7 Interfaces Between Perl and ANSI C, Visual C++

7.1 Overview

Practical Extraction and Report Language (Perl) is a relatively new language; it was developed in the late 1980s. It is UNIX-based software, and like JavaScript, Perl is also a Script language. It can be embedded in HTML and accessed on the Internet by clients.

For many years, UNIX system administrators have used Perl to handle some management processing tasks. Perl became popular as a Web language because of its ability to create CGI Scripts (Common Gateway Interface), which were originally developed for UNIX servers.

The current version of Perl is Perl 5, and the UNIX and Windows version is 5.6.1. In this chapter, we will discuss the interface between Perl and C/C++ in the Windows environment. We will use the Visual C++ 6.0 compiler. The version of the Perl is 5.6.1 for Windows. All Perl software is free, and users can download Perl's tools and packages from the Internet. The advantages of using Perl can be summarized as follows:

- powerful
- portable

- flexible
- easy to learn

In this chapter, we concentrate on the interface between Perl and C/C++. Some other interfaces are also available, such as the interface between Visual Basic and Perl, and between Java and Perl. For now, however, we only want to focus on the first kind of interface we mentioned.

In some applications, an interface makes things easier and faster. For example, say a server developed in C or C++ needs to talk to some programs developed in Perl, and some clients want to make complicated data calculations or analyses on the Web. The interface between Perl and Matlab is a good solution for that situation.

Basically, when Perl was developed, it was based on C, and libraries are compiled to executable files. It looks like communication should be easy between Perl and C because they have the same parents. That is untrue. The differences between these two languages are significant, and some special tools are needed to complete this interface. For example, you need to use XSUBPP to compile and convert Perl XS code to C code. The standard Perl code, or Perl source code is extended with `.pl`.

In Perl programming, the interface between Perl and other languages is also called an extension. The interface between Perl and C is called an extension of Perl to C. A very popular interface language used in Perl programming is XS. The XS language is an interface description file used to create an extension interface between Perl and C or C++ code. The XS interface is combined with some libraries to create a new library that can be either dynamically or statically linked to Perl. The XS interface description is written in the XS language and is the core component of the Perl extension interface.

An XSUB forms a basic unit for the XS interface, and it represents a direct translation between C/C++ and Perl so that it preserves the interface, even from Perl. After the interface XSUB is compiled by using the `xsubpp` compiler, it will provide a direct mapping between the Perl and C code, and glues both together.

The interfaces between Perl and C/C++ can be divided into two categories. One is to call the C program from within the Perl code, and the other is to call Perl from the C/C++ code. These two types of interface are shown in Figure 7-1. The top situation calls a C/C++ code from within the Perl program. Thus, we need to use the XS interface to develop an extension between Perl and C/C++ . In the second situation, the interface is created in an inverted way, by calling Perl code from a C/C++ program. An embedded Perl interpreter is used here to make this call successful. The working principles of these two styles will be discussed next.

First, let's take a look at the top style, calling to C/C++ from the Perl code. To do this interface, you need to use a tool provided by Perl, `h2xs`, which is used to create

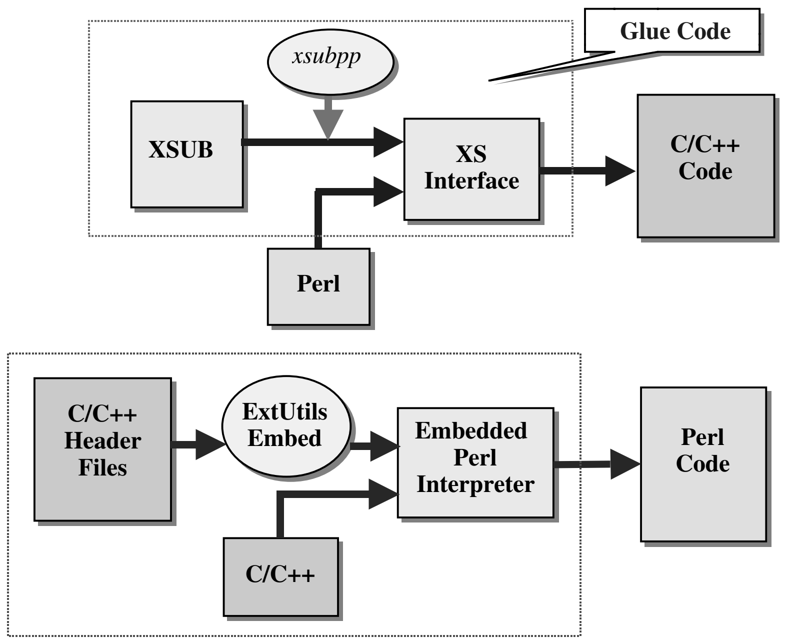


Figure 7-1

an extension between the two by converting the C code to a Perl extension. We use an example extension interface, `PerlTest`, to illustrate this process.

- Use `h2xs` to create a set of files for an extension interface (`h2xs –A –n PerlTest)`.
  By using the `h2xs` tool, a directory of `PerlTest` and the following files will be created for this extension interface:
  - `MANIFEST`
  - `Makefile.PL`
  - `PerlTest.pm`
  - `PerlTest.xs`
  - `test.pl`
  - `Changes`

  The `MANIFEST` file contains the names of all the files created in this list. The `Makefile.PL` file is a skeleton of the make file for this extension interface. After you modify the `PerlTest.xs` file and add some C/C++ codes, you need to execute `perl Makefile.PL` to create a real make file for this extension interface.
  The `PerlTest.pm` file provides all necessary libraries and environment to tell Perl how to load your extension. The `PerlTest.xs` file holds the C/C++ routines that make up the extension. You should add any additional C/C++ code,

such as your implementation code, at the end of this file. This is the only place to put your C/C++ code.

The `test.pl` is a Perl file, used to test each source code in the Perl domain. The `Changes` file provides a record for any changes or modifications made to your extension files.

- Modify the `PerlTest.xs` file to add the C/C++ code.
  As we mentioned before, you should add all of your C/C++ code in this `.xs` file because this is the only place you can put the C/C++ code.
- Call the `xsubpp` compiler to create an extension interface.
  After you have finished modifying the `PerlTest.xs` file, you can execute `perl Makefile.PL` to create a real make file for this extension interface. Next, you need to call a make program to build your extension interface. By doing this, you called the `xsubpp` compiler to convert the `PerlTest.xs` interface to the C/C++ code.
- Call the C/C++ code by using that extension interface.
  Finally, you can call the C code you added in the file, `PerlTest.xs`, from the Perl program to test your extension interface.

Now let's take a look at the bottom style from Figure 7-1—embedding a Perl interpreter into the C/C++ program. In this interface design, we embed a Perl interpreter into the C/C++ domain by using the `ExtUtils::Embed` module provided by Perl. By calling and executing the embedded Perl interpreter from within the C/C++ domain, we can easily access the code in the Perl domain. To interface and convert variables or components in two different domains (Perl and C/C++), you need to use the Perl stack area as temporary memory. Use the `push` command to store variables or components into the Perl stack, and use the `pop` command to retrieve variables or components returned from the Perl stack.

Because of space limitations, we will concentrate on the second style interface, embedding the Perl interpreter into the C/C++ domain. Three examples are provided for this kind of interface. We will give a detailed description and illustration based on these three examples step by step in Section 7.3. But first we will discuss how to install Perl on your machine.

7.2 Installation of Perl

To install Perl 5.6.1 for Windows, we need to download the Win32 version of Perl. This version is good for Windows 95/98/Me, and Windows NT and 2000/XP. During the installation of Win32 Perl, you have two choices. One is to install the Perl source files and then compile and build them on your machine based on the requirements of your machine. The other option is to install prebuilt binary code, which is easier for the beginner, because all libraries, scripts, Perl modules, PerlScripts, and CGI scripts have been integrated. In this book, we will use the second method to install Perl. The Perl we will use is ActivePerl, which is a product of ActiveState, Inc.

7.2.1 Download ActivePerl for Windows

Go to the Web site *www.activestate.com/ActivePerl/download.html*, which provides the installation requirements. The following configurations are required for the different operating systems:

- Windows NT: Need Service Pack 5+ and Windows Installer 1.1+.
- Windows 95/98/Me: Need Windows Installer 2.0+.
- Windows 95: Need DCOM for Windows 95.
- Windows 2000 & XP: No additional requirement.

As an example, for a Windows 95, 98, or Me user, click `Windows Installer 2.0+`, and an installation dialog box is displayed. Select the `Save the file` radio button and download the file `InstMsiA.exe` to a default folder (such as the `Temp` folder on your computer) by clicking `Save`.

7.2.2 Install Win32 Perl on Your Machine

When the download is completed, open the default folder that you downloaded Perl to, and double-click `InstMsiA.exe` to install ActivePerl on your computer. After the installation is finished, a new folder, `Perl`, has been created and all of the Perl files, libraries, and modules are located in that folder. Also, for the binary code installation, the system will automatically locate your C/C++ compiler and linker, and assign them to the Perl system as the default compiler and linker. The default library path is `C:\Perl\lib\CORE`. The `CORE` folder contains all core files of Perl as well as the system libraries.

After the installation, the following subfolders are added to the `C:\Perl` directory:

- `bin`: This is a system directory that contains all executable files for Perl and tools.
- `eg`: All examples are located at this subdirectory.
- `html`: All HTML files are located at this folder, including the HTML installation file, resource files, and release file in HTML format.
- `lib`: All system library files, including core libraries, are located at this directory.
- `site`: Additional library files are provided by ActivePerl.

You have to restart your computer to make the installation valid. For embedding the Perl interpreter into the C/C++ domain, the most often used files are `perl.h` (header file) and `perl56.lib` (library file).

```
# echo the input to the screen
printf("Enter an Integer Number \n");

$Number = <STDIN>;

printf("The Number You Entered is: %d\n", $Number);
print "See You…\n";
```

Figure 7-2

The system folder, `Perl\bin`, has been automatically added to your system path by the installer during the installation. So all system executable files are searchable on your computer. You can run the following command to test the installation on a command prompt window:

```
C:\>perl −w −e "print\"Perl is working!\n\";"
```

Press the `Enter` key and the message `Perl is working`! will be displayed on the screen. This command has four sections.

- `perl`: by calling this `perl` command, you call `perl.exe` to run your script.
- `−w`: this option is to turn on the Perl warning mechanism.
- `−e`: this option tells `perl.exe` that this script has only one line, which is also called one-liner script.
- Following `−e` is the Perl script code.

You can also generate a slightly complicated Perl script in a text editor, such as NotePad, and then run it in the command prompt window. For example, you can enter the code shown in Figure 7-2 into a NotePad editor.

Save this file as `"Number.pl"` on the root directory. Don't forget to use the double quotation marks to enclose this file name in the NotePad editor when you save this file, otherwise this file will be saved as a text file format.

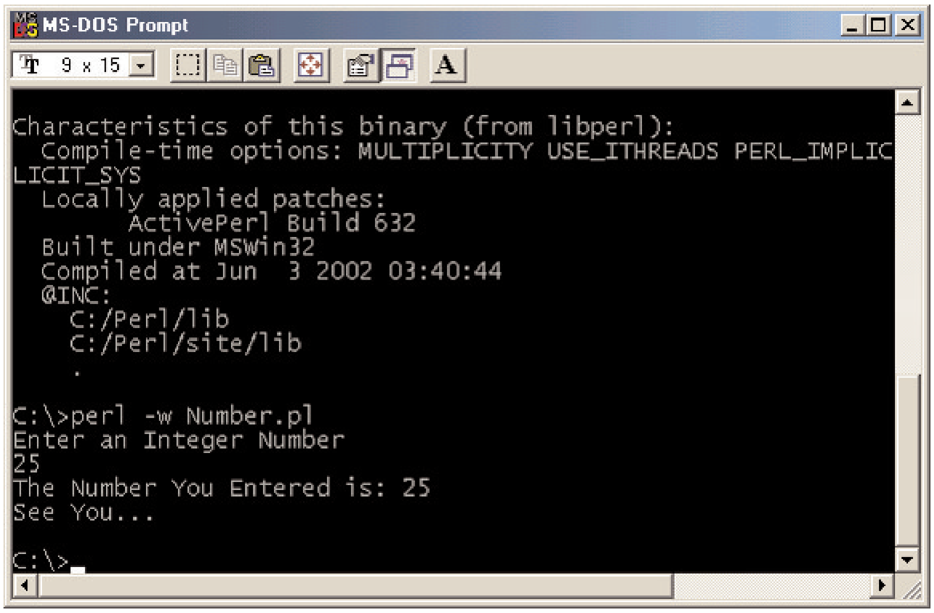
Open a command prompt window and enter the following code to run this script:

```
perl −w Number.pl
```

After you press the `Enter` key, the prompt displayed on the screen will ask you to enter an integer. Enter `25`, then press the `Enter` key again. Your Perl script will be executed, and the running result is shown in Figure 7-3.

You can use the command `perl −V` to list all options for the Perl command as well as version information.

07-P2521 2/11/03 10:09 AM Page 783



```
Characteristics of this binary (from libperl):
  Compile-time options: MULTIPLICITY USE_ITHREADS PERL_IMPLIC
LICIT_SYS
  Locally applied patches:
        ActivePerl Build 632
  Built under MSWin32
  Compiled at Jun  3 2002 03:40:44
  @INC:
    C:/Perl/lib
    C:/Perl/site/lib
    .

C:\>perl -w Number.pl
Enter an Integer Number
25
The Number You Entered is: 25
See You...

C:\>_
```

Figure 7-3

7.3 Interface Design — Embed Perl Interpreter into C/C++

7.3.1 Overview

In this section, we will discuss how to develop an interface between Perl and C/C++ by using a method of embedding a Perl interpreter into the C/C++ domain. To achieve this objective, we will utilize Visual C++ 6.0 as a compiler and a linker, and use the `ExtUtils::Embed` module provided by Perl to finish embedding the Perl interpreter into the Visual C++ 6.0 domain. We will develop some code in the Visual C++ 6.0 domain, and try to embed the Perl interpreter in to that domain, to call and run a Perl script to test this interface.

The block diagram of the functionality of this interface is shown in Figure 7-4. C/C++ code accesses the Perl script by calling and running an embedded Perl interpreter, which is embedded by using the `ExtUtils::Embed` module provided by Perl. Also inside the C/C++ domain, we need to perform some necessary initialization and cleanup jobs for the calling and running of the Perl interpreter. We also need to handle the data and components conversion between two different domains. All variables and components passed from the C/C++ domain are stored in the Perl stack area, which is similar to the popular stack with a property of Last-In-First-Out (LIFO).

