

C++ 1: Introduction to C++ ([PDF](#))

Lesson 1: **Introduction**

[Learning C++](#)

[History of C++](#)

[About Eclipse](#)

[Perspectives and the Red Leaf Icon](#)

[Writing Your First Program](#)

[What Does It All Mean?](#)

[Adding to Our Program](#)

[What Goes On Under the Hood](#)

[Finishing the Program](#)

[The User Manual](#)

[The Test Plan](#)

[A Few More Notes](#)

Lesson 2: **Expressions**

[Mathematical Expressions](#)

[Creating the Project and File](#)

[Editing and Running Your Program](#)

[Types of Numbers](#)

[Floating-Point vs. Integer Division](#)

[Dividing by Zero](#)

[Limits on Numbers](#)

[Common Problems](#)

Lesson 3: **Variables**

[Basic Programs](#)

[Variables](#)

[Variable Definitions](#)

[Variable Types: Integer](#)

[Variable Types: Floating Point](#)

[Variable Types: Characters](#)

[Escape Characters](#)

[Wide Characters](#)

[Boolean](#)

[Mixing Types](#)

Lesson 4: Arrays and For Loops

Using Arrays

The const Modifier

Our first array

for loops

Array Safety

Lesson 5: C++ Strings

Strings in C++

Characters in strings

Other Functions

Lesson 6: C-Style Strings

What is a C-Style String?

Concatenation of C-Style Strings

Comparing Strings

Tips

Converting C++ Strings to C-Style Strings

Unsafe String Functions

The future of strcpy() and strcat()

Comparisons to other types

C Strings vs. Arrays of Characters

C-Style vs. C++ Style

Lesson 7: Reading Data and if

Reading Strings

Reading Integers

if Statements

if Abuse

Equality or Assignment?

Blocks

Conditional Shortcuts

Lesson 8: Shortcuts

Operators

For Loops

For Loop Misuse

Side Effects

Lesson 9: While Loops

while, break, and continue

Fibonacci numbers

Lesson 10: Scope

What is Scope?

Global Variables

Storage Class

for Loop Scope

Hidden Variables

Lesson 11: Functions

What is a Function?

Our First Function

Void Functions and Array Parameters

Function Overloading

Default Parameters

Lesson 12: Parameters and Return Types

Passing Parameters

Pass by Value

Array Parameters

Const Parameters

References

Const Return Values

Problems with Reference Returns

Lesson 13: Final Project

Putting It All Together

Assignment

Code Design

Agile Development

Coding Notes

Testing

Revisions

Copyright © 1998-2013 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Introduction

C++ 1: Introduction to C++ Lesson 1

Welcome to the O'Reilly School of Technology's **C++** course! We're glad you've decided to go on this ride with us and learn C++ programming. By the time you finish the course, we're confident that you'll have firm grasp on this really practical programming language.

If you've already taken an O'Reilly School of Technology (OST) course, you're familiar with the *useractive* approach to learning. It's an approach where you (the user) will be active! You'll learn by doing, building live programs, testing them, and then experimenting with them, hands-on!

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type look like this.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to observe.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

Learning C++

C++ is the workhorse language of the programming world. It's used for many different applications, from high-end graphics systems to embedded processing. Chances are you already own two or three computers running C++ programs, only they aren't called computers, they're called cell phones, GPS systems, cameras, or DVD players.

Our course is designed to teach you how to do real-world, practical programming. As such, it will teach not only the best practices when it comes to design and coding, but also how to deal with the "worst practices" that seem to seep into many of the programs out there in the real world.

For this course, we will use the Advanced O'Reilly Learning Sandbox. This system allows you not only to read the lessons, but to interact with the examples. You are encouraged to experiment and try new things, all within the environment of your O'Reilly Learning Sandbox.

Of course, in the real world you'll make mistakes. Typing in a program, finding that it's broken, and sweating over it until 2 o'clock in the morning, only to find you've made a small mistake, is another way of learning. (We hope you will rarely need to use this method of learning!)

History of C++

C++ was born in 1970 when two programmers wanted a "high-level" language for a machine they were working on. They designed a language similar to an old language they had been using called B. In the programming tradition of keeping things simple, they named their language C.

C was a good language for its day. It ran on very limited hardware (4mhz, 64K memory) and did a pretty good job. The language was also designed not to get in the way of the programmer. In other words, if he wanted to do something stupid, it let him.

C is a procedural language. The data and instructions are kept separate. Ten years after its invention, people realized that they could make better programs if they combined data and the instructions that operated on the data into one thing called an object or class.

In 1980 Bjarne Stroustrup started working on a new language called "C with classes." The goal of the language was to bring classes to C while not breaking existing C code (or at least not breaking it too badly).

This language would become C++.

About Eclipse

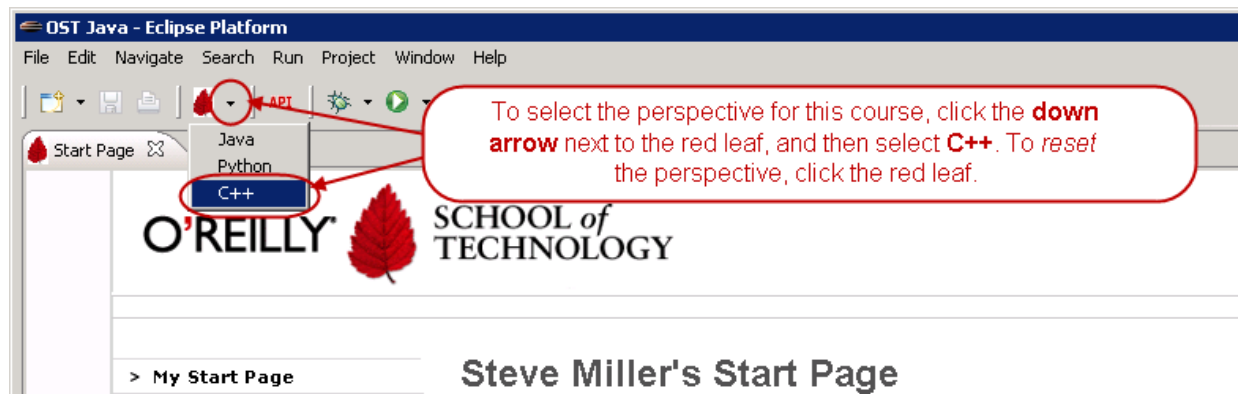
We're using an Integrated Development Environment (IDE) called Eclipse. It's the program filling up your screen right now. IDEs assist programmers by performing many of the tasks that need to be done repetitively. IDEs can also help to edit and debug code, and organize projects.

Perspectives and the Red Leaf Icon

The Ellipse plug-in for Eclipse, developed by the O'Reilly School of Technology, adds an icon to the toolbar in Eclipse. This icon is your "panic button." Eclipse is versatile and lets you move things like views and toolbars. If you ever get confused and want to return to the default perspective (window layout), the Red Leaf icon is the most efficient way to do that.

You can also change perspectives by clicking the drop-down arrow beside the icon, and then clicking a series name (JAVA, PYTHON, C++, etc.). Most of the perspectives look similar, but subtle changes may be present "under the hood," so it's best to use the perspective designed specifically for each course.

For this course, select **C++**:

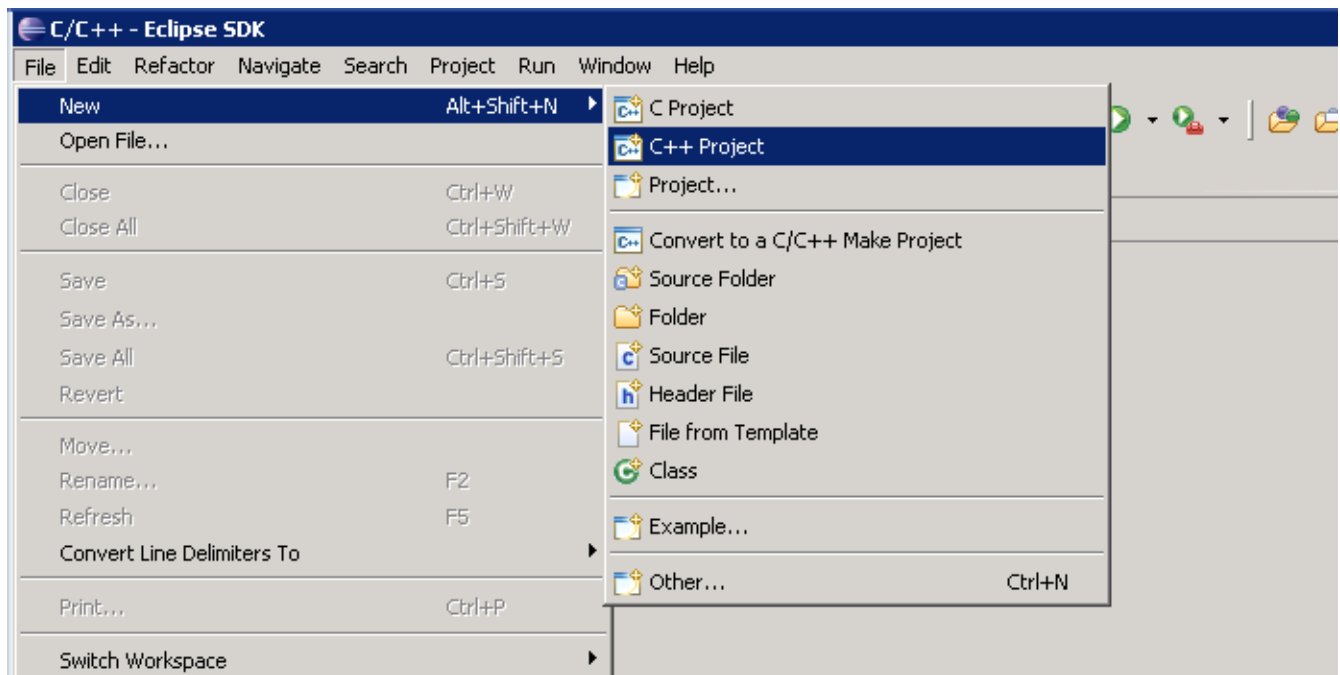


Okay, now that you understand the basic structure of an OST course, you're ready to enter and run code!

Writing Your First Program

We want you to see a working C++ program as soon as possible, so we'll resist the urge to explain too much for this first program while you're creating and running it. Let's get going—we'll fill you in on the details later!

To start your project, select **File | New | C++ Project**, as shown here:

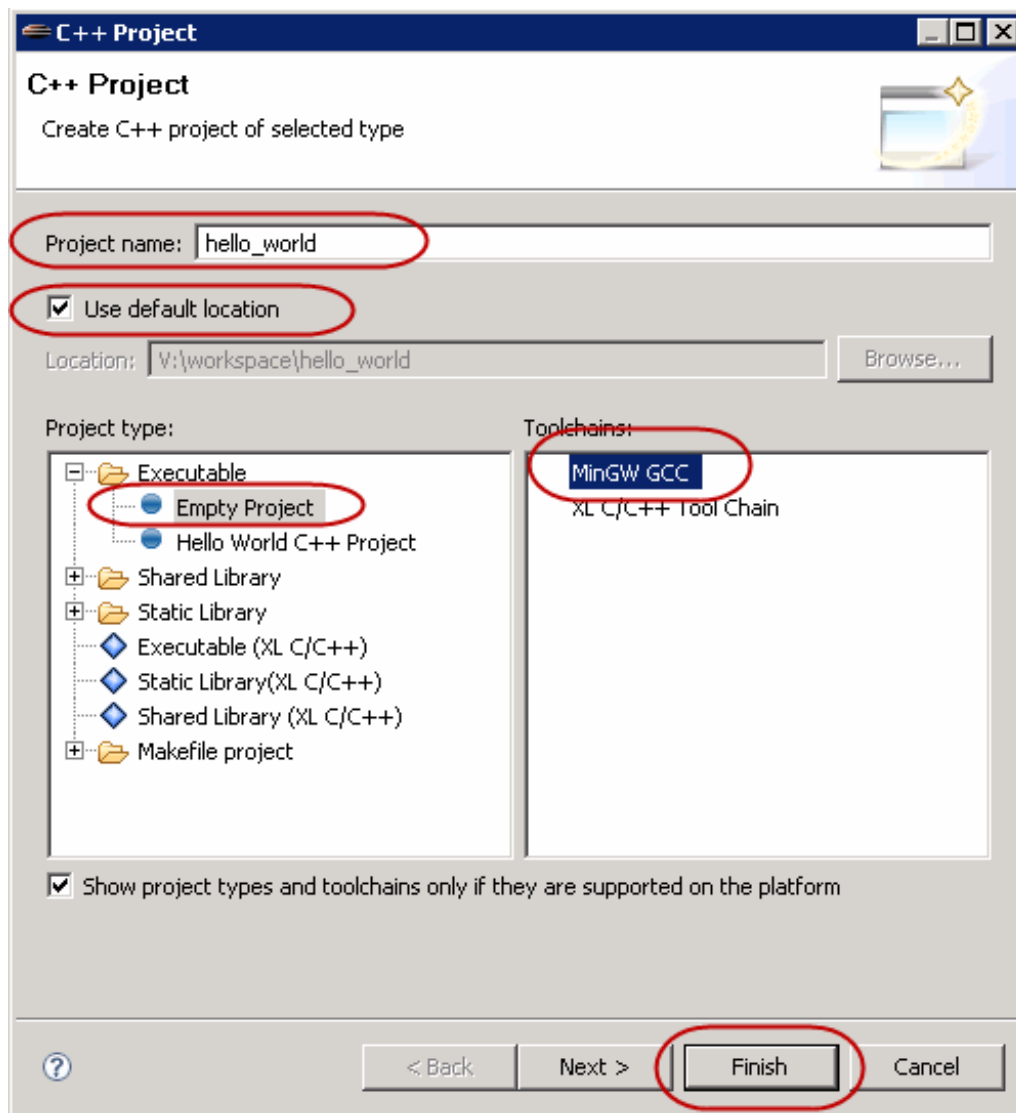


Note If **C++ Project** isn't on the menu, then the perspective hasn't been set properly. Click the down arrow next to the Red Leaf icon and then select **C++**

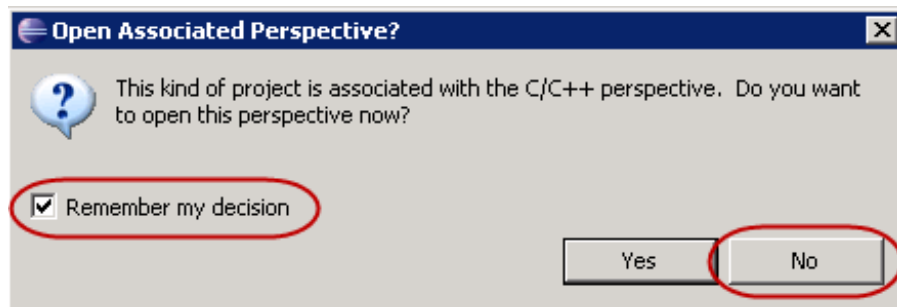
The C++ Project type dialog appears. For Project Name, enter **hello_world**.

Note Due to limitations in the GNU tools, all project names and file names should consist of only letters, digits, and underscores. Do not use punctuation or spaces in a name. It confuses the tools and causes things to break.

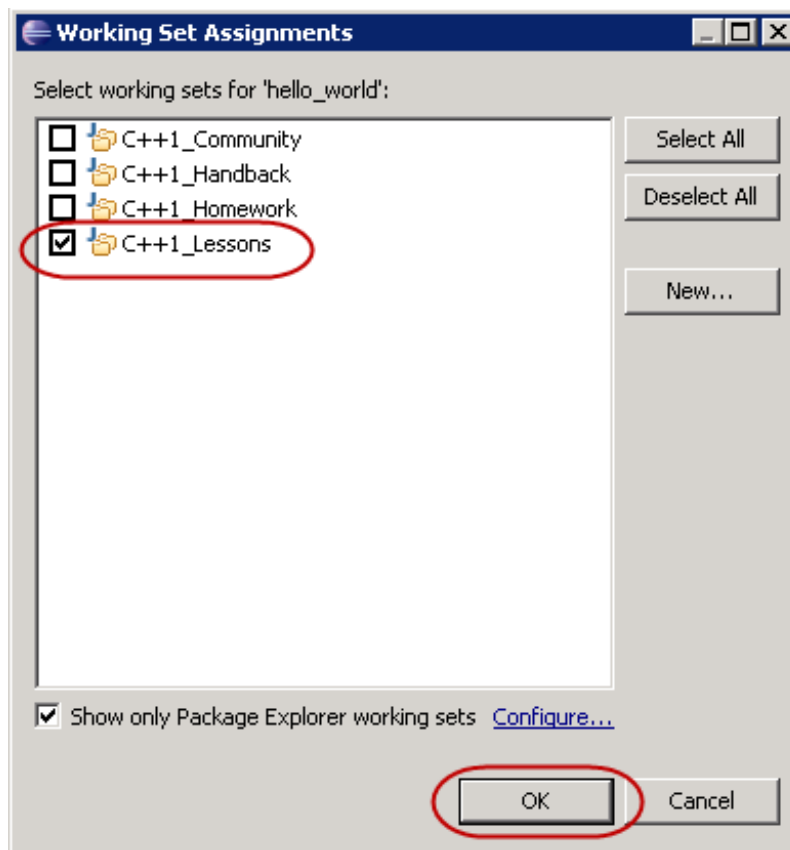
For "Project Type," select **Executable | Empty Project**. Under "Toolchains," select **MiniGW GCC** (the default). Then, click **Finish**:



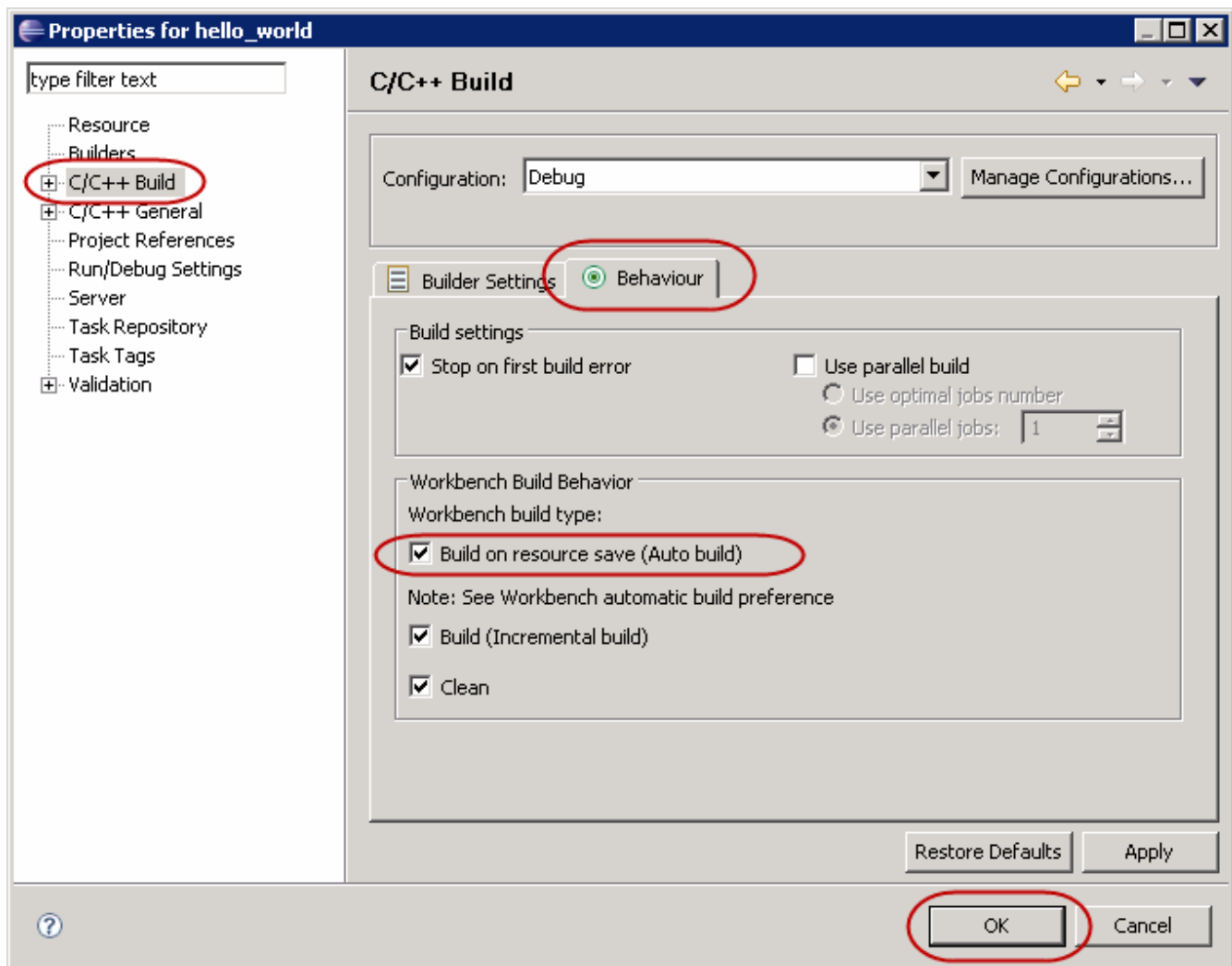
If you are prompted to open the C++ perspective, check the "Remember my decision" box and click **No**:



New projects are added by default to the Other Projects working set. To help keep your projects organized, we'll move them to the C++1_Lessons working set. Find the **hello_world** project in the Other Projects set, right-click it, and select **Assign Working Sets....** In the dialog box that appears, check the box for **C++1_Lessons** and click **OK**:

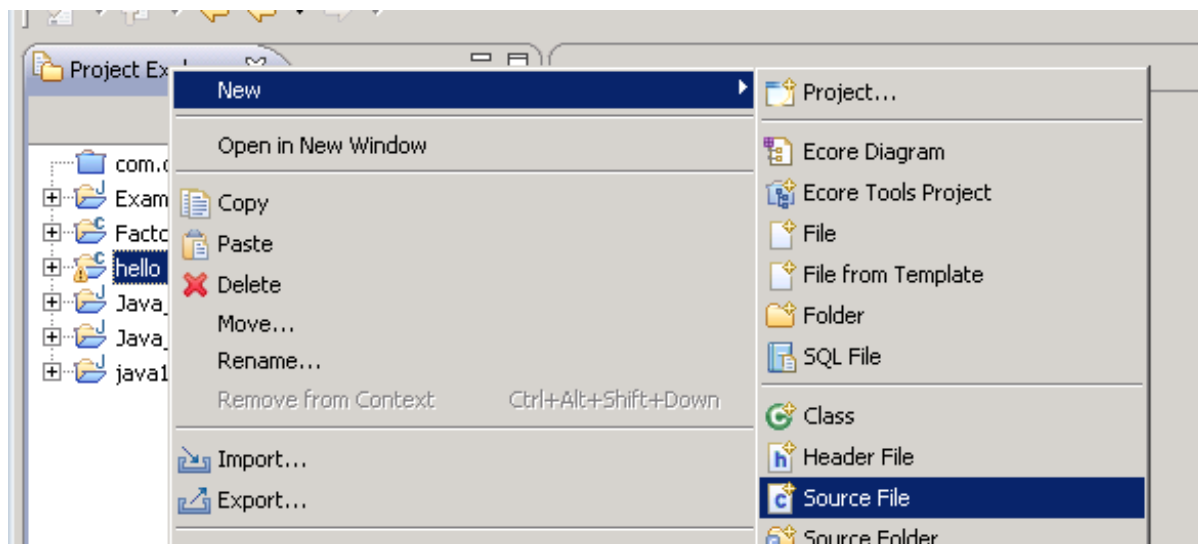


Now we'll set up the project's building behavior. Right-click the **hello_world** project again, and then select **Properties**. Select **C/C++ Build**, then click the **Behavior** tab. Check the **Build on resource save (Auto build)** box and click **OK**:

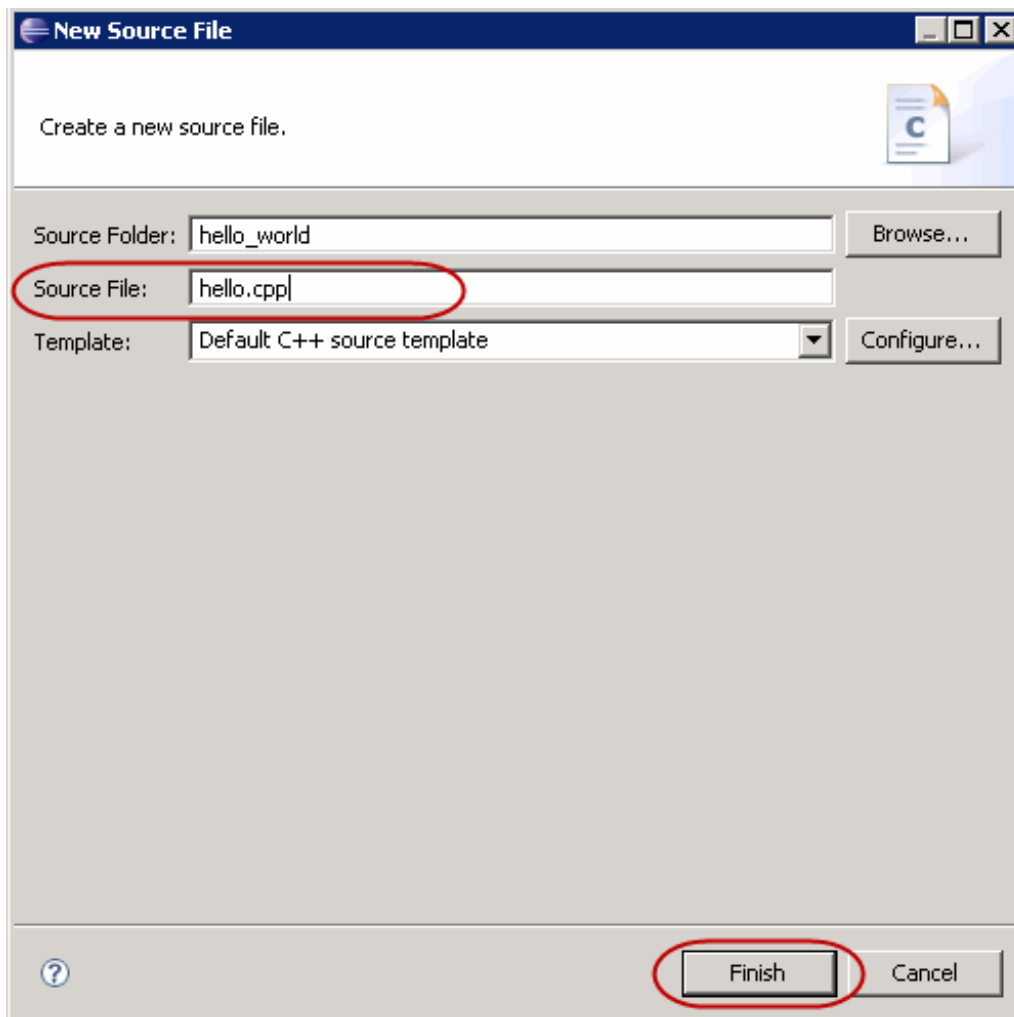


You will need to do this for every project you create. If you don't, when you try to run programs from the project, you'll see this message: "Launch failed. Binary not found." It isn't possible to set this flag as the default. But just in case you need to refresh your memory on occasion, we've provided a [checklist](#) of the steps you need to take in order to start a project.

