

5. Select the *main.cpp* file that you created earlier in Listing 1-1. (Alternatively, if you haven't yet created this file, select **New Item** instead of **Existing Item**. Name the file *main.cpp* and type the contents of Listing 1-1 into the resulting editor window.)
6. Select **Build ▸ Build Solution**. If any error messages appear in the output box, make sure you've typed Listing 1-1 correctly. If you still get error messages, read them carefully for hints.
7. Select **Debug ▸ Start Without Debugging** or press CTRL-F5 to run your program. The letters Hello, world! should print to the console (followed by Press Any Key to Continue).

macOS: Xcode

If you're running macOS, you should install the Xcode development environment.

1. Open the **App Store**.
2. Search for and install the **Xcode** IDE. Installation might take more than an hour depending on the speed of your machine and internet connection. When installation is complete, open **Terminal** and navigate to the directory where you've saved *main.cpp*.
3. Enter `clang++ main.cpp -o hello` in the Terminal to compile your program. The `-o` option tells the tool chain where to write the output. (If any compiler errors appear, check that you've entered the program correctly.)
4. Enter `./hello` in the Terminal to run your program. The text Hello, world! should appear onscreen.

To compile and run your program, open the Xcode IDE and follow these steps:

1. Select **File ▸ New ▸ Project**.
2. Select **macOS ▸ Command Line Tool** and click **Next**. In the next dialog, you can modify where to create the project's file directory. For now, accept the defaults and click **Create**.
3. Name your project **hello** and set its **Type** to **C++**. See Figure 1-3.
4. You now need to import your code from Listing 1-1 into your project. An easy way to do this is to copy and paste the contents of *main.cpp* into your project's *main.cpp*. Another way is to use Finder to replace your *main.cpp* into your project's *main.cpp*. (Normally you won't have to handle this when creating new projects. It's just an artifact of this tutorial having to handle multiple operating environments.)
5. Click **Run**.

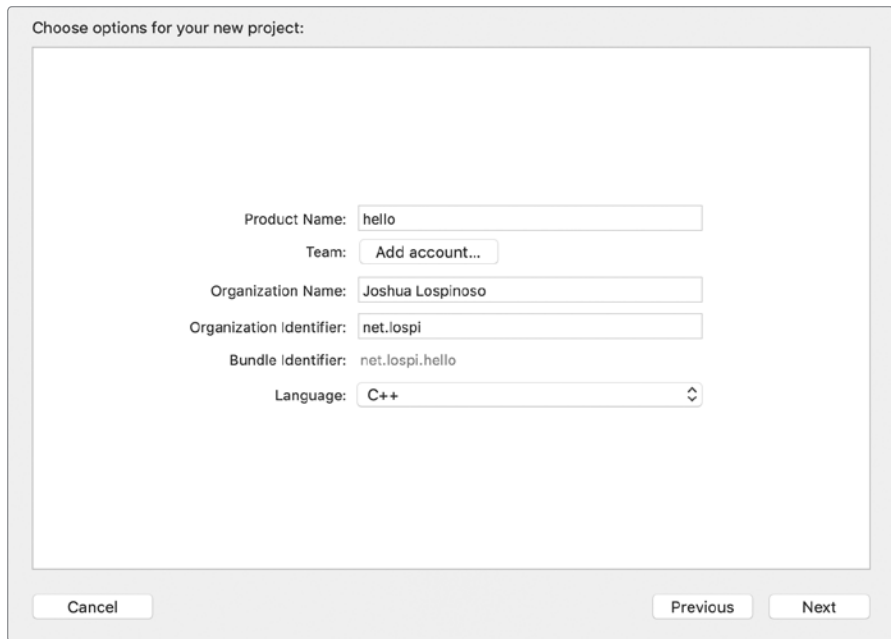


Figure 1-3: The New Project dialog in Xcode

Linux and GCC

On Linux, you can choose between two main C++ compilers: GCC and Clang. At press time, the latest stable release is 9.1 and the latest major Clang release is 8.0.0. In this section, you’ll install both. Some users find the error messages from one to be more helpful than the other.

NOTE

GCC is an initialism for GNU Compiler Collection. GNU, pronounced “guh-NEW,” is a recursive acronym for “GNU’s Not Unix!” GNU is a Unix-like operating system and a collection of computer software.

Try to install GCC and Clang from your operating system’s package manager, but beware. Your default repositories might have old versions that may or may not have C++ 17 support. If your version doesn’t have C++ 17 support, you won’t be able to compile some examples in the book, so you’ll need to install updated versions of GCC or Clang. For brevity, this chapter covers how to do this on Debian and from source. You can either investigate how to perform corollary actions on your chosen Linux flavor or set up a development environment with one of the operating systems listed in this chapter.

Installing GCC and Clang on Debian

Depending on what software the Personal Package Archives contain when you’re reading this chapter, you might be able to install GCC 8.1

and Clang 6.0.0 directly using Advanced Package Tool (APT), which is Debian's package manager. This section shows how to install GCC and Clang on Ubuntu 18.04, the latest LTS Ubuntu version at press time.

