THE EXPERT'S VOICE® IN C++

C++ Recipes

A Problem-Solution Approach

Bruce Sutherland

Apress®

Contents at a Glance

Introduction

The C++ programming language is undergoing continuous development and improvement. This effort to keep C++ on the cutting edge of language features is driven by the fact that C++ still finds an important role to play in high-performance, portable applications. Few other languages can be used on as many platforms as C++ and without having a runtime environment dependency. This is partly thanks to the nature of C++ as a compiled programming language. C++ programs are built into application binaries through a combination of processes that include compiling and linking.

Compiler choice is particularly important in today's C++ landscape, thanks to the rate at which the language is changing. Development of the C++ programming language was started by Bjarne Stoustrup in 1979, when it was called C with Classes. The language didn't see formal standardization until 1998; an updated standard was published in 2003. There was another gap of eight years until the standard was updated again with the introduction of C++11 in 2011. This version brought a considerable number of updates to the C++ programming language and is distinguished from "older" C++ with the Modern C++ moniker. A further, less significant, update to the C++ standard was introduced in late 2014, but we haven't yet begun to see compilers that support many of the features added to Modern C++.

This book introduces you to code written specifically for the C++14 standard using the Clang compiler. Clang is an open source compiler that started life as a closed source Apple project. Apple released the code to the open source community in 2007, and the compiler has been adding strengths ever since. This book explains how to install and use Clang on a computer running OS X, Windows, or Linux (Ubuntu). The examples that accompany each chapter have been compiled and tested using Clang 3.5. I chose Clang for this project because it's the compiler that provided support for the most C++14 features when I began to write this book.

The book's accompanying web site can be accessed at `www.apress.com/9781484201589`. You can find source code for all of the executable code listings contained in this book along with makefiles that can be used to build running programs.

CHAPTER 1

Beginning C++

The C++ programming language is a powerful low-level language that allows you to write programs that are compiled into machine instructions to be executed on a computer's processor. This makes C++ different from newer languages such as C# and Java. These languages are interpreted languages. This means they are not executed directly on the processor but instead are sent to another program that is responsible for operating the computer. Java programs are executed using the Java virtual machine (JVM), and C# programs are executed by the Common Language Runtime (CLR).

Thanks to C++ being a language that is compiled ahead of time, it still finds wide use in fields where absolute performance is paramount. The most obvious area where C++ is still the most predominantly used programming language is the video game industry. C++ allows programmers to write applications that take full advantage of the underlying system architecture. You might become familiar with phrases such as *cache coherency* while pursuing a career as a C++ programmer. There aren't many other languages that allow you to optimize your applications to suit the individual processors that your program is being designed to run on. This book introduces you to some of the pitfalls that can affect the performance of your applications at different times and shows you some techniques to tackle those issues.

Modern C++ is in a period where the language is seeing continual updates to its features. This has not always been the case. Despite being around since the early 1980s the C++ programming language was only standardized in 1998. A minor update and clarification of this standard was released in 2003 and is known as C++03. The 2003 update did not add any new features to the language however it did clarify some of the existing features that had gone overlooked. One of these was an update to the standard for the STL vector template to specify that the members of a vector should be stored contiguously in memory. The C++11 standard was released in 2011 and saw a massive update to the C++ programming language. C++ gained features for generalized type deduction system outside of templates, lambda and closure support, a built-in concurrency library and many more features. C++14 brings a smaller update to the language and generally builds upon the features already supplied by C++14. Features such as auto return type deduction from functions have been cleaned up, lambdas have been updated with new features and there are some new ways to define properly typed literal values.

This book strives to write portable, standards compliant C++14 code. At the time of writing it's possible to write C++14 code on Windows, Linux and OS X machines so long as you use a compiler that provides all of the language features. To this end, this book will use Clang as the compiler on Windows and Ubuntu and will use Xcode on OS X. The rest of this chapter focuses on the software you need to write programs in C++ before showing you how to acquire some of the more common options available for Windows, OS X, and Linux operating systems.

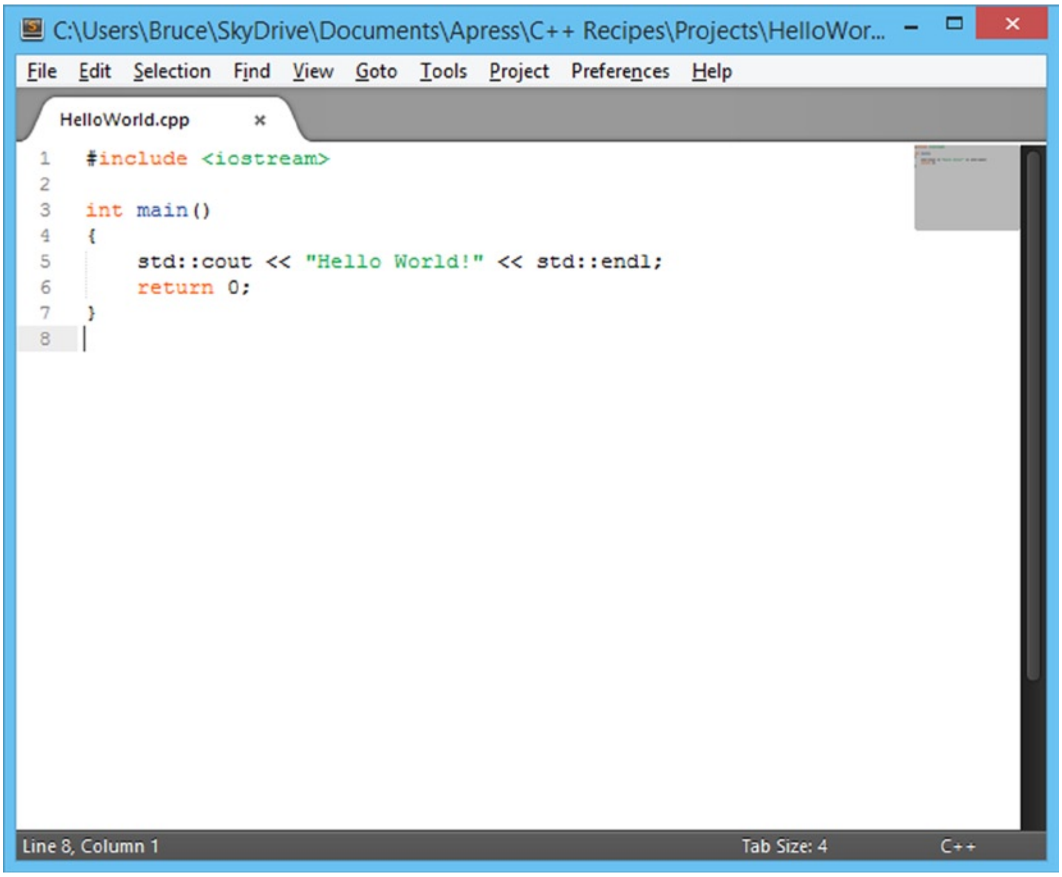
Recipe 1-1. Finding a Text Editor

Problem

C++ programs are constructed from lots of different source files that must be created and edited by one or more programmers. Source files are simply text files, which usually come in two different types: header files and source files. Header files are used to share information about your types and classes between different files, and source files are generally used to contain the methods and the actual executable code that makes up your program.

Solution

A text editor then becomes the first major piece of software you require to begin writing C++ programs. There are many excellent choices of text editors available on different platforms. My best two picks at the moment are the free Notepad++ for Windows and Sublime Text 2, which despite not being free is available on all major operating systems. Figure 1-1 shows a screenshot from Sublime Text 2. Vim and gvim are also very good options that are available for all three operating systems. These editors provide many powerful features and are excellent choices for someone willing to learn.