Okay, we're ready to create our program. Find the `hello_world` project in the `C++1_Lessons` working set and select it, then select **File | New | Source File**:



Enter the name **hello.cpp** and click **Finish**:



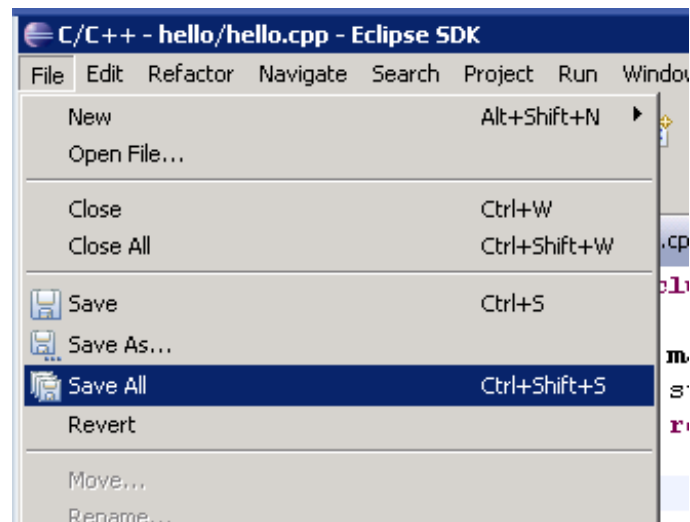
The new file **hello.cpp** appears, with a short header containing the filename, the date, and your name. In **hello.cpp**, add code as shown below:

CODE TO TYPE:

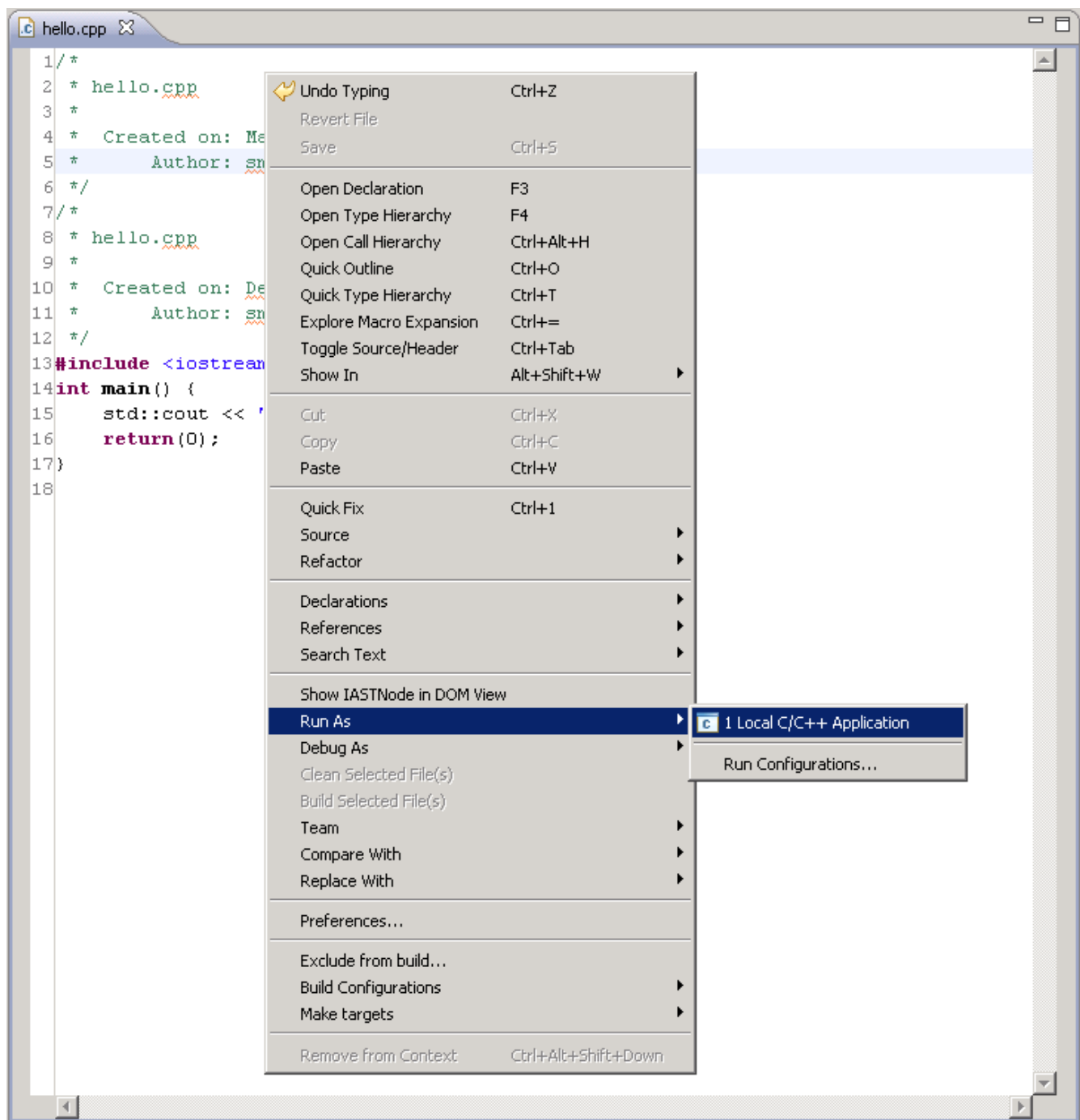
```
/*
 * hello.cpp
 *
 * Created on: Dec 15, 2009
 * Author: smiller
 */
#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    return(0);
}
```

As you type, you'll notice that the system automatically completes certain items. For instance, when you type `<`, it adds a `>`. And after you enter the line that contains **int main() {**, Eclipse automatically indents the next line by four spaces and adds the closing bracket **}** on the following line. Eclipse is your friend, and does its best to help whenever it can!


Select **File | Save All** to save your file:



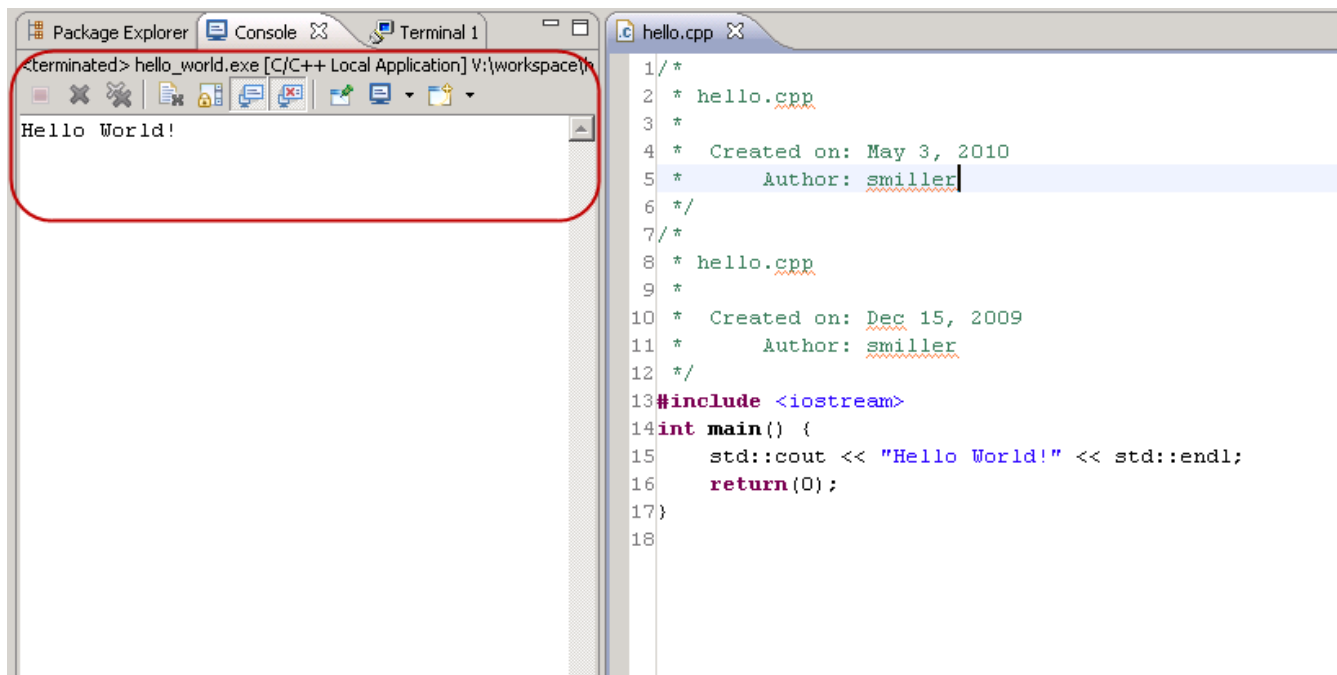
Congratulations, you've just written your first C++ program! Now let's run it. In the editor window for your **hello.cpp** program, right-click and select **Run as | Local C++ Application**:



Note

To run a program, you can also select it in the Package Explorer and click the Run icon () in the toolbar at the top of the screen. In the future, we'll use that icon when we want to run the program.

A Console window appears, containing the output from your program:



Excellent! You've run your first C++ program!

Now let's screw it up. We're doing that on purpose now, so we'll know what it looks like and how to fix it when we screw up our programs later by accident.

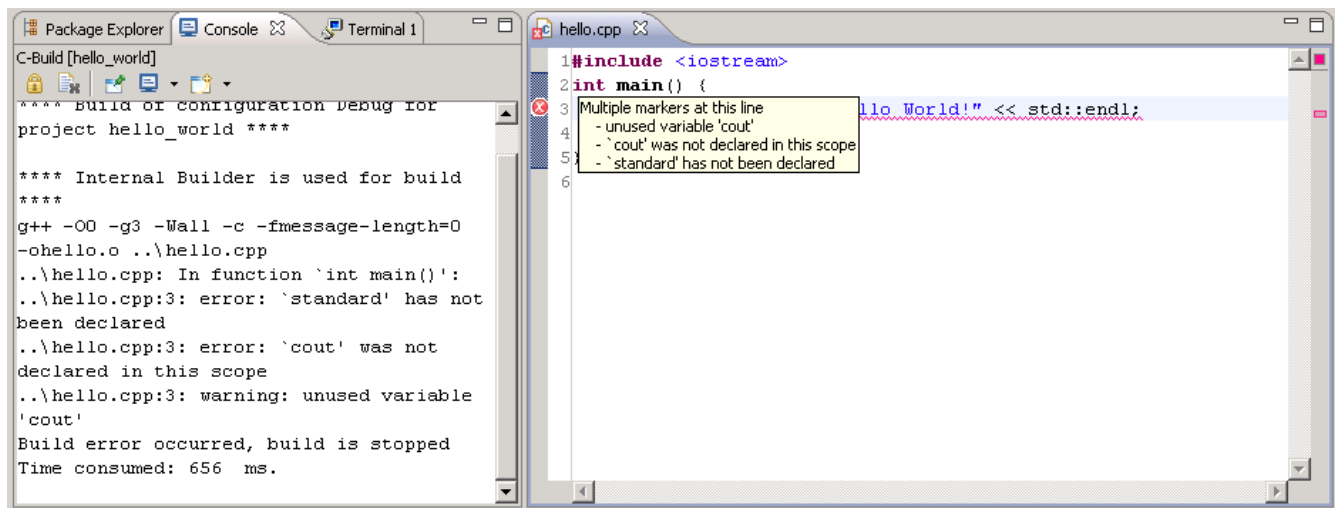
Modify **hello.cpp** as shown below:

CODE TO TYPE:

```
#include <iostream>
int main() {
    standard::cout << "Hello World!" << std::endl;
    return(0);
}
```

Note We omit the header comments from some of our examples in order to save space. In your real programs, you should always include descriptive comments.

Save your file (**File | Save all**). A red icon appears on the left side of the panel near the line you just changed. If you put the mouse pointer on the icon, you'll see this message:



This tells you that something bad happened. In particular, the compiler can't figure out what the symbol **standard** means.

Fix the problem by changing **standard** back to **std**, then save the file. The red icon should go away.

Now let's create a new problem for ourselves. Modify **hello.cpp** below as shown:

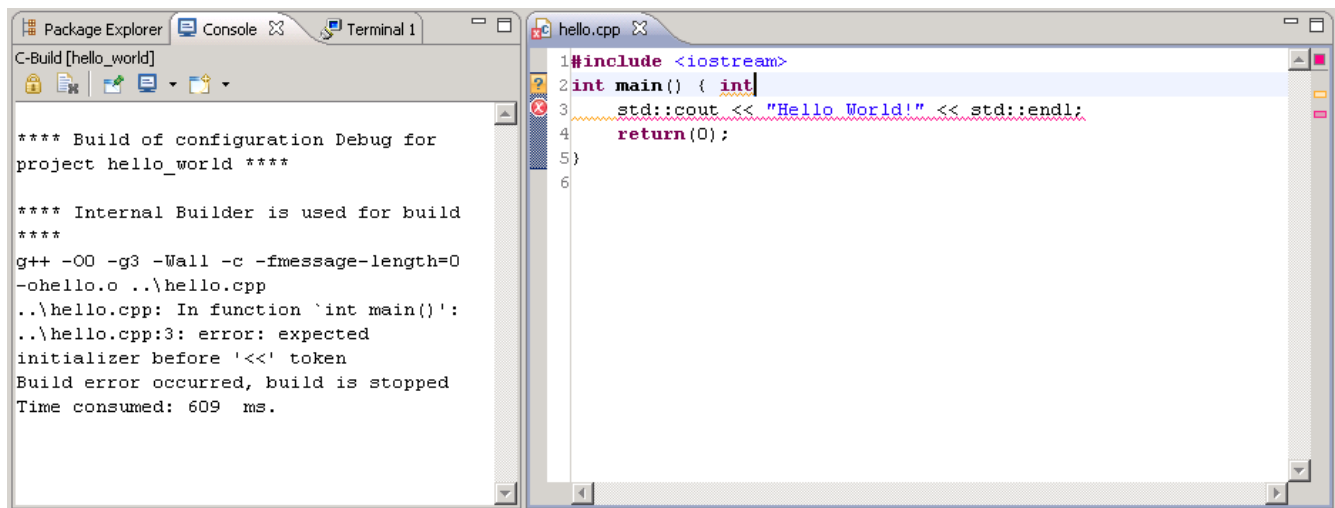
CODE TO TYPE:

```

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

```

Save the file. You'll see an **x** error icon on the the **std::cout** line:



So what happened? Eclipse flags errors with an **x** error icon. Ideally, it appears on the line where you made the mistake, but sometimes (like now), it doesn't. (However, in this case you do see a small yellow box containing a question mark on the **int main** line. Eclipse's internal parser detected an error on this line and flagged it that way.)

When Eclipse flags errors in your program, you still might need to do some investigative work to find and fix them—and it's a good idea to check the lines of code surrounding the flagged ones as well.

Fix the problem in **hello.cpp** by removing the misplaced **int**, then save your work.

What Does It All Mean?

Let's take a closer look at the program to see how it works:

OBSERVE: hello.cpp

```
#include <iostream>

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

The first line is `#include <iostream>`. This *#include line* tells C++ "I'm going to use your standard streaming I/O package (iostream)." The compiler brings in the definitions of the items for this package.

The next line, `int main()`, is the start of the code for your program. The `main()` function is special in C++; it is the first function that C++ executes. We'll get into function definitions later—for now, you just need to know that this special line starts the program.

The curly brace `{` indicates the start of the body of the code. Code enclosed in braces is called a *block*. In this program, we have one block of statements and they make up the body of the `main()` function. In later lessons we'll learn how to use multiple blocks of code.

The next line contains `std::cout << "Hello World!" << std::endl;`, which is a C++ executable statement. It tells C++ to print a message. You'll notice that the line starts with four spaces. C++ doesn't care how many spaces you use to start a line, but good programming style dictates that you use one level of indentation for each level of logic. Here we indented four spaces for each set of braces we have nested. We chose to indent four spaces as a standard because it's easy to read and we needed a standard for this course.

Let's break the `std::cout` statement down into components.

The first item, `std::cout`, is the name of a predefined variable that is used by C++ to write to the standard output (the console). (In the Eclipse programming environment, console output shows up in a window at the bottom of the screen.) The `std::cout` variable is one of the items brought in by the `#include <iostream>` statement.

The operator `<<` tells C++ to take what follows it and send it somewhere else, in this case to the output (`std::cout`). Next in our code, we have the string `"Hello World"`. This is a literal string containing the characters we want to display on the console.

This is followed by `<<` again (which sends whatever follows it to the output stream on the left) and the symbol `std::endl`. The `std::endl` symbol tells C++ to output an end-of-line character. The statement ends with a semicolon.

Next, we have the return line `return(0);`, which tells C++ to end the program and return a status of 0 to the operating system. (You'll learn more about the return statement later, but for now all you need to know is that inside `main`, it ends the program.) A return code of 0 indicates a normal exit. Codes 1-255 typically indicate that the program exited abnormally—the bigger the number, the bigger the problem.

Again, the statement ends with a semicolon.

At the very end we have a closing brace `}`. This ends the block of code that started at the brace just after the `main()` line.

Adding to Our Program

Now let's say we want to output another line of text. To do that, we can add another line to output an additional message. Edit your program as shown (using your own name instead of "Steve"):

CODE TO TYPE:


```
/*
#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    std::cout << "My name is Steve!" << std::endl;
    return(0);
}
```


 Save your file, then run it by right-clicking in **hello.cpp** in the editor window and choosing **Run as | Local C++ Application** (alternatively you could click the **Run** icon in the toolbar). You'll see this output:

OBSERVE:
Hello World! My name is Steve!

Our output appears on two lines because we used **std::endl**. Another way to make the output appear on two lines is to use the special escaped character **\n**. Modify **hello.cpp** as shown:

CODE TO TYPE:
<pre>#include <iostream> int main() { std::cout << "Hello World!" << "\n"; std::cout << "My name is Steve!" << "\n"; return(0); }</pre>

 Save your program, and run it. The output looks identical to the last run:

OBSERVE:
Hello World! My name is Steve!

\n is called an *escape character*. The backslash (\) "escapes" from the regular interpretation of keyboard characters to begin a special multi-character sequence, indicating a special character. In this case, it's a "newline," or end of line, the equivalent of pressing the Enter key.

For more information, see the [glossary's](#) description of the escape character.

But the escape character is the old-fashioned way to print a newline. The more modern practice is to use the special symbol **std::endl**. There is a subtle difference between **\n** and **std::endl**; the **std::endl** symbol causes a buffer flush, which makes sure the output appears immediately. The **\n** is not required to flush the buffer (although on most systems it does).

We cover buffering and flushing in more detail in later C++ courses.

What Goes On Under the Hood

Eclipse is a type of tool called an [IDE](#) or *Integrated Development Environment*. This sort of tool is a wrapper around many other tools. The idea of the IDE is to hide the other tools and give you a single system in which to work. But as a professional, you'll need to know what goes on "under the hood" to make the most effective use of all the tools available.

When your project is built, Eclipse figures out what needs to be done to make your programs executable, automatically. This comes in handy for many types of programs, including our small "hello world" project. Tools like Visual Studio work much the same way.

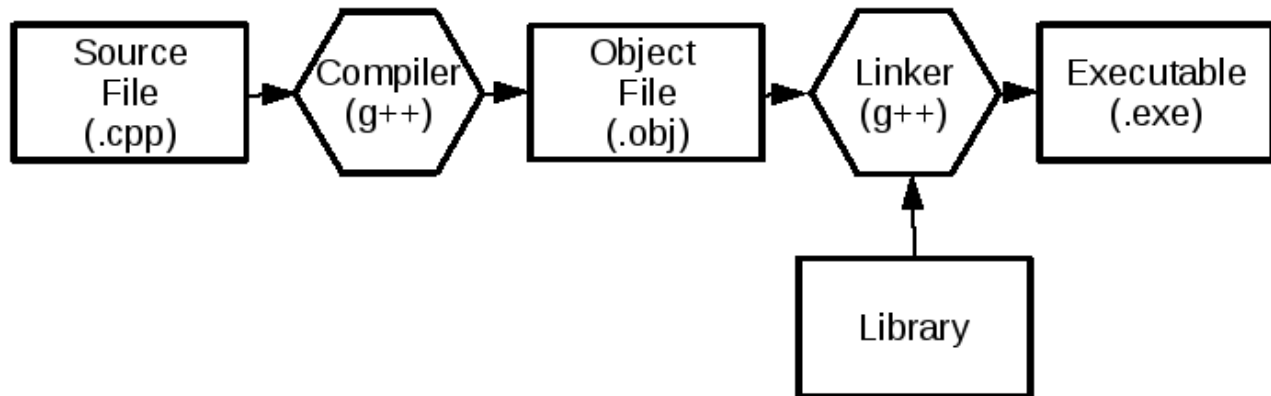
Some programs use other builder tools to make them automatically executable—in the C/C++ world, a program called [make](#) is a common tool for that. In java, *Ant* is a popular tool for that task. Both of those tools require programmers to specify a set of "rules" or "targets" that describe how a program will be built.

To convert your code into computer instructions, Eclipse uses the GNU g++ compiler. This program takes your [source code](#) (.cpp file) and turns it into an *executable*. To do this, it needs to run a series of programs. The first is the actual C++ [compiler](#). Its job is to take the human-readable source code and turn it into an [object file](#) (.o file).

You can see the object file created by the compiler. Switch to the Package Explorer tab, then expand the **Debug** folder. Inside that folder you'll see **hello.o** (among other things). If you try to open **hello.o**, you won't find anything useful—this file contains computer code. The object file is readable by the computer, but not by humans. The object file contains *only* the code for executing the task that you wrote. In the case of hello_world, that code uses **std::cout** to write out a message. The definition of **std::cout** is not in your object file; instead it is part of a standard [library](#). This

library contains generally useful definitions like `std::cout` and lots of other things. The library itself is just a bunch of object files packaged together into one file (something like a zip file, but different).

The [linker](#) takes your object file and the object files in the library and produces an executable program. Here's a graphical representation of what it does:



With some compilers, you may not see object files, because they delete object files after the link step.

With the GNU compiler, the compilation process is actually accomplished by multiple programs. The first thing `g++` does to your program is pass it through a program called the pre-processor. This program takes care of things like the **#include** directive. (We cover lots more about this program in the next C++ course.)

The next stage turns the high-level C++ code into low-level [assembly code](#). In high-level code, one statement can result in many machine instructions being created to process that statement. In assembly language, there's one statement per instruction. Also, high-level languages are machine-independent (or they are supposed to be.) Assembly code is machine-dependent. In other words, the assembly language for an Intel x86-compatible processor (such as the Intel Core 2 Duo or Intel Pentium) is entirely different from the assembly code for the iPhone's ARM processor.

Fortunately, all this complexity is hidden from you and you can ignore it most of the time. But sometimes you may need to take a peek under the hood to deal with any bugs you encounter along the way.

Finishing the Program

Our program, although syntactically and logically correct, is not complete because *it only contains the default comments produced by Eclipse*. A comment is text in the program that tells people reading the program what's going on. Comments are not read by the computer and although it is possible to write a program with no comments in it, we will not do so in this course. Professional programming means creating professional-quality programs with comments—even if that program is "Hello World."

All of your programs should begin with a comment block that contains these sections:

- The lesson and project number (so the instructor knows which question you're answering).
- A description section that describes what the program is supposed to do.
- A usage section.

Comments begin with `/*` and end with `*/`. Comments can also begin with `//` and go to the end of the line.

Our program needs a set of comments. Modify **hello.cpp** as shown:

CODE TO TYPE:

```
/*
 * Lesson 1, Example 1 (Or Assignment 1 for programs you turn in)
 *
 * Description: The classic "Hello World" program. Prints
 * out the message and that's all.
 *
 * Usage: Run it and get the message.
 */

// This is another way to comment a single line.