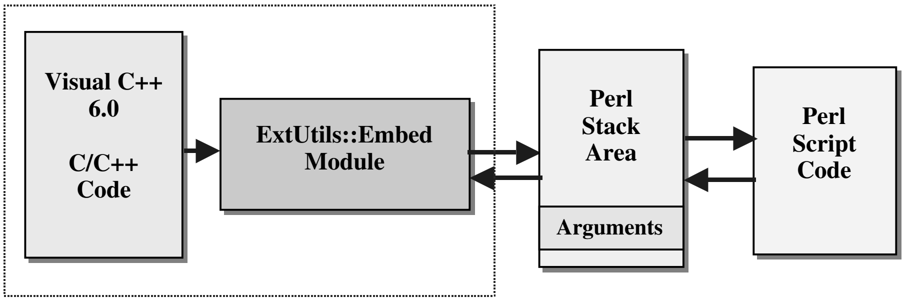


Figure 7-4

Some special functions are needed to handle these jobs. The following is a sequence for developing and calling the embedded Perl interpreter:

- Generate a new Visual C++ project.
- Add a new source file to this project; this new source file will call the embedded interpreter to interface to Perl scripts. The source file should include the following functionalities:
  - Allocate the memory spaces in the Perl stack for the `PerlInterpreter` object, which is defined in the Perl system library, and for variables and components that will be passed to the Perl domain.
  - Call a special function, `perl_parse()`, to pass and copy all components to the Perl stack area so that they are ready to run.
  - Call a special function, `perl_run()`, to run the Perl script.
  - De-allocate the memory spaces for the `PerlInterpreter` object, and all other variables as well as components.
- Create a Perl script with some subroutines that will be called by the C/C++ program.
- Test the interface by running the C/C++ code, which will call the embedded interpreter to interface to the Perl scripts.

Next, we will combine three real examples to illustrate the development and building process of this kind of interface.

7.3.2 A Simple Interface Between C and Perl

In our first example, we will not create any Perl scripts. We will only embed a Perl interpreter into our Visual C++ domain. By running this Visual C++ project, we call an embedded Perl interpreter and release controllability of the project to that Perl interpreter. The user can type any Perl codes and run them in the Visual C++ domain, but

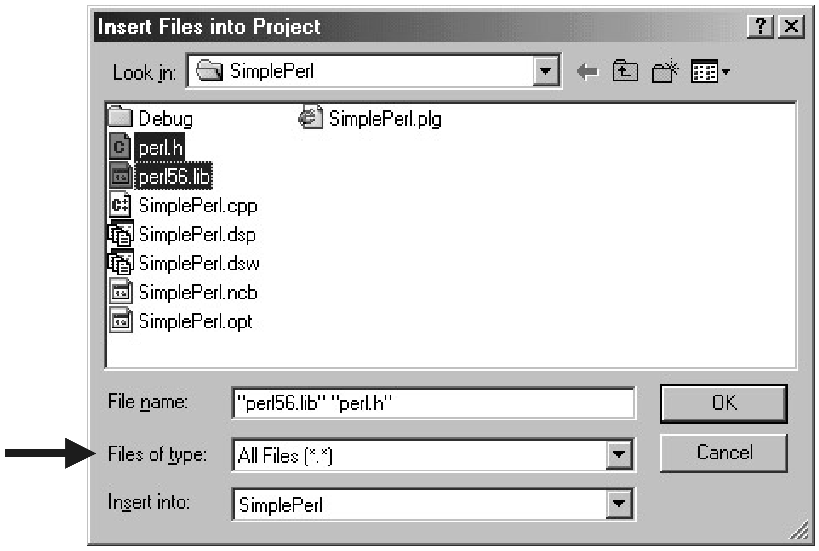


Figure 7-5

the results of running the program show that the process has indeed taken place in the Perl domain because the results are identical to what happens when you run Perl script in the Perl domain. Sound interesting? Let's do it right now.

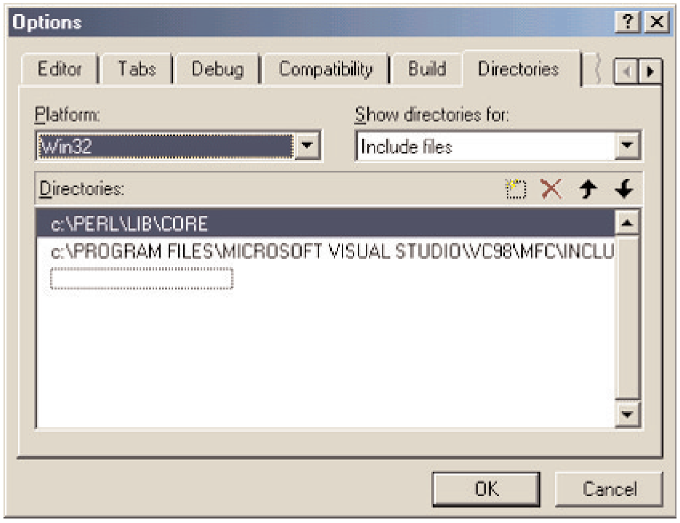
First, open Windows Explorer to create a user-defined project folder. In our case, we name our user-defined project folder `C:\PerlApp`. Launch Visual C++ 6.0 and open a new project with a type of `Win32 Console Application`. Enter `SimplePerl` in the `Project name`: input field as this project's name. Make sure the `Location`: field is `C:\PerlApp`. Click `OK` to open this new project.

Click `File|New` to create a new C++ source file named `SimplePerl.cpp`. Click `OK` to open this new project. Open Windows Explorer and copy the following files from `C:\Perl\lib\CORE` to our new project folder, `C:\PerlApp\SimplePerl`:

- `perl.h`
- `perl56.lib`

Return to the Visual C++ 6.0 workspace, and click the `Project|Add To Project|Files. . .` menu item to open the `Insert Files into Project` dialog box, which is shown in Figure 7-5.

Make sure that you select `All Files (*.*)` from the `Files of type`: list box, as the arrow in Figure 7-5 indicates. Select the files we just copied from the Perl system folder `C:\Perl\lib\CORE`, `perl.h` and `perl56.lib`, then click `OK` to add these two files into our current project. Now click the `Tools|Options` menu item to open the `Options` dialog box, which is shown in Figure 7-6.



Platform: Win32

Show directories for: Include files

Directories:

c:\PERL\LIB\CORE
c:\PROGRAM FILES\MICROSOFT VISUAL STUDIO\VC98\MFC\INCLU

Figure 7-6

Click the `Directories` tab and select `Include files` from the `Show directories for`: list box. In the `Directories`: list, double-click the dashed-line box to display a button on the right, then click that button to browse the directories and locate the folder `C:\Perl\lib\CORE`. Click `OK` to select this folder as the additional directory for our project. In the same manner, also select the folder `C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE` as an additional directory for our project.

Now, return to the Visual C++ 6.0 workspace and open the newly created source file `SimplePerl.cpp`. Enter the code shown in Figure 7-7 into this file.

A. Two system header files are declared at the beginning of this source file, `EXTERN.h` and `perl.h`. Both are provided by Perl and located at `C:\Perl\lib\CORE` system directory. The prototypes of the `ExtUtils::Embed` module are defined in the first header file. All Perl system files, including library files, script files, and other modules are defined in the second header file.

B. Declare the `PerlInterpreter` object `*my_perl`, which is defined in the `perl.h` header file. This `PerlInterpreter` is a structure defined in the Perl domain and it contains all necessary information of the Perl interpreter. Because this object's name has been defined in the `perl.h` header file, you cannot modify this object with any other name.

C. Declare the `main()` entry function in the C/C++ domain.

D. Call the `perl_alloc()` system function to allocate memory space for the `PerlInterpreter` object in the Perl stack area.

```
  /*****************************************************************************
   * NAME        : SimplePerl.cpp
   * DESC        : First test interfacing program for Perl. Create a perl env in C++.
   * DATE        : 6/18/2002
   * PGMR        : Y. Bai
   *****************************************************************************/
A #include "EXTERN.h"
  #include "perl.h"

B PerlInterpreter *my_perl;

C int main(int argc, char** argv, char** env)
  {
D     my_perl = perl_alloc();
E     perl_construct(my_perl);
F     perl_parse(my_perl, NULL, argc, argv, (char** )NULL);
G     perl_run(my_perl);
H     perl_destruct(my_perl);
I     perl_free(my_perl);

J     return 0;
  }
```

Figure 7-7

E. Call `perl_construct()` to initialize the `PerlInterpreter` object.

F. Call `perl_parse()` to pass and paste the `PerlInterpreter` object, variables, or components to the Perl domain and make the latter ready to run.

G. Call `perl_run()` to activate the Perl interpreter to execute the Perl script.

H. Call `perl_destruct()` to clean up the memory space allocated for the `Per-lInterpreter` object and variables.

I. Call `perl_free()` to remove the objects.

J. Returns a `0` to indicate to the system that this `main()` function has executed successfully.

In this source code, we didn't make any call to the Perl scripts. In other words, we didn't interface to any actual Perl scripts. We only translate our control from the C/C++ domain to the Perl domain, where the user can perform any Perl operation.

Now click `Build|Build SimplePerl.exe` to build our project. You will notice that some errors are occurring. No need to worry. We'll fix them one by one.

The first error occurs because the program cannot find the header file `#include <sys/types.h>`. This error comes from the `perl.h` header file in line `435`. This definition is an old one. The new definition of the `types.h` is under the folder `sys` in the C/C++ domain. You can change this definition based on the new definition, but here, because we will not use this `<sys/types.h>` header file, just comment it out.

You can fix this error by clicking Tools|Options, and add one more directory to the Include files; C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\SYS. Then replace the original statement, #include <sys/types.h> with #include <types.h>.

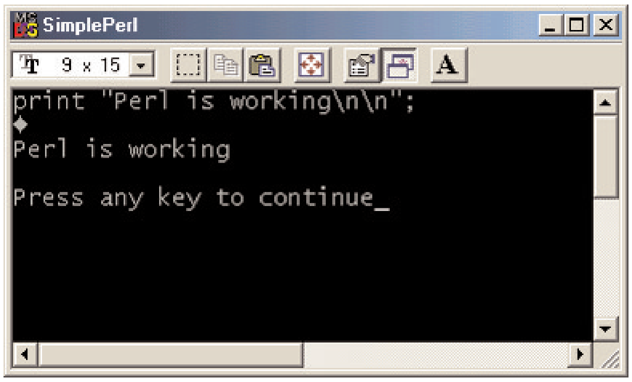
The next error occurs because the program cannot open the header file #include <sys/stat.h>. This error also happens in the perl.h header file, at line 673. This error can be solved if you add one more directory to the Include files as we did above. You also need to modify the original statement, #include <sys/stat.h>, to #include <stat.h>.

Another error may have occurred that indicates that it cannot open the header file #include <sys/types.h> in C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\SYS\STAT.h at line 65. Make the same correction; that is, replace the original statement #include <sys/types.h> with a new one: #include <types.h>.

All these errors are concerned with the versions of the different header files. You should be very careful when you touch these header files because some other programs may use them later, and problems might occur if you make a mistake in those header files.

After you have finished the modifications, you can rebuild your project by clicking Build|Build SimplePerl.exe again. This time, no errors should be displayed.

Now, click Build|Execute Simpleperl.exe to run this project. A command prompt window is displayed on screen, as shown in Figure 7-8. Nothing displays in the window. Is that supposed to happen? Yes, that is what should occur. Now, try to type something in this command window, such as print "Perl is working\n\n"; then press the Enter key. Press the Ctrl + D keys followed by another Enter key. Whatever you typed will be displayed on the screen, as shown in Figure 7-8.



```
print "Perl is working\n\n";
♦
Perl is working

Press any key to continue_
```

Figure 7-8

The reason nothing is displayed on screen when you begin to run your project is because, as you run the project, you start from within the Visual C ++ domain and run a console application by using the DOS-Like window. At that moment, you have already released controllability of the project from the Visual C++ domain to the Perl domain. Then you typed some Perl commands, for example, `print "Perl is working\n\n";`, which can be considered as a one-line Perl script. You pressed `Ctrl + D` combination keys to inform the system that you completed the Perl scripts' coding; after that you pressed the `Enter` key to begin to run this Perl script by calling the Perl interpreter. The Perl interpreter receives the start signal and begins to execute the one-line script, which is to print a string on the screen. In this way, we have finished this interface between Visual C++ and Perl.

As long as you begin to run our simple project from within the Visual C++ domain, you translate the system from the C/C++ domain to the Perl domain. After that, you can type any Perl codes on the window just as you make Perl coding in an editor. Then, you can immediately execute the codes you typed, which is very similar to executing a Perl script in the Perl environment.

You can also read and execute Perl scripts that are stored in a file from the Visual C++ domain. We can modify the C++ source file to call and run a Perl script file to perform some functionality. We can store the Perl filename to one of the arguments, `argv`, and paste it to the Perl stack area by calling the `perl_parse()` function, and finally, we can call `perl_run()` to run that Perl script.

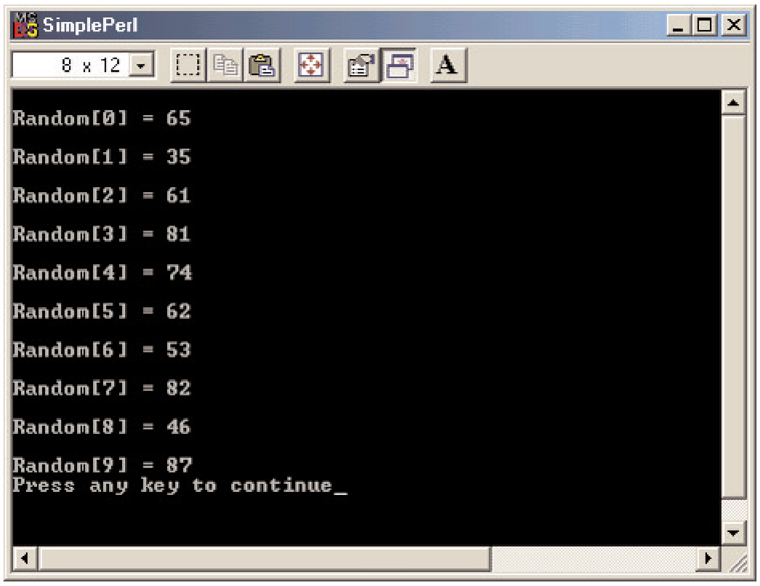
First, we create a Perl script. For testing purposes, we just utilize this script to create a random sequence with `10` elements. We can use the Perl function `rand()` to do that. The Perl script is shown in Figure 7-9. Open the NotePad editor to enter these codes, and save this file as `"Random.pl"` to our project folder `C:\PerlApp\SimplePerl`. Don't forget to use double quotation marks to enclose the filename, because the NotePad editor has only one default file type—text.

