Test your code and save programming time with C and C++ Interpreters

Interpretation

Many programming dialects advertise the convenience of an interpreted language with the power of C and C++. But if you really want interpreted C and C++, why not just use a C/C++ interpreter? A few C/C++ compilers support interpreted processing for faster coding and some interesting debugging options. *By Ankur Kumar Sharma*

C and C++ are compiled languages: The source code is translated to the machine code of the hardware platform in a complete binary file. To run any C or C++ program, you first need to write the source code, then compile and link the code to create an executable. Every time you make even a small change, you have to repeat this code/modify/compile/link/run cycle. For small programs, the compile/link step is trivial, but for even a medium-size program, this step is very time consuming and boring. A large C or C++ program can take several hours to compile and link. This process is especially annoying if you are a beginner who is prone to small errors, or even if you are a seasoned veteran who is interested in rapid prototyping.

In contrast, scripting languages like Python, Ruby, and Perl don't require a compile/link step. Scripting languages also provide the opportunity to experiment by typing program code into the interactive shell to see the results. In many cases, however, C and C++ are still better long-term solutions for compatibility and performance reasons.

As you might guess, having some way to run your C/C++ code directly – without the compile/link step – in an interactive shell can provide significant benefits for testing and prototyping. The open source community has responded to this call by providing several options for running C/C++ code in an interpreted environment.

IN THE LAB

I used Puppy Linux 5.0 and Ubuntu 9.10 64-bit desktop edition to test the C and C++ code mentioned in this article.

Interpret Modes

C and C++ interpreters can work in either *interactive mode* or *batch mode*. Interactive mode is like programming shells of various scripting languages: You type C and C++ program statements at the interpreter prompt one at a time and see the corresponding execution results. In batch mode, you run single or multiple C/C++ source files through an interpreter. Interactive mode is more suitable for experimentation with short code snippets, and batch mode is better when the source code is long or organized in multiple files.

Tcc, A Compiler with Direct C Code Run

Tcc [1], which stands for Tiny C Compiler, is available for both 32- and 64-bit target platforms. This tiny dynamo, known as a fast and efficient C compiler with a small footprint, is heading toward full ISO99 compliance and can run any C code directly without any compile/link step. Tcc comes pre-installed as the default C compiler in Damn Small Linux. It

```
File  Edit  View  Terminal  Help
richnusgeeks@ankur-laptop:~/Documents/Articles/C_Interpreters/LM$ tcc stack.c -run teststack.c
stack.c:38: warning: assignment from incompatible pointer type
stack.c:76: warning: assignment from incompatible pointer type
stack.c:96: warning: assignment from incompatible pointer type
##################
1. Push an integer,
2. Pop an integer,
3. Show the stack,
4. Exit.
##################
Enter a choice : 1
Enter the integer to push : 235678
##################
1. Push an integer,
2. Pop an integer,
3. Show the stack,
4. Exit.
##################
Enter a choice : 1
Enter the integer to push : 34567889
##################
1. Push an integer,
2. Pop an integer,
3. Show the stack,
4. Exit.
##################
Enter a choice : 3
1 :  34567889
2 :  235678
##################
1. Push an integer,
2. Pop an integer,
3. Show the stack,
4. Exit.
##################
Enter a choice : 2
The integer popped : 34567889
##################
1. Push an integer,
2. Pop an integer,
3. Show the stack,
```

Figure 1: Interpretation of multifile C code through Tcc.

```
File  Edit  View  Terminal  Help
richnusgeeks@ankur-laptop:~/Documents/Articles/C_Interpreters/LM$ cat testpicoc.c

char *sLP64  = " The platform is 64 bit.";
char *sILP32 = " The platform is 32 bit.";

int i = sizeof(int);
int l = sizeof(long);

if(l != i)
    printf("%s\n", sLP64);
else
    printf("%s\n", sILP32);

printf(" strlen(sLP64)  : %u\n", strlen(sLP64));
printf(" sizeof(sILP32) : %u\n", sizeof(sILP32));

richnusgeeks@ankur-laptop:~/Documents/Articles/C_Interpreters/LM$ picoc testpicoc.c
 The platform is 64 bit.
 strlen(sLP64)  : 24
 sizeof(sILP32) : 8
richnusgeeks@ankur-laptop:~/Documents/Articles/C_Interpreters/LM$
```

Figure 2: PicoC running in batch interactive mode.

doesn't have an interactive shell, but it can run C code organized in single or multiple C source files.