#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    return(0);
}
```



Save and run it to verify that it still works as expected.

The User Manual

Are we finished? Not yet—a program isn't any good if no one knows how to use it, so we'll write a user manual for every program.

You may wonder how to go about writing a manual for such a short program. For a short program, we just write a short manual.

To create your manual, select the `hello_world` project and then select **File | New | Other**. In the New File Wizard, select **General | File**. Name your file *manual.txt*. Now write the manual:

CODE TO TYPE:

```
Run the program.
See the message.
```

The Test Plan

Finally, the program needs a test plan. Our test plan should have these attributes:

- It should test as much of the program as is feasible.
- It must list a precise set of steps to follow (ambiguous instructions lead to results that cannot be reproduced).
- Its results must be observable.
- It must lead to a clearly observable pass/fail result.

For example, a test plan that says, "Play around with the software and see if you can break it," is a vague and therefore pretty bad test plan.

First, the term "Play around" is not precise. Different people could "play around" in different ways.

Second, what happens if someone breaks the software? Can he do it again? Often, the answer to this question is no.

Third, the term "break" is not defined. If the system crashes, that certainly is a break. But what if it merely draws something that looks odd? Is that "odd" drawing a feature or a bug?

Now we'll write our clear, brief, and excellent test plan. Create a file called *test.txt*, then type in the code as shown:

CODE TO TYPE:

```
1. Run the program.
2. Observe the message "Hello World!" (Yes -- Pass, No -- Fail)
```

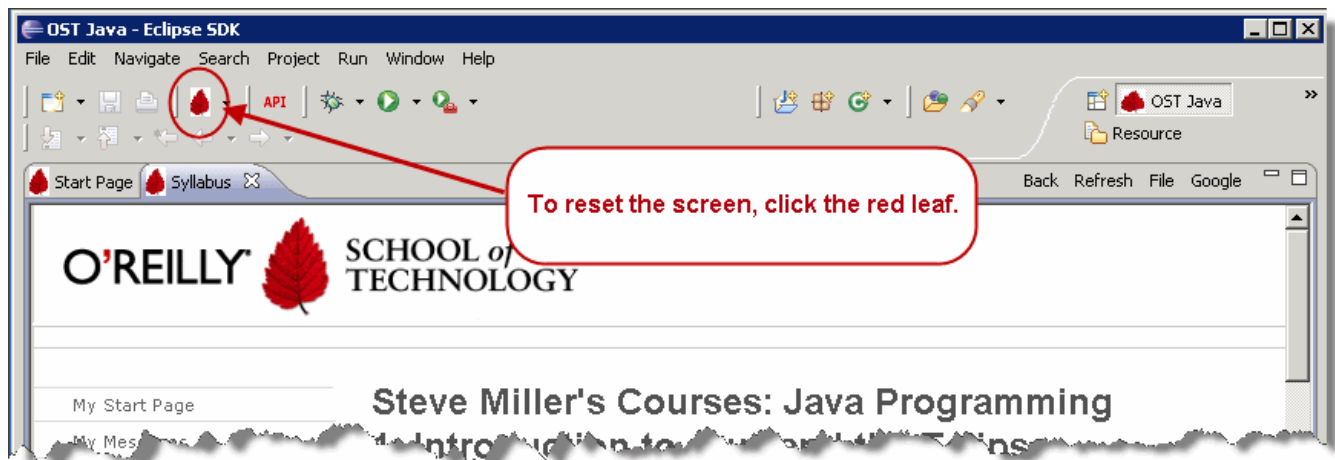
A Few More Notes

We recommend that you use the **Save All** menu option to save your work; this will save everything you have opened, and it can prevent problems when running your programs while you're working on them.

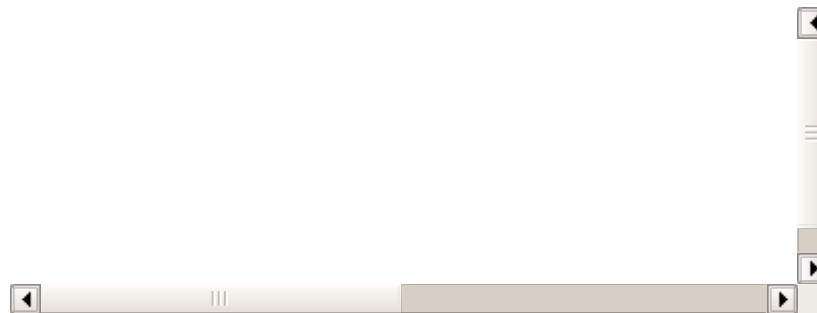
Do *not* put spaces or other special characters in file names. Limit yourself to a combination of letters, digits, underscores (_), and dashes (-) This helps to avoid problems with the tool set being used.

When you create a new source file, it always contains the standard header, including the file name, date, and your name. You should replace those elements with more detailed and specific information about your program.

Remember, when using the O'Reilly sandbox system, that Red Leaf icon is your emergency button. It will restore your screen to the original layout:



Congratulations! You have created your first C++ program and completed lesson 1 of the course! In the next lesson, we'll investigate expressions. See you there!



Copyright © 1998-2013 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Expressions

C++ 1: Introduction to C++ Lesson 2

Glad to see you're back! In this lesson we'll learn to use mathematical expressions in C++.

Mathematical Expressions

C++ understands these operations:

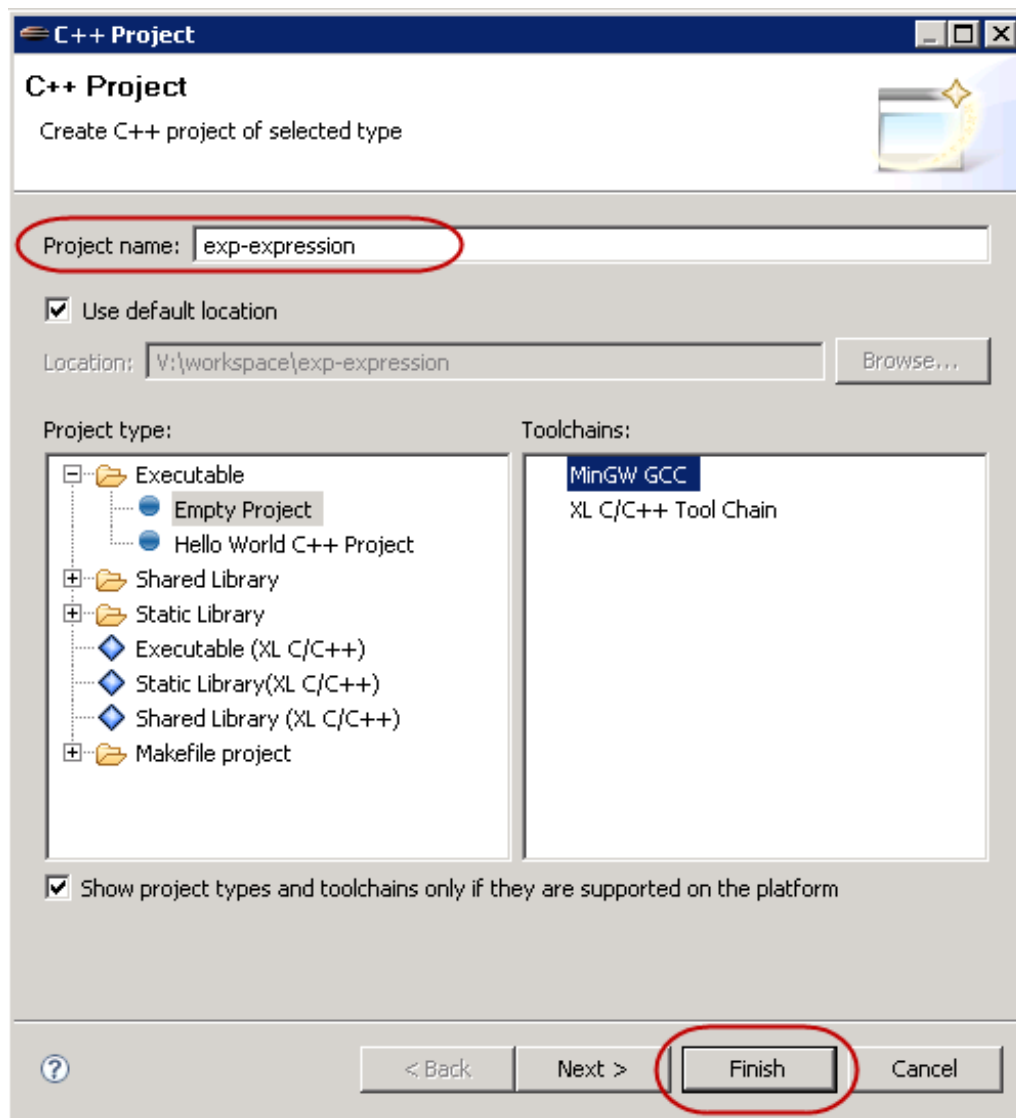
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder after division)

We can combine these operations with numbers into an expression to do useful lots of really useful work.

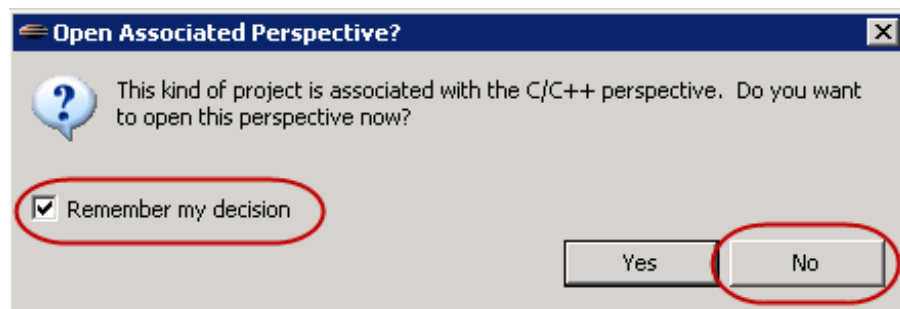
Let's get going and try an example to see how these operators work. Start a new project and name it **exp-expression**. Assign it to the **C++1_Lessons** working set. Then create a new source file in the new **exp-expression** project and name it **exp-expression.cpp**. Now create the project and source file. You can see a graphical representation of this procedure in the next section. But if you feel comfortable with the process so far, go ahead and [skip to the following section](#).

Creating the Project and File

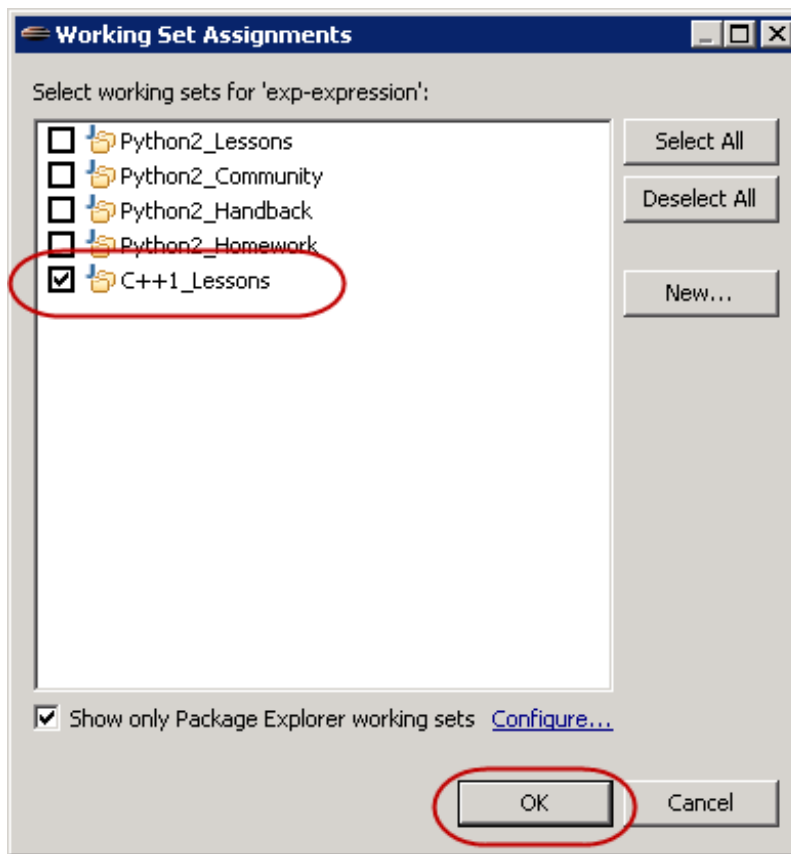
Select **File | New | C++ Project**, enter the information, and click **Finish**, as shown:



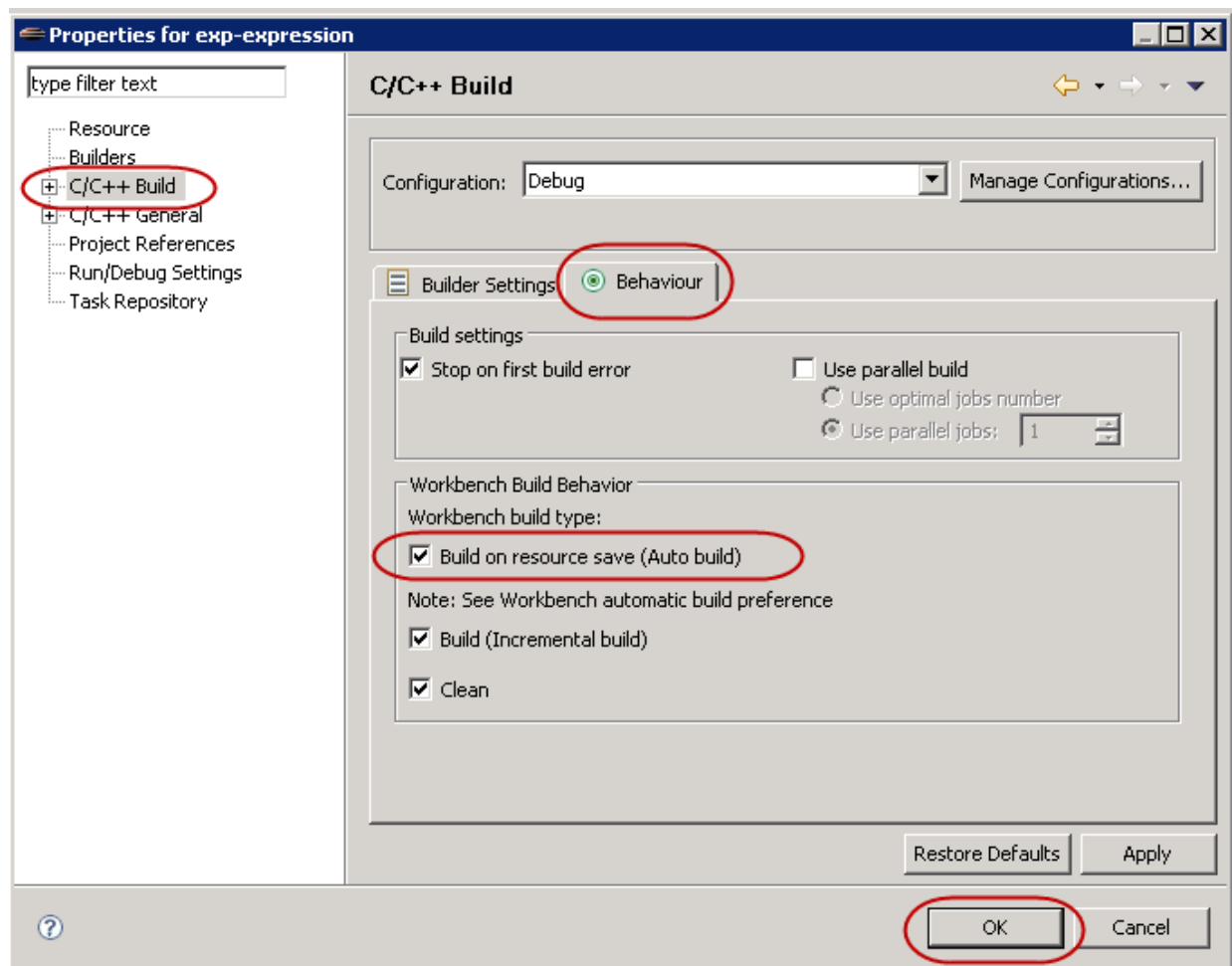
If you are prompted to change the perspective, check the **Remember my decision** box and click **No**:



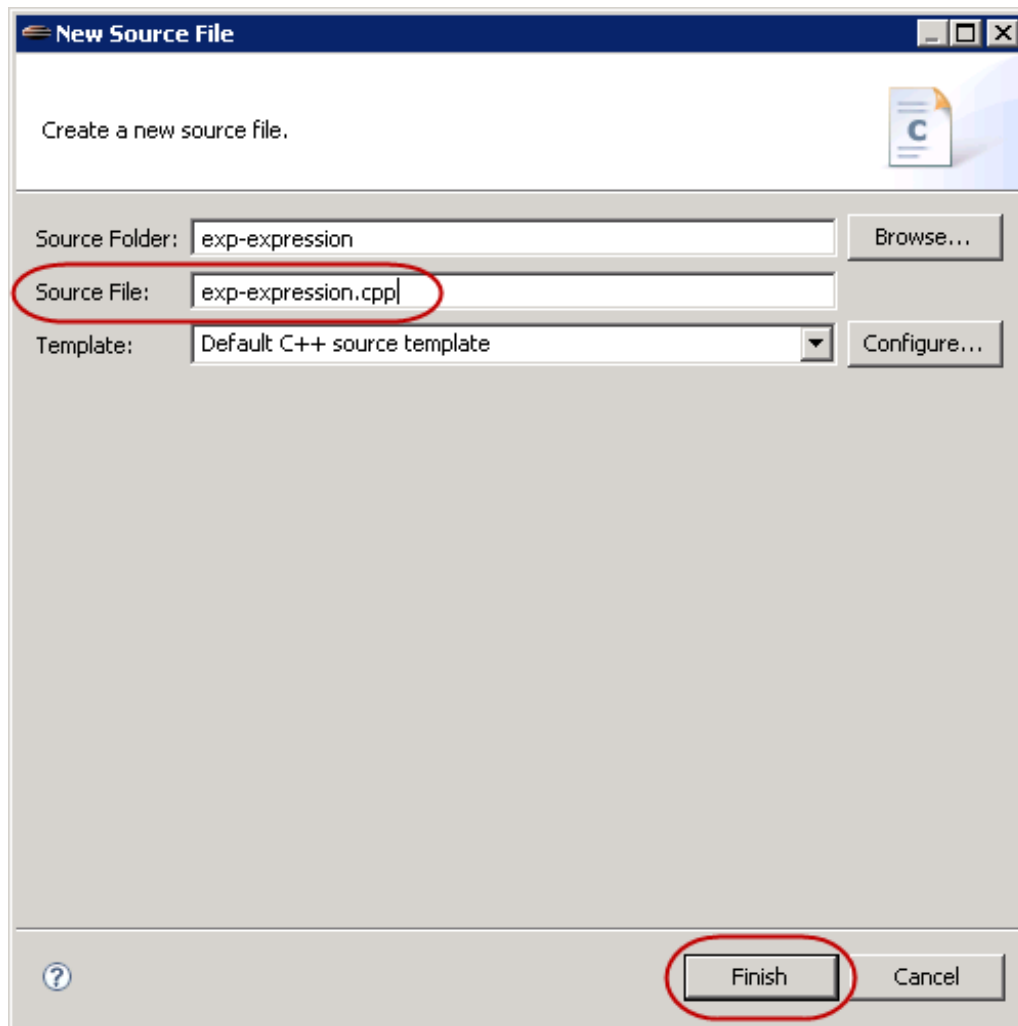
Assign the **C++1_Lessons** working set to the **exp-expression** project; right-click the project, select **Assign Working Sets...**, and assign the working set as shown:



In the Package Explorer, right-click the **exp-expression** project, select **Properties**, and set the Build properties as shown:



Now, in the **exp-expression** project, select **File | New | Source File**, and enter information as shown:



Editing and Running Your Program

Open your **exp-expression.cpp** file and enter the following code (we'll omit the automatic comments from now on to save space):

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << (1 + 2) * 4 << std::endl;
    return (0);
}
```

 Save your source file, then in the editor window for **exp-expression.cpp**, right-click and select **Run as | Local C++ Application** (or select **exp-expression.cpp** in the Package Explorer and click the  icon in the toolbar). You'll see this output:

OBSERVE:

```
The answer is 12
```

The computer evaluated, or *calculated*, the answer to the expression **(1 + 2) * 4** (which is 12) and showed you the results.

So, how did the parentheses affect the answer? I'm glad you asked! Let's remove them and see what happens—edit your code as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << +1 + 2+ * 4 << std::endl;
    return (0);
}
```



Save and run your program. This time the answer is different:

OBSERVE:

The answer is 9

The computer reads the expression $1 + 2 * 4$ from left to right, but it also follows the [Order of Operations](#):

1. exponents and roots
2. multiplication and division
3. addition and subtraction

Multiplication occurs before addition, so the computer actually does this:

- $2 * 4 = 8$
- $1 + 8 = 9$

You can change the Order of Operations by using parentheses (). In the first example, we used parentheses to force addition to occur before multiplication.

Types of Numbers

There are two major types of numbers in C++: *integers* and *floating point*. (You might also hear about a *complex* type, but that type isn't actually built into the language.)

Integers, also known as *whole numbers* have no fractional value. For example, these are all integers:

OBSERVE:

1 324290 42 -999 37

Floating-point numbers contain a fractional part. For example:

OBSERVE:

1.2 3.5 14.8 37.0

The last number presents a key concept: **37.0** is a floating-point number. Even though the fractional part is 0, its presence after the decimal point makes **37.0** a floating-point number. The number **37** is an integer.

Note

When writing floating-point numbers, always include a decimal point. It tells anyone reading your code that you intend the value to be floating point. So **1.0** is good, and **1** is bad. **0.0** is good; **0** is bad.

Floating-point numbers can also contain an optional exponent. For example:

OBSERVE:

13.33E+5

This tells C++ that the value of the number is 13.33×10^5 .

Floating-Point vs. Integer Division

Edit **exp-expression.cpp** to compute (and print) the value of the expression **1/3** as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << 1 / 3 << std::endl;
    return (0);
}
```



Save and run it, then read the output:

OBSERVE:

The answer is 0

When C++ does integer division, the result is an integer—it truncates any fractional part of the result.

Now, edit **exp-expression.cpp** again, as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << 1.0 / 3.0 << std::endl;
    return (0);
}
```



Save and run it and look over the output:

OBSERVE:

The answer is 0.333333

When C++ sees that the arguments are floating-point numbers, it performs floating-point division and gives floating-point results.

But what if one number is floating point and the other is an integer? Let's give that a try. Modify **exp-expression.cpp** as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << 1.0 / 3 << std::endl;
    return (0);
}
```




Save and run it and read over your results. Try **1/3.0** as well. Now that you've seen how mixed-mode arithmetic will affect your programs, avoid using it if at all possible. For now, that may be difficult, but later we'll learn about casting, which will let us explicitly tell the compiler which types of numbers (and therefore operations) to use.

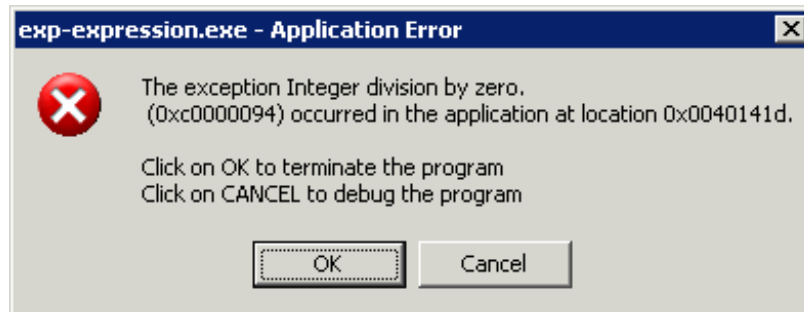
Dividing by Zero

We cannot divide anything by zero—that operation isn't defined and the expression [has no meaning](#). So what happens if we tell the computer to divide by zero? Modify your program to compute a new expression that divides by zero as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "Divide " << (1/0) << std::endl;
    return (0);
}
```

Save your program, and ignore the warning message.  Run it and observe that it terminates with this message:




The computer can't handle this type of math, so the program stops with a warning. Click **OK** to end the program.

But what happens when you divide by zero with a floating-point number? Try it and find out; edit your program below as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "Divide " << (1.0/0) << std::endl;
    return (0);
}
```

 Save and run it. You'll see output like:

OBSERVE:

Divide inf

The floating-point format used by basic Intel-compatible processors has special "numbers" defined for error conditions such as **inf**, **-inf**, and **NaN** (not a number). This is processor-dependent and although almost all modern processors now use this standard format, some don't. Results on those computers could be different.

So now you may be wondering why integer division by zero crashes your program, while floating point division by zero does not. This is by design—Intel-compatible processors have been used for decades, and at first could only perform integer arithmetic. Floating-point arithmetic was handled separately, and thus it continues to generate different errors.

Instead of worrying about the differences in error handling, it is better not to generate the error in the first place. In future lessons we'll learn how to make sure our programs work correctly. Stay tuned!

Limits on Numbers

Suppose you have an old eight-digit calculator. You type in the number 99,999,999 and then add 1 to it. The result is a nine-digit number, which the calculator can't display. So the calculator displays ERROR.

There are similar limits to the numbers in C++. Unfortunately, these are not hard limits. They can vary depending on the processor type, compiler, and operating system.

C++ has a file (named *climits*) that defines the limits on its basic types. We'll **#include** this file, and then add some code to use it. Edit your **exp-expression.cpp** program as shown:

CODE TO TYPE:

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << "INT_MAX " << INT_MAX << std::endl;

    return (0);
}
```



Save and run it. You'll see this:

OBSERVE:

INT_MAX 2147483647


Note

This example uses the C-style *constant* for the integer limit. The pure C++ way of doing this is to include the header file `<limits>`. Then you can get the maximum integer with the expression `std::numeric_limits<int>::max()`. We used the C method because it's shorter. Also, to understand the expression `std::numeric_limits<int>::max()`, you need to understand classes, static member functions, templates and template specialization—all concepts we will cover in a future course.

Let's see what happens when we go past the limit. Change the program as shown:

CODE TO TYPE:

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << "INT_MAX+1 " << INT_MAX+1 << std::endl;
    return (0);
}
```

Based on the last execution of the program, we might expect to see the number 2147483648.  Run it. Instead we see:

OBSERVE:

INT_MAX+1 -2147483648

This is called [overflow](#). It occurs when a number becomes too big or too small to fit into its type (in this case, integer). C++ does not check for overflow and will not warn you when it occurs. You won't encounter this problem too often, but if you do, now you'll be able to recognize and correct it.

Okay, now let's see what happens when we have a floating-point overflow. Because of the way floating-point numbers are stored and computed, it's hard to specify an exact maximum number. But the expression below will definitely give you an overflow, so go ahead and add it to your program as shown:

CODE TO TYPE:

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << "Float " << (9E399 * 9E399) << std::endl;
    return (0);
}
```



Run it. You'll see the special floating-point number **inf**; the floating system gives you some indication that an overflow occurred.

Common Problems

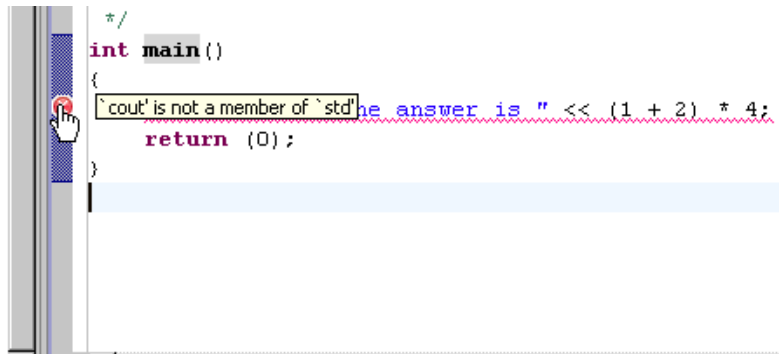
There are some common warnings and errors that you may encounter when working with C++. We'll experiment with some of them by making a few mistakes on purpose. Change your program as shown:

CODE TO TYPE:

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << "The answer is" << (1 + 2) * 4;
    return (0);
}
```

C++ flags this line with an error message because it doesn't recognize **std::cout**.

```
**** Internal Builder is used for
build ****
g++ -O0 -g3 -Wall -c
-fmessage-length=0 -oexp-expression.o
..\exp-expression.cpp
..\exp-expression.cpp: In function
`int main()':
..\exp-expression.cpp:9: error:
`cout' is not a member of `std'
Build error occurred, build is
stopped
Time consumed: 156 ms.
```



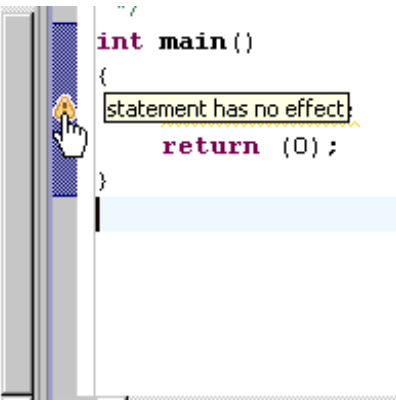
The **std::cout** functionality is defined in **iostream**, so in order to use **std::cout**, we have to include **iostream** in our program. Edit the program as shown:

CODE TO TYPE:

```
#include <iostream>
#include <climits>
int main()
{
    (1 + 2) * 4;
    return (0);
}
```

When you compile the program, you'll notice a warning triangle next to the line that contains the expression. Look in the **Console** panel to see the text associated with the warning:

```
-fmessage-length=0 -oexp-expression.o
..\exp-expression.cpp
..\exp-expression.cpp: In function
`int main()':
..\exp-expression.cpp:9: warning:
statement has no effect
g++ -oexp-expression.exe
exp-expression.o
Build complete for project
exp-expression
Time consumed: 516 ms.
```



The warning reminds us that our expression doesn't really do anything—we do not output its results or save them anywhere.

Save and run it. The output window contains absolutely nothing. In particular, it does not contain the result of the calculation. The line `(1 + 2) * 4` tells C++ to compute the value of the expression, but nothing more. It doesn't store the result anywhere, doesn't make a decision based on the result, and doesn't output it. C++ just computes the answer and then throws it away.

