/embedclassmain`. Section 5.6 illustrates invoking C++ classes from Ch space using a dynamically loaded library `libsample.dl` file.

5.1 Calling C Functions in the Main Program from Ch Space

Program 5.1 invokes a Ch script `program.ch` shown in Program 5.2. The output of executable compiled from Program 5.1 is shown in Figure 5.1. Program 5.2 calls two C functions `func1()` and `func2()`

```
/* File name: prog.c */

#include <stdio.h>
#include <embedch.h>
#include "sample.h"

int main() {
    ChInterp_t interp;
    char *argvv[]={"program.ch", NULL};
    int status;

    /* Call func1() in func.c */
    double d;
    d = func1(3.0);
    printf("func1(3.0) in prog.c = %f\n", d);

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch program */
    status = Ch_RunScript(interp, argvv);
    if(status)
      printf("Error: execution of program.ch failed\n");

    Ch_End(interp);
}
```

Program 5.1: C program invoking embedded Ch program (prog.c).

```
/* File name: program.ch */
#include <stdio.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);    // call C binary function func1()
   d2 = func2(3, 4);   // call C binary function func2()
   printf("func1(3) = %f\n", d1);
   printf("func2(3,4) = %f\n", d2);
}
```

Program 5.2: Ch Program (program.ch).

```
func1(3.0) in prog.c = 0.282240
sin(3) = 0.141120
hypot(3,4) = 5.000000
func1(3) = 0.282240
func2(3,4) = 10.000000
```

Figure 5.1: Output of executable compiled from Program 5.1.

```
#include "sample.h"

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

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

Program 5.3: C function file (func.c).

shown in Program 5.3, as well as two mathematical functions `sin()` and `hypot()` declared in the standard C library header file **math.h**. Program 5.2 is the same as one described in detail in Chapter 3 of *Ch SDK User's Guide*. In that example, C functions `func1()` and `func2()` are located in a binary library. Therefore, a separate dynamically loaded library `libsample.dl` needs to be created to interface these binary C functions.

In a typical application with Embedded Ch, C functions such as `func1()` and `func2()`, which might be in a static or dynamical library, are located inside the executable binary program that invokes embedded Ch programs. To call these C functions from a Ch script, these functions shall be exported so that Ch programs can callback these binary functions. We will demonstrates the mechanism for interface between Ch and C for such callback functions.

The procedure for calling C functions from the Ch space is still the same as one described in *Ch SDK User's Guide.* The binary module with interface wrapper is loaded first by function **dlopen**(). Then, the address of a function is obtained by function **dlsym**(). Finally, the function is invoked by function **dlrunfun**().

The dynamically loaded module is the application program itself in this case. The header file `sample.h` for both C and Ch programs is shown in in Program 5.4 The handler `_Chsample_handle` for the dynamically loaded module is obtained by the programming statement

```
_Chsample_handle = dlopen(NULL, RTLD_LAZY);
```

instead of

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

The first argument of function `dlopen()` is NULL. In HP-UX, the handler `_Chsample_handle` is NULL which is the default value in Ch when `_Chsample_handle` of pointer to void type is declared. The code which is Ch specific is included in the header file `sample.h` in Program 5.4 using the predefined system macro **_CH_**. 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.

Program 5.2 calls two Ch functions `func1()` and `func2()` through function files in Programs 5.5 and 5.6, respectively. Programs 5.5 and 5.6 each makes a call to its corresponding chdl function using **dlrunfun()**. Then the chdl file shown in Program 5.7 in turn calls the binary C functions. The macro in Program 5.7

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

```
#ifndef SAMPLE_H
#define SAMPLE_H
/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>


/*** in C/Ch space */
extern double func1(double x);
extern double func2(double x, double y);

#ifdef _CH_
#include <dlfcn.h>
void *_Chsample_handle;
#ifndef _HPUX_
_Chsample_handle = dlopen(NULL, 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
#endif


#endif /* SAMPLE_H */
```

Program 5.4: C/Ch header file (sample.h).

```
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 5.5: 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 5.6: Ch function file (func2.chf).

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**(). Macro **Ch_VaArg**() obtains arguments passed from the function **dlrunfun**() in a function file. These macros and functions are defined in the header file `ch.h` and described in detail in `Ch SDK User's Guide`.

Function files in Programs 5.5 and 5.6 should be placed in one of directories specified by the system variable **_fpath** for function file pathes as described in section 1.7. Function files shown in Programs 5.5 and 5.6, and C wrapper file shown in Program 5.7 can be created automatically from the header file `sample.h` by using utility program **c2chf** and functions **processfile()** and **processcfile()** as described in Chapter 3 of *Ch SDK User's Guide*.

```
#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 5.7: chdl file (sample_chdl.c).

```
#include <stdio.h>
#include <embedch.h>
#include "sample.h"

char *code = "\
  #include \"sample.h\" \
  double v = 4;         \
  double d;             \
  d = func2(3, v);";


int main() {
    ChInterp_t interp;
    int status;
    double *pd;

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch program */
    status = Ch_AppendRunScript(interp, code);
    if(status)
      printf("Error: execution of code failed\n");
    pd = Ch_SymbolAddrByName(interp,"d");
    printf("d in the Ch space = %f\n", *pd);
    Ch_End(interp);
}
```

Program 5.8: C program invoking scripts in the memory (prog1.c).

Program 5.1 invokes script `program.ch` in Program 5.2. However, scripts in the Ch space can also be invoked in the memory in the C space without a separate script file. Program 5.8 invokes the script below

```
#include "sample.h"
double v = 4;
double d;
d = func2(3, v);
```

directly in the memory by **Ch_AppendRunScript**(). Like Program 5.2, function `func2()` in the Ch space calls the corresponding binary function in the C space. The value for the global variable `d` in the Ch space is then obtained by **Ch_SymbolAddrByName**(). The output from executing Program 5.8 is as follows.

```
d in the Ch space = 10.000000
```

C header file `sample.h`, and program files `func.c`, `sample_chdl.c`, and `prog.c` are compiled and linked to form an executable binary program `prog.exe` so that function `func1()` can be called inside program `prog.c`. The process for compiling and linking of programs can be automated by a Makefile program. Makefiles in Programs 5.9 and 5.10 can be used to create executable program `prog.exe` used in this example for Unix and Windows, respectively, using utility programs `dlcomp` and `dllink` provided in Ch SDK for portable compilation and linking. Similarly, program `prog1.c` is compiled and linked against `func.c` and `sample_chdl.c` to create a binary executable program `prog1.exe`.

Makefile in Program 5.11 for Windows can also be used to create programs `prog.exe` and `prog1.exe` without using programs `dlcomp` and `dllink`.

If application programs such as `prog.exe` are created using other automation process such as integrated development environment, compile option -DWINDOWS, -DLINUX, -DLINUXPPC, -DSOLARIS, -DHPUX, -DDARWIN, -DFREEBSD, -DQNX, -DAIX should be added when programs such as `sample_chdl.c` with exported functions are compiled for Windows, Linux, Solaris, HP-UX, Mac OS X, FreeBSD, QNX, AIX, respectively. The binary objects should be linked with libraries `chsdk.lib and embedch.lib` in Windows as shown in Program 5.11 or `libchsdk.a and libembedch.a` in other platforms to form program `prog.exe`.

```
target: prog.exe prog1.exe

prog.exe: prog.o func.o sample_chdl.o
        ch dllink prog.exe -embedded prog.o func.o sample_chdl.o
prog.o: prog.c
        ch dlcomp libsample.dl prog.c
prog1.exe: prog1.o func.o sample_chdl.o
        ch dllink prog1.exe -embedded prog1.o func.o sample_chdl.o
prog1.o: prog1.c
        ch dlcomp libsample.dl prog1.c
func.o: func.c
        ch dlcomp libsample.dl func.c
sample_chdl.o:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I./
clean:
rm -f *.o *.exe
```

Program 5.9: Makefile to create prog.exe for Unix (Makefile).

```
target: prog.exe prog1.exe

prog.exe: prog.obj func.obj sample_chdl.obj
        ch dllink prog.exe -embedded prog.obj func.obj sample_chdl.obj
prog.obj: prog.c
        ch dlcomp libsample.dl prog.c
prog1.exe: prog1.obj func.obj sample_chdl.obj
        ch dllink prog1.exe -embedded prog1.obj func.obj sample_chdl.obj
prog1.obj: prog1.c
        ch dlcomp libsample.dl prog1.c
func.obj: func.c
        ch dlcomp libsample.dl func.c
sample_chdl.obj:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I./
clean:
        del *.obj
        del *.exp
        del *.lib
        del *.exe
```

Program 5.10: Makefile to create prog.exe for Windows (Makefile).

```
#assume CHHOME is C:/ch.
#If not, change C:/ch to CHHOME such as C:/usr/local/ch

CC= cl
LD= cl
INC=-IC:/ch/extern/include -I./
CFLAG=/MD
LFLAG=C:/ch/extern/lib/chsdk.lib \
      C:/ch/extern/lib/embedch.lib advapi32.lib

target: prog.exe prog1.exe

prog.exe: prog.obj func.obj sample_chdl.obj
        $(LD) -o prog.exe prog.obj func.obj sample_chdl.obj $(LFLAG)
prog.obj: prog.c
        $(CC) -c prog.c $(INC) $(CFLAG)
prog1.exe: prog1.obj func.obj sample_chdl.obj
        $(LD) -o prog1.exe prog1.obj func.obj sample_chdl.obj $(LFLAG)
prog1.obj: prog1.c
        $(CC) -c prog1.c $(INC) $(CFLAG)
func.obj: func.c
        $(CC) -c func.c $(INC) $(CFLAG)
sample_chdl.obj:  sample_chdl.c
        $(CC) -c sample_chdl.c $(INC) $(CFLAG)

clean:
rm -f *.o*.obj *.exe *.lib *.exp
```

Program 5.11: Makefile to create prog.exe for Windows (Makefile_Win).

The source code for examples described in this section are available in the directory CHHOME/toolkit/demos/embedch/chapters/chapter5/embedfuncmain/.

5.2 Calling C Functions in Main Program from Ch Without Function Files

In section 5.1, functions `func1()` and `func2()` are treated as regular functions using function files. To simplify the interface to the binary C function, these two functions can be treated as system functions in the Ch space and defined using the API **Ch_DeclareFunc**() with the following function calls

```
Ch_DeclareFunc(interp,"double func1(double x);",func1_chdl);
Ch_DeclareFunc(interp,"double func2(double x,double y);",func2_chdl);
```

as shown in Program 5.12. The second argument of the function **Ch_DeclareFunc**() is the function prototype in the C space. The function prototypes for wrapper functions `func1_chdl()` and `func2_chdl()` in Program 5.7 passed as the third argument of **Ch_DeclareFunc**() are placed in a header file `sample_chdl.h` in Program 5.13. This header file is included in Program 5.12.

Because function files `func1.chf` and `func2.chf` in Programs 5.5 and 5.6, respectively, are not needed in this case. The header file `sample.h` can be simplified as shown in Program 5.14. All other files including `program.ch, func.c, sample_chdl.c`, and makfiles in section 5.1 are the same. The source code for this section is available in the directory `CHHOME/toolkit/demos/embedch/chapters/chapter5/declarefunc`.

```
/* File name: prog.c */

#include <stdio.h>
#include <embedch.h>
#include "sample.h"
#include "sample_chdl.h"

int main() {
    ChInterp_t interp;
    char *argvv[]={"program.ch", NULL};
    int status;

    /* Call func1() in func.c */
    double d;
    d = func1(3.0);
    printf("func1(3.0) in prog.c = %f\n", d);

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* declare functions func1() and func2() in the Ch space */
    Ch_DeclareFunc(interp, "double func1(double x);", func1_chdl);
    Ch_DeclareFunc(interp, "double func2(double x, double y);", func2_chdl);

    /* run a Ch program */
    status = Ch_RunScript(interp, argvv);
    if(status)
      printf("Error: execution of program.ch failed\n");
    Ch_End(interp);
}
```

Program 5.12: C program invoking embedded Ch program calling C function using **Ch_DeclareFunc**() (prog.c).

```
/* File name: sample_chdl.h */

#ifndef SAMPLE_CHDL_H
#define SAMPLE_CHDL_H
#include <ch.h>

EXPORTCH double func1_chdl(void *varg);
EXPORTCH double func2_chdl(void *varg);

#endif /* SAMPLE_CHDL_H */
```

Program 5.13: C header file (sample_chdl.h).

```
/* File name: sample.h */

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

#if !defined(_CH_)
/*** in C space */
extern double func1(double x);
extern double func2(double x, double y);
#endif

#endif /* SAMPLE_H */
```

Program 5.14: C/Ch header file (sample.h).

5.3 Calling Back C Functions in the Main Program from Ch Space

In this section, callback C function in the main program from a function in the Ch space will be demonstrated using examples. In Program 5.15, the script in Program 5.16 is executed first. The script contains a function `printdata()` in the Ch space Without the main() function or function printdata() is not called explicitly, this function will not be executed when the script is invoked by function **Ch_RunScript**() in Program 5.15.

The function `printdata()` in the Ch space is called explicitly by the function **Ch_CallFuncByName**() in the C space. The function returns an integer of type int. The values for two arguments of the function with `num` of type int and `data` of array of double are passed from the function **Ch_CallFuncByName**(). The function `printdata()` in the Ch space calls the same function in the C space inside Program 5.15 through a wrapper function `printdata_chdl()`.

The header file `sample.h` in Program 5.17 obtains the handle for dynamically loaded module needed inside the function `printdata()` in the Ch space. If there are multiple functions, the definitions for functions, such as `printdata()` in the Ch space, can be placed in function files.

The output from execution of Program 5.15 is displayed in Figure 5.2.

In Programs 5.15 and 5.16, the callback function from Ch space is fixed. In some applications, it might be desirable to call different callback functions in the C space from a single Ch function. This next example illustrates the technique for handling such a callback. Program 5.18 loads the script in Program 5.19 first. When the function `callback()` is invoked explicitly by the function **Ch_CallFuncByName**() in the C space. The actual callback function `printlist()` in the C space is pushed into the Ch space as an argument `func` in the function `callback()`. Note that since the argument `func` is a pointer to function in the C space and shall not be called directly in the Ch space, it is treated as a variable of pointer to void. It cannot be declared as a pointer to function in Ch space. This pointer to function is passed back to the wrapper function `callback_chdl()`. When this callback function is invoked in the function `callback_chdl()`, it actually calls the function `printlist()` in the C space passed from an argument of the function **Ch_CallFuncByName**(). Different callback functions can be pushed into the Ch space by the function **Ch_CallFuncByName**(). Unlike function `printdata()` in Programs 5.15, function `printlist()` has no return value. It illustrates how arrays of string are handled through its argument.

The output from execution of Program 5.18 is displayed in Figure 5.3.

The source code for examples described in this section are available in the directory CHHOME/toolkit/demos/embedch/chapters/chapter5/callback.

The C functions can be defined using **Ch_DeclareFunc**() as system functions in the Ch space as de-

```
/* File: prog1.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>
#include "sample.h"

double data[4] = {10.0, 20.0, 30.0, 40.0};

int main( int argc, char *argv[] ) {
  ChInterp_t interp;
  int status;
  int num, retval;
  char *argvv[] = {"embedch1.ch", NULL};

  Ch_Initialize(&interp, NULL);
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR) {
     printf("Error: Ch_RunScript() failed\n");
     exit(1);
  }
  num = sizeof(data)/sizeof(double);
  printf("calling the printdata() in Ch\n");
  status = Ch_CallFuncByName( interp, "printdata", &retval,  num, data);
  printf("retval = %d\n", retval);
  printf("after calling the printdata() in Ch\n");
  if(status == CH_ERROR) {
     printf("Error: Ch_CallFuncByName() failed\n");
     exit(1);
  }
  Ch_End(interp);
}

int printdata(int num, double *data) {
  int i;

  for(i=0; i<num; i++)
     printf("data[%d] = %f\n", i, data[i]);
  return 100;
}

/* this function is called from the Ch space */
EXPORTCH int printdata_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int num, retval;
  double *data;

  Ch_VaStart(interp, ap, varg);
  num = Ch_VaArg(interp, ap, int);
  data = Ch_VaArg(interp, ap, double *);
  printf("data in printdata() in C is %p\n", data);
  retval = printdata(num, data);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 5.15: A C program calling back a C function from a Ch function (prog1.c).

```
/* File: embedch1.ch */
#include<stdio.h>
#include "sample.h"

/* printdata() can be placed in function file printdata.chf */
int printdata(int num, double *data) {
    void *fptr;
    int retval;

    fptr = dlsym(_Chsample_handle, "printdata_chdl");
    if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return -1;
    }
    printf("data in printdata() in Ch is %p\n", data);
    dlrunfun(fptr, &retval, printdata, num, data);
    return retval;
}

/* the function main() is optional
int main() {
    double data[4] = {1.0, 2.0, 3.0, 4.0};

    printdata(4, data);
    return 0;
}
*/
```

Program 5.16: Ch program (embedch1.ch).

```
/* this header file will initialize _Chsample_handle */
#ifndef sample_H
#define sample_H

#ifdef _CH_
#include <dlfcn.h>
void *_Chsample_handle;
#ifndef _HPUX_
_Chsample_handle = dlopen(NULL, 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
#endif


extern int printdata(int num, double *data);

#endif
```

Program 5.17: C/Ch header file (sample.h).

```
calling the printdata() in Ch
data in printdata() in Ch is 26960
data in printdata() in C is 26960
data[0] = 10.000000
data[1] = 20.000000
data[2] = 30.000000
data[3] = 40.000000
retval = 100
after calling the printdata() in Ch
```

Figure 5.2: Output of executable compiled from Program 5.15.

scribed in the section 5.2. No function file is needed to call a binary function using its corresponding system function in the Ch space. Then, Programs 5.15 and 5.16 can be modified as shown in Programs 5.20 and 5.21, respectively. Program 5.20 defines system function printdata() using the function call below.

```
Ch_DeclareFunc(interp, "int printdata(int num, double *data);",
                        printdata_chdl);
```

Similarly, Program 5.18 can be replaced by Program 5.22, which contains a function call

```
Ch_DeclareFunc(interp, "void callback(void *func, char **list);",
                       callback_chdl);
```

```
/* File: prog2.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>

char *list[] = {"string1", "string2", NULL};
void printlist(char **list);
typedef void (*FUNCPTR)(char **);

int main( int argc, char *argv[] )
{
  ChInterp_t interp;
  ChOptions_t option;
  int status;
  char *argvv[] = {"embedch2.ch", NULL};

  Ch_Initialize(&interp, NULL);
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR) {
    printf("Error: Ch_ParseScript() failed\n");
    exit(1);
  }

  printf("calling the callback()\n");
  status = Ch_CallFuncByName( interp, "callback", NULL, printlist, list);
  printf("after calling the callback()\n");
  if(status == CH_ERROR) {
    printf("Error: Ch_CallFuncByName() failed\n");
    exit(1);
  }
  /* status = Ch_CallFuncByName( interp, "callback", NULL, otherFunc, list); */

  Ch_End(interp);
}

void printlist(char **list) {
  int i;

  printf("list in printlist() in C is %p\n", list);
  for( i=0; list[i]; i++)
     printf("list[%d] = \"%s\"\n", i, list[i]);
}

/* this function is called from the Ch space */
EXPORTCH void callback_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  void (*callback)(char**);
  char **list;

  Ch_VaStart(interp, ap, varg);
  callback = Ch_VaArg(interp, ap, FUNCPTR);
  list = Ch_VaArg(interp, ap, char**);
  printf("list in callback_chdl() in C is %p\n", list);

  /* now we call the real function */
  callback(list);
  Ch_VaEnd(interp, ap);
}
```

Program 5.18: A C program calling back a C function from a Ch function (prog2.c).

```
/* File: embedch2.ch */
#include <stdio.h>
#include "sample.h"

// this prototype will not work because of the internal checking
// void callback(void (*func)(char **), char **list)
void callback(void *func, char **list) {
   void *fptr;
   fptr = dlsym(_Chsample_handle, "callback_chdl");
   if(fptr == NULL) {
        fprintf(_stderr, "Error: %s(): dlsym(): %s\n", __func__,  dlerror());
        return;
    }
    printf("list in callback() in Ch is %p \n", list);
    dlrunfun(fptr, NULL, callback, func, list);
}
```

Program 5.19: Ch program (embedch2.ch).

```
calling the callback()
list in callback() in Ch is 26930
list in callback_chdl() in C is 26930
list in printlist() in C is 26930
list[0] = "string1"
list[1] = "string2"
after calling the callback()
```

Figure 5.3: Output of executable compiled from Program 5.18.

```
/* File: prog3.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>
#include "sample.h"
EXPORTCH int printdata_chdl(void *varg);

double data[4] = {10.0, 20.0, 30.0, 40.0};

int main( int argc, char *argv[] ) {
  ChInterp_t interp;
  int status;
  int num, retval;
  char *argvv[] = {"embedch3.ch", NULL};

  Ch_Initialize(&interp, NULL);
  Ch_DeclareFunc(interp, "int printdata(int num, double *data);",
                 (ChFuncdl_t)printdata_chdl);
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR) {
     printf("Error: Ch_RunScript() failed\n");
     exit(1);
  }
  num = sizeof(data)/sizeof(double);
  printf("calling the printdata() in Ch\n");
  status = Ch_CallFuncByName( interp, "printdata", &retval,  num, data);
  printf("retval = %d\n", retval);
  printf("after calling the printdata() in Ch\n");
  if(status == CH_ERROR) {
     printf("Error: Ch_CallFuncByName() failed\n");
     exit(1);
  }
  Ch_End(interp);
}

int printdata(int num, double *data) {
  int i;

  for(i=0; i<num; i++)
     printf("data[%d] = %f\n", i, data[i]);
  return 100;
}

/* this function is called from the Ch space */
EXPORTCH int printdata_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int num, retval;
  double *data;

  Ch_VaStart(interp, ap, varg);
  num = Ch_VaArg(interp, ap, int);
  data = Ch_VaArg(interp, ap, double *);
  printf("data in printdata() in C is %p\n", data);
  retval = printdata(num, data);
  Ch_VaEnd(interp, ap);
  return retval;
}
```

Program 5.20: A C program calling back a C function from a Ch function without a function file (prog3.c).

```
/* File: embedch3.ch */
#include<stdio.h>
#include "sample.h"


/* the function main() is optional */
int main() {
    double data[4] = {1.0, 2.0, 3.0, 4.0};

    printdata(4, data);
    return 0;
}
```

Program 5.21: Ch program (embedch3.ch).

5.4 Calling C Functions in the Main Program from Ch Space with .dl File

In the example in section 5.1, file `sample_chdl.c` with interface functions shown in Program 5.7 is compiled and linked with the application program without a separate dynamically loaded library .dl file.

An alternative method with a dynamically loaded library .dl file has been used in the past. Although this method has been deprecated, it is described in this section using the same example presented in section 5.1. Functions `func1()` and `func2()` called from Ch space are located inside the executable binary program that invokes embedded Ch programs. In this case, these functions shall be exported so that Ch programs can callback these binary functions. The example below demonstrates the mechanism for interface between Ch and C for such callback functions using a dynamically loaded library. Because this example uses the same functions in section 5.1, we only list programs that are different from the ones in the example in section 5.1. In this example, function files `func1.chf` and `func2.chf`, interface file `sample_chdl.c`, Ch script `program.ch`, application program `prog.c` are the same as before. Program 5.23 for `func.c` contains two functions `func1()` and `func2()`, which are exported. To export a function for callback from Ch scripts, the function shall be defined and prototyped with type qualifier `EXPORTCH` as shown in Program 5.23. The corresponding header file `sample.h` applicable in both C and Ch spaces is shown in Program 5.24. File `func.c` is compiled and linked with Program 5.1 for program `prog.c` to form an executable binary program so that function `func1()` can be called inside program `prog.c`. Unlike the example in section 5.1, the handler `_Chsample_handle` for the dynamically loaded module is obtained by the programming statement

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

which loads the module explicitly. The statement

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

loads the dynamically loaded library when the code is parsed.

The process for compiling and linking of programs can be automated by a Makefile program. Makefiles in Programs 5.25 and 5.26 can be used to create executable program `prog.exe` and dynamically loaded library `libsample.dl`, used in this example for Unix, respectively. The similar makefiles in in Programs 5.27 and 5.28 can be used for Windows.

Similar to calling C functions in a library, for command **dllink**, the option `-embedded` is required to make dynamically loaded library .dl file so that some libraries will be included by default. 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.

```
/* File: prog4.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>
EXPORTCH void callback_chdl(void *varg);
char *list[] = {"string1", "string2", NULL};
void printlist(char **list);
typedef void (*FUNCPTR)(char **);

int main( int argc, char *argv[] )
{
  ChInterp_t interp;
  ChOptions_t option;
  int status;
  char *argvv[] = {"embedch4.ch", NULL};

  Ch_Initialize(&interp, NULL);
  Ch_DeclareFunc(interp, "void callback(void *func, char **list);",
                 (ChFuncdl_t)callback_chdl);
  /* status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR) {
    printf("Error: Ch_ParseScript() failed\n");
    exit(1);
  } */

  printf("calling the callback()\n");
  status = Ch_CallFuncByName( interp, "callback", NULL, printlist, list);
  printf("after calling the callback()\n");
  if(status == CH_ERROR) {
    printf("Error: Ch_CallFuncByName() failed\n");
    exit(1);
  }
  /* status = Ch_CallFuncByName( interp, "callback", NULL, otherFunc, list); */

  Ch_End(interp);
}

void printlist(char **list) {
  int i;

  printf("list in printlist() in C is %p\n", list);
  for( i=0; list[i]; i++)
     printf("list[%d] = \"%s\"\n", i, list[i]);
}

/* this function is called from the Ch space */
EXPORTCH void callback_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  void (*callback)(char**);
  char **list;

  Ch_VaStart(interp, ap, varg);
  callback = Ch_VaArg(interp, ap, FUNCPTR);
  list = Ch_VaArg(interp, ap, char**);
  printf("list in callback_chdl() in C is %p\n", list);

  /* now we call the real function */
  callback(list);
  Ch_VaEnd(interp, ap);
}
```


Program 5.22: A C program calling back a C function from a Ch function without a function file (prog4.c).

If applications programs such as prog.exe are created using other automation process such as integrated development environment, compile option -DWINDOWS, -DLINUX, -DLINUXPPC, -DSOLARIS, -DHPUX, -DDARWIN, -DFREEBSD, -DQNX, -DAIX should be added when programs such as `func.c` with exported functions are compiled for Windows, Linux, Solaris, HP-UX, Mac OS X, FreeBSD, QNX, AIX, respectively. The binary objects should be linked with library `embedch.lib` in Windows or `libembedch.a` in other platforms to form program `prog.exe`.

In Windows, library `prog.lib` will be generated at the same time when `prog.exe` is created. This library with symbols for exported functions shall be used to create dynamically loaded library `libsample.dl` as shown in Program 5.28.

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

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

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

Program 5.23: C function file (func.c).

```
#ifndef SAMPLE_H
#define SAMPLE_H
/* func1() and func2() use sin() and hypot() in math.h */
#include <math.h>


/*** in Ch space */
#ifdef _CH_
extern double func1(double);
extern double func2(double, double);

#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);

/*** in C space */
#else
#include <ch.h>
EXPORTCH extern double func1(double x);
EXPORTCH extern double func2(double x, double y);
#endif


#endif /* SAMPLE_H */
```

Program 5.24: C/Ch header file (sample.h).

```
target: prog.exe

prog.exe: prog.o func.o
        ch dllink prog.exe -embedded prog.o func.o
prog.o: prog.c
        ch dlcomp libfunc.dl prog.c
func.o: func.c
        ch dlcomp libfunc.dl func.c

clean:
        rm *.o *.dl *.exe
```

Program 5.25: Makefile to create prog.exe for Unix (Makefile_exe).

```
target: libsample.dl

libsample.dl: sample_chdl.o
        ch dllink libsample.dl sample_chdl.o
sample_chdl.o:  sample_chdl.c
        ch dlcomp libsample.dl sample_chdl.c -I./
clean:
        rm *.o *.dl *.exe
```

Program 5.26: Makefile to create libsample.dl for Unix (Makefile_dl).

```
target: prog.exe

prog.exe: prog.obj func.obj
       ch dllink prog.exe -embedded prog.obj func.obj
prog.obj: prog.c
       ch dlcomp libfunc.dl prog.c
func.obj: func.c
       ch dlcomp libfunc.dl func.c

clean:
       del *.obj *.exp *.lib *.dl *.exe
```

Program 5.27: Makefile to create prog.exe for Windows (Makefile_exe).

```
target: libsample.dl

libsample.dl: sample_chdl.obj
       ch dllink libsample.dl -embedded sample_chdl.obj prog.lib
sample_chdl.obj:  sample_chdl.c
       ch dlcomp libsample.dl sample_chdl.c -I./
clean:
       del *.obj *.exp *.lib *.dl *.exe
```

Program 5.28: Makefile to create libsample.dl for Windows (Makefile_dl).

5.5 Invoking C++ Classes in the Main Program from Ch Space

Interface Ch with classes in C++ located in a library has been described in Chapter 7 *Interfacing Classes and Member Functions in C++* of *Ch SDK User's Guide*. Definitions of member functions for classes are compiled and linked to form dynamically loaded library with file extension `.dl`. In many applications with embedded Ch, class definitions in C++ space invoked from Ch space are located inside the executable binary program that invokes embedded Ch programs. In this case, the member functions of classes in C++ space shall be exported so that Ch programs can callback these binary member functions.

Like interface C functions in an application program described in the previous sections, C++ classes in an application can be invoked in Ch scripts with or without a dynamically loaded library in .dl file. The example in this section below demonstrates the mechanism for interface between Ch and C++ for such

callback member functions without a dynamically loaded library. An example using a dynamically loaded library will be presented in next section.

This example is similar to the example described in section 4 of Chapter 7 in *Ch SDK User's Guide*. Program 5.29 contains the definition of class `Class1` with a constructor `Class1::Class1()` a destructor, `Class1::~Class1()`, and a member function `Class1::memfun1()` in C++ space. The definitions for the constructor, destructor, and member function are defined in Program 5.30.

Member functions can be exported for callback from Ch scripts. Head file `sampclass.h` in Program 5.29 is shared by both Ch and C++. The predefined macro `_CH_` is used to distinguish the code block used for these two different execution and compilation environments. This program is compiled and linked with Program 5.32 for program `prog.cpp` to form an executable binary program.

Class file `sampclass.cpp` in Program 5.30, interface functions `sampclass_chdl.cpp` in Program 5.31, and application program program `prog.cpp` in Program 5.32 with C++ header file `sampclass.h` in Program 5.29 are compiled and linked to create an executable program `prog.exe`. A Makefile program. Program 5.33 and 5.34 can be used to create the executable program `prog.exe` used in this example for Unix and Windows, respectively.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

/********* for Ch space ***********/
#ifdef _CH_
#include <dlfcn.h>
#ifdef _HPUX_
void *g_sample_dlhandle;
#else
extern void *g_sample_dlhandle = dlopen(NULL, 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
#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_ */
```

Program 5.29: C++/Ch header file (sampclass.h).

```
#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 in Class1::memfun1() = %d\n", m_i1);
  return m_i1;
}
```

Program 5.30: C++ class member functions file (sampclass.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;
}
```

Program 5.31: chdl file (sampclass_chdl.cpp).

```
/* File: prog.cpp */
#include <stdio.h>
#include <embedch.h>
#include "sampclass.h"

int main() {
    ChInterp_t interp;
    char *argvv[]={"script.cpp", NULL};
    int status;
    class Class1 c1;

    printf("Begin output from prog.exe:\n");
    c1.memfun1(10);
    printf("End output from prog.exe:\n");

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch program */
    status = Ch_RunScript(interp, argvv);
    if(status)
      printf("Error: execution of script.cpp failed\n");
    Ch_End(interp);
    return 0;
}
```

Program 5.32: C++ program invoking embedded Ch programs (prog.cpp).

```
target: prog.exe

prog.exe: prog.o sampclass.o sampclass_chdl.o
        ch dllink prog.exe cplusplus -embedded prog.o sampclass.o sampclass_chdl.o
prog.o: prog.cpp
        ch dlcomp libclass.dl cplusplus prog.cpp
sampclass.o: sampclass.cpp
        ch dlcomp libclass.dl cplusplus sampclass.cpp
sampclass_chdl.o: sampclass_chdl.cpp
        ch dlcomp libclass.dl cplusplus sampclass_chdl.cpp

clean:
        rm -f *.o *.obj *.dl *.exe *.lib *.exp
```

Program 5.33: Makefile to create prog.exe for Unix (Makefile).

```
target: prog.exe

prog.exe: prog.obj sampclass.obj sampclass_chdl.obj
        ch dllink prog.exe cplusplus -embedded prog.obj sampclass.obj sampclass_chdl.obj
prog.obj: prog.cpp
        ch dlcomp libclass.dl cplusplus prog.cpp
sampclass.obj: sampclass.cpp
        ch dlcomp libclass.dl cplusplus sampclass.cpp
sampclass_chdl.obj: sampclass_chdl.cpp
        ch dlcomp libclass.dl cplusplus sampclass_chdl.cpp

clean:
        rm -f *.o *.obj *.dl *.exe *.lib *.exp
```

Program 5.34: Makefile to create prog.exe for Windows (Makefile).

Similar to C programs, if applications programs in C++ such as `prog.exe` are created using other automation process such as integrated development environment, compile option -DWINDOWS, -DLINUX, -DLINUXPPC, -DSOLARIS, -DHPUX, -DDARWIN, -DFREEBSD, -DQNX, -DAIX should be added when programs such as `sampclass_chdl.cpp` with exported functions are compiled for Windows, Linux, Solaris, HP-UX, Mac OS X, FreeBSD, QNX, AIX, respectively. The binary objects should be linked with library `embedch.lib` in Windows or `libembdch.a` in other platforms to form program `prog.exe`.

Program 5.35 is the Ch program we want to invoke from C+ space with embedded Ch. It includes header file `sampclass.h` in Program 5.29 in Ch space. The definitions of the constructor, destructor, and member function for class `Class1` in Ch space defined in Program 5.36 is included by header file `sampclass.h` using the directive

```
#pragma importf <sampclass.chf>
```

Class member function file `sampclass.chf` should be located in a directory specified in the system variable **_fpath** for function files.

Like interface C functions in an application program from Ch scripts, the handler `g_sample_dlhandle` for the dynamically loaded module is obtained by the programming statement

```
g_sample_dlhandle = dlopen(NULL, RTLD_LAZY);
```

However, the dynamical loading of a module can be placed inside in a constructor, instead of in a header file, as shown in Program 5.36. Similarly, the loading of the program itself with the value of NULL for handler `g_sample_dlhandle` in HP-UX is treated differently from other platforms.

When a class is instantiated at the file scope, the constructor will be invoked at the parsing stage. Therefore, the constructor `Class1::Class1()` will open the dynamically loaded library `libsampclass.dl` and get the address of its binary counter part using generic built-in functions `dlopen()` and `dlsym()`, respectively. The constructor `Class1::Class1()`, destructor `Class1::~Class1()`, and member function `Class1::memfun1()` in Program 5.36 each calls its corresponding chdl function using **dlrunfun()**. Then the chdl file shown in Program 5.31 in turn calls the binary C++ member functions.

The output of executable compiled from Program 5.32 is shown in Figure 5.4.

```
/* File: script.cpp */
#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;
}
```

Program 5.35: Ch Program (script.cpp).

```
/****** 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;
}
```

Program 5.36: Ch member function file (sampclass.chf).

```
m_i1 in Class1::Class1() = 1
Begin output from prog.exe:
m_i1 in Class1::memfun1() = 11
End output from prog.exe:
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 in Class1::memfun1() = 101
m_i1 in Class1::memfun1() = 201
m_i1 in Class1::memfun1() = 301
m_i1 in Class1::memfun1() = 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
m_i1 in Class1::~Class1() = 11
```

Figure 5.4: Output of executable compiled from Program 5.32.

Program 5.32 runs a Ch script in Program 5.35 with classes declared in the script. But, the actual definition of the class is defined inside C++, although there is a definition of the same class with constructor, destructor, and member functions only in Ch. In some applications, it may be desirable to invoke a class both defined and declared inside C++. Program 5.37 and Program 5.38 illustrate such an application scenario.

Program 5.38 contains a function `func()` which has two arguments, one is a pointer to class and the other is an integer. Header file `sampclass.h` used in Program 5.38 is the same as the one used in Program 5.35, Header file Program 5.29 with the same class definition defined in Program 5.36 is used in Program 5.38. Program 5.37 runs the Ch script in Program 5.38 by the API **Ch_RunScript**(). Afterward, the function `func()` in the Ch space in Program 5.38 is called by the API **Ch_CallFuncByName**(). The class `c1` and integer `i` defined inside Program 5.37 are passed to the function `func()`.

The output of executable compiled from Program 5.37 is shown in Figure 5.5.

```
/* File: prog2.cpp */
#include <stdio.h>
#include <embedch.h>
#include "sampclass.h"

int main() {
    ChInterp_t interp;
    char *argvv[]={"script2.cpp", NULL};
    int status, i=10, retval;
    class Class1 c1;

    c1.memfun1(i);
    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch program */
    status = Ch_RunScript(interp, argvv);
    if(status)
      printf("Error: execution of script2.cpp failed\n");
    Ch_CallFuncByName(interp, "func", &retval, &c1, i);
    printf("retval in prog2.cpp = %d\n", retval);
    Ch_End(interp);
    return 0;
}
```

Program 5.37: C++ program invoking embedded Ch programs (prog2.cpp).

```
/* File script2.cpp */
#include "sampclass.h"

int func(class Class1 *c, int i) {
    int j;

    j = c->memfun1(i);
    printf("i = %d j = %d in func() in script2.cpp\n", i, j);
    return j+i;
}
```

Program 5.38: Ch Program (script2.cpp).

```
m_i1 in Class1::Class1() = 1
m_i1 in Class1::memfun1() = 11
m_i1 in Class1::memfun1() = 21
i = 10 j = 21 in func() in script2.cpp
retval in prog2.cpp = 31
m_i1 in Class1::~Class1() = 21
```

Figure 5.5: Output of executable compiled from Program 5.37.

5.6 Invoking C++ Classes in the Main Program from Ch Space with .dl File

In the example in the previous section, file `sampclass_chdl.c` with interface functions shown in Program 5.31 is compiled and linked with the application program without a separate dynamically loaded library .dl file.

An alternative method with a dynamically loaded library .dl file has been used in the past. Although this method has been deprecated, it is described in this section using the same example presented in the previous section. Because this example uses the same functions in the previous section, we only list programs that are different from the ones in the previous example. In this example, C++ header file in Program 5.29, interface file `sampclass_chdl.c`, Ch script `script.ch`, application program `prog.cpp` are the same as before.

To export a member function for callback from Ch scripts, the member function shall be defined and prototyped with the type qualifier `EXPORTCHCLASS` as shown in Program 5.39 for header file `sampclass.h`. Program 5.40 for class definition file `sampclass.cpp` is compiled and linked with Program 5.32 for program `prog.cpp` to form an executable binary program. The process for compiling and linking of programs can be automated by a Makefile program. Program 5.41 can be used to create executable program `prog.exe` used in this example for Unix. The similar makefile in Programs 5.42 can be used for Windows.

```
#ifndef _SAMPCLASS_H_
#define _SAMPCLASS_H_

/********* for Ch space **********/
#ifdef _CH_
#include <dlfcn.h>
extern 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;
  public:
    EXPORTCHCLASS Class1();
    EXPORTCHCLASS ~Class1();
    EXPORTCHCLASS int memfun1(int i);
#else
  public:
    Class1();
    ~Class1();
    int memfun1(int i);
#endif
};

#pragma importf <sampclass.chf>

#endif /* _SAMPCLASS_H_ */
```

Program 5.39: Ch/C++ header file (sampclass.h).

```
#include <stdio.h>
#include <ch.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 in Class1::memfun1() = %d\n", m_i1);
  return m_i1;
}
```

Program 5.40: C++ class member functions file (sampclass.cpp).

```
target: prog.exe

prog.exe: prog.o sampclass.o
        ch dllink prog.exe cplusplus -embedded prog.o sampclass.o
prog.o: prog.cpp
        ch dlcomp libclass.dl cplusplus prog.cpp
sampclass.o: sampclass.cpp
        ch dlcomp libclass.dl cplusplus sampclass.cpp

clean:
        rm *.o *.dl *.exe
```

Program 5.41: Makefile to create prog.exe for Unix (Makefile_exe).

```
target: prog.exe

prog.exe: prog.obj sampclass.obj
        ch dllink prog.exe cplusplus -embedded prog.obj sampclass.obj
prog.obj: prog.cpp
        ch dlcomp libclass.dl cplusplus prog.cpp
sampclass.obj: sampclass.cpp
        ch dlcomp libclass.dl cplusplus sampclass.cpp

clean:
        rm *.obj *.dl *.exe *.lib
```

Program 5.42: Makefile to create prog.exe for Windows (Makefile_exe).

Unlike the example in the previous section, the handler `g_sample_handle` for the dynamically loaded

module is obtained by the programming statement

```
g_sample_dlhandle = dlopen("libsampleclass.dl", RTLD_LAZY);
```

which loads the module explicitly in Program 5.43. The statement

```
extern void *_Chsample_handle = dlopen("libsampleclass.dl", RTLD_LAZY);
```

loads the dynamically loaded library when the code is parsed.

To use an instance of a class in a global variable, one may declare the handle as a global variable in the Ch space as follows.

```
void *_Chsample_handle;
```

When a program is executed by API **Ch_RunScript**() or **Ch_ExecScript**(), the destructor of the class will delete the instance of a class in the global variable in the program. In this case, the library can be explicitly loaded in the Ch space as follows.

```
Ch_Initialize(interp, option);
Ch_ParseScript(interp, prog);
Ch_ExprEval(interp, "_Chsample_handle = dlopen(\"libsampleclass.dl\", RTLD_LAZY)");
Ch_ChCallFuncBYName(interp, Cplusfunc, retval, arg); // use an instance of class
```

Makefiles in Program 5.44 and Program 5.45 can be used to compile Program 5.31 for Unix and Windows, respectively, to create a dynamically loaded library `libsampclass.dl`. In Windows, the executable binary program `prog.exe` should be created first. The library `prog.lib` will be generated at the same time when `prog.exe` is created. This library with symbols for exported member functions shall be used to create dynamically loaded library `libsampclass.dl` as shown in Program 5.45.

Note that for command **dllink**, the option `-embedded` is required to make dynamically loaded library .dl file so that some libraries will be included by default.

```
/****** 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;
}
```

Program 5.43: Ch member function file (sampclass.chf).

```
target: libsampclass.dl

libsampclass.dl: sampclass_chdl.o
        ch dllink libsampclass.dl cplusplus sampclass_chdl.o

sampclass_chdl.o: sampclass_chdl.cpp
        ch dlcomp libsampclass.dl cplusplus sampclass_chdl.cpp

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

Program 5.44: Makefile to create libsampclass.dl for Unix (Makefile_dl).

```
target: libsampclass.dl

libsampclass.dl: sampclass_chdl.obj
        ch dllink libsampclass.dl cplusplus sampclass_chdl.obj prog.lib

sampclass_chdl.obj: sampclass_chdl.cpp
        ch dlcomp libsampclass.dl cplusplus sampclass_chdl.cpp

clean:
        del *.obj *.dl *.exe *.lib *.exp
```

Program 5.45: Makefile to create libsampclass.dl for Windows (Makefile_dl).

Chapter 6

Calling Special Ch Functions from C Space

In chapters 2 and 3, we have described how to access Ch variables and call Ch functions in Ch programs that are executed by C programs with embedded Ch. Some commonly used APIs, including **Ch_SymbolAddrByName()**, **Ch_CallFuncByAddr()**, **Ch_CallFuncByName()**, **Ch_CallFuncByNameVar()**, and **Ch_ExprEval()**, have been introduced with simple samples. Some variable length arrays (VLAs), computational arrays, and string type `string_t` are available only in Ch. How to call these special Ch functions involving these data types from Ch space is described in this chapter.

6.1 Calling Ch Functions with VLAs and Computational Arrays from C Space

Calling Ch functions with VLAs from C space or with return type of computational array is described in this section.

6.1.1 Calling Ch Functions with Arguments of VLAs

Calling Ch functions with arguments of VLAs will be described in this section. 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 often 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 as assumed-shape array, more 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 equivalent to build an argument list using API **Ch_VarArgsAddArg**() for function call of **Ch_CallFuncByNameVar**() .

6.1.1.1 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. Assume that the Ch function `funct`() defined below takes two one-dimensional assumed-shape arrays as arguments.

```
int funct(int a[:], int b[:]) {
...
}
```

After the Ch program is loaded and executed, function `funct`() can be called in C space as follows.

```
int dim, ext_a, ext_b;
int retval;
int aa1[3] = {1, 2, 3};
int bb1[2] = {4, 5};
/* ... */
dim = 1, ext_a = 3, ext_b = 2; /* dim and extent of aa1 and bb1,  */
Ch_CallFuncByName(interp, "funct", &retval, aa1, dim, ext_a,
                                            bb1, dim, ext_b);
/* ... */
```

The arguments `aa1` and `bb1` of function **Ch_CallFuncByName**() are arrays in C space to be passed to the Ch function as arguments. In most cases, the number of the arguments following the third argument of API **Ch_CallFuncByName**() 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 dim and 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 `funct`() takes only two arguments, the extra arguments `dim`, `ext_a`, and `ext_b` are also passed to the Ch function by **Ch_CallFuncByName**(). The arguments `aa1` and `bb1` are only addresses in C space, and `dim`, `ext_a` and `ext_b` will provide the information about the extents of arrays `aa1` and `bb1` to the Ch function. For the same reason, if Ch function `funct`() takes an argument of X-dimensional assumed-shape array, where X can be any integer, the users should add X extra arguments in **Ch_CallFuncByName**() to provide the information of extents of all dimensions of the array. Since the Ch function `funct`() only takes assumed-shape arrays of fixed dimension as arguments, the users don't need to provide the information of dimensions in C space.

A complete sample of this case is shown in Programs 6.1 and 6.2. In Ch program `callchvla2d.ch` shown in Program 6.1, the function `sum2d`() is designed to calculate the sum of each element of a two-dimensional assumed-shape array `a`. The function **shape**() can be used to get the information of shape of an array in Ch space. In C program `callchvla2d.c`, `sum2d`() is called twice by **Ch_CallFuncByName**() to calculate the sum of arrays of different shapes. The array `aa1` has shape of (2×3). In the code below,

```
dim =2, ext1 = 2; ext2 = 3; /* dim and extents of aa1 */
Ch_CallFuncByName(interp, "sum2d", &sum_ret, aa1, dim, ext1, ext2);
```

```
#include<array.h>

double sum2d(double a[:][:]) { /* function to be called from C space */
  int i, j;
  double 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;
}
```

Program 6.1: Example of calling Ch functions with arguments of assumed-shape arrays (callchvla2d.ch).

three extra arguments `dim`, `ext1`, and `ext2`, which contain the dimension and extents of `aa1`, are passed to Ch function. For array `aa2`, its dimension and extents are also passed as extra arguments in the function call below.