To build and install Tcc, the prerequisite is GCC. To begin, download the latest Tcc source tarball from the project page, then issue the following commands in a text console:

```
tar zxvf tcc-version.tar.bz2
cd tcc-version
```

Next, issue the following commands to build and install Tcc:

```
./configure
make
sudo make install
```

If everything goes well, typing `tcc` will bring up some help text. To test the direct C code run feature, run the code in Listing 1 by typing the command `tcc -run hellotcc.c`. Tcc can also run any C code contained in a single file directly as a script if you add the line `#! location_of_tcc/tcc -run` at the top of the source file and make the source file executable with the command `chmod u+x filename.c`.

Tcc can read code from the standard input instead of from a source file. Type the following in a text console to see this feature of Tcc in action:

```
echo 'main(){printf("\n"); ➥
      system("uname -a"); ➥
      printf("\n");}' | tcc -run -
```

To run C code organized in multiple files, the `-run` option could come after one or more files in the list. For instance, you could type `tcc stack.c -run teststack.c` or `tcc teststack.c stack.c -run`, but Tcc throws an error if you type `tcc -run teststack.c stack.c`.

Figure 1 shows the output produced by Tcc running C code directly. The files used for this example are available at the *Linux Magazine* website [2]. Note that Tcc is indicating some warnings even in the direct code run mode. So you can realize the convenience and saving in efforts when running both trivial as well as complicated C programs through Tcc's direct code run feature.

LISTING 1: hellotcc.c

```
01 #include <stdio.h>
02
03 int main(int argc, char *argv[])
04 {
05
06     printf("\n Hello to FOSS from
               tcc!!!\n");
07     return 0;
08
09 }
```

PicoC, a Minimal Interactive C Code Interpreter

PicoC [3] is a very small footprint interactive C code interpreter. It offers few constructs of the C language and is mainly targeted toward small devices like embedded systems. Still, I found PicoC useful for simple to moderately difficult C code interpretation. The constructs supported by PicoC are built in as commands, so you don't have to include standard header files while working with this tiny compiler.

To build PicoC from source code, you need GCC. To build PicoC, download the latest source tarball from the project page and issue the following commands:

```
tar jxvf picoc-version.tar.bz2
cd picoc-version
```

Now type `make` to build PicoC, and, optionally, `make test` to run the tests that come with the package. If compilation completes with no errors, you should see a `picoc` executable in the source directory. You have to copy PicoC to any of the standard locations, like `/usr/local/bin` or add the absolute path of the directory to `.bashrc`:

```
export PATH=path_of_picoc_directory:$PATH
```

LISTING 2: testpicoc.c

```
01 char *sLP64  = " The platform is 64
                  bit.";
02 char *sILP32 = " The platform is 32
                  bit.";
03
04 int i = sizeof(int);
05 int l = sizeof(long);
06
07 if(l != i)
08     printf("%s\n", sLP64);
09 else
10     printf("%s\n", sILP32);
11
12 printf(" strlen(sLP64)  : %u\n",
          strlen(sLP64));
13 printf(" sizeof(sILP32) : %u\n",
          sizeof(sILP32));
```

The `tests` subdirectory in the PicoC source directory contains many examples of the C language constructs supported by PicoC. To get a taste of the interactive mode functionality, type `picoc -i` in a text console and enter the code in Listing 2 at the PicoC prompt. Or, to interpret the file in batch mode, type `picoc testpicoc.c`. Figure 2 shows a screenshot of PicoC running in batch mode.

EiC, an Extensible Interactive C Code Interpreter

EiC [4] stands for Extensible Interactive C. It is an interactive C code interpreter that can communicate with external applications. EiC provides some very useful features, like pointer safety, and some extra C programming constructs that are not found in other C code interpreters.