C:\Users\Bruce\SkyDrive\Documents\Apress\C++ Recipes\Projects\HelloWor...
File Edit Selection Find View Goto Tools Project Preferences Help
HelloWorld.cpp
Line 8, Column 1
Tab Size: 4
C++

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}

```

Figure 1-1. A screenshot from the Sublime Text 2 Editor

■ **Note** Don't feel the urge to grab a text editor straight away. Some of the recipes later in this chapter cover integrated development environments (IDEs) that include all the software you need to write, build, and debug C++ applications.

Figure 1-1 shows one of the most important features of a good text editor: it should be able to highlight the different types of keywords in your source code. You can see in the simple Hello World program in Figure 1-1 that Sublime Text 2 is capable of highlighting the C++ keywords `include`, `int`, and `return`. It has also added different-colored highlights to the main function `name` and the strings `<iostream>` and `"Hello World!"`. Once you have some experience writing code with your text editor of choice, you will become adept at scanning your source files to zero in on the area of code you are interested in, and syntax highlighting will be a major factor in this process.

Recipe 1-2. Installing Clang on Ubuntu

Problem

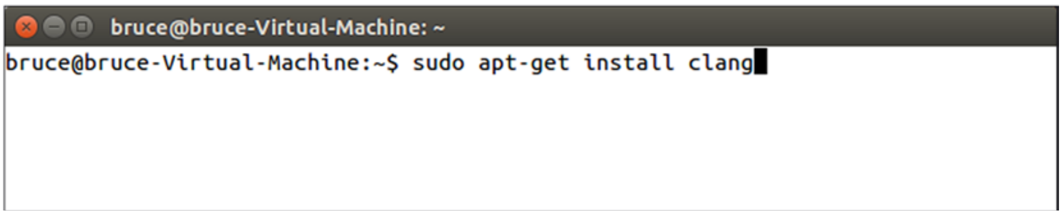
You would like to build C++ programs that support the latest C++14 language features on a computer system running Ubuntu.

Solution

The Clang compiler supports all of the latest C++14 language features and the libstdc++ library supports all of the C++14 STL features.

How It Works

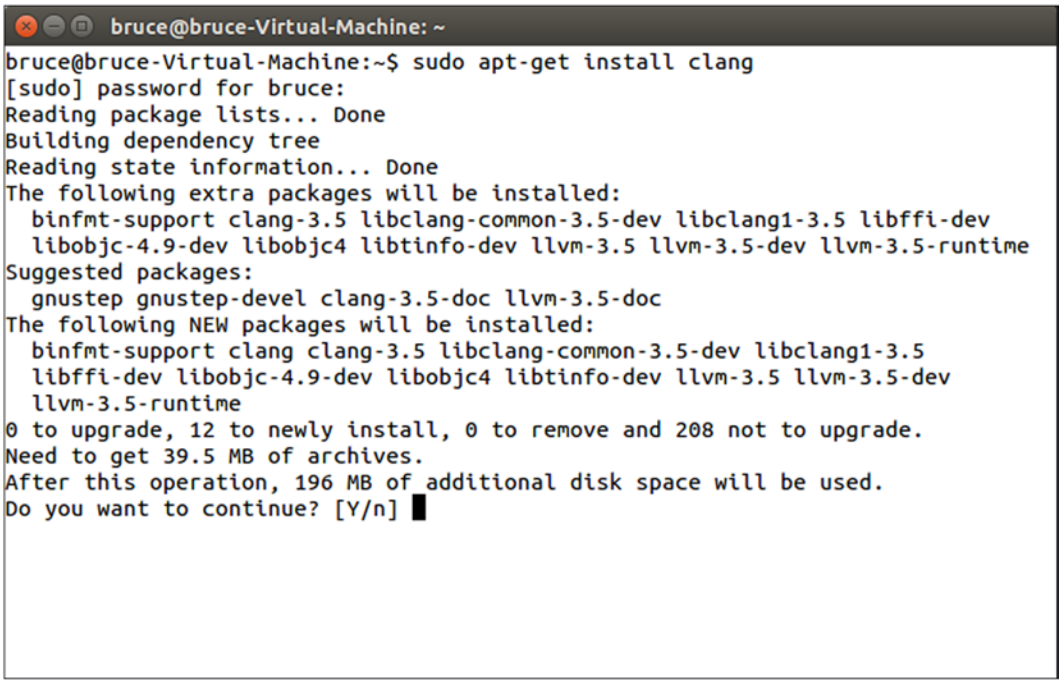
The Ubuntu operating system comes configured with package repositories that allow you to install Clang without much difficulty. You can achieve this using the `apt-get` command in a Terminal window. Figure 1-2 shows the command that you should enter to install Clang.



```
bruce@bruce-Virtual-Machine:~$ sudo apt-get install clang
```

Figure 1-2. *An Ubuntu Terminal window showing the command needed to install Clang*

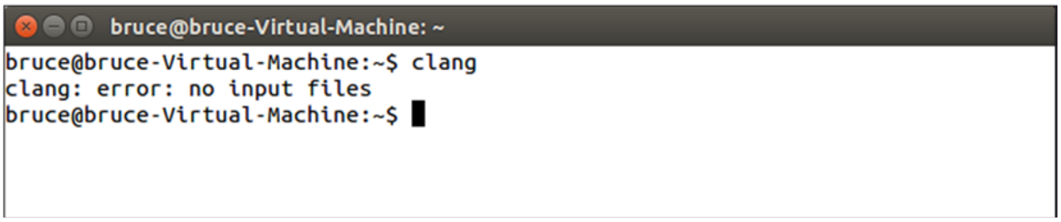
To install Clang you can enter the following command on the command line `sudo apt-get install clang`. Running this command will cause Ubuntu to query its repositories and work out all of the dependencies needed to install Clang. You will be prompted once this process has been completed to confirm that you wish to install Clang and its dependencies. You can see this prompt in Figure 1-3.



```
bruce@bruce-Virtual-Machine: ~
bruce@bruce-Virtual-Machine:~$ sudo apt-get install clang
[sudo] password for bruce:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  binfmt-support clang-3.5 libclang-common-3.5-dev libclang1-3.5 libffi-dev
  libobjc-4.9-dev libobjc4 libtinfo-dev llvm-3.5 llvm-3.5-dev llvm-3.5-runtime
Suggested packages:
  gnustep gnustep-devel clang-3.5-doc llvm-3.5-doc
The following NEW packages will be installed:
  binfmt-support clang clang-3.5 libclang-common-3.5-dev libclang1-3.5
  libffi-dev libobjc-4.9-dev libobjc4 libtinfo-dev llvm-3.5 llvm-3.5-dev
  llvm-3.5-runtime
0 to upgrade, 12 to newly install, 0 to remove and 208 not to upgrade.
Need to get 39.5 MB of archives.
After this operation, 196 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figure 1-3. *The apt-get dependency confirmation prompt*

At this point you can hit enter to continue as yes is the default option. Ubuntu will then download and install all of the software needed for you to be able to install Clang on your computer. You can confirm that this has been successful by running the `clang` command. Figure 1-4 shows what this should look like if everything was successful.



```
bruce@bruce-Virtual-Machine: ~
bruce@bruce-Virtual-Machine:~$ clang
clang: error: no input files
bruce@bruce-Virtual-Machine:~$
```

Figure 1-4. *A successful Clang installation in Ubuntu*

Recipe 1-3. Installing Clang on Windows

Problem

You would like to build C++14 based programs on the Windows operating system.