1. Open a terminal.
2. Update and upgrade your currently installed packages:

```
$ sudo apt update && sudo apt upgrade
```

3. Install GCC 8 and Clang 6.0:

```
$ sudo apt install g++-8 clang-6.0
```

4. Test GCC and Clang:

```
$ g++-8 -version
g++-8 (Ubuntu 8-20180414-1ubuntu2) 8.0.1 20180414 (experimental) [trunk
revision 259383]
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
$ clang++-6.0 --version
clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

If either command returns an error stating that the command wasn't found, the corresponding compiler did not install correctly. Try searching for information on the error you receive, especially in the documentation and forums for your respective package manager.

Installing GCC from Source

If you can't find the latest GCC or Clang versions with your package manager (or your Unix variant doesn't have one), you can always install GCC from source. Note that this takes a lot of time (as much as several hours), and you might need to get your hands dirty: installation often runs into errors that you'll need to research to resolve. To install GCC, follow the instructions available at <https://gcc.gnu.org/>. This section summarizes the far more extensive documentation available on that site.

NOTE

For brevity, this tutorial doesn't detail Clang installation. Refer to <https://clang.llvm.org/> for more information.

To install GCC 8.1 from source, do the following:

1. Open a terminal.

2. Update and upgrade your currently installed packages. For example, with APT you would issue the following command:

```
$ sudo apt update && sudo apt upgrade
```

3. From one of the available mirrors at <https://gcc.gnu.org/mirrors.html>, download the files *gcc-8.1.0.tar.gz* and *gcc-8.1.0.tar.gz.sig*. These files can be found in *releases/gcc-8.1.0*.
4. (Optional) Verify the integrity of the package. First, import the relevant GnuPG keys. You can find these listed on the mirrors site. For example:

```
$ gpg --keyserver keyserver.ubuntu.com --recv C3C45C06
gpg: requesting key C3C45C06 from hkp server keyserver.ubuntu.com
gpg: key C3C45C06: public key "Jakub Jelinek <jakub@redhat.com>" imported
gpg: key C3C45C06: public key "Jakub Jelinek <jakub@redhat.com>" imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 2
gpg:             imported: 2 (RSA: 1)
```

Verify what you downloaded:

```
$ gpg --verify gcc-8.1.0.tar.gz.sig gcc-8.1.0.tar.gz
gpg: Signature made Wed 02 May 2018 06:41:51 AM DST using DSA key ID
C3C45C06
gpg: Good signature from "Jakub Jelinek <jakub@redhat.com>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:             There is no indication that the signature belongs to the
owner.
Primary key fingerprint: 33C2 35A3 4C46 AA3F FB29 3709 A328 C3A2 C3C4
5C06
```

The warnings you see mean that I haven't marked the signer's certificate as trusted on my machine. To verify that the signature belongs to the owner, you'll need to verify the signing key using some other means (for example, by meeting the owner in person or by verifying the primary key fingerprint out of band). For more information about GNU Privacy Guard (GPG), refer to *PGP & GPG: Email for the Practical Paranoid* by Michael W. Lucas or browse to https://gnupg.org/download/integrity_check.html for specific information about GPG's integrity-checking facilities.

5. Decompress the package (this command might take a few minutes):

```
$ tar xzf gcc-8.1.0.tar.gz
```

6. Navigate to the newly created *gcc-8.1.0* directory:

```
$ cd gcc-8.1.0
```

7. Download GCC's prerequisites:

```
$ ./contrib/download_prerequisites
--snip--
gmp-6.1.0.tar.bz2: OK
mpfr-3.1.4.tar.bz2: OK
mpc-1.0.3.tar.gz: OK
isl-0.18.tar.bz2: OK
All prerequisites downloaded successfully.
```

8. Configure GCC using the following commands:

```
$ mkdir objdir
$ cd objdir
$ ../configure --disable-multilib
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
--snip--
configure: creating ./config.status
config.status: creating Makefile
```

Instructions are available at <https://gcc.gnu.org/install/configure.html>.

9. Build the GCC binaries (perhaps do this overnight, because it can take hours):

```
$ make
```

Full instructions are available at <https://gcc.gnu.org/install/build.html>.

10. Test whether your GCC binaries built correctly:

```
$ make -k check
```

Full instructions are available at <https://gcc.gnu.org/install/test.html>.

11. Install GCC:

```
$ make install
```

This command places a handful of binaries into your operating system's default executable directory, which is usually `/usr/local/bin`. Full instructions are available at <https://gcc.gnu.org/install/>.

12. Verify that GCC installed correctly by issuing the following command:

```
$ x86_64-pc-linux-gnu-gcc-8.1.0 --version
```

If you get an error indicating that the command was not found, your installation did not succeed. Refer to the gcc-help mailing list at <https://gcc.gnu.org/ml/gcc-help/>.

NOTE

You might want to alias the cumbersome `x86_64-pc-linux-gnu-gcc-8.1.0` to something like `g++8`, for example, using a command like this:

```
$ sudo ln -s /usr/local/bin/x86_64-pc-linux-gnu-gcc-8.1.0 /usr/local/bin/g++8
```

13. Navigate to the directory where you've saved `main.cpp` and compile your program with GCC:

```
$ x86_64-pc-linux-gnu-gcc-8.1.0 main.cpp -o hello
```

14. The `-o` flag is optional; it tells the compiler what to name the resulting program. Because you specified the program name as `hello`, you should be able to run your program by entering `./hello`. If any compiler errors appear, ensure that you input the program's text correctly. (The compiler errors should help you determine what went wrong.)

Text Editors

If you'd rather not work with one of the aforementioned IDEs, you can write C++ code using a simple text editor like Notepad (Windows), TextEdit (Mac), or Vim (Linux); however, a number of excellent editors are designed specifically for C++ development. Choose the environment that makes you most productive.

If you're running Windows or macOS, you already have a high-quality, fully featured IDE at your disposal, namely Visual Studio or Xcode. Linux options include Qt Creator (<https://www.qt.io/ide/>), Eclipse CDT (<https://eclipse.org/cdt/>), and JetBrains's CLion (<https://www.jetbrains.com/clion/>). If you're a Vim or Emacs user, you'll find plenty of C++ plug-ins.

NOTE

If cross-platform C++ is important to you, I highly recommend taking a look at JetBrains's CLion. Although CLion is a paid product, unlike many of its competitors, at press time JetBrains does offer reduced-price and free licenses for students and open source project maintainers.

Bootstrapping C++

This section gives you just enough context to support the example code in the chapters to come. You'll have questions about the details, and the coming chapters will answer them. Until then, don't panic!

The C++ Type System

C++ is an object-oriented language. Objects are abstractions with state and behavior. Think of a real-world object, such as a light switch. You can describe its *state* as the condition that the switch is in. Is it on or off? What is the maximum voltage it can handle? What room in the house is it in? You could also

describe the switch's *behavior*. Does it toggle from one state (on) to another state (off)? Or is it a dimmer switch, which can be set to many different states between on and off?

The collection of behaviors and states describing an object is called its *type*. C++ is a *strongly typed language*, meaning each object has a predefined data type.

C++ has a built-in integer type called `int`. An `int` object can store whole numbers (its state), and it supports many math operations (its behavior).

To perform any meaningful tasks with `int` types, you'll create some `int` objects and name them. Named objects are called *variables*.

Declaring Variables

You declare variables by providing their type, followed by their name, followed by a semicolon. The following example declares a variable called `the_answer` with type `int`:

```
int❶ the_answer❷;
```

The type, `int` ^❶, is followed by the variable name, `the_answer` ^❷.

Initializing a Variable's State

When you declare variables, you initialize them. *Object initialization* establishes an object's initial state, such as setting its value. We'll delve into the details of initialization in Chapter 2. For now, you can use the equal sign (=) following a variable declaration to set the variable's initial value. For example, you could declare and assign the `the_answer` in one line:

```
int the_answer = 42;
```

After running this line of code, you have a variable called `the_answer` with type `int` and value 42. You can assign variables equal to the result of math expressions, such as:

```
int lucky_number = the_answer / 6;
```

This line evaluates the expression `the_answer / 6` and assigns the result to `lucky_number`. The `int` type supports many other operations, such as addition +, subtraction -, multiplication *, and modulo division %.

NOTE

If you aren't familiar with modulo division or are wondering what happens when you divide two integers and there's a remainder, you're asking great questions. And those great questions will be answered in detail in Chapter 7.

Conditional Statements

Conditional statements allow you to make decisions in your programs. These decisions rest on Boolean expressions, which evaluate to true or false. For example, you can use *comparison operators*, such as “greater than” or “not equal to,” to build Boolean expressions.

Some basic comparison operators that work with `int` types appear in the program in Listing 1-2.

```
int main() {
    int x = 0;
    42 == x; // Equality
    42 != x; // Inequality
    100 > x; // Greater than
    123 >= x; // Greater than or equal to
    -10 < x; // Less than
    -99 <= x; // Less than or equal to
}
```

Listing 1-2: A program using comparison operators

This program produces no output (compile and run Listing 1-2 to verify this). While the program doesn’t produce any output, compiling it helps to verify that you’ve written valid C++. To generate more interesting programs, you’d use a conditional statement like `if`.

An `if` statement contains a Boolean expression and one or more nested statements. Depending on whether the Boolean evaluates to true or false, the program decides which nested statement to execute. There are several forms of `if` statements, but the basic usage follows:

if (❶`boolean-expression`) ❷`statement`

If the Boolean expression ❶ is true, the nested statement ❷ executes; otherwise, it doesn’t.

Sometimes, you’ll want a group of statements to run rather than a single statement. Such a group is called a *compound statement*. To declare a compound statement, simply wrap the group of statements in braces `{ }`. You can use compound statements within `if` statements as follows:

```
if (❶boolean-expression) { ❷
    statement1;
    statement2;
    --snip--
}
```

If the Boolean expression ❶ is true, all the statements in the compound statement ❷ execute; otherwise, none of them do.

You can elaborate the if statement using `else if` and `else` statements. These optional additions allow you to describe more complicated branching behavior, as shown in Listing 1-3.

