THE GREAT COURSES®

Topic
Science & Mathematics

Subtopic
Technology

Introduction to C++

Programming Concepts and Applications

Professor John Keyser
Texas A&M University

Published by
THE GREAT COURSES

Corporate Headquarters

4840 Westfields Boulevard | Suite 500 | Chantilly, Virginia | 20151-2299

[PHONE] 1.800.832.2412 | [FAX] 703.378.3819 | [WEB] www.thegreatcourses.com

Copyright © The Teaching Company, 2019

Printed in the United States of America

This book is in copyright. All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted, in any form, or by any means (electronic, mechanical, photocopying, recording, or otherwise), without the prior written permission of The Teaching Company.

John Keyser, PhD

Professor of Computer Science and Engineering
Texas A&M University

John Keyser is a Professor of Computer Science and Engineering at Texas A&M University. He earned his PhD in Computer Science from the University of North Carolina. As an undergraduate, he earned 3 bachelor's degrees—in Computer Science, Engineering Physics, and Applied Math—from Abilene Christian University.

Professor Keyser's interests in physics, math, and computing led him to a career in computer graphics, which has allowed him to combine all 3 disciplines. He has published several articles in geometric modeling, particularly looking at ways of quantifying and eliminating uncertainty in geometric calculations. Professor Keyser has been a long-standing member of the solid and physical modeling community, including previously serving on the Solid Modeling Association executive committee. He has also published several articles and coauthored a textbook in physically based simulation for graphics (*Foundations of Physically Based Modeling and Animation*). As a member of the Brain Networks Laboratory collaboration at Texas A&M, he has worked on developing a new technique for rapidly scanning vast amounts of biological data, reconstructing the geometric structures in that data, and helping visualize the results in effective ways. In addition, he has published papers on a variety of other graphics topics, including rendering and modeling.

Professor Keyser's teaching has spanned a range of courses, from introductory undergraduate courses in programming; through upper-level undergraduate courses in graphics, programming, and software development; to graduate courses in graphics modeling and simulation. Among these, he created a course called Programming Studio that has become required for all Computer Science and Computer Engineering majors at Texas A&M, and he helped develop the introductory programming course taught to all Texas A&M Engineering students.

Professor Keyser has won several teaching awards at Texas A&M, including the Distinguished Achievement Award in Teaching, which he received once at the university level and twice from the College of Engineering. As an assistant professor, he was named a Montague Scholar by the Center for Teaching Excellence, and he has received other awards, including the Tenneco Meritorious Teaching Award and the Theta Tau Most Informative Lecturer Award.

Since writing his first computer program more than 35 years ago, Professor Keyser has loved computer programming. He has particularly enjoyed programming competitions, both as a student competitor and as a team coach. Of the many computer science classes he took, the most important class turned out to be the one in which he met his wife. In his free time, he enjoys traveling with her and their 2 daughters.

Professor Keyser's other Great Course is *How to Program: Computer Science Concepts and Python Exercises*. ◆

Table of Contents

NAVIGATION TIP

To go back to the page you came from, press Alt + ← on a PC or ⌘ + ← on a Mac. On a tablet, use the bookmarks panel.

Introduction to C++: Programming Concepts and Applications

As computers become more and more a part of our everyday lives, it is easy to imagine them as controlling our lives. However, programming gives us a chance to truly be in charge of the computers. Programming lets us specify exactly how a computer should operate at the lowest levels. And by putting these low-level commands together, we can get the computer to do complex and interesting things.

This course explores the process of computer programming—specifically using the language C++. It is one of the most powerful programming languages there is, giving programmers the power to control from the lowest levels—specifying individual elements of memory—to the highest levels, where broad concepts are used to manipulate large amounts of data. C++ also supports a variety of programming paradigms, from the traditional imperative and procedural approaches that have long been a part of most languages, to the object-oriented approaches that began to dominate programming in recent decades, to generic approaches that let programmers specify behavior at an even more general level.