Solution

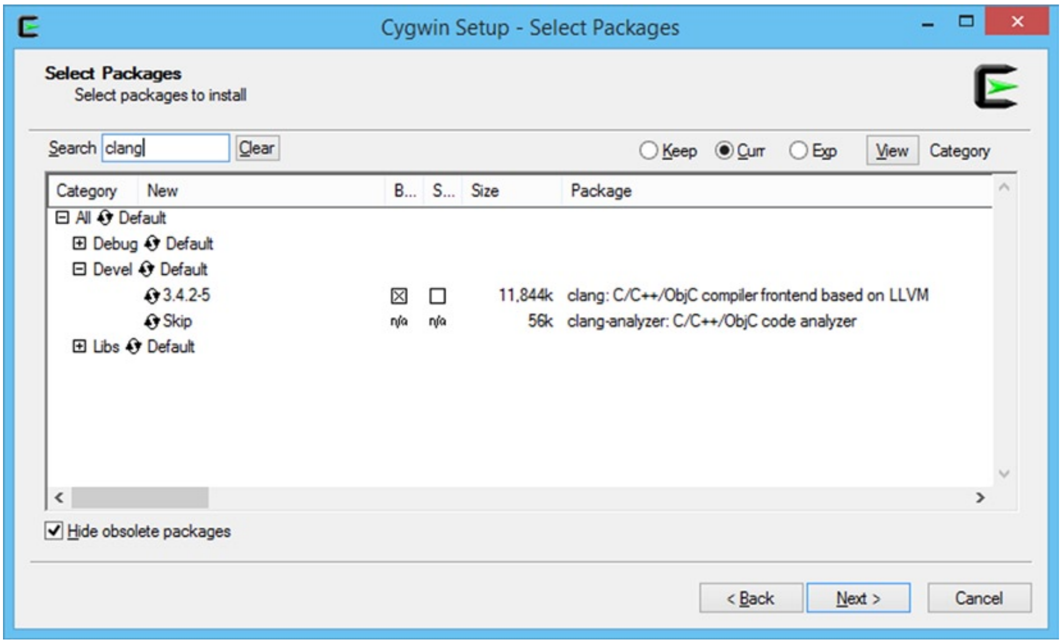
You can use Cygwin for Windows to install Clang and build applications.

How It Works

Cygwin provides a Unix-like command line environment for Windows computers. This is ideal for building programs using Clang as the Cygwin installed comes pre-configured with package repositories that include everything you need to install and use Clang on Windows computers.

You can get a Cygwin installer executable from the Cygwin website at http://www.cygwin.com. Be sure to download the 32bit version of the Cygwin installer as the default packages supplied by Cygwin currently only work with the 32bit environment.

Once you have downloaded the installer you should run it and click through until you are presented with the list of packages to install. At this point you want to select the Clang, make and libstdc++ packages. Figure 1-5 shows the Cygwin installer with the Clang package selected.



Search: clang

Keep [✓ Curr] Exp

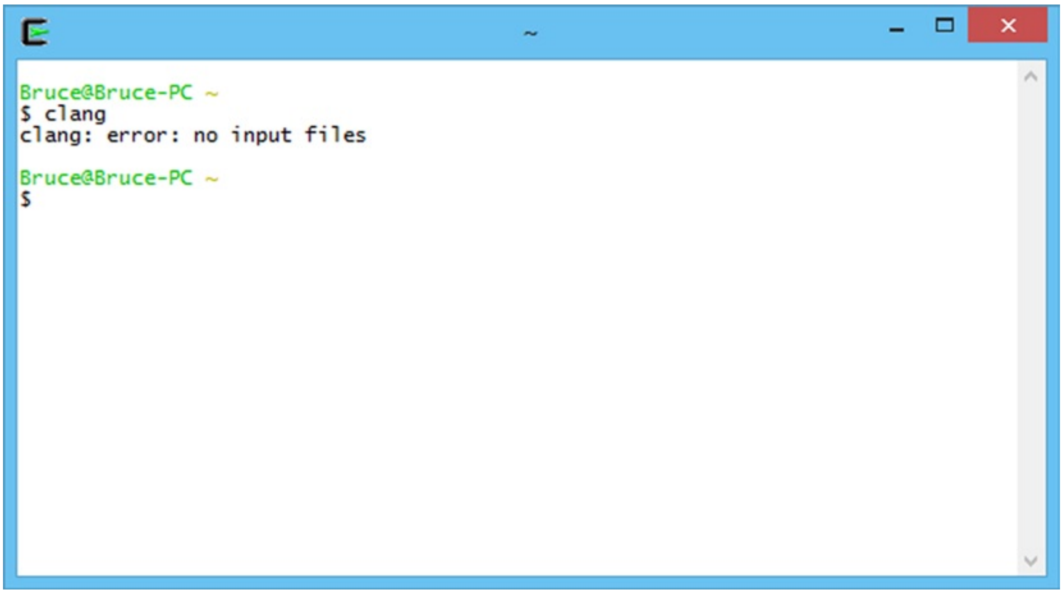
Category	New	B...	S...	Size	Package
⊟ All Default					
⊞ Debug Default					
⊟ Devel Default					
	3.4.2-5	☒	☐	11,844k	clang: C/C++/ObjC compiler frontend based on LLVM
	Skip	n/a	n/a	56k	clang-analyzer: C/C++/ObjC code analyzer
⊞ Libs Default					

☑ Hide obsolete packages

Figure 1-5. *Filtering the Clang package in the Cygwin installer*

Packages can be marked for installation in the installer by clicking on the Skip area on the line for the package. Clicking skip once moves the package version to the latest. You should select the latest packages for Clang, make and libstdc++. Once you have selected all 3 you can click Next to be taken to a window asking to confirm the installation of the dependencies needed by these three packages.

Once you have successfully downloaded and installed all of the packages that you needed to be able to run Clang you can check that it was successful by opening a Cygwin terminal and typing the `clang` command. You can see the result of this output in Figure 1-6.



```
Bruce@Bruce-PC ~
$ clang
clang: error: no input files

Bruce@Bruce-PC ~
$
```

Figure 1-6. *Successfully running Clang in a Cywgin environment in Windows*

Recipe 1-4. Installing Clang on OS X

Problem

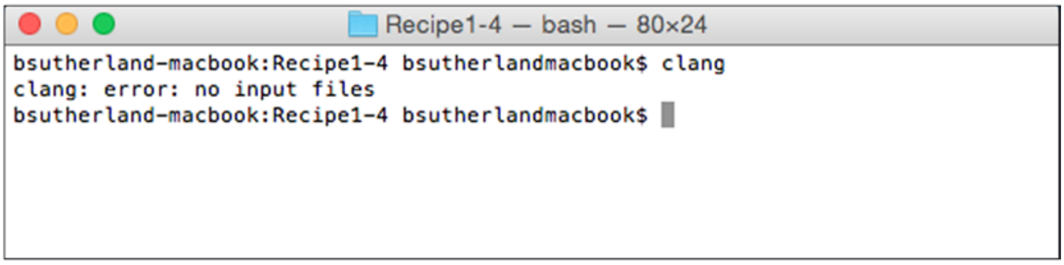
You would like to build C++14 based programs on a computer running OS X.

Solution

Apple's Xcode IDE comes with Clang as its default compiler. Installing Xcode from the OS X App Store also installs Clang.

How It Works

Install the latest version of Xcode from the App Store on your OS X computer. Once you've installed Xcode you can open a Terminal window using Spotlight and type clang to see that the compiler has been installed. Figure 1-7 shows how this should look.



```
bsutherland-macbook:Recipe1-4 bsutherlandmacbook$ clang
clang: error: no input files
bsutherland-macbook:Recipe1-4 bsutherlandmacbook$ 
```

Figure 1-7. *Running Clang on OS X after installing Xcode*

Recipe 1-5. Building Your First C++ Program

Problem

You would like to use your computer to generate executable applications from C++ source code that you write.

Solution

Generating executables from a C++ source file involves two steps; compiling and linking. The steps undertaken in Recipe 1-2, Recipe 1-3 or Recipe 1-4 depending on your operating system will have resulted in you having all of the software you need to build applications from C++14 source files. You are now ready to build your first C++14 program. Create a folder to contain you project and add a text file named HelloWorld.cpp. Enter the code from Listing 1-1 into the file and save.