```
dim = 2, ext1 = 3; ext2 = 4; /* dim and extents of aa2 */
Ch_CallFuncByName(interp, "sum2d", &sum_ret, aa2, ext1, ext2);
```

Note that the dimensions of arrays in C program, 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 two dimensions. The output from executing this example is shown in Figure 6.1. If the Ch function takes the argument of an array of reference without constraint of dimension, arrays with different dimensions can be passed to it, and the information of dimensions needs to be passed. This special case will be discussed in the next section.

6.1.1.2 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 `funct`() defined below takes an array of reference with one dimension as argument.

```
int funct(array int a[&]) {
...
}
```

After the Ch program is loaded and executed, function `funct`() can be called in C space as follows.

```
int dim, ext;
int retval;
int aa1[3] = {1, 2, 3};
/* ... */
dim = 1; /* dimension of aa1 */
ext = 3; /* extents of aa1 */
Ch_CallFuncByName(interp, "funct", &retval, aa1, dim, ext);
/* ... */
```

```
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"callchvla2d.ch", NULL};
  int dim, ext1, ext2;
  double sum_ret;
  double aa1[2][3] = { 1, 2, 3,
                       4, 5, 6};

  double aa2[3][4] = { 1, 2, 3, 4,
                       5, 6, 7, 8,
                       9, 10, 11, 12};


  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a Ch function file */
  status = Ch_RunScript(interp, argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program callchvla2d.ch failed\n");

  dim = 2; /* dimension of aa1 and aa2 */

  ext1 = 2; ext2 = 3; /* extents of aa1 */
  Ch_CallFuncByName(interp, "sum2d", &sum_ret, aa1, dim, ext1, ext2);
  printf("sum2d() has been called from C space and returned %f\n\n", sum_ret);

  ext1 = 3; ext2 = 4; /* extents of aa2 */
  Ch_CallFuncByName(interp, "sum2d", &sum_ret, aa2, dim, ext1, ext2);
  printf("sum2d() has been called from C space and returned %f\n", sum_ret);

  Ch_End(interp);
}
```

Program 6.2: Example of calling Ch functions with arguments of assumed-shape arrays (callchvla2d.c).

```
ext[0] = 2
ext[1] = 3
sum2d() has been called from C space and returned 21.000000

ext[0] = 3
ext[1] = 4
sum2d() has been called from C space and returned 78.000000
```

Figure 6.1: Output from executing `callchvla2d.exe`.

```
#include<array.h>

double sum2d(array double a[&][&]) { /* function to be called from C space */
  double retval = 0;
  int i, j;
  array int ext[2] = shape(a);
  int n = ext[0], m = ext[1];
  array double 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;
}
```

Program 6.3: Example of calling Ch functions with arguments of array of reference with fixed dimension (callchvla2d.ch).

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 can be passed to function `funct`() above.

Program 6.3 is another version of Ch function `sum2d`() which takes an argument of array of reference with the fixed dimension. The C program in Program 6.2 still works with this Ch function. The output is the same as Figure 6.1.

If the Ch function takes the argument of an array of reference without constraint of dimension, arrays with different dimensions can be passed to it. Assume that the Ch function `funct`() defined below takes two one-dimensional assumed-shape arrays as arguments.

```
int funct(array int &a, array int &b) {
...
}
```

After the Ch program is loaded and executed, function `funct`() can be called in C space as follows.

```
int dim_a, ext_a, dim_b, ext_b_1, ext_b_2;
int retval;
int aa1[3] = {1, 2, 3};
int bb1[2][2] = {4, 5,
                 6, 7};
/* ... */
dim_a = 1, ext_a = 3; /* dimension and extent of aa1 */
dim_b=2, ext_b_1=2, ext_b_2=2; /*dimension and extents of bb1*/
Ch_CallFuncByName(interp, "funct", &retval, aa1, dim_a, ext_a,
                  bb1, dim_b, ext_b_1, ext_b_2);
```

```
/* ... */
```

To call the Ch function from C space, the users needs to provide not only information of extents, like what we have done in the previous section, but also the number of dimensions. The array `aa1` is a one-dimension array with extent of 3, whereas `bb1` is a two-dimension array with extents of 2 and 2. In the argument list of **Ch_CallFuncByName**(), Extra arguments, `dim_a`, `ext_a`, `dim_b` `ext_b_1` and `ext_b_2`, which contains number of dimensions and extents of `aa1` and `bb1`, have been added and passed to the Ch function `funct`().

The complete sample of this case is shown in Programs 6.4 and 6.5. The Ch function `sumNd`() in Program 6.4 is designed to calculate the sum of each element of array `a` with different dimension. The statement

```
aa = (array double [totnum])a;
```

casts array `a`, which can have any dimensions, to the one-dimensional array `aa` with `totnum` elements. In C space (Program 6.5), this Ch function is called twice to calculate the sum of two arrays of different dimensions. For the array `aa1` which has the shape of (3×4), three extra arguments, `dim`, `ext1` and `ext2`, which represent dimensions and extents, are added after the argument `aa1`. It is shown as follows.

```
dim = 2; ext1 = 3; ext2 = 4; /* dimension and extents of aa1 */
Ch_CallFuncByName(interp, "sumNd", &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.

```
/* dimension and extents of aa2 */
dim = 3; ext1 = 2; ext2 = 3; ext3 = 4;
Ch_CallFuncByName(interp, "sumNd", &sum_ret, aa2, dim, ext1, ext2, ext3);
```

The output from executing this example is shown in Figure 6.2.

6.1.2 Calling Ch Functions with Return Type of Computational Array

In this section we will describe how to call Ch functions that return computational arrays. Ch Functions with return type of computational array can be called in C by one of functions **Ch_SymbolAddrByName()**, **Ch_CallFuncByAddr()**, **Ch_CallFuncByName()**. The second argument of these functions shall be the the address that will hold the returned array. The calling syntaxes for functions returning a fixed length array and variable length array are the same.

Program 6.6 contains two functions `func1()` and `func2()`. Function `func1()` returns a computational array of fixed length, whereas Function `func1()` returns a computational array of variable length. After program `returnarray.ch` is loaded, these two functions in the Ch space are called in the C space by function calls of

```
Ch_CallFuncByName(interp, "func1", a);
```

and

```
Ch_CallFuncByNameVar(interp, "func2", b, arglist);
```

respectively. Function `func1()` has no argument. The argument list for the extents of the variable length array of function `func2()` is built with values in the C space using API **Ch_VarArgsAddArg**() in the following two statements.

```
#include<array.h>

double sumNd(array double &a) { /* function to be called from C space */
  int i, totnum = 1;
  double 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 double aa [totnum];
  aa = (array int [totnum])a;
  for(i = 0; i < totnum; i++) {
     retval += aa[i];
  }

  return retval;
}
```

Program 6.4: Example of calling Ch function with arguments of arrays of reference (callchvlaNd.ch).

```
Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, n);
Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, m);
```

The values of the returned array are passed back and saved in the memory for arrays `a` and `b` in the C space in the third argument of functions **Ch_CallFuncByName()** and **Ch_CallFuncByNameVar()**. The output of Program 6.7 is shown in Figure 6.3.

```
#include<stdio.h>
#include<string.h>
#include<embedch.h>

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"callchvlaNd.ch", NULL};
  int dim, ext1, ext2, ext3;
  double sum_ret;
  double aa1[3][4] = { 1, 2, 3, 4,
                       5, 6, 7, 8,
                       9, 10, 11, 12};

  double  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};

  /* initialize embedded Ch */
  Ch_Initialize(&interp, NULL);

  /* run a ch function file */
  status = Ch_RunScript(interp,argvv);
  if(status == CH_ERROR)
    printf("Error: execution of program callchvlaNd.ch failed\n");

  dim = 2; ext1 = 3; ext2 = 4; /* dimension and extents of aa1 */
  Ch_CallFuncByName(interp, "sumNd", &sum_ret, aa1, dim, ext1, ext2);
  printf("sumNd() has been called from C space and returned %f\n\n", sum_ret);

  dim = 3; ext1 = 2; ext2 = 3; ext3 = 4; /* dimension and extents of aa2 */
  Ch_CallFuncByName(interp, "sumNd", &sum_ret, aa2, dim, ext1, ext2, ext3);
  printf("sumNd() has been called from C space and returned %f\n", sum_ret);

  Ch_End(interp);
}
```

Program 6.5: Example of calling Ch function with arguments of arrays of reference (callchvlaNd.c).

```
ext[0] = 3
ext[1] = 4
sumNd() has been called from C space and returned 78.000000

ext[0] = 2
ext[1] = 3
ext[2] = 4
sumNd() has been called from C space and returned 156.000000
```

Figure 6.2: Output from executing callchvlaNd.exe

```
#include<array.h>
#include<stdio.h>


array int func1() [2][3]{
   array int a[2][3];
   a = 10;
   return a;
}

array int func2(int n, int m) [:][:]{
   array int a[n][m];
   a = 20;
   return a;
}

int main() {
   int n = 2, m = 3;
   array int a[2][3];
   array int b[2][3];

   a = func1();
   b = func2(n, m);
   printf("a in Ch = \n%d", a);
   printf("b in Ch = \n%d", b);
   return 0;
}
```

Program 6.6: Ch program with functions that return computational array (returnarray.ch).

```
/*************************************************************
* File Name: returnarray.c
*************************************************************/
#include<stdio.h>
#include<embedch.h>

int main() {
    ChInterp_t interp;
    int n = 2, m = 3;
    int a[2][3], b[2][3];
    int status, retval;
    char *argvv[]={"returnarray.ch", NULL};
    ChVaList_t arglist;

    Ch_Initialize(&interp, NULL);
    status = Ch_RunScript(interp,argvv);
    if(status == CH_ERROR) {
      printf("Error: execution of program returnarray.ch failed\n");
    }

    Ch_CallFuncByName(interp, "func1", a);
    /* or Ch_CallFuncByNameVar(interp, "func1", a, NULL); */
    printf("a[1][1] in C = %d\n", a[1][1]);

    arglist = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, n);
    Ch_VarArgsAddArg(interp, &arglist, CH_INTTYPE, m);
    Ch_CallFuncByNameVar(interp, "func2", b, arglist);
    Ch_VarArgsDelete(interp, arglist);
    printf("b[1][1] in C = %d\n", b[1][1]);

    Ch_End(interp);
}
```

Program 6.7: A C program calling Ch functions that return computational array (returnarray.c).

```
a in Ch =
10 10 10
10 10 10
b in Ch =
20 20 20
20 20 20
a[1][1] in C = 10
b[1][1] in C = 20
```

Figure 6.3: Output from executing returnarray.c

6.2 Calling Ch Functions with Built-in String Type `string_t`

String is a built-in data type with type specifier `string_t` in Ch. In this section we will describe how to call Ch functions with string type `string_t`. The syntax for calling Ch functions with string type `string_t` is the same as that for calling regular Ch functions, except for the following two constraints.

```
string_t func(string_t s) {
   string_t s2;
   printf("s in func() = %s\n", s);
   s2 = stradd(s, "ABCD");
   return s2;
}
```

Program 6.8: Ch program with a function with an argument and return type of string type `string_t` (callstring.ch).

1. For a function with argument of string type `string_t`, a pointer to char shall be passed to it from C.
2. If the return type of a function is string type `string_t`, the return value can be assigned to a variable of pointer to char. The allocated memory from the function, assigned to the pointer to char, needs to be freed when it is no longer needed.

Functions `func()` in Program 6.8 has an argument and return type of string type `string_t`. The function returns a string with a value of the combination of the string from the input argument and string `"ABCD"`. After program `callstring.ch` is loaded, functions `func()` in the Ch space is called in the C space by function calls of

```
Ch_CallFuncByName(interp, "func", &retvalp, str1);
```

and

```
Ch_CallFuncByNameVar(interp, "func", &retvalp, arglist);
```

respectively. Function `func()` has one argument of string type. The variable `str1` of pointer to char is passed to function `func()` in the first call using **Ch_CallFuncByName()** from C. The variable `str2` in C is passed to Ch function function `func()` through an argument list in the last argument for the API **Ch_CallFuncByNameVar()**. The argument list is built with a value of type of pointer to char in the C space using the API **Ch_VarArgsAddArg**() in the following statement.

```
Ch_VarArgsAddArg(interp, &arglist, CH_CHARPTRTYPE, str2);
```

The values of the returned string from the Ch function are assigned to the variable `retvalp` of pointer to char in C and printed out. The allocated memory are freed afterwards. The output of Program 6.7 is shown in Figure 6.4.

```
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    char *argvv[]={"callstring.ch", NULL};
    int status;
    char *retvalp, *str1 = "firstString", str2[20] = "secondString";
    ChVaList_t arglist;

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    /* run a Ch function file */
    status = Ch_RunScript(interp, argvv);
    if(status == CH_ERROR)
      printf("Error: execution of program callchvna.ch failed\n");

    printf("calling func() from C space:\n");
    Ch_CallFuncByName(interp, "func", &retvalp, str1);
    printf("retvalp = %s\n", retvalp);
    free(retvalp);

    /* string_t func(string_t) */
    printf("calling func() from C space:\n");
    arglist = Ch_VarArgsCreate(interp);
    Ch_VarArgsAddArg(interp, &arglist, CH_CHARPTRTYPE, str2);
    Ch_CallFuncByNameVar(interp, "func", &retvalp, arglist);
    Ch_VarArgsDelete(interp, arglist);
    printf("retvalp = %s\n", retvalp);
    free(retvalp);

    Ch_End(interp);
}
```

Program 6.9: A C program calling Ch functions that string type `string_t` (callstring.c).

```
calling func() from C space:
s in func() = firstString
retvalp = firstStringABCD
calling func() from C space:
s in func() = secondString
retvalp = secondStringABCD
```

Figure 6.4: Output from executing callstring.c

Chapter 7

The Debug and Callback Interface to a Ch Program

Ch has limited built-in debugging facilities. But, Embedded Ch offers debug interfaces by means of a set of interface APIs. These interfaces allow the construction of different kinds of debuggers, profilers, IDEs, and other tools that can access and modify active Ch scripts. In this chapter, the debug and callback interface to Ch scripts is described with examples. At the end, a sample text-based debugger is presented. The same code has been used inside Ch IDE distributed in Ch Professional, Student, and Evaluation Editions.

7.1 Obtaining Program and Function Information By a Callback Function

Embedded Ch offers a user-defined callback function that can be called during the program execution. A callback function is added by the function

```
int Ch_AddCallback(ChInterp_t interp, int event, ChCallback_t callback,
                   ChPointer_t clientdata, int count);
```

Only one callback function can be specified for an interpreter. A subsequent function call of **Ch_AddCallback**() will overwrite the previously specified events, callback function, client data, and count value described below. The first argument `interp` is the interpreter. The second argument *event* specifies on which events the callback will be called. The callback function may be called in seven different events, each associated with an event mask as follows.

1. **Ch_MASKCALL** Just after Ch calls and enters a new function.
2. **Ch_MASKRET** Just before Ch leaves the function.
3. **Ch_MASKBLOCK** Just after Ch enters a new block.
4. **Ch_MASKEND** Just before Ch leaves the block.
5. **Ch_MASKLINE** When Ch is about to start the execution of a new line of code, or when it jumps back in the code (even to the same lne).
6. **Ch_MASKCOUNT** After Ch executes every *count* instructions.
7. **Ch_MASKABORT** When Ch is aborted by **Ch_Abort**().

The above event masks can be used to specify events which trigger the callback function.

To avoid the callback function being called twice at the same executed line, the open brace `'{'` following the closing parenthesis of a function definition only triggers the function call event. It will not trigger a block event as shown below for a Ch function.

```
int func()
{                  // trigger CH_MASKCALL  event only
     int i;
     ...
     {             // trigger CH_MASKBLOCK event
        ...
     }             // trigger CH_MASKEND   event

    return 0;      // trigger CH_MASKRET   event
}
```

A function return event can be triggered either by a return statement or the ending of the function block for the return type of `void` of a function. Multiple events are specified using the bit-wise `or` operator `'|'`. For example, events to trigger the callback function just after a Ch program calls and enters a new function, or just before a Ch program leaves a function, can be specified by

```
int event = CH_MASKCALL | CHMASKRET;
Ch_AddCallback(interp, event, callback, clientdata, count);
```

A callback function is disabled by setting the event to **Ch_MASKNONE**. .

The third argument of **Ch_AddCallback**() is the callback function. A callback function has type **ChCallback_t** defined as follows.

```
typedef void (*ChCallback_t)(ChInterp_t interp, ChBlock_t *calldata,
                             ChPointer_t clientdata);
```

The second argument `calldata` contains the block information for the active function or program when the code executed is outside a function for the case of a line event. The type **ChBlock_t** is a structure containing the information for a block used by functions **Ch_AddCallback**() and **Ch_ChangeStack**().

```
typedef struct ChBlock_ {
   int event;          /* event for callback */
   int count;          /* every 'count' instructions for CH_MASKCOUNT */
   int level;          /* int level2() {level1();} int leve1(){level0();}
                          int leve0() {current func}. */
   int linecurrent;    /* the current line num where the program
                          is executing. */
   int linefuncbegin;  /* the line number where the definition of
                          the function begins. */
   int linefuncend;    /* the line number where the definition of the
                          function ends. */
   const char *source; /* the file name if the source is a file, otherwise,
                          it is a string.  It contains the first 70
                          characters beginning with  "@string: " */
   const char *funcname; /* function name if the block is a function,
```

```
                          otherwise NULL */
   const char *classname; /* class name if it is a member function,
                          otherwise NULL */
   int isconstructor;  /* true if it is a constructor of class,
                          otherwise false */
   int isdestructor;   /* true if it is a destructor of class,
                          otherwise false */
}ChBlock_t;
```

The details for each member of the structure **ChBlock_t** are described on page 212. The events that trigger the callback can be obtained inside a callback by event masks using the bit-wise `and` operator `'&'`. For example, the code fragment below can determine if the callback was triggered by a call event

```
void callback (ChInterp_t interp, ChBlock_t *calldata,
               ChPointer_t clientdata)
{
   // callback triggered when Ch Ch calls and enters a new function.
   if(calldata->event & CH_MASKCALL) {
      printf("event MASKCALL is true\n");
   }
   ...
}
```

The last argument `clientdata` of a callback function is a generic pointer. It points to the client data in the fourth argument of **Ch_AddCallback**() when the callback was added.

When the event contains the event mask **Ch_MASKCOUNT**, the last argument `count` of **Ch_AddCallback**() specifies that the callback function will be called after Ch executes every *count* instructions. The member *count* of `calldata` of structure **ChBlock_t** in the callback gives the number of `count` instructions after Ch executes to trigger the callback function. The count argument of **Ch_AddCallback**() and member of structure **ChBlock_t** is meaningful only when the even mask includes **Ch_MASKCOUNT**.

As an example, Ch Program 7.1 first calls function `main()` which in turn calls function `func1()`. There is a block for an if-statement inside function `func1()`.

Program 7.2 will execute the script in Program 7.1 using Embedded Ch. A callback function is added before the script is executed by the function **Ch_RunScript**(). The events that trigger the callback function are function call event, function return event, entering a block event, and leaving a block event. They are specified by the statement

```
mask = CH_MASKCALL | CH_MASKRET | CH_MASKBLOCK | CH_MASKEND;
```

Program 7.1 calls two functions `main()` and `func1()`, and execute a block inside an if-statement. Therefore, the callback function will be triggered and executed six times. In this case, the `count` argument of **Ch_AddCallback**() is is irrelevant. The contents for the variable `s` of structure `tag` is passed as the client data in the fourth argument of the function **Ch_AddCallback**(). This client data is passed to the callback function `callback()` as the third argument. The members of the structure are printed out when the callback function is triggered by a call event for the function `main()` using the following code.

```
if(calldata->event & CH_MASKCALL &&
   !strcmp(calldata->funcname, "main"))
{
   struct tag *sp;
```

```
int i = 40;
double g = 100;
double a[5]={1,2,3,4,5};

void func1(int i) {
  double g = 200;
  int y;

  printf("func1() is called\n");
  if(i) {
     int i=20, z;
     i=30;
     printf("argument i in func1() is true\n");
  }
}

int main() {
  int i = 10, x;

  func1(i);
  return 0;
}
```

Program 7.1: A Ch program (prog1.ch).

```
        sp  = (struct tag *)clientdata;
        printf("sp->i = %d, sp->d = %f\n", sp->i, sp->d);
    }
```

The member `calldata->funcname` of the calldata contains the name of the active function when the callback was triggered.

When the callback function is called, it first prints out the information for its active function by calling function `printCalldata()`. The function `printCalldata()` first prints out the event which triggered the callback function. It then prints out other relevant members of the calldata including the count for the event **Ch_MASKCOUNT**, the stack level which will be described in the next section, the current line number where the program is executing. The current line number is useful for many applications. For example, an IDE may highlight the current executed line; a debugger may trace the number of the times when a specified line is executed. If the executed code is within a function, it will print the line numbers for the beginning and end of the active function and function name. If the active function is a member function of a class, it wil print the the class name. Whether the active member function is a constructor or destructor can be determined by the members `isconstructor` and `isdestructor`, respectively. Member `source` of **ChBlock_t** contains the file name if the source of the executed code is a file. Otherwise, it is a string such as when the code is loaded from the memory by function **Ch_AppendRunScript**(). Either file name or string of the code is printed out by function `printCalldata()`. If the code is from a string, the `source` is a string up to 70 characters starting with `"@string: "`. This prefix for the code of string is skipped in the output by the statement

```
printf("The source code of the string is %s\n",
       calldata->source+strlen("@string: "));
```

The output from executing Program 7.2 is shown in Figure 7.1. Because the callback function is called six times for the specified events, function `printCalldata()` prints out the information of the active

```
/* File Name: prog1.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>

struct tag {int i; double d;};

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);

int main() {
    ChInterp_t interp;
    char *argvv[]={"prog1.ch", NULL};
    struct tag s ={10, 20};
    int mask, count=0;
    ChPointer_t clientdata;

    Ch_Initialize(&interp, NULL);
    mask = CH_MASKCALL | CH_MASKRET | CH_MASKBLOCK | CH_MASKEND;
    clientdata = &s;
    Ch_AddCallback(interp,  mask, callback, clientdata, count);
    Ch_RunScript(interp,  argvv);
    Ch_End(interp);
    return 0;
}

int printCalldata(ChBlock_t *calldata)
{
    /* callback at the begining of a function call */
    if(calldata->event & CH_MASKCALL) {
       printf("event MASKCALL is true\n");
    }
    if(calldata->event & CH_MASKRET) {
       printf("event MASKRET is true\n");
    }
    if(calldata->event & CH_MASKBLOCK) {
       printf("event MASKBLOCK is true\n");
    }
    if(calldata->event & CH_MASKEND) {
       printf("event MASKEND is true\n");
    }
    if(calldata->event & CH_MASKLINE) {
       printf("event MASKLINE is true\n");
    }
    if(calldata->event & CH_MASKCOUNT) {
       printf("event MASKCOUNT is true\n");
    }
    if(calldata->event & CH_MASKABORT) {
       printf("event MASKABORT is true\n");
    }

    if(calldata->event & CH_MASKCOUNT) {
      printf("The count number is %d\n", calldata->count);
    }
    printf("The stack level is %d\n", calldata->level);
    printf("The current line number is %d\n", calldata->linecurrent);
```

Program 7.2: C program with a callback function (prog1.c).

```
    if(calldata->classname) {
        printf("Member function %s::%s() starts at line %d \n",
               calldata->classname, calldata->funcname, calldata->linefuncbegin);
        printf("Member function %s::%s() ends at line %d \n",
               calldata->classname, calldata->funcname, calldata->linefuncend);
        printf("Member function %s::%s() \n", calldata->classname,
                calldata->funcname);
        printf("Class name is %s \n", calldata->classname);
        if(calldata->isconstructor)
           printf("Member function %s::%s() is a constructor \n",
                  calldata->classname, calldata->funcname);
        if(calldata->isdestructor)
           printf("Member function %s::%s() is a destructor\n",
                  calldata->classname, calldata->funcname);
    }
    else if(calldata->funcname) {
        printf("Function %s() starts at line %d \n",
               calldata->funcname, calldata->linefuncbegin);
        printf("Function %s() ends at line %d \n",
               calldata->funcname, calldata->linefuncend);
        printf("The function name is %s()\n", calldata->funcname);
    }

    if(calldata->source) {
      if(!strncmp(calldata->source, "@string: ", strlen("@string: "))) {
        printf("The source code of the string is %s\n",
                calldata->source+strlen("@string: "));
      }
      else  {
        printf("The current file name is %s\n", calldata->source);
      }
    }
    else {
        printf("No source information\n");
    }
    return 0;
}

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata)
{
    int *p;

    printCalldata(calldata);

    p = (int *)Ch_SymbolAddrByName(interp, "i");
    if(p == NULL)
      printf("i is not available\n");
    else
      printf("i = %d\n", *p);
    /* print out client data and callback info once */
    if(calldata->event & CH_MASKCALL && !strcmp(calldata->funcname, "main"))
    {
       struct tag *sp;
       sp  = (struct tag *)clientdata;
       printf("sp->i = %d, sp->d = %f\n", sp->i, sp->d);
    }
    printf("\n");
}
```

Program 7.2: C program with a callback function (prog1.c) (Contd.).

```
event MASKCALL is true
The stack level is 0
The current line number is 17
Function main() starts at line 17
Function main() ends at line 22
The function name is main()
The current file name is prog1.ch
i = 0
sp->i = 10, sp->d = 20.000000

event MASKCALL is true
The stack level is 0
The current line number is 5
Function func1() starts at line 5
Function func1() ends at line 15
The function name is func1()
The current file name is prog1.ch
i = 10

func1() is called
event MASKBLOCK is true
The stack level is 0
The current line number is 10
Function func1() starts at line 5
Function func1() ends at line 15
The function name is func1()
The current file name is prog1.ch
i = 0

argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 14
Function func1() starts at line 5
Function func1() ends at line 15
The function name is func1()
The current file name is prog1.ch
i = 30

event MASKRET is true
The stack level is 0
The current line number is 15
Function func1() starts at line 5
Function func1() ends at line 15
The function name is func1()
The current file name is prog1.ch
i = 10

event MASKRET is true
The stack level is 0
The current line number is 21
Function main() starts at line 17
Function main() ends at line 22
The function name is main()
The current file name is prog1.ch
i = 10
```

Figure 7.1: Output from executing `prog1.c`.

function for each event. Note that variable `i` is used as a global variable, argument for function `func1()`, and local variable inside function `main()` and if-statement block inside function `func1()`. The address of the variable `i` is obtained by the function **Ch_SymbolAddrByName**() as follows.

```
p = (int *)Ch_SymbolAddrByName(interp, "i");
```

and its value is printed in the function `callback()`. Function **Ch_SymbolAddrByName**() searches the symbol `i` in the symbol tables within its scope. The call event for function `main()` will trigger the callback function before the local variable `i` is initialized to 10 by the initialization statement. Note that a variable is by default initialized to 0 when its memory is allocated in Ch. The local variable `i` with the value 0 is processed by the callback function. The output from the callback function when it is first called due to the call event for function `main()` appears in the first block in Figure 7.1. The line

```
sp->i = 10, sp->d = 20.000000
```

displays the client data passed from the fourth argument of the function **Ch_AddCallback**() when it was called.

The second time when the function `callback()` is triggered by the call event for the function `func1()`, the argument `i` has the value 10 passed from the function main(). Entering the if-statement block before its initialization, the local variable `is` is 0. Before the end of the if-statement block, the value for the local variable `i` is 30. The value for the argument `i` before leaving the function `func1()` is 10. Similarly, the value for the local variable `i` inside function `main()` is 10 as shown in Figure 7.1.

7.2 Changing Stack

The information for variables within its scope can be obtained by various APIs inside a callback function. For example, variable `i` declared in an if-statement block in Program 7.1 can be accessed using the API **Ch_SymbolAddrByName**() inside function `callback()` when it is triggered by either beginning block or end block event. The API **Ch_SymbolAddrByName**() can also access variables inside a function within its scope. For example, when the callback function is triggered by the end block event inside the if-statement in function `func1()` in Program 7.1, local variables y and g inside function `func1()` are accessible by the **Ch_SymbolAddrByName**() in function `callback()`. The argument `i` of function `func1()` is also accessible by th index of variables in its symbol table even though the identifier `i` is used for both local variables and argument. The global variables `i, g,` and `a` in Program 7.1, however, can be accessed by the API **Ch_GlobalSymbolAddrByName**() even when the active statement of the executed program is inside a function. However, it might be necessary to access variables in the calling function such as for a debugger. For example, when the program flow is at the end of the if-statement block inside the active function `func1()`, the user callback might want to access the local variables `i` and `x` in the calling function `main()`. To access variables in a calling function, the function stack of the Ch program needs to be changed by the API **Ch_ChangeStack**() with the following function prototype.

```
int Ch_ChangeStack(ChInterp_t interp, int level, ChBlock_t *calldata);
```

Function **Ch_ChangeStack**() changes the interpreter runtime function stack. The second argument `level` specifies the level of the function being executed. Level 0 is the current running function, whereas level n+1 is the function that has called level n. The stack will be changed to the level 0 internally when a callback function specified by **Ch_AddCallback**() returns. However, if the stack is changed not inside a callback function, it needs to be reset to level 0 to resume the execution of the original running program The third argument of the function **Ch_ChangeStack**() passes back the information for the specified level

```
/* File Name: prog2.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);

int main(int argc, char *argv[]) {
    ChInterp_t interp;
    char **argvv;
    int mask, count=0;
    ChPointer_t clientdata = NULL;

    if(argc != 2)  {
        printf("Usage: %s chscriptname\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* create char *argvv[] = {"chscriptname", NULL} dynamically */
    argvv = (char **)malloc(2*sizeof(char *));
    argvv[0] = argv[1];
    argvv[1] = NULL;

    Ch_Initialize(&interp, NULL);
    mask = CH_MASKCALL | CH_MASKRET | CH_MASKBLOCK | CH_MASKEND;
    Ch_AddCallback(interp,  mask, callback, clientdata, count);
    Ch_RunScript(interp,  argvv);
    Ch_End(interp);
    return 0;
}
```

Program 7.3: C program which changes the function stack (prog2.c).

```
void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata)
{
    int *p, level=0;
    ChBlock_t calldata2;

    if(calldata->event & CH_MASKEND && !strcmp(calldata->funcname, "func1"))
    {
       printCalldata(calldata);
       symbols(interp);
       while(Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK)
       {
          printCalldata(&calldata2);
          symbols(interp);
       }
    }
}

int symbols(ChInterp_t interp) {
    int index, totalnum, *p;
    char *name;

    totalnum = Ch_SymbolTotalNum(interp);
    printf("The total number of symbols is %d\n", totalnum);
    for(index=0; index<totalnum;index++){
        printf("symbol index %d, name %s, address %#p\n", index,
              Ch_SymbolNameByIndex(interp, index),
              Ch_SymbolAddrByIndex(interp, index));
    }
    p = (int *)Ch_SymbolAddrByName(interp, "i");
    if(p == NULL)
      printf("i is not available\n\n");
    else
      printf("i = %d\n\n", *p);
    return 0;
}
```

Program 7.3: C program which changes the function stack (prog2.c) (Contd.).

of function to the calling function. The member `level` of the argument of structure **ChBlock_t** contains the level of the function or program.

Application of **Ch_ChangeStack**() is illustrated by Program 7.3. Program 7.3 will access variables at different levels of function. Unlike Program 7.2, the file name of a Ch script program executed by Program 7.3 is provided in a command line argument `argv[1]`. It is then passed to the function **Ch_RunScript**(). so that Program 7.3 can execute different Ch scripts. The callback does not have client data. But, the events to trigger the callback function is the same as in Program 5.1. The statement

```
if(calldata->event & CH_MASKEND && !strcmp(calldata->funcname, "func1"))
{
    printCalldata(calldata);
    symbols(interp);
    while(Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK)
    ....
```

in the function `callback()` inside Program 7.3 indicates that when it is triggered by a end block event by the active function `func1()`, it will call the function `printCalldata()` which is the same as in Program 7.2 to print the information for the active function. Afterwards, it calls function `symbols()` to process the variables within its scope inside the function. In function `symbols()`, first, the total number of symbols in the symbol table for the active function is obtained by the API **Ch_SymbolTotalNum**(). Next, a for-loop is used to walk through each variable in the symbol table by its index. The first variable in the symbol table has index 0. The name and address for each variable are obtained by functions **Ch_SymbolNameByIndex**() and **Ch_SymbolAddrByIndex**(), respectively, and are printed out. Finally, if the identifier `i` is a valid variable when the callback is called, its value will be printed out. After functions `printCalldata()` and `symbols()` are called once, function `callback()` enters a while-loop to process calling functions recursive by changing the function stack.

As an example, when Program 7.1 is processed by Program 7.3 using the following command

```
prog prog1.ch
```

the output is shown in Figure 7.2. When the callback is triggered by the end block event of an if-statement in Program 7.1, it will print out the information for the active function `func1()` first. Then, all variables in its symbol table, starting with one closed to the executed statement, are displayed. In this case, the symbol consists of local variables `i` and `z` inside the block, local variables `__func__, g, y` and argument `i` of the function. The local variable `__func__` of char array contains the called function name, which is a feature added in C99. The value for the local variable `i` inside the block is printed out.

Afterwards, the condition expression

```
Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK
```

of the while-loop in function `callback()` changes the function stack recursively. The stack level for the active function `func1()` is 0. The stack level for the calling function `main()` is 1, which is available as a value for the member `level` of the variable `calldata2` of structure **ChBlock_t**. Functions `printCallData()` and `symbols()` will print out the information and symbols `__func__, i`, and `x` in function `main()`. The value for the variable `i` in the calling function `main()` is 10.

The next call of function **Ch_ChangeStack**() in the while-loop will change the stack out of the function to the program scope. The stack level will be increased to 2. At the program scope, the line number is not available. In this case, the line number becomes 0 by default. There are 5 symbols `i, g, a, func1`, and `main` in the program scope. The value for the global variable `i` is 40. The subsequent call of function **Ch_ChangeStack**() with the value 3 for the stack level will return the value **CH_INVALIDLEVEL**, which indicates that the specified stack level is invalid.

```
func1() is called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 14
Function func1() starts at line 5
Function func1() ends at line 15
The function name is func1()
The current file name is prog1.ch
The total number of symbols is 6
symbol index 0, name i, address 59ad0
symbol index 1, name z, address 59b00
symbol index 2, name __func__, address 57968
symbol index 3, name g, address 5b218
symbol index 4, name y, address 5b248
symbol index 5, name i, address 4c000
i = 30

The stack level is 1
The current line number is 20
Function main() starts at line 17
Function main() ends at line 22
The function name is main()
The current file name is prog1.ch
The total number of symbols is 3
symbol index 0, name __func__, address 4f900
symbol index 1, name i, address 531b0
symbol index 2, name x, address 4bf50
i = 10

The stack level is 2
The current line number is 0
The current file name is prog1.ch
The total number of symbols is 5
symbol index 0, name i, address 44f90
symbol index 1, name g, address 53750
symbol index 2, name a, address 506f8
symbol index 3, name func1, address 50760
symbol index 4, name main, address 52140
i = 40
```

Figure 7.2: Output from executing `prog2 prog1.ch`.

```
/* File: cplus.cpp */
#include <iostream.h>
using namespace std;

int i = 40;
double g = 100;
double a[5]={1,2,3,4,5};

class tag {
     int i;
     int y;
  public:
     tag();
     ~tag();
     int func1(int j);
};

tag::tag() {
    cout <<"tag::tag() is called\n";
    i = 50;
}

tag::~tag() {
    // cout <<"tag::~tag() is called\n";
}

int tag::func1(int j) {
    cout << "tag::func1() called\n";
    if(i) {
       int i=20, z;
       i=30;
       cout << "argument i in func1() is true\n";
    }
    return 0;
}


int main() {
  int i = 10, x;
  class tag c;
  class tag c2[2];

  c.func1(i);
  return 0;
}
```

Program 7.4: Ch program with with member functions of class (cplus.cpp).

Ch supports classes in C++. Program 7.3 can also be used to process a Ch program using classes. Program 7.4 defines a class `tag` with constructor `tag()`, destructor `~tag()`, and member function `func1()`, and private data `i` and `y`. The constructor initializes the private variable `i` with value 50. A variable `c` and array of class `c2` with two elements are instantiated in function `main()`. When the member function `tag::func1()` is called inside function `main()`, the end block of the if-statement will trigger the callback function in Program 7.3. The output from executing the command

```
prog cplus.cpp
```

is shown in Figure 7.3. In this case, function `printCalldata()` will print out both class and member function names. When a member function is called, in addition to local variables and function arguments, the symbol table will also contains members of the class. The variables `__class__` and `__class_func__` contains the class name and member function name, respectively. In addition, it has a special *this* pointer to the instance of the class itself when a member function is called. Inside the if-statement block, the value for the local variable `i` is 30. The second block in Figure 7.3 contains the information for the calling function `main()` with the stack level 1. The third block in Figure 7.3 with the stack level 2 is for the program. The symbol in the symbol table does not include the `tag` name for the class.

Ch supports nested functions. Program 7.3 can also be used to process a Ch program using nested functions. Function `nestedfunc()` in Program 7.5 contains a local function `func1()`. Function `main()` calls function `nestedfunc()` which in turn calls the local function `func1()`. When the local function `func1()` is called inside function `nestedfunc()`, the end block of the if-statement will trigger the callback function in Program 7.3. The output from executing the command

```
prog nestedfunc.ch
```

is shown in Figure 7.4. The first block in Figure 7.4 contains the information for the active function `func1()`. The second block in Figure 7.4 contains the information for the nesting function `nestedfunc()` with the stack level 1. The third block with the stack level 2 is for the calling function `main()`. The last block with the stack level 3 is for the program.

Often time, a developer may expose binary C functions as system functions to Ch programs. These system functions are declared in the Ch space by the API **Ch_DeclareFunc**(). Program 7.6 defines a system function `sysfunc()` in the Ch space by the statement

```
Ch_DeclareFunc(interp,"int sysfunc(int i);", (ChFuncdl_t)sysfunc_chdl);
```

Functions `printCalldata()` and `symbols()` in Program 7.6 are the same as ones in Programs 7.2 and 7.3. The conditional statement

```
if(calldata->event&CH_MASKCALL && !strcmp(calldata->funcname,"sysfunc"))
```

enales the callback function to just print out the information for the active function `sysfunc()` and its calling functions recursively when it is triggered by a call event.

Program 7.7 calls system function `sysfunc()`. The output from executing Program 7.6 is shown in Figure 7.5. The first block in Figure 7.5 contains the information for the active system function `sysfunc()` with the stack level 0. Instead of printing out the file name, function `printCalldata()` prints out the source code of a string for a system function prototype specified in the second argument of function **Ch_DeclareFunc**(). The current line number is 0 when the program is executing a system function. The symbol table contains the internal variable `_chretval` and function argument `i`. Inside function `sysfunc()`, the passed value from Program 7.7 for the argument of the function is 10. The second block in Figure 7.5. contains the information for the calling function `main()` with the stack level 1. The third block with the stack level 2 is for the program. The symbol table for a program does not contain system functions and variables defined by the APIs **Ch_DeclareFunc**() an **Ch_DeclareVar**(), respectively.

```
tag::tag() is called
tag::tag() is called
tag::tag() is called
tag::func1() called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 33
Member function tag::func1() starts at line 27
Member function tag::func1() ends at line 35
Member function tag::func1()
Class name is tag
The current file name is cplus.cpp
The total number of symbols is 12
symbol index 0, name i, address 595a0
symbol index 1, name z, address 595d0
symbol index 2, name __func__, address 3dde0
symbol index 3, name __class__, address 4f8b0
symbol index 4, name __class_func__, address 511a8
symbol index 5, name i, address 50b10
symbol index 6, name y, address 50b14
symbol index 7, name tag, address 56fa8
symbol index 8, name ~tag, address 50b12
symbol index 9, name func1, address 50b12
symbol index 10, name this, address 592f0
symbol index 11, name j, address 59320
i = 30

The stack level is 1
The current line number is 43
Function main() starts at line 38
Function main() ends at line 45
The function name is main()
The current file name is cplus.cpp
The total number of symbols is 5
symbol index 0, name __func__, address 47ff8
symbol index 1, name i, address 55fe0
symbol index 2, name x, address 56010
symbol index 3, name c, address 50b10
symbol index 4, name c2, address 512e0
i = 10

The stack level is 2
The current line number is 0
The current file name is cplus.cpp
The total number of symbols is 4
symbol index 0, name i, address 53750
symbol index 1, name g, address 44f90
symbol index 2, name a, address 471e8
symbol index 3, name main, address 4c0c8
i = 40
```

Figure 7.3: Output from executing `prog2 cplus.cpp`.

```
int i = 40;
double g = 100;

void nestedfunc(int i) {
  double g = 300;
  int r;

  void func1(int i) {
    double g = 200;
    int y;

    printf("func1() is called\n");
    if(i) {
       int i=20, z;
       i=30;
       printf("argument i in func1() is true\n");
    }
  }
  func1(i);
}

int main() {
  int i = 10, x;

  nestedfunc(i);
  return 0;
}
```

Program 7.5: Ch program with a nested function (nestedfunc.ch).

```
func1() is called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 17
Function func1() starts at line 8
Function func1() ends at line 18
The function name is func1()
The current file name is nestedfunc.ch
The total number of symbols is 6
symbol index 0, name i, address 58d78
symbol index 1, name z, address 58da8
symbol index 2, name __func__, address 553d0
symbol index 3, name g, address 58ce0
symbol index 4, name y, address 58d10
symbol index 5, name i, address 59c50
i = 30

The stack level is 1
The current line number is 19
Function nestedfunc() starts at line 4
Function nestedfunc() ends at line 20
The function name is nestedfunc()
The current file name is nestedfunc.ch
The total number of symbols is 5
symbol index 0, name __func__, address 3e918
symbol index 1, name g, address 59b40
symbol index 2, name r, address 59b70
symbol index 3, name func1, address 50730
symbol index 4, name i, address 5b198
i = 10

The stack level is 2
The current line number is 25
Function main() starts at line 22
Function main() ends at line 27
The function name is main()
The current file name is nestedfunc.ch
The total number of symbols is 3
symbol index 0, name __func__, address 4f870
symbol index 1, name i, address 5b0b8
symbol index 2, name x, address 5b0e8
i = 10

The stack level is 3
The current line number is 0
The current file name is nestedfunc.ch
The total number of symbols is 4
symbol index 0, name i, address 44f90
symbol index 1, name g, address 53750
symbol index 2, name nestedfunc, address 59828
symbol index 3, name main, address 5c7d0
i = 40
```

Figure 7.4: Output from executing `prog2 nestedfunc.ch`.

```
/* File Name: prog3.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);

int sysfunc(int i) {
    printf("i in sysfunc() is %d\n", i);
    return 0;
}

EXPORTCH int sysfunc_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int i, retval;

    Ch_VaStart(interp, ap, varg);
    i  = Ch_VaArg(interp, ap, int);
    printf("i in sysfunc_chdl() is %d\n", i);
    retval = sysfunc(i);
    Ch_VaEnd(interp, ap);
    return retval;
}

int main() {
    ChInterp_t interp;
    char *argvv[]={"prog3.ch", NULL};
    int mask, count=0;
    ChPointer_t clientdata = NULL;

    Ch_Initialize(&interp, NULL);
    Ch_DeclareFunc(interp,"int sysfunc(int i);", (ChFuncdl_t)sysfunc_chdl);
    mask = CH_MASKCALL | CH_MASKRET | CH_MASKBLOCK | CH_MASKEND;
    Ch_AddCallback(interp,  mask, callback, clientdata, count);
    Ch_RunScript(interp,  argvv);
    Ch_End(interp);
    return 0;
}

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata)
{
    int *p, level=0;
    ChBlock_t calldata2;

    if(calldata->event & CH_MASKCALL && !strcmp(calldata->funcname, "sysfunc"))
    {
       printCalldata(calldata);
       symbols(interp);
       while(Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK)
       {
          printCalldata(&calldata2);
          symbols(interp);
       }
    }
}
```

Program 7.6: C program defines a system function in the Ch space (prog3.c).

```
int i = 40;
double g = 100;
double a[5]={1,2,3,4,5};

int main() {
  int i = 10, x;

  sysfunc(i);
  return 0;
}
```

Program 7.7: Ch program uses the system function `sysfunc()` (prog3.ch).

```
event MASKCALL is true
The stack level is 0
The current line number is 0
Function sysfunc() starts at line 0
Function sysfunc() ends at line 0
The function name is sysfunc()
The source code of the string is int sysfunc(int i);
The total number of symbols is 2
symbol index 0, name _chretval, address 53640
symbol index 1, name i, address 590b8
i = 10

The stack level is 1
The current line number is 8
Function main() starts at line 5
Function main() ends at line 10
The function name is main()
The current file name is prog3.ch
The total number of symbols is 3
symbol index 0, name __func__, address 554d0
symbol index 1, name i, address 58fd8
symbol index 2, name x, address 59008
i = 10

The stack level is 2
The current line number is 0
The current file name is prog3.ch
The total number of symbols is 4
symbol index 0, name i, address 53880
symbol index 1, name g, address 5c1c0
symbol index 2, name a, address 473f0
symbol index 3, name main, address 57110
i = 40

i in sysfunc_chdl() is 10
i in sysfunc() is 10
```

Figure 7.5: Output from executing `prog3.c`.