Now, open our C++ source code, `SimplePerl.cpp`, and add one more statement to this source code at the fourth line:

```
char* my_argv[] = {"", "Random.pl"};
```

```
$count = 0;
$num   = 0;

for ($count = 0; $count < 10; $count++)
{
   $num = int (rand 100);
   printf("\nRandom[%d] = %d\n", $count, $num);
}
```

Figure 7-9



```
Random[0] = 65
Random[1] = 35
Random[2] = 61
Random[3] = 81
Random[4] = 74
Random[5] = 62
Random[6] = 53
Random[7] = 82
Random[8] = 46
Random[9] = 87
Press any key to continue_
```

Figure 7-10

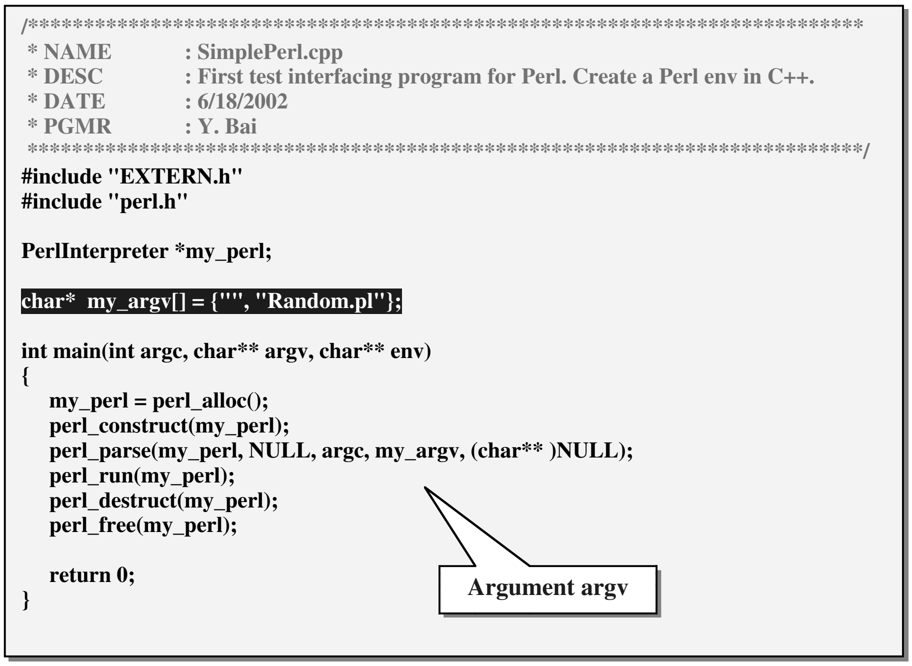
Rebuild our project by clicking `Build|Build SimplePerl.exe`, then click `Build|Execute SimplePerl.exe` to run this modified project. The running result will look like that displayed in Figure 7-10.

In the Perl script code (Figure 7-9), a `rand()` function is utilized to create a random sequence with a magnitude of `100`. Also, an `int` cast is used to translate the random numbers to integer numbers. This `rand()` function is placed inside a `for` loop, thus, in total, `10` random numbers are created. A `printf()` function, which is very similar to a function in C/C++, is called to print this random sequence on the screen. All variables defined in the Perl domain use a $ as the sign for the variable.

In the C++ source code, we assign the Perl script's name, `"Random.pl"`, to one of the arguments, `my_argv`. This argument will be passed into the Perl stack area when the `perl_paste()` function is called later on from within the C/C++ domain. The `perl_run()` function will be called after this argument is passed to run the Perl script that was passed.

To make things clear, Figure 7-11 shows a complete modified C++ source file. The modified section has been highlighted.

One point you need to pay attention to is the position in which the Perl script's name should be placed on this argument. Because the argument `my_argv` is a double pointer with a string variable type, you need to place the script's name at the second position of the string argument, which is `my_argv[1]` (index is `1`). The first position of the `my_argv` is `my_argv[0]` because the index of an array starts from `0`, not `1`.



```
/*****************************************************************************
 * NAME          : SimplePerl.cpp
 * DESC          : First test interfacing program for Perl. Create a Perl env in C++.
 * DATE          : 6/18/2002
 * PGMR          : Y. Bai
 *****************************************************************************/
#include "EXTERN.h"
#include "perl.h"

PerlInterpreter *my_perl;

char*  my_argv[] = {"", "Random.pl"};

int main(int argc, char** argv, char** env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, NULL, argc, my_argv, (char** )NULL);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);

    return 0;
}
```

Figure 7-11

7.3.3 A Rand Number Interface

Our interface design is a little more complicated. We try to call a Perl subroutine with two arguments, `Amp` and `Num`, to perform some random sequence operations in the Perl subroutine. The subroutine will first create a random sequence based on the magnitude `(Amp)` and the size `(Num)`, which are two arguments passed to the Perl domain. Then the subroutine will perform some calculations, such as to compute the mean value of the random sequence, and, finally, it will return the mean value to the C/C++ domain.

First, let's create a Perl script named `RandPerl.pl`, as shown in Figure 7-12. The subroutine is named `perlRAND`. This subroutine has two arguments, `$a` and `$n`, which represent the `Amplitude` and the `Number` of the newly created random sequence.

A. The format of a function with arguments in the Perl domain is expressed with an `@_` symbol, which represents an arguments list. The arguments passed to the Perl subroutine are defined in the C/C++ domain as the name of `my_argv[]` (will be shown later). So in the Perl domain, the subroutine with these two arguments should be expressed as `my ($a, $n) = @_`; The `@_` holds the arguments list. `$a` and `$n` are two nominal arguments passed to the Perl domain.

B. Some variables are defined at the beginning of the subroutine, and these variables will be used in this subroutine only. All variables defined in the Perl domain should be preceded by a `$` symbol and should be initialized to `0`.

C. A `rand()` Perl function is utilized to create a set of random numbers. The range of the magnitude of the random sequence is determined by a value (either an integer value or a variable), which follows the rand function name (`$a` here). For example, the `(rand)` will create a random sequence with a magnitude between `0` and `1`. In our case, the function `int(rand $a)` will create a random sequence with a magnitude between `0` and `$a`. An `int` cast is used to convert the random data to the integer values. The `for` loop is used to repeat the sequence `$n` times.
Each newly created random number will be assigned to the variable `$nums`. A `printf()` function is called to print that random number on the screen. The variable `$sum` is used to summarize the total value of all random numbers, which will be used to calculate the mean value for this random sequence in the next step.

D. The variable `$mean` is used to get the mean value of this random sequence. The calculation for this mean is straightforward (`$sum/$n`), where `$n` is the total number of elements in the newly created random sequence and is passed from the C/C++ domain. Finally, this mean value is returned to the C/C++ domain.

Save this Perl script as `RandPerl.pl` to our user-defined project directory `C:\PerlApp\CPRandom`.

```
    sub perlRAND
    {
A      my ($a, $n) = @_;
       $nums  = 0;
B      $count  = 0;
       $sum    = 0;
       $mean  = 0;

       printf("\n");
       for ($count = 0; $count < $n; $count++)
       {
C              $nums = int(rand $a) + 1;
               printf(" Random Number %d:    %d\n", $count, $nums);
               $sum += $nums;
       }

D      $mean = $sum/$n;
       return $mean;
    }
```

Figure 7-12

There is clearly no problem with those Perl script codes. Let's open Visual C++ and create another new project with a type of `Win32 Console Application`. Enter the name `CPRandom` into the `Project name`: field as the name for this new project, and make sure that the path in the `Location`: field is `C:\PerlApp\`, which is our user-defined project directory.

Create a new C++ source file with a name of `CPRandom.cpp` by clicking `File|New`. Open this new source file and enter the code shown in Figure 7-13 into this file.

A. First, a `PerlInterpreter` object, `*my_perl`, is created at the beginning of the source code. This object is a structure that provides all necessary information

```
   /*****************************************************************************
    * NAME        : CPRandom.cpp
    * DESC        : Interfacing C++ program to call a perl program to create & compute
    *             : a random sequence
    * DATE        : 6/18/2002
    * PGMR        : Y. Bai
    *****************************************************************************/
   #include "EXTERN.h"
   #include "perl.h"

A  PerlInterpreter *my_perl;

B  static void PerlRandom(int a, int n);

   int main(int argc, char** argv, char** env)
   {
C          int     amp, num;
           char*  my_argv[] = {"", "RandPerl.pl"};

           printf("\nEnter the Amplitude & Number of the Random Sequence\n");
D          scanf("%d", &amp);
           scanf("%d", &num);
           my_perl = perl_alloc();
           perl_construct(my_perl);

           perl_parse(my_perl, NULL, 2, my_argv, (char** )NULL);
           perl_run(my_perl);

E          PerlRandom(amp, num);

           perl_destruct(my_perl);
           perl_free(my_perl);

           return 0;
   }
```

Figure 7-13

```
F   static void PerlRandom(int a, int n)
    {
            dSP;                                      // initialize the perl stack pointer
            ENTER;                                    // starting perl from here...
            SAVETMPS;                                 // temporary variable
            PUSHMARK(SP);                             // save the stack pointer
            XPUSHs(sv_2mortal(newSViv(a)));           // push the amplitude into the stack
            XPUSHs(sv_2mortal(newSViv(n)));           // push the number into the stack
            PUTBACK;                                  // make local stack pointer to global
G           call_pv("perlRAND", G_SCALAR);            // call perl subroutine
            SPAGAIN;                                  // refresh the stack pointer

H           printf("\nThe returned mean of the random sequence is: %d\n", POPi);
            printf("\n");
            PUTBACK;
            FREETMPS;                                 // free returned value
            LEAVE;                                    // XPUSHed "mortal" args
    }
```

Figure 7-13 (*continued*)

for the Perl interpreter. This object is defined in the `perl.h` header file, therefore, you cannot change the name for this object.

B. A class method, `PerlRandom()`, is declared in this source file. The purpose of this method is really to interface to the Perl domain by calling a Perl subroutine, `perlRAND`, which we developed previously. Two integer arguments, `a` and `n`, are included in this function. The first argument represents the amplitude of the newly created random sequence, and the second one represents the number of elements in the newly created random sequence.

C. Declare two local integer variables, `amp` and `num`, which will be used as the actual variables for the amplitude and the number of the random sequence. These two actual variables will replace the nominal variables `a` and `n` defined in the method `PerlRandom(int a, int n)` and will be passed into the Perl domain.

Also, the Perl script file `RandPerl.pl` developed earlier will be assigned to a `my_argv[1]` argument, and that argument will be passed and pasted into the Perl stack area when the `perl_paste()` function is called and executed later on. In this way, the name of the Perl script file is passed to the Perl domain.

D. Two `scanf()` functions are utilized to retrieve the values of the amplitude and the number for a random sequence. These two values are entered by the user as the program runs. The following codes are identical to the codes we used in the previous example.

Table 7-1

Returned Macro	**Returned Type**
`POPs`	`SV`
`POPp`	`pointer`
`POPn`	`double`
`POPi`	`integer`
`POPl`	`long`

After executing the `perl_paste()` function, the name of the Perl script file has been passed and pasted to the Perl domain. The Perl script will be run when `perl_run()` is called and executed.

E. The class method `PerlRandom()`, which was defined at the beginning of this source file, will be called with two arguments, `amp` and `num`.

F. This is the implementation of the method `PerlRandom()`. All statements are explained clearly by the comments, so we will not waste time on these codes.

G. We use a C function, `call_pv()`, to call and access the subroutine `perlRAND`, which is defined in the Perl domain, to execute the creation and computation of the random sequence. Two arguments are included in this function. The first one is the subroutine's name, which is a C string pointer (`char*`), and the second is a flag, which is used to tell Perl whether this function call includes any arguments, has returned values, or whether the arguments are scalar variables or an array variable, and so on. In our application, we use `perlRAND`, which is our subroutine's name, as the first argument, and we use `G_SCALAR` as the second argument to indicate to Perl that the arguments we passed it to are scalar variables.

H. Finally, we retrieve the returned value from the Perl subroutine by using a macro `POPi` to pop that value from the stack area. We are supposed to get an integer value, so `POPi` is used. Refer to Table 7-1 for some other returned data types.

SV is a special pointer type and can be a C-string variable or a reference to the Perl subroutine. Perl provides a number of C functions, and by using these functions, you can call and access the different Perl subroutines from within the C/C++ domain to the Perl domain. In the following lines, four functions are listed and explained.

- `I32 call_sv(SV* sv, I32 flags);`
- `I32 call_pv(char* subname, I32 flags);`

- `I32 call_method(char* methname, I32 flags);`
- `I32 call_argv(char* subname, I32 flags, register char** argv);`

All the functions are fairly simple, except the first one, `call_sv()`. All the functions have a flags parameter, which is used to pass a bit mask of options to Perl. This bit mask operates identically for each of the functions.

`call_sv():` This call takes two parameters. The `sv`, is an `SV*`. This allows you to specify the Perl subroutine to be called either as a C-string (which has first been converted to an `SV`) or a reference to a subroutine.

`call_pv():` This function is similar to the `call_sv()` except that it expects its first argument to be a `C-char*`, which identifies the Perl subroutine you want to call. You need to use a full name for the subroutine's name if that subroutine is located in another package or namespace. For example, you need to use the name `"pack::random"` to call the subroutine that is named `random` and is located in a package named `pack`.

`call_method():` This function is used to call a method from a Perl class. The argument `methname` corresponds to the name to be called. Note that the class that the method belongs to is passed on the Perl stack rather than in the argument list. This class can be either the name of the class (for the class method) or a reference to an object (for a virtual method).

`call_argv():` This function calls the Perl subroutine specified by the C string stored in the `subname` parameter. It also takes the usual flags parameter. The final parameter, `argv`, consists of a NULL terminated list of C strings to be passed as parameters to the Perl subroutine.

The flags parameter has the following values:

`G_VOID:` Call a Perl subroutine without any returned value (`void`). The value returned from the `call_xx` function indicates how many items were returned from the Perl subroutine. In this case, it should be a `0`.

`G_SCALAR:` Call a Perl subroutine that returns a scalar value only.

The value returned from the `call_xx` function indicates how many items were returned from the Perl subroutine. In this case, it will be either `0` or `1`.

If `0`, that means that you have specified a `G_DISCARD` flag when you called the Perl subroutine. This means that you don't take care of the returned value. If `1`, then the subroutine actually returned a scalar value and that value is stored in the Perl stack area.

`G_ARRAY:` Call a Perl subroutine that will return an array. The value returned from the `call_xx` function indicates how many items were returned from the called Perl subroutine. If `0`, it means that you have specified a `G_DISCARD` flag when you called the Perl subroutine. If not `0`, then it will be a count of the number of items returned by the subroutine. These items will be stored on the Perl stack area in an order of Last-In-First-Out (LIFO). So you should be very careful when you pick up the returned items from the Perl stack area. In the following example, we show this situation.

G_DISCARD: By default, the `call_xx` function places the items returned from the Perl subroutine in the Perl stack area. If you are not interested in these returned items, you can set this flag to `G_DISCARD` to get rid of them automatically. But you can still indicate a returned item to the Perl subroutine by using either `G_SCALAR` or `G_ARRAY`.

G_NOARGS: When you call a Perl subroutine by using the `call_xx` function, it is assumed by default that some arguments are to be passed to the subroutine. If you are not passing any arguments to the Perl subroutine, you can save a bit of time by setting this flag. It has the effect of not creating the `@_` array for the Perl subroutine.
Although the functionality provided by this flag may seem straightforward, it should be used only if there is a good reason to do so. The reason for being cautious is that even if you have specified the `G_NOARGS` flag, it is still possible for the Perl subroutine that has been called to think that you have passed it arguments. In fact, what happens is that the Perl subroutine you called can access the `@_` array from a previous Perl subroutine. This will occur when the code that is executing the `call_xx` function has itself been called from another Perl subroutine. So additional care should be taken for this issue.

G_EVAL: It is possible for the Perl subroutine you called to terminate abnormally. Should you desire to terminate the program, you can use the `die` statement explicitly. By default, when either of these events occurs, the process will terminate immediately. If you want to trap this type of event, specify the `G_EVAL` flag. It will put an `eval{}` around the subroutine call. Whenever control returns from the `call_xx` function, you need to check the `$@` variable as you would in a normal Perl script.
The value returned from the `call_xx` function is dependent on what other flags have been specified and whether an error has been encountered. The following are all the different cases that can occur:

- If the `call_xx` function returns normally, then the value returned is as specified in the previous sections.
- If `G_DISCARD` is specified, the returned value will always be `0`.
- If `G_ARRAY` is specified and an error has occurred, the returned value will always be `0`.
- If `G_SCALAR` is specified and an error has been encountered, the returned value will be `1` and the value on the top of the stack will be `undef`. This means that if you have already detected the error by checking `$@` and you want the program to continue, you must remember to pop the `undef` from the stack.