This is perfectly legal, but pretty strange—so strange that the compiler issues a warning when you build it. That's why you got the "no effect" warning. Let's fix the program by editing it as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is" << (1 + 2) * 4;
    return (0);
}
```

Save and run it. You'll see this:

OBSERVE:

The answer is12

Do you see a problem in this code? That's right—we need a space between "is" and the answer. Add the space as shown (we use an underscore here to represent the space):


CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is_" << (1 + 2) * 4;
    return (0);
}
```

Save and run it. Hmm. There's still a problem. See if you can spot it. We'll add another output statement (as shown in blue below) to help illuminate the problem:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << (1 + 2) * 4;
    std::cout << "The answer still is " << (1 + 2) * 4;
    return (0);
}
```

 Save and run it. The output looks like this:


OBSERVE:

The answer is 12The answer still is 12

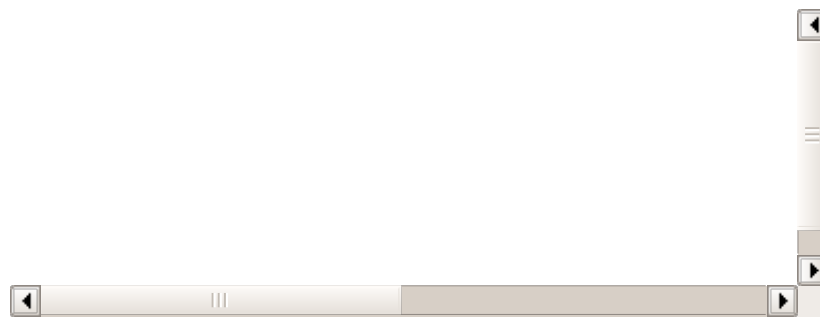
Any additional output statements will be appended to the end of this one. You need to add a newline at the end of the output line. Edit the program as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    std::cout << "The answer is " << (1 + 2) * 4 << std::endl;
    std::cout << "The answer still is " << (1 + 2) * 4 << std::endl;
    return (0);
}
```

 Save and run it to make sure that the problem has been corrected.

We covered a lot in this lesson! We went over expressions, integers, and floating point numbers. Now you're ready to tackle the next lesson, where we'll discuss program structure and variables. See you there!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Variables

C++ 1: Introduction to C++ Lesson 3

Basic Programs

We have to walk before we can run, so we're going to learn to program for now in a subset of the C++ language. Our basic program structure is:

OBSERVE:

```
/* File heading comments */
#include directives
int main() {
    data declarations

    executable statements
    return (0);
}
```

Note

In some books and code examples, the **data declarations** are placed before the **int main** line. In these lessons, they come after it. At this stage, for the types of programs we can currently write, either order will work. (We'll learn the difference between the two types of declarations later.)

It is easier to use Eclipse if you put the **data declarations** just after the first brace (**{**).

The **include directives**, **data declarations**, and **executable statements** sections are optional, so the shortest possible C++ program would have all three sections omitted:

OBSERVE:

```
int main()
{
    return(0);
}
```

Such a program may appear to be useless, but it's actually a standard Linux/Unix command, [true](#). It's used in shell scripting as a command that always returns a good status (0) to the currently running script.

Variables

A variable is a bit of your computer's memory, like a box, in which you can put a single item of data. We must declare variables before we use them. In order to declare a variable, C++ needs two pieces of information: the name and the type (what the box is called, and what type of box it is).

In order to pass this course, you'll need to add a third piece of information: A comment explaining what the variable is used for.

A variable name begins with a letter (upper or lower case) or an underscore (**_**), then continues with any combination of letters, underscores, or digits. It cannot be the same as any C++ keyword.

For example, these are legal variable names:

OBSERVE:

box_width	point_count	today
whatever	point3	dataField

These are not legal variable names:

OBSERVE:

```
3times // Begins with a digit
box-top // Contains hyphen
```

Variable names are usually lower case (**size**, **full_size** or mostly lower case (**Size**, **FullSize**, **fullSize**). Constants are usually given upper-case-only names (**PI**, **MAX_INT**). Nothing in the language forces you to follow this convention, but if you choose to ignore it, don't be surprised if a mob of angry maintenance programmers shows up outside your door with torches and pitchforks.

Style Note: There are two major ways of constructing variable names. The first is separating words with underscores:

OBSERVE:

```
box_width    point_count    data_size
```

The other is something called "[CamelCase](#)":

OBSERVE:

```
boxWidth    pointCount    dataSize
```

In practice, there's not a lot of difference in the readability of the two styles, *but there is a great benefit to picking one style and sticking to it.*

The [style guide](#) for this course requires you to use the underscore style.

Variable Definitions

A [variable](#) definition consists of three parts:

OBSERVE:

```
{type} {name}; // {Comment explaining the variable}
```

The **comment** is *not* optional—at least not in this course. The reason is that a program is a set of instructions to the computer that uses a unique (to that program!) vocabulary. I mean if you see a variable named **center_point** in code, the first question you are going to ask yourself is "Center of what?" By commenting every variable declaration, you produce a mini-dictionary describing every specialized word (variable) you use in your program. It makes understanding the program much easier.

Variable Types: Integer

The C++ keyword for the integer type is **int** (Glossary [integer](#), [int](#)). A integer variable declaration looks like this:

OBSERVE:

```
int x; // example variable
```

The **int** declaration tells C++ to declare an integer variable using the optimal size of an integer for the machine. On *most* machines, this allows numbers from 2147483647 to -2147483648 to be used. On some older systems, this is just 32767 to -32768 and on some newer systems, it's -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

This is the difference between *32-bit*, *16-bit*, and *64-bit* integers. Unless you are interested in the technical details regarding the physical storage of integers within your computer's processor, you don't need to remember the difference between bit sizes and number ranges. Instead, remember there are different ways to store this data and remember to use the constants we saw earlier, such as **INT_MAX**.

Think of a variable as a box. You can put a single integer in it and you can look in it and see what's in it. Let's look at this in action. Create a project named **assign-exp** in your **C++1_Lessons** working set, and a program named **assign-exp.cpp** that contains the following code:

CODE TO TYPE:

```
// Put in the regular heading comments
#include <iostream>

int main() {
    int play; // An integer to play around with

    play = 5;
    std::cout << "The value of play is " << play << std::endl;
    return(0);
}
```

 Save and run it. This program assigns the variable **play** a single value and then uses it in an output. You'll see the results:

OBSERVE:

The value of play is 5

The program stores "5" in the variable named **play** and then displays that value to you.

Variables can change values as programs execute, and **play** is no exception. Let's see how the value of **play** changes as the program executes. Change your program as shown:


CODE TO TYPE:

```
#include <iostream>
int main() {
    int play; // An integer to play around with
    std::cout << "At first, the value of play is " << play << std::endl;

    play = 5;
    std::cout << "The value of play is " << play << std::endl;

    play = -999;
    std::cout << "Finally, the value of play is " << play << std::endl;

    return(0);
}
```

 Save and run it. You should see the following output:

OBSERVE:

At first, the value of play is 0
The value of play is 5
Finally, the value of play is -999

If we stepped through the program, we'd see the value of **play** after each line is executed. We might see the following:

Line	value of play
int play;	<i>undefined</i>
std::cout << "At first, the value of play is " << play << std::endl;	<i>undefined</i>
play = 5;	5
std::cout << "The value of play is " << play << std::endl;	5
play = -999;	-999
std::cout << "The value of play is " << play << std::endl;	-999
return(0);	-999

BUT WAIT! This table shows that the initial value of **play** is *undefined*, but the output shows that it was **0**!

Before we assign **play** a value for the first time—before the **play = 5** line—its value is not known, and cannot be assumed to be anything. With this specific computer, compiler, and execution, the value happened to be **0**. A different computer, compiler, or execution could produce any other result, such as 100, 900, or -2147483648!

The bottom line: *don't assume your variable has a value before you give it a value!*

In C++, you can declare and initialize variables in one statement. Change `assign-exp.cpp` as shown:


CODE TO TYPE:

```
#include <iostream>
int main() {
    int play = 2; // An integer to play around with
    std::cout << "At first, the value of play is " << play << std::endl;

    play = 5;
    std::cout << "The value of play is " << play << std::endl;

    play = -999;
    std::cout << "Finally, the value of play is " << play << std::endl;

    return(0);
}
```

 Save and run it. You should see the following output:

OBSERVE:

```
At first, the value of play is 2
The value of play is 5
Finally, the value of play is -999
```


Variable Types: Floating Point

The **int** is just one of C++'s built-in types. Another major type is the floating-point number. Here's an example of a typical floating point (**float**) declaration and use. Create a project named **float-play**, assign it to the **C++1_Lessons** working set, and in it, create a **float-play.cpp** program as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    // Floating-point variable declaration...
    float area = 5.2 * 3.5; // Area of a rectangle

    std::cout << "The area is " << area << std::endl;
    return(0);
}
```

 Save and run it. You should see the following result:

OBSERVE:

```
The area is 18.2
```

Here we saved a step—we declared a variable named **area** and assigned it the value of the expression **5.2 * 3.5** all in a single line of code. C++ does the math—it evaluates the expression and assigns **18.2** to the **area** variable.

Variable Types: Characters

The character (**char**) data type stores a single character. To be more specific, it stores a single character from the ASCII character set. This character set provides for the letters a-z (upper and lower case), the digits 0-9, a set of punctuation characters, and special *control characters*.

Glossary: [ASCII Character Set](#).


Control characters are not printed, but instead control how the output appears on the screen.

Character constants are enclosed in single quotes: 'A', 'B', '?'.
Create a **char-play** project, assign it to the **C++1_Lessons** working set, and in it, create **char-play.cpp** as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    char ch1; // First play character
    char ch2; // Second play character
    char ch3; // Third play character

    ch1 = 'A';
    ch2 = 'B';
    ch3 = 'C';
    std::cout << "The characters are " << ch1 << ch2 << ch3 << std::endl;
    return(0);
}
```

 Save and run it. You should see the following:

OBSERVE:

The characters are ABC

What happens if we try to shove more than one character into a variable? Try it! Change your program as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    char ch1; // First play character
    char ch2; // Second play character
    char ch3; // Third play character


    ch1 = 'A';
    ch2 = 'B';
    ch3 = 'CDEF';
    std::cout << "The characters are " << ch1 << ch2 << ch3 << std::endl;
    return(0);
}
```

See the warning? This is the first indication that something isn't quite correct.

```
**** Internal Builder is used for
build ****
g++ -O0 -g3 -Wall -c
-fmessage-length=0 -ochar-play.o
..\char-play.cpp
..\char-play.cpp:16:11: warning:
multi-character character constant
g++ -ochar-play.exe char-play.o
Build complete for project char-play
Time consumed: 1422 ms.
```

```
char ch1; // First ;
char ch2; // Second
char ch3; // Third ;

ch1 = 'A';
ch2 = 'B';
multi-character character constant
std::cout << "The c
return(0);
}
```

 Try running the program. You'll see something like:

OBSERVE:

The characters are ABF

Looks like the computer wasn't happy with our 'CDEF' character, so it dropped **CDE** without telling us. Warnings exist for a reason—don't ignore them!

Escape Characters

There is a special character, the backslash (\), which is used to specify characters that cannot be typed inside single quotes easily, such as tab ('\t') and newline ('\n'). In these cases, the backslash "escapes" from normal rendering of t and n, allowing you to represent tabs and newlines in program output.

Some of the escape characters are:

\b	Backspace	Move the cursor to the left one character.
\f	Form feed	Go to top of a new page.
\n	New line	Go to the next line.
\r	Return	Go to the beginning of the current line.
\t	Tab	Advance to the next tab stop (eight-column boundary).
\'	Apostrophe or single quotation mark	The character ' .
\"	Double quote	The character " .
\\	Backslash	The character \ .
\nnn	<i>some character</i>	The character number <i>nnn</i> (octal).
\xNN	<i>some character</i>	The character number <i>NN</i> (hexadecimal).

Edit the program below, replacing the value we assign to **ch2** as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    char ch1; // First play character
    char ch2; // Second play character
    char ch3; // Third play character

    ch1 = 'A';
    ch2 = '\n';
    ch3 = 'C';
    std::cout << "The characters are " << ch1 << ch2 << ch3 << std::endl;
    return(0);
}
```

 Save and run it and observe the output:

OBSERVE:

The characters are A
C

Now let's try something a little different. Change the value of ch2 to **\b**, the backspace, as shown below:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    char ch1; // First play character
    char ch2; // Second play character
    char ch3; // Third play character

    ch1 = 'A';
    ch2 = '\b';
    ch3 = 'C';
    std::cout << "The characters are " << ch1 << ch2 << ch3 << std::endl;
    return(0);
}
```



Save and run it and observe the output:

OBSERVE:

The characters are A□C

The display probably doesn't look correct—depending on how your computer handles the backspace character, you might see anything from **AC** to **A**, some funny character, and then **C**.

Wide Characters

The `char` type suffers from the fact that it cannot deal with international alphabets. To help solve this problem, the wide character (**wchar**) type was created. While the `char` type defines 256 characters, `wchar` defines 65536.

However, there are languages (such as Chinese, Japanese, Farsi, Hebrew, and many others) that contain even more than 65536 characters. To display characters from those languages, a character encoding system called *Unicode* was created.

Support for wide and Unicode characters is dependent on the compiler and operating system that you are using. We'll discuss this in a future lesson.

Boolean

The boolean type (**bool**) can have the value **true** (which is also one, or 1) or **false** (which is also zero, or 0).

Create a project named *play-bool*, assign the **C++1_Lessons** working set to it, and create the program file **play-bool.cpp** as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    bool flag; // Boolean to play around with

    flag = true; // Set it to true
    std::cout << "Flag is " << flag << std::endl;

    flag = false; // Set it to false
    std::cout << "Flag is " << flag << std::endl;

    flag = (1 == 1); // == is the test for equality operator (expression is true)
    std::cout << "Flag is " << flag << std::endl;

    return(0);
}
```



Save and run it. You should see the following output:

OBSERVE:

```
Flag is 1
Flag is 0
Flag is 1
```

The computer printed the underlying values for flag as it was set to true, then false, then true.

Originally, C did not have a boolean type, so people defined their own. As a result, you may see things like **BOOL**, **BOOLEAN**, **Bool**, **TRUE**, and **FALSE** in older books or code. Be aware that these are local, non-standard items.

Now the question comes up, "What should I do about legacy types in my code?" The best answer is, if the code works, leave it alone. You could waste a lot of time and effort trying to improve working code and bring it up to current standards, merely to make it do what it already does.

If you do have to go in and change something, *and bringing the types up to date will not cause too much trouble*, by all means do so. But don't change anything just for the sake of changing it.

Mixing Types

C++ is very flexible when it comes to mixing types. If it can figure out a conversion, it will silently allow you to assign a constant of one type (such as boolean) to another (such as integer).

Here are the conversions that occur when mixing types:

Result type	Expression type	Conversion
integer	boolean	1 for true, 0 for false (not guaranteed by the standard, but everyone I know of implements it that way).
float	boolean	Same as integer.
char	integer	The character whose character number is the number being assigned. This depends on the character set on your system. For most systems, this is ASCII , so assigning <code>ch = 65</code> makes <code>ch</code> an 'A' (ASCII character number 65).
integer	float	Integer after truncation. Note: If the value of the float is bigger than the maximum size the integer can handle, the integer will get the maximum value.
integer	char	The numeric value of the character in the current character set.
float	char	The numeric value of the character in the current character set.
boolean	Numeric	If the number 0 (zero), the boolean variable gets false . If non-zero, the value true is assigned.

Let's try this. Change your program as shown:

CODE TO TYPE:

```
#include <iostream>
int main()
{
    bool flag; // Boolean to play around with

    flag = true; // Set it to true

    float test = flag; // will this work?
    flag = false;

    std::cout << "Flag is " << flag << std::endl;
    std::cout << "Test is " << test << std::endl;

    return(0);
}
```

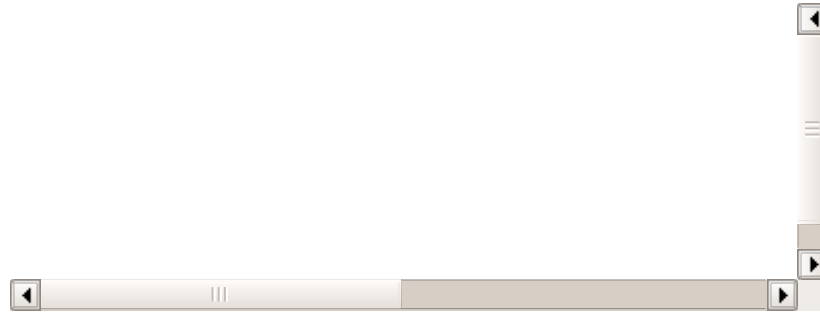


Save and run it. Your program happily stores the boolean value **1** from **flag** into **test**. Also note that, even when the value of **flag** changes to false (0), **test** retains the value originally assigned to it:

OBSERVE:

```
Flag is 0
Test is 1
```

We learned a lot about variables and types in this lesson! In the next lesson, we'll learn how to store many values of the same data type using *arrays*. See you then!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Arrays and For Loops

C++ 1: Introduction to C++ Lesson 4

Using Arrays

So far, we've only learned basic data types and every datum had a name. But suppose we want to calculate the average grade for twenty students in a class. It's a lot of work to define twenty different variables, one for each number:

OBSERVE:

```
int student1;    // A number we are going to average
int student2;    // A number we are going to average
int student3;    // A number we are going to average
// I'm not going to write out the whole thing.
...
int student20;   // A number we are going to average
```

Arrays allow us to define a set of data—like a group of mailboxes at an apartment building. Compare this to a regular variable (one that isn't an array)—it is like a mailbox outside of a single house.

A regular (non-array) variable is just a single part of your computer's memory, but an array is a chunk of continuous memory locations.

Here is how we'd declare an array to store the same set of numbers:

OBSERVE:

```
int students[20]; // The numbers we are going to average
```

The elements of the array are **students[0]** through **students[19]**. You might expect them to be 1 through 20, but in C++, array element numbering starts at 0. The numbers—0, 1, or 19—are indexes to the array.

The const Modifier

Our class might always have twenty students in it, but what if it changes, say, to 35? We would have to go through our program, find all of the **20** values relating to our **students** array, and change them manually to **35**. This process is error-prone—after all, there may be occurrences of **20** unrelated to the number of students, that we don't want to change!

Good programming practice dictates that we specify the size of our class in one place, as a variable, and then use that variable rather than the literal number in our program. Since our class size doesn't change very often, we can tell C++ that the variable is a *constant*.

The **const** modifier defines a "variable" (actually a constant) whose value cannot be changed by the program when it's running. C/C++ style conventions call for the names of constants to be all upper case; for example, **const int CLASS_SIZE = 35**.

How do constants work? Let's find out! Start a new project named **constants**, assign it to the **C++1_Lessons** working set, and in it, create a new source file named **constants.cpp** as shown:

CODE TO TYPE:

```
#include <iostream>

int main()
{
    const int CLASS_SIZE = 20;

    std::cout << "I have " << CLASS_SIZE << " students in my class." << std::endl;

    return(0);
}
```



Save and run it. If you typed everything correctly you will see the following:

OBSERVE:

I have 20 students in my class.

What happens if the program tries to change the value of **CLASS_SIZE**? Try it:

CODE TO TYPE:

```
#include <iostream>

int main()
{
    const int CLASS_SIZE = 20;

    std::cout << "I have " << CLASS_SIZE << " students in my class." << std::endl;

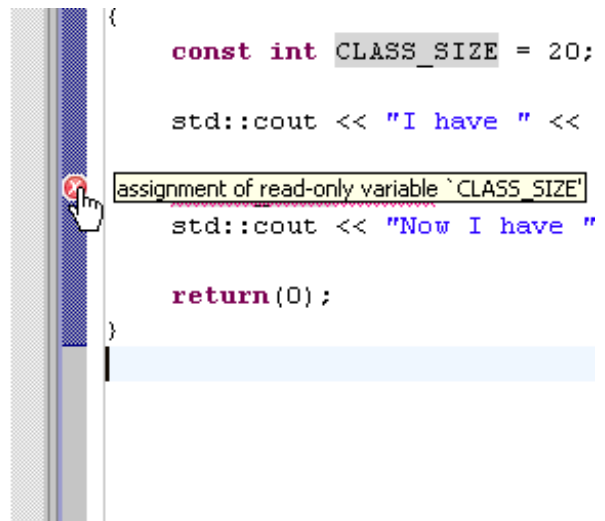
    CLASS_SIZE = 35;
    std::cout << "Now I have " << CLASS_SIZE << " students in my class." << std::endl;

    return(0);
}
```

Save your code. This time your program won't compile, because you are trying to change a constant.

```
**** Build of configuration Debug for
project constants ****

**** Internal Builder is used for
build ****
g++ -O0 -g3 -Wall -c
-fmessage-length=0 -oconstants.o
..\constants.cpp
..\constants.cpp: In function 'int
main()':
..\constants.cpp:9: error: assignment
of read-only variable 'CLASS_SIZE'
Build error occurred, build is
stopped
Time consumed: 687 ms.
```



If your number is going to change in the middle of a program run like this, it shouldn't be defined as a constant. To fix this problem, you'd need to remove the word **const** from the variable declaration and change the variable name to an appropriate style, for example, **class-size**.

You should use named constants to specify the dimensions of arrays; for example, **int**

students[CLASS_SIZE].

Good programming style dictates that you should always use named constants in this way. In other words, the dimension specification of an array should never be just a number like 20.

Our first array

Now let's look at arrays in action. Create a new project named **average**, assign it to the **C++1_Lessons** working set, and in it, create **average.cpp** as shown:

CODE TO TYPE:

```
#include <iostream>

int main()
{
    const int CLASS_SIZE = 5;

    float average = 0.0;        // average of the items
    float total = 0.0;          // the total of the data items
    float students[CLASS_SIZE]; // data to average and total

    students[0] = 34.0;
    students[1] = 27.0;
    students[2] = 46.5;
    students[3] = 82.0;
    students[4] = 22.0;

    total = students[0] + students[1] + students[2] + students[3] + students[4];
    average = total / CLASS_SIZE;
    std::cout << "Total " << total << " Average " << average << std::endl;
    return (0);
}
```



Save and run it. You should see the following result:

OBSERVE:

```
Total 211.5 Average 42.3
```

Remember how we could declare a variable and set its value at the same time? You can also do that for arrays, using {braces}. Change your program as shown:

CODE TO TYPE:


```
#include <iostream>

int main()
{
    const int CLASS_SIZE = 5;

    float average = 0.0;    // average of the items
    float total = 0.0;      // the total of the data items
    float students[CLASS_SIZE] = { 34.0, 27.0, 46.5, 82.0, 22 }; // data to average and total

    students[0] = 34.0;
    students[1] = 27.0;
    students[2] = 46.5;
    students[3] = 82.0;
    students[4] = 22.0;

    total = students[0] + students[1] + students[2] + students[3] + students[4];
    average = total / CLASS_SIZE;
    std::cout << "Total " << total << " Average " << average << std::endl;
    return (0);
}
```

 Save and run it. You should see the same results as before.

Our code so far looks good, but our total calculation is a bit unwieldy because we'll have to change the formula each time we change the **CLASS_SIZE**. Let's see how we can fix that!

for loops

We can have the computer calculate the total for us—after all, it knows how big **CLASS_SIZE** is! To do this, we can use a *for loop*—which executes a block of code a specified number of times. Change your program as shown:

CODE TO TYPE:

```
#include <iostream>


int main()
{
    const int CLASS_SIZE = 5;

    int x;

    float average = 0.0;    // average of the items
    float total = 0.0;      // the total of the data items
    float students[CLASS_SIZE] = { 34.0, 27.0, 46.5, 82.0, 22 }; // data to average and total

    total = students[0] + students[1] + students[2] + students[3] + students[4];
    for (x = 0 ; x < CLASS_SIZE ; x = x + 1)
    {
        total = total + students[x];
    }

    average = total / CLASS_SIZE;
    std::cout << "Total " << total << " Average " << average << std::endl;
    return (0);
}
```

 Save and run it. You should see the same output as before:

OBSERVE:

Total 211.5 Average 42.3

So... how does the **for** loop work? Take a look at the code:

OBSERVE:

```
for (x = 0 ; x < CLASS_SIZE ; x = x + 1)
{
    total = total + students[x];
}
```

In English, this loop might be written this way: "for **x starting at zero**, while **x is less than CLASS_SIZE**, **x is incremented by one** after we **add each student value to the total**."

Since x starts at zero and our CLASS_SIZE is 5, the code **total = total + students[x]** will run five times—with x having the values 0, 1, 2, 3, and 4.

A for loop typically has these parts:

OBSERVE:

```
for (/* Initialization */ ; /* Test */ ; /* Increment */)
{
    /* body of loop */
}
```

In our loop:

- **x = 0** is the **initialization code**. Arrays start at zero, so x must start at zero as well.
- **x < CLASS_SIZE** is the **test code**. Our array only has the number of elements (5) specified by CLASS_SIZE, but the indexes start at 0 so the last index will be 4; therefore, we want to stop looping before x is equal to CLASS_SIZE.
- **x = x + 1** is the **increment** code. x will increase by one after each iteration.
- **total = total + students[x]** is the **body of loop** code, which updates the total.

To really see what the for loop is doing, let's add some code to show the values of **x** and **total** inside the loop while it's running:

CODE TO TYPE:

```
#include <iostream>

int main()
{
    const int CLASS_SIZE = 5;

    int x;

    float average = 0.0;    // average of the items
    float total = 0.0;      // the total of the data items
    float students[CLASS_SIZE] = { 34.0, 27.0, 46.5, 82.0, 22 }; // data to average a
nd total

    for (x = 0 ; x < CLASS_SIZE ; x = x + 1)
    {
        total = total + students[x];
        std::cout << "Students[x = " << x << "]: " << students[x] << "; total: " << to
tal << std::endl;
    }

    average = total / CLASS_SIZE;
    std::cout << "Total " << total << " Average " << average << std::endl;
    return (0);
}
```



Save and run it and observe the output:

OBSERVE:

```
Students[x = 0]: 34; total: 34
Students[x = 1]: 27; total: 61
Students[x = 2]: 46.5; total: 107.5
Students[x = 3]: 82; total: 189.5
Students[x = 4]: 22; total: 211.5
Total 211.5 Average 42.3
```

A for loop can really make short work of a big list! Suppose our class size grows to ten. We can quickly update our program to handle this, simply by updating **CLASS_SIZE** and entering values for the additional students. Update your program as shown. Remember, arrays are zero-based, so we are adding students 5 through 9.