This course walks you through the range of C++ programming, providing a tour of all the key aspects of programming in C++. No prior programming knowledge is assumed, and instructions are provided to help novice programmers get everything set up to begin their programming journey.

The beginning lectures start with the basics of programming, describing how variables work, how basic input and output is handled, and how basic computations can be performed (lectures 1 and 2).

You then explore how the flow through a computer program is managed, looking at the key concepts of conditionals—where choices can be made (lecture 3)—and loops, where commands can be repeated (lecture 5). You also learn how to make use of the numerous functions that C++ provides in standard libraries (lecture 6).

Next, the course turns to structures that C++ provides for handling larger amounts of data. You are introduced to 2 methods: the array, a concept carried over from the C language that forms the basis for storing data in a large block of memory (lecture 7); and the vector, a new approach in C++ that improves on the array structure to provide additional safety and functionality (lecture 8). The way vectors are treated in C++ is very similar to the way it handles strings, which are collections of characters used to handle text (lecture 9). For large amounts of data, interactive input and output is less feasible, so you are also introduced to the way that C++ can handle files (lecture 10).

With these fundamental programming concepts established, the course turns to the heart of procedural programming: the use of functions. Functions allow you to conceptually separate your program, making it feasible to develop much larger and more complex programs. You discover the process for writing your own functions (lecture 12) and the different ways you can handle parameters, including the introduction of the idea of a reference (lecture 13).

Following this, the course turns to object-oriented programming (OOP). It's probably the most commonly used programming paradigm today, and C++ provides full support for OOP development. OOP typically involves 3 key ideas, each of which is explored: encapsulation, where similar data and functions are grouped together (lecture 16); inheritance, where variables and functions are shared from one structure to another (lecture 19); and polymorphism, where different structures can be used interchangeably (lecture 20). OOP also brings up the idea of constructors, and operator definitions are also commonly seen in OOP, so both of these ideas are addressed as well (lecture 17). Finally, though it is not an OOP-specific topic, the need to allocate and deallocate memory comes up frequently in the context of OOP, so the idea of dynamic memory allocation is explored (lecture 18).

The last major topic is generic programming, emphasizing the use of the Standard Template Library (STL) in C++. You discover key aspects of the STL and how using it can simplify your programming (lectures 22 and 23).

Throughout the course, several lectures are interspersed that deal with the larger issues of how to move from simply writing lines of code to developing larger programs (lectures 4, 11, 14, 15, 21, and 24). The topics covered include methods of testing, incremental code development, debugging, exceptions, top-down and bottom-up design, and object-oriented design. The power of OOP is used to design a game engine that can play different games (lecture 21). The course concludes with a final lecture showing how C++ can be used to develop algorithms and in particular focuses on the way an artificial intelligence algorithm can be implemented as the opponent in a game (lecture 24). ◆

01 Compiling Your First C++ Program

C++ is one of the most powerful, efficient, and flexible programming languages that exists. It allows you to program in a way that closely corresponds to machine instructions while also providing several higher-level features to make it easier for people to understand. C++ supports multiple programming styles, or paradigms, that have emerged over time, and it is particularly strong in allowing you to use a variety of different paradigms in your programs. As a result, C++ is used as the basis for a wide variety of applications across many domains.

IN THIS LECTURE:

// INTRODUCTION TO COMPUTER PROGRAMMING

Computers operate by following a set of instructions. But the instructions that a computer can understand are a bunch of 1s and 0s—a long sequence of binary numbers that encodes the instructions and data for a computer to use. For people, this is nearly impossible to follow, and this is why programming languages have been developed.

What is sometimes considered the first program was written in the 1840s by Ada Lovelace, who wrote computational instructions that could run, in theory, on a machine that had been designed but not built.

The first modern programming languages were developed in the 1950s, with Fortran being the first widely used one. In fact, Fortran is still used today! Fortran is a procedural language, and **procedural programming** is still a common approach used today, including as a style of C++ programming. Since Fortran, hundreds of languages have been developed, and each one has its own strengths and weaknesses.