G_KEEPERR: You may have noticed that using the G_EVAL flag will always clear the $@ variable and set it to a string describing the error if there was an error in the called code. This unqualified resetting of $@ can be problematic in the reliable identification of errors using the eval{} mechanism, because the possibility exists that Perl will call other code between the time the error causes $@ to be set within eval{} and the subsequent statement that checks for the value of $@ gets executed in the user's script. The G_KEEPERR flag is used in conjunction with G_EVAL in the call_xx functions that are used to implement such a situation. This flag has no effect when G_EVAL is not used. When G_KEEPERR is used, any errors in the called code will be prefixed with the string "\t (in cleanup)" and appended to the current value of $@.

Now, we are almost ready to build our project. Before we do that, we need to copy some files to our project folder and add them to our project. Open Windows Explorer and copy the following files from C:\Perl\lib\CORE to our new project folder, C:\PerlApp\CPRandom:

- perl.h
- perl56.lib

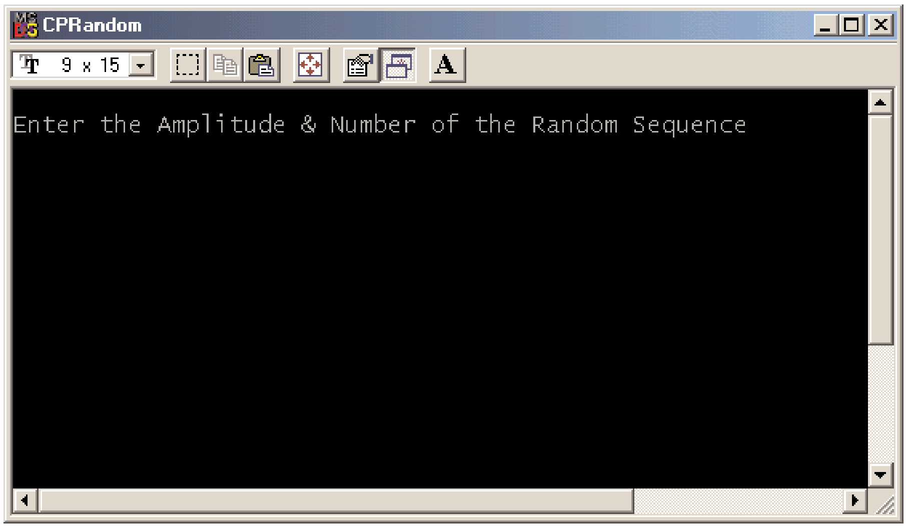
Return to the Visual C++ 6.0 workspace, click Project|Add To Project|Files. . . to open the Insert Files into Project dialog box, then select the items we just copied (perl.h and perl56.lib) from the list and click OK to add these two files to our project. Make sure that you selected All Files (*.*) from the Files of type: list box in the preceding operation.

Now click Tools|Options to open the Options dialog box, then click the Directories tab and select Include files from the Show directories for: list box. In the Directories list, double-click a dashed-line box to display a button on the right end, then click that button to browse the directories and to locate the folder C:\Perl\lib\CORE. Click OK to select this folder as the additional directory for our project. In a similar way, select the folder C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE as the additional directory for our project.

We can build our new project now by clicking Build|Build CPRandom.exe. It should not encounter an error if everything is OK. We try to run our project by clicking Build|Execute CPRandom.exe in the Visual C++ domain. A command prompt window is displayed, as shown in Figure 7-14.

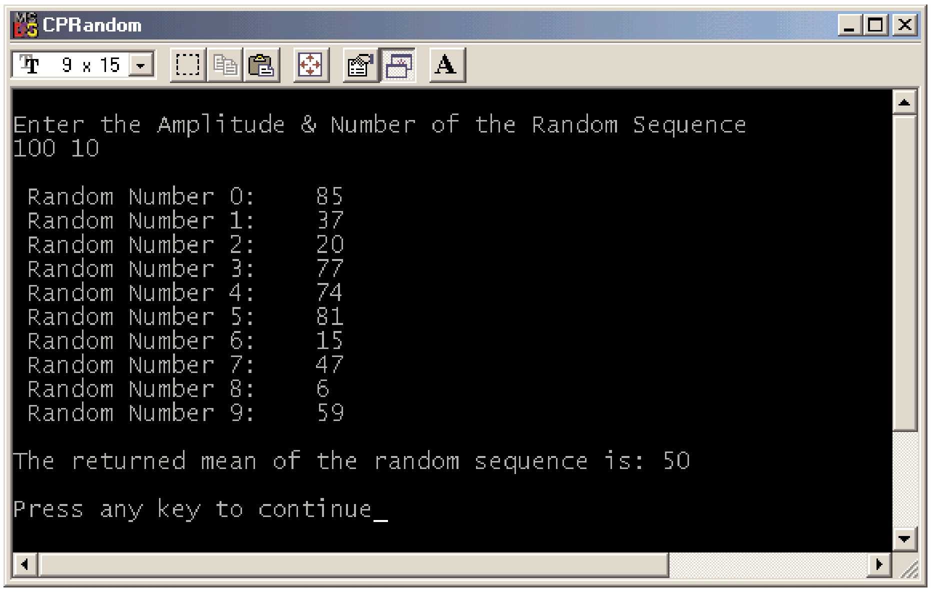
Enter some integers, such as 100 and 10, which are associated with the amplitude and the number of the random sequence, respectively. Leave a space between the two integers, and press the Enter key. The project begins to call and access the script and subroutine defined in the Perl domain. The Perl subroutine will pick up the arguments passed to it, create a set of random sequences, calculate the mean value, and finally, return the mean value to the Perl stack area.

07-P2521 2/11/03 10:09 AM Page 799



```
Enter the Amplitude & Number of the Random Sequence
```

Figure 7-14



```
Enter the Amplitude & Number of the Random Sequence
100 10

 Random Number 0:    85
 Random Number 1:    37
 Random Number 2:    20
 Random Number 3:    77
 Random Number 4:    74
 Random Number 5:    81
 Random Number 6:    15
 Random Number 7:    47
 Random Number 8:    6
 Random Number 9:    59

The returned mean of the random sequence is: 50

Press any key to continue_
```

Figure 7-15

The C/C++ program will pick up the returned mean value from the Perl stack and display it on the screen. The running result is displayed in Figure 7-15. The result of the project running is wonderful. Ten (`10`) random numbers are created and displayed on screen, and the mean value is computed and returned to the C/C++ domain. The final mean result is displayed by using the `printf()` function in the C/C++ domain, not in the Perl domain. Press any key to close this project.

7.3.4 A Modified Rand Number Interface

In this example, we modify the previous example by adding a data array to hold the random numbers. Also, we add a functionality of calculating the maximum, minimum, and mean values for the random sequence. The returned item from the Perl subroutine is a data list that contains the calculated results (the maximum, minimum, and mean values for the random sequence).

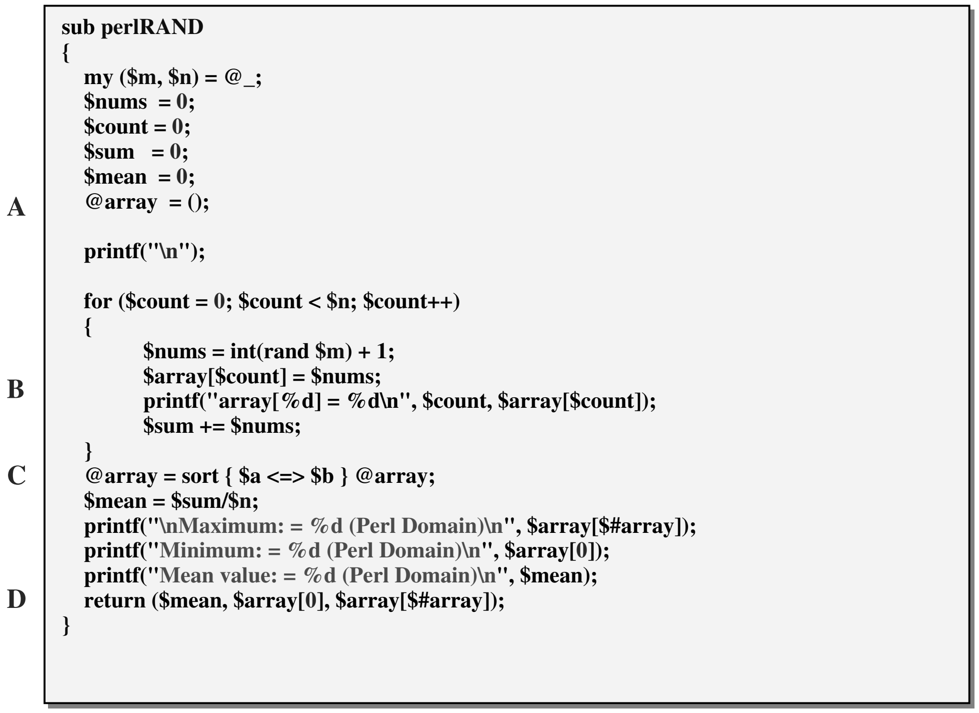
By using this example, we want to show readers how to interface to Perl from the C/C++ domain and obtain the data list, and how to retrieve the returned data items from the Perl stack area. By introducing this example, we also show readers how to handle an array in the Perl script.

Launch Visual C++ 6.0 and create a new project with a type of `Win32 Console Application`, and enter `CPRndArray` into the `Project name`: input field as the name for this new project. Make sure the `Location`: field is our user-defined project directory, `C:\PerlApp`.

Open the NotePad editor and enter the code shown in Figure 7-16 as the Perl script, which contains a subroutine, `perlRAND`. This subroutine is very similar to the last one. At the beginning of the subroutine, some variables are declared. One more item, `@array = ()`, is added in the beginning of the subroutine. This is a data array expressed in the Perl domain. Generally, you can use the following format to create a new array in the Perl domain:

```
@numArray = (1, 2, 3, 4, 5);
    @numArray = (1..100);
        @numArray = ();
```

The first line creates a new data array `numArray` and initializes it for `5` elements, with the values from `1` to `5`. The second line creates a data array that can hold up to `100` elements. The (`1..100`) statement is used to define the dimension of the array. The last line is identical to the one that we used in our example. This data array is an empty array with no elements in it. However, you can add any amount of elements to this array by assigning new elements to the array. Unlike the array defined in C/C++ , where the number of elements in a defined array cannot be beyond its upper bound, in Perl, you can still assign a `101st` element to an array that is defined to hold up to `100` elements. That is legal in Perl! Perl will automatically enlarge the capacity for the array to hold the newly added elements.



```
sub perlRAND
{
   my ($m, $n) = @_;
   $nums  = 0;
   $count = 0;
   $sum   = 0;
   $mean  = 0;
   @array  = ();

   printf("\n");

   for ($count = 0; $count < $n; $count++)
   {
           $nums = int(rand $m) + 1;
           $array[$count] = $nums;
           printf("array[%d] = %d\n", $count, $array[$count]);
           $sum += $nums;
   }
   @array = sort { $a <=> $b } @array;
   $mean = $sum/$n;
   printf("\nMaximum: = %d (Perl Domain)\n", $array[$#array]);
   printf("Minimum: = %d (Perl Domain)\n", $array[0]);
   printf("Mean value: = %d (Perl Domain)\n", $mean);
   return ($mean, $array[0], $array[$#array]);
}
```

Figure 7-16

So, we used this principle in our example.

A. Create an empty data array `@array` with no elements inside.

B. In the `for` loop, we call `rand()` to create a new random sequence with an amplitude of `$m`, which is an argument passed to the Perl subroutine from the C/C++ domain. Then we assign that random number to a local variable `$nums`. The following step assigns that local variable to the data array we created at the beginning of the subroutine by using: `$array[$count] = $nums`. In Perl, you have to use `$array[]` to access an element on the array. The next step is to use the `printf()` function to print the element of the data array.

C. We want to perform some calculations to obtain the maximum, minimum, and mean values for the random sequence. In C/C++ , we have to write a sequence of code to fulfill this job. But in Perl, a special function, `sort()`, is provided to handle this task. We use the format

```
@array = sort {$a <=> $b} @array
```

to invoke this sorting operation. The sorting result rearranges the elements in the random array in a certain order. In our case, we want to arrange the elements

in the array from the smallest to the largest in values. The smallest element is located at `$array[0]`, and the largest element is located at `$array[$n]`. `$n` is an argument passed from the C/C++ domain as the upper bound to the random sequence.
In Perl, the index of the last element in an array (or the length of the array) can be expressed as `$#array_name`. For instance, in our example, the length of the random array or the index of the last element in the array can be written as `$#array`, and the last element in the array can be expressed as

```
$array[$#array]
```

So after the sorting operation, the minimum value of the array is `$array[0]`, the maximum value is `$array[$#array]`, and the median value is `$array[$n/2]`. All these sorting results will be printed on the screen to show the user.

D. Finally, the subroutine returns the sorting results to the stack area in Perl. Please pay special attention to the order in which the results are put in the returning list: first, the mean value; second, the minimum value, `$array[0]`; and finally, the maximum value, `$array[$#array]`. These three items are pushed into the stack in that order, which means that the mean value is pushed at the bottom of the stack, the minimum value is pushed at the top of the mean value, and the maximum value is pushed on the top position of the stack. This order determines the sequence in which you pick up the items (or pop up the items) later on in the C/C++ domain. Figure 7-17 shows this situation.

As shown in Figure 7-17, three elements are positioned in that order. When you pop the stack, the top element, the maximum value (`$array[$#array]`), will be picked up first. The minimum value (`$array[0]`) will pop second, and the mean value will pop last. That is the traditional mode for a stack, which is defined as Last-In-First-Out (LIFO).

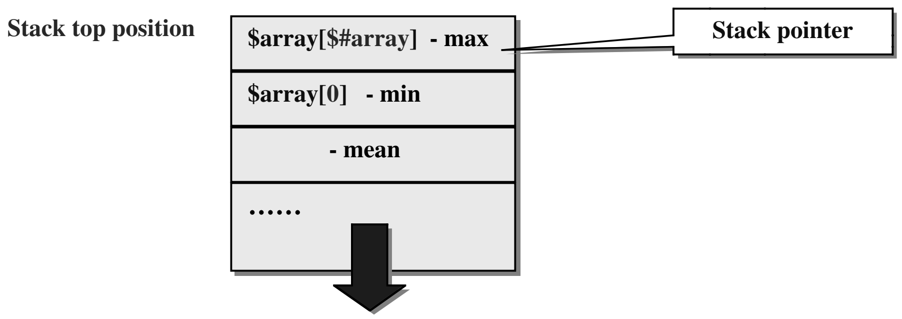


Figure 7-17

Save this subroutine as RndArrayPerl.pl to our newly created Visual C++ project folder CPRndArray, which is located at C:\PerlApp. Now open Windows Explorer and copy the following files from the directory C:\Perl\lib\CORE to our new project folder, C:\PerlApp\CPRndArray:

- perl.h
- perl56.lib

Return to the Visual C++ 6.0 workspace, click Project|Add To Project|Files. . . to open the Insert Files into Project dialog box, then select the items we just copied (perl.h and perl56.lib) from the list and click OK to add these two files into our project. Make sure that you selected All Files (*.*) from the Files of type list box in the preceding operation.