Download the source tarball from the project page and issue the following commands:

```
tar zxvf EiCsrc-version.tar.gz
cd EiC-version
```

The following builds and installs EiC:

```
./config/makeconfig
make install
```

If everything goes well, add the following EiC-specific setting in `.bashrc` to finalize the configuration:

```
EICBASE=path_of_EiC_directory
export HOMEofEiC=$EICBASE
```

Start EiC by typing `eic` in a text console; by default, it starts in interactive mode, where you can type various C code statements and see the corresponding results instantly. (Type `eic -h` to view some text-based help.)

EiC can run in a scripting mode in addition to the interactive and batch modes. Scripting mode lets you write a C program as you would a shell script; you can put various C statements in a file with or without a `main()` function. To get a feeling for EiC scripting mode, run the code shown in Listing 3 by making the file executable with the command `chmod u+x testscript.eic` and then typing `./testscript.eic` in a text console.

In Listing 3, note `#include stdio.h`. With EiC you can alternatively include the headers without angle brackets.

Now I'll hack into some unique features not found with other C code interpreters. EiC can memorize all the C code statements entered on its interactive prompt. Actually, EiC creates a file called `EiChist.lst`, where it stores all the commands that were interpreted without error. By default, this file is created again in the directory every time EiC is started in the interactive mode, but you can alter this behavior by launching EiC with the `-n` switch. If you launch EiC with the `-r` switch, it is initialized by interpreting all the commands previously recorded in `EiChist.lst`. To reinitialize EiC without wiping out the old `EiChist.lst` file, you can also combine these two switches with `-nr`. Using this feature of EiC, you can continue from the state you left last time. Edit `EiChist.lst` manually to start EiC from a customized state, or start EiC in interactive mode with the `-R` switch. EiC asks about re-entering every command from `EiChist.lst` by choosing either `Y` (yes), `N` (no), or `E` (edit). You can drop or edit previous C code statements entered at the EiC interactive prompt one by one.

Another distinct feature of EiC is that it is *pointer safe*. Anytime you try to violate a stack or heap memory limits through pointers, EiC catches them and throws errors. Enter the C statements shown in Listing 4 in interactive mode, interpret `pointertest.c` in batch mode, or run the file in scripting mode and see how EiC throws errors and stops on the first pointer violation.

Before building and distributing a final version, you can verify your C programs against various memory limits through EiC's pointer safety feature. By doing this, you will save a lot of time debugging very hard-to-find and hard-to-fix memory and pointer bugs. The pointer safety feature of EiC is turned on by default, but you can turn it off feature with the modifier `unsafe`. So, if you replace `int *pPtr` by `int * unsafe pPtr` in the `pointertest.c` file, EiC proceeds without any errors regarding `pPtr[79]`. The unsafe pointer feature in EiC is useful when you interact with external, compiled C components.

CINT, a Heavyweight C and C++ Interpreter

Now comes the big daddy of all the C interpreters. CINT is the C and C++ code interpretation component of the object-oriented data analysis package ROOT [5], although it can be used as a standalone application. According to the CINT man page, CINT covers a whopping 95%

LISTING 3: testscript.eic

```
01 #! /usr/local/bin/eic -f
02
03 #include stdio.h
04
05 unsigned mystrlen(char *sStr) {
06
07     unsigned l;
08
09     for(l=0; *sStr; ++sStr, ++l);
10
11     return l;
12
13 }
14
15 void mystrcpy(char *sSrc, char
                *sDst) {
16
17     while(*sDst++ = *sSrc++);
18
19 }
20
21 int main()
22 {
23
24     char *sSrc = "This is source
                    string.";
25     char sDst[] = "This is
          destination string.";
26
27     printf("\n Before mystrcpy()
              =>\n");
28     printf(" sSrc : %s\n", sSrc);
29     printf(" sDst : %s\n", sDst);
30
31     if(mystrlen(sDst) >
          mystrlen(sSrc))
32         mystrcpy(sSrc, sDst);
33
34     printf("\n After mystrcpy()
              =>\n");
35     printf(" sSrc : %s\n", sSrc);
36     printf(" sDst : %s\n", sDst);
37
38     return 0;
39
40 }
```

of ANSI C and 90% of C++ features. CINT also provides features similar to those of GDB to debug the interpreted source code. CINT scripts have the ability to communicate with compiled components and external applications.