C++ has been used to develop many of the programs you've probably experienced—from Microsoft Windows to the code underlying Google's search and to website functionality on YouTube, Facebook, Amazon, and PayPal. It's been used to control devices ranging from ship engines to Mars Rovers and in phone systems worldwide.

In the 1980s, Bjarne Stroustrup wanted to retain the low-level efficiency advantages of C while adding some of the more modern programming approaches. Thus, C++ was born!

C++ was developed as a sort of extension of the programming language C, which itself was a descendent of an even earlier language called B. All of these came out of Bell Labs, later known as AT&T Labs. While B never got much use, C—which was developed in the early 1970s—became very widely used and still is today.

One big advantage of C++ was that it added the ability to do **object-oriented programming**, which lets you group your data and operations together in more useful ways. But C++ is not only an object-oriented language. Other new features have been added, too, allowing it to reflect the most recent advances in programming language design.

This all makes C++ incredibly powerful, letting you work across a full range of programming paradigms. Like C, you can work in just about as low a level of detail as you want, specifying individual bits, or you can work at a much higher level, using cutting-edge programming constructs that let you write code once and apply that to a wide range of applications.

C++ is also a language that continues to be developed today. There's a large and active international C++ standards committee that is continually working to determine ways to improve the language, including adding new features that integrate the best and most recent programming practices.

As you go through this course, you'll find it a much more rewarding and valuable experience if you're able to practice writing your own code. See lecture 01b—C++ QUICK START.

// WHAT HAPPENS WHEN YOU PROGRAM

Programmers generate a program written in a programming language, such as C++. The program is just a text file, and you're free to write it in many different ways. You could use your own word processor if you want and just save it as a text file. You could use Notepad or WordPad on a PC or TextEdit on a Mac. There are also editors written specifically to help people write code, such as Notepad++.

Once you write a program, you send it to a compiler, which takes the instructions written in C++ and translates them into machine instructions—the instructions that the computer can understand. So, a compiled program is a program that's ready to be run, or **executed**, by a machine. This compiled version of the program is called an executable program, and you can then run it on your computer whenever you want.

As people develop code, they'll usually go through this process many times to see if it's working. To make it easier to develop code this way, programmers use an **integrated development environment** (IDE), which will have an editor that lets you write code, usually with some special features to help you write code in that language. The IDE will have an easy way to automatically save that code, have the compiler compile it into machine language, and then run it—often just with one click of a button.

And there are usually other features built into the IDE, such as a **debugger**, which is a tool that helps you find and fix errors in your program. The IDE integrates all these things—and more—into one package so that you can edit and compile and run and debug your code all in one program.

// YOUR FIRST PROGRAM

There's a very long-standing tradition in computer science that the first program you develop—in any language—is a **Hello, World!** program, which is just a program that displays the message ***Hello, World!*** to the user.

Here's what a **Hello, World!** program looks like in C++.

You set up an IDE and type the program into the IDE's editor window. When you tell the IDE to run the program, it will both compile the program and then run the executable that the compiler produced. The result of the program is a line of **output** that **prints** out the words ***Hello, World!***.

```
1  // Program 1_1                          a
2  // Our first program!
3  #include <iostream>                     b
4  using namespace std;
5
6  int main()
7  {                                       c
8      cout << "Hello, World!" << endl;
9  }                                       d
```

A selection of programs will be shown in this guide. To view all programs discussed in this course, go to TheGreatCourses.com/CPlusPlus.

In programming terms, the word ***print*** just means "display" or "show."

The first lines you'll see are comment lines (a). Comments are ways of giving notes about the code, but they do not affect what the code actually does.

The next few lines are header information (b). The **#include** line gives you access to a **stream** of input and output. The **using namespace** line makes it easier for you to write code.

The next line has **int main** and then a pair of parentheses (6). This is the part of the code that marks the beginning of the program—the main part. It lets the computer know that this is where the code you want to run begins.