7.3 Manipulating Local and Global Variables

Except for APIs starting with the prefix `Ch_GlobalSymbol` for handling global variables, most other APIs in Embedded Ch can be used to manipulate both local and global variables. Given a name, an API can search the variable in symbol tables within its scope. All information about a variable or expression can be obtained by a proper API. The value for a variable can also be changed. Details of using APIs for manipulating variables and functions in the Ch space have been presented Chapters 2 and 3.

In this section, two sample application programs are presented to illustrate how to use various APIs in a callback function to manipulate variables in a Ch script. The first example prints out values for each variable in symbol tables in a script. The second example shows how to set a value for a variable and evaluate an expression using variables. Based on the ideas presented in these two examples, comprehensive debuggers, profilers, IDEs, and other tools can be developed.

7.3.1 Obtaining Values of Variables

The callback function in the presented program in this section can print out the value for each variable within the scope of the executed program at its active line of the code. It will print out the values for the local variables, members of the class for a member function, and arguments of the function first. Then, it then recursively prints out values for variables in the calling functions. Finally, it prints out the values for global variables. For a variable of class, structure, union type, the name and value for each member of the object will be printed out. For a member of class, structure, and union type, it will be processed recursively. The program can also handle scripts using classes and nested functions.

Program 7.8 lists the names and values of all local variables of the active function and variables in its upper level stacks. Function `main()` in Program 7.8 is the same as one in Program 7.3. Function `callback()` in Program 7.8 is the same as one in Program 7.3, except that it calls function `chPrintSymbolValuesInStack()` in Program 7.9 instead of `symbols()` to pring symbols and their corresponding values recursively for all valid stack levels using a while-loop. Function `chPrintSymbolValuesInStack()` calls `chPrintSymbolValue()` in a for-loop to print each symbol and its value in a symbol table. Program 7.9 is also used in a sample debugger program presented in the next section. Like `symbols()`, function `chPrintSymbolValuesInStack()` obtains the total number of variables in the symbol table for a given stack level. Each variable in the symbol table is then processed in a for-loop. Functions **Ch_SymbolNameByIndex**() and **Ch_SymbolAddrByIndex**() give the name and address of a variable according to its index number in the symbol table.

The data type of the variable is obtained by the function **Ch_DataType**(). If the data type is of structure, class, or union type, the user defined tag information for the variable is obtained by the following statement

```
udtag = Ch_UserDefinedTag(interp, name);
```

Otherwise, the value for `udtag` is NULL. The tag information `udtag` will be used to obtain the information for the user defined type and its members later on. The function **Ch_FuncType**() gives the function type of a variable. If the variable is a function type, of pointer to function or function, its value will be printed out. For function type of constructor, destructor, or member function, only its definition is printed out. Because header files may contain many function prototypes, a variable of function prototype without function definition will be ignored in function `chPrintSymbolValue()`. Function **Ch_VarType**() can be used to test whether an identifier is a local or global variable. It is used to test if a function is a local nested function in this example. The global variable `displayAll` in Program 7.9 is a flag. If it is set to 0, symbols for constructor, destructor, member functions, internally declared variables `__func__`, `__class__`, `__class_func__`, handles for dynamically loaded standard libraries starting with `_Ch` such as `_Chstdio_handle`, function `dlclose`, and functions starting with `_dlclose_`

```
/* File Name: prog4.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <embedch.h>

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);
int printCalldata(ChBlock_t *calldata);
extern void chPrintSymbolValuesInStack(ChInterp_t interp, int level);

int main(int argc, char *argv[]) {
    ChInterp_t interp;
    char **argvv;
    int mask, count=0;
    ChPointer_t clientdata = NULL;

    if(argc != 2)  {
        printf("Usage: %s chscriptname\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* create char *argvv[] = {"chscriptname", NULL} dynamically */
    argvv = (char **)malloc(2*sizeof(char *));
    argvv[0] = argv[1];
    argvv[1] = NULL;

    Ch_Initialize(&interp, NULL);
    mask = CH_MASKCALL | CH_MASKRET | CH_MASKBLOCK | CH_MASKEND;
    Ch_AddCallback(interp,  mask, callback, clientdata, count);
    Ch_RunScript(interp,  argvv);
    Ch_End(interp);
    return 0;
}

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata)
{
    int level=0;
    ChBlock_t calldata2;

    if(calldata->event & CH_MASKEND && !strcmp(calldata->funcname, "func1"))
    {
       printCalldata(calldata);
       chPrintSymbolValuesInStack(interp, calldata->level);
       while(Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK)
       {
          printCalldata(&calldata2);
          chPrintSymbolValuesInStack(interp, calldata2.level);
       }
    }
}
```

Program 7.8: C program for printing values of all variables within its scope (prog4.c).

```
/* File Name: stack.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>
#include "debug.h"


/* stactic functions within the file */
static int printUserDefined(ChInterp_t interp, ChType_t datatype,
                      void *addr, ChUserDefinedTag_t udtag, int nestlevel);
static int printValue(ChInterp_t interp, ChType_t datatype,
                      void *addr, ChUserDefinedTag_t udtag, int nestlevel);

int displayAll=1;    /* if 1, display definition of [member] functions, etc.
                        if 0, do not display definition of [member] functions, etc. */

void chPrintSymbolValuesInStack(ChInterp_t interp, int level) {
    int index, totalnum, isfunc;
    char *stackname, *name, *classname;
    void *addr;

    stackname = Ch_StackName(interp, level, &isfunc, &classname);
    totalnum = Ch_SymbolTotalNum(interp);
    if(isfunc == 1)
       printf("Function %s() at stack level %d has %d symbols:\n", stackname, level,
              totalnum);
    else if(isfunc == 2)
       printf("Member Function %s::%s() at stack level %d has %d symbols:\n",
               classname, stackname, level, totalnum);
    else
       printf("Program %s at stack level %d has %d symbols:\n", stackname, level, totalnum);
    for(index=0; index<totalnum;index++){
          name =  Ch_SymbolNameByIndex(interp, index);
          addr  =  Ch_SymbolAddrByIndex(interp, index);
          chPrintSymbolValue(interp, name, addr);
    }
    printf("\n");  /* separate the stack level in the output */
}

void chPrintSymbolValue(ChInterp_t interp, char *name, void *addr) {
    int i, j, dim, n, m, size, nestlevel=1;
    ChType_t datatype;
    ChFuncType_t functype;
    ChUserDefinedTag_t udtag;

    datatype = Ch_DataType(interp, name);
    functype = Ch_FuncType(interp, name);
```

Program 7.9: Functions for printing values of all variables in a stack (stack.c).

```
if(functype)
{
    if(functype == CH_FUNCPTRTYPE) /*  pointer to func */
      printf("%-16s0X%p\n", name, *(void**)addr);
    else if(functype == CH_FUNCTYPE) {
      char *funcname;
      if(!((!strcmp(name, "dlclose") || !strncmp(name, "_dlclose_", 9) ||
            !strncmp(name, "_Ch_", 4))
            && !displayAll)) /* suppress _dlclose_math, etc */
      {
        funcname = (char *)malloc(strlen(name)+strlen("():local")+1);
        strcpy(funcname, name);
        strcat(funcname, "()");
        if(Ch_VarType(interp, name) == CH_LOCALVARTYPE)
           strcat(funcname, ":local"); /* local function */
        printf("%-16s0X%p\n", funcname, (void*)addr);
        free(funcname);
      }
    }
    else {
      if(displayAll) {
         if(functype != CH_FUNCPROTOTYPE) /* do not print func prototype */
           printf("%-16s", name);
         if(functype == CH_FUNCCONSTYPE)
           printf("constructor\n");
         else if(functype == CH_FUNCDESTTYPE)
           printf("destructor\n");
         else if(functype == CH_FUNCMEMBERTYPE)
           printf("member function\n");
      }
    }
}
else if(Ch_ArrayType(interp, name))
{
  if(!((!strcmp(name, "__func__") || !strcmp(name, "__class__") ||
        !strcmp(name, "__class_func__") || !strncmp(name, "_Ch_", 4)) && !displayAll))
  {
         printf("%-16s", name);
         udtag = Ch_UserDefinedTag(interp, name);
         dim = Ch_ArrayDim(interp, name);
         n = Ch_ArrayExtent(interp, name, 0);
         if(dim == 1) {
           for(i=0; i<n; i++) {
               if(datatype == CH_CHARTYPE && *(char*)addr == '\0')
                  break; /* skip the rest characters of s[10]="abc" */
               size = printValue(interp, datatype, addr, udtag, nestlevel);
               addr = (char *)addr + size;
               if(datatype != CH_CHARTYPE)
                 printf(" ");
           }
         }
         else if(dim == 2) {
           m = Ch_ArrayExtent(interp, name, 1);
           for(i=0; i<n; i++) {
             if(i!=0)
                printf("%-16s", "");
             for(j=0; j<m; j++) {
                size = printValue(interp, datatype, addr, udtag, nestlevel);
                addr = (char *)addr + size;
                printf(" ");
             }
             if(i!=n-1)
                printf("\n");
           }
         }
         printf("\n");
  }
}
```

Program 7.9: Functions for printing values of all variables in a stack (stack.c) (Contd.).

```
    else {
      if(!(!strncmp(name, "_Ch", 3) && !displayAll)) /* suppress _Chstdio_handle, etc */
      {
            printf("%-16s", name);
            if(addr!=NULL) {  /* for "extern int i;", addr is NULL */
               udtag = Ch_UserDefinedTag(interp, name);
               printValue(interp, datatype, addr, udtag, nestlevel);
            }
            printf("\n");
      }
    }
}

int printUserDefined(ChInterp_t interp, ChType_t datatype,
                     void *addr, ChUserDefinedTag_t udtag, int nestlevel)
{
    int i, j, size =0, indent, index;
    ChType_t memtype;
    char format[16], expr[8];
    void *memaddr;
    ChUserDefinedInfo_t udinfo;
    ChMemInfo_t meminfo;

    /* format is "\n  %-14s", or "\n    %-12s" */
    strcpy(format, "\n");
    indent = -16;
    for(i=0; i<nestlevel; i++) {
        strcat(format, "  ");
        indent += 2;
    }
    sprintf(expr, "%d", indent);
    strcat(format, "%");
    strcat(format, expr);
    strcat(format, "s");

    Ch_UserDefinedInfo(interp, udtag, &udinfo);
    for(index=0; index<udinfo.totnum;index++){
         Ch_UserDefinedMemInfoByIndex(interp, udtag, index, &meminfo);
         if(meminfo.isconstructor) {
             if(displayAll) {
                printf(format, meminfo.memname);
                printf("constructor");
             }
         }
         else if(meminfo.isdestructor) {
             if(displayAll) {
                printf(format, meminfo.memname);
                printf("destructor");
             }
         }
         else if(meminfo.ismemberfunc) {
             if(displayAll) {
                printf(format, meminfo.memname);
                printf("member function");
             }
         }
```

Program 7.9: Functions for printing values of all variables in a stack (stack.c) (Contd.).

```
        else {
            printf(format, meminfo.memname);
            memtype = meminfo.dtype;
            memaddr = (char *)addr +meminfo.offset;
            if(meminfo.udtag) /* member is struct, class, or union */
               nestlevel++;
            if(meminfo.arraytype)
            {
              if(meminfo.dim == 1) {
                for(i=0; i < meminfo.extent[0]; i++) {
                  if(memtype == CH_CHARTYPE && *(char*)memaddr == '\0')
                     break; /* skip the rest characters of s[10]="abc" */
                  size = printValue(interp, memtype, memaddr, meminfo.udtag, nestlevel);
                  memaddr = (char *)memaddr + size;
                  if(datatype != CH_CHARTYPE)
                     printf(" ");
                }
              }
              else if(meminfo.dim == 2) {
                for(i=0; i < meminfo.extent[0]; i++) {
                  if(i!=0)
                     printf("%-16s", "");
                   for(j=0; j < meminfo.extent[1]; j++) {
                       size = printValue(interp, meminfo.dtype, memaddr,
                                         meminfo.udtag, nestlevel);
                       memaddr = (char *)memaddr + size;
                       printf(" ");
                   }
                   if(i!=meminfo.extent[0]-1)
                      printf("\n");
                 }
              }
            }
            else {
                printValue(interp, memtype, memaddr, meminfo.udtag, nestlevel);
            }
        }
    }
    return udinfo.size;
}
```

Program 7.9: Functions for printing values of all variables in a stack (stack.c) (Contd.).

```
int printValue(ChInterp_t interp, ChType_t datatype,
               void *addr, ChUserDefinedTag_t udtag, int nestlevel) {
   int size = 0;

    /* Note: all data types are implemented in the distributed stack.c */
    switch(datatype) {
      case CH_CHARTYPE:
         printf("%c", *(char*)addr);
         size = sizeof(char);
         break;
      case CH_INTTYPE:
         printf("%d", *(int*)addr);
         size = sizeof(int);
         break;
      case CH_FLOATTYPE:
         printf("%.2f", *(float*)addr);
         size = sizeof(float);
         break;
      case CH_DOUBLETYPE:
         printf("%.4f", *(double*)addr);
         size = sizeof(double);
         break;
      case CH_LCOMPLEXTYPE:
         printf("%.4f+i%.4f", *(double*)addr,  *((double*)addr+1));
         size = 2*sizeof(double);
         break;
      case CH_STRUCTTYPE:
      case CH_CLASSTYPE:
      case CH_UNIONTYPE:
         size = printUserDefined(interp, datatype, addr, udtag, nestlevel);
         break;
      case CH_CHARPTRTYPE:
         if(*(char**)addr)
           printf("%s", *(char**)addr);
         else
           printf("NULL");
         size = sizeof(char *);
         break;
      case CH_VOIDPTRTYPE:
      case CH_INTPTRTYPE:
      case CH_FLOATPTRTYPE:
      case CH_DOUBLEPTRTYPE:
      case CH_STRUCTPTRTYPE:
      case CH_CLASSPTRTYPE:
      case CH_UNIONPTRTYPE:
         printf("0X%p", *(void**)addr);
         size = sizeof(void *);
         break;
      case CH_UNDEFINETYPE:    /* invalid expr */
         printf("invalid expression");
         break;
      default:
         printf("datatype %d not supported in printValue() in file stack.c, "
                "add it based on %s/extern/include/ch.h",
                datatype, Ch_Home(interp));
         break;
    }

    return size;
}
```


Program 7.9: Functions for printing values of all variables in a stack (stack.c) (Contd.).

such as `_dlclose_math` will not be displayed. This might be desirable to minimize the displayed symbols for clarity. In Program 7.9, this flag is set 1 so that all these variables will be displayed.

Function **Ch_ArrayType**() will return a non-zero value if the variable is of array type. If the variable is not of array type, the value of the variable is printed out by the function `printValue()` in Program 7.9. If a variable is not user defined type, the arguments `datatype` for the data type and `addr` for the address of the variable will be able to obtain the value of the variable. Each case in the `switch` statement in the function `printValue()` prints out the value based on its data type. For example, the statement

```
printf("%d", *(int*)addr);
```

prints out a variable of type int. Many other data types such as `short` listed in the header file `CHHOME/extern/include/ch.h` can be added to cases for the switch statement. Function `printValue()` returns the size of the processed data type. This returned size of the data type for a variable can be used to print out values for an entire array. For a variable of user defined type, it calls function `printUserDefined()` in Program 7.9 to print out each member of the user defined type, which will be described later.

If a variable is array type, the statements

```
dim = Ch_ArrayDim(interp, name);
n = Ch_ArrayExtent(interp, name, 0);
```

obtain the dimension of the array and the extent for its first dimension. For a two dimensional array, the extent for the second dimension is obtained by the statement

```
m = Ch_ArrayExtent(interp, name, 1);
```

The values for each element of the two dimensional array are processed by two nested for-loops. The returned value of the function `printValue()` for the size of an element of the array is used to obtain the address of the next element of the array by the statement.

```
addr = (char *)addr + size;
```

This updated address is then passed to the function `printValue()`. A one-dimensional array of char is treated as a string. Characters after a terminating null character `'\0'` for a string in an array of char are not printed out. For example, if the array `s`, declared with 10 elments, was filled with the string `"abc"`, only the first 3 elements of the array will be printed out.

If the data type of a variable is structure, class, or union, function `printUserDefined()` is called to print out the value for each member of the user defined type inside function `printValue()`. Member names and their values of a variable of user defined type are printed out with a proper indentation. A member of a user defined type itself might be also a user defined type. In this case, its names and values are further indented. For example, the values for variables `k` and `c` in the code below

```
int k = 5;
struct tag1 {
    int i, x;
};
class tag2 {
    int i, y;
    struct tag s;
} c = {10, 20, 30, 40};
```

will be displayed by the function `printUserDefined()` as follows.

```
k               5
c
  i             10
  y             10
  s
    i           30
    x           40
```

The name for variable k of int type is printed out by the statement

```
printf("%-16s", name);
```

near the end of function chPrintSymbolValue(). Member names of a variable are printed out by the statement

```
printf(format,meminfo.memname);
```

in the function printUserDefined(). Not only the data type and address of a variable, but also the tag and nested level of the user defined type are passed to function printUserDefined(). The argument nestlevel is used to format the indentation of the displayed output. At the beginning of the function printUserDefined(). based on the value for nestlevel, the format specifier format for the output of member i, y and s of class c is formed as "\n %-14s", whereas the format specifier for members i and x of variable s of structure type, which is also a member of the class c, is "\n %-12s".

The function **Ch_UserDefinedInfo**() is called to obtain the information of the user defined type. The variable udinfo of the structure type **ChUserDefinedInfo_t** contains the data type, tag name, size, and number of members of the user defined type. The member field udinfo.totnum is used as the upper limit of a for-loop to process all members inside the user defined type. The member field udinfo.size contains the size of the user defined type. The return value of function printUserDefined() as the size of the variable is used to handle a variable of array type.

A member of a user defined type itself is a variable. The information about each member of of variable of user defined type is obtained by the function ChUserDefinedMemInfoByIndex(). A member variable in function printUserDefined() is handled similar to a regular variable in function chPrintSymbolValue(). The variable meminfo of structure **ChMemInfo_t** contains all necessary information for a member of user defined type. Whether if it is constructor, destructor, or member function can be determined by the values of meminfo.isconstructor, meminfo.isdestructor, or meminfo.ismemberfunc. The value meminfo.dtype gives the data type of the member. The address for the member is obtained by the expression

```
memaddr = (char *)addr + meminfo.offset;
```

If the member itself is user defined type, the value meminfo.udtag contains the tag for the user defined type. The nestlevel is incremented for printing the names and values of its members. For a member of array type, it is handled similar to a regular array as in function chPrintSymbolValue(). The value meminfo.dim gives the dimension of the array. The array meminfo.extent contains the extents for each dimension of the array, up to 7 dimension.

Similar to Program 7.3, three sample scripts will be processed by Program 7.8. Functions func1() and main() in Program 7.10 are the same as ones in Program 7.1. Program 7.10 contains a header file stdio.h and more global variables of different data types. The output from executing the command

```
prog4 prog4.ch
```

```
#include <stdio.h>
#include <stdbool.h>

int i = 40, *ptri;
double g = 100;
double a[5]={1,2,3,4,5};
double b[2][3]={1,2,3,4,5,6}, (*ptrd)[3];
int n = 2, m = 3;
double v1[n];
double v2[n][m];
double complex z = complex(3, 4);
char str1[]="string1";
char str2[10]="string2";
char *str3="string3";
char *str4[] = {"string41", "string42", NULL};
struct tag {int i; double d;} s = {10, 20};
struct tag *sp = &s;
struct tag sa[2] = {10, 20, 30, 40};
enum tage {red, green, blue} e1 = blue;
string_t str5 = "abcd";
long long ll = 100;
char c = 'A';
bool bo = true;

ptri = &i;
ptrd = b;
v1[0] = 1, v1[1] = 2;
v2[0][0]=1; v2[1][2]=6;

void func1(int i) {
  double g = 200;
  int y;

  printf("func1() is called\n");
  if(i) {
     int i=20, z;
     i=30;
     printf("argument i in func1() is true\n");
  }
}

int func2(int i) {
  printf("func2() is called\n");
  return 2*i;
}

int main() {
  int i = 10, x;

  func1(i);
  return 0;
}
```

Program 7.10: Ch program with user defined data type (prog4.ch).

```
func1() is called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 39
Function func1() starts at line 30
Function func1() ends at line 40
The function name is func1()
The current file name is prog4.ch
Function func1() at stack level 0 has 6 symbols:
i               30
z               0
__func__        func1
g               200.0000
y               0
i               10

The stack level is 1
The current line number is 50
Function main() starts at line 47
Function main() ends at line 52
The function name is main()
The current file name is prog4.ch
Function main() at stack level 1 has 3 symbols:
__func__        main
i               10
x               0

The stack level is 2
The current line number is 0
The current file name is prog4.ch
Program prog4.ch at stack level 2 has 81 symbols:
dlclose()       0X5a340
_Chstdio_handle 0Xff1f1470
_dlclose_libstdio()0X5ad40
i               40
ptri            0X5da50
g               100.0000
a               1.0000 2.0000 3.0000 4.0000 5.0000
b               1.0000 2.0000 3.0000
                4.0000 5.0000 6.0000
ptrd            0X56d88
n               2
m               3
v1              1.0000 2.0000
v2              1.0000 0.0000 0.0000
                0.0000 0.0000 6.0000
z               3.0000+i4.0000
str1            string1
str2            string2
str3            string3
str4            string41 string42 NULL
s
  i             10
  d             20.0000
sp              0X5bcf0
sa
  i             10
  d             20.0000
  i             30
  d             40.0000
e1              2
str5            abcd
ll              100
c               A
bo              1
func1()         0X5fdf8
func2()         0X610d8
main()          0X616a8
```


Figure 7.6: Output from executing `prog4 prog4.ch`.

is shown in Figure 7.6. When the script in Program 7.10 is processed, a block end event in the function `func1()` will trigger the callback. The first two blocks in the output in Figure 7.6 are similar to ones in Figure 7.2. Instead of displaying the address of each variable in a given stack level in Figure 7.2, the output in Figure 7.6 displays the value of a each variable. At the stack level 3, there are 75 global variables. The output for a large number of function prototypes defined in the header file `stdio.h` are suppressed in function `chPrintSymbolValue()`. Functions `dlclose(), exit()` ad `_dlclose_libstdio()` and global variable `_Chstdio_handle` are used to handle dynamically loaded library for functions in defined in the header file `stdio.h`. Like the variable `ptdi` of pointer to int, the value for the variable `ptrd` of pointer to C array is also the address it points to. All elements of arrays `a, b, v1, v2` are displayed. An array of char is treated as a string. The values of arrays `str1` and `str2` are displayed similar as strings for variables `str3` and `str4`. For the variable `s` of structure and `sa` of array of structure, the values for each member are also displayed.

Like Program 7.3, Program 7.8 can also process a script using C++ classes. The output from executing the command

```
prog4 cplus.cpp
```

for the script in Program 7.4 is shown in Figure 7.7. The output in Figure 7.7 is similar to one in Figure 7.3, except that, instead of the addresses, the values of each variable are displayed.

Similarly, when Program 7.8 is used to process Program 7.5 with a nested function, The output from executing the command

```
prog4 nestedfunc.ch
```

is shown in Figure 7.8. The function `func1()` in the stack level 1 is displayed as a local function in Figure 7.8. For comparison, the output in Figure 7.8 is similar to one in Figure 7.4, except that, instead of the addresses, the values of each variable are displayed.

In function `chPrintSymbolValue()`, all variables and their values are printed out, except for function prototypes without function definition. However, to build a GUI debuger, one may suppress the display of member function definitions, constructor, destructor of a class, function definitions, built-in variables `__func__`, `__class__`, `__class_name__`, and Ch specific global variables `_Chlibname_handle` for handling functions in libraries.

```
tag::tag() is called
tag::tag() is called
tag::tag() is called
tag::func1() called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 33
Member function tag::func1() starts at line 27
Member function tag::func1() ends at line 35
Member function tag::func1()
Class name is tag
The current file name is cplus.cpp
Member Function tag::func1() at stack level 0 has 12 symbols:
i               30
z               0
__func__        func1
__class__       tag
__class_func__  tag::func1
i               50
y               0
tag             constructor
~tag            destructor
func1           member function
this            0X51a58
j               10

The stack level is 1
The current line number is 43
Function main() starts at line 38
Function main() ends at line 45
The function name is main()
The current file name is cplus.cpp
Function main() at stack level 1 has 5 symbols:
__func__        main
i               10
x               0
c
  i             50
  y             0
  tag           constructor
  ~tag          destructor
  func1         member function
c2
  i             50
  y             0
  tag           constructor
  ~tag          destructor
  func1         member function
  i             50
  y             0
  tag           constructor
  ~tag          destructor
  func1         member function

The stack level is 2
The current line number is 0
The current file name is cplus.cpp
Program cplus.cpp at stack level 2 has 4 symbols:
i               40
g               100.0000
a               1.0000 2.0000 3.0000 4.0000 5.0000
main()          0X4d010
```

Figure 7.7: Output from executing `prog4 cplus.cpp`.

```
func1() is called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 17
Function func1() starts at line 8
Function func1() ends at line 18
The function name is func1()
The current file name is nestedfunc.ch
Function func1() at stack level 0 has 6 symbols:
i               30
z               0
__func__        func1
g               200.0000
y               0
i               10

The stack level is 1
The current line number is 19
Function nestedfunc() starts at line 4
Function nestedfunc() ends at line 20
The function name is nestedfunc()
The current file name is nestedfunc.ch
Function nestedfunc() at stack level 1 has 5 symbols:
__func__        nestedfunc
g               300.0000
r               0
func1():local   0X51678
i               10

The stack level is 2
The current line number is 25
Function main() starts at line 22
Function main() ends at line 27
The function name is main()
The current file name is nestedfunc.ch
Function main() at stack level 2 has 3 symbols:
__func__        main
i               10
x               0

The stack level is 3
The current line number is 0
The current file name is nestedfunc.ch
Program nestedfunc.ch at stack level 3 has 4 symbols:
i               40
g               100.0000
nestedfunc()    0X5a770
main()          0X5d718
```

Figure 7.8: Output from executing `prog4 nestedfunc.ch`.

7.3.2 Changing Values of Variables

Many development tools such as a debugger need to change the value of a variable or evaluate an expression during the execution of a script. In this section, a sample program will be used to illustrate how to change the value of a variable in a script and evaluate a Ch expression in the callback function.

Program 7.11 is used to process the script in Program 7.12. Program 7.12 is modified from Program 7.1. Program 7.12 contains some printing statements to view the values of variables in the Ch space changed in a callback function. The output from executing Program 7.11 is shown in Figure 7.9.

The callback function in Program 7.11 is triggered similarly as in previous examples. The block end event in the if-statement in function `func1()` in Program 7.12 allows the compound statements within the if-statement in the function `callback()` executed. It first prints out the stack information for the active function using function `printCalldata()` as presented in Program 7.2. It then sets the variable `y` in Program 7.12 with the value 50 for the variable `j` in the C space using function `setVariable()`. In an application, the name of the variable might be obtained interactively. Before it is assigned a value, it needs to be checked if it is a valid lvalue by functions **Ch_VarType**() and **Ch_FuncType**(). Afterwards, expressions `i*y` and `2*i+r+g` are evaluated by function `exprCalculation()`. At this point, the values for variables `i`, `y`, `r` and `g` are 30, 50, 200, and 500, respectively. Function `exprCalculation()` first tests if an expression is valid or not by the API **Ch_ExprParse**(), then calculate the expression using the API **Ch_ExprCalc**(), and finally display the expression and its value.

After calling function **Ch_ChangeStack**() with the stack level 1, the program stack goes to the calling function `main()`. Function `printCalldata()` is called again to display the information of the function `main()`. The local variable `x` in the function `main()` and and global variable `r` in the Ch space are assigned with the values of variables `j` and `d` in the C space, respectively. At this point, the values for variables `i`, `x`, `r` and `g` are 10, 50, 500, and 100, respectively.

After the callback function is finished, printf statements in Program 7.12 display the values of local and global variables as shown in Figure 7.9.

The ideas presented in Program 7.8 can also be used to handle variables of user defined data type. As an example, after obtaining the address of a member of a class, its value can be assigned in the C space directly as shown below.

```
void *addr; /* address for a member of a class */
int *ptri;
...

ptri = addr;
*ptri = 10;  /* assign the value for a member of a class */
```

```
/* File Name: prog5.c */
#include <stdio.h>
#include <string.h>
#include <embedch.h>

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);

int main() {
    ChInterp_t interp;
    char *argvv[]={"prog5.ch", NULL};
    int mask, count=0;
    ChPointer_t clientdata = NULL;

    Ch_Initialize(&interp, NULL);
    mask = CH_MASKCALL | CH_MASKRET | CH_MASKBLOCK | CH_MASKEND;
    Ch_AddCallback(interp,  mask, callback, clientdata, count);
    Ch_RunScript(interp,  argvv);
    Ch_End(interp);
    return 0;
}

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata)
{
    int  level=0;
    ChBlock_t calldata2;
    char name[256], expr[256];
    int j;
    double d;

    if(calldata->event & CH_MASKEND && !strcmp(calldata->funcname, "func1"))
    {
       printCalldata(calldata);
       /* use scanf() or fgets() to obtain a variable name such as "y" from the user */
       strcpy(name, "y");
       j = 50; /* use scanf() to obtain the value from the user */
       if(Ch_VarType(interp, name) && !Ch_FuncType(interp, name))
       {
          setVariable(interp, name, &j);
       }
       d = 500.0; /* use scanf() to obtain the value from the user */
       setVariable(interp, "g", &d);
       /* use scanf() or fgets() to obtain an expression */
       strcpy(expr, "i*y");
       exprCalculation(interp, expr);
       strcpy(expr, "2*i+r+g");
       exprCalculation(interp, expr);
       printf("\n");

       if(Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK) {
         printCalldata(&calldata2);
         setVariable(interp, "x", &j);
         setVariable(interp, "r", &d);
       }
       /* use scanf() or fgets() to obtain an expression */
       strcpy(expr, "i*x");
       exprCalculation(interp, expr);
       strcpy(expr, "2*i+r+g");
       exprCalculation(interp, expr);
    }
}
```


Program 7.11: C program for obtaining information for variables of user defined type (prog5.c).

```
int setVariable(ChInterp_t interp, const char *name, void *data) {
    ChType_t datatype;
    int retval;

    datatype = Ch_DataType(interp, name);
    switch(datatype) {
      case CH_INTTYPE:
         retval = Ch_SetVar(interp, name, datatype, *(int *)data);
         break;
      case CH_DOUBLETYPE:
         retval = Ch_SetVar(interp, name, datatype, *(double*)data);
         break;
      default:
         printf("datatype %d not supported, add it based on %s/extern/include/ch.h\n",
                 datatype, Ch_Home(interp));
         break;
    }
    return retval;
}

int exprCalculation(ChInterp_t interp, const char *expr) {
    ChType_t datatype;
    int retval;
    int i;
    double d;

    if(Ch_ExprParse(interp, expr) == CH_ERROR)
       return -1;
    datatype = Ch_DataType(interp, expr);
    switch(datatype) {
      case CH_INTTYPE:
         retval = Ch_ExprCalc(interp, expr, datatype, &i);
         printf("%s = %d\n", expr, i);
         break;
      case CH_DOUBLETYPE:
         retval = Ch_ExprCalc(interp, expr, datatype, &d);
         printf("%s = %f\n", expr, d);
         break;
      default:
         printf("datatype %d not supported, add it based on %s/extern/include/ch.h\n",
                 datatype, Ch_Home(interp));
         break;
    }
    return retval;
}
```

Program 7.11: C program for obtaining information for variables of user defined type (prog5.c) (Contd.).

```
#include <stdio.h>

int i = 40;
double g = 100;
double r = 200;

void func1(int i) {
  double g = 200;
  int y;

  printf("func1() is called\n");
  if(i) {
     int i=20, z;
     i=30;
     printf("argument i in func1() is true\n");
  }
  printf("i in func1() is %d\n", i);
  printf("y in func1() is %d\n", y);
  printf("g in func1() is %f\n", g);
}

int main() {
  int i = 10, x;

  func1(i);
  printf("x in main() is %d\n", x);
  printf("g in main() is %f\n", g);
  printf("r in main() is %f\n", r);
  return 0;
}
```

Program 7.12: Ch program with user defined data type (prog5.ch).

```
func1() is called
argument i in func1() is true
event MASKEND is true
The stack level is 0
The current line number is 16
Function func1() starts at line 7
Function func1() ends at line 20
The function name is func1()
The current file name is prog5.ch
i*y = 1500
2*i+r+g = 760.000000

The stack level is 1
The current line number is 25
Function main() starts at line 22
Function main() ends at line 30
The function name is main()
The current file name is prog5.ch
i*x = 500
2*i+r+g = 620.000000
i in func1() is 10
y in func1() is 50
g in func1() is 500.000000
x in main() is 50
g in main() is 100.000000
r in main() is 500.000000
```

Figure 7.9: Output from executing `prog5.c`.

```
************************ Debug Menu ************************
load filename             load 'filename' for debugging
start [args]              start the program with debugging
run [args]                run the program without debugging
step                      step into a function or next line
next                      step over a function or next line
cont                      continue till hitting a breakpoint or ends
up                        change stack to the calling function
down                      change stack to the called function
stack                     display stack names in all stacks
locals                    display variables and values within its scope
variables                 display variables and values in all stacks
watch expr                add an expression into the watch list
remove expr               remove an expression from the watch list
remove                    remove all expressions from the watch list
stopat filename # [cond] set a new breakpoint in a file at line #
stopin funcname [cond]   set a new breakpoint in a function
stopvar varname [cond]   set a new breakpoint for a controlling variable
clearline filename #     clear a breakpoint in a file at line #
clearfunc funcname       clear a breakpoint for a function
clearvar varname         clear a breakpoint for a variable
clear                     clear all breakpoints
help                      display this debug menu
assign var=expr           assign a value to a variable
call func()               call a function
print expr                print out the value of an expression
expr                      print out the value of an expression
abort                     abort the debugger
........................................................
debug>
```

Figure 7.10: The menu for the user interface of the debug program.

7.4 A Sample Debugger

Based on the debug and callback interface described in the previous sections, a text-based debugger is presented in this section. Using the concept illustrated in this example, one may develop sophisticated GUI-based debuggers or remote debuggers.

This debug program has all capabilities available in a typical debugger for binary C programs. The interactive user interface for our debugger is shown in Figure 7.10. The menu on the left before a colon shows a command and the description on the right explains the action taken for the command. With this debugger, the user can `load` a program, `start` the program with debugging, or `run` the program without debugging. The user can execute the program line by line either by command `step` or `next`. The command `step` will step into a function whereas or the command `next` will step over the function to the next line. During the debugging, the command `cont` can be invoked to continue the execution of the program till it hits a breakpoint or the program ends. The user can change the function stack during debugging. It can go `up` to its calling function or move `down` to the called function so that the variables within its

scope can be accessed in the debugger. The function or program names in all stacks are displayed by the command `stack`. Names and their corresponding values of variables in the current scope are displayed by the command `locals`. Names and their corresponding values of variables in all stacks can be displayed by the command `variables`. The command `watch` adds an expression including a single variable into the list of watched expressions when the program is stopped at a breakpoint or stepped into next statement. An expression can be removed from the list of the watched expressions by the `remove expr` command. All expressions in the watched list can be removed by the `remove` command. Before the program execution or during the debugging of an executed program, new breakpoints can be added to `stop` the program execution or `cleared`. A breakpoint can be setup based on three specifications: file name and line number, function, and controlling variable. When a breakpoint is setup in a function, the program will stop at its first executable line of the function. When a breakpoint is setup for a variable, the program will stop when the value of the variable changes. Each breakpoint can have an optional conditional expression. When a breakpoint location is reached, the conditional expression is evaluated if it exists. The breakpoint is hit only if the expression is either true or has changed which needs to be specified when the breakpoint was added. In this implementation, by default, the breakpoint is hit only if the expression is true. If this sample debugger is modified as a GUI-based debugger, the above debugging capabilities could be implemented using clickable buttons or type-in commands in an interactive input/output window in the user interface. The `help` command displays the debug menu shown in Figure 7.10.

The variables, expressions, and functions can be manipulated by the last three commands in Figure 7.10. The command `assign` assigns a value to a variable, `call` invokes a function, and `print` prints out the value of a variable or expression including functions. It is invalid to print an expresion of void type including a function with return type void.

One can also just type an expression, the value of the expression will be displayed. If the expression is a function with the returning type of void, only the function is called. The sample debugger can debug multiple programs, one at a time. The command `abort` terminates the program.

The header file `debug.h` in Program 7.13 defines macros, structures, and function prototypes for the debug program. This sample debugger does not use a global variable. Therefore, it can be conveniently extended for debugging multi-thread applications with multiple instances of Embedded Ch. Boolean values `TRUE` and `FALSE` are defined for handling variables with only true or false values. The macro `MAX_PROMPT_STR` is used to allocate memory for a string containing the user input at the prompt `debug->`. The macro `MAX_NUM_ARGS`, is used for handling command line arguments, which will be described later.

The enumerate type `action_t` contains enumerate numbers for each command in the user interface menu. The enumerate type `runmode_t` contains enumerate numbers for each command that can execute a Ch script. They include `start, run, step, next,` and `cont`.

The structure `breakPoint_t` is for a linked list of breakpoints. It can be used to hold any of three types of breakpoints of file, function, or variable type. The member `btype` is used to hold the breakpoint type. A union inside the structure contains the information for a breakpoint of different types. A breakpoint of file type needs both file name `filename` and line number `linenum`. A breakpoint of function type needs a function name `funcname` and field `called`. The program will stop at the first executable line of the function when the breakpoint is hit. The field `called` is set to 1 when the first executable line of the function is being executed. It is reset to 0 when the function returns so that the program will stop at the function again next time when it is called. A breakpoint of controlling variable type needs both variable name `varname` and its previous value saved in a memory pointed by an opaque pointer `varvalue`. When the controlling variable changes, the breakpoint will be hit. The members `condexpr`, `condvalue`, and `condtrue` are used to hold a conditional expression for a breakpoint. The value for the conditional expression `condexpr` of different data type is saved in the memory pointed by an opaque pointer `condvalue`. If `condtrue` is true, that the conditional value is true triggers the breakpoint. Otherwise, the change of the conditional value triggers the breakpoint. The structure `debigInfo_t` contains

all information relevant to the debug. The member `progname` of the structure `debigInfo_t` contains the program loaded by the command `load`. The member `argvv` with the program name and optional command line arguments, up to the maximum number specified by the macro `MAX_NUM_ARGS`, is passed to the API **Ch_ParseScript**() or **Ch_RunScript**(). The member `interp` saves the interpreter instance. The member `option` contains the information including the directory of Embedded Ch used in the API **Ch_Initialize**(). Members `callback, clientdata, calldata`, and `count` contain the callback information related to **Ch_AddCallback**(). The member `level` is used to keep track the function stack level for commands `up` and `down`. The remaining members `runmode, bkpt, wexpr, previousfuncname`, and `stepovernum` are used to handle breakpoints of different commands to execute a script. The member `runmode` indicates which command invokes the current statement in the callback function. The member `bkpt` contains the head of the linked list of breakpoints. The member `wexpr` contains the head of the linked list of watched expressions. The member `previousfuncname` with the function name in the previous callback is used to implement a breakpoint of function type. If the function name in the current callback is different from that in the previous callback, the function breakpoint will be triggered. The member `stepovernum` is used to implement the command `next` to step over a function.

The remaining part of the header file `debug.h` contains function prototypes for external functions used in the sample debugger. A external function name starts with the prefix `ch` such as `chInitDebug()`. Static functions in the file scope are also listed as comments in this header file. They are separated into four filles. Functions in file `stack.c` in Program 7.9 are for printing symbols and their values in a symbol list. Functions in `callback.c` are mainly for callback interface with Embedded Ch. Functions in `menu.c` are for user interface. File `debug.c` contains functions for initializing and cleaning up a debugger, functions corresponding to commands in the user interface shown in Figure 7.10, and function for processing the interactive user input menu.

The sample debugger in Program 7.14 can debug multiple programs, one at a time, in each iteration in a while-loop. Program 7.14 first calls function `chHelp()` in Program 7.15 to display the debug menu shown in Figure 7.10. It then enters an infinite while-loop for menu driven debugging. A debugger is initialized by the function `chInitDebug()` defined in Program 7.16. The output and error streams are set without buffer by the statements

```
setbuf(stdout, NULL); /* no buffer for debugging purpose */
setbuf(stderr, NULL); /* no buffer for debugging purpose */
```

so that the proper sequence for input and output can be maintained when this sample code is modified for a GUI-based debugger. In Windows, the input and output might need to be redirected by function **Ch_Reopen**(). Each element for command line options in `dgptr->argvv` is set to NULL. For an application with its customized distribution of Embedded Ch, the member `dgptr->option` needs to be setup properly as shown in Program 7.17. For statements in function `ChInitDebug()`,

```
dgptr->callback = chCallback;
dgptr->clientdata = (ChPointer_t)dgptr;
```

the member `callback` is initialized with the callback function. The member `clientdata` is initialized to the debugger `debug` itself so that it will be passed as client data to the callback function. These members will be used as arguments of the API **Ch_AddCallback**() to register the callback function. Like a GUI program, this sample debug program is also event-driven. After initialization, the program waits for the user input from the function `chProcessActionMenu()`. After a script is processed, the program is terminated by function `chEndDebug()`. Function `chCleanDebug()` cleans the allocated memory. Note that the API **Ch_End**() may invoke some functions registered by the standard C function **atexit**() in the Ch space in a header file. When these registered functions are called, the callback function will be triggered

which will use the linked list of breakpoints in the debugger. Therefore, the function **Ch_End**() must be called first before `chCleanDebug()` is called, which in turn calls `chClearWatchExprs()` to clear all expressions in the watched list and `chClearBreakPoints()` to clear all breakpoints. One may change the function `main()` as a regular function so that it can be called to debug different programs.

Before we explain other functions, we will take a look at how the user interface function `chProcessActionMenu()` is implemented in Program 7.16. The while-loop inside function `chProcessActionMenu()` processes the user input. Using a while-loop, the user can input commands multiple times. For example, the user can set multiple breakpoints or print out values of different variables during the debugging. When the user types a command such as `step` to execute the script, the while-loop will terminate. In the while-loop, it first calls the local static function `getACtionMenu()` to obtain the user input. The user input is then compared with the command string. Some commands such as `step` have no argument whereas others such as `load` have an argument. The user input is passed back to the calling function by the argument `line` of function `getACtionMenu()`. The `getACtionMenu()` returns an enumerate number corresponding to a command. For each command, a corresponding function in Program 7.16 is called inside the while-loop of the function `chProcessActionMenu()`.

To debug a program, the user needs to specify a program first. For example, to load program `prog1.ch`, the user may enter the command as follows.

```
debug> load prog1.ch
```

Function `chLoadFile()` in Program 7.16 corresponds to the enumerate number `ACTION_LOAD` in Program 7.15 for the command `load` in Figure 7.10. The program name is copied to the member `progname` of the debugger.

Function `chStart()` starts debugging a program. The command optional arguments for the command `start` and `run` are processed by the local static function getArgvFromStr(). For example, the command

```
debug> load prog6.ch
debug> start opt1 opt2
```

will pass the arguments `prog6.ch`, `opt1` and `opt2` to `argv[0]`, `argv[1]`, and `argv[2]` of the main function

```
int main(int argc, char *argv[])
```

of the Ch script `prog6.ch`, respectively. An optional argument with space should be enclosed within two double quotation marks as shown below.

```
debug> start opt1 "opt2 with space" opt3
```

If the user types command `step`, `next`, or `cont` before the program has started, the function `chStart()` will be called. The argument `runmode` of function `chStart()` therefore can have a different execution command, which is saved in the member `runmode` of the debugger. Function `chStart()` starts the Embedded Ch by calling the API **Ch_Initialize**(). The instance of the interpreter is saved as a member of the debugger. Before the program is executed by the APIs **Ch_ParseScript**() and **Ch_ExecScript**(), the callback function with the new line event mask is added by the API **Ch_AddCallback**(). Typically, the user needs to load a program and set breakpoints before starting a program.

To execute the loaded program without debugging or executing the program in the mid of the debugging, the user can type command `run`. The corresponding function `chRun()` will be called. It unregisters the callback function by the event mask **Ch_MASKNONE**. If the loaded program is already started, it just changes the event mask of the callback function by calling the API **Ch_AddCallback**() only. Otherwise, a new instance of Embedded Ch needs to be used to execute the program. The execution commands `step,`

`next`, and `cont` are similar. If an Embedded Ch has not started yet, the program will start an interpreter first by calling function `chStart()`. The event mask all contains the new line event `CHMASKLINE` and function return event `CHMASKRET`. The event `CHMASKRET` is used to handle a breakpoint for function. The expression `dgptr->runmode` with different values will be used inside the callback function to handle different commands. To step over a function, the function call event **Ch_MASKCALL** and function return event **Ch_MASKRET** are added to the event mask.