Now click Tools|Options to open the Options dialog box, then click the Directories tab and select Include files from the Show directories for: list box. In the Directories list, double-click a dashed-line box to display a button on the right end, then click that button to browse the directories and to locate the folder C:\Perl\lib\CORE. Click OK to select this folder as the additional directory for our project. In a similar way, select the folder C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE as the additional directory for our project. You may need to add another include directory, C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\SYS.

Click File|New from the Visual C++ domain to create a new C++ source file. Enter CPRndArray in the File name: input field as the name for this source file. Click OK to open this new source file, and enter the code shown in Figure 7-18 into this file.

This source file looks very similar to the one in the last project. First, we declare a function, PerlRandom(), which is used to interface to Perl and to call a Perl subroutine to perform creating a random sequence and computing the maximum, minimum, and mean values based on that random sequence. The Perl subroutine perlRAND is defined inside a Perl script named RndArrayPerl.pl, so we assign this script's name to the argument my_argv[1].

A. When we use call_pv() to call the subroutine perlRAND, the first argument of this function is a C-string that contains the subroutine's name. The second argument is a flag that is used to tell the Perl interpreter that we need this function to return an array of items by using G_ARRAY. For other flags' definitions, refer to the last section to get details.

B. When the subroutine returns, we need to pick up the returned items from an array. To be precise, we need to retrieve the returned items from the Perl stack area. Refer to Figure 7-17. The returned items are stored in the stack in the same

```
/*****************************************************************************
 * NAME          : CPRndArray.cpp
 * DESC          : C++ program to call a Perl script to create & compute random data
 * DATE          : 6/19/2002
 * PGMR          : Y. Bai
 *****************************************************************************/
#include "EXTERN.h"
#include "perl.h"

PerlInterpreter *my_perl;
static void PerlRandom(int a, int n);
int main(int argc, char** argv, char** env)
{
            int     amp, num;
            char*  my_argv[] = {"", "RndArrayPerl.pl"};

            printf("\nEnter the Amplitude & Number of the Random Sequence\n");
            scanf("%d", &amp);
            scanf("%d", &num);
            my_perl = perl_alloc();
            perl_construct(my_perl);

            perl_parse(my_perl, NULL, 2, my_argv, (char** )NULL);
            perl_run(my_perl);

            PerlRandom(amp, num);

            perl_destruct(my_perl);
            perl_free(my_perl);

            return 0;
}
```

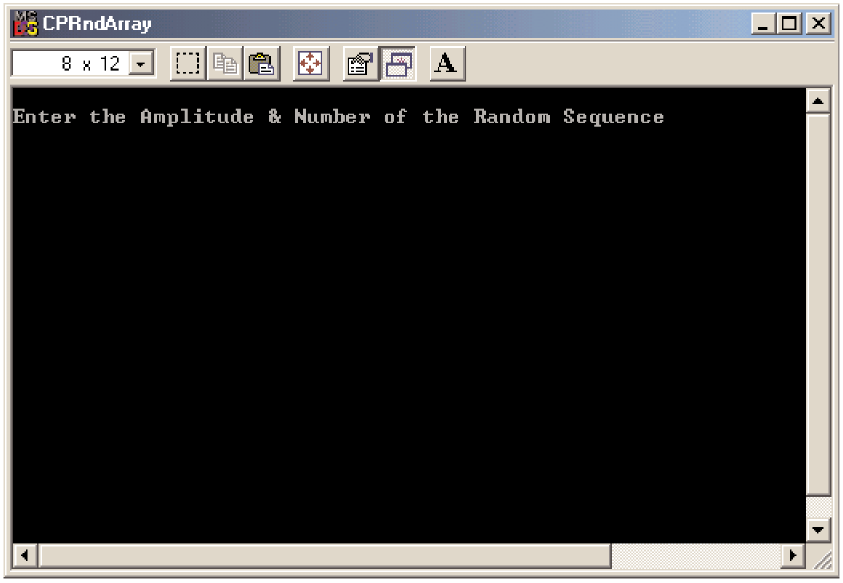
```
    static void PerlRandom(int a, int n)
    {
                dSP;                                        // initialize the perl stack pointer
                ENTER;                                      // starting perl from here...
                SAVETMPS;                                   // temporary variable
                PUSHMARK(SP);                               // save the stack pointer
                XPUSHs(sv_2mortal(newSViv(a)));             // push the amplitude into the stack
                XPUSHs(sv_2mortal(newSViv(n)));             // push the number into the stack
                PUTBACK;                                    // make local stack pointer to global
A               call_pv("perlRAND", G_ARRAY);               // call perl subroutine
                SPAGAIN;                                    // refresh the stack pointer

                printf("\nThe returned max of the random sequence is: %d\n", POPi);
B               printf("The returned min of the random sequence is: %d\n", POPi);
                printf("The returned mean of the random sequence is: %d\n", POPi);
                printf("\n");
                PUTBACK;
                FREETMPS;                                   // free returned value
                LEAVE;                                      // XPUSHed "mortal" args
    }
```

Figure 7-18

order in which they are pushed. This means that in the top position should be the maximum value, `$array[$#array]`. So when we use the macro `POPi` to pop up the first item from the stack, we should obtain this maximum value. When we continue to use the `POPi` macro to pop the next item, it should be the minimum value, `#array[0]`. Finally, the `POPi` macro returns the mean value from the stack. So, in this way, we retrieve all returned items from the Perl stack. You may have noticed that the order in which we call the macro `POPi` is very important in picking up the returned items. The first `POPi` will return the top element stored on the stack, the second `POPi` will return the item that is located in the next position on the stack, and finally, the third `POPi` will return the item that is located on the bottom of the stack.
As long as you keep the correct order to call the `POPi` macro, you can always obtain the correct returned items with no problem.

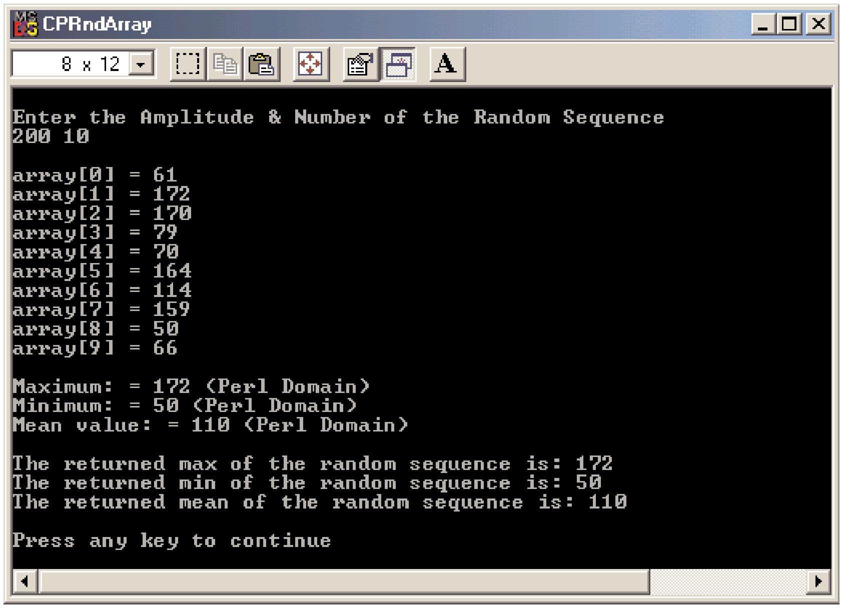
We can build our new project now by clicking the `Build|Build CPRndArray.exe` menu item. It should not have any errors if everything was entered correctly. Now, click the `Build|Execute CPRndArray.exe` menu item to run our project. A command window is displayed on screen, as shown in Figure 7-19.

Enter some numbers in the program, such as `200` and `10`, which represent the amplitude and the number of the elements in the newly created random sequence. Keep a space between the two entered numbers and press the `Enter` key to continue running the project. After the project is completed, the results shown in Figure 7-20 should be displayed on the command window.



```
Enter the Amplitude & Number of the Random Sequence
```

Figure 7-19



```
Enter the Amplitude & Number of the Random Sequence
200 10

array[0] = 61
array[1] = 172
array[2] = 170
array[3] = 79
array[4] = 70
array[5] = 164
array[6] = 114
array[7] = 159
array[8] = 50
array[9] = 66

Maximum: = 172 (Perl Domain)
Minimum: = 50 (Perl Domain)
Mean value: = 110 (Perl Domain)

The returned max of the random sequence is: 172
The returned min of the random sequence is: 50
The returned mean of the random sequence is: 110

Press any key to continue
```

Figure 7-20

While the program was running, we printed all the random numbers we created. Also, we printed out the calculation results (sorting results, actually) in the Perl domain by using Perl statements. Those lines are clearly marked with `<Perl Domain>`. Finally, we printed out the returned results in the C/C++ domain by using the C/C++ `printf()` function. The program's results show that our project is successful. Press any key to end our project.

7.3.5 Evaluating a Perl Statement from the C/C++ Domain

In this section, we want to show readers how to evaluate pieces of Perl code from the C/C++ domain.

Perl provides two API functions to evaluate code developed in the Perl domain (`eval_sv` and `eval_pv`). These are the only routines you need to execute snippets of Perl code from within a C/C++ program. Your code can be any length, and it can contain multiple statements. Also, it can employ some other functions provided by Perl to include external Perl files.

`Eval_pv` lets us evaluate individual Perl strings and extract variables for conversion into C types. The next example illustrates the implementation of this function.

```
  /*****************************************************************************
   * NAME:   EvalPerl.cpp
   * DESC:    Evaluate Perl code from within C/C++ domain
   * DATE:    6/22/2002
   * PGRM:   Y. Bai
   ****************************************************************************/
  #include "EXTERN.H"
  #include "perl.h"
  PerlInterpreter  *my_perl;
  int main()
  {
A           STRLEN  size;
B           char*   embed[] = { "", "-e", "0" };

            my_perl = perl_alloc();
            perl_construct(my_perl);
C           perl_parse(my_perl, NULL, 3, embed, NULL);
            perl_run(my_perl);

            /* treat rand as an integer */
D           eval_pv("$rand = 100; $rand *= 5;", TRUE);
E           printf(" \nEvaluated: (Integer): rand = %d\n", SvIV(get_sv("rand", FALSE)));

            /* treat rand as a float */
            eval_pv("$rand = 3.0; $rand **= 3;", TRUE);
F           printf(" \nEvaluated: (Float) rand = %.3f\n", SvNV(get_sv("rand", FALSE)));

            /* treat rand as a string */
            eval_pv("$rand = 'Test Eval_pv in Perl Domain';", TRUE);
G             printf(" \nEvaluated: (String) rand = %s\n\n", SvPV(get_sv("rand", FALSE), size));

            perl_destruct(my_perl);
            perl_free(my_perl);

            return 0;
  }
```

Figure 7-21

Launch Visual C++ 6.0 and create a new project with a type of `Win32 Console Application`. Enter `EvalPerl` in the `Project name`: input field as the name for this new project. Make sure that the `Location`: field contains our user-default program directory, `C:\PerlApp`. Click `OK` to open this new project.

Click `File|New` to create a new C++ source file, and enter `EvalPerl` in the `File name`: field, which represents our new source file's name. Click `OK` to open this new source file. Enter the code shown in Figure 7-21 into this new source file.

A. `STRLEN` is a macro used in Perl to define a length variable.

B. The argument array `embed[]`, which will be passed into the Perl domain, is declared and initialized here.

C. A `perl_parse()` function is called to transfer and paste the arguments to the Perl domain.

D. The `eval_pv()` function is called to pass a Perl statement from within the C/C++ domain to the Perl domain to perform an evaluation in there. The purpose of this function is to tell Perl to evaluate the given string and return an `SV*` result. The prototype of this function is

```
SV* eval_pv(const char* p, I32 croak_on_error);
```

Perl has three `typedefs` that handle Perl's three main data types:

- `SV`: scalar value
- `AV`: array value
- `HV`: hash value

The `eval_pv()` function returns a pointer that points to a scalar value. Two arguments are taken by this function. The first one is a string that contains a piece of Perl code, which will be passed into the Perl domain to be evaluated. The second argument is a `typedef` system constant with 32-bit, which works as a flag to tell Perl to pop up an error message if this evaluation encountered any mistake. In our application, we passed a Perl string `"$rand = 100; $rand * = 5;"` to the Perl domain, and set TRUE to the flag, which means that a `croak_on_error` flag will be shown if any error was encountered.

E. When the evaluation is completed and returned to the C/C++ domain, we use a C function, `printf()`, to display the evaluation result. The returned item from the `eval_pv()` is a pointer that points to a scalar value (`SV*`). A macro, `SvIV()`, is used to access the `SV*` pointer and to pick up the returned value.
In Perl, four types of values can be loaded:

- `IV`: integer value
- `NV`: double value
- `PV`: string value
- `SV`: another scalar value

Also in Perl, the following macros are provided to access the different pointers:

- `SvIV(SV*)`
- `SvUV(SV*)`
- `SvNV(SV*)`
- `SvPV(SV*, STRLEN len)`
- `SvPV_nolen(SV*)`

All of these macros can automatically convert the actual scalar type into an `IV`, `UV`, double, or a string.
The macro `SvIV` will take care of converting a signed integer pointer to an associated signed integer value. The `SvUV` will handle converting an unsigned integer pointer to the associated value.

In the `SvPV` macro, the length of the string returned is placed into the variable `len` (this is a macro, so you cannot use `&len`). You can use the `SvPV_nolen` macro if you do not care what the length of the data is.

A macro, `get_sv()`, is used to retrieve a pointer to its associated `SV`, which has the following prototype:

```
SV* get_sv("package::varname", FALSE);
```

In our case, we used `get_sv("rand", FALSE)` to get the pointer that points to the scalar variable, `rand`. Because we do everything in our current package, no package name is needed here. The returned pointer will be converted to the associated scalar value by the `SvIV` macro.

F. This line is very similar to the last line. The only difference is that we use the `SvNV` macro to replace the macro, `SvIV`, to obtain the returned double value.

G. Similarly with the last line, we use a macro `SvPV` to get a string value. This macro needs an additional argument, `len`, so a `size` variable is used here.

