C++ Primer, Fifth Edition

GCC 4.7.0
Code Distribution README

Barbara E. Moo
bmoo@att.net

August 9, 2012

Overview

This distribution contains the source code of all the complete programs and many of the program fragments from *C++ Primer*. The code in this distribution will work with the GCC 4.7.0 compiler or later. If you have an earlier release of the GNU compiler, some files in this distribution will not compile. For earlier compilers, please use the `GCC pre C++11 distribution`.

A few C++11 features are not yet implemented in GCC 4.7.0. Where possible, we have provided workarounds for these missing C++11 features. Please see `CompilerNotes.pdf` in this directory for more information on the missing features and workarounds.

Conditional Compilation

We have written the code so that it will work with the current and future releases of the compiler. Source files that require an unimplemented feature use a series of preprocessor macros (see Section 2.6.3) to detect whether the compiler supports a particular feature. If the feature is not yet implemented, the sample programs use a workaround. If there is no plausible workaround, the code compiles and generates an executable file that does nothing.

Briefly, those files requiring a workaround use preprocessor variables and `#ifdef` directives decide which code to compile. For example:

```
#ifdef VARIABLE_NAME
     code1 //  this code will be compiled if VARIABLE_NAME has been defined
#else
     code2 //  this code will be compiled if VARIABLE_NAME has not been defined
#endif
```

If there is a `#define` for `VARIABLE_NAME`, then `code1` will be compiled and `code2` is skipped. Otherwise, `code1` is skipped and `code2` is compiled. We have written the code so that the code in `code1` is the valid C++11 version of the code and `code2` contains the workaround.

Section 6.5.3 covers conditional compilation in more detail.

The preprocessor variables that correspond to unimplemented features are defined in `Version_test.h`. Each source file that uses an unimplemented feature includes this header.

When new versions of the compiler are released, we will update this header on the book's website, http://www.informit.com/title/0321714113.

Building Executables

The code is divided into 19 subdirectories corresponding to the Chapters in *C++ Primer*. Each subdirectory contains a makefile that makes the source in that directory. These makefiles rely on the file named `GNU_makefile_template` in the top-level directory. The makefiles are fairly simple and we have provided comments in the hope that even those who are not familiar with makefiles can understand how to compile these programs by hand if so desired.

The top level directory also has its own `makefile` that will make the entire source tree. The top level `makefile` also has targets `clean` and `clobber` to remove the object files or object and executable files respectively.

To use make on most UNIX based operating systems you invoke the command named `make`:

```
# UNIX machines
$ make               # compiles all the programs
$ make clean         # removes all the object files and stackdumps
$ make clobber       # removes executable, object and stackdump files
```

Input and Output

The code in these subdirectories includes all of the complete programs covered in *C++ Primer*.

In addition, we include executable versions of some of the otherwise incomplete program fragments. In general, such programs print details about the internal state of the program. To understand the output, you will have to understand the program. This is intentional.

The input, if any, to these programs varies:

- Some programs print a prompt and wait for input on the standard input
- Other programs read the standard input but do not print a prompt
- Others take one or more arguments specifying a file name to read
- Yet others take a file name argument and read the standard input

Each Chapter subdirectory contains a `README` that explains the input, if any, expected by each executable file.

Sample Data Files

For those programs that expect input, we have provided small, sample input files. Input files are always found in a subdirectory named `data`. For example, the program `add_item` in directory `1` reads `Sales_item` transactions from the standard input but does not prompt for input. If the program is invoked:

```
add_item
```

it will wait for input, which you can provide by typing appropriate transactions. Alternatively, you can pass the data file we provide. Assuming the following is executed in the directory named `1` writing

```
add_item < data/add_item  # UNIX
```

will bind the standard input to the `add_item` file in the `data` subdirectory and will execute the program `add_item` on the data in that file.

Some of the programs take argument(s) that must be supplied when the program is executed. These programs will fail if they are invoked with no argument. For example, the `word_transform` program in the Chapter `11` directory requires two input files. You might invoke this program as follows:

```
# word_transform takes two files, samples are in the data directory
# this execution uses the file data/rules to transform the text in data/text
word_transform data/rules data/text  # UNIX
```

See Section 1.1.1 for a description of how C++ programs are run, page 22 for how files are bound to the standard input and standard output, and Section 6.2.5 for how arguments are passed to a C++ program.

Running the Programs

Each directory contains a simple shell script named `runpgms` that runs the executables made in that directory.

Note that on some systems, the default setting for the `PATH` variable does not include the current directory. In this case, if you try to execute

```
add_item < data/add_item  # UNIX
```

the `add_item` program will not be found. Instead, you must write

```
./add_item < data/add_item   # UNIX
```

where `./`tells the system to run the `add_item` program in the current directory.