Listing 1-1. Your first C++14 Program

```
#include <iostream>

#include <string>

int main(void)
{
    using namespace std::string_literals;

    auto output = "Hello World!"s;
    std::cout << output << std::endl;

    return 0;
}
```

The code in Listing 1-1 is a C++ program that will only compile when using a C++14 compatible compiler. The Recipes 2-4 in this chapter contain instructions on how you can obtain a compiler that can be used to compile C++14 code for Windows, Ubuntu and OS X. You can build a working application once you have created a folder and the source file containing the code in Listing 1-1. You do this using a makefile. Create a file named makefile in the folder alongside your HelloWorld.cpp file. The makefile should not have a file extension which may seem a little strange to developers used to the Windows operating system however this is completely normal for Unix based operating systems such as Linux and OS X. Enter the code from Listing 1-2 into your makefile.

Listing 1-2. The makefile Needed to Build the Code in Listing 1-1

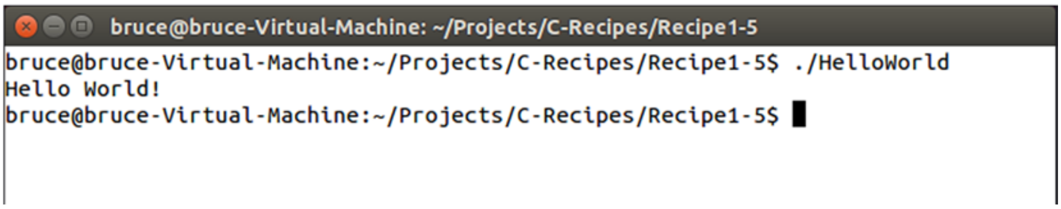
```
HelloWorld: HelloWorld.cpp
	clang++ -g -std=c++1y HelloWorld.cpp -o HelloWorld
```

■ **Note** The whitespace before the `clang++` command in Listing 1-2 is a tab. You cannot replace the tab with spaces as `make` will fail to build. Ensure that your recipes in a makefile always begin with tabs.

The text in Listing 1-2 consists of the instructions needed to build an application from your HelloWorld.cpp source file. The first word on the first line is the name of the target of the makefile. This is the name that the application executable will be given when the building process has been completed. In this case we will be building an executable named HelloWorld. This is followed by the prerequisites needed to build the program. Here you have listed HelloWorld.cpp as the only prerequisite as it is the only source file used to build the executable.

The target and prerequisites are then followed by a list of recipes that are carried out in order to build your application. In this small example you have a line that invokes the clang++ compiler to generate executable code from the HelloWorld.cpp file. The parameter passed to `clang++` using `-std=c++1y` asks Clang to build using the C++14 language standard and the `-o` switch specifies the name of the object output file generated by the compilation process.

Browse to the folder you created to store the source file and makefile using a command shell such as cmd on Windows or Terminal on Linux or OS X and type make. This will invoke the GNU make program and will automatically read and execute your makefile. This will output an executable file into the same folder that you can then run from the command line. You should be able to do this now and see that the text Hello World is output on your command line. Figure 1-8 shows what this would look like in an Ubuntu Terminal window.



```
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe1-5$ ./HelloWorld
Hello World!
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe1-5$ 
```

Figure 1-8. *The Output Generated by Runnung HelloWorld in an Ubuntu Terminal*

Recipe 1-6. Debugging C++ programs using GDB in Cygwin or Linux

Problem

You are writing a C++14 program and would like to be able to debug the application from the command line.

Solution

Both Cygwin for Windows and Linux based operating systems like Ubuntu can install and use the GDB command line debugger for C++ applications.

How It Works

You can use the Cygwin installer for Windows or the Package Manager installed with your favorite Linux distribution to install the GDB debugger. This will give you a command line C++ debugger that can be used to inspect the functionality of your C++ programs. You can practice this using the source, makefile and application generated as part of Recipe 1-5. To generate debugging information for your program you should update the makefile to contain he contents of Listing 1-3 and run make to generate a debuggable executable file.

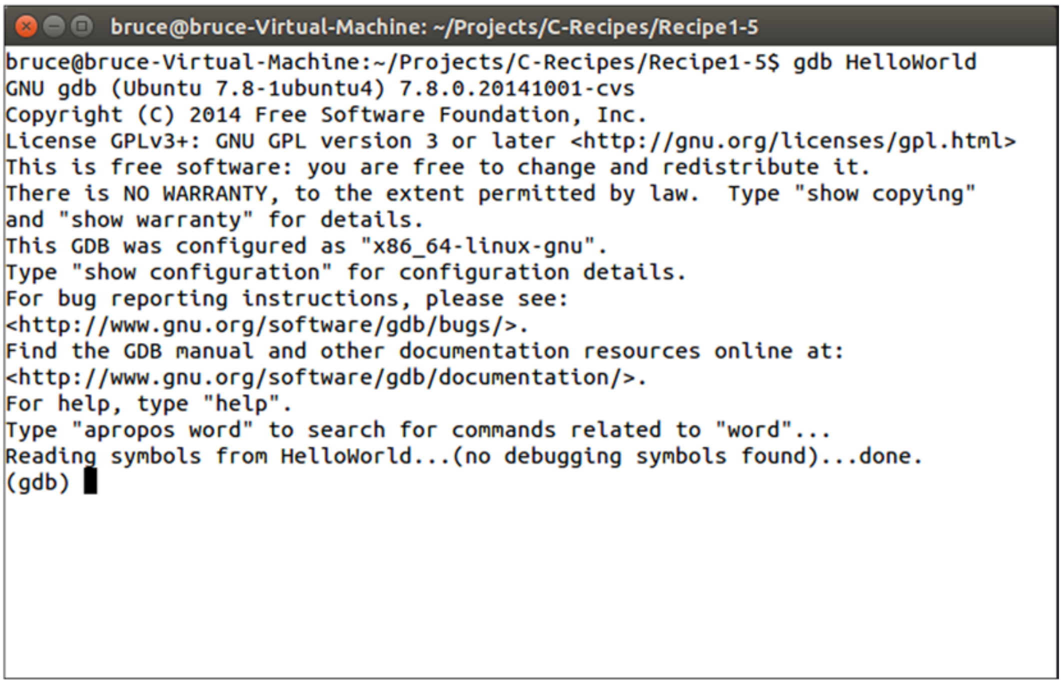
Listing 1-3. A makefile to Generate a Debuggable Program

```
HelloWorld: HelloWorld.cpp
	clang++ -g -std=c++1y HelloWorld.cpp -o HelloWorld
```

Once you have followed Recipe 1-5, updated the makefile to contain the contents of Listing 1-5 and generated an executable you can run GDB on your application by browsing to the folder on your command line and typing `gdb HelloWorld`. The new `-g` switch passed to Clang in the makefile from Listing 1-3 asks the compiler to generate additional information in the application that helps debuggers to provide you with accurate information about the program while it is executing in the debugger.

■ **Note** You may be presented with a notice informing you that your program is already up to date if you had built previously. Simply delete the existing executable file if this occurs.