CODE TO TYPE:

```
#include <iostream>

int main()
{
    const int CLASS_SIZE = 10;

    int x = 0;

    float average = 0.0;    // average of the items
    float total = 0.0;      // the total of the data items
    float students[CLASS_SIZE] =
        { 34.0, 27.0, 46.5, 82.0, 22, 72.3, 55.9, 91.2, 90.0, 43.8 }; // data to average and total

    for (x = 0 ; x < CLASS_SIZE ; x = x + 1)
    {
        total = total + students[x];
        std::cout << "Students[x = " << x << "]: " << students[x] << "; total: " << total << std::endl;
    }

    average = total / CLASS_SIZE;
    std::cout << "Total " << total << " Average " << average << std::endl;
    return (0);
}
```



Save and run it. See how we didn't have to change the rest of the program—it just worked! Your output should look like this:

OBSERVE:

```
Students[x = 0]: 34; total: 34
Students[x = 1]: 27; total: 61
Students[x = 2]: 46.5; total: 107.5
Students[x = 3]: 82; total: 189.5
Students[x = 4]: 22; total: 211.5
Students[x = 5]: 72.3; total: 283.8
Students[x = 6]: 55.9; total: 339.7
Students[x = 7]: 91.2; total: 430.9
Students[x = 8]: 90; total: 520.9
Students[x = 9]: 43.8; total: 564.7
Total 564.7 Average 56.47
```

Array Safety

C++ makes it easy to store a lot of information in an array. What happens if you make a mistake—like accidentally loop over too many array elements? Edit the program as shown:

CODE TO TYPE:

```
//*****  
//*** WARNING: In order to see what happens when we violate the rules, this program del  
iberately overflows an array.  
//*****  
#include <iostream>  
  
int main()  
{  
    const int CLASS_SIZE = 10;  
  
    int x = 0;  
  
    float average = 0.0;    // average of the items  
    float total = 0.0;      // the total of the data items  
    float students[CLASS_SIZE] =  
        { 34.0, 27.0, 46.5, 82.0, 22 72,3, 55.9, 91.2, 90.0, 43.8 }; // data to averag  
e and total  
  
    for (x = 0 ; x < 15 ; x = x + 1) // OOPS!!  
    {  
        total = total + students[x];  
        std::cout << "Students[x = " << x << "]: " << students[x] << "; total: " << to  
tal << std::endl;  
    }  
  
    average = total / CLASS_SIZE;  
    std::cout << "Total " << total << " Average " << average << std::endl;  
    return (0);  
}
```

Save the file. No error or warnings will be generated, because the computer assumes you know what you are doing, and has not checked to make sure you are staying within the bounds of the **students** array. What happens when you try to run the program? Run it to find out!



Your program may run, and it may crash. If it runs, you might see output like this:

OBSERVE:

```
Students[x = 0]: 34; total: 34  
Students[x = 1]: 27; total: 61  
Students[x = 2]: 46.5; total: 107.5  
Students[x = 3]: 82; total: 189.5  
Students[x = 4]: 22; total: 211.5  
Students[x = 5]: 72.3; total: 283.8  
Students[x = 6]: 55.9; total: 339.7  
Students[x = 7]: 91.2; total: 430.9  
Students[x = 8]: 90; total: 520.9  
Students[x = 9]: 43.8; total: 564.7  
Students[x = 10]: 3.21401e-039; total: 564.7  
Students[x = 11]: 5.95261e-039; total: 564.7  
Students[x = 12]: 1129.4; total: 1129.4  
Students[x = 13]: 0; total: 1129.4  
Students[x = 14]: 1.96182e-044; total: 1129.4  
Total 211.5 Average 21.15
```


See the **junk** added at the end? Your specific output may look different. C++ didn't check to make sure you were staying within the bounds of your array, and happily went on with the program. It is like trying to get mail out of 15 mailboxes, when only 10 physical mailboxes exist in your apartment building.

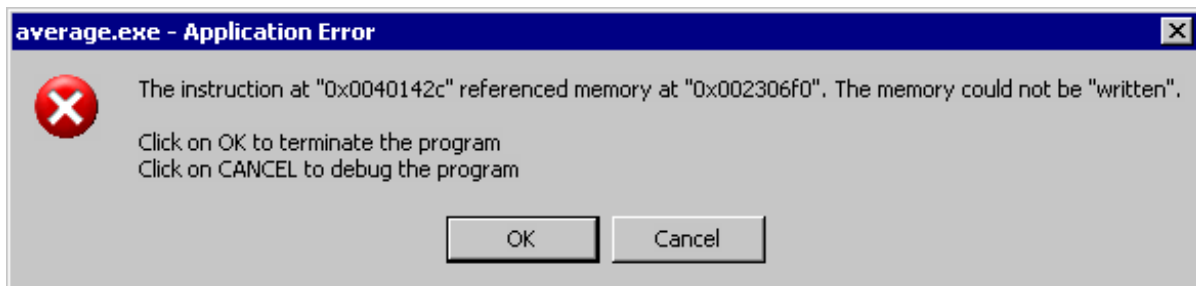
What happens if we try to write to an array location outside of CLASS_SIZE? Try it:

Code to Edit: average.cpp

```
//*****  
//** WARNING: In order to see what happens when we violate the rules, this program del  
iberately overflows an array.  
//*****  
#include <iostream>  
  
int main()  
{  
    const int CLASS_SIZE = 10;  
  
    int x = 0;  
  
    float average = 0.0;    // average of the items  
    float total = 0.0;      // the total of the data items  
    float students[CLASS_SIZE] =  
        { 34.0, 27.0, 46.5, 82.0, 22.3, 55.9, 91.2, 90.0, 43.8 }; // data to average and total  
  
    students[500] = 500.0;  
  
    for (x = 0 ; x < 15 ; x = x + 1) // OOPS!!  
    {  
        total = total + students[x];  
        std::cout << "x: " << x << " students[x]: " << students[x] << " total: " << total << std::endl;  
    }  
  
    average = total / CLASS_SIZE;  
    std::cout << "Total " << total << " Average " << average << std::endl;  
    return (0);  
}
```

Save the file. Once again, no error or warnings are generated, because the computer assumes you know what you are doing and has not checked to make sure you are staying within the bounds of the **students** array. What happens when you try to run the program?

 Run it to find out! Your program will crash, and give an error like this:



C++ did not check to make sure your array index of 500 was valid. Instead, that line executed, and crashed your program. In C++, when you try to access or write something you are not supposed to access, any of these things might happen:

- your program could crash.
- your computer could crash, depending on the operating system.
- your program could run and calculate things incorrectly.
- your program could run normally, only to crash later (perhaps waiting until it's shipped to 10,000 customers, when it suddenly decides to wipe out all their important data!).

If you remember the good old days of computers and operating systems—MS-DOS, Windows 95, and MacOS 7, to name a few—you probably remember bugs in applications that could crash the entire computer. Those bugs were often array problems just like this!

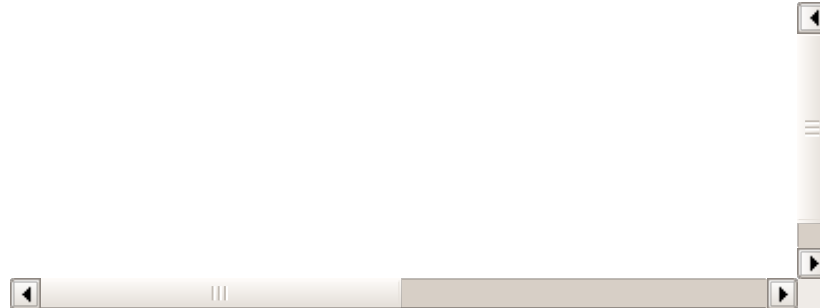
In some other languages, you might not need to worry so much about checking your array indices. In C++, you do. In a

future course, we will discuss ways to double-check your programs to make sure you are accessing arrays correctly.

Note

In the real world, almost no one checks the array indices before using them to access an array. Instead, they run without checking and spend millions of dollars later to debug the strange and hard-to-find bugs caused by bad index values. CHECK YOUR CODE!

We covered a lot in this lesson! We learned how to use arrays, and how to loop over arrays using for loops. In the next lesson we'll learn about another type of variable: *strings*. See you then!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

C++ Strings

C++ 1: Introduction to C++ Lesson 5

Strings in C++

Strings are sequences of characters you can use in a computer program. There are two basic ways to use strings—the newer C++ style strings (which we will learn about in this lesson), and older "C-Style" strings that we will review in the next lesson.

To use C++ strings, we need to **#include <string>** at the top of our programs. We can then declare a string with a statement like:

OBSERVE:

```
std::string variableName; // comment
```

Let's see how this works. Create a project named **string**, assign it to the **C++1_Lessons** working set, create a source file named **string.cpp**, and enter the following code:

CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name

    first = "Joyce";
    last = "Kilmer";

    std::cout << "My name is " << first << " " << last << std::endl;
    return(0);
}
```

Replace we'll use Joyce Kilmer in the examples, but you should replace *Joyce* and *Kilmer* with any names you like. As you can see, string constants are enclosed in double quotes ("). Remember that character constants are enclosed in single quotes ('). It might help to remember that one tick (') is for a single character and multiple ticks (") for multiple characters (strings).



Save and run the program. You'll see something like:

OBSERVE:

```
My name is Joyce Kilmer
```

Excellent! So far, strings work exactly like any other variable such as an **int** or **float**.

We learned how to add two numbers in a previous lesson. You can perform a similar operation on strings—using the plus operator (+), you can *concatenate* two strings. To see how this works, change your program as shown:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name

    first = "your first name";
    last = "your last name";
    full = first + " " + last;

    std::cout << "My name is " << full << std::endl;
    return(0);
}
```

 Save and run it. You'll see the same output:

OBSERVE:

My name is Joyce Kilmer

In this program, we concatenated the two variables **first** and **last** (with the space " " in between). What if we tried to concatenate **"Joyce"**, a space, and **"Kilmer"** directly? Change your program as shown:

CODE TO TYPE:

```
#include <iostream>
#include <string>

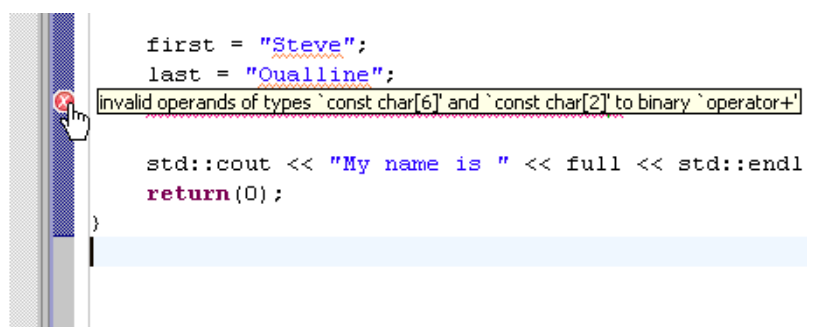
int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name

    first = "Joyce";
    last = "Kilmer";
    full = "Joyce" + " " + "Kilmer";

    std::cout << "My name is " << full << std::endl;
    return(0);
}
```

Oops! Looks like C++ isn't happy with us. It gives the following error:

```
g++ -O0 -g3 -Wall -c
-fmessage-length=0 -ostring.o
..\string.cpp
..\string.cpp: In function 'int
main()':
..\string.cpp:12: error: invalid
operands of types 'const char[6]' and
'const char[2]' to binary 'operator+'
Build error occurred, build is
stopped
Time consumed: 718 ms.
```



This is because of the way the concatenate (+) operator is defined. There must be a **std::string** on at least one side of the +.

Change your program to fix this problem:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first;    // First name
    std::string last;     // Last name
    std::string full;     // The full name

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + "Kilmer";

    std::cout << "My name is " << full << std::endl;
    return(0);
}
```

 Save and run it; you'll see the same results as before.

Characters in strings

You can access any character in the string using the subscript (`[]`) operator. Change **string.cpp** as shown:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first;    // First name
    std::string last;     // Last name
    std::string full;     // The full name
    char first_initial;   // The first initial

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + last;
    first_initial = first[0]; // Assigns first_initial the value 'J'.

    std::cout << "My first initial is " << first_initial << std::endl;
    return(0);
}
```

 Save and run it, and observe the output:

OBSERVE:

My first initial is J

This `[]` syntax might remind you of an array. This is because a string is just like an array of characters!

There is a problem here—the index is not checked in this operation. So if the index is out of range, the expression returns an undefined value. In other words, this operation is not safe—and you know how we feel about safety!

Fortunately, there is another way of getting the character, the `at()` function. Change the program as shown:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name
    char first_initial; // The first initial

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + last;
    first_initial = first.at(0); // Assigns first_initial the value 'J'. This is
    the safe way of doing this.

    std::cout << "My first initial is " << first_initial << std::endl;
    return(0);
}
```

 Save and run it. You'll see the same output as before:

OBSERVE:

My first initial is J

Now just for fun, change the line that gets the first initial to attempt to grab character 99 (an illegal index), as shown below:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name
    char first_initial; // The first initial

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + last;
    first_initial = first.at(99); // Assigns first_initial the value 'J'. This i
    s the safe way of doing this

    std::cout << "My first initial is " << first_initial << std::endl;
    return(0);
}
```

 Save and run it. You'll see the following on the console:

OBSERVE:

**This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.**

The program can't access the 99th character in the string, so it threw in the towel. This isn't quite a crash, but it isn't a good way to end your program.

Other Functions

One of the more common things to do with a string is to extract a *substring*; for example, some substrings of the full name "Joyce Kilmer" are "Joyce," "e Ki," and "Kilmer."

C++ strings let you take substrings using the **substr** function. Let's try it!

CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name
char first_initial; // The first initial

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + last;
first_initial = first.at(0); // Assigns first_initial the value 'J'. This is the safe way of doing this

    std::cout << "My substring is " << full.substr(4, 5) << std::endl;
    return(0);
}
```



Save and run it. You will see the following output:

OBSERVE:

```
My substring is e Kil
```

Let's take a look at the substr code:

OBSERVE:

```
full.substr(4, 5)
```

The **4** is the position of the starting character for the substring—in this case, the 'e' in Joyce. The **5** is the length of the substring—in this example, the program returns the characters 'e,' space, 'K,' 'i,' and 'l.'

If you omit the second number, a different substring is returned. Try it:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + last;

    std::cout << "My substring is " << full.substr(4, 5) << std::endl;
    return(0);
}
```

 Save and run it. This time you will see:

OBSERVE:

My substring is e Kilmer

C++ also has a function to tell you how many characters are in a string. It's called **length()**—let's try it:


CODE TO TYPE:

```
#include <iostream>
#include <string>

int main()
{
    std::string first; // First name
    std::string last;  // Last name
    std::string full;  // The full name

    first = "Joyce";
    last = "Kilmer";
    full = first + " " + last;

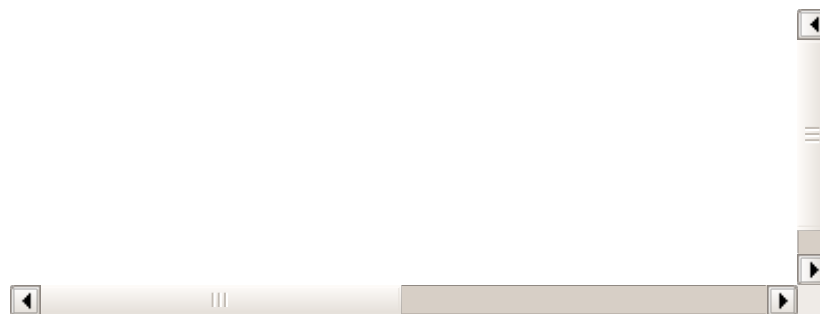
    std::cout << "My name has " << full.length() << " characters in it." <<std::
endl;
    return(0);
}
```

 Save and run it. Sure enough, it counts the characters, including spaces!

OBSERVE:

My name has 12 characters in it.

In the next lesson we will learn more about strings and discuss C-Strings. See you then!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

C-Style Strings

C++ 1: Introduction to C++ Lesson 6

What is a C-Style String?

In the last lesson we learned about C++ strings. There is an additional, different type of string—the C-style string.

C-style strings are the primary string type of the C language. Since C++ is a superset of C, it too has use C-style strings. In previous lessons we examined characters and arrays—a C-style string is essentially an array of characters with the special character null ('\0') at the end.

Since C-style strings are arrays, they have a fixed maximum length. "Dynamic" strings are possible, but they require the programmer to manually manage memory—a process that is full of pitfalls and gotchas. Many security problems and bugs refer to [buffer overflows](#)—which are typically errors with C-style strings that result in memory corruption.

There are many who believe that you should never have a C-style string in a C++ program. Unfortunately "should" and the "real world" are two quite different things. There is still a lot of legacy code out there that uses C-style strings. As a programmer in the real world, you will see this type of string and have to deal with it.

So let's start with an experimental C-style string program. Create a project named **c-string**, assign it to your **C++1_Lessons** working set, and create a program **c-string.cpp** containing the following:


Code to Type: c-string.cpp

```
#include <iostream>

int main()
{
    char name[] = {'S', 'a', 'm', '\0'};    // The name for this example

    std::cout << "The name is " << name << std::endl;
    return(0);
}
```

This assigns the value "Sam" to a C string **name**, which can be output to **std::cout**.

 Save and run it, and observe the output:

OBSERVE:

The name is Sam

Now what would happen if you didn't end the string with '\0'? Let's see. Change your program as shown:

Code to Edit: c-string.cpp

```
#include <iostream>
int main()
{
    char name[] = {'S', 'a', 'm', '\0'};    // The name for this example
    char other[] = {'J', 'o', 'e'};        // Another name with an error

    std::cout << "The name is " << name << " and other is " << other << std::endl;
    return(0);
}
```

 Save and run it, and observe the output:

OBSERVE:

The name is Sam and other is Joew

Your actual output might differ from this example—in fact, it might even look normal. This is another situation where the behavior is undefined. If you do not put an end-of-string marker (0) in your C-style string, it is anybody's guess what will occur.

Remember when we accidentally used an array element we weren't supposed to use? What happened? Because there was no end-of-string marker, C++ did not stop at the end of **other**. It continued to write out characters from random memory until it actually found an end-of-string character.


C++ allows for even more compact initialization, using double quotes ("). Let's use it to fix the error:

Code to Edit: c-string.cpp

```
#include <iostream>
int main()
{
    char name[] = {'S', 'a', 'm', '\0'};    // The name for this example
    char other[] = "Joe";    // Another name with an error

    std::cout << "The name is " << name << " and other is " << other << std::endl;
    return(0);
}
```

This form of initialization creates an array *four* characters long and assigns it four character values, the fourth and last being the end-of-string character.

 Save and run it, and observe the output:

OBSERVE:

The name is Sam and other is Joe

You cannot assign a value to an existing C-style string. This is because a C string is an array and you cannot change its value like you can with other variables. Try it:

Code to Edit: c-string.cpp

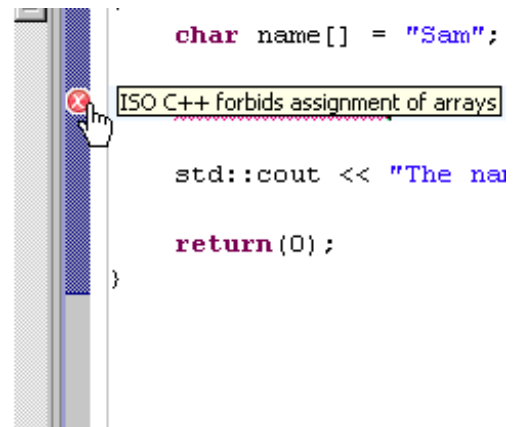
```
#include <iostream>
int main()
{
    char name[] = "Sam";    // The name for this example
char other[] = "Joe";    // Another name with an error
    name = "Joe"; // Will this work?

    std::cout << "The name is " << name << " and other is " << other << std::endl;
    return(0);
}
```

When you save this file you will see an error:

```
**** Build of configuration Debug for project
c-string ****

**** Internal Builder is used for build
****
g++ -O0 -g3 -Wall -c -fmessage-length=0
-oc-string.o ..\c-string.cpp
..\c-string.cpp: In function `int main()':
..\c-string.cpp:6: error: ISO C++ forbids
assignment of arrays
Build error occurred, build is stopped
Time consumed: 625 ms.
```



This error is generated because C strings require the programmer to worry about memory and do extra work when copying them. Think of the mailbox analogy from the array lesson—here we created a mailbox with three slots, containing the values "S," "a," and "m." Using **name = "Joe"**; is like buying a new mailbox with three slots and trying to shove the new mailbox with slots "J," "o" and "e" where the old mailbox is still hanging.

Instead of trying to smash one mailbox in place of another, we must manually open each slot and copy the contents from the new to the old. This is done using the **strncpy()** function (see the reference information at cplusplus.com). Note this reference uses an old header file (**string.h**) instead of the current one (**cstring**).

This function has three parameters:

strncpy() Syntax

```
std::strncpy( destination , source , size );
```

destination is where the data is to be put, **source** is the source of the data, and **size** is the maximum number of characters to put in the destination.

Before we can use **strncpy**, we need to get out our tape measure—an operator named **sizeof()**. **sizeof()** returns the size of something in "char" units. If we use it on a C-style string, it will return the maximum number of characters that can be stored in the string.

This is very important because our strings (mailboxes) are fixed in size. If we try to copy too many characters from our new mailbox to the old, bad things could happen. Copying too many characters is yet another way to overflow the buffer and possibly cause the program to crash.

Let's see an example. Edit **c-string.cpp** as shown:

Code to Edit: c-string.cpp

```
#include <cstring>
#include <iostream>

int main()
{
    char name[4]; // Short name

    std::cout << "Size is " << sizeof(name) << std::endl;
    std::strncpy(name, "Joe", sizeof(name));
    std::cout << "Name is now " << name << std::endl;

    return (0);
}
```



Save and run it. You'll see:

OBSERVE:

```
Size is 4
Name is now Joe
```

You might be wondering why you see **Size is 4**—after all, **Joe** is only three letters long! The fourth character is the end of string ('\0') null character. Remember: this character is required, so you must take it into account.

What happens when we try to copy a larger string into a smaller variable? Let's try it:

Code to Edit: c-string.cpp


```
#include <cstring>
#include <iostream>

int main()
{
    char name[4]; // Short name

    std::cout << "Size is " << sizeof(name) << std::endl;
    std::strncpy(name, "Joe", sizeof(name));
    std::cout << "Name is now " << name << std::endl;

    std::strncpy(name, "Steve", sizeof(name));
    std::cout << "Name is now " << name << std::endl;

    return (0);
}
```

 Save and run it. This time, you'll see something like this:

OBSERVE:

```
Size is 4
Name is now Joe
Name is now Stev a"
```

The output from your program may be slightly different, see how "Steve" wasn't copied correctly, and the output contains extra characters?

We used the **sizeof** operator to compute the maximum number of characters that can be stored in the variable **name**. *This included the null character.* "Steve" is six characters—"Steve" plus the end-of-string character.

If the source string has fewer characters than the size of the array, then **std::strncpy()** copies the string and adds an end-of-string ('\0') to the end. But since the source is bigger here, it copies *size's* number of characters and does not append the end-of-string. So in order to make things work, we must do so ourselves:

Code to Edit: c-string.cpp

```
#include <cstring>
#include <iostream>


int main()
{
    char name[4]; // Short name

    std::cout << "Size is " << sizeof(name) << std::endl;
    std::strncpy(name, "Joe", sizeof(name));
    std::cout << "Name is now " << name << std::endl;

    std::strncpy(name, "Steve", sizeof(name));
    name[sizeof(name)-1] = '\0';
    std::cout << "Name is now " << name << std::endl;

    return (0);
}
```

Since **sizeof(name)** is 4 in our example, **sizeof(name)-1** will be 3. Remember arrays (and C-style strings, which are arrays of characters) are zero-based, so 3 is the last mailbox in the **name** string.

 Save and run it. This time you'll see:

OBSERVE:

```
Size is 4
Name is now Joe
Name is now Ste
```

We only see the first three characters of "Steve" because **name** is only big enough to contain four characters—"Ste" plus the null character. Believe it or not, changing the size of C-style strings is not as straightforward as you might expect. That topic will be covered in a future course.

Concatenation of C-Style Strings

In the last lesson, we covered the **length()** function for C++-style strings. The function **std::strlen()** returns the length of a C-style string. In other words, it returns the number of characters actually in the string, as opposed to **sizeof()**, which returns the *capacity*.

The function to perform concatenation of C-style strings is **std::strncat()**. Unlike concatenation of C++ strings, concatenation of C-style strings requires some planning to make sure you don't overflow any buffers. The function takes three parameters:

OBSERVE:

```
std::strncat( destination , source, size );
```

You must carefully calculate **size** in order to make sure you don't overflow the **destination**. The easiest way to do this is to always use the following code:

Concatenation Design Pattern

```
std::strncat( destination , source, sizeof(dest) - std::strlen(dest) - 1 );
destination[sizeof(destination)-1] = '\0';
```

The calculation for the **size** parameter for **std::strncat()** works like this:

Code	Description
<code>sizeof(destination)</code>	Start with the size of the destination string in characters.
<code>- std::strlen(destination)</code>	Subtract the number of characters already in the string.
<code>- 1</code>	Subtract one more for the end-of-string ('\0') character.

Let's try an example! Edit **c-string.cpp** as shown:

Code to Edit: c-string.cpp

```
#include <cstring>
#include <iostream>

int main()
{
    char name[25]; // Short name with plenty of space

    std::cout << "Size is " << sizeof(name) << std::endl;
    std::strncpy(name, "Joe", sizeof(name));
    std::cout << "Name is now " << name << std::endl;

    std::strncat(name, " Smith", sizeof(name) - std::strlen(name) - 1);
    name[sizeof(name)-1] = '\0';
    std::cout << "Name is now " << name << std::endl;

    return (0);
}
```



Save and run it, and observe the output:

OBSERVE:

```
Size is 25  
Name is now Joe  
Name is now Joe Smith
```

Success!

Comparing Strings

The C-style string comparison function is `std::strcmp()`. It takes two parameters:

OBSERVE:

```
std::strcmp( string1, string2)
```

It returns:

- 0 if the strings are equal
- A positive value if the first character that does not match has a greater value in string1 than in string2
- A negative value if the first character that does not match has a greater value in string2 than in string1

Let's see how it works. Create a **compare-c** project and assign it to your **C++1_Lessons** working set. Then, create a program named **compare-c.cpp** as shown:

Code to Type: compare-c.cpp

```
#include <cstring>  
#include <iostream>  
  
int main()  
{  
    char str1[] = "Steve";  
    char str2[] = "Steven";  
  
    int result = std::strcmp(str1, str2);  
    std::cout << "Result is " << result << std::endl;  
    return (0);  
}
```



Save and run it, and observe the output:

OBSERVE:

```
Result is -1
```

You might be asking yourself why we can't use the `==` equality operator to check to see if two strings are the same. Let's try it to see how it might work:

Code to Edit: compare-c.cpp

```
#include <cstring>
#include <iostream>

int main()
{
    char str1[] = "Steve";
    char str2[] = "Steve";

    if (str1 == str2)
    {
        std::cout << "Same!" << std::endl;
    }
    else
    {
        std::cout << "Not the same!" << std::endl;
    }

    return (0);
}
```



Save and run it, and observe the output:

OBSERVE:

Not the same!

Why is this? The two strings obviously have the same value.

Using our mailbox metaphor, the `==` equality operator checks to see if the two mailboxes are the same (like if they have the same serial number)—it doesn't check their contents. This fails because the mailboxes are not the same.