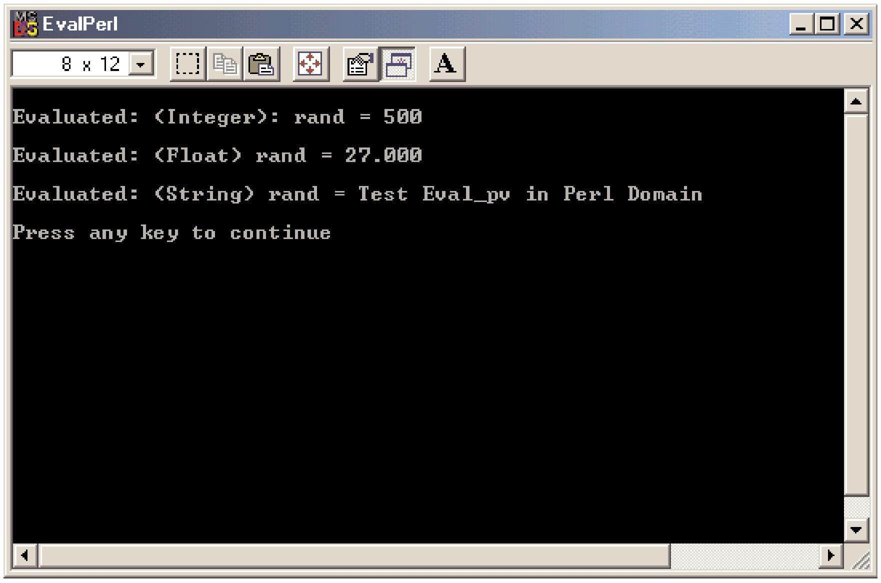
All other code in this source file is either straightforward or has been discussed in the previous section. Before we can build this code, we need to open Windows Explorer to copy the Perl library, `perl56.lib`, from the system directory `C:\Perl\lib\CORE` to our current project folder `C:\PerlApp\EvalPerl`.

Return to the Visual C++ 6.0 workspace, click `Project|Add To Project|Files . . .` to open the `Insert Files into Project` dialog box, then select the item we just copied (`perl56.lib`) from the list and click `OK` to add this file into our project. Make sure that you selected the `All Files (*.*)` item from the `Files of type` list box in the preceding operation.

Now click `Tools|Options` to open the `Options` dialog box, then click the `Directories` tab and select `Include files` from the `Show directories for`: list box. In the `Directories` list, double-click a dashed-line box to display a button on the right end, then click that button to browse the directories and to locate the folder `C:\Perl\lib\CORE`. Click `OK` to select this folder as the additional directory for our project. In a similar way, select the folder `C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE` as the additional directory for our project. You may need to add another include directory, `C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\SYS`.

Now, click the `Build|Build EvalPerl.exe` menu item to build our project. Then click `Build|Execute EvalPerl.exe` to run it. A command window is displayed on the screen, and the running results are also displayed on this screen, as shown in Figure 7-22. It looks good for our project. Press any key to end the project.

By using the `eval_pv()` function, you can evaluate any piece of code in Perl, even load an extension library. You can evaluate a Perl code that needs to load an `IO`



```
Evaluated: (Integer): rand = 500
Evaluated: (Float) rand = 27.000
Evaluated: (String) rand = Test Eval_pv in Perl Domain
Press any key to continue
```

Figure 7-22

module, or an `LWP` module. But you have to be very careful because some additional information is needed to handle that job. For example, if you want to embed a script that uses a Perl module (such as Socket or LWP) that itself uses a C/C++ library, you may encounter some compiling errors.

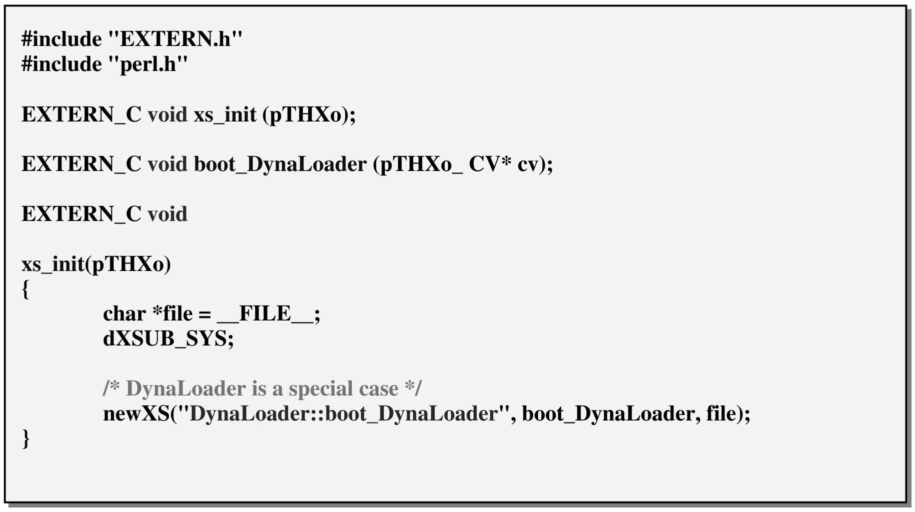
Can't load module `IO`, dynamic loading not available in this Perl. *You may need to build a new Perl executable that either supports dynamic loading or has the IO module statically linked into it.*

Your interpreter does not know how to talk to these extensions on its own. A little glue is needed to help this interface. In the following section, we will discuss an example that will be used to interface to a Web site with the help of a glue code. That example uses the `eval_pv()` function to access the code in the Perl domain.

7.3.6 Develop an Interface to Get a Web Page

In this section, we use the `eval_pv()` function call to execute some Perl scripts that need to load some C/C++ libraries to access the Internet and get a Web page's contents. Because the scripts we need to evaluate will use some Perl modules that themselves need to use a C/C++ library, a compiling error will happen. In order to solve this problem, we need a glue function or code. So first let's take care of this glue code.

Recall in the previous sections, each time before calling a Perl script from within the C/C++ domain, we used a `perl_paste()` C function to transfer and paste all



```
#include "EXTERN.h"
#include "perl.h"

EXTERN_C void xs_init (pTHXo);

EXTERN_C void boot_DynaLoader (pTHXo_ CV* cv);

EXTERN_C void

xs_init(pTHXo)
{
        char *file = __FILE__;
        dXSUB_SYS;

        /* DynaLoader is a special case */
        newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, file);
}
```

Figure 7-23

arguments and components from within the C/C++ domain to the Perl domain. If you paid attention to the second argument of that function, you realized that a NULL was used in there. This is the place we can insert a glue code or glue function. The purpose of this glue code or function is to provide an initial connection between the Perl and linked C/C++ code. We will first create this glue code and use that code to connect Perl script and C/C++ code together. Before we continue, let's first create a new project in Visual C++.

Launch Visual C++ 6.0 and create a new project with a type of `Win32 Console Application`, and enter `EmbPerl` in the `Project name`: space as the name for this project. Make sure the `Location`: is our user-defined project directory, `C:\PerlApp`. Click `OK` to open this new project.

Now let's return to handling the glue code. Open a command prompt window and switch to our current project directory, `C:\PerlApp\EmbPerl`. Enter the following code to create a file named `perlxsi.c`, which includes the glue code, `xs_init`:

```
perl -MextUtils::Embed -e xsinit
```

Press the `Enter` key when you have finished, and a new file named `perlxsi.c` is created in the current directory. Open this file and take a look at it. Figure 7-23 shows the contents of this file. You can put any extension library under the `DynaLoader` section. Generally, if your extension is dynamically loaded, `DynaLoader` creates `Module::bootstrap()` for you. You do not need to manually add that extension yourself.

Open Windows Explorer to copy the perl56.lib file from C:\Perl\lib\CORE to our project folder C:\PerlApp\EmbPerl. Now, return to Visual C++, and click Project|Add To Project|Files. . . to open the Insert Files into Project dialog box, then select the item we just copied (perl56.lib) from the list and click OK to add this file into our project. Also, add the glue file perlxsi.c, which was just created, to our current project folder. Make sure that you select All Files (*.*) from the Files of type list box in the preceding operation.

Click Tools|Options to open the Options dialog box, then click the Directories tab and select Include files from the Show directories for: list box. In the Directories list, double-click a dashed-line box to display a button on the right end, then click that button to browse the directories and to locate the folder C:\Perl\lib\CORE. Click OK to select this folder as the additional directory for our project. In a similar way, select the folder C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE as the additional directory for our project. You may need to add another include directory, C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\SYS.

In order to make the current project folder searchable, you need to add this folder as a variable to your system path on your machine. For Windows 95/98/Me, you can open the Autoexec.bat file to add this folder (C:\PerlApp\EmbPerl\Debug). For Windows NT and 2000, you can open the System icon to add your folder as an environment variable.

Now, click the File|New menu item to create a new C++ source file. Enter EmbPerl in the File name: field as the name for this source file, and click OK to open this new source file. Enter the code shown in Figure 7-24 into this file.

A. Make a declaration for our glue code. This piece of code is located in the glue file, perlxsi.c, so an EXTERN_C macro is used to tell the compiler where this code is located.

B. The perl_parse() function is called to transfer and paste all variables and components to the Perl domain. The second argument, xs_init, is the glue code and it will make the initial connection between Perl and C/C++ . Also, this will provide a connection to dynamically loaded libraries that are created from within the C/C++ domain.
The third argument is the number of evaluations we need to perform on the Perl domain. The next argument is the embed string array, which tells Perl that an evaluation will start soon.

C. After perl_run() is called, the first evaluation is executed, use LWP::Simple. The LWP, also called libwww-perl, is a massive package that contains a number of different modules that pertain to basically all things Web-related. It is most often used for its built-in HTTP client, but it also provides a lot of other Web-related features.

```
  /*****************************************************************************
   * NAME          : EmbPerl.cpp
   * DESC          : Call eval_pv() to connect web & get the contents of the web page
   * DATE          : 6/22/2002
   * PGMR          : Y. Bai
   *****************************************************************************/
  #include "EXTERN.h"
  #include "perl.h"

A EXTERN_C void xs_init (pTHXo);

  PerlInterpreter  *my_perl;

  int main(int argc,  char** argv,  char** env)
  {
          char*   embed[]     = { "", "-e", "0"  };
          char*   my_argv[] = {"", "PerlWeb.pl"};

          my_perl = perl_alloc();
          perl_construct(my_perl);

B         perl_parse(my_perl,  xs_init,  2,  embed,  NULL);
          perl_run(my_perl);
          /* load the LWP Module library */
C         eval_pv("use LWP::Simple;", TRUE);
          /* monitor the completion status */
          printf("Complete use LWP::Simple;\n");
          /* execute web connection and get the web page  */
D         eval_pv("my $content = get('http://www.jcsu.edu/dirpre.htm'); print $content;",
                                                                            TRUE);
          /* clean up environment  */
          perl_destruct(my_perl);
          perl_free(my_perl);

          return 0;
  }
```

Figure 7-24

The `Simple` is a module in the `LWP` package, and its full name is `LWP::Simple`. This module allows you to easily utilize the `HTTP` client features of `LWP`. Of course, there are some more advanced modules you can use, but those modules are more complicated and more difficult.

Here we first utilize `use LWP::Simple` to import this module; after evaluation of this string is completed, the environment variables for connecting to the Web are ready to use.

D. Next we call the `get()` function to connect to a specified Web site; here we use a directory Web page of Johnson C. Smith University, and retrieve the page contents. The retrieved Web page is assigned to a `$content` variable. A Perl function, `print`, then is used to display the contents on the screen.

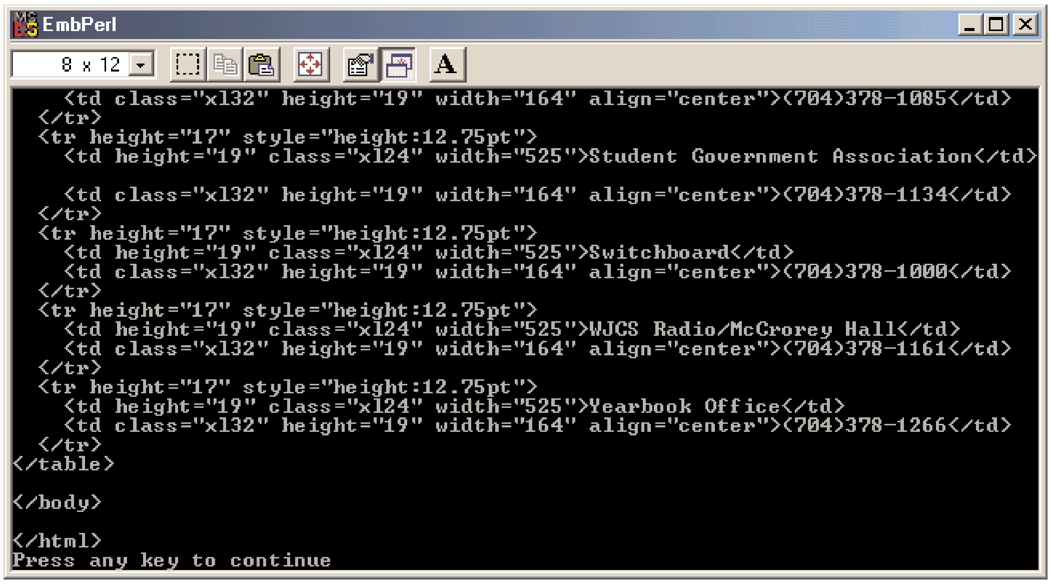
You can put any Web site in this evaluation string and you can access any Web page in this way. Sounds very good. Now let's build our project by clicking the `Build|Build EmbPerl.exe` menu item. If everything is ok, you should not encounter any errors.

Click the `Build|Execute EmbPerl.exe` menu item to run our project. A command window will pop up and, a moment later, the content of the Web page, the directory of Johnson C. Smith University, is displayed on the window, as shown in Figure 7-25.

The function `get()` will return a `undef` if it encounters any error. You can use some other modules to make your program more powerful. For example, you can use the `LWP::UserAgent` module to improve the quality of your Web-related tasks.

The `libwww-perl` collection is a set of Perl modules that provides a simple and consistent application programming interface (API) to the World Wide Web. The main focus of the library is to provide classes and functions that allow you to write WWW clients. The library also contains modules that are of more general use and even classes that help you implement simple `HTTP` servers.

Most modules in this library provide an object-oriented API. The user agent, and the information that is sent or received from the WWW server, are all represented by



```
    <td class="xl32" height="19" width="164" align="center">(704)378-1085</td>
  </tr>
  <tr height="17" style="height:12.75pt">
    <td height="19" class="xl24" width="525">Student Government Association</td>

    <td class="xl32" height="19" width="164" align="center">(704)378-1134</td>
  </tr>
  <tr height="17" style="height:12.75pt">
    <td height="19" class="xl24" width="525">Switchboard</td>
    <td class="xl32" height="19" width="164" align="center">(704)378-1000</td>
  </tr>
  <tr height="17" style="height:12.75pt">
    <td height="19" class="xl24" width="525">WJCS Radio/McCrorey Hall</td>
    <td class="xl32" height="19" width="164" align="center">(704)378-1161</td>
  </tr>
  <tr height="17" style="height:12.75pt">
    <td height="19" class="xl24" width="525">Yearbook Office</td>
    <td class="xl32" height="19" width="164" align="center">(704)378-1266</td>
  </tr>
</table>

</body>

</html>
Press any key to continue
```

Figure 7-25

objects. This makes a simple and powerful interface to these services. The interface is easy to extend and customize for your own needs.

7.3.7 Develop a Persistent Interpreter

When developing some long-running projects or applications, it is better to maintain a persistent interpreter in the C/C++ domain. In this way, we can save time in allocating and constructing a new interpreter multiple times. Another advantage of using a persistent interpreter is that we can speed up our application as it is running because we don't need to keep reloading the interpreter into memory. Instead, we only need to do that once, when we first run our project.

The problem introduced by this persistent interpreter is the conflict between the different variables and components defined in the different namespaces and packages. This is because the C/C++ code does not have any information about which piece of Perl code will be run first and which one will be run second if multiple Perl scripts are working together on one server developed in the C/C++ domain. So you have to pay special attention to this issue. Otherwise, it is very easy to cause your machine to crash.