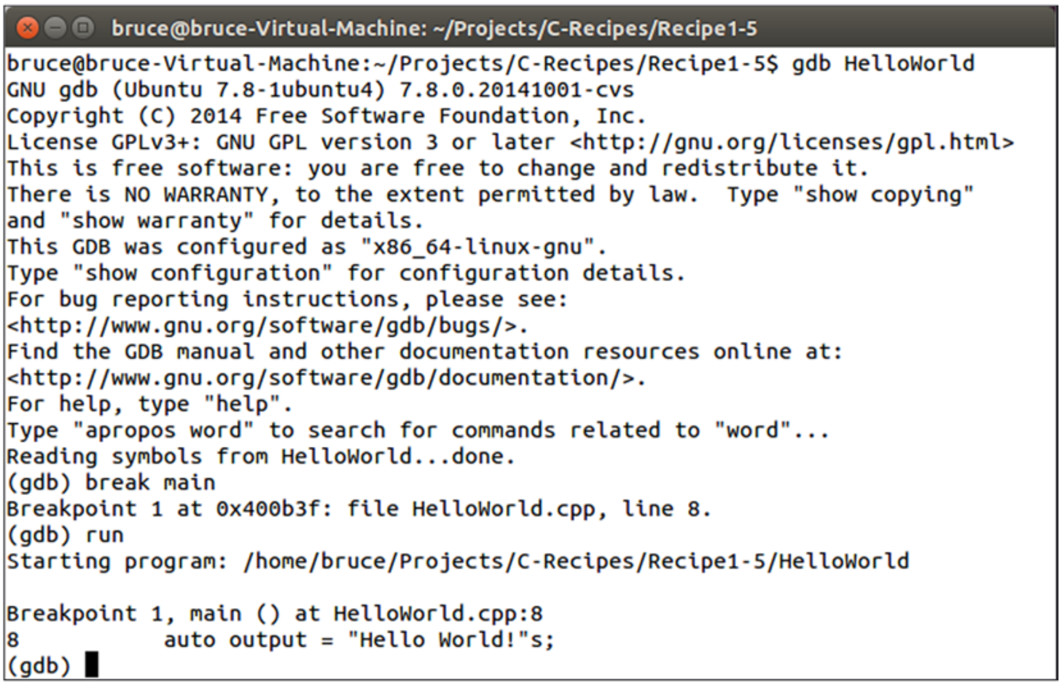
Running GDB in HelloWorld should result in your command line running GDB and providing output such as that shown in Figure 1-9.



```
bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe1-5
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe1-5$ gdb HelloWorld
GNU gdb (Ubuntu 7.8-1ubuntu4) 7.8.0.20141001-cvs
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 HelloWorld...(no debugging symbols found)...done.
(gdb)
```

Figure 1-9. *A Running Instance of GDB*

You now have a running debugger that you can use to inspect the running program while it is executing. The program has not yet begun when GDB first starts, this allows you to configure some breakpoints before you get started. To set a breakpoint you can use the `break` command or the `b` shorthand for the same command. Type `break main` into the GDB command prompt and hit enter. This should result in GDB echoing the command back to you along with the address of the program where the breakpoint was set and the filename and line number it detected for the function supplied. You can now type run into your window to execute the program and have GDB halt at your breakpoint. The output should resemble that shown in Figure 1-10.



```
bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe1-5
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe1-5$ gdb HelloWorld
GNU gdb (Ubuntu 7.8-1ubuntu4) 7.8.0.20141001-cvs
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 HelloWorld...done.
(gdb) break main
Breakpoint 1 at 0x400b3f: file HelloWorld.cpp, line 8.
(gdb) run
Starting program: /home/bruce/Projects/C-Recipes/Recipe1-5/HelloWorld

Breakpoint 1, main () at HelloWorld.cpp:8
8           auto output = "Hello World!"s;
(gdb) █
```

Figure 1-10. *The Output as Seen When GDB Halts at the Breakpoint Set in* `main`

At this point you have several options that allow you to continue the execution of your program. You can see a list of the most common commands below.

`step`

> The `step` command is used to step into a function that is to be called at the current line.

`next`

> The `next` command is used to step over the current line and stop on the next line of the same function.

`finish`

> The `finish` command is used to execute all of the code remaining in the current function and stop on the next line in the function that called the current function.

`print <name>`

> The `print` command followed by the name of a variable can be used to print the value of a variable in your program.

`break`

> The `break` command can be used with a line number, a function name or a source file and line number to set a breakpoint in your programs source code.

`continue`

The `continue` command is used to resume code execution after it has been halted at a breakpoint.

`until`

The `until` command can continue execution from a loop and stop on the first line immediately after the loop execution has finished.

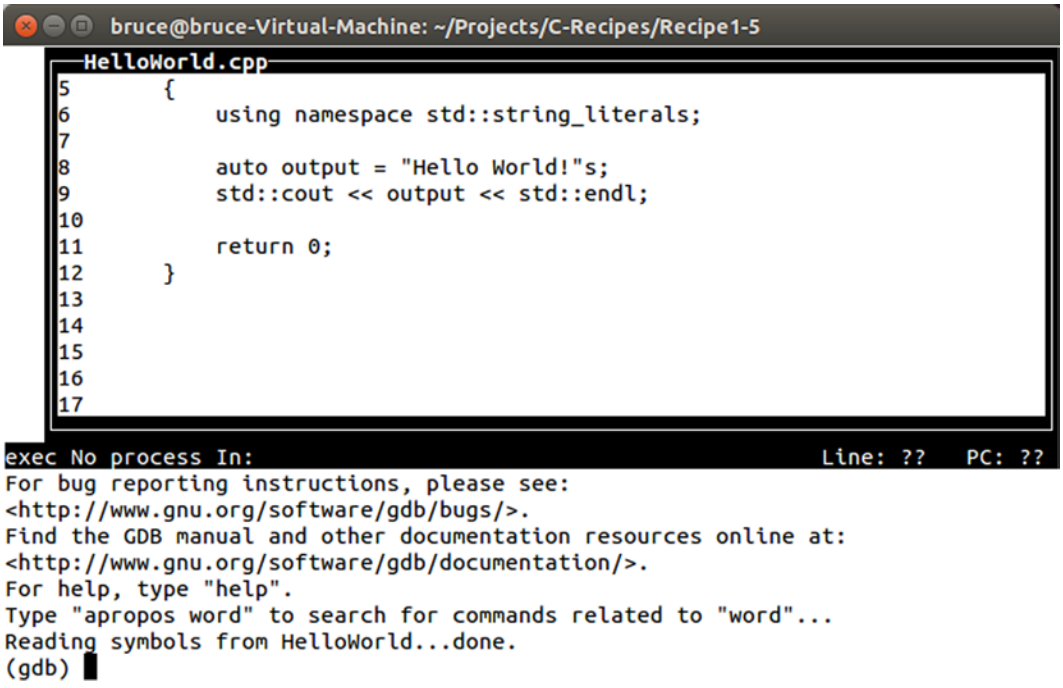
`info`

The `info` command can be used with either the `locals` command or the `stack` command to show information about the current local variables or stack state in the program.

`help`

You can type `help` followed by any command to have GDB give you information about all of the different ways that a given command can be used.

The GDB debugger can also be run with the command `-tui`. This will give you a view of the source file you are currently debugging at the top of the window. You can see how this looks in Figure 1-11.



```
bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe1-5
HelloWorld.cpp
5       {
6           using namespace std::string_literals;
7
8           auto output = "Hello World!"s;
9           std::cout << output << std::endl;
10
11          return 0;
12      }
13
14
15
16
17
exec No process In:                                        Line: ??   PC: ??
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 HelloWorld...done.
(gdb)
```

Figure 1-11. *GDB with a Source Window*

Recipe 1-7. Debugging Your C++ Programs on OS X

Problem

The OS X operating system does not provide any easy method for installing and using GDB.

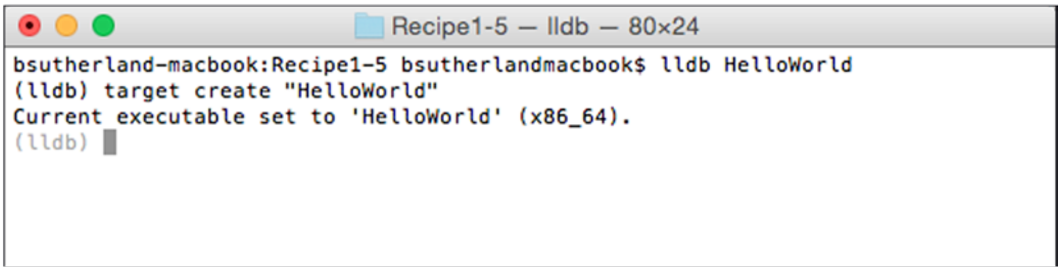
Solution

Xcode comes with the LLDB debugger than can be used on the command line in-place of GDB.

How It Works

The LLDB debugger is, in essence, very similar to the GDB debugger used in Recipe 1-6. Changing between GDB and LLDB is simply a case of learning how to carry out the same simple tasks in both by using the commands provided by each to carry out the same task.