Tips

Converting C++ Strings to C-Style Strings

You can convert from C-style strings to C++ style strings, and vice versa. To get a C-style string from a C++-style string, use the `c_str()` function. You can assign a C-style string to a C++-style string. Try it:

Code to Edit: c-string.cpp

```
#include <iostream>
int main()
{
    char name[] = "Sam";    // The name for this example
    char other[] = "Joe";   // Another name
    char c_style[4];        // New variable to hold C++ string

    std::string cpp_style;

    cpp_style = name;    // Assigning to a C++-style string

    std::strncpy(c_style, cpp_style.c_str(), 4);    // Copy C-string to a variable

    std::cout << "The name is " << name << " and other is " << other << std::endl;
    std::cout << "cpp_style as a C-style string is " << c_style << std::endl;

    return(0);
}
```

In this example we:

1. Copied a C-style string to a C++ style string, using direct assignment: `cpp_style = name`
2. Copied a C++ string to a C-style string, using `c_str()` and `std::strncpy`



Save and run it, and observe the output. To us, the results look the same:

OBSERVE:

```
The name is Sam and other is Joe
cpp_style as a C-style string is Sam
```

Unsafe String Functions

Many people don't use the string copy design pattern we provided, and thus buffer overflow problems occur in many programs. Instead, they use the function [`std::strcpy\(\)`](#). The standard form of this function is:

Unsafe Use of `strcpy()`

```
// Unsafe. Do not use.
std::strcpy(destination, source);
```

Where **destination** is where the data will be copied into and **source** is the string to copy.

Paranoid programmers will ask themselves "What happens if the source is bigger than the destination?" The **`strcpy()`** function does not check length and if the source is too big, it will happily write random memory, corrupting your program.

Tip Paranoia, in programmers, is actually a *good* quality.

In the [style guide](#), we recommend that you *not* use the **`strcpy()`** function.

In the real world, you might encounter legacy code containing **`strcpy()`**—a lot of code still uses this function. So, what should you do when you see it? Ideally, to make the program safe, replace all **`strcpy()`** functions with **`strncpy()`**, but any time you change a program there is risk—for example, you may not make the change correctly. If the code is working, even if it is messy, the best thing to do is to *leave it alone*. Unless there is a bug in the program that forces you to rewrite the code, *leave working code alone*.

There are two times you would want to upgrade **`strcpy()`** to the C-string **`strncpy()`** design pattern. The first is if you are changing the code anyway. (Always leave code better than when you found it.) The second is when you are trying to track down a memory corruption bug—in this case, the change might fix the bug.

Another unsafe function is [`std::strcat\(\)`](#). It performs much the same function as **`std::strcpy()`**, except that it does concatenation instead of copying.

`std::strcat` (unsafe)

```
// Unsafe. Do not use.
std::strcat(destination, source);
```

This function adds the **source** string to the end of the **destination** *with no regard for the size of the destination*.

The future of `strcpy()` and `strcat()`

People have done all sorts of things to get around the limitations of **`strcpy()`** and **`strcat()`** for years. Currently, the OpenBSD folks have devised new functions, **`strlcpy()`** and **`strlcat()`**, designed to overcome the safety problems with **`strcpy()`** and **`strcat()`**, respectively. However, their effort has not made it into the standard yet.

For more information, see [strlcopy in Wikipedia](#).

Comparisons to other types

C Strings vs. Arrays of Characters

C-style strings and arrays of characters are two different things. An array of characters is of fixed length and contains no markers or other special characters. In other words, `char name[50]` contains 50 characters of any type, no more or less.

Note In a character array, the null character (`'\0'`) need not be present; it's just another character, with no special meaning.

A C-style string is built on the character array type. It states that you have an array of characters, with an end-of-string marker (`'\0'`) to end it. So, null (`'\0'`) cannot be part of the string.

So all C-style strings are arrays of characters, but all arrays of characters are not C-style strings. Let's try an example. Edit your `compare-c.cpp` program as shown:

Code to Edit: `compare-c.cpp`

```
#include <cstring>
#include <iostream>

int main()
{
    char state[2] = {'C', 'A'};

    std::cout << "This probably looks funny: " << std::endl;
    std::cout << state << std::endl;

    std::cout << "This looks better: " << std::endl;
    std::cout << state[0] << state[1] << std::endl;

    return (0);
}
```

You might look at the state declaration and think it is an error, that it should be `char state[3]` in order to reserve one character for the end of string.

But this is not a mistake. `state` is *not* a C-style string; it's a character array. It holds two characters. No more, no less (there are no one-character state name abbreviations).



Save and run it, and observe the output. Generally, you'll see something like:

OBSERVE:

```
This probably looks funny:
CA Ĳ"
This looks better:
CA
```

When you use `std::cout << state <<`, C++ will assume that `state` is a C-style string (it's not). It will then look for the end-of-string marker (there is none) and write the state abbreviation followed by some random garbage.

You must write a character array one character at a time:

Writing a Character Array

```
std::cout << state[0] << state[1] << std::endl;
```

The bottom line: You can write out a C-style string using `<<` but not a character array.

C-Style vs. C++ Style

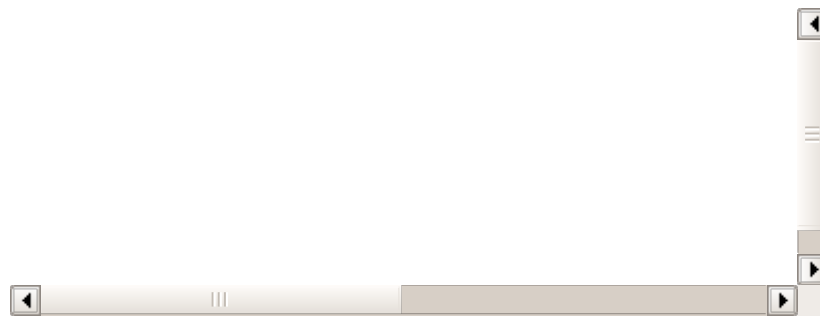
There are advantages and disadvantages to using each type of string.

C-style

C++ strings can store any length string automatically. You must explicitly declare the

Size	maximum size of C-style strings.
Memory	The memory used by C-style strings is precisely controlled. They do not grow or shrink depending on what data you put into them. C++ style strings manage their own memory. They can grow and shrink. They can also use memory in surprising ways if you are not careful.
Operations	Almost all of the operations you can use on C++ style strings are safe. Almost all of the operations you can use on C-style strings can be dangerous. You must be very careful about safety so you do not cause any buffer overflows.
Efficiency	C-style strings are more efficient than C++ style strings. However, as a practical matter, most programs are not CPU limited so efficiency makes little difference in a program. Safety does, and that's where C++ style strings win.
OS Interaction	If you are interacting directly with an operating system like Windows or Linux, you will find that many of the lower-level operating system functions (raw <code>read</code> and others) use C-style strings as arguments. This means that if you pass data from one part of the OS to another, C-style strings are more efficient.

Have you had enough of strings yet? In our next lesson we'll finally move on from output to input, and we'll start learning how to make decisions in our programs. See you there!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Reading Data and if

C++ 1: Introduction to C++ Lesson 7

Reading Strings

In earlier lessons we used `std::cout` to output data to the console. If we want to read data in from the console, we can use—surprise!—`std::cin`.

Let's look at a short program that demonstrates the use of `std::cin`. Create a project named **reading**, assign it to the **C++1_Lessons** working set, then type a **reading.cpp** program in it as shown:


Code to Type: reading.cpp

```
#include <iostream>

int main() {
    std::string name;

    std::cout << "Enter your name: ";
    std::cin >> name;

    std::cout << "You typed " << name << std::endl;
    return (0);
}
```

 Save and run it. Enter a first name, a space, and a last name (separated by a space) and press **Enter**. You see something like:

OBSERVE:

```
Enter your name: Jimi Hendrix
You typed Jimi
```

The program only reads the first name. You probably know "Jimi" by his first name, but that won't work for everyone!

But all is not lost! The solution is to use the standard function [std::getline](#). The general form of this function is:

getline syntax

```
std::getline( input-stream , string );
```

The *input-stream* is the source of the data—in our case, it's `std::cin`. Change your reading program as shown:


Code to Edit: reading.cpp

```
#include <iostream>

int main() {
    std::string name;

    std::cout << "Enter your name: ";
    std::getline(std::cin, name);

    std::cout << "You typed " << name << std::endl;
    return (0);
}
```

 Save and run it again. Enter the first and last name and press **Enter**. This time the result is much better:

OBSERVE:

Enter your name: **Jimi Hendrix**
You typed Jimi Hendrix

std::getline reads a full line up to and including the end-of-line character. The result (minus the end of line) is stored in the string.

Reading Integers

std::cin and the **>>** operator work for all sorts of variables, including integers. Edit your reading program as shown:

reading.cpp

```
#include <iostream>

int main() {
    int value = 0;        // a value to double

    std::cout << "Enter a value: ";
    std::cin >> value;
    std::cout << "Twice " << value << " is " << value * 2 << std::endl;
    return (0);
}
```



Save and run it, then click in the console window, type **123** and press **Enter**. You will see:

OBSERVE:

Enter a value: **123**
Twice 123 is 246

Let's look at this code more closely.

Observe: reading.cpp

```
#include <iostream>

int main() {
    int value = 0;        // a value to double

    std::cout << "Enter a value: ";
    std::cin >> value;
    std::cout << "Twice " << value << " is " << value * 2 << std::endl;
    return (0);
}
```

Back in the first lesson, we learned that the **<<** operator means to "put whatever's on the right into whatever's on the left." Since we are using **std::cin** and want to get data from the console, we use **>>** to go the other direction and "get whatever's on the right *from* whatever's on the left." In our case, we get data from **std::cin** and put it in the variable named **value**.

Also, look at the line **std::cout << "Enter a value: ";**. Notice that there's no **std::endl** at the end. We omitted this deliberately to make this output line a prompt.

In real-world programming, paranoia is part of the job. So one question you need to ask at this point is "What happens if I type something that isn't a number?" Let's try it! Run your program again, enter "Jimi Hendrix," and press **Enter**.

This time you'll see the following:

OBSERVE:

```
Enter a value: Jimi Hendrix
Twice 0 is 0
```

When you try to assign an inappropriate type of data to the **value** variable, it doesn't change.

if Statements

Now that we can read data, we are going to create a program that reads a number and tells you if it's even. Create a project named **if**, assign it to your **C++1_Lessons** working set. In that project, create a program named **if.cpp** as shown:

Code to Type: if.cpp

```
/*
 * if Show the use of the if statement
 *
 * In this case the program will tell you if a number
 * is even or odd.
 *
 * Usage:
 *     Run the program.
 *     Type in a number when prompted.
 *     Get the answer.
 */
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( (number % 2) == 0)
    {
        std::cout << number << " is even" << std::endl;
    }
    return(0);
}
```



Save and run the program. Enter an even number like **8**. You will see the following output:

OBSERVE:

```
Enter a number: 8
8 is even
```

If you enter an odd number like 7, you won't see any output at all.

How does the **if** statement work? The key line is:

if Statement in Use

```
if ( (number % 2) == 0)
```

This says to C++, "compute the value of the expression **number % 2**." The % is the modulus operator—it calculates the remainder after division. If the value is equal to (**==**) zero, the program executes the statements in the following block (enclosed in curly braces {}).

if statements have the following structure:

OBSERVE:

```
if ( conditional test )
{
    Code to execute when conditional test is true.
}
else
{
    Code to execute when conditional test is false. You do not need to have an "else."
}
```

There are other comparison operators you can use (in if statements and elsewhere):

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Let's try a different comparison. Change your program as shown:

Code to Edit: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( number >= 100 )
    {
        std::cout << number << " is big!" << std::endl;
    }
    else
    {
        std::cout << number << " is not so big." << std::endl;
    }
    return(0);
}
```

In this program, we changed our conditional test slightly and added an "else" clause.



Save and run it. Enter **250**; you will see:

OBSERVE:

```
Enter a number: 250
250 is big!
```

If you enter a smaller number, like -250, you'll see the other message instead:

OBSERVE:

```
Enter a number: -250
-250 is not so big.
```

if Abuse

Equality or Assignment?

In our first example of the `if` statement, we used the `==` equality operator to see if a number was even or not. What would happen if you used a single `=` instead? Try it:

Code to Edit: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( number = 100 )
    {
        std::cout << number << " is one hundred!" << std::endl;
    }
    else
    {
        std::cout << number << " is not one hundred." << std::endl;
    }

    return(0);
}
```



Save and run it, and enter **25**. You'll see:

OBSERVE:

```
Enter a number: 25
100 is big!
```

Try it with other numbers: -0, -25, 150. No matter what you type, the computer thinks you entered 100!

This is because we didn't use the **equality operator**—instead, we changed the value of **number**. In other words, this line:

OBSERVE:

```
if (number = 100)
```

was essentially interpreted as:

OBSERVE:

```
number = 100;
if (number != 0) {
```

This sort of error can be tricky to spot. An easy way to avoid this issue is to always place constants on the left side of the comparison. Change your code to the following:

Code to Edit: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

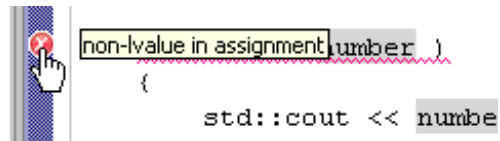
    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( 100 = number )
    {
        std::cout << number << " is one hundred!" << std::endl;
    }
    else
    {
        std::cout << number << " is not one hundred." << std::endl;
    }

    return(0);
}
```



Save your program. See how there is a new error:



This error indicates you are trying to change the value of "100" to whatever is stored in **number**, but "100" isn't a variable, so you cannot do that sort of assignment.

Fix the error by using the == equality operator:

Code to Edit: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( 100 == number )
    {
        std::cout << number << " is one hundred!" << std::endl;
    }
    else
    {
        std::cout << number << " is not one hundred." << std::endl;
    }

    return(0);
}
```

Blocks

C++ allows you to write **if** statements in a compact way. Change your program as shown:

Code to Edit: if.cpp


```
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( 100 == number )
        std::cout << number << " is one hundred!" << std::endl;
    else
        std::cout << number << " is not one hundred." << std::endl;

    return(0);
}
```

 Save and run your program a few times with different numbers. You will see this program works just like the prior version. The if statement does not require you to use braces {}—it executes the statement immediately following the if or else.

OBSERVE:

```
if ( conditional test )
    Code to execute when conditional test is true.
else
    (Optional) Code to execute when conditional test is false. You do not need to have an "else."
```

This can be very problematic, though. Change your program as shown (including spaces):

Code to Edit: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( 50 <= number ) // if #1
        if ( 100 == number ) // if #2
            std::cout << number << " is one hundred!" << std::endl;
    else // else #1
        std::cout << number << " is not so big." << std::endl;
    else // else #2
        std::cout << number << " is ??? " << std::endl;

    return(0);
}
```

Confused? You should be! Without braces (and proper indentation), it is very hard to figure out what exactly will happen in this program. Which **else** goes with which **if**? Possible answers include:

- if #1 goes with else #1
- if #2 goes with else #1

- if #1 goes with else #2
- If you don't write code like this, you won't have to worry about silly questions like this.

The correct answer is number **4**!

You need to know this so you can debug other people's code. People who value compact code over readability, understandability, and safety. However, since we find readability, understandability, and safety valuable and we write good code, we will never write code like this.

Note

Some people tell you to always use {} for the statements affected by an **if**. The *Perl* language requires it. We considered this, but decided that, for the most part, letting the programmer decide whether {} or a single statement is clearer.

Conditional Shortcuts

If statements also let you take shortcuts in the **conditional test**. In the C++ world, **0** is false, and anything else is true. With this in mind, people have devised shortcuts, such as the following:

Code to Edit: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( ! number )
    {
        std::cout << number << " is zero!" << std::endl;
    }
    else
    {
        std::cout << number << " is not zero." << std::endl;
    }

    return(0);
}
```

The **!** is the logical negation operator—it adds a "not" to the condition. The conditional statement in this program might read like "if not number"—which is pretty confusing.



Save and run the program. Enter 0 (zero); you'll see this:

OBSERVE:

```
Enter a number: 0
0 is zero.
```

If you enter a non-zero number (even -25), you'll see the following:

OBSERVE:

```
Enter a number: -25
-25 is not zero.
```

This code is not good because it isn't clear exactly what is going on. Instead, be explicit with your if statements. Change your program as shown:

Code to Type: if.cpp

```
#include <iostream>

int main()
{
    int number; // A number we are going to check

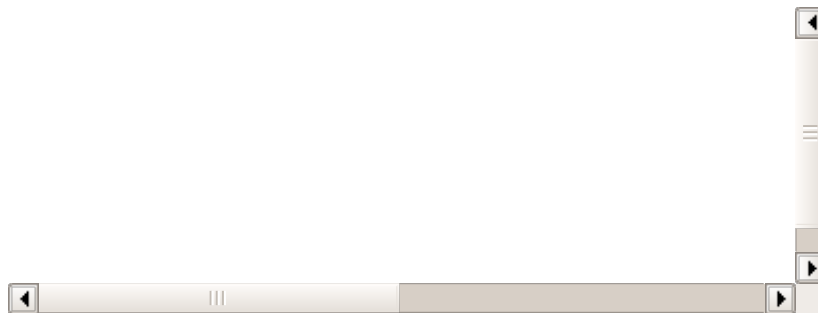
    std::cout << "Enter a number: ";
    std::cin >> number;

    if ( 0 != number )
    {
        std::cout << number << " is not zero." << std::endl;
    }
    else
    {
        std::cout << number << " is zero." << std::endl;
    }

    return(0);
}
```

This program is clearer—the condition reads "if number is not equal to zero." Much better!

You made it! In the next lesson we will shift gears and discuss some shortcuts that are common in C++ programs. See you then!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Shortcuts

C++ 1: Introduction to C++ Lesson 8

There are many different ways to accomplish the same task in C++. Some code patterns occur very often, so the developers of C and C++ have included shortcuts that can make code shorter and easier to understand.

Operators

Consider the following operation:

Incrementing a Variable's Value
<code>x = x + 5;</code>

This code pattern occurs frequently in C++ (and other languages), so the makers of C++ added a shortcut operator to the language:


Incrementing with a Shortcut
<code>x += 5;</code>

Other shortcut operators are:

Longhand	Short cut
<code>a = a - b;</code>	<code>a -= b;</code>
<code>a = a * b;</code>	<code>a *= b;</code>
<code>a = a / b;</code>	<code>a /= b;</code>
<code>a = a % b;</code>	<code>a %= b;</code>
<code>a = a + 1;</code>	<code>a++;</code>
<code>a = a - 1;</code>	<code>a--;</code>

Let's experiment with some shortcut operators. Start a new project named **short cut**, and assign it to your **C++_Lessons** working set. In the new project, create a source file named **short cut.cpp**.

Code to Type: shortcut.cpp
<pre>#include <iostream> int main() { int x = 0; // to play std::cout << "x is " << x << std::endl; x += 5; std::cout << "x is " << x << std::endl; x++; std::cout << "x is " << x << std::endl; return(0); }</pre>

 Save and run your program. You'll see the value of **x** change:

OBSERVE:

```
x is 0
x is 5
x is 6
```

One of the most common uses of these shortcuts is in **for** loops. Let's take a look:

Code to Edit: shortcut.cpp

```
#include <iostream>

int main() {

    int x = 0; // to play

    for (x = 0 ; x < 25; x++ )
    {
        std::cout << "x is " << x << std::endl;
    }

    return(0);
}
```



Save and run it. You'll see:

OBSERVE:

```
x is 0
x is 1
x is 2
x is 3
x is 4
x is 5
x is 6
x is 7
x is 8
x is 9
x is 10
x is 11
x is 12
x is 13
x is 14
x is 15
x is 16
x is 17
x is 18
x is 19
x is 20
x is 21
x is 22
x is 23
x is 24
```

We can use **+=** to have our loop skip by five instead:

Code to Edit: shortcut.cpp

```
#include <iostream>

int main() {

    int x = 0; // to play

    for (x = 0 ; x < 25; x += 5 )
    {
        std::cout << "x is " << x << std::endl;
    }

    return(0);
}
```



Save and run it. See the difference:

OBSERVE:

```
x is 0
x is 5
x is 10
x is 15
x is 20
```

For Loops

It turns out the for loop has a few more tricks. Suppose we wanted to write a program to add several numbers, but no more than ten. We don't want to include any negative numbers, and if we encounter a zero, we will know our list is done.

We can make this happen with two shortcuts: **continue** and **break**. Clear your program and type the following:

Code to Edit: shortcut.cpp

```
#include <iostream>

int main() {
    int total;    // Total so far
    int count;    // Count of the numbers
    int number;   // A number to read

    total = 0;

    for (count = 0; count < 10; ++count)
    {
        std::cout << "Enter a number: ";
        std::cin >> number;

        // Skip all negative numbers
        if (number < 0) {
            std::cout << "Negative numbers don't count." << std::endl;
            continue;
        }

        if (number == 0) {
            std::cout << "I guess you want to end the list" << std::endl;
            break;
        }

        total += number;
        std::cout << "The new total is " << total << std::endl;
    }

    std::cout << "The grand total is " << total << std::endl;
    return(0);
}
```



Save and run it. Enter a few positive numbers, a negative number, and finally a zero. You'll see something like this:

OBSERVE:

```
Enter a number: 1
The new total is 1
Enter a number: 1
The new total is 2
Enter a number: 1
The new total is 3
Enter a number: 1
The new total is 4
Enter a number: 1
The new total is 5
Enter a number: -5
Negative numbers don't count.
Enter a number: 9
The new total is 14
Enter a number: 0
I guess you want to end the list
The grand total is 14
```

When the program encounters a negative number, it outputs a message and then runs **continue**. This skips the rest of the for loop, and goes on to the next number.

When you enter a zero, the program outputs a message and then runs **break**, which completely exits the for loop and outputs the grand total.

Instead of using **continue** and **break**, you could have accomplished similar results by using if statements and perhaps another variable, but continue and break make the loop much easier to understand.

Suppose you want to allow users to enter as many numbers as they want—and only quit the program when they enter

a zero. You can change your for loop to accomplish this as well:

Code to Edit: shortcut.cpp

```
#include <iostream>

int main() {
    int total;    // Total so far
    int count;    // Count of the numbers
    int number;   // A number to read

    total = 0;

    for (;;)
    {
        std::cout << "Enter a number: ";
        std::cin >> number;

        // Skip all negative numbers
        if (number < 0) {
            std::cout << "Negative numbers don't count." << std::endl;
            continue;
        }

        if (number == 0) {
            std::cout << "I guess you want to end the list" << std::endl;
            break;
        }

        total += number;
        std::cout << "The new total is " << total << std::endl;
    }

    std::cout << "The grand total is " << total << std::endl;
    return(0);
}
```



Save and run it. It will continue until you enter a zero.

Earlier, we discussed the sections of a for loop:

OBSERVE:

```
for (/* Initialization */ ; /* Test */ ; /* Increment */)
{
    /* body of loop */
}
```

The initialization, test, and increment sections are all optional. Only the semicolons (;) are required.

WARNING This can create an infinite loop if you don't have an appropriate **break** in your code!

For Loop Misuse

The goal of programming is to be as clear and correct as possible. However, some people think it's to use as few characters as possible. I hope you never have to debug their code.

One of the tricks they have is to put the comma (,) operator into a **for** statement.

Coding Horror

```
for (twos = 0, threes = 0; twos < 100; twos +=2, threes += 3)
```


The comma (,) operator can be used to string two C++ statements together and have the compiler treat them as one. *Do not use it!* All it really does is make it easy to write bad code. In this case, we have two initialization statements: **twos = 0** and **threes = 0**. Because of the comma operator, C++ does not object to them both being inside a **for** initialization.

The same holds true for the increment section. Two statements have been stuffed into a place where only one should go.

What this loop is supposed to do is to count up two variables, **twos** by **2** and **threes** by **3**, all in one loop. What it really does is to cause good programmers to curse the people who think they are being clever by writing such code.

Side Effects

A side effect is an effect that occurs in addition to the main effect of a statement. C++ allows you to use the ++ and -- operators inside other expressions. Let's take a look. Clear your program and enter the following:

Code to Edit: shortcut.cpp

```
#include <iostream>

int main() {
    int total_size;    // Total so far
    int current_size;  // Count of the numbers

    total_size = 5;
    current_size = -3;

    current_size = ++total_size;

    std::cout << "current_size: " << current_size << std::endl;
    std::cout << "total_size: " << total_size << std::endl;

    return(0);
}
```

Do you know what the program will do?



Save and run it. The results may surprise you:

OBSERVE:

```
current_size: 6
total_size: 6
```

This is bad code! The line **current_size = ++total_size;** does two things (in this order):

1. Increments **total_size**.
2. Assigns the value of **total_size** to **current_size**.

This is bad programming style because it makes the code harder to read. Two short operations are much easier to maintain and understand than one complex one.

There is another problem with side effects. Change your program as shown:

Code to Edit: shortcut.cpp

```
#include <iostream>


int main() {
    int total_size;    // Total so far
    int current_size;  // Count of the numbers

    total_size = 0;
    current_size = 1;

    total_size = (++current_size * 5) + (++current_size * 3);

    std::cout << "current_size: " << current_size << std::endl;
    std::cout << "total_size: " << total_size << std::endl;

    return(0);
}
```

 Save and run it. Once again, the output may surprise you:

OBSERVE:

```
current_size: 3
total_size: 19
```

The code `total_size = (++current_size * 5) + (++current_size * 3)` tells C++ to:

1. Increment `current_size` and multiply the result by 5
2. Increment `current_size` and multiply the result by 3
3. Add the results from steps 1 and 2 together.

There is no rule that tells C++ which step (step 1 or step 2) to execute first. Depending on the compiler, the execution order could:

1. Increment the **FIRST** `current_size` from 1 to 2 and multiply the result by 5 and get 10.
2. Increment the **SECOND** `current_size` and multiply the result by 3 and get 9.
3. Add the results from steps 1 and 2 together and get 19.

Or, it could:

1. Increment the **SECOND** `current_size` from 1 to 2 and multiply the result by 3 and get 6.
2. Increment the **FIRST** `current_size` from 2 to 3 and multiply the result by 5 and get 15.
3. Add the results from steps 1 and 2 together and get 21.

So, which result is right? They both are! The C++ standard allows for this ambiguity. That's one of the reasons the style guide prohibits this type of coding. We want to make the code safer and more reliable.

You should never use the increment (`++`) and decrement (`--`) operators inside another expression, especially an assignment statement. But some people do, so it's important to know how they work.

The prefix increment (`++x`) operator increments the variable and returns the result **after** incrementing.

The postfix increment (`x++`) operator increments the variable and returns the result **before** incrementing.

The mnemonic I use is that if you see the `++` first, then C++ increments first, then returns the value. If you see the value first, C++ returns the value first, then increments it.

Now let's rewrite our program, so it does one thing at a time, with no side effects.

Code to Edit: shortcut.cpp

```
#include <iostream>

int main() {
    int total_size;    // Total so far
    int current_size;  // Count of the numbers

    int first_term;    // for first term
    int second_term;   // for second term

    total_size = 0;
    current_size = 1;

    current_size += 1;
    first_term = current_size * 5;


    current_size += 1;
    second_term = current_size * 3;

    total_size = first_term + second_term;

    std::cout << "current_size: " << current_size << std::endl;
    std::cout << "total_size: " << total_size << std::endl;

    return(0);
}
```

This program is a little longer than the last, but it is very clear what is happening.