One solution to that problem is to translate the filename of all scripts into a unique package with a unique package name. After that, we can compile all the code into that package using the `eval` function provided by Perl. As the program is running, each file can be compiled and loaded only once, as it is needed. Also, each file can be unloaded or cleaned by the instructions in the application if that file is no longer being used. We can use the `call_argv()` function to call the subroutine `Embed::LongTerm::eval_file`, which will be developed next, and pass the filename and a Boolean flag to clean up that file if it is no longer needed.

First, let's develop a Perl script, `LongTerm.pl`, which will be located in the package `Embed` with a unique name. Open NotePad and enter the code shown in Figure 7-26 into that editor. Save this file as `LongTerm.pl` at our user-defined folder `C:\PerlApp` when you have finished.

Now launch Visual C++ 6.0 and create a new project with the type of `Win32 Console Application`. Enter `LongTerm` in the `Project name`: input field as the name for this project. Make sure the `Location`: field is our user-defined directory `C:\PerlApp`.

Click the `File|New` menu item to create a new C++ source file, enter `LongTerm` in the `File name`: input field as the name for this source file, and click `OK` to open this new source file. Enter the code shown in Figure 7-27 into this file.

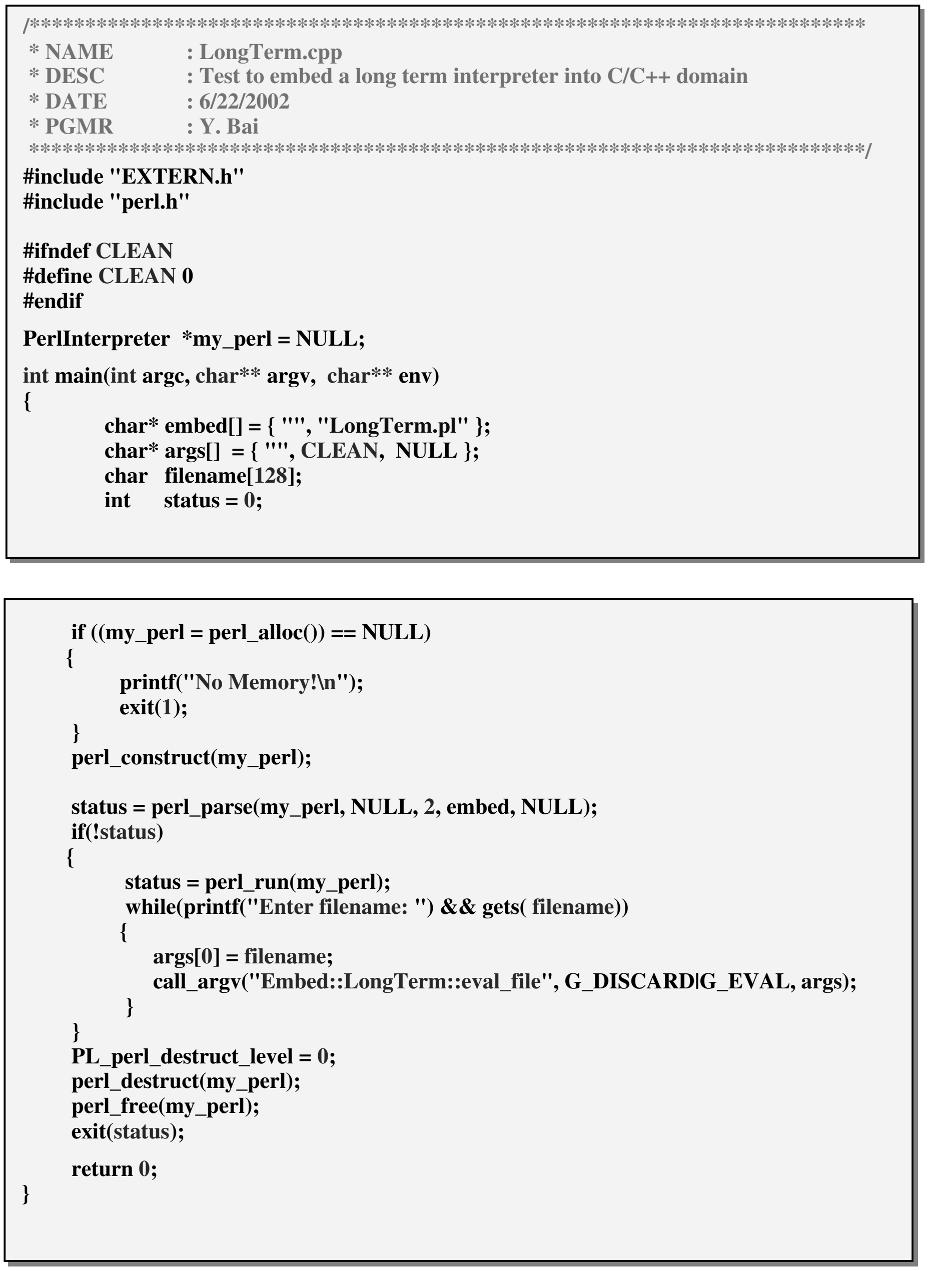
Open Windows Explorer to copy the `perl56.lib` file from `C:\Perl\lib\CORE` to our project folder `C:\PerlApp\LongTerm`. Also, copy the Perl script we developed before, `LongTerm.pl`, from the directory `C:\PerlApp` to our current project folder `C:\PerlApp\LongTerm`.

```
package Embed::LongTerm;

use strict;
our %Cache;
use Symbol qw(delete_package);

sub valid_package
{
   my($string) = @_;
   $string =~ s/([^A-Za-z0-9\/])/sprintf("_%2x", unpack("C", $1))/eg;
   $string =~ s|/(\d)|sprintf("/_%2x", unpack("C", $1))|eg;
   $string =~ s|/|::|g;

   return "Embed" . $string;
}

sub eval_file
{
   my ($filename, $delete) = @_;
   my $package = valid_package($filename);
   my $mtime = -M $filename;

   local *FH;
   open FH, $filename or die "open '$filename' $!";
   local($/) = undef;
   my $sub = <FH>;

   close FH;
```

```
   # wrap the code into a subroutine inside our unique package
   my $eval = qq{package $package; sub handler {$sub; }};
   {
      my ($filename, $mtime, $package, $sub);
      eval $eval;
   }
   die $@ if $@;

   # cache it unless we are cleaning out each time
   $Cache{$package}{mtime} = $mtime unless $delete;
   eval {$package->handler;};
   die $@ if $@;

   delete_package($package) if $delete;
}
```

Figure 7-26

Now, return to Visual C++, and click `Project|Add To Project|Files. . .` to open the `Insert Files into Project` dialog box, then select the item we just copied (`perl56.lib`) from the list and click `OK` to add this file into our project. Also,



```
/*****************************************************************************
 * NAME          : LongTerm.cpp
 * DESC          : Test to embed a long term interpreter into C/C++ domain
 * DATE          : 6/22/2002
 * PGMR          : Y. Bai
 *****************************************************************************/
#include "EXTERN.h"
#include "perl.h"

#ifndef CLEAN
#define CLEAN 0
#endif

PerlInterpreter  *my_perl = NULL;

int main(int argc, char** argv,  char** env)
{
            char* embed[] = { "", "LongTerm.pl" };
            char* args[]  = { "", CLEAN,  NULL };
            char   filename[128];
            int      status = 0;

       if ((my_perl = perl_alloc()) == NULL)
       {
              printf("No Memory!\n");
              exit(1);
       }
       perl_construct(my_perl);

       status = perl_parse(my_perl, NULL, 2, embed, NULL);
       if(!status)
       {
              status = perl_run(my_perl);
              while(printf("Enter filename: ") && gets( filename))
              {
                  args[0] = filename;
                  call_argv("Embed::LongTerm::eval_file", G_DISCARD|G_EVAL, args);
              }
       }
       PL_perl_destruct_level = 0;
       perl_destruct(my_perl);
       perl_free(my_perl);
       exit(status);

       return 0;
}
```

Figure 7-27

add the script file `LongTerm.pl`, which was created before, to our current project folder. Make sure that you select `All Files (*.*)` from the `Files of type` list box in the preceding operation.

```
#LongTest.pl

my $string = "Welcome to LongTerm";
keep($string);

sub keep
{
  printf("OK, Good-bye to: @_\n");
}
```

Figure 7-28

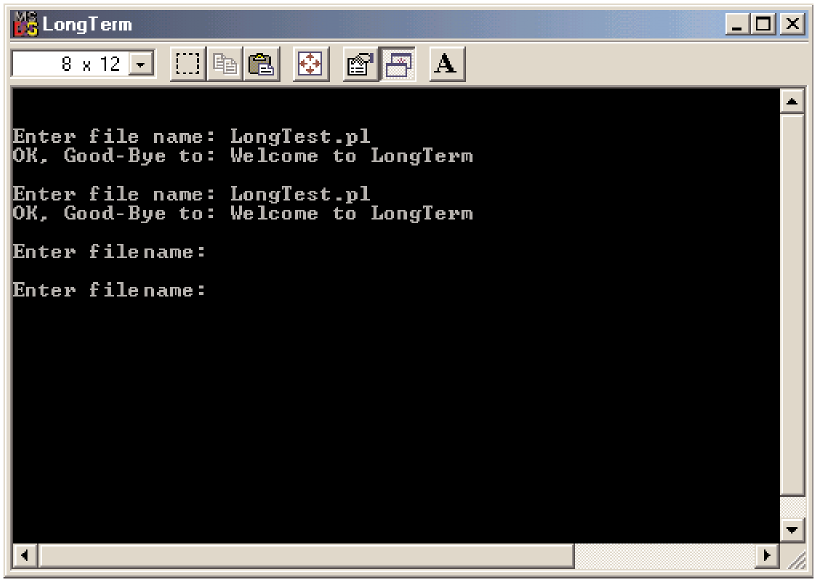
Click `Tools|Options` to open the `Options` dialog box, then click the `Directories` tab and select `Include files` from the `Show directories for`: list box. In the `Directories` list, double-click a dashed-line box to display a button on the right end, then click that button to browse the directories and to locate the folder `C:\Perl\lib\CORE`. Click `OK` to select this folder as the additional directory for our project. In a similar way, select the folder `C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE` as the additional directory for our project. You may need to add another include directory, `C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\SYS`.

Now we need to develop a test script file. Open the NotePad editor and enter the code shown in Figure 7-28 into this file. Save this file as `LongTest.pl` to our current project folder `C:\PerlApp\LongTerm`.

The purpose of this test file is to test the long-term interpreter. We can develop as many such files as we want, and save them into the Embed package with a unique name. As the program runs, we do not need to create a new interpreter and delete it when we begin another program or another file.

Now return to Visual C++, and click `Project|Add To Project|Files. . .` to open the `Insert Files into Project` dialog box, then select the test script file, `LongTest.pl`, and add it to our project. Now, let's build our project by clicking the `Build|Build LongTerm.exe` menu item from the Visual C++ domain. Everything should be ok. Then click the `Build|Execute LongTerm.exe` menu item to run our project. A command prompt window will be displayed on screen, as shown in Figure 7-29.

A prompt will ask you to enter a filename. Enter the test filename `LongTest.pl`, then press the `Enter` key. A message `"OK, Good-bye to: Welcome to LongTerm"`, is displayed on screen. This is what we expected. We created this file, and when we run the project, we do not directly call this file. Instead, we ask the Perl interpreter to call this file and run that script. The execution result of that script is to print out the welcome message in the `keep` subroutine in the Perl domain (refer to Figure 7-28).



```
Enter file name: LongTest.pl
OK, Good-Bye to: Welcome to LongTerm

Enter file name: LongTest.pl
OK, Good-Bye to: Welcome to LongTerm

Enter filename:

Enter filename:
```

Figure 7-29

When this file is completed, we can send a Boolean flag to clean it up if we do not need it anymore. The project will run continually and ask the user to enter the next filename. In this way, you can run as many as files as you want without having to load and unload the interpreter repeatedly.

Press `Ctrl+C` to terminate this project. Some users may encounter multiple interpreters running in one application. In the next section, we will discuss how to handle this issue.

7.3.8 Develop Multiple Interpreter Instances

In this section, we will discuss how to call multiple interpreters and run them at the same time. You need to use the `-DMULTIPLICITY` flag when building Perl script. Another issue is that when you install and build Perl, you have to make sure that you used options to turn on some switches to make the multiple interpreters function available; otherwise, you cannot run this situation.

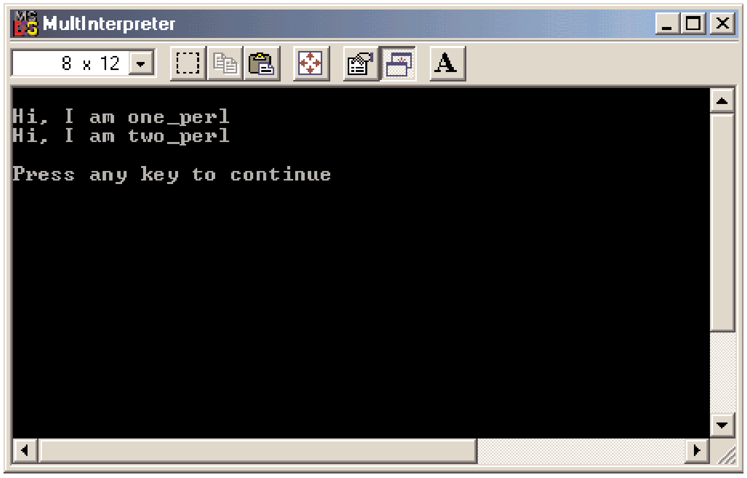
The following simple example uses two interpreters in the C/C++ domain. Launch Visual C++ 6.0 and create a new project with the type of `Win32 Console Application`, and enter `MultInterpreter` in the `Project name`: field as the name for this new project. Also, create a new C++ source file named `MultInterpreter.cpp`. Click `OK` to open that newly created source file and enter the code shown in Figure 7-30. All of the code is straightforward and has been explained in previous sections.

```
/******************************************************************************
 * NAME: MultInterpreter.cpp
 * DESC:  Test to call multiple interpreter instances from within C/C++ domain
 * DATE:  7/22/1998
 * PGMR: perlembed
 ******************************************************************************/
#include "EXTERN.h"
#include "perl.h"

PerlInterpreter *one_perl;
PerlInterpreter *two_perl;

#define HELLO "-e", "print qq(Hi, I am $^X\n)"

int main(int argc,  char** argv,  char** env)
{
        one_perl = perl_alloc();
        two_perl = perl_alloc();

        char*  one_args[] = {"one_perl", HELLO };
        char*  two_args[] = {"two_perl", HELLO };

        perl_construct(one_perl);
        perl_construct(two_perl);

        perl_parse(one_perl, NULL, 3, one_args, (char**)NULL);
        perl_parse(two_perl, NULL, 3, two_args, (char**)NULL);

        perl_run(one_perl);
        perl_run(two_perl);

        perl_destruct(one_perl);
        perl_destruct(two_perl);

        perl_free(one_perl);
        perl_free(two_perl);

        return 0;
}
```

Figure 7-30*

Build this project and click `Build|Execute MultInterpreter.exe` to run it. Two embedded interpreters will execute two print functions to show that multiple interpreters have been invoked. The running result of this project is shown in Figure 7-31. Press any key to end this project.