Functions `chUpStack()` and `chDownStack()` corresponding to commands `up` and `down` change the function stack of the executed program by calling the API **Ch_ChangeStack**(). In a GUI-based debugger, the members of `dgptr->calldata` and the passed argument `calldata` of the API **Ch_ChangeStack**() containing the information of the functions in the stack could be used to update the displayed program in a GUI editor. The current and highest possible stack levels can be obtained by the API **Ch_StackLevel**(). The name in a stack can be obtained by the CPI **Ch_StackName**(). Command `stack` corresponding to function `chStack()` displays function, member function, or program name and corresponding stack level in each stack by calling function **Ch_StackName**() in a while-loop. Command `locals` displays names and values of variables in the current scope by calling function `chLocals()` which in turn calls the function `chPrintSymbolValuesInStack()` in Program 7.9 described in the previous section. When the control of the program execution is inside a function, command `locals` displays the values of local variables and arguments of the function. When the control of the program execution is not in a function of a script, command `locals` displays the values of global variables of the program. Function `chVariales()` corresponding to command `variables` displays names and values for all variables within its scope in each stack recursively by calling function `chPrintSymbolValuesInStack()`. The command `watch` calls function `chAddWatchExpr()` to add an expression to a list of watched expressions and variables. The memory allocated for an expression inside the function `chAddWatchExpr()` is freed in function `chRemoveWatchExpr()` and `chRemoveWatchExprs()`. The command `remove expr` calls function `chRemoveWatchExpr()` to remove the expression `expr` from the watched list. For the command `remove`, function `chRemoveWatchExprs()` is called to remove all expressions in the watched list. Function `chRemoveWatchExprs()` is also called by `chCleanDebug()` at the end of the program execution. Function `chProcessWatchExprs()` is called in function `chCallback()` in Program 7.17 when the progarm is stopped by `step` or next command, or in a breakpoint to display each expression and its value in the list of watched expressions and variables. In a typical GUI-based debugger, the function `chProcessWatchExprs()` needs to be modified to display expressions and their corresponding values in a separate window.

Function `chSetBreakPointLine()` sets a new breakpoint specified by a file name and line number. The program breaks execution when it reaches this location. Function `chSetBreakPointFunc()` sets a new breakpoint for a function. The program breaks execution when it reaches the first executable line of the function. Function `chSetBreakPointVar()` sets a new breakpoint for a controlling variable. The variable is evaluated while the program is running. The program breaks execution when the value of the variable changes. When each of these functions is called, a breakpoint is appended to the linked list of breakpoints. The optional conditional expression and triggering method for each breakpoint are passed as the last two arguments of these functions. For example, the syntaxes for setting a breakpoint in a file and line number are as follows.

```
debug> stopat filename #
debug> stopat filename # condexpr
debug> stopat filename # condexpr condtrue
```

When a breakpoint location is reached, the optional expression `condexpr` is evaluated. If the argument `condtrue` is true or missing, the breakpoint will be hit if the value for the expression is true; otherwise, the breakpoint will be hit if the value for the expression has changed. For example, the command

```
debug> stopat prog1.ch 6
```

sets a breakpoint in file `prog1.ch` at line 6. The command

```
debug> stopat prog1.ch 6 i+j 1
```

sets a breakpoint in file `prog1.ch` at line 6. When the breakpoint location in file prog1.ch at line 6 is reached, the expression `i+j` is evaluated and the breakpoint will be hit if the value for the expression `i+j` is true. The above command is the same as

```
debug> stopat prog1.ch 6 i+j
```

The command

```
debug> stopat prog1.ch 6 i+j 0
```

sets a breakpoint in file `prog1.ch` at line 6. When the breakpoint location in file prog1.ch at line 6 is reached, the expression `i+j` is evaluated and the breakpoint will be hit if the value for the expression `i+j` has changed. On the other hand, functions `chClearBreakPointLine()` `chClearBreakPointFunc()`, and `chClearBreakPointVar()` remove a breakpoint of line, function, and variable type in the linked list, respectively. Function `chClearBreakPoints()` removes all breakpoints in the debugger.

Functions `chAssign()` and `chCall()` in Program 7.16 correspond to commands `assign` and `call`, respectively. The contents of functions `chAssign()` and `chCall()` are the same. Inside these functions, an assignment expression such as `a=2*10` or function call expression such as `func()` is parsed by the API **Ch_ExprParse**() to check any syntax error. If the expression is valid, it is evaluated by the API **Ch_ExprEval**().

The argument of command `print` is an expression. The expression and its value, obtained by the API **Ch_ExprValue**(), are printed out by function `chPrintSymbolValue()` in Program 7.9. The expression of command `print` is processed inside the function `chPrint()`. Functions **Ch_ExprValue**() and **Ch_DeleteExprValue**() are used to obtain the value of expression of different data types. The calculated result of different data type is then displayed. One can also just type expression, its value will be printed out. A valid expression is processed by function `chExpr()`. Function `chExpr()` is similar to function function `chPrint()`, except that it will not display an error message when the expression is of a function with return type of void. As an example, commands

```
debug> assign i=2*10
debug> call func()
debug> print i
20
debug> 2*i
40
debug>
```

assign the variable `i` with the value of 10, call function `func()`, and print out the value of the expression `2*i` when the variable `i` is valid in its current scope.

The last function `chAbort()` in Program 7.16 corresponds to the command `abort`. The function `chAbort()` is similar to function `chEndDebug()` in Program 7.17. When the interpreter is executing the callback function which invokes the function `chAbort()`, the interpreter is aborted by the API **Ch_Abort**() for the command `abort`. The allocated memory for the debugger is then freed by the function `chCleanDebug()`.

The callback function `chCallback()` in Program 7.17 is the central component for the debugger program. The registered client data by the API **Ch_AddCallback**() points to the debugger. Statements

```
dgptr = (debugInfo_t *)clientdata;
dgptr->calldata = calldata;
dgptr->level = calldata->level;
```

retrieves the debugger first. The calldata and stack level are then saved in the debugger. The breakpoints in the linked list of the debugger is used inside the callback function.

If the callback is triggered by the function return event, it is first checked if the returning function had just been stopped by a breakpoint for function. If this is the case, the field `called` for this breakpoint is reset to 0 by the statement

```
bp->u.func.called = FALSE;
```

so that the program will stop at this function again when it is called.

If the callback is triggered by the `next` command to step over a function or a statement, the conditional expression of the if-statement

```
if(dgptr->runmode == RUNMODE_STEPOVER) {
```

will be true. If the callback is triggered by the function call event, the member `dgptr->stepovernum` will increment. If the callback is triggered by the function return event, the member `dgptr->stepovernum` will decrement. If the value of `dgptr->stepovernum` is not zero, the callback is triggered by a statement inside a function stepped over. Therefore, the flow of the program control returns from the callback function.

When the program is executed by the command `start, step, next,` or `cont`, the conditional expression of the if-statement

```
if(calldata->event & CH_MASKLINE)
```

will be true. If the statement is executed by the command `step` or `next`, the function `chProcessActionMenu()` will be called to obtain the user's input. The flow of the program control will return from the callback function.

If the program is executed by the command `start` or `cont`, all breakpoints in the linked list will be searched by a while-loop against the current executed statement. If a breakpoint was set for a file at the specified line number, the file name and line number in the breakpoint and current statement will be compared. If they are matched, the optional conditional expression for the breakpoint is checked by the function `chTestBreakPointCondition()`. If this function returns 0, the breakpoint is hit and the function `chProcessActionMenu()` will be called to process the user input. For a function breakpoint, it will test if the current function matches with the function set in the breakpoint. Only when the function is entered the first time, the breakpoint will be hit. When the first executable line of the function with a breakpoint is executed, the field `called` for the breakpoint is set to 1 by the statement

```
bp->u.func.called = TRUE;
```

so that the function will not be stopped during its execution. The field `called` will be set to 0 when the function returns so that the program will stop at the function again when it is called next time. For a breakpoint with a controlling variable, it will first find the address of the variable by the statement

```
varptr = (void*)Ch_SymbolAddrByName(interp, bp->u.var.varname);
```

If the variable is accessible within the current program scope, the value of the variable will be saved in `bp->u.var.varvalue` when it is encountered the first time. Otherwise, it will be checked against the

previously saved value. If the current value is different from the saved one, the new value will be saved and the breakpoint is hit.

Function `chTestBreakPointCondition()` uses the APIs **Ch_ExprParse**() and **Ch_ExprCalc**() to evaluate the conditional expression in `bp->condexpr`. The value of the expression is saved in `bp->condvalue`. The memory for `bp->condvalue` is allocated dynamically based on the data type and its size of the expression when the conditional expression is processed for the first time. If the member `bp->condtrue` is true, we only test if the conditional value is true or not. Otherwise, we test if the conditional value has changed. If the breakpoint should be hit, the function returns 0. Otherwise, it returns -1. The allocated memory for breakpoints are freed by the function `chClearBreakPoints()` called in function `chCleanDebug()` at the end of program execution.

7.4.1 A Sample Debugger Using C++

The above sample debugger has been ported to C++. The sample code is available at CHHOME/toolkit/demos/embedch/chapters/chapter7/cpp directory when Embedded Ch is installed.

```
/* File: debug.h, For the run time debug */
#ifndef _DEBUG_H
#define _DEBUG_H

#include <embedch.h>

#ifdef  __cplusplus
extern "C" {
#endif

/* define boolean value */
#define TRUE 1
#define FALSE 0

#define MAX_PROMPT_STR  2048 /* maximum prompt string length */
#define MAX_NUM_ARGS    100  /* maximum number of arguments for commands 'start' and 'run' */

/* action from the user through a GUI */
typedef enum {
    ACTION_UNDEFINED,
    /* for debug commands or menus in a GUI */
    ACTION_LOAD,             /* command "load filename" */
    ACTION_START,            /* command ""start [args]" with debugging  */
    ACTION_RUN,              /* command "run [args] without debugging  */
    ACTION_STEPINTO,         /* command "step" into */
    ACTION_STEPOVER,         /* command "next" to step over */
    ACTION_CONTINUE,         /* command "cont" to continue */
    ACTION_UPSTACK,          /* command "up" function stack */
    ACTION_DOWNSTACK,        /* command "down" function stack " */
    ACTION_STACK,            /* command "stack" function names " */
    ACTION_LOCALS,           /* command "locals" variables and values */
    ACTION_VARIABLES,        /* command "variables" variables and values in stacks */
    ACTION_ADDWATCHEXPR,     /* command "watch" to add watch expr */
    ACTION_REMOVEWATCHEXPR,  /* command "remove expr" remove a watch expr */
    ACTION_REMOVEWATCHEXPRS, /* command "remove" watch expressions */

    /* one may click a line in an edited code in GUI to set a new breakpoint */
    ACTION_SETBREAKPOINTLINE,/* command "stopat filename # [cond]" in file at line # */
    ACTION_SETBREAKPOINTFUNC,/* command "stopin funcname [cond]" in function */
    ACTION_SETBREAKPOINTVAR, /* command "stopvar varname [cond]" when the value of
                                a variable changes */
    /* one may use a popup dialog to clear a breakpoint */
    ACTION_CLEARBREAKPOINTLINE, /* command "clearline filename #" */
    ACTION_CLEARBREAKPOINTFUNC, /* command "clearfunc funcname" */
    ACTION_CLEARBREAKPOINTVAR,  /* command "clearvar varname" */
    ACTION_CLEARBREAKPOINTS,    /* command "clear" */
    ACTION_HELP,                /* command "help" */

    /* Use as commands in an interactive I/O window to
       assign a value, call a functoin, and print an expression */
    ACTION_ASSIGN,  /* command "assign var=expr" */
    ACTION_CALL,    /* command "call func()" */
    ACTION_PRINT,   /* command "print expr" */
    ACTION_EXPR,    /* command "expr" */

    ACTION_ABORT    /* command "abort" */
} action_t;
```

Program 7.13: The header file with defined data structures and function prototypes for the debug program (debug.h).

```
/* run mode */
typedef enum {
    RUNMODE_UNDEFINED,
    /* from commands in a GUI */
    RUNMODE_START,            /* command "start" with debugging  */
    RUNMODE_RUN,              /* command "run" without debugging  */
    RUNMODE_STEPINTO,         /* command "step" into */
    RUNMODE_STEPOVER,         /* command "next" to step over */
    RUNMODE_CONTINUE          /* command "cont" to continue */
} runmode_t;

/* The structure for a breakpoint */
typedef struct BreakPoint {
  action_t btype; /* the breakpoint type. One of values of
                     ACTION_SETBREAKPOINTLINE,
                     ACTION_SETBREAKPOINTFUNC,
                     ACTION_SETBREAKPOINTVAR. */
  union {
    struct {
      char *filename; /* file name */
      int linenum;    /* the line number for the breakpoint */
    }file;
    struct {
      char *funcname; /* function name */
      int called;     /* 1 if the func is called, reset to 0 when
                         function is returned */
    }func;
    struct {
      char *varname;  /* variable name */
      void *varvalue; /* value of variable */
    }var;
  }u;
  /* If the conditional expression for a breakpoint is NULL, or
     the conditional value is true or has changed,
     the breakpoint will be handled; otherwise, skip it */
  char *condexpr;  /* condition expression */
  void *condvalue; /* condition value */
  int condtrue; /* 1 for the conditional value is true to trigger the breakpoint
                   0 for the condtional value has changed to trigger the breakpoint */
  struct BreakPoint *next; /* next breakpoint in the linked list */
} breakPoint_t;

/* The structure for an expression in the watch list */
typedef struct WatchExpression {
  ChType_t exprtype; /* expression type, reserved for future use */
  char *exprname;    /* expression name */
  void *exprvalue;   /* expression value, reserved for future use */
  struct WatchExpression *next; /* next expression in the linked list */
} watchExpression_t;
```

Program 7.13: The header file with defined data structures and function prototypes for the debug program (debug.h) (Contd.)

```
/* The structure for information for debug */
typedef struct {
  char *progname;      /* a Ch program to be debugged */
  char *argvv[MAX_NUM_ARGS];  /* a Ch program with command line option to be debugged */
  ChInterp_t interp;   /* Ch interpreter */
  ChOptions_t *option; /* Ch initialization options for Ch_Initialize(interp, option) */
  void (*callback)(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);
  ChPointer_t clientdata;
  ChBlock_t *calldata; /* callback data */
  int count;           /* count value for the callback */
  int level;           /* stack level */
  runmode_t runmode;   /* run mode from a menu */
  breakPoint_t *bkpt;  /* the head of breakpoints */
  watchExpression_t *wexpr; /* the head of watch expressions */
  char *prevfuncname;  /* previous function name for handling "stop in" */
  int stepovernum;     /* number of functions stepped over by "step over" */
} debugInfo_t;

/* functions in stack.c */
/* static functions within the file
static int printValue(ChInterp_t interp, ChType_t datatype,
               void *addr, ChUserDefinedTag_t udtag, int nestlevel);
static int printUserDefined(ChInterp_t interp, ChType_t datatype,
               void *addr, ChUserDefinedTag_t udtag, int nestlevel); */
extern void chPrintSymbolValue(ChInterp_t interp, char *name, void *addr);
extern void chPrintSymbolValuesInStack(ChInterp_t interp, int level);

/* functions in callback.c */
extern void chCallback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdat
a);
extern int chTestBreakPointCondition(ChInterp_t interp, breakPoint_t *bp);

/* functions in menu.c, the console can be replaced by a GUI */
extern int chHelp(void);

/* functions in debug.c */
/* static functions within the file
static action_t getActionMenu(debugInfo_t *dgptr, char *str);
static void printErrorNoFile(void);
static int getArgvFromStr(char **argv, char *args); */
extern int chInitDebug(debugInfo_t *dgptr);
extern int chEndDebug(debugInfo_t *dgptr);
extern int chCleanDebug(debugInfo_t *dgptr);
extern int chLoadFile(debugInfo_t *dgptr, char *progname);
extern int chStart(debugInfo_t *dgptr, runmode_t runmode, char *args);
extern int chRun(debugInfo_t *dgptr, char *args);
extern int chStepInto(debugInfo_t *dgptr);
extern int chStepOver(debugInfo_t *dgptr);
extern int chContinue(debugInfo_t *dgptr);
extern int chUpStack(debugInfo_t *dgptr);
extern int chDownStack(debugInfo_t *dgptr);
extern int chStack(debugInfo_t *dgptr);
extern int chLocals(debugInfo_t *dgptr);
extern int chVariables(debugInfo_t *dgptr);
```

Program 7.13: The header file with defined data structures and function prototypes for the debug program (debug.h) (Contd.)

```
extern int chAddWatchExpr(debugInfo_t *dgptr, char *exprname);
extern int chProcessWatchExprs(debugInfo_t *dgptr);
extern int chRemoveWatchExpr(debugInfo_t *dgptr, char *exprname);
extern int chRemoveWatchExprs(debugInfo_t *dgptr);
extern int chSetBreakPointLine(debugInfo_t *dgptr, char *filename, int linenum,
                               char *condexpr, int condtrue);
extern int chSetBreakPointFunc(debugInfo_t *dgptr, char *funcname,
                               char *condexpr, int condtrue);
extern int chSetBreakPointVar(debugInfo_t *dgptr, char *varname,
                               char *condexpr, int condtrue);
extern int chClearBreakPointLine(debugInfo_t *dgptr, char *filename, int linenum);
extern int chClearBreakPointFunc(debugInfo_t *dgptr, char *funcname);
extern int chClearBreakPointVar(debugInfo_t *dgptr, char *varname);
extern int chClearBreakPoints(debugInfo_t *dgptr);
extern int chAssign(debugInfo_t *dgptr, char *expr);
extern int chCall(debugInfo_t *dgptr, char *expr);
extern int chPrint(debugInfo_t *dgptr, char *expr);
extern int chExpr(debugInfo_t *dgptr, char *expr);
extern int chAbort(debugInfo_t *dgptr);
/* function below for Menu processing */
extern int chProcessActionMenu(debugInfo_t *dgptr);

#ifdef  __cplusplus
}
#endif

#endif  /* _DEBUG_H*/
```

Program 7.13: The header file with defined data structures and function prototypes for the debug program (debug.h) (Contd.)

```
/* File Name: prog6.c */
#include "debug.h"

int main() {
    chHelp();
    while (1) {
      debugInfo_t debug;

      chInitDebug(&debug);
      chProcessActionMenu(&debug);
      chEndDebug(&debug);
    }
    return 0;
}
```

Program 7.14: The debug program (prog6.c).

```
/* File Name menu.c */
#include <stdio.h>
#include <string.h>
#include "debug.h"

int chHelp() {
    printf("************************ Debug Menu ************************\n");
    printf("load filename            load 'filename' for debugging\n");
    printf("start [args]             start the program with debugging\n");
    printf("run [args]               run the program without debugging\n");
    printf("step                     step into a function or next line\n");
    printf("next                     step over a function or next line\n");
    printf("cont                     continue till hitting a breakpoint or ends\n");
    printf("up                       change stack to the calling function\n");
    printf("down                     change stack to the called function\n");
    printf("stack                    display stack names in all stacks\n");
    printf("locals                   display variables and values within its scope\n");
    printf("variables                display variables and values in all stacks\n");
    printf("watch expr               add an expression into the watch list \n");
    printf("remove expr              remove an expression from the watch list \n");
    printf("remove                   remove all expressions from the watch list \n");
    printf("stopat filename # [cond] set a new breakpoint in a file at line #\n");
    printf("stopin funcname [cond]   set a new breakpoint in a function \n");
    printf("stopvar varname [cond]   set a new breakpoint for a controlling variable\n");
    printf("clearline filename #     clear a breakpoint in a file at line #\n");
    printf("clearfunc funcname       clear a breakpoint for a function \n");
    printf("clearvar varname         clear a breakpoint for a variable \n");
    printf("clear                    clear all breakpoints \n");
    printf("help                     display this debug menu\n");
    printf("assign var=expr          assign a value to a variable\n");
    printf("call func()              call a function \n");
    printf("print expr               print out the value of an expression\n");
    printf("expr                     print out the value of an expression\n");
    printf("abort                    abort the debugger\n");
    printf("............................................................\n");
    return 0;
}
```

Program 7.15: The user interface for the debug program (menu.c).

```
/* File Name: debug.c */
#include <stdio.h>
#include <string.h>
#include "debug.h"

#if defined(_WIN32)
#define R_OK    4       /* Test for Read permission */
#define F_OK    0       /* Test for existence of File */
#else
#include <unistd.h>
#endif

/* static functions within the file */
static void printErrorNoFile(void);
static action_t getActionMenu(debugInfo_t *dgptr, char *str);
static int getArgvFromStr(char **argv, char *args);

int chInitDebug(debugInfo_t *dgptr) {
    char *proghome;
    ChOptions_t *option;

    setbuf(stdout, NULL); /* no buffer for debugging purpose */
    setbuf(stderr, NULL); /* no buffer for debugging purpose */
    dgptr->progname = NULL;
    memset(dgptr->argvv, '\0', MAX_NUM_ARGS*sizeof(char*));
    dgptr->interp = NULL;
    /* To use your distribution of Embedded Ch, modify code here to setup dgptr->interp:
       option = (ChOptions_t*)malloc(sizeof(ChOptions_t));
       option->shelltype = CH_REGULARCH;
       proghome = getenv("PROG_HOME"); // from an environment variable
      // or obtain the value for proghome from a registry value in Windows,
      //   using Windows API RegOpenKeyEx() and RegQueryValueEx().
      option->chhome = (char *)malloc(strlen(proghome)+ strlen("/embedch")+1);
      strcpy(option->chhome, proghome);
      strcat(option->chhome, "/embedch");
      dgptr->option = option;
    */
    dgptr->option = NULL;
    dgptr->callback = chCallback;
    dgptr->clientdata = (ChPointer_t)dgptr;
    dgptr->count = 0;
    dgptr->level = 0;
    dgptr->runmode = RUNMODE_UNDEFINED;
    dgptr->bkpt = NULL;
    dgptr->wexpr = NULL;
    dgptr->stepovernum = 0;
    return 0;
}

int chEndDebug(debugInfo_t *dgptr) {
    if(dgptr->interp)
       Ch_End(dgptr->interp);
    chCleanDebug(dgptr);
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c).

```
int chCleanDebug(debugInfo_t *dgptr) {
    int i =1;
    watchExpression_t *wexpr, *head;

    if(dgptr->progname)
      free(dgptr->progname);
    while(dgptr->argvv[i] != NULL) {
      free(dgptr->argvv[i]);
      i++;
    }
    if(dgptr->option) {
      if(dgptr->option->chhome)
        free(dgptr->option->chhome);
      free(dgptr->option);
    }

    /* clean the list of watched expressions and breakpoints*/
    chRemoveWatchExprs(dgptr);
    chClearBreakPoints(dgptr);
    return 0;
}

int chLoadFile(debugInfo_t *dgptr, char *progname) {
    if(access(progname, F_OK) || access(progname, R_OK)) {
        printf("Error: file \"%s\" does not exist or not readable\n", progname);
        return -1;
    }
    dgptr->progname = strdup(progname);
    return 0;
}

int chStart(debugInfo_t *dgptr, runmode_t runmode, char *args) {
    int mask, status;
    ChInterp_t interp;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    /* create dgptr->argvv from args for Ch_ParseScript() */
    dgptr->argvv[0] = dgptr->progname;
    getArgvFromStr(dgptr->argvv, args);

    dgptr->runmode = runmode;
    Ch_Initialize(&interp, dgptr->option);
    dgptr->interp = interp;
    mask = CH_MASKLINE | CH_MASKRET;
    Ch_AddCallback(dgptr->interp,  mask, dgptr->callback,
                   dgptr->clientdata, dgptr->count);
    status = Ch_ParseScript(interp,  dgptr->argvv);
    if(status == CH_OK)
       status = Ch_ExecScript(interp,  dgptr->argvv[0]);
    return status;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chRun(debugInfo_t *dgptr, char *args) {
    int mask, status;
    ChInterp_t interp;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    /* create dgptr->argvv from args for Ch_ParseScript() */
    dgptr->argvv[0] = dgptr->progname;
    getArgvFromStr(dgptr->argvv, args);

    dgptr->runmode = RUNMODE_RUN;
    mask = CH_MASKNONE;
    if(dgptr->interp != NULL) {/* already start by chStart() */
      status = Ch_AddCallback(dgptr->interp,  mask, dgptr->callback,
                              dgptr->clientdata, dgptr->count);
    }
    else {
      Ch_Initialize(&interp, dgptr->option);
      dgptr->interp = interp;
      Ch_AddCallback(dgptr->interp,  mask, dgptr->callback,
                     dgptr->clientdata, dgptr->count);
      status = Ch_ParseScript(interp,  dgptr->argvv);
      if(status == CH_OK)
         status = Ch_ExecScript(interp,  dgptr->argvv[0]);
    }
    return status;
}

int chStepInto(debugInfo_t *dgptr) {
    int mask;

    dgptr->runmode = RUNMODE_STEPINTO;
    if(dgptr->interp == NULL)  {
       return chStart(dgptr, RUNMODE_STEPINTO, NULL);
    }
    mask = CH_MASKLINE | CH_MASKRET;
    return Ch_AddCallback(dgptr->interp,  mask, dgptr->callback,
                          dgptr->clientdata, dgptr->count);
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chStepOver(debugInfo_t *dgptr){
    int mask;

    dgptr->runmode = RUNMODE_STEPOVER;
    if(dgptr->interp == NULL)  {
       return chStart(dgptr, RUNMODE_STEPOVER, NULL);
    }
    mask = CH_MASKLINE | CH_MASKCALL | CH_MASKRET;
    return Ch_AddCallback(dgptr->interp,  mask, dgptr->callback,
                          dgptr->clientdata, dgptr->count);
}

int chContinue(debugInfo_t *dgptr) {
    int mask;

    dgptr->runmode = RUNMODE_CONTINUE;
    if(dgptr->interp == NULL)  {
       return chStart(dgptr, RUNMODE_CONTINUE, NULL);
    }
    mask = CH_MASKLINE | CH_MASKRET;
    return Ch_AddCallback(dgptr->interp,  mask, dgptr->callback,
                          dgptr->clientdata, dgptr->count);
}

int chUpStack(debugInfo_t *dgptr) {
    ChBlock_t calldata;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    if(Ch_ChangeStack(dgptr->interp, ++(dgptr->level), &calldata) == CH_OK) {
      /* if for a GUI-based debugger, update the edited source code in a GUI
         window using info in members of calldata and dgptr->calldata here */
      printf("The current line number is %d\n", dgptr->calldata->linecurrent);
      printf("The line number in the calling function %d\n", calldata.linecurrent);
      return 0;
    }
    else
      return -1;
}

int chDownStack(debugInfo_t *dgptr) {
    ChBlock_t calldata;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    if(Ch_ChangeStack(dgptr->interp, --(dgptr->level), &calldata) == CH_OK) {
      return 0;
    }
    else
      return -1;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chStack(debugInfo_t *dgptr) {
    ChInterp_t interp;
    int level, isfunc;
    char *name, *classname;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    interp = dgptr->interp;
    level= dgptr->level;
    printf("Stack Level  Stack Name\n");
    while(name = Ch_StackName(interp, level, &isfunc, &classname))
    {
       if(isfunc == 1)
         printf("%4d           %s()%\n", level, name);
       else if(isfunc == 2)
         printf("%4d           %s::%s()%\n", level, classname, name);
       else
         printf("%4d           %s%\n", level, name);
       level++;
    }
    return 0;
}

int chLocals(debugInfo_t *dgptr) {
    ChInterp_t interp;
    int level;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    interp = dgptr->interp;
    level= dgptr->level;
    chPrintSymbolValuesInStack(interp, level);
    return 0;
}

int chVariables(debugInfo_t *dgptr) {
    ChInterp_t interp;
    int level;
    ChBlock_t calldata2;

    if(dgptr->progname == NULL) {
       printErrorNoFile();
       return -1;
    }
    interp = dgptr->interp;
    level= dgptr->level;
    chPrintSymbolValuesInStack(interp, level);
    while(Ch_ChangeStack(interp, ++level, &calldata2) == CH_OK)
    {
       chPrintSymbolValuesInStack(interp, level);
    }
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
/* add an expression or variable to the watched list */
int chAddWatchExpr(debugInfo_t *dgptr, char *exprname) {
    watchExpression_t *temp = dgptr->wexpr;
    watchExpression_t *wexpr = (watchExpression_t *)malloc(sizeof(struct WatchExpression));

    wexpr->exprtype = CH_UNDEFINETYPE; /* not used */
    wexpr->exprname = strdup(exprname);
    wexpr->exprvalue = NULL;           /* not used */
    wexpr->next = NULL;
    if(temp == NULL) /* no element in the linked list */
      dgptr->wexpr = wexpr;
    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 = wexpr;
    }
    return 0;
}

/* pricocess expressions and variables in the watched list */
int chProcessWatchExpr(debugInfo_t *dgptr) {
    watchExpression_t *wexpr = dgptr->wexpr;

    while(wexpr) {
      chPrint(dgptr, wexpr->exprname);
      wexpr = wexpr->next;
    }
    return 0;
}

/* remove an expr from the watched list */
int chRemoveWatchExprs(debugInfo_t *dgptr, char *exprname) {
    watchExpression_t *tmp1, *tmp2;

    tmp1 = dgptr->wexpr;
    if (tmp1 == NULL)
        return 0;
    else if (!strcmp(tmp1->exprname, exprname))
    {
        dgptr->wexpr = tmp1->next;
        free(tmp1->exprname);
        if(tmp1->exprvalue)
             free(tmp1->exprvalue);
        free(tmp1);
    }
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
    else {
        while (tmp1->next)
        {
          if (!strcmp(tmp1->exprname, exprname))
          {
            tmp2 = tmp1->next;
            tmp1->next = tmp1->next->next;
            free(tmp2->exprname);
            if(tmp2->exprvalue)
               free(tmp2->exprvalue);
            free(tmp2);
            return 0;
          }
          tmp1 = tmp1->next;
        }
    }
    return 0;
}

/* remove all exprs from the watched list */
int chRemoveWatchExprs(debugInfo_t *dgptr) {
    watchExpression_t *tmp1, *head;

    head = dgptr->wexpr;
    while(head) {
       tmp1 = head;
       head = head->next;
       free(tmp1->exprname);
       if(tmp1->exprvalue)
           free(tmp1->exprvalue);
    }
    dgptr->wexpr = NULL;
    return 0;
}

int chSetBreakPointLine(debugInfo_t *dgptr, char *filename, int linenum,
                        char *condexpr, int condtrue) {
    breakPoint_t *temp = dgptr->bkpt;
    breakPoint_t *bp = (breakPoint_t *)malloc(sizeof(struct BreakPoint));

    bp->btype = ACTION_SETBREAKPOINTLINE;
    bp->u.file.filename = strdup(filename);
    bp->u.file.linenum = linenum;
    if(condexpr && strlen(condexpr) != 0)
      bp->condexpr = strdup(condexpr);
    else
      bp->condexpr = NULL;
    bp->condvalue = NULL;
    bp->condtrue = condtrue;
    bp->next = NULL;

    if(temp == NULL) /* no element in the linked list */
      dgptr->bkpt = bp;
    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 = bp;
    }
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chSetBreakPointFunc(debugInfo_t *dgptr, char *funcname, char *condexpr,
                        int condtrue) {
    breakPoint_t *temp = dgptr->bkpt;
    breakPoint_t *bp = (breakPoint_t *)malloc(sizeof(struct BreakPoint));

    bp->btype = ACTION_SETBREAKPOINTFUNC;
    bp->u.func.funcname = strdup(funcname);
    bp->u.func.called = FALSE;
    if(condexpr && strlen(condexpr) != 0)
      bp->condexpr = strdup(condexpr);
    else
      bp->condexpr = NULL;
    bp->condvalue = NULL;
    bp->condtrue = condtrue;
    bp->next = NULL;

    if(temp == NULL) /* no element in the linked list */
      dgptr->bkpt = bp;
    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 = bp;
    }
    return 0;
}

int chSetBreakPointVar(debugInfo_t *dgptr, char *varname, char *condexpr,
                       int condtrue) {
    breakPoint_t *temp = dgptr->bkpt;
    breakPoint_t *bp = (breakPoint_t *)malloc(sizeof(struct BreakPoint));

    bp->btype = ACTION_SETBREAKPOINTVAR;
    bp->u.var.varname = strdup(varname);
    bp->u.var.varvalue = NULL;
    if(condexpr && strlen(condexpr) != 0)
      bp->condexpr = strdup(condexpr);
    else
      bp->condexpr = NULL;
    bp->condvalue = NULL;
    bp->condtrue = condtrue;
    bp->next = NULL;

    if(temp == NULL) /* no element in the linked list */
      dgptr->bkpt = bp;
    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 = bp;
    }
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chClearBreakPointLine(debugInfo_t *dgptr, char *filename, int linenum) {
    breakPoint_t *tmp1, *tmp2;

    tmp1 = dgptr->bkpt;
    if (tmp1 == NULL)
      return 0;
    else if (!strcmp(tmp1->u.file.filename, filename) &&
              tmp1->u.file.linenum == linenum)
    {
      dgptr->bkpt = tmp1->next;
      free(tmp1->u.file.filename);
      if(tmp1->condexpr)
        free(tmp1->condexpr);
      if(tmp1->condvalue)
        free(tmp1->condvalue);
      free(tmp1);
    }
    else {
      while (tmp1->next) {
        if (!strcmp(tmp1->next->u.file.filename, filename) &&
             tmp1->next->u.file.linenum == linenum)
        {
          tmp2 = tmp1->next;
          tmp1->next = tmp1->next->next;
          free(tmp2->u.file.filename);
          if(tmp2->condexpr)
            free(tmp2->condexpr);
          if(tmp2->condvalue)
            free(tmp2->condvalue);
          free(tmp2);
          return 0;
        }
        tmp1 = tmp1->next;
      }
    }
    return 0;
}

/* remove a breakpoint from the list for a function */
int chClearBreakPointFunc(debugInfo_t *dgptr, char *funcname) {
    breakPoint_t *tmp1, *tmp2;

    tmp1 = dgptr->bkpt;
    if (tmp1 == NULL)
      return 0;
    else if (!strcmp(tmp1->u.func.funcname, funcname))
    {
      dgptr->bkpt = tmp1->next;
      free(tmp1->u.func.funcname);
      if(tmp1->condexpr)
        free(tmp1->condexpr);
      if(tmp1->condvalue)
        free(tmp1->condvalue);
      free(tmp1);
    }
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
    else {
      while (tmp1->next)
      {
        if (!strcmp(tmp1->next->u.func.funcname, funcname))
        {
          tmp2 = tmp1->next;
          tmp1->next = tmp1->next->next;
          free(tmp2->u.func.funcname);
          if(tmp2->condexpr)
            free(tmp2->condexpr);
          if(tmp2->condvalue)
            free(tmp2->condvalue);
          free(tmp2);
          return 0;
        }
        tmp1 = tmp1->next;
      }
    }
    return 0;
}

/* remove a breakpoint from the list for a monitoring variable */
int chClearBreakPointVar(debugInfo_t *dgptr, char *varname) {
    breakPoint_t *tmp1, *tmp2;

    tmp1 = dgptr->bkpt;
    if (tmp1 == NULL)
       return 0;
    else if (!strcmp(tmp1->u.var.varname, varname)) {
      dgptr->bkpt = tmp1->next;
      free(tmp1->u.var.varname);
      if(tmp1->condexpr)
        free(tmp1->condexpr);
      if(tmp1->condvalue)
        free(tmp1->condvalue);
      free(tmp1);
    }
    else {
      while (tmp1->next) {
        if (!strcmp(tmp1->next->u.var.varname, varname)) {
          tmp2 = tmp1->next;
          tmp1->next = tmp1->next->next;
          free(tmp2->u.var.varname);
          if(tmp2->condexpr)
            free(tmp2->condexpr);
          if(tmp2->condvalue)
            free(tmp2->condvalue);
          free(tmp2);
          return 0;
        }
        tmp1 = tmp1->next;
      }
    }
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chClearBreakPoints(debugInfo_t *dgptr) {
    breakPoint_t *bp, *head;

    head = dgptr->bkpt;
    while(head) {
      bp = head;
      head = head->next;
      if(bp->btype == ACTION_SETBREAKPOINTLINE)
        free(bp->u.file.filename);
      else if(bp->btype == ACTION_SETBREAKPOINTFUNC)
        free(bp->u.func.funcname);
      else if(bp->btype == ACTION_SETBREAKPOINTVAR) {
        free(bp->u.var.varname);
        if(bp->u.var.varvalue)
           free(bp->u.var.varvalue);
      }
      if(bp->condexpr)
        free(bp->condexpr);
      if(bp->condvalue)
        free(bp->condvalue);
      free(bp);
    }
    dgptr->bkpt = NULL;
    return 0;
}


int chAssign(debugInfo_t *dgptr, char *expr) {
    ChInterp_t interp;
    int retval;

    interp = dgptr->interp;
    if(interp == NULL || Ch_ExprParse(interp, expr) == CH_ERROR)
       return -1;
    retval = Ch_ExprEval(interp, expr);
    return retval;
}

int chCall(debugInfo_t *dgptr, char *expr) {
    ChInterp_t interp;
    int retval;

    interp = dgptr->interp;
    if(interp == NULL || Ch_ExprParse(interp, expr) == CH_ERROR)
       return -1;
    retval = Ch_ExprEval(interp, expr);
    return retval;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
int chPrint(debugInfo_t *dgptr, char *expr) {
    ChInterp_t interp;
    ChValueNode_t valuenode;
    void *addr;
    int retval =0;

    interp = dgptr->interp;
    valuenode = Ch_ExprValue(interp, expr, &addr);
    if(valuenode != NULL) {
       chPrintSymbolValue(interp, expr, addr);
       Ch_DeleteExprValue(interp, valuenode);
    }
    else
       retval = -1;
    return retval;
}

int chExpr(debugInfo_t *dgptr, char *expr) {
    ChInterp_t interp;
    ChValueNode_t valuenode;
    void *addr;
    int retval =0;

    interp = dgptr->interp;
    valuenode = Ch_ExprValue(interp, expr, &addr);
    if(valuenode != NULL) {
       if(Ch_DataType(interp, expr) != CH_VOIDTYPE)
         chPrintSymbolValue(interp, expr, addr);
       Ch_DeleteExprValue(interp, valuenode);
    }
    else
       retval = -1;
    return retval;
}

int chAbort(debugInfo_t *dgptr) {
    int i =1;

    if(dgptr->interp)
       Ch_Abort(dgptr->interp);
    chCleanDebug(dgptr);
    exit(0);
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
/* process the user input interactively. This action menu is
best implemented using a GUI */
int chProcessActionMenu(debugInfo_t *dgptr) {
    action_t act;
    int abort = FALSE, linenum, condtrue=TRUE;
    char name[MAX_PROMPT_STR], line[MAX_PROMPT_STR], condexpr[MAX_PROMPT_STR]={'\0'}, *ptr;

    while(abort==FALSE) {
        act = getActionMenu(dgptr, line);
        switch(act) {
            case ACTION_LOAD:
              ptr = line+strlen("load ");
              chLoadFile(dgptr, ptr);
              break;
            case ACTION_START:
              ptr = line+strlen("start");
              chStart(dgptr, RUNMODE_START, ptr);
              abort = TRUE;
              break;
            case ACTION_RUN:
              chRun(dgptr, line+strlen("run"));
              abort = TRUE;
              break;
            case ACTION_STEPINTO:
              chStepInto(dgptr);
              abort = TRUE;
              break;
            case ACTION_STEPOVER:
              chStepOver(dgptr);
              abort = TRUE;
              break;
            case ACTION_CONTINUE:
              chContinue(dgptr);
              abort = TRUE;
              break;
            case ACTION_UPSTACK:
              chUpStack(dgptr);
              break;
            case ACTION_DOWNSTACK:
              chDownStack(dgptr);
              break;
            case ACTION_STACK:
              chStack(dgptr);
              break;
            case ACTION_LOCALS:
              chLocals(dgptr);
              break;
            case ACTION_VARIABLES:
              chVariables(dgptr);
              break;
            case ACTION_ADDWATCHEXPR:
              ptr = line+strlen("watch ");
              chAddWatchExpr(dgptr, ptr);
              break;
            case ACTION_REMOVEWATCHEXPR:
              ptr = line+strlen("remove ");
              chRemoveWatchExpr(dgptr, ptr);
              break;
            case ACTION_REMOVEWATCHEXPRS:
              chRemoveWatchExprs(dgptr);
              break;
```


Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
            case ACTION_SETBREAKPOINTLINE:
              ptr = line+strlen("stopat ");
              sscanf(ptr, "%s%d%s%d", name, &linenum, condexpr, condtrue);
              chSetBreakPointLine(dgptr, name, linenum, condexpr, condtrue);
              break;
            case ACTION_SETBREAKPOINTFUNC:
              ptr = line+strlen("stopin ");
              sscanf(ptr, "%s%s%d", name, condexpr, condtrue);
              chSetBreakPointFunc(dgptr, name, condexpr, condtrue);
              break;
            case ACTION_SETBREAKPOINTVAR:
              ptr = line+strlen("stopvar ");
              sscanf(ptr, "%s%s%d", name, condexpr, condtrue);
              chSetBreakPointVar(dgptr, name, condexpr, condtrue);
              break;
            case ACTION_CLEARBREAKPOINTLINE:
              ptr = line+strlen("clearline ");
              sscanf(ptr, "%s%d", name, &linenum);
              chClearBreakPointLine(dgptr, name, linenum);
              break;
            case ACTION_CLEARBREAKPOINTFUNC:
              ptr = line+strlen("clearfunc ");
              chClearBreakPointFunc(dgptr, ptr);
              break;
            case ACTION_CLEARBREAKPOINTVAR:
              ptr = line+strlen("clearvar ");
              chClearBreakPointVar(dgptr, ptr);
              break;
            case ACTION_CLEARBREAKPOINTS:
              chClearBreakPoints(dgptr);
              break;
            case ACTION_HELP:
              chHelp();
              break;
            case ACTION_ASSIGN:
              ptr = line+strlen("assign ");
              if(chAssign(dgptr, ptr) == CH_ERROR)
                printf("Error: \"%s\" is not valid in this scope\n", ptr);
              break;
            case ACTION_CALL:
              ptr = line+strlen("call ");
              if(chCall(dgptr, ptr) == CH_ERROR)
                printf("Error: \"%s\" is not defined in this scope\n", ptr);
              break;
            case ACTION_PRINT:
              ptr = line+strlen("print ");
              if(chPrint(dgptr, ptr) == CH_ERROR)
                printf("Error: \"%s\" is not defined in this scope\n", ptr);
              break;
            case ACTION_EXPR:
              if(chExpr(dgptr, line) == CH_ERROR)
                printf("Error: \"%s\" is not defined in this scope\n", line);
              break;
            case ACTION_ABORT:
              chAbort(dgptr);
              break;
            default: /* ACTION_UNDEFINED */
              ptr = strchr(line, ' ');
              if(ptr!=NULL)
                 *ptr = '\0';
              printf("%s: not found (type 'help')\n", line);
              break;
        }
    }
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
/*-------------------------------------------------------
* Get the action item from user' menu
* return a proper enum in enum Action --------- successful
* return ACTION_UNDEFINED             --------- failed
*-------------------------------------------------------------*/
action_t getActionMenu(char *str) {
    action_t act;
    const char *prompt = "debug> ";

    printf("%s", prompt);
    fgets(str,MAX_PROMPT_STR,stdin);
    str[strlen(str)-1] = '\0';
    if(!strncmp(str, "load ", strlen("load "))) {
        act = ACTION_LOAD;
    }
    else if(!strncmp(str, "start", strlen("start"))) {
        act = ACTION_START;
    }
    else if(!strncmp(str, "run", strlen("run"))) {
        act = ACTION_RUN;
    }
    else if(!strcmp(str, "step")) {
        act = ACTION_STEPINTO;
    }
    else if(!strcmp(str, "next")) {
        act = ACTION_STEPOVER;
    }
    else if(!strcmp(str, "cont")) {
        act = ACTION_CONTINUE;
    }
    else if(!strcmp(str, "up")) {
        act = ACTION_UPSTACK;
    }
    else if(!strcmp(str, "down")) {
        act = ACTION_DOWNSTACK;
    }
    else if(!strcmp(str, "stack")) {
        act = ACTION_STACK;
    }
    else if(!strcmp(str, "locals")) {
        act = ACTION_LOCALS;
    }
    else if(!strcmp(str, "variables")) {
        act = ACTION_VARIABLES;
    }
    else if(!strncmp(str, "watch ", strlen("watch "))) {
        act = ACTION_ADDWATCHEXPR;
    }
    else if(!strcmp(str, "remove")) {
        act = ACTION_REMOVEWATCHEXPRS;
    }
    else if(!strncmp(str, "remove ", strlen("remove "))) {
        act = ACTION_REMOVEWATCHEXPR;
    }
    else if(!strncmp(str, "stopat ", strlen("stopat "))) {
        act = ACTION_SETBREAKPOINTLINE;
    }
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
    else if(!strncmp(str, "stopin ", strlen("stopin "))) {
        act = ACTION_SETBREAKPOINTFUNC;
    }
    else if(!strncmp(str, "stopvar ", strlen("stopvar "))) {
        act = ACTION_SETBREAKPOINTVAR;
    }
    else if(!strncmp(str, "clearline ", strlen("clearline "))) {
        act = ACTION_CLEARBREAKPOINTLINE;
    }
    else if(!strncmp(str, "clearfunc ", strlen("clearfunc "))) {
        act = ACTION_CLEARBREAKPOINTFUNC;
    }
    else if(!strncmp(str, "clearvar ", strlen("clearvar "))) {
        act = ACTION_CLEARBREAKPOINTVAR;
    }
    else if(!strcmp(str, "clear")) {
        act = ACTION_CLEARBREAKPOINTS;
    }
    else if(!strcmp(str, "help")) {
        act = ACTION_HELP;
    }
    else if(!strncmp(str, "assign ", strlen("assign "))) {
        act = ACTION_ASSIGN;
    }
    else if(!strncmp(str, "call ", strlen("call "))) {
        act = ACTION_CALL;
    }
    else if(!strncmp(str, "print ", strlen("print "))) {
        act = ACTION_PRINT;
    }
    else if(interp != NULL && Ch_ExprParse(interp, str) == CH_OK) {
        act = ACTION_EXPR;
    }
    else if(!strcmp(str, "abort")) {
        act = ACTION_ABORT;
    }
    else {
        act = ACTION_UNDEFINED;
    }
    return act;
}

void printErrorNoFile(void) {
     /* use MessageBox() for GUI in Windows */
     printf("Error: no program to be debugged\n");
}

/* this function creates char *argvv[] = {"prog.ch", "opt1", "opt2", "opt3", NULL}
    from command 'start opt1 opt2 opt3' dynamically. It also handles argument within
    double quotation marks 'start opt1 "opt2 with space" opt3' */
int getArgvFromStr(char **argv, char *args) {
    int i = 1;
    char *str, *token, buf[MAX_PROMPT_STR];

    str = args;
    while(str!=NULL && *str==' ') { /* skip the blank space */
        str++;
    }
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
    if(str == NULL || *str == '\0')
        return 0;
    if(*str == '"') {
    }
    token =strtok(str, " \t"); /* tab and space as delimiters */
    if(*token == '"')
        { /* start "str1 str2" str3 */
          strcpy(buf, token+1); /* kickout '"' and get "str1" */
          token = strtok(NULL, "\"");
          if(token != NULL) /* not  start "str " or start "str"  */
          {
            strcat(buf, " ");  /* \t is treated as space */
            strcat(buf, token); /* get "str1 str2" */
            if(token[strlen(token)+1] != '\0' && /* not finished */
               token[strlen(token)+1] != ' ' &&  /* not space */
               token[strlen(token)+1] != '\t')   /* not tab */
            {
              token = strtok(NULL, " \t");
              strcat(buf, token); /* get "str1 str2str3" */
            }
          }
          token = buf;
    }
    argv[i++] = strdup(token);

    do      /* process the rest token of the command line */
    {
      token =strtok(NULL, " \t"); /* tab and space as delimiters */
      if(token != NULL)
      {
        if(*token == '"')
        { /* start "str1 str2" str3 */
          strcpy(buf, token+1); /* kickout '"' and get "str1" */
          token = strtok(NULL, "\"");
          if(token != NULL) /* not  start "str " or start "str"  */
          {
            strcat(buf, " ");  /* \t is treated as space */
            strcat(buf, token); /* get "str1 str2" */
            if(token[strlen(token)+1] != '\0' && /* not finished */
               token[strlen(token)+1] != ' ' &&  /* not space */
               token[strlen(token)+1] != '\t')   /* not tab */
            {
              token = strtok(NULL, " \t");
              strcat(buf, token); /* get "str1 str2str3" */
            }
          }
          token = buf;
        }
        argv[i++] = strdup(token);
      }
    }while(token && i < MAX_NUM_ARGS);
    return 0;
}
```