 Save and run it. You'll see:

OBSERVE:

```
current_size: 3
total_size: 19
```

We covered a lot in this lesson! In the next we will examine another loop we can use in our programs: *while* loops. See you then!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

While Loops

C++ 1: Introduction to C++ Lesson 9

while, break, and continue

Welcome back! In the previous lessons we have used for loops to do some repetitive work. In this lesson, we'll learn about *while* loops—another way to do repetitive work.

The basic form of the while loop is:

```
while Syntax

while (condition)
{
    statement;
}
```

In the last lesson, we used an "empty" for loop that would let us input as many numbers as we wanted, and give us the grand total after we entered a zero. We can do the same thing using a while loop instead. Start a new project named **while**, and assign it to the **C++1_Lessons** working set. In the new project, create a new file named **while.cpp** as shown:

Code to Type: while.cpp

```
#include <iostream>

int main() {
    int total;    // Total so far
    int number;   // A number to read

    total = 0;


    while( true )
    {
        std::cout << "Enter a number: ";
        std::cin >> number;

        // Skip all negative numbers
        if (number < 0) {
            std::cout << "Negative numbers don't count." << std::endl;
            continue;
        }

        if (number == 0) {
            std::cout << "I guess you want to end the list" << std::endl;
            break;
        }

        total += number;
        std::cout << "The new total is " << total << std::endl;
    }

    std::cout << "The grand total is " << total << std::endl;
    return(0);
}
```

 Save and run your program. You'll soon notice it runs exactly the same as the for loop we used before—right down to the **continue** and **break** statements.

Suppose we want our program to run until we enter a zero OR our grand total is greater than 50. We can alter the **condition** in our loop to accomplish this:

Code to Edit: while.cpp

```
#include <iostream>

int main() {
    int total;    // Total so far
    int number;   // A number to read

    total = 0;

    while( total <= 50 )
    {
        std::cout << "Enter a number: ";
        std::cin >> number;

        // Skip all negative numbers
        if (number < 0) {
            std::cout << "Negative numbers don't count." << std::endl;
            continue;
        }

        if (number == 0) {
            std::cout << "I guess you want to end the list" << std::endl;
            break;
        }

        total += number;
        std::cout << "The new total is " << total << std::endl;

    }

    std::cout << "The grand total is " << total << std::endl;
    return(0);
}
```



Save and run it. Enter the numbers 5, 10, 15, 20, and 1—you should see your program stop:

OBSERVE:

```
Enter a number: 5
The new total is 5
Enter a number: 10
The new total is 15
Enter a number: 15
The new total is 30
Enter a number: 20
The new total is 50
Enter a number: 1
The new total is 51
The grand total is 50
```

Fibonacci numbers

The [Fibonacci](#) numbers are numbers in a sequence starting with "0 1," where each subsequent number is the sum of the previous two:

Fibonacci Number Calculations

```
0 + 1 = 1
  1 + 1 = 2
    1 + 2 = 3
      2 + 3 = 5
        3 + 5 = 8
          5 + 8 = 13... and so on.
```

Thus, the first Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, and 13. (Yes, 1 is in there twice.)

We can use a while loop to calculate the sequence of Fibonacci numbers less than 100. Create a new project named **fib** and assign it to your **C++1_Lessons** working set, and in that project, create a source file named **fib.cpp** as shown:

Code to Type: fib.cpp

```
#include <iostream>

int main()
{
    int    old_number;    // previous Fibonacci number
    int    current_number; // current Fibonacci number
    int    next_number;   // next number in the series

    // start things out
    old_number = 0;
    current_number = 1;

    std::cout << "0 "; // Output first number

    while (current_number < 100) {
        std::cout << current_number << ' ';
        next_number = current_number + old_number;
        old_number = current_number;
        current_number = next_number;
    }
    std::cout << std::endl;

    return (0);
}
```



Save and run it. You should see the Fibonacci sequence:

OBSERVE:

0 1 1 2 3 5 8 13 21 34 55 89

How did we do that? Let's look at the program more closely.

fib.cpp

```
#include <iostream>

int main()
{
    int    old_number = 0;    // previous Fibonacci number
    int    current_number = 1; // current Fibonacci number
    int    next_number;      // next number in the series

    std::cout << old_number; // Output first number

    while (current_number < 100) {
        std::cout << current_number << ' ';
        next_number = current_number + old_number;
        old_number = current_number;
        current_number = next_number;
    }
    std::cout << std::endl;

    return (0);
}
```

From the formula for Fibonacci numbers, we know the starting values and that the next number is the sum of the

previous ("old") and current numbers, so we created variables for the **old_number** (set to 0), the **current_number** (set to 1), and **next_number** (which we'll set in our loop).

The first Fibonacci number is 0—the initial value of **old_number**—so we'll just output it before we start the loop. (But rather than print "0", we'll use the variable, so if we ever want to change it, we only need to change it in one place in the program.)

We want to continue looping while the number is less than 100, so we can add a **while loop** to our program.

Inside the while loop, we **display the current number**, then **add the old and current numbers to get the value of the next number**.

Now, to move the sequence along, we shift the values: **move the current_number value to old_number and the next_number value to current_number** before the next calculation.

Finally, we do a little housekeeping. Our program outputs everything on a single line, so we **add an end-of-line character** after the loop finishes.

Excellent! To get a better idea how Fibonacci numbers, and while loops work, let's add some **cout** statements. Add the colorized code:

Code to Edit: fib.cpp

```
#include <iostream>
#include <iomanip>

int main()
{
    int    old_number = 0;        // previous Fibonacci number
    int    current_number = 1;    // current Fibonacci number
    int    next_number;          // next number in the series
    int    iteration_count = 1;

    std::cout << "Iteration  old  current next <100?" << std::endl;

    while (current_number < 100) {

        std::cout << std::setw(2) << iteration_count;
        std::cout << std::setw(12) << old_number;
        std::cout << std::setw(9) << current_number;

        next_number = current_number + old_number;
        old_number = current_number;
        current_number = next_number;

        std::cout << std::setw(5) << next_number;
        if (current_number < 100)
        {
            std::cout << "  T";
        }
        else
        {
            std::cout << "  F";
        }
        std::cout << std::endl;

        iteration_count = iteration_count + 1;
    }
    std::cout << std::endl;

    return (0);
}
```

We included **iomanip**—specifically the **std::setw()** function—to help format our output. This function sets the padded width of the next item in the **cout** chain. For example, the following code ensures that **iteration_code** is output as exactly two characters:

OBSERVE:

```
std::cout << std::setw(2) << iteration_count;
```



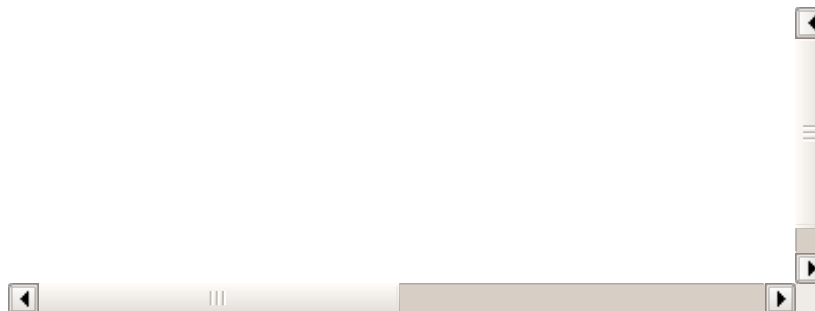
Save and run it. This time, you'll see a beautifully formatted table:

OBSERVE:

Iteration	old	current	next	<100?
1	0	1	1	T
2	1	1	2	T
3	1	2	3	T
4	2	3	5	T
5	3	5	8	T
6	5	8	13	T
7	8	13	21	T
8	13	21	34	T
9	21	34	55	T
10	34	55	89	T
11	55	89	144	F

The output shows how variables are saved for the next trip around the while loop. By the tenth iteration we have the first Fibonacci number (stored in **current_number**) that is greater than or equal to 100, so the program stops.

You've added some extremely helpful tools to your C++ tool kit! In the next lesson, we'll discuss the *scope* of our variables. See you then!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Scope

C++ 1: Introduction to C++ Lesson 10

What is Scope?

Up to this point, we've used very basic variable declarations. We've declared variables at the top of our programs and used them throughout the entire program. In other words, all variables existed through the entire program.

In this lesson, we'll see how to create variables that exist for only a portion of the program.

We'll start with a short program. Create a project named **var-exp** and assign it to your **C++1_Lessons** working set. In this new project, create a program named **var-exp.cpp** as shown:

Code to Type: var-exp.cpp

```
#include <iostream>

int main()
{
    std::string state = "Texas";
    std::cout << "State is " << state << std::endl;
    return (0);
}
```



Save and run this program. As you might expect, you will see the following output:

OBSERVE:

State is Texas

Now, edit the program [as shown](#):

```
#include <iostream> int main() { std::string state = "Texas"; std::cout << "State is " << state << std::endl; return (0); }
```

Code to Edit: var-exp.cpp

```
#include <iostream>

int main()
{
    std::string state = "Texas";
    std::cout << "State #1 is " << state << std::endl;
    {
        std::string city = "Austin";
        std::cout << "City is " << city << std::endl;
    }
    return (0);
}
```



Save and run it. You'll see:

OBSERVE:

State #1 is Texas
City is Austin

Edit the program again as shown:

Code to Edit: var-exp.cpp

```
#include <iostream>

int main()
{
    std::string state = "Texas";
    std::cout << "State #1 is " << state << std::endl;
    {
        std::string city = "Austin";
        std::cout << "City is " << city << std::endl;
    }
    std::cout << "City #2 is " << city << std::endl;
    return (0);
}
```



Save your program; this time you'll notice an error:

The compiler is telling us that **city** isn't defined—but we *did* define it... didn't we?

Not quite. The variable **city** has a *local scope*. Scope is the portion of a program in which the variable is known. **city** only exists within the curly brace ({}) block enclosing it—not outside of it.

Compare this to the **state** variable. It also has a local scope, but its enclosing braces include the entire program.


Is the **state** variable accessible from the middle of our program? Let's see. Change your program:

}

Code to Edit: var-exp.cpp

```
#include <iostream>

int main()
{
    std::string state = "Texas";
    std::cout << "State #1 is " << state << std::endl;
    {
        std::string city = "Austin";
        std::cout << "City is " << city << std::endl;
        std::cout << "State #2 is " << state << std::endl;
    }
    std::cout << "City #2 is " << city << std::endl;
    return (0);
}
```

 Save and run it. You'll see:

OBSERVE:

```
State #1 is Texas
City is Austin
State #2 is Texas
```


Though we added a block in braces and a local variable to our list of variables, **state** is still within the scope of the program.

Now, let's see if we can access **state** AFTER the block in braces. Change your program:

Code to Edit: var-exp.cpp

```
#include <iostream>

int main()
{
    std::string state = "Texas";
    std::cout << "State #1 is " << state << std::endl;
    {
        std::string city = "Austin";
        std::cout << "City is " << city << std::endl;
        std::cout << "State #2 is " << state << std::endl;
    }
    std::cout << "State #3 is " << state << std::endl;
    return (0);
}
```

 Save and run it. You'll see:

OBSERVE:

```
State #1 is Texas
City is Austin
State #2 is Texas
State #3 is Texas
```

Global Variables

Like their geographical counterparts, you're in the state when you're in the city, but you're not (necessarily) in the city when you're in the state.

In our example, the **state** variable is actually a local variable. It exists only inside the curly braces that enclose it (but it's still available within other braces nested in those braces). Right now that happens to be our entire program, but we will deal with more complex programs in the next few lessons.

A variable that exists outside of **main()** and in fact exists everywhere is called a *global* variable. **std::cout**, for example, is a global variable. It exists before **main()** starts and after it ends.

Let's declare our own global variable named **country**. Edit your program as shown:

Code to Edit: var-exp.cpp

```
#include <iostream>

std::string country = "USA";    // A global variable

int main()
{
    std::string state = "Texas";
    std::cout << "State #1 is " << state << std::endl;
    std::cout << "Country #1 is " << country << std::endl;
    {
        std::string city = "Austin";
        std::cout << "City is " << city << std::endl;
        std::cout << "State #2 is " << state << std::endl;
        std::cout << "Country #2 is " << country << std::endl;
    }
    std::cout << "State #3 is " << state << std::endl;
    std::cout << "Country #3 is " << country << std::endl;
    return (0);
}
```



Save and run it. You'll see:

OBSERVE:

```
State #1 is Texas
Country #1 is USA
City is Austin
State #2 is Texas
Country #2 is USA
State #3 is Texas
Country #3 is USA
```

In the city, you can "see" the state and the country. In the state, you can "see" the country but not the city. In the country, you can't see the state or city. It's an imperfect analogy, but it's okay for the purpose of illustration, right?

Storage Class

The storage class of a variable can be permanent or temporary. The local variables we've defined are temporary. They are created when they are declared and disappear when their enclosing block ends.

Global variables are permanent. They are created (and initialized) when the program starts, and are not destroyed until the program ends.

Let's create a quick program to take a look at this situation. Create a project named **var-types** and assign it to the **C++1_Lessons** working set. In the new project, create a program named **var-types.cpp** as shown:


Code to Type: var-types.cpp

```
#include <iostream>
int global = 1; // Global variable to play around with

int main()
{
    int loop;    // A loop counter

    for (loop = 0; loop < 3; ++loop) {
        int temp = 1; // A local variable to play around with

        std::cout << "global is " << global << std::endl;
        std::cout << "temp is " << temp << std::endl;
        ++global;
        ++temp;
        // Almost useless comment
    }
    return (0);
}
```

 Save and run it and observe the output:

Output of var-types

```
global is 1
temp is 1
global is 2
temp is 1
global is 3
temp is 1
```

The value of *temp* never changes—but why? The answer: *scope*. Let's take a closer look at the life cycle of **temp**.

var-types.cpp

```
#include <iostream>
int global = 1; // Global variable to play around with

int main()
{
    int loop;    // A loop counter

    for (loop = 0; loop < 3; ++loop) {
        int temp = 1; // A local variable to play around with

        std::cout << "global is " << global << std::endl;
        std::cout << "temp is " << temp << std::endl;
        ++global;
        ++temp;
        // Almost useless comment
    }
    return (0);
}
```

The variable is created by the line **int temp = 1;**. It then is incremented by the line **++temp**, so its value is 2. It then is destroyed just after the line **// Almost useless comment**, so it now has no value because it doesn't exist. The **for** loop starts another loop. The variable is born again with the line **int temp = 1;**.

It's important to remember that the scope of **temp** is local and the storage class is temporary.

We can make a local variable permanent by putting the keyword **static** in front of it. Edit your program as shown:

Code to Edit: var-types.cpp

```
#include <iostream>
int global = 1; // Global variable to play around with

int main()
{
    int loop;    // A loop counter

    for (loop = 0; loop < 3; ++loop) {
        int temp = 1; // A local variable to play around with
        static int perm = 1; // A local, permanent variable to play with

        std::cout << "global is " << global << std::endl;
        std::cout << "temp is " << temp << std::endl;
        std::cout << "perm is " << perm << std::endl;
        ++global;
        ++temp;
        ++perm;
        // Almost useless comment
    }
    return (0);
}
```



Now, save and run it again and observe the output:

Output of var-types

```
global is 1
temp is 1
perm is 1
global is 2
temp is 1
perm is 2
global is 3
temp is 1
perm is 3
```

The storage class of **perm** is permanent. It is created and initialized when the program is created, and that means that it is initialized once. Every time through the loop it is incremented by one, unlike **temp**, which is re-initialized every time through the loop.

On the other hand, the variable **perm** always stays around.

Note

The **static** keyword is the most overloaded keyword in C++. It has many different meanings, depending on where you use it. For local variables, it changes the storage class to permanent. We will examine its other uses as they arise.

for Loop Scope

In general, the scope of a local variable is restricted to the block ({}) in which it resides. The **for** statement is special in that you can declare the loop variable right inside the **for** itself.

Edit var-types.cpp as shown:

Code to Edit: var-types.cpp

```
#include <iostream>
int global = 1; // Global variable to play around with

int main()
{
    int loop; // A loop counter

    for (int loop = 0; loop < 3; ++loop) {
        int temp = 1; // A local variable to play around with
        static int perm = 1; // A local, permanent variable to play with

        std::cout << "loop is " << loop << std::endl;
        std::cout << "global is " << global << std::endl;
        std::cout << "temp is " << temp << std::endl;
        std::cout << "perm is " << perm << std::endl;
        ++global;
        ++temp;
        ++perm;
        // Almost useless comment
    }
    return (0);
}
```



Save and run it. You'll see:

OBSERVE:

```
loop is 0
global is 1
temp is 1
perm is 1
loop is 1
global is 2
temp is 1
perm is 2
loop is 2
global is 3
temp is 1
perm is 3
```

In this case, the **loop** variable has a scope of the entire body of the **for** loop.

Hidden Variables

Now we'll discuss *hidden* variables. First let's see them in action. Create a program called *hidden.cpp* and type in the program below.


Code to Type: hidden.cpp

```
/*
 * hidden -- A very good demonstration of what not to do.
 * More of a puzzle than a useful program.
 */
#include <iostream>

int main()
{
    int a_var = 2;
    int b_var = 5;

    std::cout << "a_var #1 is " << a_var << std::endl;
    std::cout << "b_var #1 is " << b_var << std::endl << std::endl;
    {
        int a_var = 3;
        std::cout << "a_var #2 is " << a_var << std::endl;
        std::cout << "b_var #2 is " << b_var << std::endl << std::endl;
    }
    std::cout << "a_var #3 is " << a_var << std::endl;
    std::cout << "b_var #3 is " << b_var << std::endl;

    return (0);
}
```

 Save and run it. You'll see:

OBSERVE:

```
a_var #1 is 2
b_var #1 is 5

a_var #2 is 3
b_var #2 is 5

a_var #3 is 2
b_var #3 is 5
```

It looks like **a_var** switched values in the middle of the program to **3** and then back to **2**. But did it?

Not quite. So what's happening? Let's start by looking at the scope of **b_var** since we haven't played any games with it.

Scope of b_var

```
int main()
{
    int a_var = 2;
    int b_var = 5;

    std::cout << "a_var #1 is " << a_var << std::endl;
    std::cout << "b_var #1 is " << b_var << std::endl << std::endl;
    {
        int a_var = 3;
        std::cout << "a_var #2 is " << a_var << std::endl;
        std::cout << "b_var #2 is " << b_var << std::endl << std::endl;
    }
    std::cout << "a_var #3 is " << a_var << std::endl;
    std::cout << "b_var #3 is " << b_var << std::endl;

    return (0);
}
```


b_var's scope includes all of the code shown in **green**. Now we'll show the scope of **int a_var = 3;**—we need to say **int a_var = 3;** to identify the variable instead of **a_var**, because there are two **a_var** variables. This confusion should provide a clue as to why hidden variables are a bad thing.

Here's the scope of **int a_var = 3;**:

Scope of a_var(3)
<pre>int main() { int a_var = 2; int b_var = 5; std::cout << "a_var #1 is " << a_var << std::endl; std::cout << "b_var #1 is " << b_var << std::endl << std::endl; { int a_var = 3; std::cout << "a_var #2 is " << a_var << std::endl; std::cout << "b_var #2 is " << b_var << std::endl << std::endl; } std::cout << "a_var #3 is " << a_var << std::endl; std::cout << "b_var #3 is " << b_var << std::endl; return (0); }</pre>

Now let's add **int a_var = 2** to the mix:

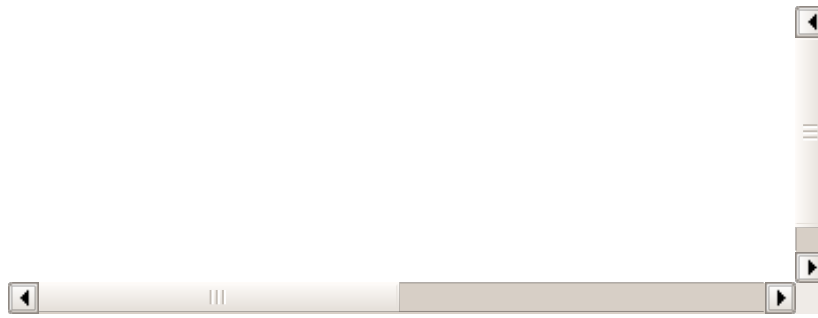
Scope of a_var(2) and a_var(3)
<pre>int main() { int a_var = 2; int b_var = 5; std::cout << "a_var #1 is " << a_var << std::endl; std::cout << "b_var #1 is " << b_var << std::endl << std::endl; { int a_var = 3; std::cout << "a_var #2 is " << a_var << std::endl; std::cout << "b_var #2 is " << b_var << std::endl << std::endl; } std::cout << "a_var #3 is " << a_var << std::endl; std::cout << "b_var #3 is " << b_var << std::endl; return (0); }</pre>

Because **a_var(3)** has the innermost scope where it is defined, it *hides* **a_var(2)** in the middle of the program. Thus, the scope for **a_var(2)** has a hole in it.

In other words, the declaration of **a_var(3)** hides **a_var(2)** in the **dark red** area.

Avoid using hidden variables whenever possible. That's because if you say something like "And here I print the value of **a_var**," someone has to ask you, "*Which a_var?*" There's enough confusion in the programming world now without us adding more!

You made it! In the next lesson we will put scope to work with *functions*. Stay tuned!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Functions

C++ 1: Introduction to C++ Lesson 11

What is a Function?

In prior lessons, our programs were fairly short. Repetition so far has been limited to for and while loops.

In this lesson we'll learn how to use *functions* to organize frequently-used code. A function is like a black box—you provide the box with some parameters, and it returns the result.

Our First Function

Let's suppose we want to compute the area of a right triangle (the formula is **area = (base * height) / 2**).

Create a project named **triangle** and assign it to your **C++1_Lessons** working set. In the new project, create a source file named **triangle.cpp** as shown:

Code to Type: triangle.cpp

```
#include <iostream>
int main()
{
    float width;
    float height;
    float area;

    width = 5.0;
    height = 2.0;
    area = (width * height) / 2.0;
    std::cout << "Area of the first triangle is " << area << std::endl;

    return(0);
}
```



Save and run the program. You'll see:

OBSERVE:

Area of the first triangle is 5

Excellent! Suppose you now want to calculate area for two more triangles. Extend your program as shown:

Code to Edit: triangle.cpp

```
#include <iostream>
int main()
{
    float width;
    float height;
    float area;

    width = 5.0;
    height = 2.0;
    area = (width * height) / 2.0;
    std::cout << "Area of the first triangle is " << area << std::endl;

    width = 3.8;
    height = 5.9;
    area = (width * height) / 2.0;
    std::cout << "Area of the second triangle is " << area << std::endl;

    width = 2.8;
    height = 1.6;
    area = (width * height) / 2.0;
    std::cout << "Area of the third triangle is " << area << std::endl;

    return (0);
}
```



Save and run the program. You'll see:

OBSERVE:

```
Area of the first triangle is 5
Area of the second triangle is 11.21
Area of the third triangle is 2.24
```

In our program, we have lines of code that repeat:

OBSERVE:

```
area = (width * height) / 2.0;
std::cout << "Area of the (nth) triangle is " << area << std::endl;
```

Let's take those lines of code and turn them into a function. Edit your program:

Code to Edit: triangle.cpp

```
#include <iostream>

/*
 * triangle -- compute the area of a triangle
 *
 * Parameters
 *     width -- the width of the triangle
 *     height -- the height of the triangle
 *
 * Returns
 *     the area of the triangle
 */

float triangle(float width, float height) {
    float area = (width * height) / 2.0;
    return (area);
}

int main()
{
    float width;
    float height;
    float area;

    width = 5.0;
    height = 2.0;
    area = (width * height) / 2.0;
    std::cout << "Area of the first triangle is " << area << std::endl;

    width = 3.8;
    height = 5.9;
    area = (width * height) / 2.0;
    std::cout << "Area of the second triangle is " << area << std::endl;

    width = 2.8;
    height = 1.6;
    area = (width * height) / 2.0;
    std::cout << "Area of the third triangle is " << area << std::endl;

    return (0);
}
```

Let's take a closer look at the triangle() function:

OBSERVE:

```
float triangle(float width, float height) {
    float area = (width * height) / 2.0;
    return (area);
}
```

The first line specifies the **return type**, **name**, and **parameters** of the function. Our function named **triangle** will return a **float** value. It requires two parameters—the first is a **float named width**, the second is a **float named height**. After calculating the **area**, the function **returns** it.

Now that we've defined our function, let's update our program to call it:

Code to Edit: triangle.cpp

```
#include <iostream>

/*
 * triangle -- compute the area of a triangle
 *
 * Parameters
 *     width -- the width of the triangle
 *     height -- the height of the triangle
 *
 * Returns
 *     the area of the triangle
 */

float triangle(float width, float height) {
    float area = (width * height) / 2.0;
    return (area);
}

int main()
{
    float width;
    float height;
    float area;

    width = 5.0;
    height = 2.0;
    area = (width * height) / 2.0;
    area = triangle(5.0, 2.0);
    std::cout << "Area of the first triangle is " << area << std::endl;

    width = 3.8;
    height = 5.9;
    area = (width * height) / 2.0;
    area = triangle(3.8, 5.9);
    std::cout << "Area of the second triangle is " << area << std::endl;

    width = 2.8;
    height = 1.6;
    area = (width * height) / 2.0;
    area = triangle(2.8, 1.6);
    std::cout << "Area of the third triangle is " << area << std::endl;

    return (0);
}
```



Save and run it. You'll see the same output as before:

OBSERVE:

```
Area of the first triangle is 5
Area of the second triangle is 11.21
Area of the third triangle is 2.24
```

For each triangle calculated, we replaced three lines of code with one. We're making progress! Do you another area where we could save a few lines of code?

Void Functions and Array Parameters

In our program, we could also add code to the `triangle()` function to display the message like::

OBSERVE:

```
float triangle(float width, float height) {  
    float area = (width * height) / 2.0;  
  
    std::cout << "Area of the triangle is " << area << std::endl;  
  
    return (area);  
}
```

This message would be a *side effect* of the **triangle()** function. A function has a side effect if it outputs messages, modifies files, or otherwise performs some action besides returning a value. Generally speaking, side effects are not a good idea.

Instead, let's create a separate function to output the message. Change your program:

Code to Edit: triangle.cpp

```
#include <iostream>  
  
/*  
 * triangle -- compute the area of a triangle  
 *  
 * Parameters  
 *     width -- the width of the triangle  
 *     height -- the height of the triangle  
 *  
 * Returns  
 *     the area of the triangle  
 */  
float triangle(float width, float height) {  
    float area = (width * height) / 2.0;  
    return (area);  
}  
  
/*  
 * print_it -- output a message  
 *  
 * Parameters  
 *     area -- the area of the triangle  
 *     what -- the name of the triangle  
 */  
void print_it(float area, char what[])  
{  
    std::cout << "The area of the " << what << " triangle is " << area << std::e  
endl;  
}  
  
int main()  
{  
    float area; // Area of a triangle  
  
    area = triangle(5.0, 2.0);  
    std::cout << "Area of the first triangle is " << area << std::endl;  
  
    area = triangle(3.8, 5.9);  
    std::cout << "Area of the second triangle is " << area << std::endl;  
  
    area = triangle(2.8, 1.6);  
    std::cout << "Area of the third triangle is " << area << std::endl;  
  
    return (0);  
}
```

Let's take a closer look at **print_it()**:

OBSERVE:

```
void print_it(float area, char what[])
{
    std::cout << "The area of the " << what << " triangle is " << area << std::e
    ndl;
}
```

This function named `print_it()` returns `void`, which means "nothing" in C++. In other words, our function does not return any value. It has two parameters—a `float` named `area` and a `C-Style string` named `what` (a character array named `what`). The character array parameter has no dimension. That is because the dimension is determined by the code that calls the `print_it()`, as you will see shortly.