```
❶ if (boolean-expression-1) statement-1
❷ else if (boolean-expression-2) statement-2
❸ else statement-3
```

Listing 1-3: An if statement with else if and else branches

First, *boolean-expression-1* ❶ is evaluated. If *boolean-expression-1* is true, *statement-1* is evaluated, and the if statement stops executing. If *boolean-expression-1* is false, *boolean-expression-2* ❷ is evaluated. If true, *statement-2* is evaluated. Otherwise, *statement-3* ❸ is evaluated. Note that *statement-1*, *statement-2*, and *statement-3* are mutually exclusive and together they cover all possible outcomes of the if statement. Only one of the three will be evaluated.

You can include any number of `else if` clauses or omit them entirely. As with the initial if statement, the Boolean expression for each `else if` clause is evaluated in order. When one of these Boolean expressions evaluates to true, evaluation stops and the corresponding statement executes. If no `else if` evaluates to true, the `else` clause's *statement-3* *always* executes. (As with the `else if` clauses, the `else` is optional.)

Consider Listing 1-4, which uses an if statement to determine which statement to print.

```
#include <stdio>

int main() {
    int x = 0; ❶
    if (x > 0) printf("Positive.");
    else if (x < 0) printf("Negative.");
    else printf("Zero.");
}
-----
Zero.
```

Listing 1-4: A program with conditional behavior

Compile the program and run it. Your result should also be Zero. Now change the x value ❶. What does the program print now?

NOTE Notice that *main* in Listing 1-4 omits a return statement. Because *main* is a special function, return statements are optional.

Functions

Functions are blocks of code that accept any number of input objects called *parameters* or *arguments* and can return output objects to their callers.

You declare functions according to the general syntax shown in Listing 1-5.

```
return-type❶ function_name❷(par-type1 par_name1❸, par-type2 par_name2❹) {  
    --snip--  
    return❺ return-value;  
}
```

Listing 1-5: The general syntax for a C++ function

The first part of this function declaration is the type of the return variable ❶, such as `int`. When the function returns a value ❺, the type of return-value must match return-type.

Then you declare the function's name ❷ after declaring the return type. A set of parentheses following the function name contains any number of comma-separated input parameters that the function requires. Each parameter also has a type and a name.

Listing 1-5 has two parameters. The first parameter ❸ has type `par-type1` and is named `par_name1`, and the second parameter ❹ has type `par-type2` and is named `par_name2`. Parameters represent the objects passed into a function.

A set of braces following that list contains the function's body. This is a compound statement that contains the function's logic. Within this logic, the function might decide to return a value to the function's caller. Functions that return values will have one or more `return` statements. Once a function returns, it stops executing, and the flow of the program returns to whatever called the function. Let's look at an example.

Example: A Step Function

For demonstration purposes, this section shows how to build a mathematical function called `step_function` that returns -1 for all negative arguments, 0 for a zero-valued argument, and 1 for all positive arguments. Listing 1-6 shows how you might write the `step_function`.

```
int step_function(int ❶x) {  
    int result = 0; ❷  
    if (x < 0) {  
        result = -1; ❸  
    } else if (x > 0) {  
        result = 1; ❹  
    }  
    return result; ❺  
}
```

Listing 1-6: A step function that returns -1 for negative values, 0 for zero, and 1 for positive values

The `step_function` takes a single argument `x` ❶. The result variable is declared and initialized to 0 ❷. Next, the `if` statement sets `result` to -1 ❸ if `x` is less than 0. If `x` is greater than 0, the `if` statement sets `result` to 1 ❹. Finally, `result` is returned to the caller ❺.

Calling Functions

To call (or *invoke*) a function, you use the name of the desired function, parentheses, and a comma-separated list of the required parameters. The compiler reads files from top to bottom, so the function's declaration must appear before its point of first use.

Consider the program in Listing 1-7, which uses the `step_function`.

```
int step_function(int x) {  
    --snip--  
}  
  
int main() {  
    int value1 = step_function(100); // value1 is 1  
    int value2 = step_function(0);   // value2 is 0  
    int value3 = step_function(-10); // value3 is -1  
}
```

Listing 1-7: A program using the `step_function`. (This program produces no output.)

Listing 1-7 calls `step_function` three times with different arguments and assigns the results to the variables `value1`, `value2`, and `value3`.

Wouldn't it be nice if you could print these values? Fortunately, you can use the `printf` function to build output from different variables. The trick is to use `printf` format specifiers.

***printf* Format Specifiers**

In addition to printing constant strings (like `Hello, world!` in Listing 1-1), `printf` can combine multiple values into a nicely formatted string; it is a special kind of function that can take one or more arguments.

The first argument to `printf` is always a *format string*. The format string provides a template for the string to be printed, and it contains any number of special *format specifiers*. Format specifiers tell `printf` how to interpret and format the arguments following the format string. All format specifiers begin with `%`.

For example, the format specifier for an `int` is `%d`. Whenever `printf` sees a `%d` in the format string, it knows to expect an `int` argument following the format specifier. Then `printf` replaces the format specifier with the argument's actual value.

NOTE

The `printf` function is a derivative of the `wprintf` function offered in BCPL, a defunct programming language designed by Martin Richards in 1967. Providing the specifiers `%H`, `%I`, and `%O` to `wprintf` resulted in hexadecimal and octal output via the functions `WRITEHEX`, `WRITED`, and `WRITEOCT`. It's unclear where the `%d` specifier comes from (perhaps the `D` in `WRITED`?), but we're stuck with it.

Consider the following `printf` call, which prints the string Ten 10, Twenty 20, Thirty 30:

```
printf("Ten %d❶, Twenty %d❷, Thirty %d❸", 10❹, 20❺, 30❻);
```

The first argument, "Ten %d, Twenty %d, Thirty %d", is the format string. Notice that there are three format specifiers %d ❶ ❷ ❸. There are also three arguments after the format string ❹ ❺ ❻. When `printf` builds the output, it replaces the argument at ❶ with the one at ❹, the argument at ❷ with the one at ❺, and the argument at ❸ with the one at ❻.

IOSTREAMS, PRINTF, AND INPUT OUTPUT PEDAGOGY

People have really strong opinions about which standard output method to teach C++ newcomers. One option is `printf`, which has a lineage that traces back to C. Another option is `cout`, which is part of the C++ standard library's `iostream` library. This book teaches both: `printf` in Part I and `cout` in Part II. Here's why.

This book builds your C++ knowledge brick by brick. Each chapter is designed sequentially so you don't need a leap of faith to understand code examples. More or less, you'll know exactly what every line does. Because `printf` is fairly primitive, you'll have enough knowledge by Chapter 3 to know exactly how it works.

In contrast, `cout` involves a whole lot of C++ concepts, and you won't have sufficient background to understand how it works until the end of Part I. (What's a stream buffer? What's `operator<<`? What's a method? How does `flush()` work? Wait, `cout` flushes automatically in the destructor? What's a destructor? What's `setf`? Actually, what's a format flag? A `BitmaskType`? Oh my, what's a manipulator? And so on.)

Of course, `printf` has issues, and once you've learned `cout`, you should prefer it. With `printf` you can easily introduce mismatches between format specifiers and arguments, and this can cause strange behavior, program crashes, and even security vulnerabilities. Using `cout` means you don't need format strings, so you don't need to remember format specifiers. You'll never get mismatches between format strings and arguments. `Iostreams` are also *extensible*, meaning you can integrate input and output functionality into your own types.

This book teaches modern C++ directly, but on this particular topic it compromises a bit of modernist dogma in exchange for a deliberate, linear approach. As an ancillary benefit, you'll be prepared to encounter `printf` specifiers, which is likely to happen at some point in your programming career. Most languages, such as C, Python, Java, and Ruby, have facilities for `printf` specifiers, and there are analogs in C#, JavaScript, and other languages.

Revisiting step_function

Let's look at another example that uses `step_function`. Listing 1-8 incorporates variable declarations, function calls, and `printf` format specifiers.

```
#include <stdio> ❶

int step_function(int x) { ❷
    --snip--
}

int main() { ❸
    int num1 = 42; ❹
    int result1 = step_function(num1); ❺

    int num2 = 0;
    int result2 = step_function(num2);

    int num3 = -32767;
    int result3 = step_function(num3);

    printf("Num1: %d, Step: %d\n", num1, result1); ❻
    printf("Num2: %d, Step: %d\n", num2, result2);
    printf("Num3: %d, Step: %d\n", num3, result3);

    return 0;
}

-----
Num1: 42, Step: 1 ❻
Num2: 0, Step: 0
Num3: -32767, Step: -1
```

Listing 1-8: A program that prints the results of applying `step_function` to several integers

Because the program uses `printf`, `stdio` ❶ is included. The `step_function` ❷ is defined so you can use it later in the program, and `main` ❸ establishes the defined entry point.

NOTE

Some listings in this book will build on one another. To save trees, you'll see the use of the `--snip--` notation to denote no changes to the reused portion.

Inside `main`, you initialize a few `int` types, like `num1` ❹. Next, you pass these variables to `step_function` and initialize result variables to store the returned values, like `result1` ❺.

Finally, you print the returned values by invoking `printf`. Each invocation starts with a format string, like `"Num1: %d, Step: %d\n"` ❻. There are two `%d` format specifiers embedded in each format string. Per the requirements of `printf`, there are two parameters following the format string, `num1` and `result1`, that correspond to these two format specifiers.

Comments

Comments are human-readable annotations that you can place into your source code. You can add comments to your code using the notation `//` or `/**/`. These symbols, `//` or `/**/`, tell the compiler to ignore everything from the first forward slash to the next newline, which means you can put comments in-line with your code as well as on their own lines:

```
// This comment is on its own line
int the_answer = 42; // This is an in-line comment
```

You can use the `/**/` notation to include multiline comments in your code:

```
/*
 * This is a comment
 * That lives on multiple lines
 * Don't forget to close
 */
```

The comment starts with `/*` and ends with `*/`. (The asterisks on the lines between the starting and ending forward slash are optional but are commonly used.)

When to use comments is a matter of eternal debate. Some programming luminaries suggest that code should be so expressive and self-explanatory as to render comments largely unnecessary. They might say that descriptive variable names, short functions, and good tests are usually all the documentation you need. Other programmers like to place comments all over the place.

You can cultivate your own philosophy. The compiler will totally ignore whatever you do because it never interprets comments.

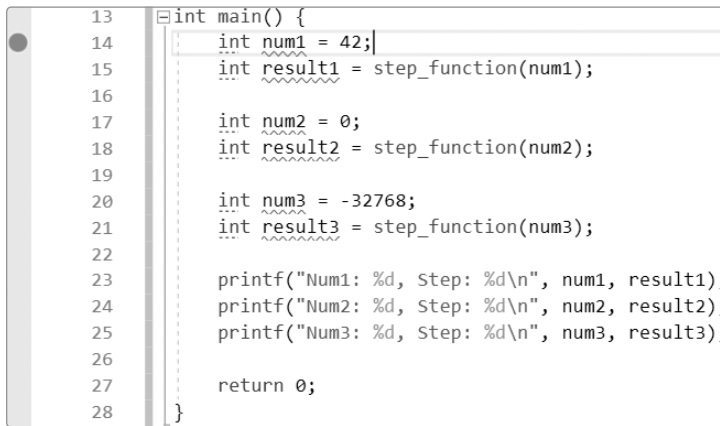
Debugging

One of the most important skills for a software engineer is efficient, effective debugging. Most development environments have debugging tools. On Windows, macOS, and Linux, the debugging tools are excellent. Learning to use them well is an investment that pays off very quickly. This section provides a quick tour of how to use a debugger to step through the program in Listing 1-8. You can skip to whichever environment is most relevant to you.

Visual Studio

Visual Studio has an excellent, built-in debugger. I suggest that you debug programs in its *Debug* configuration. This causes the tool chain to build a target that enhances the debugging experience. The only reason to debug in *Release* mode is to diagnose some rare conditions that occur in Release mode but not in Debug mode.

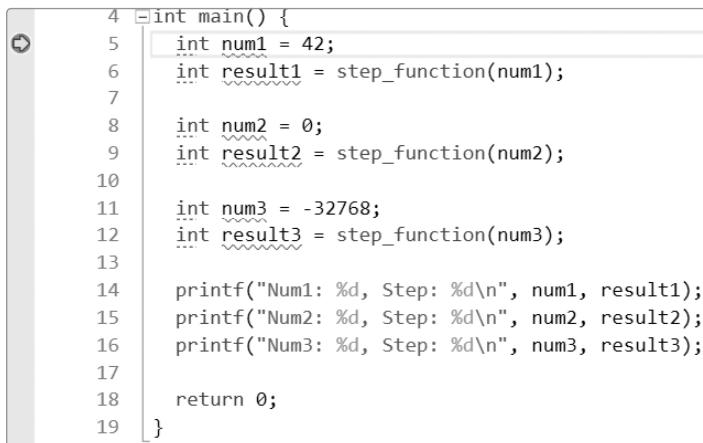
1. Open *main.cpp* and locate the first line of *main*.
2. Click the margin just to the left of the line number corresponding to the first line of *main* to insert a breakpoint. A red circle appears where you clicked, as shown in Figure 1-4.

A screenshot of a code editor window. On the left, a vertical margin shows line numbers 13 through 28. A red circle, representing a breakpoint, is placed in the margin next to line 14. The code in the editor is as follows:

```
13 int main() {  
14     int num1 = 42;  
15     int result1 = step_function(num1);  
16  
17     int num2 = 0;  
18     int result2 = step_function(num2);  
19  
20     int num3 = -32768;  
21     int result3 = step_function(num3);  
22  
23     printf("Num1: %d, Step: %d\n", num1, result1);  
24     printf("Num2: %d, Step: %d\n", num2, result2);  
25     printf("Num3: %d, Step: %d\n", num3, result3);  
26  
27     return 0;  
28 }
```

Figure 1-4: Inserting a breakpoint

3. Select **Debug ▶ Start Debugging**. The program will run up to the line where you’ve inserted a breakpoint. The debugger will halt program execution, and a yellow arrow will appear to indicate the next instruction to be run, as shown in Figure 1-5.

A screenshot of the same code editor window as in Figure 1-4. The program has been executed up to the breakpoint on line 14. A yellow arrow, indicating the next instruction to be executed, is now positioned to the left of line 15. The code is the same as in the previous figure:

```
4 int main() {  
5     int num1 = 42;  
6     int result1 = step_function(num1);  
7  
8     int num2 = 0;  
9     int result2 = step_function(num2);  
10  
11     int num3 = -32768;  
12     int result3 = step_function(num3);  
13  
14     printf("Num1: %d, Step: %d\n", num1, result1);  
15     printf("Num2: %d, Step: %d\n", num2, result2);  
16     printf("Num3: %d, Step: %d\n", num3, result3);  
17  
18     return 0;  
19 }
```

Figure 1-5: The debugger halts execution at the breakpoint.

4. Select **Debug ▶ Step Over**. The step over operation executes the instruction without “stepping into” any function calls. By default, the keyboard shortcut for step over is F10.

5. Because the next line calls `step_function`, select **Debug ▸ Step Into** to call `step_function` and break on the first line. You can continue debugging this function by stepping into/over its instructions. By default, the keyboard shortcut for step into is F11.
6. To allow execution to return to `main`, select **Debug ▸ Step Out**. By default, the keyboard shortcut for this operation is SHIFT-F11.
7. Inspect the Autos window by selecting **Debug ▸ Windows ▸ Auto**. You can see the current value of some of the important variables, as shown in Figure 1-6.

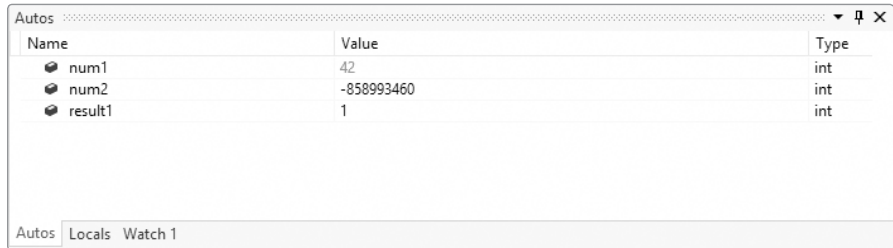


Figure 1-6: The Autos window shows the values of variables at the current breakpoint.

You can see `num1` is set to 42 and `result1` is set to 1. Why does `num2` have a gibberish value? Because the initialization to 0 hasn't happened yet: it's the next instruction to execute.

NOTE

The debugger has just emphasized a very important low-level detail: allocating an object's storage and initializing an object's value are two distinct steps. You'll learn more about storage allocation and object initialization in Chapter 4.

The Visual Studio debugger supports many more features. For more information, check out the Visual Studio documentation link available at <https://ccc.codes/>.

Xcode

Xcode also has an excellent, built-in debugger that's completely integrated into the IDE.

1. Open `main.cpp` and locate the first line of `main`.
2. Click the first line and then select **Debug ▸ Breakpoints ▸ Add Breakpoint at Current Line**. A breakpoint appears, as shown in Figure 1-7.