Program 7.16: Functions corresponding to the debug user interface menu. (debug.c) (Contd.).

```
/* File Name: callback.c */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <embedch.h>
#include "debug.h"

void chCallback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata) {
    debugInfo_t *dgptr;
    breakPoint_t *bp;
    void *varptr;
    size_t size;

    dgptr = (debugInfo_t *)clientdata;
    dgptr->calldata = calldata;
    dgptr->level = calldata->level;
    bp = dgptr->bkpt;

    /* reset a called function with a breakpoint */
    if(calldata->event & CH_MASKRET)  { /* return from a function */
      while(bp) {
        if(bp->btype == ACTION_SETBREAKPOINTFUNC)
        {
          if (calldata->funcname != NULL &&
              bp->u.func.called == TRUE&& /* return from this function */
              !strcmp(bp->u.func.funcname, calldata->funcname))
          {
               bp->u.func.called = FALSE; /* reset */
          }
        }
        bp = bp->next;
      }
      bp = dgptr->bkpt;
    }

    /* for "step over" functions */
    if(dgptr->runmode == RUNMODE_STEPOVER) {
       if(calldata->event & CH_MASKCALL)  { /* step over a new function */
          dgptr->stepovernum += 1;
       }
       else if(calldata->event & CH_MASKRET)  {
          if(dgptr->stepovernum > 0) /* returning from a function stepped over */
            dgptr->stepovernum -= 1;
       }
       /* executing statements inside a function stepped over */
       if(dgptr->stepovernum != 0)
          return;
    }
```

Program 7.17: The callback for the debug program (callback.c).

```
    /* for "start", "step into", "step over", and "cont" */
    if(calldata->event & CH_MASKLINE)
    {
      if(dgptr->runmode == RUNMODE_STEPINTO || dgptr->runmode == RUNMODE_STEPOVER) {
         chProcessWatchExprs(dgptr);
         chProcessActionMenu(dgptr);
         return;
      }
      /* process the list of breakpoints */
      while(bp) {
        if(bp->btype == ACTION_SETBREAKPOINTLINE) {
          if (calldata->source != NULL &&
              !strcmp(bp->u.file.filename, calldata->source) &&
              bp->u.file.linenum == calldata->linecurrent)
          {
             /* check the conditional expr for each breakpoint if it exists */
             if(chTestBreakPointCondition(interp, bp) == 0) {
               chProcessWatchExprs(dgptr);
               chProcessActionMenu(dgptr);
               return;
             }
          }
        }
        else if(bp->btype == ACTION_SETBREAKPOINTFUNC) {
          if (calldata->funcname != NULL &&
              bp->u.func.called == FALSE && /* first time into this function */
              !strcmp(bp->u.func.funcname, calldata->funcname))
          {
             if(chTestBreakPointCondition(interp, bp) == 0) {
               bp->u.func.called = TRUE;
               chProcessWatchExprs(dgptr);
               chProcessActionMenu(dgptr);
               return;
             }
          }
        }
        else if(bp->btype == ACTION_SETBREAKPOINTVAR)
        {
          varptr = (void*)Ch_SymbolAddrByName(interp, bp->u.var.varname);
          if(varptr) {
             size = Ch_DataSize(interp, bp->u.var.varname);
             if(bp->u.var.varvalue == NULL)
             {
               bp->u.var.varvalue = (void *)malloc(size);
               /* save the value for the variable */
               memcpy(bp->u.var.varvalue, varptr, size);
             }
             else {
               if(memcmp(bp->u.var.varvalue, varptr, size)) {
                  if(chTestBreakPointCondition(interp, bp) == 0) {
                    /* save the value for the variable */
                    memcpy(bp->u.var.varvalue, varptr, size);
                    chProcessWatchExprs(dgptr);
                    chProcessActionMenu(dgptr);
                    return;
                  }
               }
             }
          }
        }
        bp = bp->next;
      }
    }
}
```

Program 7.17: The callback for the debug program (callback.c) (Contd.).

```
/* If the conditional expression for a breakpoint is NULL, or
   the conditional value is, true or has changed,
   this function returns 0. Otherwise, it returns -1 */
int chTestBreakPointCondition(ChInterp_t interp, breakPoint_t *bp) {
    char *expr, *cptr;
    void *exprptr;
    ChType_t datatype;
    size_t size;
    int i, retval = -1;

    expr = bp->condexpr;
    if(expr == NULL)
      return 0;


    if(interp == NULL || Ch_ExprParse(interp, expr) == CH_ERROR)
       return -1;
    datatype = Ch_DataType(interp, expr);
    size = Ch_DataSize(interp, expr);
    exprptr = (void *)malloc(size);
    if(Ch_ExprCalc(interp, expr, datatype, exprptr) == CH_ERROR)
       return -1;

    if(bp->condvalue == NULL) /* first time */
    {
       bp->condvalue = exprptr;
    }
    else {
        if(bp->condtrue) {
          for(i=0, cptr = exprptr; i < size; i++)
          {
            if(*cptr != '\0') {/* not equal 0 */
               retval = 0;
               break;
            }
          }
        }
        else if(memcmp(bp->condvalue, exprptr, size)) {
          memcpy(bp->condvalue, exprptr, size); /* save the value for the expression */
          retval = 0;
        }
        free(exprptr);
    }
    return retval;
}
```

Program 7.17: The callback for the debug program (callback.c) (Contd.).

Appendix A

APIs for Embedding Ch —<embedch.h>

A.1 Information in Header File embedch.h

The header **embedch.h** defines macros and APIs for Embedded Ch.

Data Types

The following data type are defined in the header file **embedch.h**.

Data Type	Description
ChBlock_t	A structure containing the information for a function block. used by **Ch_AddCallback**() and **Ch_ChangeStack**().
ChCallback_t	The data type for callback function used by **Ch_AddCallback**().
ChFile_t	File descriptor I/O redirection in **Ch_Reopen**(), **Ch_Flush**(), and **Ch_Close**().
ChFuncType_t	The returned function type by **Ch_FuncType**().
ChFuncdl_t	A generic pointer to a binary interface function for **Ch_DeclareFunc**().
ChOptions_t	Options for setting embedded Ch as an argument of **Ch_Initialize**().
ChPointer_t	A generic pointer type for **Ch_SetGlobalUserData**() and **Ch_GetGlobalUserData**().
ChUserDefinedTag_t	A generic pointer type for the tag of a user defined struct/class/union type.
ChUserDefinedInfo_t	A structure containing the information for a user defined struct/class/union.
ChMemInfo_t	A structure containing the information for a member of a user defined type.
ChValueNode_t	A generic pointer for an internal value of an expression for **Ch_ExprValue**() and **Ch_DeleteExprValue**().
ChVarType_t	The returned variable type by **Ch_VarType**().

Type **ChOptions_t** is a structure to set the option of Embedded Ch by by **Ch_Initialize**().

```
typedef struct ChOptions_ {
   int shelltype;  /* shell type */
   char *chhome;   /* Embedded Ch home directory */
} ChOptions_t;
```

Type **ChBlock_t** is a structure containing the information for a function block used by **Ch_AddCallback**() and **Ch_ChangeStack**().

```
typedef struct ChBlock_ {
   int event;          /* mask of callback event */
   int count;          /* every 'count' instructions for CH_MASKCOUNT */
   int level;          /* int level2() {level1();} int leve1(){level0();}
                          int leve0() {current func}.  Level 0 is
                          the current running function, whereas level n+1
                          is the function that has called level n. */
   int linecurrent;    /* the current line num where the given function
                          is executing. */
   int linefuncbegin;  /* the line number where the definition of
                          the function begins. */
   int linefuncend;    /* the line number where the definition of the
                          function ends. */
   const char *source; /* the file name if the source is a file, otherwise,
                          it is a string.  It contains the first 70
                          characters beginning with  "@string: " */
   const char *funcname; /* function name if the block is a function,
                          otherwise NULL */
   const char *classname; /* class name if it is a member function,
                          otherwise NULL */
   int isconstructor;  /* true if it is a constructor of class,
                          otherwise false */
   int isdestructor;   /* true if it is a destructor of class,
                          otherwise false */
}ChBlock_t;
```

Type **ChUserDefinedInfo_t** is a structure used to obtain the information for a user defined structure, class, or union by **Ch_UserDefinedInfo**().

```
typedef struct ChUserDefinedInfo_ {
    ChType_t dtype; /* user defined data type: CH_STRUCTTYPE, CH_CLASSTYPE,
                       CH_UNIONTYPE, or CH_UNDEFINETYPE */
    char *tagname;  /* tag name */
    int size;       /* size of class/struct/union */
    int totnum;     /* total number of members in class/struct/union */
}ChUserDefinedInfo_t;
```

Type **ChMemInfo_t** is a structure used to obtain the information for a member of a user defined structure, class, or union by **Ch_UserDefinedMemInfoByIndex**() and **Ch_UserDefinedMemInfoByName**().

```
typedef struct ChMemInfo_ {
    int index;        /* index number of the member */
    char *memname;    /* member name */
    int offset;       /* offset of the member from the starting memory */
    ChType_t dtype;   /* data type of the member */
    int ispublic;     /* 1: public;         0: private */
    int isfunc;       /* 1: function type;  0: not function type */
    int ismemberfunc; /* 1: member funct;   0: not member funct */
    int isconstructor;/* 1: constructor;    0: not constructor */
```

```
    int isdestructor; /* 1: destructor;     0: not destructor */
    int isvararg;     /* 1: funct with variable num arguments; 0: not */
    int arraytype;    /* one of array types */
    int dim;          /* array dim */
    int extent[7];    /* extent for each dimension, up to 7 */
    int isbitfield;   /* 1: bit field;      0: not bit field */
    int fieldsize;    /* struct tag{int i:3, j:10}; fieldsize of j is 10*/
    int fieldoffset;  /* struct tag{int i:3, j:10}; fieldoffset of j is 3*/
    ChUserDefinedTag_t udtag; /* tag for a member of struct/class/union */
} ChMemInfo_t;
```

Functions

The following functions are declared in the header file **embedch.h**.

Function	Description
Ch_Abort	Abort the execution of a embedded Ch script.
Ch_AddCallback	Add events and callback function.
Ch_AppendParseScript	Append and parse a fragment of a Ch code in C space.
Ch_AppendParseScriptFile	Append and parse a Ch program file in C space.
Ch_AppendRunScript	Append and Run a fragment of a Ch code in C space.
Ch_AppendRunScriptFile	Append and Run a Ch program file in C space.
Ch_ArrayDim	Obtain the dimension of an array.
Ch_ArrayExtent	Obtain the number of elements in the specified dimension of an array.
Ch_ArrayNum	Obtain the number of elements of an array.
Ch_ArrayType	Determine if the variable is an array.
Ch_ChangeStack	Change the stack (function) of an executed Ch program.
Ch_Close	Close a file descriptor.
Ch_DataSize	Get the size a variable or expression in the Ch space.
Ch_DataType	Get the data type of a variable or expression in the Ch space.
Ch_DeclareFunc	Declare system functions in the Ch space.
Ch_DeclareTypedef	Change a system variable as a type specifier in the Ch space.
Ch_DeclareVar	Declare system variables in the Ch space.
Ch_DeleteExprValue	Delete a value node for an expression in the Ch space.
Ch_End	End the embedded Ch.
Ch_ExecScript	Run a parsed Ch program in C space.
Ch_ExecScriptM	Run a parsed Ch program in C space.
Ch_ExprCalc	Calculate an expression in the Ch space.
Ch_ExprEval	Evaluate an expression in the Ch space.
Ch_ExprParse	Parse an expression in the Ch space.
Ch_ExprValue	Obtain the value of an expression in the Ch space.
Ch_Flush	Flush the contents of the buffer associated with a file descriptor.
Ch_FuncArgArrayDim	Obtain the dimension of an array argument of a function.
Ch_FuncArgArrayExtent	Obtain the number of elements in the specified dimension of an array argument of a function.
Ch_FuncArgArrayNum	Obtain the number of elements of array argument of a function.
Ch_FuncArgArrayType	Determine if an argument of function is an array.

Ch_FuncArgDataType	Obtain the data type of an argument of a function.
Ch_FuncArgFuncArgNum	Obtain the number of arguments of a function.
Ch_FuncArgIsFunc	Determine if an argument of function is function type.
Ch_FuncArgIsFuncVarArg	Determine if an argument of function is a function with a variable number of argument.
Ch_FuncArgNum	Obtain the number of the arguments of a function.
Ch_FuncArgUserDefinedName	Obtain the name of the user defined data type of an argument of function.
Ch_FuncArgUserDefinedSize	Obtain the size of the user defined data of an argument of function.
Ch_FuncType	Determine the function type of a variable.
Ch_GetGlobalUserData	Get the global user data set by **Ch_SetGlobalUserData**().
Ch_GlobalSymbolAddrByIndex	Get the address of a global variable based on its index in the symbol table.
Ch_GlobalSymbolAddrByName	Get the address of a global variable based on its name.
Ch_GlobalSymbolNameByIndex	Get the name of a global variable based on its index in the symbol table.
Ch_GlobalSymbolIndexByName	Obtain the index number of a global variable based on its name.
Ch_GlobalSymbolTotalNum	Obtain the total number of symbols of global variables in the symbol table.
Ch_InitGlobalVar	enable/disable initialization of global variables at parse time.
Ch_Initialize	Initialize embedded Ch for executing a Chprogram.
Ch_IsFunc	Obsolete, use **Ch_FuncType**()
Ch_IsFuncVarArg	Determine if the variable is function with a variable number of arguments.
Ch_IsVariable	Obsolete, use **Ch_VarType**()
Ch_ParseScript	Parse a Ch program in C space.
Ch_Reopen	Open a new file descriptor and associate an old one with the new one.
Ch_RunScript	Run a Ch program in C space.
Ch_RunScriptM	Run a Ch program in C space.
Ch_SetGlobalUserData	Set the global user data to be obtained by **Ch_GetGlobalUserData**().
Ch_SetVar	Set a value in C space to a variable in Ch space.
Ch_StackLevel	Obtain the highest possible level of the stack.
Ch_StackName	Obtain the name in a stack.
Ch_SymbolAddrByIndex	Get the address of a variable based on its index in the symbol table.
Ch_SymbolAddrByName	Get the address of a variable based on its name within its scope.
Ch_SymbolNameByIndex	Get the name of a variable based on its index in the symbol table.
Ch_SymbolIndex	Obsolete, use **Ch_SymbolIndexByName**()
Ch_SymbolIndexByName	Obtain the index num of a variable in the symbol table based on its name.
Ch_SymbolTotalNum	Obtain the total number of symbols for variables in the symbol table.
Ch_UserDefinedInfo	Obtain the information for a user defined struct/class/union type.
Ch_UserDefinedMemInfoByName	Obtain the information for a member of variable of user defined type.
Ch_UserDefinedMemInfoByIndex	Obtain the information for a member of variable of user defined type.
Ch_UserDefinedName	Obtain the name of the user defined data type of a variable. Obsolete, use **Ch_UserDefinedInfo**().
Ch_UserDefinedSize	Obtain the size of the user defined data type of a variable. Obsolete, use **Ch_UserDefinedInfo**().
Ch_UserDefinedTag	Obtain the tag for a variable of user defined struct/class/union type.
Ch_VarType	Determine if a symbol is a global or local variable.

Macros

The following macros are defined in the header file **embedch.h**.

Macro	Description
Macros for member field `shelltype` of structure `ChOptions`	
CH_REGULARCH	Use regular shell.
CH_SAFECH	Use safe shell.
File descriptor for the fourth argument of function **Ch_Reopen**()	
STDIN_FILENO	Standard input.
STDOUT_FILENO	Standard output.
STDERR_FILENO	Standard error output.
Event mask for the second argument of function **Ch_AddCallback**(). The mask specifies on which events the callback will be called.	
Ch_MASKNONE	No callback.
Ch_MASKCALL	Just after Ch calls and enters a new function.
Ch_MASKRET	Just before Ch leaves the function.
Ch_MASKBLOCK	Just after Ch enters a new block.
Ch_MASKEND	Just before Ch leaves the block.
Ch_MASKLINE	When Ch is about to start the execution of a new line of code, or when it jumps back in the code (even to the same lne).
Ch_MASKCOUNT	After Ch executes every `count` instructions.
Ch_MASKABORT	When Ch is aborted by **Ch_Abort**().
One of return values for function **Ch_ChangeStack**()	
CH_INVALIDLEVEL	invalid function level.

The data type **ChFuncType_t** defined inside the header file **embedch.h** has the following values.

Value	Description
CH_NOTFUNCTYPE	not function
CH_FUNCTYPE	regular function
CH_FUNCPROTOTYPE	function prototype without function definition
CH_FUNCPTRTYPE	pointer to function
CH_FUNCMEMBERTYPE	function of a class
CH_FUNCCONSTYPE	constructor of a class
CH_FUNCDESTTYPE	destructor of a class

The data type **ChVarType_t** defined inside the header file **embedch.h** has the following values.

Value	Description
CH_NOTVARTYPE	not a variable.
CH_GLOBALVARTYPE	a global variable.
CH_LOCALVARTYPE	a local variable.

Portability
This header file has no known portability issues.

A.2 Relevant Information in Header File ch.h

The header file **ch.h** is included in the header file **embedch.h**. The detailed information about header file **ch.h** is presented in an appendix in *Ch SDK User's Guide*. In this section, some relevant information in header file **ch.h** is highlighted.

Data Types
The following data type are defined in the header file **ch.h**.

Data Type	Description
ChInfo_t	Information for Embedded Ch obtained by **Ch_Version**().
ChInterp_t	A Ch interperer. The first argument for all Ch SDK APIs.
ChType_t	The internal Ch data type.
ChVaList_t	A Ch variable number of arguments in the C space.

Functions
The following functions, defined inside the header file **ch.h**, are often used for embedding Ch into applications.

Function	Description
Ch_CallFuncByAddr	Call a Ch function by its address from the C address space.
Ch_CallFuncByAddrv	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_CallFuncByNamev	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_GlobalSymbolAddrByName	Get the address of a global variable based on its name.
Ch_Home	Get the home directory in which Embedded Ch is installed
Ch_VaArg	Obtain a value of a function argument in Ch passed to a dynamically loaded library.

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_VarArgsDelete	Delete a Ch style variable argument list and release its memory.
Ch_VaEnd	For a normal return from a function.
Ch_VaStart	Obtain an instance of Ch interpreter and initialize a *handle* for reference to arguments of a function in Ch.
Ch_Version	Obtain the version information for Ch.

Macros

The return value for most Embedded Ch APIs has the following values defined inside the header file **ch.h**.

Value	Description
CH_ABORT	1 when **Ch_Abort**() is called.
CH_ERROR	-1 for failure of a Ch SDK API call.
CH_OK	0 for success of a Ch SDK API call.

Ch_Abort

Synopsis
#include <**embedch.h**>
int Ch_Abort(**ChInterp_t** *interp*);

Purpose
Abort the execution of a embedded Ch program.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

Description
When **Ch_End**(`interp`) is called, the instance of the interpreter `interp` is completly deleted and its related memory is freed. After **Ch_End**(`interp`) is called, no Ch SDK or Embedded Ch SDK API shall be called for that interperter interp because it does not exist any more. **Ch_End**(`interp`) can also not be invoked indirectly through any Ch SDK or Embedded Ch SDK API.

Ch_Abort(`interp`) is equivalent to abort() in C. **Ch_Abort**(`interp`) will abort the execution of the embedded Ch script upon the completion of the statement currently in execution. For example, a Ch function may call a binary C function which contains **Ch_Abort**(`interp`) call to abort the execution of the function. This Ch function can be invoked by APIs such as **Ch_RunScript**() and **Ch_CallFuncByName**().

The resources for the interpreter shall be deleted by function call of **Ch_End**(`interp`) later.

Ch_Abort(`interp`) be called in a thread to abort execution of an embedded Ch script in another thread. The execution of the script can be aborted by itself either in the Ch space using abort() or **Ch_Abort**() in the binary C space in the same thread or a different thread.

Example 1 in Unix
Abort the execution of a Ch script by **Ch_Abort**() from a different thread. The interpreter handle `interp` is a global variable. You may need to add the lib for pthread by the option -lpthread.

chabort_thread1.c —- A C program aborting a Ch script from a different thread

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<embedch.h>

char * code = "int func() { while(1) {printf(\"hello\\n\");} return 0;};";

ChInterp_t interp;
void *thread_function1(void *arg){
   int retval;

   printf("thread_function() called before\n");
   if(Ch_Initialize(&interp, NULL) == CH_ERROR) {
     printf("Error: Ch_Initialize() failed in thread_function()\n");
```

```
   }
   printf("thread_function() called2 end\n");
   if (Ch_AppendRunScript(interp, code) == CH_ERROR) {
     printf("Error: Ch_AppendRunScript() failed in thread_function()\n");
   }
   printf("thread_function() called3 end\n");
   if (Ch_CallFuncByName(interp, "func",  &retval) == CH_ERROR) {
     printf("Error: Ch_CallFuncByName() failed in thread_function()\n");
   }
   printf("thread_function() called4 end\n");
   Ch_End(interp);
   return NULL;
}

void *thread_function2(void *arg){
   sleep(1);
   Ch_Abort(interp);
   return NULL;
}

int main (){
   pthread_t id[2];

   pthread_create(&id[0], NULL, thread_function1, NULL);
   pthread_create(&id[1], NULL, thread_function2, NULL);
   pthread_join(id[0], NULL);
   pthread_join(id[1], NULL);
   return 0;
}
```

Example 2 in Unix

Abort the execution of a Ch script by **Ch_Abort**() from a different thread. The interpreter handle `interp` is a local variable. You may need to add the lib for pthread by the option -lpthread.

chabort_thread2.c —- A C program aborting a Ch script from a different thread

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<embedch.h>

char * code = "int func() { while(1) {printf(\"hello\\n\");} return 0;} func();";

void *thread_function1(void *arg){
   ChInterp_t interp;
   int retval;

   interp = (ChInterp_t)arg;
   printf("thread_function() called2 end\n");
   if (Ch_AppendRunScript(interp, code) == CH_ERROR) {
     printf("Error: Ch_AppendRunScript() failed in thread_function()\n");
   }
   printf("thread_function() called3 end\n");
   if (Ch_CallFuncByName(interp, "func",  &retval) == CH_ERROR) {
     printf("Error: Ch_CallFuncByName() failed in thread_function()\n");
   }
   printf("thread_function() called4 end\n");
   return NULL;
}
```

```
void *thread_function2(void *arg){
   ChInterp_t interp;

   interp = (ChInterp_t)arg;
   sleep(1);
   Ch_Abort(interp); /* about func() called by Ch_AppendRunScript(); */
   sleep(1);
   Ch_Abort(interp); /* about func() called by  Ch_CallFuncByName(); */
   return NULL;
}

int main (){
   pthread_t id[2];
   ChInterp_t interp;

   printf("main() called\n");
   if(Ch_Initialize(&interp, NULL) == CH_ERROR) {
     printf("Error: Ch_Initialize() failed in main()\n");
   }
   pthread_create(&id[0], NULL, thread_function1, interp);
   pthread_create(&id[1], NULL, thread_function2, interp);
   pthread_join(id[0], NULL);
   pthread_join(id[1], NULL);
   Ch_End(interp);
   return 0;
}
```

Example 3 in Windows

Abort the execution of a Ch script by **Ch_Abort**() from a different thread. The interpreter handle `interp` is a local variable. This program *chabort_thread2_win.c* for Windows is similar to *chabort_thread2.c* for Unix. The Windows equivalent version of *chabort_thread1.c* is available in the distribution as *chabort_thread1_win.c*.

chabort_thread2_win.c —- A C program aborting a Ch script from a different thread in Windows

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include<embedch.h>

#define N 2
char * code = "int func() { while(1) {printf(\"hello\\n\");} return 0;} func();";

DWORD WINAPI thread_function1(PVOID arg) {
    int retval;
    ChInterp_t interp;

    interp = (ChInterp_t) arg;
    printf("thread_function() called2 end\n");
    if (Ch_AppendRunScript(interp, code) == CH_ERROR) {
      printf("Error: Ch_AppendRunScript() failed in thread_function()\n");
    }
    printf("thread_function() called3 end\n");
    if (Ch_CallFuncByName(interp, "func",  &retval) == CH_ERROR) {
      printf("Error: Ch_CallFuncByName() failed in thread_function()\n");
    }
    printf("thread_function() called4 end\n");
    return 100;
```

```
}

DWORD WINAPI thread_function2(PVOID arg) {
    ChInterp_t interp;

    interp = (ChInterp_t) arg;
    _sleep(1000);
    Ch_Abort(interp); // about func() called by Ch_AppendRunScript();
    _sleep(1000);
    Ch_Abort(interp); // about func() called by  Ch_CallFuncByName();
    return 200;
}

int main() {
    ChInterp_t interp;
    HANDLE a_thread[N];
    DWORD a_threadId[N];
    DWORD thread_result[N];
    int i, arg[N];

    if(Ch_Initialize(&interp, NULL) == CH_ERROR) {
      printf("Error: Ch_Initialize() failed in main()\n");
    }

    // Create three new threads.
    for(i = 0; i < N; i++)  {
      arg[i] = i+1;
      if(i ==0) {
        a_thread[i] = CreateThread(NULL, 0, thread_function1, (PVOID)interp,
                                   0, &a_threadId[i]);
      }
      else if( i == 1) {
        a_thread[i] = CreateThread(NULL, 0, thread_function2, (PVOID)interp,
                                   0, &a_threadId[i]);
      }
      if (a_thread[i] == NULL) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
      }
    }
    printf("Waiting for threads to finish...\n");
    for(i = 0; i < N; i++)  {
      if (WaitForSingleObject(a_thread[i], INFINITE) != WAIT_OBJECT_0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
      }
    }
    Ch_End(interp);

    // Retrieve the code returned by the thread.
    for(i = 0; i < N; i++)  {
      GetExitCodeThread(a_thread[i], &thread_result[i]);
      printf("Thread joined, it returned %d\n", thread_result[i]);
    }
    exit(EXIT_SUCCESS);
}
```

See Also
Ch_Initialize(), **Ch_End**().

Ch_AddCallback

Synopsis
#include <**ch.h**>
int Ch_AddCallback(**ChInterp_t** *interp*, **int** *event*, **ChCallback_t** *callback*, **ChPointer_t** *clientdata*, **int** *count*);

Purpose
Add events and callback function.

Return Value
If the function is successful, it returns **CH_OK**. Otherwise, it returns **CH_ERROR**.

Parameters
interp A Ch interpreter.

event The events in which the callback function will be called.

clientdata The data passed to the callback function.

callback The callback function which will be called when an event occurs.

count The callback function which will be called after Ch executes every `count` instructions. This argument is meaningful only when the event contains the event mask **Ch_MASKCOUNT**.

Description
Function **Ch_AddCallback**() adds a callback function in the third argument of function **Ch_AddCallback**(). The second argument *event* specifies on which events the callback will be called. The event masks below can be used to specify events when the callback function is called. It can also be used to obtain the event inside a callback function to determine which event triggered the callback function.

Macro	Description
Ch_MASKNONE	No callback.
Ch_MASKCALL	Just after Ch calls and enters a new function.
Ch_MASKRET	Just before Ch leaves the function.
Ch_MASKBLOCK	Just after Ch enters a new block.
Ch_MASKEND	Just before Ch leaves the block.
Ch_MASKLINE	When Ch is about to start the execution of a new line of code, or when it jumps back in the code (even to the same lne).
Ch_MASKCOUNT	After Ch executes every `count` instructions.
Ch_MASKABORT	When Ch is aborted by **Ch_Abort**().

The event mask **Ch_MASKCOUNT**along with the fifth argument `count` of function **Ch_AddCallback**() specifies that after Ch executes every `count` instructions, the callback function will be called. These instructions are internal to a Ch program. They may contain multiple binary operations or even binary C functions.

To avoid the callback function being called twice at the same executed line, a function block will not trigger a block event as shown below for a Ch function.

```
int func()
{                   //CH_MASKCALL  event
     int i;
     ...
     {              //CH_MASKBLOCK event
        ...
     }              //CH_MASKEND   event

    return 0;       //CH_MASKRET   event
}
```

Multiple events are specified using the bit-wise `or` operator '`|`'. For example, events to trigger the callback function just after a Ch program calls and enters a new function, or just before a Ch program leaves a function, can be specified by

```
int event = CH_MASKCALL | CH_MASKRET;
Ch_AddCallback(interp, event, callback, clientdata, count);
```

For the new line event **Ch_MASKLINE**, the non-executable lines for the openning block '`{`' and closing block '`}`' will not trigger the callback function. A line for a declaration statement, except for declaration with initialization or declaration of a variable length array, will also not trigger the callback function. A hidden statement

```
static const char __func__[] = "function_name";
```

for a function block also will not trigger the callback function when the new line event is specified. When the new line event is **Ch_MASKLINE**, how the callback function is triggered is shown as follows.

```
int main()      // this line will not trigger the callback
{               // this line will not trigger the callback
    int x;      // this line will not trigger the callback
    int n = 10; // initialization triggers the callback
    int a[n];   // declaration of VLA triggers the callback
    {           // this line will not trigger the callback
       x = 10;  // assignment statement triggers the callback
    }           // this line will not trigger the callback
    return 0;   // return statement triggers the callback
}               // this line will not trigger the callback
```

To trigger lines including non-executable openning block '`{`' and closing block '`}`' statements, set the mask as follows.

```
int event = CH_MASKLINE | CH_MASKBLOCK | CH_MASKEND;
```

If the Ch script does not contain function main(), the callback will be invoked when it is triggered by the first callback event. If the Ch script contains function main(), the callback will be enabled only when the function main() beginning its execution. Therefore, the included header files by a C program will not invoke callbacks.

When the function atexit() in Ch is executed, the callback is disabled to avoid execution of the cleanup functions in Ch.

A callback function can be disabled by setting the event to **Ch_MASKNONE**.

A callback function of type **ChCallback_t** has the following function prototype.

```
void callback (ChInterp_t interp, ChBlock_t *calldata,
               ChPointer_t clientdata);
```

The first argument `interp` is a Ch interpreter handle. The second argument `calldata` contains the information of the current block in a running Ch program. The last argument `clientdata` points to the data passed in the fourth argument of function **Ch_AddCallback**(). Only one callback function can be specified for an interpreter. A subsequent function call of **Ch_AddCallback**() will overwrite the previously specified events, callback function, client data, and count value. A callback function is invoked during the program execution. Functions such as **Ch_RunScript**() and **Ch_ParseScript**() that will remove the existing Ch program shall not be called inside the callback function. The callback function cannot call **Ch_End**() to terminate itself. For the same reason, functions such as **Ch_ExecScript**() that will execute the existing Ch program at the beginning shall not be called inside the callback function.

Type **ChBlock_t** is a structure containing the information for a block used by **Ch_AddCallback**() and **Ch_ChangeStack**().

```
typedef struct ChBlock_ {
   int event;          /* event for callback */
   int count;          /* every 'count' instructions for CH_MASKCOUNT */
   int level;          /* int level2() {level1();} int leve1(){level0();}
                          int leve0() {current func}. */
   int linecurrent;    /* the current line num where the program
                          is executing. */
   int linefuncbegin;  /* the line number where the definition of
                          the function begins. */
   int linefuncend;    /* the line number where the definition of the
                          function ends. */
   const char *source; /* the file name if the source is a file, otherwise,
                          it is a string.  It contains the first 70
                          characters beginning with  "@string: " */
   const char *funcname; /* function name if the block is a function,
                          otherwise NULL */
   const char *classname; /* class name if it is a member function,
                          otherwise NULL */
   int isconstructor;  /* true if it is a constructor of class,
                          otherwise false */
   int isdestructor;   /* true if it is a destructor of class,
                          otherwise false */
}ChBlock_t;
```

Field `event` gives the event which triggers the callback function being called. To obtain the event which caused the callback function be called can be obtained by the bit-wise `and` operator '&'. For example, the

events just after a Ch program calls and enters a new function, and just before a Ch program leaves a function, can be obtained inside a callback function as follows.

```
void callback (ChInterp_t interp, ChBlock_t *calldata,
               ChPointer_t clientdata)
{
   /* callback at the beginning of a function call */
   if(calldata->event & CH_MASKCALL) {
      printf("event MASKCALL is true\n");
   }
   if(calldata->event & CH_MASKRET) {
      printf("event MASKRET is true\n");
   }
   ...
}
```

Field *count*] gives the number of `count` instructions after Ch executes to trigger the callback function. This field is valid only when the event contains the event mask **Ch_MASKCOUNT**.

Field `level` gives the level of the function being executed. Level 0 is the current running function, whereas level n+1 is the function that has called level n. For example, when function `level0()` is being called, it has level 0. Function `level1()` which calls function `level0()` has level 1. Function `level2()` which calls function `level1()` in turn has level 2.

```
void level2 {
     level1();
}
void level1 {
     level0();
}
void level0 {
     this_function_is_being_executed.
}
```

The stack for the current function can be changed by the API **Ch_ChangeStack**() based on the function level.

Field `linecurrent` contains the current line number where the Ch program is executing. When the stack is changed to the top level by **Ch_ChangeStack**(), the level is 0.

Field `linefuncbegin` contains the line number where the definition of the executed function begins.

Field `linefuncend` contains the line number where the definition of the executed function ends.

Field `source` contains the file name if the source of the executed code is a file. Otherwise, it is a string such as when the code is loaded from the memory by function **Ch_AppendRunScript**(). If the code is from a string, the `source` is a string up to 70 characters starting with `"@string: "`.

Field `funcname` contains the function name if the executed line of the code is within a function. Otherwise, it is NULL.

Field `classname` contains the class tag name if the executed line of code is within a member function of a class. Otherwise, it is NULL.

Field `isconstructor` is 1 if the executed line of code is within a constructor of a class. Otherwise, it is 0.

Field `isdestructor` is 1 if the executed line of code is within a destructor of a class. Otherwise, it is 0.

Example
See Program 7.2.

See Also
Ch_ChangeStack().

Ch_AppendParseScript

Synopsis
#include <**embedch.h**>
int Ch_AppendParseScript(**ChInterp_t** *interp*, **const char** **code*);

Purpose
Append a fragment of the Ch code to an existing Ch program inside the interpreter executed by functions **Ch_RunScript**() or **Ch_ParseScript**() and **Ch_ExecScript**(). The Ch code will be parsed without execution.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

code A fragment of the Ch code to be appended to the existing Ch program inside the interpreter.

Description
When a program is run in Ch, it first will be parsed, then executed. The process is similar to compile and execution using a C compiler. Like **Ch_RunScript**(),

But, **Ch_AppendParseScript**() parses the code only. The code is not executed. Functions **Ch_ExprEval**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), or **Ch_CallFuncByNameVar**() shall be called to execute the code. This implies that only appended functions can be executed. The parsed code by **Ch_AppendParseScript**(). can not be invoked by again later by function by **Ch_AppendRunScript**().
Restrictions

1. The code passed can process the preprocessing directive **#include** in the code. Some preprocessing directives such as **#define** have to be handled separately from other code. For example,

   ```
   Ch_AppendParseScript(interp, "#define A 2");
   ```

   All preprocessing directives can exist in the code passed by function **Ch_RunScript**(), **Ch_ParseScript**(), **Ch_AppendRunScriptFile**(), or **Ch_AppendParseScriptFile**().

2. The code passed cannot contain comments delimitered by '//', such as

   ```
   // this is comment
   ```

If the code passed is larger than 5120 bytes, **Ch_AppendParseScript**() saves it in a temporary file internally first, then calls the API **Ch_AppendParseScriptFile**() to execute the file, and finally removes the tempoary file is removed. Each line in the temporary file shall be less than 5120 characters. Make sure the carriage return character `'\n'` is used to separate lines. Alternatively, if the code is larger than 5120 bytes, it can be broken into multiple segments. The user can then call the function multiple times to load these segments of code as shown below.

```
Ch_AppendParseScript(interp, code1);
Ch_AppendParseScript(interp, code2);
```

This allows the code to run from the memory and is more efficient.
Example
Refer to Program 1.7.

See Also
Ch_AppendRunScript(), **Ch_RunScript**(), **Ch_ParseScript**(), **Ch_ExecScript**().

Ch_AppendParseScriptFile

Synopsis
#include <**embedch.h**>
int Ch_AppendParseScriptFile(**ChInterp_t** *interp*, **const char** **filename*);

Purpose
Append a fragment of the Ch code in a file to an existing Ch program inside the interpreter executed by functions **Ch_RunScript**() or **Ch_ParseScript**() and **Ch_ExecScript**(). The Ch code will be parsed without execution.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

filename File name that contains the Ch code to be appended to the existing Ch program inside the interpreter.

Description
When a program is run in Ch, it first will be parsed, then executed. The process is similar to compile and execution using a C compiler. Like **Ch_RunScript**(),

But, **Ch_AppendParseScriptFile**() parses the code only. The code is not executed. Functions **Ch_ExprEval**(), **Ch_CallFuncByAddr**(), or **Ch_CallFuncByName**() shall be called to execute the code. This implies that only appended functions can be executed.

The restrictions of **Ch_AppendParseScript**() do not apply to **Ch_AppendParseScriptFile**().

See Also
Ch_AppendParseScript(), **Ch_AppendRunScript**(), **Ch_AppendRunScriptFile**(), **Ch_RunScript**(), **Ch_ParseScript**(), **Ch_ExecScript**().

Ch_AppendRunScript

Synopsis
#include <**embedch.h**>
int Ch_AppendRunScript(**ChInterp_t** *interp*, **const char** **code*);

Purpose
Append a fragment of the Ch code to an existing Ch program inside the interpreter executed by functions **Ch_RunScript**() or **Ch_ParseScript**() and **Ch_ExecScript**(). The Ch code will be parsed and executed.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

code A fragment of the Ch code to be appended to the existing Ch program inside the interpreter.

Description
When a program is run in Ch, it first will be parsed, then executed. The process is similar to compile and execution using a C compiler. Like **Ch_RunScript**(), **Ch_AppendRunScript**() performs these two steps together for a program or functions passed through its argument. The function **Ch_AppendRunScript**() executes a fragment of Ch program from C space. The variable *code* contains the fragment of the Ch code.

Restrictions

1. The code passed can process the preprocessing directive **#include** in the code. Some preprocessing directives such as **#define** have to be handled separately from other code. For example,

   ```
   Ch_AppendRunScript(interp, "#define A 2");
   ```

   All preprocessing directives can exist in the code passed by function **Ch_RunScript**(), **Ch_ParseScript**(), **Ch_AppendRunScriptFile**(), or **Ch_AppendParseScriptFile**().

2. The code passed cannot contain comments delimitered by '//', such as

   ```
   // this is comment
   ```

If the code passed is larger than 5120 bytes, **Ch_AppendRunScript**() saves it in a temporary file internally first, then calls the API **Ch_AppendRunScriptFile**() to execute the file, and finally removes the tempoary file is removed. Each line in the temporary file shall be less than 5120 characters. Make sure the carriage return character `'\n'` is used to separate lines. a character Alternatively, if the code is larger than 5120 bytes, it can be broken into multiple segments. The user can then call the function multiple times to load these segments of code as shown below.

```
Ch_AppendRunScript(interp, code1);
Ch_AppendRunScript(interp, code2);
```

This allows the code to run from the memory and is more efficient.
Example
Refer to Program 1.7.

See Also
Ch_AppendParseScript(), **Ch_RunScript**(), **Ch_ParseScript**(), **Ch_ExecScript**().

Ch_AppendRunScriptFile

Synopsis
#include <**embedch.h**>
int Ch_AppendRunScriptFile(**ChInterp_t** *interp*, **const char** **filename*);

Purpose
Append a fragment of the Ch code in a file to an existing Ch program inside the interpreter executed by functions **Ch_RunScript**() or **Ch_ParseScript**() and **Ch_ExecScript**(). The Ch code will be parsed and executed.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

filename File name that contains the Ch code to be appended to the existing Ch program inside the interpreter.

Description
When a program is run in Ch, it first will be parsed, then executed. The process is similar to compile and execution using a C compiler. Like **Ch_RunScript**(), **Ch_AppendRunScriptFile**() performs these two steps together for a program or functions passed through its argument. The function **Ch_AppendRunScriptFile**() executes a fragment of Ch program from C space. The variable *filename* contains the file name of the Ch code.

The restrictions of **Ch_AppendRunScript**() do not apply to **Ch_AppendRunScriptFile**().

See Also
Ch_AppendParseScript(), **Ch_AppendParseScriptFile**(), **Ch_AppendRunScript**(), **Ch_RunScript**(), **Ch_ParseScript**(), **Ch_ExecScript**().

Ch_ArrayDim

Synopsis
#include <**ch.h**>
int Ch_ArrayDim(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Obtain the dimension of a local or global variable of array type in the Ch space if it is array type.

Return Value
If the local or global variable is of array type, This function returns the dimension of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

name Name of the variable or expression in the Ch space.

Description
This function returns the dimension of a local or global variable in the Ch space. The extent of each dimension of the array can be obtained by function **Ch_ArrayExtent**().

Example
See **Ch_DataType**().

See Also
Ch_DataType(), **Ch_ArrayExtent**(), **Ch_ArrayNum**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Ch_ArrayExtent

Synopsis
#include <**ch.h**>
int Ch_ArrayExtent(**ChInterp_t** *interp*, **const char** **name*, **int** *dim*);

Purpose
Obtain the number of elements in the specified dimension of a local or global variable of array type in the Ch space.

Return Value
If the local or global variable is of array type, This function returns the number of elements in a specified dimension of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

name Name of the variable or expression in the Ch space.

dim An integer specifying for which dimension the number of elements will be obtained, 0 for the first dimension.

Description
This function returns the number of elements in the specified dimension of a local or global variable of array type in the Ch space. For array `a[n][m]`, If `0` is passed to this function as the third argument, the extent of the first dimension, i.e. `n`, will be returned. Otherwise, if `1` is passed to, `m` will be returned.

Example
See **Ch_DataType**().

See Also
Ch_DataType(), Ch(), **Ch_ArrayDim**(), **Ch_ArrayNum**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Ch_ArrayNum

Synopsis
#include <**ch.h**>
int Ch_ArrayNum(ChInterp_t *interp*, **const char** **name*);

Purpose
Obtain the number of elements of a local or global variable of array type in the Ch space if it is array type.

Return Value
If the local or global variable is of array type, This function returns the number of elements of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

name Name of the variable or expression in the Ch space.

Description
This function returns the number of elements of a local or global variable in the Ch space. The dimension of the array can be obtained by function **Ch_ArrayDim**().

Example
See **Ch_DataType**().

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Ch_ArrayType

Synopsis
#include <**ch.h**>
ChType_t Ch_ArrayType(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Determine if a local or global variable in the Ch space is an array and its array type.

Return Value
Based on its argument, this function returns one of the macros below, defined in header file **ch.h**.

Macro	Description	Example
CH_UNDEFINETYPE	not an array.	`int i`
CH_CARRAYTYPE	C array	`int a[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[&])`

The macro **CH_UNDEFINETYPE**has the value 0. All others are non-zero values.

Parameters
interp A Ch interpreter.

name Name of the variable or expression in the Ch space.

Description
The function **Ch_ArrayType**() determines if the argument is array type. Unlike function **Ch_FuncArgArrayType**(), a poiner to C array is not considered as an array by function **Ch_ArrayType**() for the convenience of runtime manipulation of variables. For example, for the script code

```
int (*p)[2][3];
```

The array type of `p` using function **Ch_DataType**() is **CH_UNDEFINETYPE**. The data type of an array shall be determined by function **Ch_DataType**().

Example
See **Ch_DataType**().

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayNum**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Ch_ChangeStack

Synopsis
#include <**ch.h**>
int Ch_ChangeStack(**ChInterp_t** *interp*, **int** *level*, **ChBlock_t** **block*);

Purpose
Change the stack for the current function of an executed Ch program.

Return Value
If the function is successful, it returns **CH_OK**. If the specified stack `level` is invalid, it returns **CH_INVALIDLEVEL**. Otherwise, it returns **CH_ERROR**.

Parameters
interp A Ch interpreter.

level The level of the function being executed.

block The block information for the specified level of function.

Description
Function **Ch_ChangeStack**() changes the interpreter runtime function stack. The second argument `level` specifies the level of the function being executed. Level 0 is the currently executed function, whereas level n+1 is the function that has called level n. For example, when function `level0()` is being called, it has level 0. Function `level1()` which calls function `level0()` has level 1. Function `level2()` which calls function `level1()` in turn has level 2.