Note

Some other languages might call `print_it()` a procedure, because it does not return a value, and call `triangle` a function because it does return a value. C++ just has one construct, a function.

Our function does not have a `return` statement because it isn't required inside a `void` function. Let's add it, and revise our program to call `print_it()`:

Code to Edit: triangle.cpp

```
#include <iostream>

/*
 * triangle -- compute the area of a triangle
 *
 * Parameters
 *     width -- the width of the triangle
 *     height -- the height of the triangle
 *
 * Returns
 *     the area of the triangle
 */
float triangle(float width, float height) {
    float area = (width * height) / 2.0;
    return (area);
}

/*
 * print_it -- output a message
 *
 * Parameters
 *     area -- the area of the triangle
 *     what -- the name of the triangle
 */
void print_it(float area, char what[])
{
    std::cout << "The area of the " << what << " triangle is " << area << std::e
    ndl;

    return;
}

int main()
{
    float area; // Area of a triangle

    area = triangle(5.0, 2.0);
std::cout << "Area of the first triangle is " << area << std::endl;
    print_it(area, "first");

    area = triangle(3.8, 5.9);
std::cout << "Area of the second triangle is " << area << std::endl;
    print_it(area, "second");

    area = triangle(2.8, 1.6);
std::cout << "Area of the third triangle is " << area << std::endl;
    print_it(area, "third");

    return (0);
}
```



Save and run it. You'll see the same output again:

OBSERVE:

```
The area of the first triangle is 5
The area of the second triangle is 11.21
The area of the third triangle is 2.24
```

Function Overloading

Our `print_it()` function works fine, but what if we didn't always want to pass the second parameter? In C++, we can *overload* a function, which means that we can have two or more different functions with the same

name and different parameters.

Let's define a new `print_it()` with only one parameter instead of two. Edit your program as shown:

Code to Edit: triangle.cpp

```
#include <iostream>

/*
 * triangle -- compute the area of a triangle
 *
 * Parameters
 *     width -- the width of the triangle
 *     height -- the height of the triangle
 *
 * Returns
 *     the area of the triangle
 */
float triangle(float width, float height) {
    float area = (width * height) / 2.0;
    return (area);
}

/*
 * print_it -- output a message
 *
 * Parameters
 *     area -- the area of the triangle
 *     what -- the name of the triangle
 */
void print_it(float area, char what[])
{
    std::cout << "The area of the " << what << " triangle is " << area << std::e
ndl;

    return;
}

/*
 * print_it -- output a message without the "what"
 *
 * Parameters
 *     area -- the area of the triangle
 */
void print_it(float area)
{
    std::cout << "The area of the triangle is " << area << std::endl;

    return;
}

int main()
{
    float area; // Area of a triangle

    area = triangle(5.0, 2.0);
    print_it(area, "first");

    area = triangle(3.8, 5.9);
    print_it(area, "second");

    area = triangle(2.8, 1.6);
    print_it(area, "third");

    return (0);
}
```



Save and run it. You'll see this slightly different result:

OBSERVE:

```
The area of the triangle is 5
The area of the second triangle is 11.21
The area of the third triangle is 2.24
```

The first time we call `print_it()` with only one parameter, and C++ runs the correct `print_it()` function that accepts one parameter (the second one in the program). The next two times, we call it with two parameters, and C++ runs the correct function that accepts two parameters (the first one in the program).

C++ keeps track of this by using *function signatures*. A function signature is the combination of the function name, return data type, parameter data types, and names. In our program, there are actually four functions:

1. `float triangle(double width, double height)`
2. `void print_it(float area, char what[])`
3. `void print_it(float area)`
4. `int main()`

In C++, the function name and parameters must be unique. In other words, you cannot define two separate functions like this:

1. `void print_it(float area)`
2. `int print_it(float area)`

Default Parameters

C++ lets you define functions with default parameters. For example, you could decide that triangles have a default height of 2.0. You can add a small bit of code to the parameter list to specify this:

OBSERVE:

```
float triangle(float width, float height = 2.0) {
```

To see this change in action, edit your program:

Code to Edit: triangle.cpp

```
#include <iostream>

/*
 * triangle -- compute the area of a triangle
 *
 * Parameters
 *     width -- the width of the triangle
 *     height -- the height of the triangle
 *
 * Returns
 *     the area of the triangle
 */
float triangle(float width, float height = 2.0) {
    float area = (width * height) / 2.0;
    return (area);
}

/*
 * print_it -- output a message
 *
 * Parameters
 *     area -- the area of the triangle
 *     what -- the name of the triangle
 */
void print_it(float area, char what[])
{
    std::cout << "The area of the " << what << " triangle is " << area << std::endl;

    return;
}

void print_it(float area)
{
    std::cout << "The area of the triangle is " << area << std::endl;

    return;
}

int main()
{
    float area; // Area of a triangle

    area = triangle(5.0, 2.0);
    print_it(area);

    area = triangle(3.8, 5.9);
    print_it(area, "second");

    area = triangle(2.8, 1.6);
    print_it(area, "third");

    return (0);
}
```



Save and run it. You'll see the same results as before:

OBSERVE:

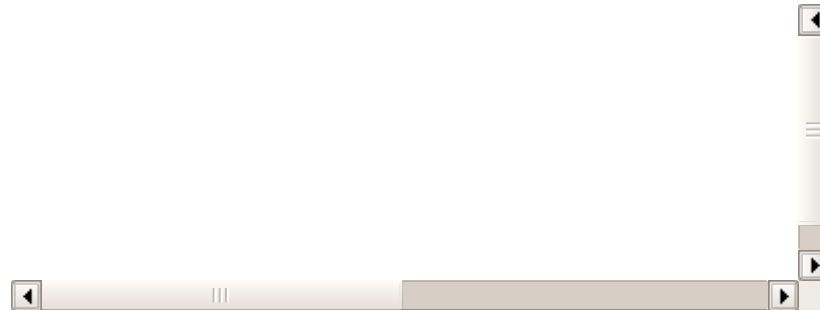
The area of the first triangle is 5
The area of the second triangle is 11.21
The area of the third triangle is 2.24

If we DO provide a height, the function uses it; otherwise, it uses the default 2.0.

WARNING

Using default parameters hides information and can make programs confusing. While you may encounter default parameters in existing programs, you would be wise to avoid using them in new programs.

We covered a lot of information in this lesson! In the next lesson, we'll continue our discussion of parameters and types. Stay tuned!



Copyright © 1998-2013 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Parameters and Return Types

C++ 1: Introduction to C++ Lesson 12

Welcome back! In the last lesson we discussed functions, and learned how to write our own functions with parameters. In this lesson, we'll explain just how parameters work.

Passing Parameters

C++ provides many ways of passing parameters to functions. Knowing what choices you have and how to use them is key to writing good C++ code.

Pass by Value

Let's start with a short program. Create a project named **var-pass** and assign it to your **C++1_Lessons** working set. In the new project, create a program named **var-pass.cpp** as shown:

Code to Type: var-pass.cpp

```
#include <iostream>

void test(int a)
{
    std::cout << "test: a is " << a << std::endl;
    a = 5;
    std::cout << "test: a is " << a << std::endl;
}

int main()
{
    int i;    // Variable to play around with

    i = 2;
    test(i);
    std::cout << "main: i is " << i << std::endl;
    return(0);
}
```



Save and run the program, and observe the output:

Output of var-pass

```
test: a is 2
test: a is 5
main: i is 2
```

We passed **i** as a parameter to **test()**, where it was labeled **a**. We changed the value of **a** inside **test()**, but that change did not make its way back to **i**. Why did this happen?

It happened because the parameters to the **test()** function were *passed by value*. Internally, C++ copies the values of any variables passed by value and passes the copy to the function. Any changes to this copy will not be reflected in the calling procedure. This is the default way C++ passes parameters to functions.

Remember when we discussed how variables were like mailboxes? Passing by value is like making photocopies of your mail, then putting the copies in the mailbox (the function), keeping the originals safe.

C++ lets you **pass by reference** as well, using the reference (**&**) operator. Edit your program as shown:

Code to Edit: var-pass.cpp


```
#include <iostream>

void test(int& b)
{
    std::cout << "test: b is " << b << std::endl;
    b = 6;
    std::cout << "test: b is " << b << std::endl;
}

int main()
{
    int i;    // Variable to play around with

    i = 2;
    test(i);
    std::cout << "main: i is " << i << std::endl;

    return(0);
}
```

 Save and run it. You see:

Output of var-pass

```
test: b is 2
test: b is 6
main: i is 6
```

In this program, **i** passed by reference, so the **test()** function changed its value: **main: i is 6!**

With pass by reference, C++ makes the parameter **b** a reference to main's local variable **i**. A reference means that these variables are the same thing. As a result, any change to **b** is a change to **i**.

Using the mailbox analogy again, passing by reference is like putting the original letter in the mailbox, so the recipient (the function) can look at and potentially change it.

Array Parameters

Now let's see how functions work with arrays. We'll add an array parameter to our program.


Code to Edit: var-pass.cpp

```
#include <iostream>

void test(int k[])
{
    std::cout << "test: k[0] is " << k[0] << std::endl;
    k[0] = 7;
    std::cout << "test: k[0] is " << k[0] << std::endl;
}

int main()
{
    int k[3] = {10, 20, 30}; // Array to play around with

    test(k);
    std::cout << "main: k[0] is " << k[0] << std::endl;
    return(0);
}
```

 Save and run it. You'll see:

OBSERVE:

```
test: k[0] is 10
test: k[0] is 7
main: k[0] is 7
```

Notice that the value of **k[0]** is changed, even though we didn't use the **&** reference operator. That's because arrays are automatically passed by reference. In fact, there is no way to make them pass by value.

What happens if you pass **k[0]** to a function instead? Let's try it:

Code to Edit: var-pass.cpp

```
#include <iostream>

void test(int a)
{
    std::cout << "test: a is " << a << std::endl;
    a = 5;
    std::cout << "test: a is " << a << std::endl;
}

int main()
{
    int k[3] = {10, 20, 30}; // Array to play around with

    test(k[0]);
    std::cout << "main: k[0] is " << k[0] << std::endl;
    return(0);
}
```



Save and run it. You'll see:

OBSERVE:

```
test: a is 10
test: a is 5
main: k[0] is 10
```

A single array element is just like a single variable—so by default, C++ passes by value.

Const Parameters

Let's now produce a program with a function that returns the maximum of two integers. Start a new project named **max** and assign it to your **C++1_Lessons** working set. Create a source file named **max.cpp** as shown:


Code to Type: max.cpp

```
#include <iostream>

// Function comments
int max(int i1, int i2)
{
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    return (0);
}
```

 Save and run it. You'll see:

OBSERVE:

Max is 2

Notice that the values of **i1** and **i2** are never changed in the **max()** function. Given the nature of the function, they never should be changed.

But what if someone decided to change the parameter? Edit the program as shown:


Code to Edit: max.cpp

```
#include <iostream>

// Function comments
int max(int i1, int i2)
{
    i1 = 99;
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    return (0);
}
```

 Save and run it. You'll see:

OBSERVE:

Max is 99

The function should not change the value of the parameters! To help ensure that the parameters don't change, put the modifier **const** in front of them. Edit your program as shown:


Code to Edit: max.cpp

```
#include <iostream>

int max(const int i1, const int i2)
{
    i1 = 99;
    if (i1 > i2)
        return (i1);
    return(i2);
}

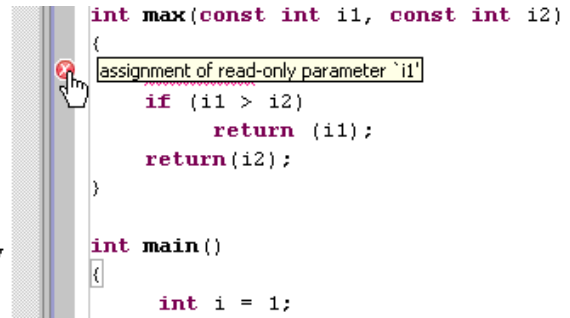
int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    return (0);
}
```

 Save your program—you'll see this error:

```
**** Build of configuration Debug for project
var-pass ****

**** Internal Builder is used for build
****
g++ -O0 -g3 -Wall -c -fmessage-length=0
-o var-pass.o ..\var-pass.cpp
..\var-pass.cpp: In function 'int max(int, int)':
..\var-pass.cpp:6: error: assignment of read-only
parameter 'i1'
Build error occurred, build is stopped
```



Code that attempts to change the value of a constant parameter results in a compile-time error. In this case, the code **i1 = 99** is the offender.

References

References let different variables point to the same underlying value. It is like having a two-sided mailbox—the postal worker opens the box on one side to deposit mail, and you open the box on the other side to retrieve it. Both doors point to the same contents. Let's take a deeper look at references. Change your code as shown:


Code to Edit: max.cpp

```
#include <iostream>

int& max(const int& i1, const int& i2)
{
    i1 = 99;
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    return (0);
}
```

 Save and run it—you'll see the original message again.

Now, let's add a new variable (**l**), which is a reference to **j**. We'll then zero this reference, which should cause **i** to be zeroed. Change the code as shown:

Code to Edit: max.cpp


```
#include <iostream>

int& max(int& i1, int& i2)
{
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    {
        int& l = j; // l now refers to j

        l = 0; // Since l is j, j is now zero
        std::cout << "j #1 is " << j << std::endl;
    }
    std::cout << "j #2 is " << j << std::endl;
    return (0);
}
```

 Save and run it. Sure enough, **l** and **i** both point to the same place, and **i** ends up being set to 0:

OBSERVE:

```
Max is 2
j #1 is 0
j #2 is 0
```

This is because our latest and greatest **max()** function returns a reference to **i**. Because it's a reference, we can use it on the *left* side of the equals sign (=) as well as the right.

Make the changes indicated in the program to remove the reference **k** and replace it with the reference **max(i, j)**.

Code to Edit: max.cpp

```
#include <iostream>

int& max(int& i1, int& i2)
{
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    {
        max(i, j) = 0;    // Assigning a reference
        i = 0; // Since i is j, j is now zero
        std::cout << "j #1 is " << j << std::endl;
    }
    std::cout << "j #2 is " << j << std::endl;

    return (0);
}
```



Save and run it. You'll see:

OBSERVE:

```
Max is 2
j #1 is 0
j #2 is 0
```

Sure enough, the reference returned by **max(i,j)**, which pointed to the same place as j, was changed to zero.

Const Return Values

In the last example, we could assign a value to the reference returned by **max(i,j)**. To keep this from happening, let's change the function so it returns a **const** reference.

Code to Edit: max.cpp

```
#include <iostream>

const int& max(int& i1, int& i2)
{
    if (i1 > i2)
        return (i1);
    return(i2);
}

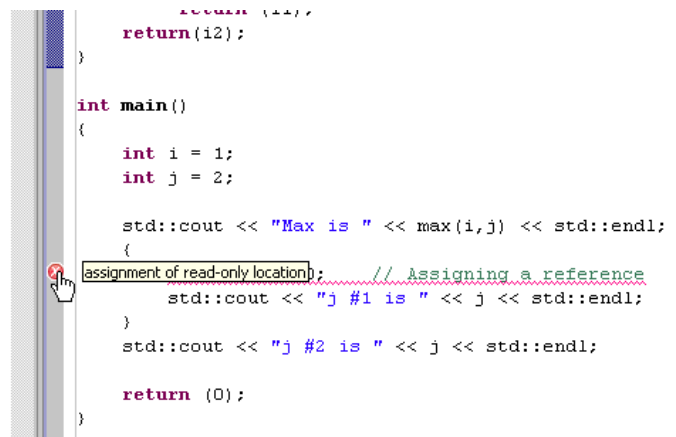
int main()
{
    int i = 1;
    int j = 2;

    std::cout << "Max is " << max(i,j) << std::endl;
    {
        max(i, j) = 0;    // Assigning a reference
        std::cout << "j #1 is " << j << std::endl;
    }
    std::cout << "j #2 is " << j << std::endl;

    return (0);
}
```

Once you save your file you will notice that we can no longer assign a value to **max()**. Because it is a constant, we can only retrieve the result, not change it:

```
**** Internal Builder is used for build
****
g++ -O0 -g3 -Wall -c -fmessage-length=0
-o var-pass.o ..\var-pass.cpp
..\var-pass.cpp: In function 'int main()':
..\var-pass.cpp:17: error: assignment of
read-only location
Build error occurred, build is stopped
Time consumed: 688 ms.
```



Problems with Reference Returns

Returning a reference as an **int** has its benefits, but it can be very tricky to use. Let's change our code so that **max()** assigns the result to a local variable and returns that variable instead.

Code to Edit: max.cpp

```
#include <iostream>

const int& max(int& i1, int& i2)
{
    int result; // Which one to use
    if (i1 > i2)
        result = i1;
    else
        result = i2;

    // This is a bad thing to do
    return(result);
}

int main()
{
    int i = 1;
    int j = 2;

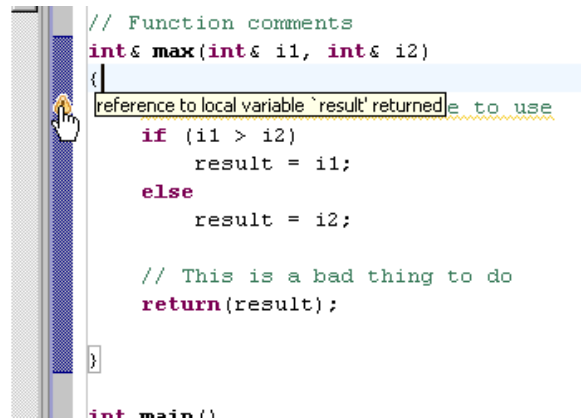
    std::cout << "Max is " << max(i,j) << std::endl;
    {
        max(i, j) = 0; // Assigning a reference
        std::cout << "i is " << i << std::endl;
    }
    return (0);
}
```



Save it—you'll see a warning:

```
**** Build of configuration Debug for project
var-pass ****

**** Internal Builder is used for build
****
g++ -O0 -g3 -Wall -c -fmessage-length=0
-o var-pass.o ..\var-pass.cpp
..\var-pass.cpp: In function 'int& max(int&,
int&)':
..\var-pass.cpp:7: warning: reference to local
variable 'result' returned
g++ -o var-pass.exe var-pass.o
Build complete for project var-pass
Time consumed: 1282 ms.
```



The compiler warns you that you are returning a reference to a *local variable*. This is reminding you that the variable **result** will be destroyed when **max()** returns. This is like removing your mailbox from the wall—the postal worker can't put anything into the box, and you can't either.

So... what does the reference generated by the **return** statement refer to? *Absolutely nothing*. The value referenced is not in scope, so it is not legal to reference it. This is yet another area of undefined behavior in C++. Your program might work just fine, or it could crash, or it could cause strange and mysterious data errors.

When a reference refers to something that is no longer in scope, it's called a *dangling reference*. **You can't (legally) return a reference to a local variable from inside a function.**

Now let's try another experiment with our program. Edit the program so that it uses expressions as parameters to **max()**:

Code to Edit: max.cpp

```
#include <iostream>

const int& max(int& i1, int& i2)
{
    int result; // Which one to use
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    int answer;
    answer = max(i+1, j+1);

    std::cout << "Max is " << answer << std::endl;
    return (0);
}
```

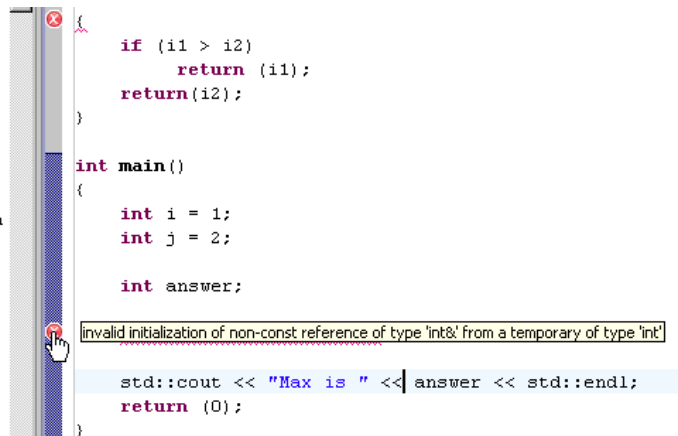
This program too has a dangling reference that is hard to spot, but the compiler won't let it by.



Save it and you'll see:

```
**** Build of configuration Debug for project
var-pass ****

**** Internal Builder is used for build
****
g++ -O0 -g3 -Wall -c -fmessage-length=0
-o var-pass.o ..\var-pass.cpp
..\var-pass.cpp: In function 'int main()':
..\var-pass.cpp:17: error: invalid initialization
of non-const reference of type 'int&' from a
temporary of type 'int'
..\var-pass.cpp:4: error: in passing argument 1
of 'int& max(int&, int&)'
Build error occurred, build is stopped
Time consumed: 609 ms.
```



When a function expects a reference parameter such as **max()**, the compiler performs a number of operations behind the scenes:

1. It creates a temporary variable and assigns it the value of the expression.
2. It performs the function call.
3. It destroys the temporary variable.

So the code the compiler generates looks something like:

The compiler's version of max.cpp

```
#include <iostream>

int& max(int& i1, int& i2)
{
    if (i1 > i2)
        return (i1);
    return(i2);
}

int main()
{
    int i = 1;
    int j = 2;

    int answer;

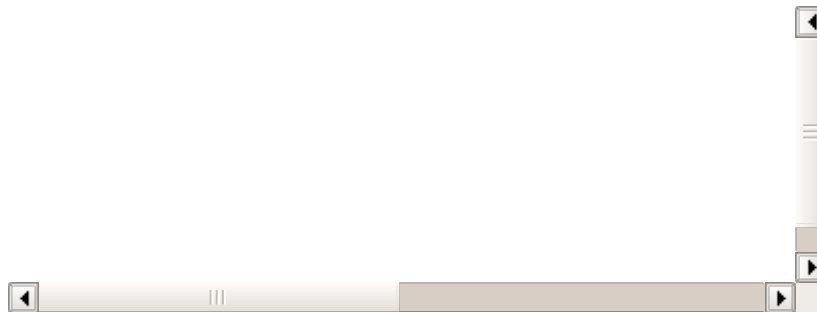
    {
        int tmp1 = i + 1;
        int tmp2 = j + 1;

        answer = max(tmp1, tmp2);
        // At this point answer is a reference to tmp2
    }
    // At this point tmp2 does not exist

    std::cout << "Max is " << answer << std::endl;
    return (0);
}
```

You will rarely need to use references to variables as return values, but when you do need to use them, be very careful about what you are doing.

You made it! In the next lesson we'll discuss your final project for the course. Good luck!



Copyright © 1998-2013 O'Reilly Media, Inc.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Final Project

C++ 1: Introduction to C++ Lesson 13

Putting It All Together

At this point we've learned enough to create your very own program from scratch.

Assignment

To start, create a project named **final_project** assigned to your **C++1_Homework** working set, with a source file named **final_project.cpp**.

Your assignment is to create a program that will count the number of words in a file. It is up to you to define what a "word" is. "hello" is a word. But what about "high-five"—is that one word or two? And how about something like "ground_point"? Some programmers might consider that one word and others think it was two. Did you consider something like "O'Reilly"?

Whatever you decide to consider a word, document your decision in a file named **requirements.txt**.

The specification is as exact a description as you can get of what the program is going to do. Sometimes you don't know how a program is going to end up until you start coding. In that case, you start with a preliminary specification and refine it as you go on.

At the end of coding, the specification should be detailed enough that you can use it as a user guide.

Create a file named **spec.txt** and write your specification in it.

Code Design

The design is a general outline of how you are going to create your program.

The design for this word count program should be fairly short. A good paragraph describing the general operation of the code should do it. Create a file named **design.txt** and enter your design.

Agile Development

Years ago there was programming technique called "fast prototyping." Today, it has been renamed "agile development." Basically, it means that you create the smallest working program you can. Test it, enhance it, and repeat until you get what you want. The idea is that you start with a working copy and build a working program on top of that. That way, you have a working program to see where you are and to give you an idea of what the computer can do. This lets you refine your specification as you learn more about what's happening with your system.

In this case, there are three stages that we go through in our development:

1. Make a program that reads a character at a time, counting nothing.
2. Make the program count characters.
3. Change the program to count words.

Coding Notes

To help you get started, the following is code that will read each character from a file called **input.txt** and will output it. It uses **fstream**.

Code to Type: final_project.cpp

```
#include <iostream>
#include <fstream>

int main () {
    char c;
    std::ifstream myfile ("input.txt");

    // Make sure the file can be read from
    if (myfile.is_open())
    {
        // While we have not reached the end of file (EOF)
        while (! myfile.eof() )
        {
            c = myfile.get();
            std::cout << c;
        }
        myfile.close();
    }

    else std::cout << "Unable to open file";

    return 0;
}
```

Create a file named **input.txt** in your project, add some appropriate text to it, and then save and run this code to see how it works. Your program will prompt for user input instead of using a file.

Testing

Testing comes next. Write up a test plan for your program named **test.txt**. Take care to test all major components of your program.

I would suggest that you create a file named "input.txt" containing the input that you want to use for testing the program.

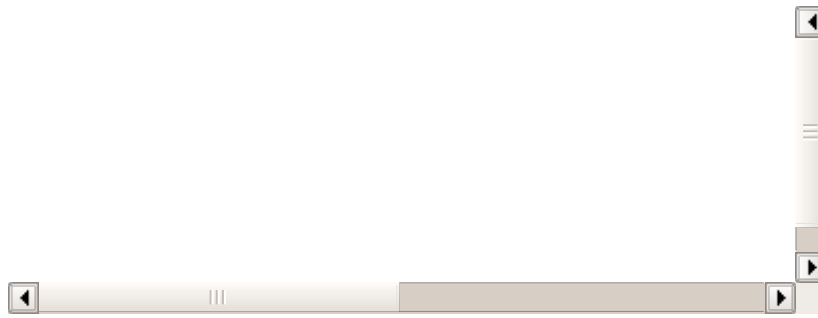
To actually test the program:

1. Open the input.txt file.
2. Select **Edit | Select All**.
3. Select **Edit | Copy**.
4. Run the program.
5. Click in the console and select **Edit | Paste** to paste the test file's contents into your program's input.
6. Type **[Ctrl+Z]** to signal end of file.

Revisions

Your program counts words. Think of how you might want to enhance it. Create a file named "rev.txt" containing the wish list for your new program. Then be thankful that you've reached the end of the lesson and you don't have to implement any of those revisions!

When you finish, hand in your project. GOOD LUCK!



Copyright © 1998-2013 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*