You can execute LLDB on your HelloWorld executable by browsing to the directory containing HelloWorld in Terminal and typing `lldb HelloWorld`. This will give you output that resembles that of Figure 1-12.



```
bsutherland-macbook:Recipe1-5 bsutherlandmacbook$ lldb HelloWorld
(lldb) target create "HelloWorld"
Current executable set to 'HelloWorld' (x86_64).
(lldb) 
```

Figure 1-12. *The LLDB Debugger Running in an OS X Terminal*

■ **Note** You will need to compile your program using the `-g` switch. Take a look at Listing 1-3 to see where this goes if you are unsure.

Once you have LLDB running as shown in Listing 1-12 you can set a breakpoint on the first line of main by typing `breakpoint set -f HelloWorld.cpp -l 8`, or `b main` as shorthand. You can use the `run` command to begin execution and have it halt at the breakpoint that you've just set. When the program stops you can use the `next` command to step over the current line and halt on the next line. You could have used the `step` command to step into a function on the current line and halt on the first line of the function. The `finish` command will step out of the current function.

You can quit LLDB by typing `q` and hitting enter. Restart LLDB and type `breakpoint set -f HelloWorld.cpp -l 9`. Follow this with the `run` command and LLDB should print the source around the line where the application has stopped. You can now type `print output` to see the value stored by the output variable. You can also use the `frame variable` command to see all of the local variables in the current stack frame.

These simple commands will allow you to use the LLDB debugger adequately enough while working through the samples provided along with this book. The following list can be used as a handy cheat sheet while working with LLDB.

`step`

> The `step` command is used to step into a function that is to be called at the current line.

`next`

> The `next` command is used to step over the current line and stop on the next line of the same function.

`finish`

> The `finish` command is used to execute all of the code remaining in the current function and stop on the next line in the function that called the current function.

`print <name>`

> The `print` command followed by the name of a variable can be used to print the value of a variable in your program.

`breakpoint set --name <name>`

`breakpoint set -file <name> --line <number>`

> The `breakpoint` command can be used with a line number, a function name or a source file and line number to set a breakpoint in your programs source code.

`help`

> You can type `help` followed by any command to have GDB give you information about all of the different ways that a given command can be used.

Recipe 1-8. Switching C++ Compilation Modes

Problem

You would like to be able to switch between the different C++ standards before compiling your programs.

Solution

The `std` switch is supplied by Clang so that you can specify the C++ standard to be used when compiling.

How It Works

Clang builds with the C++98 standard by default. You can use the std argument with Clang++ to tell the compiler to use a standard other than the default. Listing 1-4 shows a makefile that is configured to build a program using the C++14 standard.

Listing 1-4. Building with C++14

```
HelloWorld: HelloWorld.cpp
        clang++ -std=c++1y HelloWorld.cpp -o HelloWorld
```

The makefile in Listing 1-4 shows how you can specify that Clang should build your source file using C++14. This example was written using Clang 3.5 that uses the `c++1y` command to represent C++14.

Listing 1-5 shows how you can build a program using C++11.

Listing 1-5. Building with C++11

```
HelloWorld: HelloWorld.cpp
        clang++ -std=c++11 HelloWorld.cpp -o HelloWorld
```

In Listing1-5 you want to use the `c++11` option with the `std` switch to build with C++11. Finally, Listing 1-6 shows how to configure Clang to explicitly build with C++98.

Listing 1-6. Building with C++98

```
HelloWorld: HelloWorld.cpp
        clang++ -std=c++98 HelloWorld.cpp -o HelloWorld
```

The makefile in Listing 1-6 can be used to explicitly build with C++98. You can achieve the same result by leaving out the `std` command altogether and Clang will build using C++98 by default.

■ **Note** It's not guaranteed that every compiler will use C++98 by default. Check with your compiler's documentation if you're unsure which standard is the default. You can also be adventurous with Clang and enable its experimental C++17 support using the `c++1z` option!

Recipe 1-9. Building with the Boost Library

Problem

You would like to write a program using the Boost library.

Solution

Boost is supplied as source code that can be included with and compiled into your application.

How It Works

Boost is a large C++ library that includes all sorts of great functionality. Coverage of the entire library is out of the scope of this book; however the string formatting library will be used. You can acquire the Boost library from the Boost website at `http://www.boost.org/`.

You will be able to get a compressed folder from the Boost website that contains the latest version of the Boost library. The only folder you absolutely need to be able to include basic boost functionality is the boost folder itself. I have downloaded Boost 1.55 and therefore I have created a folder inside my project folder named `boost_1_55_0` and copied the boost folder into this location from the downloaded version.

Once you have a project folder set up with a downloaded copy of Boost you can include Boost header files into your source code. Listing 1-7 shows a program that uses the `boost::format` function.

Listing 1-7. Using `boost::format`

```
#include <iostream>
#include "boost/format.hpp"

using namespace std;

int main()
{
    std::cout << "Enter your first name: " << std::endl;
    std::string firstName;
    std::cin >> firstName;

    std::cout << "Enter your surname: " << std::endl;
    std::string surname;
    std::cin >> surname;

    auto formattedName = str( boost::format("%1% %2%"s) % firstName % surname );
    std::cout << "You said your name is: " << formattedName << std::endl;

    return 0;
}
```

The code in Listing 1-7 shows how you can include a Boost header into a source file and how that file's functions can be used in your program.

■ **Note** Don't worry about how the `format` function works if it's not immediately clear, it is covered in Chapter 3.

You must also tell the compiler where to look for the Boost header files in a makefile otherwise your program will not compile. Listing 1-8 shows the contents of the makefile that can be used to build this program.

Listing 1-8. A makefile to Build with Boost

```
main: main.cpp
        clang++ -g -std=c++1y -Iboost_1_55_0 main.cpp -o main
```

The makefile in Listing 1-8 passes the `-I` option to Clang++. This option is used to tell Clang that you would like to include the given folder in the search paths used when including files using the `#include` directive. As you can see I have passed the `boost_1_55_0` folder that I created in my project folder. This folder contains the boost folder that you can see used when including a Boost header in Listing 1-7.

■ **Note** If you're having trouble getting this example to work and aren't sure of where to put the Boost header files you can download the samples that accompany this book from the `www.apress.com/9781484201589`.

CHAPTER 2

Modern C++

Development of the C++ programming language began in 1979 as the C with Classes language. The name C++ was formally adopted in 1983 and development of the language continued throughout the 1980s and 1990s without the adoption of a formal language standard. This all changed in 1998 when the first ISO standard of the C++ programming language was adopted. There have been three updates to the standard published since that time, one in 2003, again in 2011 and the latest in 2014.

Note The standard published in 2003 was a minor update to the 1998 standard that didn't introduce much in the way of new features. For this reason, it won't be discussed in any great detail in this book.

This book is primarily going to focus on the very latest C++ programming standard, C++14. Whenever I mention the C++ programming language you can be assured that I am talking about the language as described by the current ISO standard. If I am discussing features that were introduced in 2011 then I will explicitly mention the language as C++11 and for any features that were introduced prior to 2011 I will use the name C++98.

This chapter will look at the programming features added to the language in the latest standard and with C++11. Many of the modern features of C++ were added in the C++11 standard and have been expanded with the C++14 standard therefore it is important to be able to identify the differences when working with compilers that support a standard that is not the latest.

Recipe 2-1. Initializing Variables

Problem

You would like to be able to initialize all variables in a standard manner.

Solution

Uniform initialization was introduced in C++11 and can be used to initialize a variable of any type.

How It Works

It's necessary to understand the deficiencies with variable initialization in C++98 to appreciate why uniform initialization is an important language feature in C++11. Listing 2-1 shows a program that contains a single class, `MyClass`.