```
void level2 {
     int i2 = 20;
     level1();
}
void level1 {
     int i1 =10;
     level0();
}
void level0 {
     i0 = 1;
     this_part_of_the_function_level0_is_being_executed.
}
```

As an example, if a callback function is invoked when the code inside function `level0()` in the above code is executed. Other API functions can be used to access and change the value of the variable `i0` inside function `level0()`. To access the value for the variable `i1` inside the calling function `level1()`. The stack of the current function needs to be changed as follows.

```
ChBlock_t block;
level = 1;
ChChangeStack(interp, level, &block);
```

The information for the current function `level1()` after changing the stack is passed back to the third argument `block` of **ChBlock_t** type which is a structure containing the information for a block. It is described in detail in the reference for the API **Ch_AddCallback**(). In the above example, after the function call, the value of `block.level` for the member `level` of `block` is 1. The member `linecurrent` of `block` is 0 at the top level.

The stack will be changed to the level 0 internally when a callback function specified by **Ch_AddCallback**() returns. However, if the stack is changed not inside a callback function, it needs to be reset to level 0 to resume the execution of the original running program as shown below.

```
ChBlock_t block;
ChChangeStack(interp, 0, &block);
```

Example
See Program 7.3.

See Also
Ch_AddCallback(), **Ch_StackLevel**(), **Ch_StackName**().

Ch_Close

Synopsis
#include <**embedch.h**>
int Ch_Close(**ChInterp_t** *interp*, **ChFile_t** *filedes*);

Purpose
Close a file descriptor.

Return Value
Upon successful completion, **Ch_Close**() returns 0. Otherwise, it returns EOF.

Parameters
interp A Ch interpreter.

fildes The file descriptor for the stream to be closed.

Description
The **Ch_Close**() function calls **fclose()** to flush the stream and close the file associated with the file descriptor `filedes` returned from function **Ch_Reopen**().

Example
Refer to Program 1.12.

See Also
Ch_Reopen(), **Ch_Flush**().

Ch_DataSize

Synopsis
#include <**ch.h**>
size_t **Ch_DataSize**(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Get the size of a local or global variable, or an expression in a Ch program.

Return Value
Similar to the `sizeof` operator in C, this function returns the size of of a local or global variable, or an expression in a Ch program in the number of bytes.

Parameters
interp A Ch interpreter.

name Name of a variable or expression whose data type is to be obtained.

Description
The function **Ch_DataSize**() gets the size of a variable or expression in the Ch space within the executed program scope. If a passed argument is of array type, the function will return the number of bytes for the array.

Example
See **Ch_DataType**().

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Ch_DataType

Synopsis
#include <**ch.h**>
ChType_t Ch_DataType(**ChInterp_t** *interp,* **const char** **name*);

Purpose
Get the data type of a local or global variable named *name* in a Ch program.

Return Value
This function returns a macro for data type **ChType_t** defined in header file **ch.h** described in *Ch SDK User's Guide.* If a passed argument is of array type, the function will return the type of the element of the array. For example, **CH_INTTYPE** and **CH_DOUBLETYPE** are two macros defined for data type of int and double in Ch.

For a variable of C array or Ch computational array, it gives the data type of its element.

Parameters
interp A Ch interpreter.

name Name of the variable or expression whose data type is to be obtained.

Description
The function **Ch_DataType**() gets the data type of a variable in Ch space within the executed program scope. If a variable of array type including pointer to computational array, it gives the data type of its element. Unlike function **Ch_FuncArgDataType**(), For a pointer to C array, it gives the pointer type of its element for the convenience of runtime manipulation of variables. For example, for the script code

```
int a[3], (*p)[2][3];
array int b[3], (*c)[2][3];
```

The data type of `a, b, c` using function **Ch_DataType**() is **CH_INTTYPE**. The data type for `p` is **CH_INTPTRTYPE**.

For a variable of boolean type, it is treated as a char in Ch. For example, for the script code

```
#include <stdbool.h>
bool bo = true;
```

The data type of `bo` using function **Ch_DataType**() is **CH_CHARTYPE**.

The argument of *name* is a variable name in Ch space.

Example
In this example, information about global variables in the embedded Ch program `varinfo.ch` are obtained in the hosting C program `varinfo.c`.

varinfo.c —- A C program to get information for variables in Ch.

```
/*************************************************************
* File Name: varinfo.c
*************************************************************/
#include <stdio.h>
```

```
#include <limits.h>
#include <embedch.h>

struct tag {
  int i;
  double f;
};

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"varinfo.ch", NULL};
  int *ap, *bp;
  struct tag *sp, **sp2;
  ChUserDefinedTag_t udtag;
  ChUserDefinedInfo_t udinfo;
  ChMemInfo_t meminfo;

  Ch_Initialize(&interp, NULL);
  status = Ch_RunScript(interp,argvv);
  if(status == CH_ERROR) {
    printf("Error: execution of program varinfo.ch failed\n");
  }
  if(Ch_DataType(interp, "n") == CH_INTTYPE) {
    printf("n is an int type\n");
  }
  printf("Ch_DataSize(interp, \"n\") = %d\n", Ch_DataSize(interp, "n"));
  if(Ch_DataType(interp, "d") == CH_DOUBLETYPE) {
    printf("d is double type\n");
  }

  if(Ch_DataType(interp, "a") == CH_INTTYPE) {
    printf("a is a value of int \n");
  }
  if(Ch_ArrayType(interp, "a") == CH_CARRAYTYPE) {
    printf("a is a C array\n");
    printf("Array dim of a is %d\n", Ch_ArrayDim(interp, "a"));
    printf("Array elements of a is %d\n", Ch_ArrayNum(interp, "a"));
    printf("Extent of the 1st dim of a = %d\n", Ch_ArrayExtent(interp, "a", 0));
    ap =  (int *)Ch_SymbolAddrByName(interp, "a");
    printf("*ap = %d\n", *ap);
  }
  printf("Ch_DataSize(interp, \"a\") = %d\n", Ch_DataSize(interp, "a"));
  if(Ch_DataType(interp, "a2") == CH_INTTYPE) {
    printf("a2 is a value of int \n");
  }
  if(Ch_ArrayType(interp, "a2") == CH_CARRAYVLATYPE) {
    printf("a2 is a C VLA array\n");
    printf("Array dim of a2 is %d\n", Ch_ArrayDim(interp, "a2"));
    printf("Array elements of a2 is %d\n", Ch_ArrayNum(interp, "a2"));
    printf("Extent of the 1st dim of a2 = %d\n", Ch_ArrayExtent(interp, "a2", 0));
  }
  printf("Ch_DataSize(interp, \"a2\") = %d\n", Ch_DataSize(interp, "a2"));
  if(Ch_DataType(interp, "pa") == CH_INTPTRTYPE) {
    if(Ch_ArrayType(interp, "pa") == CH_UNDEFINETYPE)
       printf("pa is not an array\n");
    printf("pa is a pointer to int\n");
    ap =  *(int **)Ch_SymbolAddrByName(interp, "pa");
    printf("ap = %p\n", ap);
```

```
}

if(Ch_DataType(interp, "func") == CH_FLOATTYPE) {
  printf("func() return type is float\n");
}
if(Ch_DataType(interp, "func()+3.0") == CH_DOUBLETYPE) {
  printf("func()+3.0 is double type\n");
}
/* or if(Ch_FuncType(interp, "func") == CH_FUNCTYPE){ */
if(Ch_FuncType(interp, "func")) {
  printf("func() is a function\n");
  printf("Number of arguments of function func() is %d\n",
         Ch_FuncArgNum(interp, "func"));
}
if(Ch_FuncType(interp, "funcv") == CH_FUNCPROTOTYPE){
  printf("funcv() is a function prototype\n");
  printf("Number of arguments of function funcv() is %d\n",
         Ch_FuncArgNum(interp, "funcv"));
}
if(Ch_IsFuncVarArg(interp, "funcv")){
  printf("funcv() is a function of variable argument number\n");
}
if(Ch_FuncType(interp, "funcvptr") == CH_FUNCPTRTYPE){
  printf("funcvptr() is a pointer to function\n");
  printf("Number of arguments of function funcvptr() is %d\n",
           Ch_FuncArgNum(interp, "funcvptr"));
}
if(Ch_IsFuncVarArg(interp, "funcvptr")){
  printf("funcvptr() is a function of variable argument number\n");
}

if(Ch_DataType(interp, "s") == CH_STRUCTTYPE) {
  udtag = Ch_UserDefinedTag(interp, "s");
  Ch_UserDefinedInfo(interp, udtag, &udinfo);
  printf("udinfo.dtype == CH_STRUCTTYPE = %d\n",
          udinfo.dtype == CH_STRUCTTYPE);
  printf("struct size of s is %d\n", udinfo.size);
  printf("struct name of s is %s\n", udinfo.tagname);
  printf("the number of members of s is %d\n", udinfo.totnum);
  Ch_UserDefinedMemInfoByName(interp, udtag, "f", &meminfo);
  printf("meminfo.index = %d\n", meminfo.index);
  printf("meminfo.memname = %s\n", meminfo.memname);
  printf("meminfo.offset = %d\n", meminfo.offset);
  printf("meminfo.dtype == CH_DOUBLETYPE = %d\n",
          meminfo.dtype == CH_DOUBLETYPE);
  printf("meminfo.ispublic = %d\n", meminfo.ispublic);
  printf("meminfo.isfunc = %d\n", meminfo.isfunc);
  sp = Ch_SymbolAddrByName(interp, "s");
  printf("sp->i = %d\n", sp->i);
  printf("sp->f = %f\n", sp->f);
}
printf("Ch_DataSize(interp, \"s\") = %d\n", Ch_DataSize(interp, "s"));
if(Ch_DataType(interp, "sp") == CH_STRUCTPTRTYPE) {
  udtag = Ch_UserDefinedTag(interp, "sp");
  Ch_UserDefinedInfo(interp, udtag, &udinfo);
  printf("struct size of sp is %d\n", udinfo.size);
  printf("struct name of sp is %s\n", udinfo.tagname);
  sp2 = Ch_SymbolAddrByName(interp, "sp");
  printf("*sp2->i = %d\n", (*sp2)->i);
```

```
  printf("*sp2->f = %f\n", (*sp2)->f);
}
if(Ch_DataType(interp, "c") == CH_CLASSTYPE) {
  udtag = Ch_UserDefinedTag(interp, "c");
  Ch_UserDefinedInfo(interp, udtag, &udinfo);
  printf("udinfo.dtype == CH_CLASSTYPE = %d\n",
          udinfo.dtype == CH_CLASSTYPE);
  printf("class size of c is %d\n", udinfo.size);
  printf("class name of c is %s\n", udinfo.tagname);
}
if(Ch_DataType(interp, "u") == CH_UNIONTYPE) {
  udtag = Ch_UserDefinedTag(interp, "u");
  Ch_UserDefinedInfo(interp, udtag, &udinfo);
  printf("union size of u is %d\n", udinfo.size);
  printf("union name of u is %s\n", udinfo.tagname);
}

if(Ch_DataType(interp, "sa") == CH_STRUCTTYPE) {
  printf("sa is struct type\n");
  udtag = Ch_UserDefinedTag(interp, "sa");
  Ch_UserDefinedInfo(interp, udtag, &udinfo);
  printf("struct size of sa is %d\n", udinfo.size);
  printf("struct name of sa is %s\n", udinfo.tagname);
  if(Ch_ArrayType(interp, "sa"))
    printf("sa is an array\n");
  sp =  (struct tag*)Ch_SymbolAddrByName(interp, "sa");
  printf("sp->i = %d\n", sp->i);
  printf("sp->f = %f\n", sp->f);
  sp++;
  printf("sp->i = %d\n", sp->i);
  printf("sp->f = %f\n", sp->f);
}

if(Ch_DataType(interp, "b") == CH_INTTYPE) {
  printf("b is a value of int \n");
}
if(Ch_ArrayType(interp, "b") == CH_CHARRAYTYPE) {
  printf("b is a Ch computational array\n");
  printf("Array dim of b is %d\n", Ch_ArrayDim(interp, "b"));
  printf("Array elements of b is %d\n", Ch_ArrayNum(interp, "b"));
  printf("Extent of the 1st dim of b = %d\n", Ch_ArrayExtent(interp, "b", 0));
  printf("Extent of the 2nd dim of b = %d\n", Ch_ArrayExtent(interp, "b", 1));
  bp =  (int *)Ch_SymbolAddrByName(interp, "b");
  printf("*bp = %d\n", *bp);
}
if(Ch_DataType(interp, "b2") == CH_INTTYPE) {
  printf("bp is a value of int \n");
}
if(Ch_ArrayType(interp, "b2") == CH_CHARRAYVLATYPE) {
  printf("b2 is a Ch VLA array\n");
  printf("Array dim of b2 is %d\n", Ch_ArrayDim(interp, "b2"));
  printf("Array elements of b2 is %d\n", Ch_ArrayNum(interp, "b2"));
  printf("Extent of the 1st dim of b2 = %d\n", Ch_ArrayExtent(interp, "b2", 0));
  printf("Extent of the 2nd dim of b2 = %d\n", Ch_ArrayExtent(interp, "b2", 1));
}
if(Ch_DataType(interp, "arrayfunc") == CH_DOUBLETYPE) {
  printf("arrayfunc() return type is double\n");
}
if(Ch_ArrayType(interp, "arrayfunc") == CH_CHARRAYTYPE){
```

```
    printf("arrayfunc() is Ch array\n");
    printf("Array dim of arrayfunc is %d\n", Ch_ArrayDim(interp, "arrayfunc"));
    printf("Array elements of arrayfunc is %d\n", Ch_ArrayNum(interp, "arrayfunc"));
    printf("Extent of the 1st dim of arrayfunc = %d\n",
            Ch_ArrayExtent(interp, "arrayfunc", 0));
  }
  if(Ch_FuncType(interp, "arrayfunc")){
    printf("arrayfunc() is a function\n");
    printf("Number of arguments of function arrayfunc() is %d\n",
           Ch_FuncArgNum(interp, "arrayfunc"));
  }
  if(Ch_DataType(interp, "arrayfunc2") == CH_DOUBLETYPE) {
    printf("arrayfunc2() return type is double\n");
  }
  if(Ch_ArrayType(interp, "arrayfunc2") == CH_CHARRAYVLATYPE){
    printf("arrayfunc2() is Ch VLA array\n");
    printf("Array dim of arrayfunc2 is %d\n", Ch_ArrayDim(interp, "arrayfunc2"));
    if(Ch_ArrayExtent(interp, "arrayfunc2", 0) == INT_MAX);
      printf("Extent of the 1st dim of arrayfunc2 is determined at runtime\n");
    if(Ch_ArrayExtent(interp, "arrayfunc2", 1) == INT_MAX);
      printf("Extent of the 2nd dim of arrayfunc2 is determined at runtime\n");
  }
  if(Ch_FuncType(interp, "arrayfunc2")){
    printf("arrayfunc2() is a function\n");
    printf("Number of arguments of function arrayfunc2() is %d\n",
           Ch_FuncArgNum(interp, "arrayfunc2"));
  }

  Ch_End(interp);
}
```

varinfo.ch —- A Ch program.

```
#include <array.h>

int n =10;
double d = 1.1;
int a[2]={10,20}, a2[n], (*pa)[3];
struct tag {
  int i;
  double f;
};
class tagc {
  public:
    int i;
    double f;
};
union tagu {
    int i;
    double f;
};
struct tag s ={100, 200}, *sp,
           sa[2]={10, 20, 30, 40};
class tagc c;
union tagu u;

float func() {
  return 0.0;
}
float funcv(int i, ...);
```

```
float (*funcvptr)(int i, ...);
array int b[2][3]={1,2,3,4,5,6}, b2[n][2*n];
array double arrayfunc(void)[3] {
  array double a[3];
  return a;
}
array double arrayfunc2(int i, int j)[:][:];

int main () {
   sp = &s;
}
```

Output

```
n is an int type
Ch_DataSize(interp, "n") = 4
d is double type
a is a value of int
a is a C array
Array dim of a is 1
Array elements of a is 2
Extent of the 1st dim of a = 2
*ap = 10
Ch_DataSize(interp, "a") = 8
a2 is a value of int
a2 is a C VLA array
Array dim of a2 is 1
Array elements of a2 is 10
Extent of the 1st dim of a2 = 10
Ch_DataSize(interp, "a2") = 40
pa is not an array
pa is a pointer to int
ap = 0
func() return type is float
func()+3.0 is double type
func() is a function
Number of arguments of function func() is 0
funcv() is a function prototype
Number of arguments of function funcv() is 1
funcv() is a function of variable argument number
funcvptr() is a pointer to function
Number of arguments of function funcvptr() is 1
funcvptr() is a function of variable argument number
udinfo.dtype == CH_STRUCTTYPE = 1
struct size of s is 16
struct name of s is tag
the number of members of s is 2
meminfo.index = 1
meminfo.memname = f
meminfo.offset = 8
meminfo.dtype == CH_DOUBLETYPE = 1
meminfo.ispublic = 1
meminfo.isfunc = 0
sp->i = 100
sp->f = 200.000000
Ch_DataSize(interp, "s") = 16
struct size of sp is 16
struct name of sp is tag
*sp2->i = 100
*sp2->f = 200.000000
```

```
udinfo.dtype == CH_CLASSTYPE = 1
class size of c is 16
class name of c is tagc
union size of u is 8
union name of u is tagu
sa is struct type
struct size of sa is 16
struct name of sa is tag
sa is an array
sp->i = 10
sp->f = 20.000000
sp->i = 30
sp->f = 40.000000
b is a value of int
b is a Ch computational array
Array dim of b is 2
Array elements of b is 6
Extent of the 1st dim of b = 2
Extent of the 2nd dim of b = 3
*bp = 1
bp is a value of int
b2 is a Ch VLA array
Array dim of b2 is 2
Array elements of b2 is 200
Extent of the 1st dim of b2 = 10
Extent of the 2nd dim of b2 = 20
arrayfunc() return type is double
arrayfunc() is Ch array
Array dim of arrayfunc is 1
Array elements of arrayfunc is 3
Extent of the 1st dim of arrayfunc = 3
arrayfunc() is a function
Number of arguments of function arrayfunc() is 0
arrayfunc2() return type is double
arrayfunc2() is Ch VLA array
Array dim of arrayfunc2 is 2
Extent of the 1st dim of arrayfunc2 is determined at runtime
Extent of the 2nd dim of arrayfunc2 is determined at runtime
arrayfunc2() is a function
Number of arguments of function arrayfunc2() is 2
```

See Also

Ch_DataSize(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Ch_DeclareFunc

Synopsis
#include <**embedch.h**>
int Ch_DeclareFunc(**ChInterp_t** *interp*, **const char** **funcprototype*, **ChFuncdl_t** *funcptr*);

Purpose
Declare and define a system function in the Ch space corresponding to a binary function in the C/C+ space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

funcprototype A function prototype for the defined function.

funcptr A pointer to a wrapper function to interface the binary function.

Description
Function **Ch_DeclareFunc**() can be used to call a binary function in the C space from the Ch space in an Embedded Ch engine. Unlike using Ch SDK, no extra function files with file extension .chf and dynamically loaded libraries with file extension .dl are needed. In addition to global variables, a Ch environment contains system variables, such as _path and _fpath. The system variables are persistent. When a new program is loaded, these system variables remain in the system. Similar to **Ch_DeclareVar**(), functions defined by **Ch_DeclareFunc**() are system functions in the Ch space. as if the function prototype `funcptototype` is pre-appended with the type qualifier `__declspec(global)`. For example, the declaration

```
Ch_DeclareFunc(interp, "double func(int i);", funcptr);
```

is treated as if the the function protpotype was declared as

```
__declspec(global) double func(int i);
```

A new data type for return value and arguments of a system function can be created using the API **Ch_DeclareTypedef**().

Both global functions and system functions can be removed by a pragma statement as shown below

```
#pragma remvar(func)
```

in the Ch space, which can accomplished using Embedded Ch by

```
Ch_AppendParseScript(interp, "#pragma remvar(func)");
```

The same identifier cannot be declared as a variable for both global function and system function with function definitions for both.

A struct type used as a function argument or return type of a system function defined by function **Ch_DeclareFunc**() can be declared by function **Ch_DeclareVar**(). To call binary functions from an application with Embedded Ch. The application typically defines binary functions as system functions in the Ch space using functions **Ch_DeclareVar**(). and **Ch_DeclareFunc**(). Then, run Ch programs by

functions **Ch_ParseScript**(), **Ch_RunScript**(), **Ch_RunScriptM**(), as well as **Ch_AppendParseScript**(), **Ch_AppendParseScriptFile**(), **Ch_AppendRunScript**(), **Ch_AppendRunScriptFile**(). When multiple Ch programs are executed by function **Ch_RunScriptM**() in a single instance of Embedded Ch, the same system functions defined by **Ch_DeclareFunc**() will be used.

Like a global function, a system function prototype can be declared multiple times. An identifier can be declared as both global and system variable. In this case, the identifier will be treated as a global variable inside a program. For a variable of function type, only one function definition can be defined if the identifier is used for both global and system variable.

The argument `funcptr` is a wrapper function to interface the binary function. The wraper function is the same as that described in Ch SDK User's Guide. Wrapper functions can be generated automatically from a header file using the utility program **c2chf** in Ch SDK.

Two Special Cases

Functions returning a struct type and functions with a variable number of arguments are two special cases. One have to pay attention to the two special cases when calling binary function from a Ch program in the Embedded Ch.

As shown in function `func4()` in the example below, for a function returning a struct type, the return type for the wrapper function is `void`. The returned structure is passed as an extra first argument of pointer to struct.

For a function with a variable number of arguments, the list of arguments in the Ch space is changed to the list of arguments in the C space by the function **Ch_VaVarArgsCreate**() and the used temporary memory is cleaned by the function **Ch_VaVarArgsDelete**() as shown in function `func5()` the example below.

Example 1

Use binary functions in the C space as system functions in the Ch space.

```
/* File Name: declarefunc.c */

#include <stdio.h>
#include <embedch.h>

struct tag {int i, *p;};

/* function of void type */
void func1(double x) {
    printf("x in func1() is %f\n", x);
}

/* function with a return value */
double func2(double x, double y) {
    printf("x in func2() is %f\n", x);
    printf("y in func2() is %f\n", y);
    return x+y;
}

/* function with arguments of struct and pointer to struct,
   returning a pointer to struct */
struct tag *func3(struct tag *sp, struct tag s) {
    printf("sp->i in func3() is %d\n", sp->i);
    printf("s.i in func3() is %d\n", s.i);
    sp->i *= s.i;
    return sp;
}
```

```
/* SPECIAL CASE for function returning a struct */
/* similar to the method described in Ch SDK */
struct tag func4(int i, struct tag s) {
    struct tag s2;
    s2 = s;
    printf("s.i in func4() is %d\n", s.i);
    s2.i *= i;
    return s2;
}

/* SPECIAL CASE for function with variable number of arguments */
int vfunc5(int n, va_list ap) {
    int i;
    double d;

    i = va_arg(ap, int);
    d = va_arg(ap, double);
    printf("i in vfunc5() = %d\n", i);
    printf("d in vfunc5() = %f\n", d);
    return  2*n;
}

int func5(int n, ...) {
    va_list ap;
    int retval;

    va_start(ap, n);
    retval = vfunc5(n, ap);
    va_end(ap);
    return retval;
}

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

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

EXPORTCH double func2_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    double x, 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;
}

EXPORTCH struct tag *func3_chdl(void *varg) {
```

```
    ChInterp_t interp;
    ChVaList_t ap;
    struct tag *sp, s;

    Ch_VaStart(interp, ap, varg);
    sp = Ch_VaArg(interp, ap, struct tag*);
    s = Ch_VaArg(interp, ap, struct tag);
    sp = func3(sp, s);
    Ch_VaEnd(interp, ap);
    return sp;
}

/* SPECIAL CASE for function returning a struct */
EXPORTCH void func4_chdl(void *varg) { /* return type void */
    ChInterp_t interp;
    ChVaList_t ap;
    struct tag *retval; /* return value  */
    int i;
    struct tag  s;

    Ch_VaStart(interp, ap, varg);
    retval = Ch_VaArg(interp, ap, struct tag*);
    i = Ch_VaArg(interp, ap, int);
    s = Ch_VaArg(interp, ap, struct tag);
    *retval = func4(i, s); /* return value */
    Ch_VaEnd(interp, ap);
}

/* SPECIAL CASE for function with variable number of arguments */
EXPORTCH int func5_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    void *ap_c;  /* equivalent to va_list ap_c */
    ChVaList_t ap_ch;
    int n;
    void *memhandle;
    int retval;

    Ch_VaStart(interp, ap, varg);
    n = Ch_VaArg(interp, ap, int);
    ap_ch = Ch_VaArg(interp, ap, ChVaList_t);
    /* get variable argument list in C */
    ap_c = Ch_VaVarArgsCreate(interp, ap_ch, &memhandle);
    retval = vfunc5(n, ap_c);
    /* delete the temp memory */
    Ch_VaVarArgsDelete(interp, memhandle);
    Ch_VaEnd(interp, ap);
    return retval;
}

int main() {
    ChInterp_t interp;
    int status;
    char str[1024];

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);

    Ch_DeclareVar(interp,"int ch_i = 10, *ch_p;");
```

```
    Ch_DeclareVar(interp,"struct tag {int i, *p;} s = {10, NULL}, s2, *sp;");
    /* create new data type tagptr_t in the Ch space */
    Ch_DeclareVar(interp,"struct tag *tagptr_t;");
    Ch_DeclareTypedef(interp,"tagptr_t");

    Ch_DeclareFunc(interp,"void func1(double x);", (ChFuncdl_t)func1_chdl);
    Ch_ExprEval(interp, "func1(ch_i)");

    Ch_DeclareFunc(interp,"double func2(double x, double y);", (ChFuncdl_t)func2_chdl);
    Ch_ExprEval(interp, "printf(\"func2(2,3) = %f\n\",func2(2, 3))");

    /* or Ch_DeclareFunc(interp,"struct tag *func3(struct tag *sp, struct tag s);", */
    Ch_DeclareFunc(interp,"tagptr_t func3(tagptr_t sp, struct tag s);",
                   (ChFuncdl_t)func3_chdl);
    Ch_ExprEval(interp, "sp =func3(&s, s)");
    Ch_ExprEval(interp, "printf(\"sp->i = %d\n\",sp->i)");

    /* SPECIAL CASE for function returning a struct */
    Ch_DeclareFunc(interp,"struct tag func4(int n, struct tag s);",
                   (ChFuncdl_t)func4_chdl);
    Ch_ExprEval(interp, "s2 =func4(10, s)");
    Ch_ExprEval(interp, "printf(\"s2.i = %d\n\",s2.i)");

    /* SPECIAL CASE for function with variable number of arguments */
    Ch_DeclareFunc(interp,"int func5(int n, ...);", (ChFuncdl_t)func5_chdl);
    Ch_ExprEval(interp, "func5(3, 10, 12.34)");
    Ch_End(interp);
    return 0;
}
```

Output

```
x in func1() is 10.000000
x in func2() is 2.000000
y in func2() is 3.000000
func2(2,3) = 5.000000
sp->i in func3() is 10
s.i in func3() is 10
sp->i = 100
s.i in func4() is 100
s2.i = 1000
i in vfunc5() = 10
d in vfunc5() = 12.340000
```

Example 2
See Program 5.12.

See Also
Ch_DeclareTypedef(), **Ch_DeclareVar**().

Ch_DeclareTypedef

Synopsis
#include <**embedch.h**>
int Ch_DeclareTypedef(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Change a system variable to a typedefed data type in the Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

name A system variable in the Ch space.

Description
Function **Ch_DeclareVar**() can be used to declare system variables of valid data type in Ch, including simple type, pointer typer, struct and class type, function type, etc. The type qualifier `__declspec(global)` is added at the beginning of the declaration statement of function **Ch_DeclareVar**(). The statement

```
Ch_DeclareVar(interp, "int a, b;");
```

is equivalent to declare system variables `a` and `b` in the Ch space as follows.

```
__declspec(global) int a, b;
```

Because type specifiers `__declspec(global)` and `typedef` cannot be used together to declare a variable. **Ch_DeclareVar**() cannot be used to define a new data type with `typedef` as shown below.

```
Ch_DeclareVar(interp, "typedef int newtype_t;");
```

The new data type for a system variable can be created by the function **Ch_DeclareTypedef**() as shown below.

```
Ch_DeclareVar(interp, "int newtype_t;");
Ch_DeclareTypdef(interp, "newtype_t");
Ch_DeclareVar(interp, "newtype_t sysvar1;");
```

In this case,

```
Ch_DeclareVar(interp, "newtype_t sysvar1;");
```

is equivalent to

```
__declspec(global) int sysvar1;
```

in the Ch space.
Example
See **Ch_DeclareFunc**().
See Also
Ch_DeclareVar(), **Ch_DeclareFunc**(), **Ch_AppendParseScript**(), **Ch_AppendRunScript**(),

Ch_DeclareVar

Synopsis
#include <**embedch.h**>
int Ch_DeclareVar(**ChInterp_t** *interp*, **const char** **declaration*);

Purpose
Declare system variables in the Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

declaration A declaration statement including the initialization in the Ch space.

Description
A C program has global variables. Global variables in a Ch script loaded by **Ch_ParseScript**(), **Ch_RunScript**(), **Ch_RunScriptM**(), as well as **Ch_AppendParseScript**(), **Ch_AppendParseScriptFile**(), **Ch_AppendRunScript**(), **Ch_AppendRunScriptFile**() in an instance of Ch engine will be removed when a new Ch program is loaded. In addition to global variables, a Ch environment contains system variables, such as _path and _fpath. The system variables are persistent. When a new program is loaded, these system variables remain in the system. Functions **Ch_AppendParseScript**() and **Ch_AppendRunScript**() can be used to declare variables of global scope like.

```
Ch_AppendParseScript(interp, "int a, b;");
```

Function **Ch_DeclareVar**() can be used to declare system variables of valid data type in Ch, including simple type, pointer typer, struct and class type, function type, etc. The type qualifier `__declspec(global)` is added at the beginning of the declaration statement of function **Ch_DeclareVar**(). The statement

```
Ch_DeclareVar(interp, "int a, b;");
```

is equivalent to declare system variables `a` and `b` in the Ch space as follows.

```
__declspec(global) int a, b;
```

Because type specifiers `__declspec(global)` and `typedef` cannot be used together to declare a variable, a new data type can be created by using both **Ch_DeclareVar**() and **Ch_DeclareTypedef**() together as shown below.

```
Ch_DeclareVar(interp, "int newtype_t;");
Ch_DeclareTypedef(interp, "newtype_t");
```

Both global variables and system variables can be removed by a pragma statement as shown below

```
#pragma remvar(a)
#pragma remvar(b)
```

in the Ch space, which can accomplished using Embedded Ch by

```
Ch_AppendParseScript(interp, "#pragma remvar(a)");
Ch_AppendParseScript(interp, "#pragma remvar(b)");
```

An identifier can be declared as both global and system variable. In this case, the identifier will be treated as a global variable inside a program. For a variable of function type, only one function definition can be defined if the identifier is used for both global and system variable.

A struct type used as a function argument or return type of a system function defined by function **Ch_DeclareFunc**() can be defined by .

Example
Declare and use system variables of different data types in the Ch space.

```
/* File Name: declarevar.c */
#include <stdio.h>
#include <embedch.h>

int main() {
    ChInterp_t interp;
    int status;
    int c_i;

    /* initialize embedded Ch */
    Ch_Initialize(&interp, NULL);
    /*  declare variable ch_i in Ch space */
    Ch_DeclareVar(interp,"int ch_i;");
    /*  declare and initialize variables ch_i, ch_j, ch_k, ch_p */
    Ch_DeclareVar(interp,"int ch_j = 10, ch_k = 20;");
    Ch_DeclareVar(interp,"int *ch_p = &ch_i;");
    /* declare struct */
    Ch_DeclareVar(interp,"struct tag1 {int ch_i, *ch_p;} s;");
    /* declare function */
    Ch_DeclareVar(interp,"int func(int i) {int i2; i2 = 5*i; return i2;}");

    c_i = 10;
    /* assign c_i in C space to ch_i in Ch space */
    Ch_SetVar(interp, "ch_i", CH_INTTYPE, c_i);
    Ch_ExprEval(interp, "printf(\"ch_i = %d\n\",ch_i)");
    Ch_ExprEval(interp, "printf(\"*ch_p = %d\n\",*ch_p)");
    /* assign &c_i in C space to ch_p in Ch space */
    Ch_SetVar(interp, "ch_p", CH_INTPTRTYPE, &c_i);
    /* call function func() */
    Ch_ExprEval(interp, "printf(\"func(ch_i) = %d\n\",func(ch_i))");
    Ch_End(interp);
    return 0;
}
```

Output

```
ch_i = 10
*ch_p = 10
func(ch_i) = 50
```

See Also
Ch_DeclareFunc(), **Ch_DeclareTypedef**(), **Ch_AppendParseScript**(), **Ch_AppendRunScript**(),

Ch_DeleteExprValue

Synopsis
#include <**ch.h**>
int Ch_DeleteExprValue(**ChInterp_t** *interp*, **ChValueNode_t** *vn*);

Purpose
Delete a value node for an expression in the Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

vn The value node of an expression to be deleted.

Description
When an expression in the Ch space is evaluated by the function **Ch_ExprValue**(), a value node for the result of the evaluated expression is returned to keep track of the temporary memory. This memory should be deleted later by this function **Ch_DeleteExprValue**(). When the value node corresponding to the result of an expression is deleted, the result cannot be accessed any more.

Example
See function `chPrint()` in Program 7.16.

See Also
Ch_SymbolAddrByName(), **Ch_DataType**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_ExprCalc**(), **Ch_ExprParse**(), **Ch_ExprEval**(), **Ch_ExprValue**().

Ch_End

Synopsis
#include <**embedch.h**>
int Ch_End(**ChInterp_t** *interp*);

Purpose
End embedded Ch.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

Description
The function **Ch_End**() ends embedded Ch which is initialized by the API **Ch_Initialize**(). It is mandatory if the API **Ch_Initialize**() is called.
Normally, functions registered in **atexit**() for Embedded Ch programs are executed in **Ch_End**(). When a Ch script is executed by **Ch_ExecScriptM**() or **Ch_RunScriptM**(), registered functions in **atexit**() will be executed when the script is executed. Therefore, a single instance of Ch interpreter initialized by **Ch_Initialize**() can be used to process multiple Ch scripts using **Ch_ExecScriptM**() or **Ch_RunScriptM**().
Example
Refer to Program 1.1.

See Also
Ch_Abort(), **Ch_Initialize**(), **Ch_Home**(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_RunScript**().

Ch_ExecScript

Synopsis
#include <**embedch.h**>
int Ch_ExecScript(**ChInterp_t** *interp*, **const char** **progname*);

Purpose
Execute a Ch program parsed by function **Ch_ParseScript**().
Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

programe The program name, the first element of the second argument of array of pointer to char in function **Ch_ParseScript**().

Description
When a program is run in Ch, it first will be parsed, then executed. The function **Ch_ExecScript**() executes a Ch program parsed by function **Ch_ParseScript**() in C space. The variable *progname* contains the file name of the Ch program parsed by function **Ch_ParseScript**(). The API **Ch_Initialize**() and **Ch_ParseScript**() should be called before this function is called. After function **Ch_ExecScript**() is called, variables and functions inside the script can be accessed by other APIs in the same interpreter `interp`. However, after function **Ch_ExecScriptM**() is called, variables and functions inside the script cannot be accessed by other APIs in the same interpreter `interp`.

Example
Refer to Program 1.4.

See Also
Ch_AppendParseScript(), **Ch_AppendRunScript**(), **Ch_ExecScriptM**(), **Ch_ParseScript**(), **Ch_RunScript**(). **Ch_Initialize**(), **Ch_Home**(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_End**().

Ch_ExecScriptM

Synopsis
#include <**embedch.h**>
int Ch_ExecScriptM(**ChInterp_t** *interp*, **const char** **progname*);

Purpose
Execute a Ch program parsed by function **Ch_ParseScript**().
Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

programe The program name, the first element of the second argument of array of pointer to char in function **Ch_ParseScript**().

Description
When a program is run in Ch, it first will be parsed, then executed. The function **Ch_ExecScriptM**() executes a Ch program parsed by function **Ch_ParseScript**() in C space. The variable *progname* contains the file name of the Ch program parsed by function **Ch_ParseScript**(). The API **Ch_Initialize**() and **Ch_ParseScript**() should be called before this function is called. After function **Ch_ExecScript**() is called, variables and functions inside the script can be accessed by other APIs in the same interpreter `interp`. However, after function **Ch_ExecScriptM**() is called, variables and functions inside the script cannot be accessed by other APIs in the same interpreter `interp`. But, a single instance of Ch interpreter initialized by **Ch_Initialize**() can be used to process multiple Ch scripts using **Ch_ParseScript**() and **Ch_ExecScriptM**(), or **Ch_RunScriptM**().

Example
Refer to Program 1.8.

See Also
Ch_AppendParseScript(), **Ch_AppendRunScript**(), **Ch_ExecScript**(), **Ch_ParseScript**(), **Ch_RunScript**(). **Ch_RunScriptM**(). **Ch_Initialize**(), **Ch_Home**(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_End**().

Ch_ExprCalc

Synopsis
#include <**ch.h**>
int Ch_ExprCalc(**ChInterp_t** *interp*, **const char** **expr*, **ChType_t** *datatype*, **void** **result*);

Purpose
Calculate an expression in Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

expr Expression to be executed in Ch space.

datatype The data type of the result.

result Pointer to the memory for holding the result of the expression.

Description
The function **Ch_ExprCalc**() executes an expression in Ch space. It can be used to evaluate an expression or call a function in Ch space. The expression `expr` may invoke functions located in 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 an operand of an expression. If the actual argument for parameter `result` of data type `datatype`, is not a NULL pointer, the result will be assigned to the memory pointed by `result` based on the data type **ChType_t** of the argument `datatype` listed in the header file ch.h described in *Ch SDK User's Guide.* The user has to make sure a memory of the proper size is allocated for `result`. If the passed argument of `result` is NULL, no value will be assigned. In this case, the *datatype* is ignored. The data type of the `result` shall be compatible with the data type of the expression.

Example
In this example, there are two global variables in program `exprcalc.ch`. *ii* is an integer and *dd* is a value of double type. We can calculate the expression `dd = dd + ii` from C space by function **Ch_ExprCalc**() and get the new value of *dd* by function **Ch_SymbolAddrByName**(). The result can also be obtained by passing the address for the result in the third argument of function. **Ch_ExprCalc**(). We can call the Ch function `fun`() through evaluating expression "fun()" in `exprcalc.c`.

chexprcalc.c —- A C program to calculate expressions in Ch address space

```
/*************************************************************
* Calculate an expression and function in Ch address space
*************************************************************/
```

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

EXPORTCH void chexprcalc_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int *pii;
  double *pdd;
  double result;

  Ch_VaStart(interp, ap, varg);
  pii = Ch_SymbolAddrByName(interp,"ii");
  pdd = Ch_SymbolAddrByName(interp,"dd");
  printf("ii = %d\n",*pii);
  printf("dd = %f\n",*pdd);
  if(!Ch_ExprParse(interp,"fun()")) {
    Ch_ExprCalc(interp,"fun()", CH_INTTYPE, NULL);
  }
  else
    printf("Error: fun() is invalid Ch expression\n");

  Ch_ExprCalc(interp,"dd = dd + ii", CH_DOUBLETYPE, &result);
  printf("\nAfter dd = dd + ii\n");
  printf("ii = %d\n",*pii);
  printf("dd = %f\n",*pdd);
  printf("result = %f\n",result);
  Ch_ExprCalc(interp,"ii*2", CH_DOUBLETYPE, &result);
  printf("ii*2= %f\n",result);
  Ch_ExprCalc(interp,"fun()", CH_VOIDTYPE, NULL);
  Ch_ExprCalc(interp,"hypot(3, 4.0)", CH_DOUBLETYPE, &result);
  printf("hypot(3, 4) = %f\n",result);

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

chexprcalc.ch —- A Ch function file with global variables and functions

```
/**********************************
*  Test the function Ch_ExprCalc() in C space
**********************************/

#include<dlfcn.h>

void chexprcalc() {

  void *dlhandle, *fptr;

  dlhandle = dlopen("libch.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "chexprcalc_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  dlrunfun(fptr, NULL, chexprcalc);
```

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

  return;
}

/*************** application *****************/

int ii = 10;
double dd = 1.1;
void fun() {
  printf("in fun() in exprcalc.ch, ii = %d, dd = %f\n", ii, dd);
}
chexprcalc();
```

Output

```
ii = 10
dd = 1.100000
in fun() in exprcalc.ch, ii = 10, dd = 1.100000

After dd = dd + ii
ii = 10
dd = 11.100000
result = 11.100000
ii*2= 20.000000
in fun() in exprcalc.ch, ii = 10, dd = 11.100000
hypot(3, 4) = 5.000000
```

See Also

Ch_SymbolAddrByName(), **Ch_DataType**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_ExprParse**(), **Ch_ExprValue**(), **Ch_DeleteExprValue**().

Ch_ExprEval

Synopsis
#include <**ch.h**>
int Ch_ExprEval(ChInterp_t *interp*, **const char** **expr*);

Purpose
Execute an expression in Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

expr Expression to be executed in Ch space.

Description
The function **Ch_ExprEval**() executes an expression in Ch space. It can be used to evaluate an expression or call a function in Ch space. The expression `expr` may invoke functions located in 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 an operand of an expression.
Example
In this example, there are two global variables in program `expreval.ch`. *ii* is an integer and *dd* is a value of double type. We can evaluate the expression `dd = dd + ii` from C space by function **Ch_ExprEval**() and get the new value of *dd* by function **Ch_SymbolAddrByName**(). We can call the Ch function `fun`() through evaluating expression "fun()" in `expreval.c`.

chexpreval.c —- A C program to evaluate expressions in Ch address space

```
/*****************************************************************
* Execute expression and function in Ch address space
*****************************************************************/

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

EXPORTCH void chexpreval_chdl(void *varg) {
  ChInterp_t interp;
  ChVaList_t ap;
  int *pii;
  double *pdd;

  Ch_VaStart(interp, ap, varg);
  pii = Ch_SymbolAddrByName(interp,"ii");
  pdd = Ch_SymbolAddrByName(interp,"dd");
  printf("ii = %d\n",*pii);
```

```
  printf("dd = %f\n",*pdd);
  if(!Ch_ExprParse(interp,"fun()")) {
    Ch_ExprEval(interp,"fun()");
  }
  else
    printf("Error: fun() is invalid Ch expression\n");

  Ch_ExprEval(interp,"dd = dd + ii");
  printf("\nAfter dd = dd + ii\n");
  printf("ii = %d\n",*pii);
  printf("dd = %f\n",*pdd);
  Ch_ExprEval(interp,"fun()");

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

chexpreval.c —- A Ch function file with global variables and functions

```
/**********************************
*  Test the function Ch_ExprEval() in C space
**********************************/

#include<dlfcn.h>

void chexpreval() {

  void *dlhandle, *fptr;

  dlhandle = dlopen("libch.dl", RTLD_LAZY);
  if(dlhandle == NULL) {
    printf("Error: %s(): dlopen(): %s\n", __func__, dlerror());
    return;
  }
  fptr = dlsym(dlhandle, "chexpreval_chdl");
  if(fptr == NULL) {
    printf("Error: %s(): dlsym(): %s\n", __func__, dlerror());
    return;
  }

  dlrunfun(fptr, NULL, chexpreval);

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

  return;
}

/*************** application ****************/


int ii = 10;
double dd = 1.1;

void fun() {
  printf("in fun() in expreval.ch, ii = %d, dd = %f\n", ii, dd);
}
```

```
chexpreval();
```

Output

```
ii = 10
dd = 1.100000
in fun() in expreval.ch, ii = 10, dd = 1.100000

After dd = dd + ii
ii = 10
dd = 11.100000
in fun() in expreval.ch, ii = 10, dd = 11.100000
```

See Also

Ch_SymbolAddrByName(), **Ch_DataType**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_ExprCalc**(), **Ch_ExprParse**(), **Ch_ExprValue**(), **Ch_DeleteExprValue**().

Ch_ExprParse

Synopsis
#include <**ch.h**>
int Ch_ExprParse(**ChInterp_t** *interp*, **const char** **expr*);

Purpose
Parse an expression in Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

expr Expression in Ch space to be parsed.

Description
The function **Ch_ExprParse**() parses an expression in Ch space. If the expression is syntactically correct, it can be evaluated by function **Ch_ExprEval**(). The expression `expr` may invoke functions located in 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 an operand of an expression.
Example
See **Ch_ExprParse**().

See Also
Ch_SymbolAddrByName(), **Ch_DataType**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_ExprEval**(), **Ch_ExprValue**(), **Ch_DeleteExprValue**().

Ch_ExprValue

Synopsis
#include <**ch.h**>
ChValueNode_t Ch_ExprValue(**ChInterp_t** *interp*, **const char** *expr, **void** `*result`);

Purpose
Obtain the value of an expression in the Ch space.

Return Value
This function returns the value node of an expression in the Ch space on success and NULL on failure.

Parameters
interp A Ch interpreter.

expr The expression to be evaluated in the Ch space.

result the result of the evaluated expression.

Description
The function **Ch_ExprValue**() executes an expression in the Ch space. It can be used to evaluate an expression or call a function in Ch space. The expression `expr` may invoke functions located in 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 an operand of an expression.

The address for the result of the evaluated expression is passed through a generic pointer in the third argument of the function. The function returns a value node for the result of the evaluated expression. This value node is used to keep track of the temporary memory of the result of the evaluated expression. This value node should deleted by the function **Ch_DeleteExprValue**() later to free the allocated memory. See API **Ch_SymbolAddrByName**() how the returned address of an expression is handled for some special cases.

Unlike function **Ch_ExprCalc**(), the user does not need to declare a variable of a proper data type to obtain the value of an expression.

Example
See function `chPrint()` in Program 7.16.

See Also
Ch_SymbolAddrByName(), **Ch_DataType**(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**(), **Ch_ExprCalc**(), **Ch_ExprParse**(), **Ch_ExprEval**(), **Ch_DeleteExprValue**().

Ch_Flush

Synopsis
#include <**embedch.h**>
int Ch_Flush(**ChInterp_t** *interp*, **ChFile_t** *filedes*);

Purpose
Flush the contents of the buffer associated with a file descriptor.

Return Value
Upon successful completion, **Ch_Flush**() returns 0. Otherwise, it returns EOF.

Parameters
interp A Ch interpreter.

fildes The file descriptor for the stream to be flushed.

Description
The **Ch_Flush**() function calls **fflush()** to flush the stream associated with the file descriptor `filedes` returned from function **Ch_Reopen**().

See Also
Ch_Reopen(), **Ch_Close**().

Ch_FuncArgArrayDim

Synopsis
#include <**ch.h**>
int Ch_FuncArgArrayDim(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Obtain the dimension for an argument of array type for a function of global variable in the Ch space.
Return Value
If the argument of the function for a global variable in the Ch space is of array type, This function returns the dimension of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
This function returns the dimension of array type for a function of global variable in the Ch space. The extent of each dimension of the array can be obtained by function **Ch_FuncArgArrayExtent**().

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgArrayExtent

Synopsis
#include <**ch.h**>
int Ch_FuncArgArrayExtent(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*, **int** *dim*);

Purpose
Obtain the number of elements in the specified dimension of for an argument of array type for a function of global variable in the Ch space.

Return Value
If the argument for a function of global variable is of array type, This function returns the number of elements in a specified dimension of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

dim An integer specifying for which dimension the number of elements will be obtained, 0 for the first dimension.

Description
This function returns the number of elements in the specified dimension of an argument of array type of a function in the Ch space. For array `a` in `int func(int a[n][m])`, If `0` is passed to this function as the fourth argument, the extent of the first dimension, i.e. `n`, will be returned. Otherwise, if `1` is passed to, `m` will be returned.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgArrayNum

Synopsis
#include <**ch.h**>
int Ch_FuncArgArrayNum(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Obtain the number of elements for an argument of array type for a function of global variable in the Ch space.
Return Value
If the argument of the function for a global variable in the Ch space is of array type, This function returns the number of elements of the array. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
This function returns the number of elements of array type for a function of global variable in the Ch space. The dimension of the array can be obtained by function **Ch_FuncArgArrayDim**().

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(),
Ch_FuncArgIsFuncVarArg(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgArrayType

Synopsis
#include <**ch.h**>
ChType_t Ch_FuncArgArrayType(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Determine if an argument of a function in the Ch space is array and its array type.

Return Value
Based on its argument, this function returns one of the macros below, 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.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
The function **Ch_FuncArgArrayType**() determines if the argument of a function is array type. The data type of the array for an argument of a function shall be determined by function **Ch_FuncArgDataType**(). Unlike function **Ch_ArrayType**(), a poiner to C array is treated as an array by function **Ch_FuncArgArrayType**() so that the complete information for an argument can be obtained.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgDataType

Synopsis
#include <**ch.h**>
ChType_t Ch_FuncArgDataType(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Get the data type for an argument of a function in a Ch program.

Return Value
This function returns a macro for data type **ChType_t** defined in header file **ch.h** described in *Ch SDK User's Guide.* If a passed argument is of array type, the function will return the type of the element of the array. 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.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
The function **Ch_FuncArgDataType**() gets the data type for an argument of a function in the Ch space. If a variable of pointer to computational array, it gives the data type of its element. Unlike function **Ch_FuncArgDataType**(), For a pointer to C array, it also gives data type of its element so that the complete information for an argument can be obtained.

Example
In this example, information about arguments of functions in the embedded Ch program `funcarg.ch` are obtained in the hosting C program `funcarg.c`.

funcarg.c —- A C program to get information for global variables in Ch.