```

#include "step_function.h"
#include <stdio>

int main() {
    int num1 = 42;
    int result1 = step_function(num1);

    int num2 = 0;
    int result2 = step_function(num2);

    int num3 = -32768;
    int result3 = step_function(num3);

    printf("Num1: %d, Step: %d\n", num1, result1);
    printf("Num2: %d, Step: %d\n", num2, result2);
    printf("Num3: %d, Step: %d\n", num3, result3);

    return 0;
}

```

Figure 1-7: Inserting a breakpoint

3. Select **Run**. The program will run up to the line with the inserted breakpoint. The debugger will halt program execution, and a green arrow will appear to indicate the next instruction to be run, as shown in Figure 1-8.

```

#include "step_function.h"
#include <stdio>

int main() {
    int num1 = 42;
    int result1 = step_function(num1);

    int num2 = 0;
    int result2 = step_function(num2);

    int num3 = -32768;
    int result3 = step_function(num3);

    printf("Num1: %d, Step: %d\n", num1, result1);
    printf("Num2: %d, Step: %d\n", num2, result2);
    printf("Num3: %d, Step: %d\n", num3, result3);

    return 0;
}

```

Thread 1: breakpoint 1.1

Figure 1-8: The debugger halts execution at the breakpoint.

4. Select **Debug ▶ Step Over** to execute the instruction without “stepping into” any function calls. By default, the keyboard shortcut for step over is F6.
5. Because the next line calls `step_function`, select **Debug ▶ Step Into** to call `step_function` and break on the first line. You can continue debugging this function by stepping into/over its instructions. By default, the keyboard shortcut for step into is F7.

6. To allow execution to return to main, select **Debug ▶ Step Out**. By default, the keyboard shortcut for step out is F8.
7. Inspect the Autos window at the bottom of the *main.cpp* screen. You can see the current value of some of the important variables, as shown in Figure 1-9.

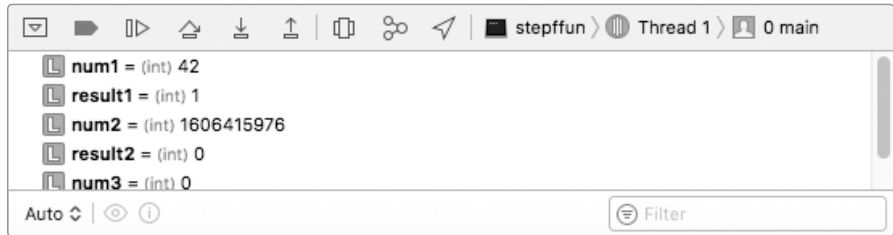


Figure 1-9: The Autos window shows the values of variables at the current breakpoint.

You can see `num1` is set to 42 and `result1` is set to 1. Why does `num2` have a gibberish value? Because the initialization to 0 hasn't happened yet: it's the next instruction to execute.

The Xcode debugger supports many more features. For more information, check out the Xcode documentation link at <https://ccc.codes/>.

GCC and Clang Debugging with GDB and LLDB

The GNU Project Debugger (GDB) is a powerful debugger (<https://www.gnu.org/software/gdb/>). You can interact with GDB using the command line. To enable debugging support during compilation with `g++` or `clang++`, you must add the `-g` flag.

Your package manager will most likely have GDB. For example, to install GDB with Advanced Package Tool (APT), enter the following command:

```
$ sudo apt install gdb
```

Clang also has an excellent debugger called the Low Level Debugger (LLDB), which you can download at <https://lldb.lldm.org/>. It was designed to work with the GDB commands in this section, so for brevity I won't cover LLDB explicitly. You can debug programs compiled with GCC debug support using LLDB, and you can debug programs compiled with Clang debug support using GDB.

NOTE Xcode uses LLDB in the background.

To debug the program in Listing 1-8 (on page 20) using GDB, follow these steps:

1. In a command line, navigate to the folder where you've stored your header and source files.

2. Compile your program with debug support:

```
$ g++-8 main.cpp -o stepfun -g
```

3. Debug your program using gdb; you should see the following interactive console session:

```
$ gdb stepfun
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stepfun...done.
(gdb)
```

4. To insert a breakpoint, use the command `break`, which takes a single argument corresponding to the name of the source file and the line where you want to break, separated by a colon (:). For example, suppose you want to break on the first line of *main.cpp*. In Listing 1-8, that is on line 5 (although you might need to adjust placement depending on how you've written the source). You can create a breakpoint using the following command at the (gdb) prompt:

```
(gdb) break main.cpp:5
```

5. You can also tell gdb to break at a particular function by name:

```
(gdb) break main
```

6. Either way, you can now execute your program:

```
(gdb) run
Starting program: /home/josh/stepfun
Breakpoint 1, main () at main.cpp:5
5      int num1 = 42;
(gdb)
```

7. To single step into an instruction, you use the `step` command to follow each line of the program, including steps into functions:

```
(gdb) step
6      int result1 = step_function(num1);
```

8. To continue stepping, press `ENTER` to repeat the last command:

```
(gdb)
step_function (x=42) at step_function.cpp:4
```

9. To step back out of a function invocation, you use the `finish` command:

```
(gdb) finish
Run till exit from #0  step_function (x=42) at step_function.cpp:7
0x000000000400546 in main () at main.cpp:6
6      int result1 = step_function(num1);
Value returned is $1 = 1
```

10. To execute an instruction without stepping into a function call, you use the `next` command:

```
(gdb) next
8      int num2 = 0;
```

11. To inspect the current value of variables, you use the `info locals` command:

```
(gdb) info locals
num2 = -648029488
result2 = 32767
num1 = 42
result1 = 1
num3 = 0
result3 = 0
```

Notice that any variables that have not yet been initialized will not have sensible values.

12. To continue execution until the next breakpoint (or until the program completes), use the `continue` command:

```
(gdb) continue
Continuing.
Num1: 42, Step: 1
Num2: 0, Step: 0
Num3: -32768, Step: -1
[Inferior 1 (process 1322) exited normally]
```

13. Use the `quit` command to exit `gdb` at any time.

GDB supports many more features. For more information, check out the documentation at <https://sourceware.org/gdb/current/onlinedocs/gdb/>.

Summary

This chapter got you up and running with a working C++ development environment, and you compiled your first C++ program. You learned about the components of a build tool chain and the roles they play in the compilation process. Then you explored a few essential C++ topics, such as types, declaring variables, statements, conditionals, functions, and `printf`. The chapter wrapped up with a tutorial on setting up a debugger and stepping through your project.

NOTE

If you have problems setting up your environment, search on your error messages online. If that fails, post your question to Stack Overflow at <https://stackoverflow.com/>, the C++ subreddit at https://www.reddit.com/r/cpp_questions/, or the C++ Slack channel at <https://cpplang.now.sh/>.

EXERCISES

Try these exercises to practice what you've learned in this chapter. (The book's companion code is available at <https://ccc.codes>.)

1-1. Create a function called `absolute_value` that returns the absolute value of its single argument. The absolute value of an integer `x` is the following: `x` (itself) if `x` is greater than or equal to 0; otherwise, it is `x` times `-1`. You can use the program in Listing 1-9 as a template:

```
#include <stdio>

int absolute_value(int x) {
    // Your code here
}

int main() {
    int my_num = -10;
    printf("The absolute value of %d is %d.\n", my_num,
        absolute_value(my_num));
}
```

Listing 1-9: A template for a program that uses an `absolute_value` function

1-2. Try running your program with different values. Did you see the values you expect?

1-3. Run your program with a debugger, stepping through each instruction.

1-4. Write another function called `sum` that takes two `int` arguments and returns their sum. How can you modify the template in Listing 1-9 to test your new function?

1-5. C++ has a vibrant online community, and the internet is awash with excellent C++ related material. Investigate the CppCast podcast at <http://cppcast.com/>. Search for CppCon and C++Now videos available on YouTube. Add <https://cppreference.com/> and <http://www.cplusplus.com/> to your browser's bookmarks.

1-6. Finally, download a copy of the International Organization for Standardization (ISO) C++ 17 Standard from <https://isocpp.org/std/the-standard/>. Unfortunately, the official ISO standard is copyrighted and must be purchased. Fortunately, you can download a “draft,” free of charge, that differs only cosmetically from the official version.

Note *Because the ISO standard's page numbers differ from version to version, this book will refer to specific sections using the same naming schema as the standard itself. This schema cites sections by enclosing the section name with square brackets. Subsections are appended with period separation. For example, to cite the section on the C++ Object Model, which is contained in the Introduction section, you would write [intro.object].*

FURTHER READING

- *The Pragmatic Programmer: From Journeyman to Master* by Andrew Hunt and David Thomas (Addison-Wesley Professional, 2000)
- *The Art of Debugging with GDB, DDD, and Eclipse* by Norman Matloff and Peter Jay Salzman (No Starch Press, 2008)
- *PGP & GPG: Email for the Practical Paranoid* by Michael W. Lucas (No Starch Press, 2006)
- *The GNU Make Book* by John Graham-Cumming (No Starch Press, 2015)