*Source: This program is reprinted by permission of the GNU GPL and Artistic Licenses. Available: *www.perldoc.com/perl5.6/pod/perlembed.html.*



```
Hi, I am one_perl
Hi, I am two_perl

Press any key to continue
```

Figure 7-31

7.3.9 Calling an Anonymous Perl Subroutine from C/C++

In the previous sections, we always used the `call_pv()` function to access Perl subroutines with a definite name that was enclosed in a string. In this section, we use another function, `call_sv()`, to call some anonymous subroutines defined in Perl. This method gives more flexibility to the user. By using `call_sv()`, you don't need to specify the subroutine's name clearly or explicitly; you can simply pass a reference to the subroutine, and the `call_sv()` can identify the associated subroutine for you automatically.

Open Visual C++ 6.0 and create a new project named `Anonymous` with a `Win32 Console Application` type under our user-defined project folder `C:\PerlApp`. Create a new source file named `Anonymous.cpp`, and click `OK` to open this new source file. Open Windows Explorer and copy the Perl library file `perl56.lib` from the folder `C:\Perl\lib\CORE` to our user-defined project folder `C:\PerlApp\Anonymous`.

In the Visual C++ 6.0 domain, click the `Project|Add To Project|Files . . .` menu item to open a dialog box. In the opened dialog box, select the Perl library file `perl56.lib`, and click `OK` to add this file to our current project. Enter the code shown in Figure 7-32 into the newly created source file.

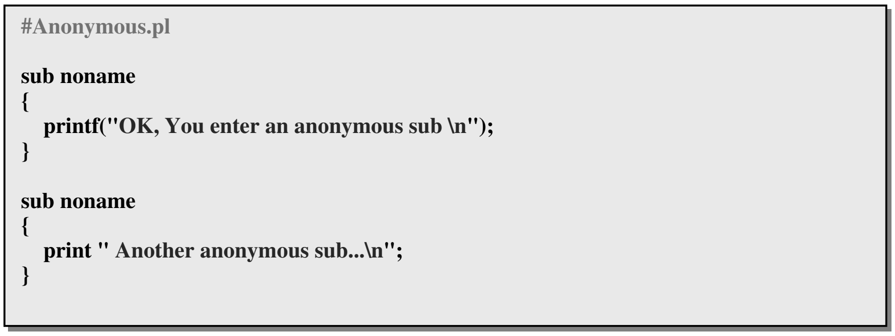
A. First, we test to use `eval_pv()` to retrieve a reference to a subroutine that has no name on it. This one-line subroutine prints a message to show that it has been called. After we have obtained the reference to that anonymous subroutine, we can easily access that subroutine by using the `call_sv()` function. The name variable is a reference with a type of `SV*`. This call has no returned

```
/*****************************************************************************
 * NAME          : Anonymous.cpp
 * DESC          : Test to use call_sv() to call anonymous subroutine from C/C++
 * DATE          : 6/23/2002
 * PGMR          : Y. Bai
 *****************************************************************************/
#include "EXTERN.h"
#include "perl.h"

PerlInterpreter  *my_perl;

int main(int argc,  char** argv,  char** env)
{
            SV*     name;
            char*   embed[]    = { "", "-e", "0" };
            char*   my_argv[] = {"", "Anonymous.pl"};

            my_perl = perl_alloc();
            perl_construct(my_perl);

            perl_parse(my_perl, NULL, 3, embed, NULL);
            perl_run(my_perl);

            /* call eval_pv() to get the reference to subroutine */
A           name = eval_pv("sub { printf('This is an anonymous sub...\n\n');} ", TRUE);
            printf("\n");
            call_sv(name, G_VOID|G_NOARGS);

            /* call eval_pv() to get a name from two subs with a same name */
B           perl_parse(my_perl, NULL, 2, my_argv, (char** )NULL);
            perl_run(my_perl);
            name = eval_pv("noname", TRUE);
            printf("\n");
            call_sv(name, G_VOID|G_NOARGS);

            perl_destruct(my_perl);
            perl_free(my_perl);

            return 0;
}
```

Figure 7-32

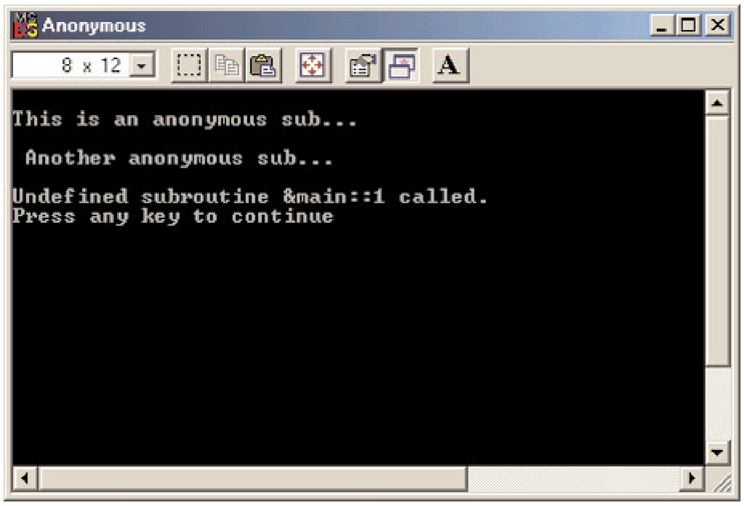
item and no arguments, so a combination of flags, G_VOID|G_NOARGS, is used to pass it into the Perl domain.

B. Here we want to make a test for a Perl script that has two subroutines with the same name. In order to do that, we first create a Perl script named Anonymous.pl, which is shown in Figure 7-33.
One printing function is used for each subroutine. Two subroutines have the same name, noname. Save this script to our current project folder C:\PerlApp\



```
#Anonymous.pl

sub noname
{
   printf("OK, You enter an anonymous sub \n");
}

sub noname
{
   print " Another anonymous sub...\n";
}
```

Figure 7-33



```
This is an anonymous sub...

 Another anonymous sub...

Undefined subroutine &main::1 called.
Press any key to continue
```

Figure 7-34

`Anonymous`. First, we use the `perl_paste()` function to transfer and paste arguments to the Perl domain, which includes the Perl script's name. Then we use `eval_pv()` to identify and pick up the subroutine name. Finally, we use `call_sv()` to access this subroutine and execute it.

Now build our project by clicking the `Build|Build Anonymous.exe` menu item, then click `Build|Execute Anonymous.exe` to run our project. A command window will be displayed on screen and the running results will look like the screen shown in Figure 7-34.

The first calling of the anonymous subroutine has no problem and the default printing function is executed and displayed on the screen, `"This is an anonymous sub . . ."`. But the second calling of two anonymous subroutines encounters a problem, as shown in Figure 7-34. Refer to Figure 7-33 for the Perl script `Anonymous.pl`. There are two subroutines in this script, and both subroutines have the same name, `noname`. According to the execution result, the second subroutine is called and the `print` function is executed. `"Another anonymous sub . . ."` is displayed to show this result. Following the result, a warning message is displayed: `"Undefined subroutine &main::1 called"`. The system cannot find the desired subroutine based on the information provided by the user when this subroutine was called, so the system picks up the default subroutine, the first subroutine (this first is based on the stack order), which is, in actuality, the second subroutine in the script. That is the reason why we obtain the result in Figure 7-34. Press any key to end this project.

The second section of this example is not closely related to the anonymous subroutine calling from within the C/C++ domain, but it provides a good illustration of how the system responds to the anonymous calling when multiple anonymous subroutines exist.

7.3.10 Call a Perl Method from Within the C/C++ Domain

In the previous sections, we tested different calling functions that access Perl scripts from within the C/C++ program. In this section, we want to show readers how to call a Perl method from the C/C++ domain. This is very similar to an OOP methodology. The following example shows readers how to call a constructor and a class method defined in the Perl script.

Open the NotePad editor and enter the code shown in Figure 7-35 into the editor. Save this file as `Rand.pl` to our user-defined project folder `C:\PerlApp`. The class

```
{
    package  Rand;

    sub new                            // constructor
    {
            my($type) = shift;
            bless [@_]
    }

    sub RandID                         // class method
    {
            my ($class) = @_;
            print "This is Class $class version 1.0\n";
    }
}
```

Figure 7-35

name is defined as `Rand`, and the constructor is defined as a subroutine with a `new` keyword. You can use any legal name as your class name, but you should use a `new` keyword before the constructor. The class method is defined as another subroutine `RandID`. The argument is `$class`.

Now, open Visual C++ 6.0 and create a new project with a type of `Win32 Console Application`, and enter `CallMethod` in the `Project name`: input field as the name for this project. Click `OK` to open this new project. Click `File|New` to generate a new C++ source file named `CallMethod.cpp`; click `OK` to open this new source file. Enter the code shown in Figure 7-36 into this source file.

A. Two functions (consider them class methods from OOP's point of view), `InitClass()` and `RandClass()`, are declared at the beginning of this file. We will use these two functions to access the methods defined in the Perl scripts later. The first function is used to access the constructor to create a new object of `Rand`. This constructor needs three character arguments. The second function is used to access the class method `RandID`, which has one integer argument.

B. After everything is set up, we call these two functions to access the constructor to generate a new object and then to call the class method to print something to confirm that we have executed that call successfully. For the constructor, three character strings, `"max"`, `"min"`, and `"mean"` are passed to the constructor. The class method has an integer value of `1` as the argument, which is passed to that method in the Perl domain.

```
  /*****************************************************************************
   * NAME         : CallMethod.cpp
   * DESC         : Test to call a perl method from within C/C++
   * DATE         : 6/23/2002
   * PGMR         : Y. Bai
   *****************************************************************************/
  #include "EXTERN.h"
  #include "perl.h"

  PerlInterpreter  *my_perl;

A static void InitClass(char* a, char* b, char* c);
  static void RandClass(int  n);

  int main(int argc,  char** argv,  char** env)
  {
          char*   my_argv[] = {"", "Rand.pl"};

          my_perl = perl_alloc();
          perl_construct(my_perl);
```

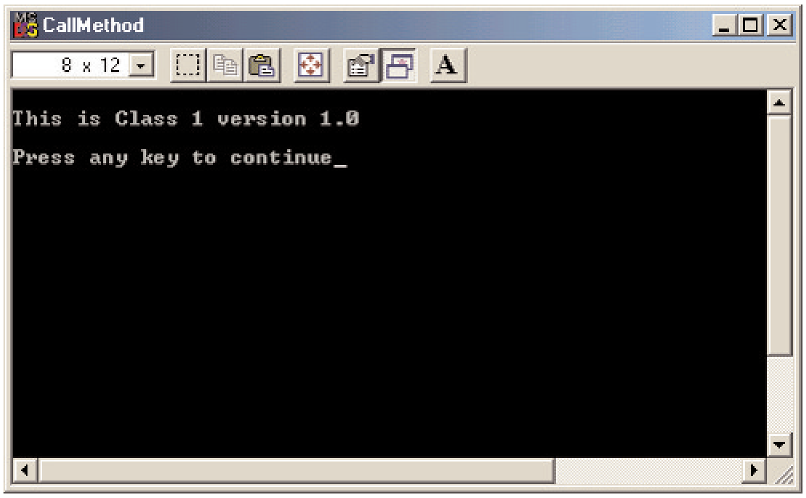
Figure 7-36

```
            perl_parse(my_perl, NULL, 3, my_argv, NULL);
            perl_run(my_perl);

B           InitClass("max", "min", "mean");
            RandClass(1);

            perl_destruct(my_perl);
            perl_free(my_perl);

            return 0;
    }
C   static void InitClass(char* a, char* b, char* c)
    {
            dSP;
            PUSHMARK(SP);
            XPUSHs(sv_2mortal(newSVpv(a, 3)));
            XPUSHs(sv_2mortal(newSVpv(b, 3)));
            XPUSHs(sv_2mortal(newSVpv(c, 3)));
            PUTBACK;
            call_pv("Rand::new", G_DISCARD);
            PUTBACK;
    }
D   static void RandClass(int  n)
    {
            dSP;                                    // initialize the perl stack pointer
            ENTER;                                  // starting perl from here...
            SAVETMPS;                               // temporary variable
            PUSHMARK(SP);                           // save the stack pointer
            XPUSHs(sv_2mortal(newSViv(n)));         // push the number into the stack
            PUTBACK;                                // make local stack pointer to global
            call_pv("Rand::RandID", G_SCALAR);
            SPAGAIN;                                // refresh the stack pointer
            printf("\n");
            PUTBACK;
            FREETMPS;                               // free returned value
            LEAVE;                                  // XPUSHed "mortal" args
    }
```

Figure 7-36 (*continued*)

C. In the `InitClass()` function, a C-function `call_pv()` is called with the first argument `Rand::new` as the method's name, which is defined in the Perl class `Rand`. This is very similar to a traditional C++ calling strategy. You can directly use the class name following the constructor's name because the constructor can be considered a special class method (no returned value). So a `G_DISCARD` flag is used to ignore the returned value.

D. Similarly, inside the function `RandClass()`, another `call_pv()` is called to access the class method `RandID`. An integer value is passed to this method,



```
This is Class 1 version 1.0
Press any key to continue_
```

Figure 7-37

so a `G_SCALAR` flag is used here to tell Perl that this call has a scalar value passed to it, and that the value has been pushed into the Perl stack area by using the `XPUSHs` macro in the previous section of this function implementation.

The entire coding for this project is straightforward and all statements have already been discussed in previous sections. But, before we can build this project, we need to perform the following tasks:

- Open Windows Explorer to copy the Perl script `Rand.pl`, which was developed before, from the folder `C:\PerlApp` to our newly created project folder `C:\PerlApp\CallMethod`.
- Copy the Perl library file `perl56.lib` from the Perl system directory `C:\Perl\lib\CORE` to our new project folder `C:\PerlApp\CallMethod`.
- In the Visual C++ 6.0 domain, click the `Project|Add To Project|Files . . .` menu item to open a dialog box. In the opened dialog box, select the copied files, `perl56.lib` and `Rand.pl`. Then, click `OK` to add these files to our current project.

Now, click the `Build|Build CallMethod.exe` menu item to build our project. Then click `Build|Execute CallMethod.exe` to run it. A command window will be displayed on the screen, and the running results will look like those shown in Figure 7-37.

A message, `"This is Class 1 version 1.0"`, is displayed in the window. This message is located inside the class method `RandID`, and it is activated by a function call `call_pv()` from within the C/C++ domain. Everything looks good, and we have finished the development of a program to call Perl methods from within the C/C++ domain.

All example projects developed in these sections (7.3.2–7.3.10) can be found on the accompanying CD-ROM in the `Chapter 7\PerlApp` folder. All project files, including the C/C++ source files, object files, executable files, Perl script files, and Perl library files, are stored under this folder. If you want to develop an independent project, you need to copy all Perl system header files, such as `EXTERN.h` and `perl.h`, to your project folder, and then, by using the `Project|Add To Project|Files` menu item in the Visual C++ window, add all these header files to your project.

To make things simple, in all of the projects we developed in this chapter, we only include those header files by using the `Tools|Options` menu item in Visual C++, without copying them to our project. So all header files still stay in their original Perl system directory.