```
/*****************************************************************
* File Name: funcarg.c
*****************************************************************/
#include <stdio.h>
#include <embedch.h>

struct tag {
  int i;
  double f;
};

int main() {
  ChInterp_t interp;
  int status;
  char *argvv[]={"funcarg.ch", NULL};
  struct tag *sp, **sp2;
```

```
Ch_Initialize(&interp, NULL);
status = Ch_RunScript(interp,argvv);
if(status == CH_ERROR) {
  printf("Error: execution of program funcarg.ch failed\n");
}

if(Ch_FuncArgDataType(interp,"f1", 0) == CH_INTTYPE)
  printf("arg 1 of f1 is int \n");
if(Ch_FuncArgDataType(interp,"f1", 1) == CH_UNDEFINETYPE)
  printf("arg 2 of f1 is undefined \n");
if(Ch_FuncArgDataType(interp,"f2", 0) == CH_UNDEFINETYPE)
  printf("arg 1 of f2 is undefined \n");
if(Ch_FuncArgDataType(interp,"f3", 1) == CH_DOUBLEPTRTYPE)
  printf("arg 2 of f3 is double *\n");
if(Ch_FuncArgDataType(interp,"f3", 2) == CH_CHARPTRTYPE)
  printf("arg 3 of f3 is char*\n");
if(Ch_FuncArgDataType(interp,"f4", 1) == CH_INTTYPE)
  printf("arg 2 of f4 is int \n");
if(Ch_FuncArgDataType(interp,"f4", 2) == CH_INTTYPE)
  printf("arg 3 of f4 is int \n");
if(Ch_FuncArgDataType(interp,"f4", 3) == CH_INTTYPE)
  printf("arg 4 of f4 is int \n");
if(Ch_FuncArgDataType(interp,"f4", 4) == CH_INTTYPE)
  printf("arg 5 of f4 is int \n");
if(Ch_FuncArgDataType(interp,"f5", 1) == CH_INTTYPE)
  printf("arg 2 of f5 is int \n");
if(Ch_FuncArgDataType(interp,"f5", 2) == CH_INTTYPE)
  printf("arg 3 of f5 is int \n");
if(Ch_FuncArgDataType(interp,"f5", 3) == CH_INTTYPE)
  printf("arg 4 of f5 is int \n");
if(Ch_FuncArgDataType(interp,"f5", 4) == CH_INTTYPE)
  printf("arg 5 of f5 is int \n");
if(Ch_FuncArgDataType(interp,"f5", 5) == CH_INTTYPE)
  printf("arg 6 of f5 is int \n");
if(Ch_FuncArgDataType(interp,"f6", 0) == CH_STRUCTTYPE)
  printf("arg 1 of f6 is struct\n");
if(Ch_FuncArgDataType(interp,"f6", 1) == CH_STRUCTPTRTYPE)
  printf("arg 2 of f6 is struct* \n");
if(Ch_FuncArgDataType(interp,"f6", 2) == CH_CLASSTYPE)
  printf("arg 3 of f6 is class* \n");
if(Ch_FuncArgDataType(interp,"f6", 3) == CH_CLASSPTRTYPE)
  printf("arg 4 of f6 is class* \n");
if(Ch_FuncArgDataType(interp,"f6", 4) == CH_UNIONTYPE)
  printf("arg 5 of f6 is union \n");
if(Ch_FuncArgDataType(interp,"f6", 5) == CH_UNIONPTRTYPE)
  printf("arg 6 of f6 is union* \n");
if(Ch_FuncArgDataType(interp,"f6", 6) == CH_STRUCTTYPE)
  printf("arg 7 of f6 is struct\n");
if(Ch_FuncArgDataType(interp,"f7", 0) == CH_INTTYPE)
  printf("arg 1 of f7 is int \n");
if(Ch_FuncArgDataType(interp,"f7", 1) == CH_INTTYPE)
  printf("arg 2 of f7 is int \n");
if(Ch_FuncArgDataType(interp,"f7", 2) == CH_INTTYPE)
  printf("arg 3 of f7 is int\n");
if(Ch_FuncArgDataType(interp,"f8", 0) == CH_INTTYPE)
  printf("arg 1 of f8 is int\n");
if(Ch_FuncArgDataType(interp,"f8", 1) == CH_INTTYPE)
  printf("arg 2 of f8 is int \n");
```

```
if(Ch_FuncArgDataType(interp,"f8", 2) == CH_INTTYPE)
  printf("arg 3 of f8 is int\n");

if(!Ch_FuncArgArrayType(interp,"f1", 0))
  printf("arg 1 of f1 is not array\n");
if(!Ch_FuncArgArrayType(interp,"f1", 4))
  printf("arg 4 of f1 is not array\n");
if(!Ch_FuncArgArrayType(interp,"f2", 0))
  printf("arg 1 of f2 is not array\n");
if(Ch_FuncArgArrayType(interp,"f4", 1) == CH_CARRAYPTRTYPE)
  printf("arg 2 of f4 is pointer to C array\n");
if(Ch_FuncArgArrayType(interp,"f4", 2) == CH_CARRAYVLATYPE)
  printf("arg 3 of f4 is C VLA array\n");
if(Ch_FuncArgArrayType(interp,"f4", 3) == CH_CARRAYVLATYPE)
  printf("arg 4 of f4 is C VLA array\n");
if(Ch_FuncArgArrayType(interp,"f4", 4) == CH_CARRAYVLATYPE)
  printf("arg 5 of f4 is C VLA array\n");
if(Ch_FuncArgArrayType(interp,"f5", 1) == CH_CHARRAYPTRTYPE)
  printf("arg 2 of f5 is pointer to C array\n");
if(Ch_FuncArgArrayType(interp,"f5", 2) == CH_CHARRAYVLATYPE)
  printf("arg 3 of f5 is Ch VLA array\n");
if(Ch_FuncArgArrayType(interp,"f5", 3) == CH_CHARRAYVLATYPE)
  printf("arg 4 of f5 is Ch VLA array\n");
if(Ch_FuncArgArrayType(interp,"f5", 4) == CH_CHARRAYVLATYPE)
  printf("arg 5 of f5 is Ch VLA array\n");
if(Ch_FuncArgArrayType(interp,"f5", 5) == CH_CHARRAYVLATYPE)
  printf("arg 6 of f5 is Ch VLA array\n");
if(Ch_FuncArgArrayType(interp,"f6", 6) == CH_CARRAYPTRTYPE)
  printf("arg 7 of f6 is pointer to C array\n");
if(Ch_FuncArgArrayType(interp,"f8", 0) == CH_CHARRAYTYPE)
  printf("arg 1 of f8 is Ch array\n");
if(Ch_FuncArgArrayType(interp,"f8", 1) == CH_CHARRAYVLATYPE)
  printf("arg 2 of f8 is Ch VLA array\n");
if(Ch_FuncArgArrayType(interp,"f8", 2) == CH_CHARRAYVLATYPE)
  printf("arg 3 of f8 is Ch VLA array\n");

printf("number of elements of arg 1 of f1 is %d\n", Ch_FuncArgArrayNum(interp,"f1", 0));
printf("number of elements of arg 4 of f1 is %d\n", Ch_FuncArgArrayNum(interp,"f1", 3));
printf("number of elements of arg 1 of f2 is %d\n", Ch_FuncArgArrayNum(interp,"f2", 0));
printf("number of elements of arg 2 of f4 is %d\n", Ch_FuncArgArrayNum(interp,"f4", 1));
printf("number of elements of arg 3 of f4 is %d\n", Ch_FuncArgArrayNum(interp,"f4", 2));
printf("number of elements of arg 4 of f4 is %d\n", Ch_FuncArgArrayNum(interp,"f4", 3));
printf("number of elements of arg 5 of f4 is %d\n", Ch_FuncArgArrayNum(interp,"f4", 4));
printf("number of elements of arg 2 of f5 is %d\n", Ch_FuncArgArrayNum(interp,"f5", 1));
printf("number of elements of arg 3 of f5 is %d\n", Ch_FuncArgArrayNum(interp,"f5", 2));
printf("number of elements of arg 4 of f5 is %d\n", Ch_FuncArgArrayNum(interp,"f5", 3));
printf("number of elements of arg 5 of f5 is %d\n", Ch_FuncArgArrayNum(interp,"f5", 4));
printf("number of elements of arg 6 of f5 is %d\n", Ch_FuncArgArrayNum(interp,"f5", 5));
printf("number of elements of arg 7 of f6 is %d\n", Ch_FuncArgArrayNum(interp,"f6", 6));
printf("number of elements of arg 1 of f8 is %d\n", Ch_FuncArgArrayNum(interp,"f8", 0));
printf("number of elements of arg 2 of f8 is %d\n", Ch_FuncArgArrayNum(interp,"f8", 1));
printf("number of elements of arg 3 of f8 is %d\n", Ch_FuncArgArrayNum(interp,"f8", 2));

printf("dim of arg 1 of f1 is %d\n", Ch_FuncArgArrayDim(interp,"f1", 0));
printf("dim of arg 4 of f1 is %d\n", Ch_FuncArgArrayDim(interp,"f1", 3));
printf("dim of arg 1 of f2 is %d\n", Ch_FuncArgArrayDim(interp,"f2", 0));
printf("dim of arg 2 of f4 is %d\n", Ch_FuncArgArrayDim(interp,"f4", 1));
printf("dim of arg 3 of f4 is %d\n", Ch_FuncArgArrayDim(interp,"f4", 2));
printf("dim of arg 4 of f4 is %d\n", Ch_FuncArgArrayDim(interp,"f4", 3));
```

```
printf("dim of arg 5 of f4 is %d\n", Ch_FuncArgArrayDim(interp,"f4", 4));
printf("dim of arg 2 of f5 is %d\n", Ch_FuncArgArrayDim(interp,"f5", 1));
printf("dim of arg 3 of f5 is %d\n", Ch_FuncArgArrayDim(interp,"f5", 2));
printf("dim of arg 4 of f5 is %d\n", Ch_FuncArgArrayDim(interp,"f5", 3));
printf("dim of arg 5 of f5 is %d\n", Ch_FuncArgArrayDim(interp,"f5", 4));
printf("dim of arg 6 of f5 is %d\n", Ch_FuncArgArrayDim(interp,"f5", 5));
printf("dim of arg 7 of f6 is %d\n", Ch_FuncArgArrayDim(interp,"f6", 6));
printf("dim of arg 1 of f8 is %d\n", Ch_FuncArgArrayDim(interp,"f8", 0));
printf("dim of arg 2 of f8 is %d\n", Ch_FuncArgArrayDim(interp,"f8", 1));
printf("dim of arg 3 of f8 is %d\n", Ch_FuncArgArrayDim(interp,"f8", 2));

printf("Extent 1 of arg 1 of f1 is %d\n", Ch_FuncArgArrayExtent(interp,"f1", 0, 0));
printf("Extent 1 of arg 4 of f1 is %d\n", Ch_FuncArgArrayExtent(interp,"f1", 3, 0));
printf("Extent 1 of arg 1 of f2 is %d\n", Ch_FuncArgArrayExtent(interp,"f2", 0, 0));
printf("Extent 1 of arg 2 of f4 is %d\n", Ch_FuncArgArrayExtent(interp,"f4", 1, 0));
printf("Extent 2 of arg 2 of f4 is %d\n", Ch_FuncArgArrayExtent(interp,"f4", 1, 1));
printf("Extent 1 of arg 3 of f4 is %d\n", Ch_FuncArgArrayExtent(interp,"f4", 2, 0));
printf("Extent 1 of arg 4 of f4 is %d\n", Ch_FuncArgArrayExtent(interp,"f4", 3, 0));
printf("Extent 1 of arg 5 of f4 is %d\n", Ch_FuncArgArrayExtent(interp,"f4", 4, 0));
printf("Extent 1 of arg 2 of f5 is %d\n", Ch_FuncArgArrayExtent(interp,"f5", 1, 0));
printf("Extent 2 of arg 2 of f5 is %d\n", Ch_FuncArgArrayExtent(interp,"f5", 1, 1));
printf("Extent 1 of arg 3 of f5 is %d\n", Ch_FuncArgArrayExtent(interp,"f5", 2, 0));
printf("Extent 1 of arg 4 of f5 is %d\n", Ch_FuncArgArrayExtent(interp,"f5", 3, 0));
printf("Extent 1 of arg 5 of f5 is %d\n", Ch_FuncArgArrayExtent(interp,"f5", 4, 0));
printf("Extent 1 of arg 6 of f5 is %d\n", Ch_FuncArgArrayExtent(interp,"f5", 5, 0));
printf("Extent 1 of arg 7 of f6 is %d\n", Ch_FuncArgArrayExtent(interp,"f6", 6, 0));
printf("Extent 1 of arg 1 of f8 is %d\n", Ch_FuncArgArrayExtent(interp,"f8", 0, 0));
printf("Extent 2 of arg 1 of f8 is %d\n", Ch_FuncArgArrayExtent(interp,"f8", 0, 1));
printf("Extent 1 of arg 2 of f8 is %d\n", Ch_FuncArgArrayExtent(interp,"f8", 1, 0));
printf("Extent 2 of arg 2 of f8 is %d\n", Ch_FuncArgArrayExtent(interp,"f8", 1, 1));
printf("Extent 1 of arg 3 of f8 is %d\n", Ch_FuncArgArrayExtent(interp,"f8", 2, 0));
printf("Extent 2 of arg 3 of f8 is %d\n", Ch_FuncArgArrayExtent(interp,"f8", 2, 1));

printf("arg 1 of f1 is function %d\n", Ch_FuncArgIsFunc(interp,"f1", 0));
printf("arg 4 of f1 is function %d\n", Ch_FuncArgIsFunc(interp,"f1", 3));
printf("arg 1 of f7 is function %d\n", Ch_FuncArgIsFunc(interp,"f7", 0));
printf("arg 2 of f7 is function %d\n", Ch_FuncArgIsFunc(interp,"f7", 1));
printf("arg 3 of f7 is function %d\n", Ch_FuncArgIsFunc(interp,"f7", 2));
printf("arg 4 of f7 is function %d\n", Ch_FuncArgIsFunc(interp,"f7", 3));
printf("arg 1 of f8 is function %d\n", Ch_FuncArgIsFunc(interp,"f8", 0));
printf("arg 2 of f8 is function %d\n", Ch_FuncArgIsFunc(interp,"f8", 1));
printf("arg 3 of f8 is function %d\n", Ch_FuncArgIsFunc(interp,"f8", 2));

printf("arg 1 of f1 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f1", 0));
printf("arg 4 of f1 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f1", 3));
printf("arg 1 of f7 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f7", 0));
printf("arg 2 of f7 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f7", 1));
printf("arg 3 of f7 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f7", 2));
printf("arg 4 of f7 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f7", 3));
printf("arg 1 of f8 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f8", 0));
printf("arg 2 of f8 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f8", 1));
printf("arg 3 of f8 is variable arg %d\n", Ch_FuncArgIsFuncVarArg(interp,"f8", 2));


printf("func arg num for arg 1 of f1 is %d\n", Ch_FuncArgFuncArgNum(interp,"f1", 0));
printf("func arg num for arg 4 of f1 is %d\n", Ch_FuncArgFuncArgNum(interp,"f1", 3));
printf("func arg num for arg 1 of f7 is %d\n", Ch_FuncArgFuncArgNum(interp,"f7", 0));
printf("func arg num for arg 2 of f7 is %d\n", Ch_FuncArgFuncArgNum(interp,"f7", 1));
printf("func arg num for arg 3 of f7 is %d\n", Ch_FuncArgFuncArgNum(interp,"f7", 2));
```

```
  printf("func arg num for arg 4 of f7 is %d\n", Ch_FuncArgFuncArgNum(interp,"f7", 3));
  printf("func arg num for arg 1 of f8 is %d\n", Ch_FuncArgFuncArgNum(interp,"f8", 0));
  printf("func arg num for arg 2 of f8 is %d\n", Ch_FuncArgFuncArgNum(interp,"f8", 1));
  printf("func arg num for arg 3 of f8 is %d\n", Ch_FuncArgFuncArgNum(interp,"f8", 2));


  printf("data size for arg 1 of f1 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f1", 0));
  printf("data size for arg 4 of f1 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f1", 0));
  printf("data size for arg 1 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 0));
  printf("data size for arg 2 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 1));
  printf("data size for arg 3 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 2));
  printf("data size for arg 4 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 3));
  printf("data size for arg 5 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 4));
  printf("data size for arg 6 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 5));
  printf("data size for arg 7 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 6));
  printf("data size for arg 8 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 7));
  printf("data size for arg 9 of f6 is %d\n", Ch_FuncArgUserDefinedSize(interp,"f6", 8));

  printf("data name for arg 1 of f1 is %p\n", Ch_FuncArgUserDefinedName(interp,"f1", 0));
  printf("data name for arg 4 of f1 is %p\n", Ch_FuncArgUserDefinedName(interp,"f1", 0));
  printf("data name for arg 1 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 0));
  printf("data name for arg 2 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 1));
  printf("data name for arg 3 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 2));
  printf("data name for arg 4 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 3));
  printf("data name for arg 5 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 4));
  printf("data name for arg 6 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 5));
  printf("data name for arg 7 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 6));
  printf("data name for arg 8 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 7));
  printf("data name for arg 9 of f6 is %s\n", Ch_FuncArgUserDefinedName(interp,"f6", 8));

  Ch_End(interp);
}
```

funcarg.ch —- A Ch program with global variables.

```
#include <array.h>

struct tag {
  int i;
  double f;
};
class tagc {
  public:
    int i;
    double f;
};
union tagu {
    int i;
    double f;
};

int f1(int a);
int f2();
void f3(int a, double *dp, char *cp);
void f4(int n, int a1[2][3], int a2[n][n], int a3[:], int a4[&]);
void f5(int n, array int a1[2][3],  array int a2[n][n],
        array int a3[:], array int a4[&], array int &a);
void f6(struct tag s, struct tag *sp,
        class tagc c, class tagc *cp,
        union tagu u, union tagu *up,
```

```
        struct tag sa[4],
        struct tag (*f1)(int i),
        struct tag *(*f2)(int i));
void f7(int (*f1)(int), int (*f2)(),
        int (*f3)(int, double *, char *),
        int (*f4)(int, ...));
void f8(array double (*f1)(int)[2][3],
        array double (*f2)(int)[:][:],
        array double (*f3)(int)[&][&]);
```

Output

```
arg 1 of f1 is int
arg 2 of f1 is undefined
arg 1 of f2 is undefined
arg 2 of f3 is double *
arg 3 of f3 is char*
arg 2 of f4 is int
arg 3 of f4 is int
arg 4 of f4 is int
arg 5 of f4 is int
arg 2 of f5 is int
arg 3 of f5 is int
arg 4 of f5 is int
arg 5 of f5 is int
arg 6 of f5 is int
arg 1 of f6 is struct
arg 2 of f6 is struct*
arg 3 of f6 is class*
arg 4 of f6 is class*
arg 5 of f6 is union
arg 6 of f6 is union*
arg 7 of f6 is struct
arg 1 of f7 is int
arg 2 of f7 is int
arg 3 of f7 is int
arg 1 of f1 is not array
arg 4 of f1 is not array
arg 1 of f2 is not array
arg 2 of f4 is pointer to C array
arg 3 of f4 is C VLA array
arg 4 of f4 is C VLA array
arg 5 of f4 is C VLA array
arg 2 of f5 is pointer to C array
arg 3 of f5 is Ch VLA array
arg 4 of f5 is Ch VLA array
arg 5 of f5 is Ch VLA array
arg 6 of f5 is Ch VLA array
arg 7 of f6 is pointer to C array
arg 1 of f8 is Ch array
arg 2 of f8 is Ch VLA array
arg 3 of f8 is Ch VLA array
number of elements of arg 1 of f1 is 0
number of elements of arg 4 of f1 is 0
number of elements of arg 1 of f2 is 0
number of elements of arg 2 of f4 is 6
number of elements of arg 3 of f4 is 1
number of elements of arg 4 of f4 is 2147483647
number of elements of arg 5 of f4 is 2147483647
number of elements of arg 2 of f5 is 6
```

```
number of elements of arg 3 of f5 is 1
number of elements of arg 4 of f5 is 2147483647
number of elements of arg 5 of f5 is 2147483647
number of elements of arg 6 of f5 is 2147483647
number of elements of arg 7 of f6 is 4
number of elements of arg 1 of f8 is 6
number of elements of arg 2 of f8 is 1
number of elements of arg 3 of f8 is 1
dim of arg 1 of f1 is 0
dim of arg 4 of f1 is 0
dim of arg 1 of f2 is 0
dim of arg 2 of f4 is 2
dim of arg 3 of f4 is 2
dim of arg 4 of f4 is 1
dim of arg 5 of f4 is 1
dim of arg 2 of f5 is 2
dim of arg 3 of f5 is 2
dim of arg 4 of f5 is 1
dim of arg 5 of f5 is 2147483647
dim of arg 6 of f5 is 2147483647
dim of arg 7 of f6 is 1
dim of arg 1 of f8 is 2
dim of arg 2 of f8 is 2
dim of arg 3 of f8 is 2
Extent 1 of arg 1 of f1 is 0
Extent 1 of arg 4 of f1 is 0
Extent 1 of arg 1 of f2 is 0
Extent 1 of arg 2 of f4 is 2
Extent 2 of arg 2 of f4 is 3
Extent 1 of arg 3 of f4 is 2147483647
Extent 1 of arg 4 of f4 is 2147483647
Extent 1 of arg 5 of f4 is 2147483647
Extent 1 of arg 2 of f5 is 2
Extent 2 of arg 2 of f5 is 3
Extent 1 of arg 3 of f5 is 2147483647
Extent 1 of arg 4 of f5 is 2147483647
Extent 1 of arg 5 of f5 is 2147483647
Extent 1 of arg 6 of f5 is 2147483647
Extent 1 of arg 7 of f6 is 4
Extent 1 of arg 1 of f8 is 2
Extent 2 of arg 1 of f8 is 3
Extent 1 of arg 2 of f8 is 2147483647
Extent 2 of arg 2 of f8 is 2147483647
Extent 1 of arg 3 of f8 is 2147483647
Extent 2 of arg 3 of f8 is 2147483647
arg 1 of f1 is function 0
arg 4 of f1 is function 0
arg 1 of f7 is function 1
arg 2 of f7 is function 1
arg 3 of f7 is function 1
arg 4 of f7 is function 1
arg 1 of f8 is function 1
arg 2 of f8 is function 1
arg 3 of f8 is function 1
arg 1 of f1 is variable arg 0
arg 4 of f1 is variable arg 0
arg 1 of f7 is variable arg 0
arg 2 of f7 is variable arg 0
arg 3 of f7 is variable arg 0
```

```
arg 4 of f7 is variable arg 1
arg 1 of f8 is variable arg 0
arg 2 of f8 is variable arg 0
arg 3 of f8 is variable arg 0
func arg num for arg 1 of f1 is -1
func arg num for arg 4 of f1 is -1
func arg num for arg 1 of f7 is 1
func arg num for arg 2 of f7 is 0
func arg num for arg 3 of f7 is 3
func arg num for arg 4 of f7 is 1
func arg num for arg 1 of f8 is 1
func arg num for arg 2 of f8 is 1
func arg num for arg 3 of f8 is 1
data size for arg 1 of f1 is -1
data size for arg 4 of f1 is -1
data size for arg 1 of f6 is 16
data size for arg 2 of f6 is 16
data size for arg 3 of f6 is 16
data size for arg 4 of f6 is 16
data size for arg 5 of f6 is 8
data size for arg 6 of f6 is 8
data size for arg 7 of f6 is 16
data size for arg 8 of f6 is 16
data size for arg 9 of f6 is 16
data name for arg 1 of f1 is 0
data name for arg 4 of f1 is 0
data name for arg 1 of f6 is tag
data name for arg 2 of f6 is tag
data name for arg 3 of f6 is tagc
data name for arg 4 of f6 is tagc
data name for arg 5 of f6 is tagu
data name for arg 6 of f6 is tagu
data name for arg 7 of f6 is tag
data name for arg 8 of f6 is tag
data name for arg 9 of f6 is tag
```

See Also

Ch_FuncArgArrayDim(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgFuncArgNum

Synopsis
#include <**ch.h**>
int Ch_FuncArgFuncArgNum(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Obtain the number of the arguments of the function type for an argument of a function in the Ch space.

Return Value
This function returns the number of the arguments of the function type for an argument of a function in the Ch space. If the argument is not function type, it returns -1.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
This function gets the number of the arguments of a function or pointer to function type for an argument of a function in the Ch space. If the function takes a variable number of arguments, it only returns the fixed number of arguments. In this case, function **Ch_FuncArgIsFuncVarArg**() returns 1.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgIsFunc

Synopsis
#include <**ch.h**>
int Ch_FuncArgIsFunc(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Determine if an argument of a function in the Ch space 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.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
The function **Ch_FuncArgIsFunc**() determines if the argument is a function or pointer to function.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayType**(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgIsFuncVarArg

Synopsis
#include <**ch.h**>
int Ch_FuncArgIsFuncVarArg(ChInterp_t *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Determine if an argument of a function in the Ch space is a function or pointer to function with a variable number of arguments.

Return Value
If the argument is a function or pointer to function with a variable number of arguments, this function returns 1. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
The function **Ch_FuncArgIsFuncVarArg**() determines if the argument is a function or pointer to function with a variable number of argument.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgNum

Synopsis
#include <**ch.h**>
int Ch_FuncArgNum(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Obtain the number of the arguments of the function, pointer to function, or member function of class.

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.

name Name of the variable in the Ch space.

Description
This function gets the number of the arguments of a function, pointer to function, or member function of a class. If the function takes a variable number of arguments, it only returns the fixed number of arguments. In this case, function **Ch_IsFuncVarArg**() returns 1.

Example
See **Ch_DataType**().

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_UserDefinedName**(), **Ch_UserDefinedSize**().

Ch_FuncArgUserDefinedName

Synopsis
#include <**ch.h**>
char Ch_FuncArgUserDefinedName(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Obtain the tag name of a user defined data type in terms of struct, class, or union and its pointer type for an argument of function in the Ch space.

Return Value
If the argument of the function in the Ch space is a user defined data type, this function returns the address of the tag name of the user defined data type. Otherwise, it returns NULL.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
This function returns the address for the tag name of struct/class/union or its pointer type for an argument of function in the Ch space. Function **Ch_FuncArgUserDefinedSize**() can be used to obtain the size of the user defined type.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), **Ch_FuncArgIsFunc**(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedSize**().

Ch_FuncArgUserDefinedSize

Synopsis
#include <**ch.h**>
int Ch_FuncArgUserDefinedSize(**ChInterp_t** *interp*, **const char** **funcname*, **int** *argnum*);

Purpose
Obtain the size of a user defined data type in terms of struct, class, or union and its pointer type for an argument of function in the Ch space.

Return Value
If the argument of the function in the Ch space is a user defined data type, this function returns the size of the user defined data type. Otherwise, it returns -1.

Parameters
interp A Ch interpreter.

funcname Name of the function in a global variable.

argnum The argument number in the argument list of the function, 0 for the first argument.

Description
This function returns the size of struct/class/union or its pointer type for an argument of function in the Ch space. Function **Ch_FuncArgUserDefinedName**() can be used to obtain the address of the tag name for the user defined type.

Example
See **Ch_FuncArgDataType**().

See Also
Ch_FuncArgDataType(), **Ch_FuncArgArrayDim**(), **Ch_FuncArgArrayExtent**(), **Ch_FuncArgArrayType**(), ChFuncArgIsFunc(), **Ch_FuncArgIsFuncVarArg**(), **Ch_FuncArgFuncArgNum**(), **Ch_FuncArgUserDefinedName**().

Ch_FuncType

Synopsis
#include <**ch.h**>
ChFuncType_t Ch_FuncType(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Determine if a variable in the Ch space is a function type.

Return Value
If the argument is not a variable of function type, this function returns **CH_NOTFUNCTYPE** with the value of 0. If the argument is a variable of function type, this function returns a non-zero value of **ChFuncType_t**. The data type **ChFuncType_t** defined inside the header file **embedch.h** has the following values.

Value	Description
CH_NOTFUNCTYPE	not function
CH_FUNCTYPE	regular function
CH_FUNCPROTOTYPE	function prototype without function definition
CH_FUNCPTRTYPE	pointer to function
CH_FUNCMEMBERTYPE	function of a class
CH_FUNCCONSTYPE	constructor of a class
CH_FUNCDESTTYPE	destructor of a class

Parameters
interp A Ch interpreter.

name Name of the variable in the Ch space.

Description
The function **Ch_FuncType**() determines if the argument is a variable of function type. The function type includes function with definition, function prototype without function definition, pointer to function, member function of a class, contructor or destructor of a class

Example
See **Ch_DataType**(), Program 7.8.

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_IsFuncVarArg**(), **Ch_FuncArgNum**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**(), **Ch_VarType**().

Ch_GetGlobalUserData

Synopsis
#include <**embedch.h**>
ChPointer_t Ch_GetGlobalUserData(**ChInterp_t** *interp*);

Purpose
Get the global user data for an instance of the Embedded Ch engine.

Return Value
This function returns the global user data of generic data type if it was set by function **Ch_SetGlobalUserData**() previously. Otherwise, it returns NULL.

Parameters
interp A Ch interpreter.

Description
When the Embedded Ch is embedded in a multi-engine environment, this function allows the user to handle different instances. The global user data for an Embedded Ch engine set by function **Ch_SetGlobalUserData**() is obained by this function.

Example
See **Ch_SetGlobalUserData**().

See Also
Ch_SetGlobalUserData().

Ch_GlobalSymbolAddrByIndex

Synopsis
#include <**ch.h**>
void ***Ch_GlobalSymbolAddrByIndex**(**ChInterp_t** *interp*, **int** *num*);

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 the index number is out of the valid range for its symbol table, the function returns NULL. If the symbol 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, Ch computational array, or pointer to Ch computational array, the returned value is the address of the first element. For a variable of pointer to C array, the returned value is the address of the pointer to C array, i.e, the address of the first element. For example, for variables of a and p declared in Ch

```
int (*pa)[3], *p, a[2][3];
```

the returned values for both `pa` and `p` are the same of pointer to pointer to int. The return value for `a` is the address of the first element `a[0][0]`. The data type for variable `a` returned from **Ch_DataType**() is CH_INTTYPE. whereas the data type for variables `pa` and `p` returned from **Ch_DataType**() is the same as CH_INTPTRTYPE. Note that in C, the address of function is the same as the function. For example, the output from the two printing statements below are the same.

```
int func() { /* ... */}
printf("address of func = %p\n", func);
printf("address of func = %p\n", &func);
```

Parameters
interp A Ch interpreter.

num The index number of global variable in the symbol table in a Ch program.

Description
Based on its index number of a global variable in the symbol table, the function **Ch_GlobalSymbolAddrByIndex**() obtains the address of the global variable inside a dynamically loaded library or embedded Ch program. The index number for the symbol table starts with 0.

Example 1
See Program 2.4.

Example 2

In this example, function `func()` in Ch will be called. Just before the function `func()` returns, the callback function `callback()` in the C space will be executed. APIs starting with `Ch_GlobalSymbol`

are used to access variables in the symbol table for global variables.

globalsymbol.c —- A C program to run globalsymbol.ch

```
/*****************************************************************
* File Name: symbol.c
*****************************************************************/
#include<stdio.h>
#include<embedch.h>

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata);

int main() {
    ChInterp_t interp;
    char *argvv[]={"globalsymbol.ch", NULL};
    int mask, count=0;
    ChPointer_t clientdata = NULL;

    Ch_Initialize(&interp, NULL);
    mask = CH_MASKRET;
    Ch_AddCallback(interp,  mask, callback, clientdata, count);
    Ch_RunScript(interp,  argvv);
    Ch_End(interp);
    return 0;
}

void callback(ChInterp_t interp, ChBlock_t *calldata, ChPointer_t clientdata)
{
    int index, totalnum, clevel, hlevel, *p;
    char *name;
    void *addr;
    if(!strcmp(calldata->funcname, "func"))
    {
      Ch_StackLevel(interp, &clevel, &hlevel);
      printf("the current stack level is %d\n", clevel);
      printf("the highest stack level is %d\n", hlevel);
      totalnum = Ch_GlobalSymbolTotalNum(interp);
      printf("The total number of symbols is %d\n", totalnum);
      for(index=0; index<totalnum;index++){
          name = Ch_GlobalSymbolNameByIndex(interp, index);
          addr = Ch_GlobalSymbolAddrByIndex(interp, index);
          printf("symbol index %d, name %s, address %p\n",
                 index, name, addr);
      }
      p = (int *)Ch_GlobalSymbolAddrByName(interp, "i");
      if(p == NULL)
        printf("i is not available\n");
      else
        printf("i = %d\n", *p);

      if(Ch_VarType(interp, "::localvariable") != CH_GLOBALVARTYPE)
         printf("localvariable is not CH_GLOBALVARTYPE\n");
      if(Ch_VarType(interp, "::func") == CH_GLOBALVARTYPE)
        printf("func is CH_GLOBALVARTYPE\n");
    }
}
```

globalsymbol.ch —- A Ch program with global variables and functions

```
int i = 10;
```

```
double d = 1.1;
int prototype();
extern int e;
struct tag {
  int j;
} s;
int func() {
    int localvariable =20;
    printf("printed from globalsymbol.ch, i = %d, d = %f\n", i, d);
    return 0;
}
int main() {
    func();
    return 0;
}
```

Output

```
printed from globalsymbol.ch, i = 10, d = 1.100000
the current stack level is 0
the highest stack level is 2
The total number of symbols is 7
symbol index 0, name i, address 5a000
symbol index 1, name d, address 58380
symbol index 2, name prototype, address 0
symbol index 3, name e, address 0
symbol index 4, name s, address 54028
symbol index 5, name func, address 45e90
symbol index 6, name main, address 51938
i = 10
localvariable is not CH_GLOBALVARTYPE
func is CH_GLOBALVARTYPE
```

See Also

Ch_GlobalSymbolTotalNum(), **Ch_GlobalSymbolIndexByName**(), **Ch_GlobalSymbolAddrByName**(), **Ch_GlobalSymbolNameByIndex**(), **Ch_SymbolAddrByIndex**(), **Ch_VarType**().

Ch_GlobalSymbolIndexByName

Synopsis
#include <**ch.h**>
int Ch_GlobalSymbolIndexByName(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Get the index number of a global variable in the symbol table of a Ch program.

Return Value
The index number of a global variable in the symbol table of a Ch program. If the `name` is not in the list of the symbol table the function returns -1.

Parameters
interp A Ch interpreter.

name The symbol name of a global variable in a Ch program.

Description
The function **Ch_GlobalSymbolIndexByName**() obtains the index number of a global variable in the symbol table in a Ch program inside a dynamically loaded library or embedded Ch program. The index number for the symbol table starts with 0.

Example
See **Ch_GlobalSymbolAddrByIndex**().
See Program 2.4.

See Also
Ch_GlobalSymbolTotalNum(), **Ch_GlobalSymbolAddrByName**(), **Ch_GlobalSymbolAddrByIndex**(), **Ch_GlobalSymbolNameByIndex**(), **Ch_SymbolIndexByName**().

Ch_GlobalSymbolNameByIndex

Synopsis
#include <**ch.h**>
char ***Ch_GlobalSymbolNameByIndex**(**ChInterp_t** *interp*, **int** *num*);

Purpose
Get the symbol name of a global variable in a Ch program.

Return Value
The address for the symbol name of a global variable in a Ch program. If the index number is out of the valid range for its symbol table, the function returns NULL.

Parameters
interp A Ch interpreter.

num The index number of global variable in the symbol table in a Ch program.

Description
Based on its index number of a global variable in the symbol table, the function **Ch_GlobalSymbolNameByIndex**() obtains the address for the symbol name of a global variable inside a dynamically loaded library or embedded Ch program. The index number for the symbol table starts with 0.

Example
See **Ch_GlobalSymbolAddrByIndex**().
See Program 2.4.

See Also
Ch_GlobalSymbolTotalNum(), **Ch_GlobalSymbolIndexByName**(), **Ch_GlobalSymbolAddrByName**(), **Ch_GlobalSymbolAddrByIndex**(), **Ch_SymbolNameByIndex**().

Ch_GlobalSymbolTotalNum

Synopsis
#include <**ch.h**>
int Ch_GlobalSymbolTotalNum(**ChInterp_t** *interp*);

Purpose
Get the total number of global variables in the symbol table in a Ch program.

Return Value
The total number of global variables in the symbol table in a Ch program.

Parameters
interp A Ch interpreter.

Description
The function **Ch_GlobalSymbolTotalNum**() obtains the total number of global variables in the symbol table inside a dynamically loaded library or embedded Ch program. The symbol table contains user defined functions, but no generic functions. It also contains tag names for classes, structures, and unions as well as names defined by **typedef**. The index number for the symbol table starts with 0.

GlobalSymbols in header files including system header files are treated the same as in a user's program. Global variables qualified by **__declspec(global)** are not included in the symbol table.

Example
See **Ch_GlobalSymbolAddrByIndex**().
See Program 2.4.

See Also
Ch_GlobalSymbolIndexByName(), **Ch_GlobalSymbolAddrByName**(), **Ch_GlobalSymbolAddrByIndex**(), **Ch_GlobalSymbolNameByIndex**(), **Ch_SymbolTotalNum**(), **Ch_VarType**().

Ch_InitGlobalVar

Synopsis
#include <**embedch.h**>
int Ch_InitGlobalVar(**ChInterp_t** *interp*, **int** *flag*);

Purpose
Enable or disable initialization of global variables at the parse time.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

flag The flag to enable or disable initialization of global variables at the parse time. If flag is true with the value of 1, it enables initialization of global variables at the parse time. Otherwise, if the flag is 0, it disables initialization of global variables at the parse time, which is the default.

Description
By default, for a global variable declared with initialization, the initialization is performed at the runtime after the entire program is parsed. For a class with constructor, the constructor of a global variable of class type is executed at runtime. In some applications, it may be desirable to change this default behavior by calling this function using

```
Ch_InitGlobalVar(interp, 1);
```

before a program is parsed by functions **Ch_ParseScript**(), **Ch_RunScript**(), **Ch_AppendParseScript**(), and **Ch_AppendRunScript**(). The initialization of a global variable or execution of the constructor of a variable of class type is then performed at the parse time. If the same interpreter is used to handle different scripts, this function can be called to change the default behavior before a script is parsed.

Example

In this example, the script initializes the global variable `g` and declares variable `c` of class with a constructor. Function **Ch_InitGlobalVar**() is called in case 3 to enable the execution of the constructor at the parse time. Therefore, the proper execution sequence is maintained. The constructor is executed first. Then, the member function is executed. Finally, the destructor is executed.

initglobalvar.c —- A C program to run initglobalvar.ch

```
#include <embedch.h>
#include <stdio.h>

int main() {
```

```
    ChInterp_t interp;
    char *argvv[]={"initglobalvar.ch", NULL};

    printf("case 1: Ch_InitGlobalVar(interp, 0)\n");
    Ch_Initialize(&interp, NULL);
    Ch_ParseScript(interp,  argvv);
    Ch_CallFuncByName(interp,"func", NULL);
    Ch_ExprEval(interp,"printf(\"g=%d\\n\", g)");
    Ch_ExecScript(interp,  argvv[0]);
    Ch_End(interp);

    printf("\ncase 2: Ch_InitGlobalVar(interp, 0)\n");
    Ch_Initialize(&interp, NULL);
    Ch_ParseScript(interp,  argvv);
    Ch_ExecScript(interp,  argvv[0]);
    Ch_CallFuncByName(interp,"func", NULL);
    Ch_ExprEval(interp,"printf(\"g=%d\\n\", g)");
    Ch_End(interp);

    printf("\ncase 3: Ch_InitGlobalVar(interp, 1)\n");
    Ch_Initialize(&interp, NULL);
    Ch_InitGlobalVar(interp, 1);
    Ch_ParseScript(interp,  argvv);
    Ch_CallFuncByName(interp,"func", NULL);
    Ch_ExprEval(interp,"printf(\"g=%d\\n\", g)");
    Ch_ExecScript(interp,  argvv[0]);
    Ch_End(interp);
    return 0;
}
```

initglobalvar.ch —- A Ch program with initialization of global variables

```
#include <stdio.h>

class tag {
    public:
      tag();
      ~tag();
      void member1();
      void member2();
};

tag::tag() {
    printf("tag::tag() called\n");
}

tag::~tag() {
    printf("tag::~tag() called\n");
}

void tag::member1() {
    printf("tag::member1() called\n");
}
void tag::member2() {
    printf("tag::member2() called\n");
}

class tag c;
int g = 10;
c.member2();
```

```
void func() {
    c.member1();
}
```

Output

```
case 1: Ch_InitGlobalVar(interp, 0)
tag::member1() called
g=0
tag::tag() called
tag::member2() called
tag::~tag() called

case 2: Ch_InitGlobalVar(interp, 0)
tag::tag() called
tag::member2() called
tag::~tag() called
tag::member1() called
g=10

case 3: Ch_InitGlobalVar(interp, 1)
tag::tag() called
tag::member1() called
g=10
tag::member2() called
tag::~tag() called
```

See Also

Ch_ParseScript(), **Ch_RunScript**(), **Ch_AppendParseScript**(), **Ch_AppendRunScript**().

Ch_Initialize

Synopsis
#include <**embedch.h**>
int Ch_Initialize(**ChInterp_t** **interpp*, **ChOptions_t** **option*);

Purpose
Initialize embedded Ch for executing Ch programs from C space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

option Options for setting embedded Ch. `ChOptions_t` is defined as a struct as the following:

```
typedef struct ChOptions{
  int shelltype;  // shell type
  char *chhome;   // Embedded Ch home directory
} ChOptions_t;
```

Field `shelltype` indicates the type of the Ch shell. It could be **CH_REGULARCH** for a regular Ch shell or **CH_SAFECH** for a safe Ch shell. By default, its value is **ChOptions_t**.

The field `chhome` contains the home directory for the Embedded Ch. In most applications, the Embedded Ch home directory is different from that for a standalone Ch.

In Windows, the dynamically linked library `ch.dll` for for standalone Ch is installed in Windows system directory. To avoid conflict, in general, the dynamically linked library `ch.dll` for Embedded Ch shall be located at the same directory which contains the hosting application.

Description
The function **Ch_Initialize**() initializes embedded Ch for executing Ch programs from the C space. The Ch shell type and the startup file to be used are indicated in the argument *option*. This function should be called before any Ch program is invoked. The embedded Ch initialized will be ended by the API **Ch_End**().

If the argument `option` is NULL, the default value for ChOptions is used. It is assumed the value for the field `sehlltype` is **CH_REGULARCH**, the field value for `chhome` is the home directory CHHOME of the standalone Ch installed in the system. The startup file in CHHOME/config/chrc will be executed. The Embedded Ch in the application searches for the dynamically loaded libraries such as `chmt1.dl` and `chmt2.dl` in the Windows system directory such as `C:/Windows/System32` for Windows and in the directory `/usr/lib` for other platforms.

If the argument `option` is not NULL, the startup file option.chhome/config/chrc will be executed. The Embedded Ch in the application searches for the dynamically loaded libraries such as `chmt1.dl` and `chmt2.dl` in the directory `C:/myApplication/embedch/bin` for Windows and in the directory `/dir/for/myApplication/embedch/extern/lib` for other platforms.

Normally, registered functions in **atexit**() for Embedded Ch programs are executed in **Ch_End**(). When a Ch script is executed by **Ch_ExecScriptM**() or **Ch_RunScriptM**(), functions registered in **atexit**() will be executed when the script is executed. Therefore, a single instance of Ch interpreter initialized by **Ch_Initialize**() can be used to process multiple Ch scripts using **Ch_ExecScriptM**() or **Ch_RunScriptM**().

Example
Refer to Programs 1.1 and 1.14.

See Also
Ch_Home(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_ExecScript**(), **Ch_ExecScriptM**(), **Ch_RunScript**(), **Ch_RunScriptM**(), **Ch_End**().

Ch_IsFuncVarArg

Synopsis
#include <**ch.h**>
int Ch_IsFuncVarArg(ChInterp_t *interp*, **const char** **name*);

Purpose
Determine if a variable in the Ch space is a function or pointer to function.

Return Value
If the argument is a function type with a variable number of arguments, this function returns 1. Otherwise, it returns 0.

Parameters
interp A Ch interpreter.

name Name of the variable in the Ch space.

Description
The function **Ch_IsFuncVarArg**() determines if the argument is a function, pointer to function, or member function of a class with a variable number of argument. The function **Ch_SymbolAddrByName**() can be used to test if a function has a definition or not. For a variable of function prototype without function definition, the function **Ch_SymbolAddrByName**() returns NULL.

Example
See **Ch_DataType**().

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**(), **Ch_VarType**().

Ch_ParseScript

Synopsis
#include <**embedch.h**>
int Ch_ParseScript(**ChInterp_t** *interp*, **const char** **argv*);

Purpose
Parse a Ch program.
Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

argv File name and arguments of the Ch program. The last field must be NULL.

Description
When a program is run in Ch, it first will be parsed, then executed. Unlike function **Ch_RunScript**(), which performs these two steps together. The function **Ch_ParseScript**() parses a Ch program only. Later, the parsed program will be executed through function **Ch_ExecScript**() or **Ch_ExecScriptM**(). The variable *argv* contains the file name and arguments of the Ch program to be executed. It must end with NULL in the last element of the variable. The API **Ch_Initialize**() should be called before this function is called.

Example
Refer to Program 1.4.

See Also
Ch_AppendParseScript(), **Ch_AppendRunScript**(), **Ch_ExecScript**(), **Ch_ExecScriptM**(), **Ch_RunScript**(). **Ch_RunScriptM**(). **Ch_Initialize**(), **Ch_Home**(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_End**().

Ch_Reopen

Synopsis
#include <**embedch.h**>
ChFile_t Ch_Reopen(**ChInterp_t** *interp*, **const char** **filename*, **const char** **mode*, **int** *filedes*);

Purpose
Open a new file descriptor and associate an old one with the new one.

Return Value
This function returns a file descriptor if successful and -1 on failure.

Parameters
interp A Ch interpreter.

filename File name associated with the stream.

mode The mode of the opened file.

fildes The file descriptor of **STDIN_FILENO**, **STDOUT_FILENO**, and **STDERR_FILENO** for the stream to be associated with the `filename`.

Description
The file descriptors **STDIN_FILENO**, **STDOUT_FILENO**, and **STDERR_FILENO** correspond to **stdin, stdout,** and **stderr** streams, respectively. Similar to function **freopen()**, the **Ch_Reopen**() function first attempts to flush the stream associated with the file descriptor `filedes` and then close it. Failure to flush or close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared. The ChReopen() function opens the file whose pathname is the string pointed to by `filename` and associates the stream pointed to by the file descriptor with it. The `mode` argument is used just as in function **fopen**().

Example
Refer to Program 1.12.

See Also
Ch_Flush(), **Ch_Close**().

Ch_RunScript

Synopsis
#include <**embedch.h**>
int Ch_RunScript(**ChInterp_t** *interp*, **const char** ***argv*);

Purpose
Execute a Ch program from C space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

argv File name and arguments of the Ch program. The last field must be NULL.

Description
When a program is run in Ch, it first will be parsed, then executed. **Ch_RunScript**() performs these two steps together.

The function **Ch_RunScript**() executes a Ch program from C space. The variable *argv* contains the file name and arguments of the Ch program to be executed. It must end with NULL in the last element of the variable. The API **Ch_Initialize**() should be called before this function is called. After function **Ch_RunScript**() is called, variables and functions inside the script can be accessed by other APIs in the same interpreter `interp`. However, after function **Ch_RunScriptM**() is called, variables and functions inside the script cannot be accessed by other APIs in the same interpreter `interp`.

Example
Refer to Program 1.1.

See Also
Ch_AppendParseScript(), **Ch_AppendRunScript**(), **Ch_ParseScript**(), **Ch_ExecScript**(). **Ch_ExecScriptM**(). **Ch_RunScriptM**(). **Ch_Initialize**(), **Ch_Home**(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_End**().

Ch_RunScriptM

Synopsis
#include <**embedch.h**>
int Ch_RunScriptM(**ChInterp_t** *interp*, **const char** ***argv*);

Purpose
Execute a Ch program from C space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

argv File name and arguments of the Ch program. The last field must be NULL.

Description
When a program is run in Ch, it first will be parsed, then executed. **Ch_RunScriptM**() performs these two steps together.

The function **Ch_RunScriptM**() executes a Ch program from C space. The variable *argv* contains the file name and arguments of the Ch program to be executed. It must end with NULL in the last element of the variable. The API **Ch_Initialize**() should be called before this function is called. After function **Ch_RunScript**() is called, variables and functions inside the script can be accessed by other APIs in the same interpreter `interp`. However, after function **Ch_RunScriptM**() is called, variables and functions inside the script cannot be accessed by other APIs in the same interpreter `interp`. But, a single instance of Ch interpreter initialized by **Ch_Initialize**() can be used to process multiple Ch scripts using **Ch_ParseScript**() and **Ch_ExecScriptM**(), or **Ch_RunScriptM**().

Example
Refer to Program 1.8.

See Also
Ch_AppendParseScript(), **Ch_AppendRunScript**(), **Ch_ParseScript**(), **Ch_ExecScript**(). **Ch_ExecScriptM**(). **Ch_RunScript**(). **Ch_Initialize**(), **Ch_Home**(), **Ch_SymbolAddrByName**(), **Ch_DataType**(), **Ch_ExprEval**(), **Ch_End**().

Ch_SetGlobalUserData

Synopsis
#include <**embedch.h**>
int Ch_SetGlobalUserData(**ChInterp_t** *interp*, **ChPointer_t** *userdata*);

Purpose
Set the global user data for an instance of the Embedded Ch engine.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

user the user data in a generic data type for an instance of the Embedded Ch.

Description
When the Embedded Ch is embedded in a multi-engine environment, this function allows the user to handle different instances. The global user data can be shared and accessed by Embedded Ch API functions within the same instance of the Ch engine using function **Ch_GetGlobalUserData**(). Using a structure, different data can be accessed by different APIs.

Example 1
The program has two instances of Embedded Ch engine. Each has its own global user data `s1` and `s2`. These data are passed to the function `func()` in the C space.

```
/* File Name: userdata.c */
#include <stdio.h>
#include <embedch.h>

struct tag {int i; double d;};

int func(struct tag *sp, double x) {
    printf("sp->i  = %d, sp->d = %f\n", sp->i, sp->d);
    printf("x in func() = %f\n", x);
    return 0;
}

EXPORTCH int func_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    ChPointer_t userdata;
    struct tag *sp;
    double x;
    int retval;

    Ch_VaStart(interp, ap, varg);
    /* get the user data set by Ch_SetGlobalUserData() */
    userdata = Ch_GetGlobalUserData(interp);
```

```
    sp = (struct tag*)userdata;
    x = Ch_VaArg(interp, ap, double);
    retval = func(sp, x);
    Ch_VaEnd(interp, ap);
    return retval;
}

int main() {
    ChInterp_t interp1, interp2;
    struct tag s1 = {10, 20.0};
    struct tag s2 = {100, 200.0};

    Ch_Initialize(&interp1, NULL);
    /* set the global user data for the 1st Embedded Ch engine interp1 */
    Ch_SetGlobalUserData(interp1, (ChPointer_t)&s1);
    Ch_DeclareFunc(interp1,"int func(double x);", (ChFuncdl_t)func_chdl);
    Ch_ExprEval(interp1, "func(50)");

    Ch_Initialize(&interp2, NULL);
    /* set the global user data for the 2ns Embedded Ch engine interp2 */
    Ch_SetGlobalUserData(interp2, (ChPointer_t)&s2);
    Ch_DeclareFunc(interp2,"int func(double x);", (ChFuncdl_t)func_chdl);
    Ch_ExprEval(interp2, "func(500)");

    /* close interp1 and interp2 */
    Ch_End(interp1);
    Ch_End(interp2);
    return 0;
}
```

Output

```
sp->i  = 10, sp->d = 20.000000
x in func() = 50.000000
sp->i  = 100, sp->d = 200.000000
x in func() = 500.000000
```

Example 2

CHHOME/toolkit/demos/embedch/chapters/appendix/threadswindata.c is an example that illustrates how to share the global data in different Ch scripts running in a multi-thread binary C program in Windows.

Example 3

CHHOME/toolkit/demos/embedch/chapters/chapter1/threadsync.c program starts 4 threads. Each thread has its own Ch interpreter. Functions and variables in each interpreter can be accessed by other threads or the main thread. This multi-thread binary C program can be compiled to run portably across different platforms in Windows and Unix.

See Also

Ch_GetGlobalUserData().

Ch_SetVar

Synopsis
#include <**embedch.h**>
int **Ch_SetVar**(**ChInterp_t** *interp*, **const char** **name*, **ChType_t** *atype*, ...);
int **Ch_SetVar**(**ChInterp_t** *interp*, **const char** **name*, **ChType_t** *atype*, type value);
int **Ch_SetVar**(**ChInterp_t** *interp*, **const char** **name*, **ChType_t** *atype*, const char *tagname, type value);
int **Ch_SetVar**(**ChInterp_t** *interp*, **const char** **name*, **ChType_t** *atype*, **ChType_t** *etype*, type value,
int dim, int extent1, ...);

Syntax
Ch_SetVar(*interp*, *name*, *atype*, *value*);
Ch_SetVar(*interp*, *name*, *CH_INTTYPE*, *value*);
Ch_SetVar(*interp*, *name*, *CH_PROCPTRTYPE*, *NULL*);
Ch_SetVar(*interp*, *name*, *CH_PROCPTRTYPE*, *Ch_SymbolAddrByName(interp, "fp")*);
Ch_SetVar(*interp*, *name*, *CH_STRUCTTYPE*, *tagname*, *structvalue*);
Ch_SetVar(*interp*, *name*, *CH_CLASSTYPE*, *tagname*, *classtvalue*);
Ch_SetVar(*interp*, *name*, *CH_UNIONTYPE*, *tagname*, *unionvalue*);
Ch_SetVar(*interp*, *name*, *CH_CARRAYTYPE*, *etype*, *dim*, *extent1*, ...);
Ch_SetVar(*interp*, *name*, *CH_CHARRAYTYPE*, *etype*, *dim*, *extent1*, ...);

Purpose
Assign a value in the C space to a variable in the Ch space.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

Parameters
interp A Ch interpreter.

name The variable name in the Ch space.

atype The type of the argument to be added. The values for data types **ChType_t** are defined in header file **ch.h**. They are listed in *Ch SDK User's Guide.*

... The argument to be added.

Description
This function is typically used to access values and data in the C space from the Ch space. The memory in the C space can be shared by variables in the Ch space through this function. The variables in the Ch space can be either in a Ch script file or created dynamically at runtime by one of functions **Ch_AppendParseScriptFile**(), **Ch_AppendParseScript**(), **Ch_AppendRunScriptFile**(), **Ch_AppendRunScript**(). The data type and argument for function **Ch_SetVar**() are similar to

those for function **Ch_VarArgsAddArg**() described in detail in *Ch SDK User's Guide*.

Example
In this example, the values and their addresses of variables of different data type in program `setvar.cpp` in the C++ space are shared by variables in Ch script `setvar.ch` in the Ch space. Using **Ch_SetVar**() in the space, the address of function `func()` is assigned to variable `fp` both in the Ch space. Ch function `callback()` is called by function `Ch_CallFuncByName(interp, "callback", &retval)` in the C++ space.

setvar.cpp —- A C++ program to share the data in C++ space with variables in Ch space.

```
/*************************************************************
* File Name: setvar.cpp
*************************************************************/
#include <stdio.h>
#include <embedch.h>

struct tag {
  int i;
  double f;
}s ={10, 20}, s2 ={100, 200},
sa[4]={10, 20, 1,2,3,4,5,6};

class tagc {
  public:
    int i;
    double f;
}c ={10, 20};

union tagu {
    int i;
    double f;
}u;


int main() {
    ChInterp_t interp;
    int status;
    char *argvv[]={"setvar.ch", NULL};
    int retval=0;
    int i = 5;
    double d = 10;
    const char *cp = "string in main()";
    int n=2, m=3;
    int a[2][3] = {1,2,3,
                   4,5,6};
    int b[3] = {1,2,3};

    u.i=10;
    Ch_Initialize(&interp, NULL);
    status = Ch_RunScript(interp,argvv);
    if(status == CH_ERROR) {
      printf("Error: execution of program setvar.ch failed\n");
    }

    Ch_SetVar(interp, "i", CH_INTTYPE, 10);
    Ch_ExprEval(interp, "printf(\"i printed from setvar.c = %d\n\", i)");
```

```
    Ch_SetVar(interp, "dp", CH_DOUBLEPTRTYPE, &d);
    Ch_SetVar(interp, "chp", CH_CHARPTRTYPE, cp);
    Ch_SetVar(interp, "a", CH_CARRAYTYPE, CH_INTTYPE, a, 2, n, m);
    Ch_SetVar(interp, "b", CH_INTPTRTYPE, b);
    Ch_SetVar(interp, "a2", CH_CHARRAYTYPE, CH_INTTYPE, a, 2, n, m);
    Ch_SetVar(interp, "b2", CH_CHARRAYTYPE, CH_INTTYPE, b, 1, m);
    Ch_SetVar(interp, "sp", CH_STRUCTPTRTYPE, &s);
    Ch_SetVar(interp, "s", CH_STRUCTTYPE, "tag", &s);
    Ch_SetVar(interp, "c", CH_CLASSTYPE, "tagc", &c);
    Ch_SetVar(interp, "cp", CH_CLASSPTRTYPE, &c);
    Ch_SetVar(interp, "u", CH_UNIONTYPE, "tagu", &u);
    Ch_SetVar(interp, "up", CH_UNIONPTRTYPE, &u);
    Ch_SetVar(interp, "sa", CH_STRUCTPTRTYPE, sa);
    Ch_SetVar(interp, "fp", CH_PROCPTRTYPE, NULL);
    Ch_SetVar(interp, "fp2", CH_PROCPTRTYPE, Ch_SymbolAddrByName(interp, "func"));
    Ch_SetVar(interp, "stream", CH_FILEPTRTYPE,
             *(FILE**)Ch_SymbolAddrByName(interp, "_stdout"));
    Ch_CallFuncByName(interp, "callback", &retval);
    Ch_End(interp);
}
```

setvar.ch —- A Ch program with variables sharing data in C++ space.

```
#include<array.h>
#include<stdio.h>

struct tag {
  int i;
  double f;
};
class tagc {
  public:
    int i;
    double f;
};
union tagu {
    int i;
    double f;
};

int i;
double *dp;
char *chp;
int (*a)[3], *b;
array int a2[2][3], b2[3];
struct tag s, *sp;
class tagc c, *cp;
union tagu u, *up;
struct tag *sa;
int (*fp)();
int (*fp2)();
FILE *stream;

int func() {
   printf("func() called\n");
   return 0;
}


int callback() {
```

```
   printf("*dp = %f\n", *dp);
   printf("chp = %s\n", chp);
   printf("a[1][1] = %d\n", a[1][1]);
   printf("b[1] = %d\n", b[1]);
   printf("a2 = \n%d", a2);
   printf("b2 = \n%d", b2);
   printf("s.i = %d\n", s.i);
   printf("sp->i = %d\n", sp->i);
   printf("c.i = %d\n", c.i);
   printf("cp->i = %d\n", cp->i);
   printf("u.i = %d\n", u.i);
   printf("up->i = %d\n", up->i);
   printf("sa->i = %d\n", sa->i);
   sa++;
   printf("sa->i = %d\n", sa->i);
   printf("fp = %p\n", fp);
   printf("fp2 = %p\n", fp2);
   fp2();
   fprintf(stream, "callback() called\n");
   return 0;
}
```

Output

```
i printed from setvar.c = 10
*dp = 10.000000
chp = string in main()
a[1][1] = 5
b[1] = 2
a2 =
1 2 3
4 5 6
b2 =
1 2 3
s.i = 10
sp->i = 10
c.i = 10
cp->i = 10
u.i = 10
up->i = 10
sa->i = 10
sa->i = 1
fp = 0
fp2 = 49a40
func() called
callback() called
```

Example 2

Programs 2.5, 2.6, 7.11.

See Also

Ch_AppendParseScriptFile(), **Ch_AppendParseScript**(), **Ch_AppendRunScriptFile**(), **Ch_AppendRunScript**(); and **Ch_VarArgsAddArg**() in *Ch SDK User's Guide*.

Ch_StackLevel

Synopsis
#include <**ch.h**>
int Ch_StackLevel(**ChInterp_t** *interp*, **int** *clevel, **int** *hlevel);

Purpose
Obtain the current and highest possible stack levels.

Return Value
This function returns **CH_OK** on success and **CH_ERROR** on failure.

The function returns the highest possible level of the stack. The stack levels range from 0 to a positive integer number.

Parameters
interp A Ch interpreter.

cleve The current stack level.

hleve The highest possible stack level at the point of execution.

Description
Function **Ch_StackLevel**() obtains the current stack level and highest possible stack level. Level 0 is the currently executed function, whereas level n+1 is the function that has called level n. The program scope outside any function is the top level. For example, when function `level0()` is being called in program `proc.c`, it has level 0. Function `level1()` which calls function `level0()` has level 1. Function `level2()` which calls function `level1()` in turn has level 2. Program `prog.c` has the stack level 3, which is the highest possible level.

```
/* File name: prog.c */
void level2 {
     int i2 = 20;
     level1();
}
void level1 {
     int i1 =10;
     level0();
}
void level0 {
     i0 = 1;
     this_part_of_the_function_level0_is_being_executed.
}
int main() {
     level2();
     return 0;
}
```

As an example, a callback function is invoked when the code inside function `level0()` in the above code is executed. The current stack level 0 and highest possible stack level 3 can be obtained by the code below.

```
int clevel, hlevel;
ChStackLevel(interp, &clevel, &hlevel);
// clevel becomes 0, hlevel becomes 3
```

Example
See **Ch_GlobalSymbolAddrByIndex**().
See Also
Ch_ChangeStack(), **Ch_StackName**().

Ch_StackName

Synopsis
#include <**ch.h**>
char * **Ch_StackName**(**ChInterp_t** *interp*, **int** `level`, **int** *`isfunc`, **char** *`classname`);

Purpose
Obtain the name in a stack.

Return Value
If the function is successful, it returns the name in the stack. If the specified stack `level` is invalid (beyond its range) or the program is executed at the top level, it returns NULL.

Parameters
interp A Ch interpreter.

level The level of the stack.

isfunc It is 1 if the stack is a function; it is 2 if the stack is a member function, contructor, or destructor; else it is 0.

classname It is NULL if the stack is not a member function.

Description
Function **Ch_StackName**() obtains the name in the stack. The second argument `level` specifies the level of the stack. Level 0 is the currently executed function, whereas level n+1 is the function that has called level n. The third argument determines if the stack is a program, function, or member function (including constructor and destructor). The program scope outside any function is the top level. For example, when function `level0()` is being called in program `proc.c`, it has level 0. Function `level1()` which calls function `level0()` has level 1. Function `level2()` which calls function `level1()` in turn has level 2. Program `prog.c` has the stack level 3.

```
/* File name: prog.c */
void level2 {
     int i2 = 20;
     level1();
}
void level1 {
     int i1 =10;
     level0();
}
void level0 {
     i0 = 1;
     this_part_of_the_function_level0_is_being_executed.
}
int main() {
```

```
        level2();
        return 0;
    }
```

As an example, a callback function is invoked when the code inside function `level0()` in the above code is executed. The function name `level1` of the calling function `level1()` and program name `prog.c` can be obtained as follows.

```
char *funcname, *progname;
int level = 1, isfunc;
funcname = ChStackName(interp, level, &isfunc); // isfunc becomes 1
level = 3;
progname = ChStackName(interp, level, &isfunc); // isfunc becomes 0
```

Example
See function `chStack()` in Program 7.16.

See Also
Ch_ChangeStack(), **Ch_StackLevel**().

Ch_SymbolAddrByIndex

Synopsis
#include <**ch.h**>
void ***Ch_SymbolAddrByIndex**(**ChInterp_t** *interp*, **int** *num*);

Purpose
Get the address of a variable in the symbol table for variables and arguments of a function or in the symbol table for global variables in a Ch program.

Return Value
If the control flow of the Ch program is inside a function, the function returns the address of a variable in a function; if the index number is out of the valid range for its symbol table, the function returns NULL. If the control flow of the Ch program is outside a function, the function returns the address of a global variable; if the index number is out of the valid range for its symbol table, the function returns NULL. If the symbol 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, Ch computational array, or pointer to Ch computational array, the returned value is the address of the first element. For a variable of pointer to C array, the returned value is the address of the pointer to C array, i.e, the address of the first element. For example, for variables of a and p declared in Ch

```
int (*pa)[3], *p, a[2][3];
```

the returned values for both `pa` and `p` are the same of pointer to pointer to int. The return value for `a` is the address of the first element `a[0][0]`. The data type for variable `a` returned from **Ch_DataType**() is CH_INTTYPE. whereas the data type for variables `pa` and `p` returned from **Ch_DataType**() is the same as CH_INTPTRTYPE. Note that in C, the address of function is the same as the function. For example, the output from the two printing statements below are the same.

```
int func() { /* ... */}
printf("address of func = %p\n", func);
printf("address of func = %p\n", &func);
```

Parameters
interp A Ch interpreter.

num The index number of global variable in the symbol table in a Ch program.

Description
Based on its index number of a variable in the symbol table, the function **Ch_SymbolAddrByIndex**() obtains the address of the variable inside a dynamically loaded library or embedded Ch program. If the control flow of the Ch program is inside a function, the symbol table contains local variables including function arguments within its scope inside the function. If the control flow of the Ch program is inside a member function of a class, the symbol table contains local variables including members of the class and function arguments within its scope inside the function. If the control flow of the Ch program is outside a function, the symbol table contains global variables. The index

number for the symbol table starts with 0.

Example
See Programs 2.4 and 7.3.

See Also
Ch_SymbolTotalNum(), **Ch_SymbolIndexByName**(), **Ch_SymbolAddrByName**(), **Ch_SymbolNameByIndex**().

Ch_SymbolAddrByName

Synopsis
#include <**ch.h**>
void ***Ch_SymbolAddrByName**(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Get the address of a variable in the symbol table for variables and arguments of a function or in the symbol table for global variables in a Ch program.

Return Value
The address of a 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. variable of C array, Ch computational array, or pointer to Ch computational array, the returned value is the address of the first element. For a variable of pointer to C array, the returned value is the address of the pointer to C array, i.e, the address of the first element. For example, for variables of a and p declared in Ch

```
int (*pa)[3], *p, a[2][3];
```

the returned values for both `pa` and `p` are the same of pointer to pointer to int. The return value for `a` is the address of the first element `a[0][0]`. The data type for variable `a` returned from **Ch_DataType**() is CH_INTTYPE. whereas the data type for variables `pa` and `p` returned from **Ch_DataType**() is the same as CH_INTPTRTYPE. Note that in C, the address of function is the same as the function. For example, the output from the two printing statements below are the same.

```
int func() { /* ... */}
printf("address of func = %p\n", func);
printf("address of func = %p\n", &func);
```

Parameters
interp A Ch interpreter.

name The name of a variable in a Ch program.

Description
The function **Ch_SymbolAddrByName**() obtains the address of a variable within its most inner scope of a Ch program. For example, it will first search for the variable in the block scopes then in the function scope, and finally in the program scope. The variable can be valid data type in Ch. The value of a variable in Ch can be changed in a dynamically loaded object.

Example
See an example in **Ch_DataType**().

See Also
Ch_GlobalSymbolAddrByName(), **Ch_CallFuncByAddr**(), **Ch_CallFuncByName**().

Ch_SymbolIndexByName

Synopsis
#include <**ch.h**>
int Ch_SymbolIndexByName(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Get the index number of a variable in the symbol table for variables and arguments of a function or in the symbol table for global variables in a Ch program.

Return Value
The index number of a variable in the symbol table of a Ch program. If the `name` is not in the list of the symbol table the function returns -1.

Parameters
interp A Ch interpreter.

name The symbol name of a global variable in a Ch program.

Description
The function **Ch_SymbolIndexByName**() obtains the index number of a variable in the symbol table in a Ch program inside a dynamically loaded library or embedded Ch program. If the control flow of the Ch program is inside a function, the symbol table contains local variables including function arguments within its scope inside the function. If the control flow of the Ch program is inside a member function of a class, the symbol table contains local variables including members of the class and function arguments within its scope inside the function. If the control flow of the Ch program is outside a function, the symbol table contains global variables. The index number for the symbol table starts with 0.

Example
See Programs 2.4 and 7.2.

See Also
Ch_SymbolTotalNum(), **Ch_SymbolAddrByName**(), **Ch_SymbolAddrByIndex**(), **Ch_SymbolNameByIndex**().

Ch_SymbolNameByIndex

Synopsis
#include <**ch.h**>
char ***Ch_SymbolNameByIndex**(**ChInterp_t** *interp*, **int** *num*);

Purpose
Get the symbol name of a variable in the symbol table for variables and arguments of a function or in the symbol table for global variables in a Ch program.

Return Value
The address for the symbol name of a variable in a Ch program. If the index number is out of the valid range for its symbol table, the function returns NULL.

Parameters
interp A Ch interpreter.

num The index number of a variable in the symbol table in a Ch program.

Description
Based on its index number of a variable in the symbol table, the function **Ch_SymbolNameByIndex**() obtains the address for the symbol name of aglobal variable inside a dynamically loaded library or embedded Ch program. If the control flow of the Ch program is inside a function, the symbol table contains local variables including function arguments within its scope inside the function. If the control flow of the Ch program is inside a member function of a class, the symbol table contains local variables including members of the class and function arguments within its scope inside the function. If the control flow of the Ch program is outside a function, the symbol table contains global variables. The index number for the symbol table starts with 0.

Example
See Programs 2.4 and 7.3.

See Also
Ch_SymbolTotalNum(), **Ch_SymbolIndexByName**(), **Ch_SymbolAddrByName**(), **Ch_SymbolAddrByIndex**(),

Ch_SymbolTotalNum

Synopsis
#include <**ch.h**>
int Ch_SymbolTotalNum(**ChInterp_t** *interp*);

Purpose
Get the total number of variables in the symbol table for variables and arguments of a function or in the symbol table for global variables in a Ch program.

Return Value
The total number of variables in the symbol table in a Ch program.

Parameters
interp A Ch interpreter.

Description
The function **Ch_SymbolTotalNum**() obtains the total number of variables in the symbol table inside a dynamically loaded library or embedded Ch program. If the control flow of the Ch program is inside a function, the symbol table contains local variables including function arguments within its scope inside the function. If the control flow of the Ch program is inside a member function of a class, the symbol table contains local variables including members of the class and function arguments within its scope inside the function. If the control flow of the Ch program is outside a function, the symbol table contains global variables. The symbol table contains user defined functions, but no generic functions. It also contains tag names for classes, structures, and unions as well as names defined by **typedef**. The index number for the symbol table starts with 0.

Symbols in header files including system header files are treated the same as in a user's program. Global variables qualified by **__declspec(global)** are not included in the symbol table.

Example
See Programs 2.4 and 7.3.

See Also
Ch_SymbolIndexByName(), **Ch_SymbolAddrByName**(), **Ch_SymbolAddrByIndex**(), **Ch_SymbolNameByIndex**(), **Ch_VarType**().

Ch_UserDefinedInfo

Synopsis
#include <**ch.h**>
int Ch_UserDefinedInfo(**ChInterp_t** *interp*, **ChUserDefinedTag_t** *udtag*, **ChUserDefinedInfo_t** **udinfo*);

Purpose
Obtain the information for a user defined structure, class, or union type in the Ch space.

Return Value
If the function is successul, it return **CH_OK**; otherwise, it returns **CH_ERROR**.

Parameters
interp A Ch interpreter.

udtag The tag for a user defined type, a return value from the function **Ch_UserDefinedTag**().

udinfo A pointer to structure containing the information for a user defined type.

Description
This function **Ch_UserDefinedInfo**() can be used to obtain the user defined data type, tag name, size, number of members of a user defined structure, class, or union or its pointer type based on its tag obtained from the function **Ch_UserDefinedTag**().

Type **ChUserDefinedInfo_t** is a structure containing the information for a user defined struct, class, or union type.

```
typedef struct ChUserDefinedInfo_ {
    ChType_t dtype; /* user defined data type: CH_STRUCTTYPE,
                       CH_CLASSTYPE, CH_UNIONTYPE, or CH_UNDEFINETYPE */
    char *tagname;  /* tag name */
    int size;       /* size of class/struct/union */
    int totnum;     /* total number of members in class/struct/union */
}ChUserDefinedInfo_t;
```

Field `dtype` gives the name of the user defined data type. If it is a structure , the value for `dtype` is ChSTRUCTTYPE. If it is a class, the value for `dtype` is ChCLASSTYPE. If it is a union, the value for `dtype` is ChUNIONTYPE. It will be **CH_UNDEFINETYPE**if the input argument `udtag` is NULL.

Field `tagname` gives the tag name of the user defined type. It will be NULL if the input argument `udtag` is NULL.

Field `size` gives the size of the user defined type. It will be 0 if the input argument `udtag` is NULL.

Field `totnum` gives the number of members of the user defined type.
It will be 0 if the input argument `udtag` is NULL.

Example
See **Ch_DataType**(), Programs 2.9 and 7.8.

See Also
Ch_DataType(), **Ch_UserDefinedMemInfoByIndex**(), **Ch_UserDefinedMemInfoByName**(), **Ch_UserDefinedTag**().

Ch_UserDefinedMemInfoByIndex

Synopsis
#include <**ch.h**>
int Ch_UserDefinedMemInfoByIndex(**ChInterp_t** *interp*, **ChUserDefinedTag_t** *udtag*, **int** *index*, **ChMemInfo_t** **meminfo*);

Purpose
Obtain the information for a member of variable of the user defined structure, class, or union type based on its index number in the symbol table for its type.

Return Value
If the function is successul, it return **CH_OK**; otherwise, it returns **CH_ERROR**.

Parameters
interp A Ch interpreter.

udtag The tag for a user defined type, a return value from the function **Ch_UserDefinedTag**().

index The index number of the member in the symbol table for the user defined type.

meminfo A pointer to structure containing the information for a member of a user defined type.

Description
This function **Ch_UserDefinedMemInfoByIndex**() can be used to obtain the information for a member of variable of the user defined structure, class, or union type based on its index number in the symbol table for its type. The function **Ch_UserDefinedMemInfoByName**() can be used to obtain the information for a member based on the member name.

Type **ChMemInfo_t** is a structure containing the information for a member of a user defined structure, class, or union type.

```
typedef struct ChMemInfo_ {
    int index;        /* index number of the member */
    char *memname;    /* member name */
    int offset;       /* offset of the member from the starting memory */
    ChType_t dtype;   /* data type of the member */
    int ispublic;     /* 1: public;         0: private */
    int isfunc;       /* 1: function type;  0: not function type */
    int ismemberfunc; /* 1: member funct;   0: not member funct */
    int isconstructor;/* 1: constructor;    0: not constructor */
    int isdestructor; /* 1: destructor;     0: not destructor */
    int isvararg;     /* 1: funct with variable num arguments; 0: not */
    int arraytype;    /* one of array types */
    int dim;          /* array dim */
    int extent[7];    /* extent for each dimension, up to 7 */
    int isbitfield;   /* 1: bit field;      0: not bit field */
    int fieldsize;    /* struct tag{int i:3, j:10}; fieldsize of j is 10*/
    int fieldoffset;  /* struct tag{int i:3, j:10}; fieldoffset of j is 3*/
    ChUserDefinedTag_t udtag; /* tag for a member of struct/class/union */
} ChMemInfo_t;
```

Field `index` gives the index number of the member in the symbol table for the user defined type. The index number for the first member is 0.

Field `memname` gives the name of the member.

Field `offset` gives the offset of the address for the member in the memory from the beginning of the variable of the user defined type.

Field `dtype` gives the data type of the member.

Field `ispublic` is 1 if the member is public. If the member of class is private, it is 0.

Field `isfunc` is 1 if the member is function type including function, pointer to function, and member function. Othewise, it is 0.

Field `ismemberfunct` is 1 if the member is a member function, constructor, or destructor of a class. Otherwise, it is 0. If the member if a pointer to function, it is 0.

Field `isconstructor` is 1 if the member is the constructor of a class. Otherwise, it is 0.

Field `isdestructor` is 1 if the member is the destructor of a class. Otherwise, it is 0.

Field `isvararg` is 1 if the member is a function type and its argument contains a variable number of arguments. For example, for members `func1` and `func2`, the value for `isvararg` is 1.

```
class tag {
    int func1(int i, ...);
    int (*func2)(int j, ...);
};
```

Field `arraytype` is not 0 if the member is an array type. Similar to the return value of functions **Ch_ArrayType**() and **Ch_FuncArgArrayType**(). It contains one of the following macros defined in header file **ch.h**.

Macro	Description	Example
CH_UNDEFINETYPE	not an array.	`int i`
CH_CARRAYTYPE	C array	`int a[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[&])`

A poiner to C array is not considered as an array for this member arraytype.

Field `dim` gives the dimension of array if the member is a C array or Ch computational array.

Field `extent` contains the extent for each dimension of array if the member is a C array or Ch computational array. Only arrays of dimension up to 7 can be handled. For example, for member `a` in the structure `tag`

```
struct tag {
    int a[3][4];
    ...
};
```

the value for `dim` is 2. `extent[0]` is 3 and `extent[1]` is 4.

Fields `isbitfield, fieldsize,` and `fieldoffset` are for handling members of bit fields. If the member is a bit field, isbitfield is 1. Otherwise, it is 0. Fields `fieldsize` and `fieldoffset` give the field size and offset of the field, respectively. For example, for member `j`

```
struct tag {
    int i:3;
    int k:3;
    int j:10;
    ...
};
```

`fieldsize` is 10 and `fieldoffset` is 6.

Field `udtag` gives the tag of a user defined structure, class, or union type for the member if the member is a user defined type of **CH_STRUCTTYPE**, **CH_CLASSTYPE** , or **CH_UNIONTYPE**. For example, for members `s` and `sp` in the structure `tag1`

```
struct tag1 {
    struct tag2 s;
    struct tag3 *sp;
};
```

the values for `udtag` gives the tag for structures `tag2` and structures `tag3`, respectively. This field can be used to obtain the information about the user defined type and its members.

If the input argument `udtag` is NULL or `memname` is not a member name of the user defined type, the function returns **CH_ERROR**, the member `dtype` of the input argument `meminfo` is assigned with the value of **CH_UNDEFINETYPE**. This value can alos be used to test if the `memname` is a valid member name or the function call is successful.

Example
See **Ch_DataType**(), Programs 2.9 and 7.8.

See Also
Ch_UserDefinedInfo(), **Ch_UserDefinedMemInfoByName**(), **Ch_UserDefinedTag**(), **Ch_DataType**(), **Ch_ArrayType**(), **Ch_FuncArgArrayType**().

Ch_UserDefinedMemInfoByName

Synopsis
#include <**ch.h**>
int Ch_UserDefinedMemInfoByName(**ChInterp_t** *interp*, **ChUserDefinedTag_t** *udtag*, **const char** **memname*, **ChMemInfo_t** **meminfo*);

Purpose
Obtain the information for a member of variable of the user defined structure, class, or union type based on the name of the member.

Return Value
If the function is successul, it return **CH_OK**; otherwise, it returns **CH_ERROR**.

Parameters
interp A Ch interpreter.

udtag The tag for a user defined type, a return value from the function **Ch_UserDefinedTag**().

memname The name of the member.

meminfo A pointer to structure containing the information for a member of a user defined type.

Description
This function **Ch_UserDefinedMemInfoByName**() can be used to obtain the information for a member of variable of the user defined structure, class, or union type based on its name of the member. The function **Ch_UserDefinedMemInfoByIndex**() can be used to obtain the information for a member based on the index number of the member in the symbol table for its type.

If the input argument `udtag` is NULL or `index` is not a valid index of the user defined type, the function returns **CH_ERROR**and the member `dtype` of the input argument `meminfo` is assigned with the value of **CH_UNDEFINETYPE**. This value can alos be used to test if the `index` is a valid index or the function call is successful.

Example
See **Ch_DataType**().
See Program 2.9.

See Also
Ch_DataType(), **Ch_UserDefinedInfo**(), **Ch_UserDefinedMemInfoByIndex**(), **Ch_UserDefinedTag**().

Ch_UserDefinedTag

Synopsis
#include <**ch.h**>
ChUserDefinedTag_t Ch_UserDefinedTag(ChInterp_t *interp*, **const char** **name*);

Purpose
Obtain the tag for a variable of user defined structure, class, or union type in the Ch space.

Return Value
If the argument of the variable in the Ch space is a user defined structure, class, or union type, this function returns the tag for the user defined data type. Otherwise, it it returns NULL.
For a variable for a tag name, array or pointer type in the Ch space, the function also returns its tag.
Parameters
interp A Ch interpreter.

name Name or tag name for a variable or expression of user defined type in the Ch space.

Description
This function **Ch_UserDefinedTag**() returns the tag of a user defined structure, class, or union or its pointer type for a variable in the Ch space. Function **Ch_UserDefinedInfo**() can be used to obtain the tag name, size, number of members of the user defined type.

Example
See **Ch_DataType**(), Programs 2.9 and 7.8.

See Also
Ch_DataType(), **Ch_UserDefinedInfo**(), **Ch_UserDefinedMemInfoByIndex**(), **Ch_UserDefinedMemInfoByName**().

Ch_VarType

Synopsis
#include <**ch.h**>
ChVarType_t Ch_VarType(**ChInterp_t** *interp*, **const char** **name*);

Purpose
Determine if a symbol is a variable in the Ch space.

Return Value
If the argument is not a variable, this function returns **CH_NOTVARTYPE** with the value of 0. If the argument is a variable, this function returns a non-zero value of **ChVarType_t**. The data type **ChVarType_t** defined inside the header file **embedch.h** has the following values.

Value	Description
CH_NOTVARTYPE	not a variable.
CH_GLOBALVARTYPE	a global variable.
CH_LOCALVARTYPE	a local variable.

Parameters
interp A Ch interpreter.

name Name of the variable in the Ch space.

Description
Function **Ch_SymbolTotalNum**() count all symbols including variables, tag names for classes, structures, and unions as well as names defined by **typedef**. The function **Ch_VarType**() determines if a symbol is a variable, global or local variable. For symbols `tag`, `tage`, `red`, `type_t` in

```
struct tag{int i;};
enum tag2{red, blue, green};
typedef int type_t;
```

function **Ch_VarType**() returns **CH_NOTVARTYPE**which is defined as 0.

If the function **Ch_VarType**() is called when a Ch program runs in a function or block scope, the argument `name` is a local variable, it returns **CH_LOCALVARTYPE**. If the argument `name` is a global variable, it returns **CH_GLOBALVARTYPE**. To check a variable of the global scope when a Ch program is running in a function or block, the scope resolution operator `::` can be used to indicate a variable of global scope by `Ch_VarType(interp, "::name")`.

Function **Ch_VarType**() can be used to test if a function is a nested local function.

Example
See Program 2.3, Program 7.8, **Ch_GlobalSymbolAddrByIndex**().

See Also
Ch_DataType(), **Ch_ArrayDim**(), **Ch_ArrayExtent**(), **Ch_ArrayType**(), **Ch_FuncType**(), **Ch_FuncArgNum**(), **Ch_IsFuncVarArg**(), **Ch_UserDefinedTag**(), **Ch_UserDefinedInfo**().

Appendix B

Porting Code with Embedded Ch APIs to the Latest Version

B.1 Porting Code with Embedded Ch APIs to Ch Version 6.0

1. The default home directory of Embedded Ch has been changed. For the function call,

   ```
   Ch_Initialize(&interp, NULL);
   ```

   the default home directory for Embedded Ch is CHHOME/toolkit/embedch, instead of CHHOME. The dynamically loaded library for an instance of the interpreter is CHHOME/toolkit/embedch/extern/lib/chmt1.dl, etc, instead of using /usr/lib/chmt1.dl, etc in Unix and C:/Windows/System32/chmt1.dl, etc in Windows.

2. To load a dynamically loaded library `libsample.dl` when the program is parsed, change

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

   to

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

   in your Ch script code.

3. Changed the handling of **Ch_SymbolAddrByName**(`interp,"a"`) for array 'int a[2][3];' in Ch space to be consistent with handling of C array. Change

   ```
   int *pa;
   pa = *(int **)Ch_SymbolAddrByName(interp,"a");
   ```

   to

   ```
   int *pa;
   pa = (int *)Ch_SymbolAddrByName(interp,"a");
   ```

4. Functions **Ch_UserDefinedName**() and **Ch_UserDefinedSize**() are deprecated. Use **Ch_UserDefinedInfo**(). Change

```
int size;
char *name;

size = Ch_UserDefinedSize(interp, "s");
name = Ch_UserDefinedName(interp, "s");
```

to

```
int size;
char *name;
ChUserDefinedTag_t udtag;
ChUserDefinedInfo_t udinfo;

udtag = Ch_UserDefinedTag(interp, "s");
Ch_UserDefinedInfo(interp, udtag, &udinfo);
size = udinfo.size;
name = udinfo.tagname;
```

5. Function **Ch_SymbolIndex**() is deprecated. Use **Ch_SymbolIndexByName**(). Change

```
index = Ch_SymbolIndex(interp, name);
```

to

```
index = Ch_SymbolIndexByName(interp, name);
```

6. Function **Ch_IsFunc**() is deprecated. Use **Ch_FuncType**(). Change

```
if(Ch_IsFunc(interp, name)) {
  ...
}
```

to

```
if(Ch_FuncType(interp, name)) {
  ...
}
```

7. Function **Ch_IsVariable**() is deprecated. Use **Ch_VarType**(). Change

```
if(Ch_IsVariable(interp, name)) {
  ...
}
```

to

```
if(Ch_VarType(interp, name)) {
  ...
}
```

8. The deprecated members `chrc, chrcname, chmtdir` of struct `ChOptoin_t` have been removed. See section 1.7 on how to distribute applications with Embedded Ch.

9. The location for the distribution of dynamically loaded lib chmt#.dl in Windows has been changed from EMBEDCH_HOME/bin/chmt#.dl to EMBEDCH_HOME/extern/lib/chmt#.dl so that they are located in the same location for all platforms. For multiple instances of Embedded Ch engine, copy EMBEDCH_HOME/extern/lib/chmt1.dl to EMBEDCH_HOME/extern/lib/chmt2.dl, etc.

B.2 Porting Code with Embedded Ch APIs to Ch Version 5.5

To simplify the distribution of Embedded Ch, the member fields `chrc, chrcname, chmtdir` of struct `ChOptoin_t` are deprecated.

if the user specifies the home directory of the Embedded Ch in the application, The file option.chhome/config/chrc is used as the system startup file. The directory option.chhome/bin in Windows and option.chhome/extern/lib in other platforms shall contain dynamically loaded libs chmt1.dl, chmt2.dl, etc.

If the user does not specify the home directory of Embedded Ch, the startup file CHHOME/config/chrc is used. the dynamically loaded libs chmt1.dl, chmt2.dl, etc. are located in in C:/Windows/System32 for Windows and /usr/lib for Unix.

```
/* we assume that you distribute CHHOME/toolkit/embedch along with
   your application. It is distributed in your application home sub
   directory such as c:\my_applications\embedch or
   /usr/my_applications/embedch */
#include <embedch.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
  int status, len;
  ChInterp_t interp;
  ChOptions_t option;
  char *proghome;

  /* get the home dir for your application program from
     an environment variable or registry value (PROG_HOME)
     It can be directory either c:\my_applications\ or
     /usr/my_applications
     */
  proghome = getenv("PROG_HOME");

  /* initialize embedded Ch */
  option.shelltype = CH_REGULARCH;

  len = strlen(proghome)+ strlen("/embedch")+1;
  option.chhome = (char *)malloc(len);
  strcpy(option.chhome, proghome);
  strcat(option.chhome, "/embedch");
```

```
  Ch_Initialize(&interp, &option);
  /* ... */
}
```

Index