Listing 2-1. The C++Most Vexing Parse Problem

```
class MyClass
{
private:
    int m_Member;

public:
    MyClass() = default;
    MyClass(const MyClass& rhs) = default;
};

int main()
{
    MyClass objectA;
    MyClass objectB(MyClass());
    return 0;
}
```

The code in Listing 2-1 will generate a compile error in C++ programs. The problem exists in the definition of `objectB`. A C++ compiler will not see this line as defining a variable named `objectB` of type `MyClass` calling a constructor that takes the object constructed by calling the `MyClass` constructor. This is what you might expect the compiler to see however what it actually sees is a function declaration. The compiler thinks that this line is declaring a function named `objectB` that returns a `MyClass` object and has a single, unnamed function pointer to a function that returns a `MyClass` object and is passed no parameters.

Compiling the program shown in Listing 2-1 causes Clang to generate the following warning:

```
main.cpp:14:20: warning: parentheses were disambiguated as a function
      declaration [-Wvexing-parse]
    MyClass objectB(MyClass());
                   ^~~~~~~~~~~
main.cpp:14:21: note: add a pair of parentheses to declare a variable
    MyClass objectB(MyClass());
                    ^
                    (        )
```

The Clang compiler has properly identified that the code entered in Listing 2-1 contains a vexing parse problem and even helpfully suggests wrapping the `MyClass` constructor being passed as a parameter in another pair of parentheses to solve the problem. C++11 has provided an alternative solution in uniform initialization. You can see this in Listing 2-2.

Listing 2-2. Using Uniform Initialization to Solve the Vexing Parse Problem

```
class MyClass
{
private:
    int m_Member;

public:
    MyClass() = default;
    MyClass(const MyClass& rhs) = default;
};

int main()
{
    MyClass objectA;
    MyClass objectB{MyClass{}};
    return 0;
}
```

You can see in Listing 2-2 that uniform initialization replaces parentheses with braces. This syntax change informs the compiler that you would like to use uniform initialization to initialize your variable. Uniform initialization can be used to initialize almost all types of variables.

■ **Note** The paragraph above mentions that uniform initialization can be used to initialize *almost* all variables. It can have trouble when initializing aggregates or plain old data types however you won't need to worry about those for now.

The ability to prevent narrowing conversions is another benefit of using uniform initialization. The code in Listing 2-3 will fail to compile when using uniform initialization.

Listing 2-3. Using Uniform Initialization to Prevent Narrowing Conversions

```
int main()
{
    int number{ 0 };
    char another{ 512 };

    double bigNumber{ 1.0 };
    float littleNumber{ bigNumber };

    return 0;
}
```

The compiler will throw errors when compiling the code in Listing 2-3 as there are two narrowing conversions present in the source. The first occurs when trying to define a char variable with the literal value 512. A char type can store a maximum value of 255 therefore the value 512 would be narrowed into this data type. A C++11 or newer compiler will not compile this code due to this error. The initialization of the float from a double type is also a narrowing conversion. Narrowing conversions occur when data is transferred from one type to another in where the destination type cannot store all of the values represented by the source type. Precision is lost in the case of a double being converted to a float therefore the compiler correctly will not build this code as-is. The code in Listing 2-4 uses a `static_cast` to inform the compiler that the narrowing conversions are intentional and to compile the code.

Listing 2-4. Using a static_cast to Compile Narrowing Conversions

```
int main()
{
    int number{ 0 };
    char another{ static_cast<char>(512) };

    double bigNumber{ 1.0 };
    float littleNumber{ static_cast<float>(bigNumber) };

    return 0;
}
```

Recipe 2-2. Initializing Objects with Initializer Lists

Problem

You would like to construct objects from multiple objects of a given type.

Solution

Modern C++ provides initializer lists that can be used to supply many objects of the same type to a constructor.

How It Works

Initializer lists in C++11 build upon uniform initialization to allow you to initialize complex types with ease. A common example of a complex type that can be difficult to initialize with data is a vector. Listing 2-5 shows two different calls to a standard vector constructor.

Listing 2-5. Constructing vector Objects

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    using MyVector = vector<int>;

    MyVector vectorA( 1 );
    cout << vectorA.size() << " " << vectorA[0] << endl;

    MyVector vectorB( 1, 10 );
    cout << vectorB.size() << " " << vectorB[0] << endl;

    return 0;
}
```

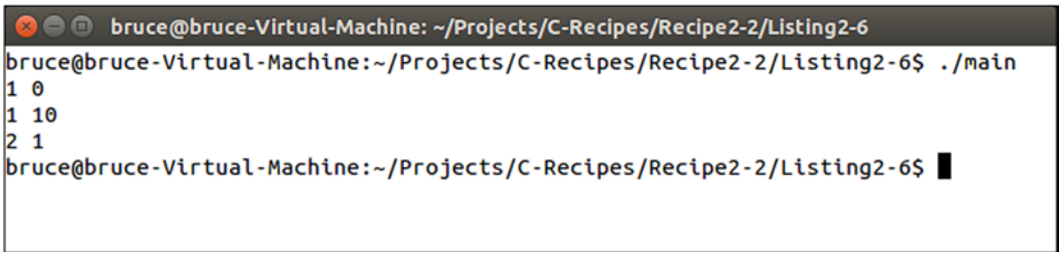
The code in Listing 2-5 might not do what you expect at first glance. The vectorA variable will be initialized with a single int containing 0. You might expect that it would contain a single integer containing 1 but this would be incorrect. The first parameter to a vector constructor determines how many values the initial vector will be set up to store and in this case we are asking it to store a single variable. You might similarly expect vectorB to contain two values, 1 and 10 however what we have here is a vector that contains one value and that value is 10. The vectorB variable is constructed using the same constructor as vectorA however it specifies a value to use to instantiate the members of the vector rather than using the default value.

The code in Listing 2-6 uses an initializer list in conjunction with uniform initialization to construct a vector that contains two elements with the specified values.

Listing 2-6. Using Uniform Initialization to Construct a vector

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    using MyVector = vector<int>;

    MyVector vectorA( 1 );
    cout << vectorA.size() << " " << vectorA[0] << endl;

    MyVector vectorB( 1, 10 );
    cout << vectorB.size() << " " << vectorB[0] << endl;

    MyVector vectorC{ 1, 10 };
    cout << vectorC.size() << " " << vectorC[0] << endl;

    return 0;
}
```

The code in Listing 2-6 creates three different vector objects. You can see the output generated by this program in Figure 2-1.



```
bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe2-2/Listing2-6
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-2/Listing2-6$ ./main
1 0
1 10
2 1
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-2/Listing2-6$
```

Figure 2-1. *The Output Generated by Listing 2-6*

The console output shown in Figure 2-1 shows the size of each `vector` and the value stored in the first element of each `vector`. You can see that the first `vector` contains a single element and that its value is 0. The second `vector` also contains a single element however its value is 10. The third `vector` is constructed using uniform initialization and it contains two values and the value of its first element is 1. The value of the second element will be 10. This can cause a significant different to the behavior of your programs if you are not taking particular care to ensure that the correct type of initialization has been used with your types. The code in Listing 2-7 shows a more explicit use of the `initializer_list` to construct a `vector`.