If you don't mind the overhead of many extra components, you can install CINT by typing `sudo apt-get install root-system` in a text console. Then, type `cint -help`, and you should see some help text.

Alternatively, to build and install the latest version of CINT from source, install the Readline library and sources with the command:

```
sudo apt-get install libreadline-dev
```

Next, download the latest source tarball of ROOT from its download page [6] and issue the following commands:

```
tar -zxvf root-version.source.tar.gz
cd root/cint
```

Finally, to build CINT, enter:

```
./configure
make
```

If the compilation goes well, you should see executables like `cint` and `makecint` in the `bin` subdirectory. To set executable and library search paths, as well as CINT-specific settings, you should add the following environmental variables in `.bashrc`:

```
CINTBASE=<path of cint directory>
export CINTSYSDIR=$CINTBASE
export PATH=$CINTBASE/bin:$PATH
export LD_LIBRARY_PATH=➥
    $CINTBASE/lib:$LD_LIBRARY_PATH
export MANPATH=➥
    $CINTBASE/doc:$MANPATH
```

Now type `sudo ldconfig` in a text console to configure dynamic linker runtime bindings.

To see CINT in action, create a file with the contents shown in Listing 5 then type `cint hellocint.cpp` in a text console. To interpret C and C++ code organized in multiple files, you have to provide CINT with a list of source files; the last file in the list must contain the `main()` function.

To run the stack example described earlier, type `cint stack.c test.c`. If CINT can't find a `main()` function or any other error, it starts an interactive session, in which you can issue various CINT commands.

To use the interpreted code debugging feature of CINT, put a breakpoint on a desired function and use the `-b` option along with the list of source code files. Now examine or change the program execution state through various debugging options mentioned in CINT the man page. You can also see the debugging options by entering `h` at the CINT interpreter prompt.

LISTING 4: pointertest.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int main()
05 {
06
07     int * pPtr, iArray[100];
08
09     printf(" pPtr : 0x%0x\n", pPtr);
10     printf(" iArray[20] : %d\n", iArray[20]);
11
12     pPtr = malloc(57);
13
14     pPtr[79]    = 7;
15     printf(" pPtr[79] : %d\n", pPtr[79]);
16
17     iArray[-8] = 6;
18     printf(" iArray[-8] : %d\n", iArray[-8]);
19
20     return 0;
21
22 }
```

Beyond C and C++ Interpretation

You can use the functions provide by EiC to embed the functionality for C code interpretation in an external C application. Run a C source file with the function `EiC_run(int argc, char **argv)`, where `argc` represents the number of arguments passed and `argv` represents an array of strings consisting of a C source file name along with other command-line arguments. To run a C code file named `myeic.c` use:

LISTING 5: hellocint.cpp

```
01 #include <iostream>
02 using namespace std;
03
04 int main()
05 {
06
07     cout << endl
08          <<" Hello to FLOSS from
               CINT!!!"
09          << endl;
10
11     return 0;
12
13 }
```

```
char *argv = {"myeic.c", ...};
int argc   = sizeof(argv)/➥
             sizeof(char *);
EiC_run(argc, argv);
```

Also, you can pass C or preprocessor commands to EiC with the function `EiC_parseString(char *command, ...)`. To build apps with these functions, include `eic.h`, found in the `include` subdirectory

LISTING 6: mycint.hpp