Listing 2-7. Explicit initializer_list Usage

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    using MyVector = vector<int>;

    MyVector vectorA( 1 );
    cout << vectorA.size() << " " << vectorA[0] << endl;

    MyVector vectorB( 1, 10 );
    cout << vectorB.size() << " " << vectorB[0] << endl;

    initializer_list<int> initList{ 1, 10 };
    MyVector vectorC(initList);
    cout << vectorC.size() << " " << vectorC[0] << endl;

    return 0;
}
```

The code in Listing 2-7 contains an explicit `initializer_list` that is used to construct a `vector`. The code in Listing 2-6 implicitly created this object when constructing a `vector` using uniform initialization. There's usually little need to explicitly create initializer lists like this however it's important that you understand what the compiler is doing when you write code using uniform initialization.

Recipe 2-3. Using Type Deduction

Problem

You would like to write portable code that doesn't have a high maintenance cost when changing types.

Solution

C++ provides the auto keyword that can be used to let the compiler deduce the type for a variable automatically.

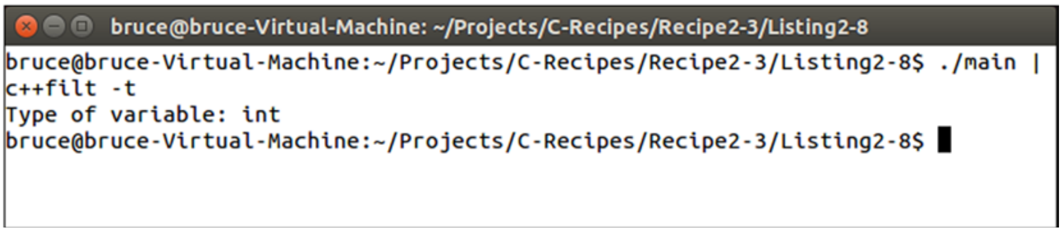
How It Works

C++98 compilers had the ability to automatically deduce the type of a variable however this functionality was only available while you were writing code that used templates and you omitted the type specialization. Modern C++ has extended this type deduction support to many more scenarios. The code in Listing 2-8 shows the use of the `auto` keyword and the `typeid` method of working out the type of a variable.

Listing 2-8. Using the `auto` Keyword

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main()
{
    auto variable = 1;
    cout << "Type of variable: " << typeid(variable).name() << endl;

    return 0;
}
```

The code in Listing 2-8 shows how to create a variable with automatically deduced type in C++. The compiler will automatically work out that you wanted to create an int variable with this code and that's the type that will be output by the program, sort of. The Clang compiler will output its internal representation of an integer type which is actually `i`. You can pass this output to a program named `c++filt` to convert this into a normal typename. Figure 2-2 shows how this can be achieved.


bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe2-3/Listing2-8

```
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-3/Listing2-8$ ./main |
c++filt -t
Type of variable: int
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-3/Listing2-8$ 
```

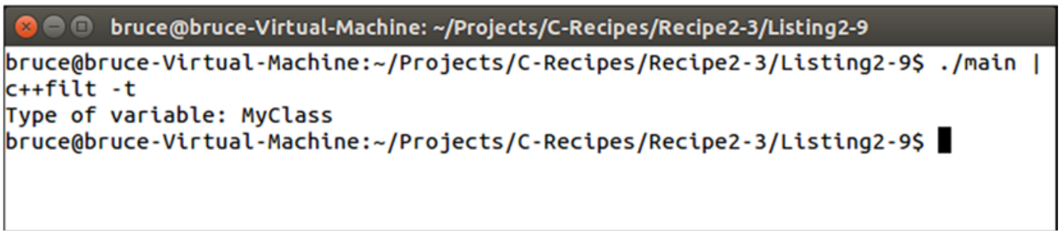
Figure 2-2. *Using c++filt to Produce Proper Type Output From Clang*

The `c++filt` program has successfully converted the Clang type i into a human readable C++ type format. The auto keyword also works with classes. Listing 2-9 shows this.

Listing 2-9. Using `auto` with a `class`

```
#include <iostream>
#include <typeinfo>

using namespace std;

class MyClass
{
};

int main()
{
    auto variable = MyClass();
    cout << "Type of variable: " << typeid(variable).name() << endl;

    return 0;
}
```

This program will print out the name MyClass as you can see in Figure 2-3.


bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe2-3/Listing2-9

```
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-3/Listing2-9$ ./main |
c++filt -t
Type of variable: MyClass
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-3/Listing2-9$
```

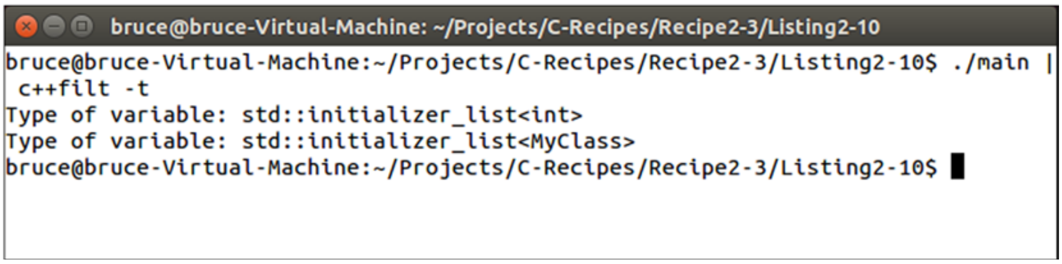
Figure 2-3. *Using `auto` with MyClass*

Unfortunately there are times where the `auto` keyword can produce less than desirable results. You will definitely come unstuck if you try to combine the `auto` keyword with uniform initialization. Listing 2-10 shows the use of the auto keyword with uniform initialization.

Listing 2-10. Using `auto` with Uniform Initialization

```
#include <iostream>
#include <typeinfo>

using namespace std;

class MyClass
{
};

int main()
{
    auto variable{ 1 };
    cout << "Type of variable: " << typeid(variable).name() << endl;
    auto variable2{ MyClass{} };
    cout << "Type of variable: " << typeid(variable2).name() << endl;

    return 0;
}
```

You might expect that the code in Listing 2-10 will produce a variable of type int and a variable of type MyClass however this is not the case. Figure 2-4 shows the output generated by the program.



bruce@bruce-Virtual-Machine: ~/Projects/C-Recipes/Recipe2-3/Listing2-10

```
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-3/Listing2-10$ ./main |
 c++filt -t
Type of variable: std::initializer_list<int>
Type of variable: std::initializer_list<MyClass>
bruce@bruce-Virtual-Machine:~/Projects/C-Recipes/Recipe2-3/Listing2-10$ 
```

Figure 2-4. *Output Generated When using auto with Uniform Initialization*

A quick look at Figure 2-4 shows the immediate problem encountered when using the `auto` keyword along with uniform initialization. The C++ uniform initialization feature automatically creates an `initializer_list` variable that contains the value of the type we want, not the type and value itself. This leads to a relatively simple piece of advice, do not use uniform initialization when defining variables using `auto`. I'd recommend not using `auto` even if the type you want is actually an `initializer_list` as the code is much easier to understand and much less error prone if you don't mix and match you variable initialization styles. There's a final piece of advice to bear in mind, use auto for local variables as much as possible. It's impossible to declare an auto variable and not define it therefore it's impossible to have an undefined local `auto` variable. You can use this piece of knowledge to cut down on one potential source of bugs in your programs.

Recipe 2-4. Using auto with Functions

Problem

You would like to create more generic functions using type deduction to increase code maintainability.

Solution

Modern C++ allows you to use type deduction for function parameters and for return types.

How It Works

C++ allows you to utilize type deduction when working with function using two methods. Types can be deduced for function parameters by creating a template function and calling that function without explicit specializers. The return type can be deduced for a function using the auto keyword in place of its return type. Listing 2-11 shows the use of auto to deduce the return type for a function.

Listing 2-11. Deducing a Function's Return Type Using `auto`

```
#include <iostream>

using namespace std;

auto AutoFunctionFromReturn(int parameter)
{
    return parameter;
}

int main()
{
    auto value = AutoFunctionFromReturn(1);
    cout << value << endl;

    return 0;
}
```

The `AutoFunctionFromReturn` function's return type in Listing 2-11 is automatically deduced. The compiler inspects the type of the variable returned from the function and uses that to deduce the type to be returned. This all works properly because the compiler has everything it needs inside the function to be able to deduce the type. The `parameter` variable is being returned therefore the compiler can use its type as the return type for the function.

Things get a bit more complicated when you need to build with a C++11 compiler. Building Listing 2-11 using C++11 results in the following error.

```
main.cpp:5:1: error: 'auto' return without trailing return type
auto AutoFunctionFromReturn(int parameter)
```

Listing 2-12 includes a function with automatic return type deduction that works in C++11.

Listing 2-12. Return Type Deduction in C++11

```
#include <iostream>

using namespace std;

auto AutoFunctionFromReturn(int parameter) -> int
{
    return parameter;
}

int main()
{
    auto value = AutoFunctionFromReturn(1);
    cout << value << endl;

    return 0;
}
```