```
01 #ifndef MYCINT_HPP
02 #define MYCINT_HPP
03
04 #include <iostream>
05 using namespace std;
06
07 class CMakeCintDemo {
08
09     private:
10
11         int iState;
12
13     public:
14
15         CMakeCintDemo();
16
17         CMakeCintDemo(int iValue);
18
19         ~CMakeCintDemo();
20
21         void getState() const;
22
23 };
24
25 #endif
```

of the EiC source directory and link with the `libeic` and `libstdClib` libraries in the `lib` subdirectory of the EiC source directory. To play more with the embedding capabilities of EiC, follow `embedEiC.c`, which is found in the `main/example` subdirectory of the EiC source directory.

CINT is extensible through external functionalities that are coded in C and C++, which means you can create customized versions of CINT that contain your external C and C++ functionalities as added features. This feature of CINT should be used to interpret source code with custom extensions, without the need to supply headers and additional source files.

To embed your external functionalities in CINT, you need Makecint. Makecint is an *interpreter compiler* that is built during the build process for CINT. Makecint automates the process of embedding external functionalities coded in C and C++ implementations, and it generates the necessary wrapper code automatically to create your customized version of CINT.

To try your luck with this powerful feature of CINT, create the files shown in Listings 6 and 7 and issue the following command in a text console:

```
makecint -mk Makefile -o mycint ➥
         -H mycint.hpp ➥
         -C++ mycint.cpp
```

The `-mk` switch sets the name of the Makefile generated by Makecint, `-o` sets the name of the customized version of CINT, and `-H` and `-C++` are switches that let you mention the headers and sources containing the external functionalities you want to embed in CINT. For more information about the various options supported by Makecint, consult the man page.

If you examine the directory listing, you can see various additional source and object files created by Makecint to build your customized version of CINT. Now, to build the customized interpreter, just type `make` in the text console to create an executable `mycint` in the current directory.

LISTING 7: mycint.cpp

```
01 #include "mycint.hpp"
02
03 CMakeCintDemo::CMakeCintDemo() {
04
05   cout << endl
06        << " CMakeCintDemo::CMakeCint
             Demo():this = "
07        << hex
08        << this
09        << "."
10        << endl;
11
12 }
13
14 CMakeCintDemo::CMakeCintDemo(
     int iValue):iState(iValue) {
15
16   cout << endl
17        << " CMakeCintDemo::CMakeCint
             Demo(int iValue):this = "
18        << hex
19        << this
20        << dec
21        << ", iValue = "
22        << iState
23        << "."
24        << endl;
25
26 }
27
28 CMakeCintDemo::~CMakeCintDemo() {
29
30   cout << endl
31        << " CMakeCintDemo::~CMake
             CintDemo():this = "
32        << hex
33        << this
34        << "."
35        << endl;
36
37 }
38
39 void CMakeCintDemo::getState()
        const {
40
41   cout << endl
42        << " CMakeCintDemo::
             getState():this = "
43        << hex
44        << this
45        << dec
46        << ", iState = "
47        << iState
48        << "."
49        << endl;
50
51 }
52
```

LISTING 8: testmycint.cpp

```
01 int main()
02 {
03
04     CMakeCintDemo cMCT1, cMCT2(13);
05
06     cMCT2.getState();
07
08 }
```

To test your newly created customized version of CINT, run the code shown in Listing 8 by typing `./mycint testmycint.cpp` in the text console.

Your external functionalities are now included in the code interpreter itself.

Conclusion

C and C++ interpreters provide a big productivity boost. These interpreters are very helpful for C and C++ education, as well as for quick prototyping and experimentation with C and C++ programs.

EiC and CINT go beyond mere interpretation and provide extra functionalities, such as tracing down the memory limit violations at the coding stage, embedding C code interpretation into external applications, and communicating with compiled C and C++ applications. All these code interpreters can make C and C++ development more productive, more flexible, and more enjoyable. ■■■

INFO

[1] Tcc homepage: *http://bellard.org/tcc/*

[2] Code for this article: *http://www.linux-magazine.com/Resources/Article-Code*

[3] PicoC project page: *http://code.google.com/p/picoc/*

[4] EiC project page: *http://eic.sourceforge.net*

[5] ROOT: *http://root.cern.ch/*

[6] ROOT download page: *http://root.cern.ch/drupal/content/downloading-root*

AUTHOR

Ankur Kumar Sharma is a software developer and researcher who likes to play and croon classic rock songs on his guitar. He also enjoys reading self-help books, writing, and exploring all the interesting things in life. He blogs at *http://www.richnusgeeks.com*.