Next, you'll see a pair of curly braces (d), which group a set, or block, of commands together inside. In this case, the curly braces are grouping together all the commands inside of the **main** part of the program.

Curly braces are a critical part of C++ programming that date all the way back to a predecessor language in the 1960s called BCPL—and before that to the way you write a set in mathematics, such as the set containing the numbers {1,2,3}.

Finally, you have your actual line of code giving a command (8). This is the line of code that tells the computer specifically what you want to do: It commands the output of **Hello, World!**. This statement is important; all the rest of this program is basically the same as in other programs. This one statement is the key line that does something specific.

First, notice that the line begins with the word ***cout***, which is an expression meaning "console output." The **console** is the default output area, so **cout** means that there's going to be output, and the output will be going to the console—it is output to the default output area. In an IDE, the default output area might be an output pane within the main window, or it might pop up a new window to be a console window. If you're running the code in a browser, it'll show the result in that window that you're running from.

Next is what looks like a double less-than sign. This is the output **stream operator** (c). Think of the less-than signs as arrows pointing in the direction that the information stream is flowing. In this case, you have something that you want to print, so that thing you're printing will flow toward the console output. Notice there are 2 stream operators on one line, which shows that different items are being streamed in order: The thing to the left will reach the console output before the thing to the right.

After the stream operator is the text that you actually want to output. Notice that the text you want to see is enclosed in a pair of quotation marks. You have to use quotation marks to identify text in your code; otherwise, the letters will get misinterpreted by the compiler. The words **Hello** and **World**, along with punctuation—a comma and an exclamation point—appear inside the quotation marks.

The next thing to get streamed to the output is the word ***endl***, which is a way of saying "end line" for output. If you don't include **endl**, then the next thing that'll be printed out will come immediately after the previous one on the same line; you include **endl** so that the next thing to be printed will come at the beginning of the next line.

> The **endl** indicates an end of line in what is output, not necessarily the end of a line of code.

Finally, there is a semicolon, which appears at the end of the line of code. In C++, the semicolon indicates that you've reached the end of a line of code.

> Just as a period ends a sentence in ordinary English, a semicolon ends a statement in C++.
>
> Just like a sentence you write can go across more than one line—or you can have 2 sentences on the same written line—a statement in C++ can go across more than one line, or you can have more than one C++ statement in the same written line. It's the semicolon, not just moving to a new line, that ends a statement.

The only times you don't have a semicolon are for a few special cases:

- the **#include** line isn't actually a command you want the computer to perform as part of the program but rather a command to the compiler that translates the code to machine instructions
- the curly braces after **main** are not a command but rather a way of grouping commands together.

Any time you are finished with an individual command, it will need to have a semicolon at the end.

Unlike some other languages, the order and spacing of your C++ code doesn't matter. You can put multiple instructions of code on one line of text with no space needed after the semicolons, or you can spread a single instruction over multiple lines. The compiler basically ignores everything about spacing as it converts the program to machine instructions. Instead, the way you separate one command from another is by using the semicolon to mark the end of a statement.

Comments can be anything that helps people understand what the code is for. Comments are skipped entirely by the compiler; when a compiler sees a double forward slash, it always reads what follows as a comment. While processing code, it skips over it, all the way to the end of the line. So, including comments is purely to help people reading the code.

There's not much need for comments in small pieces of code compared to how important they are in bigger programs. But getting in the habit of including comments to explain code will help whenever you do end up writing somewhat larger programs.

A comment could be for people who didn't write the code but need to come along later and understand it, or it can help you tell your future self what you had in mind for each part of the code.

In C++, there are 2 ways of writing comments. The first is one where you begin the comment with a double forward slash (e), which is a way of indicating that everything from that point on in that one line is a comment.

The other type of comment is one that opens with /* and closes with */ (f). This type of comment is helpful if you want to write multiline comments. It will begin with /* and continue until whenever it's closed by */, even if that's several lines later. You can insert a multiline comment anywhere you wish by using this format. You could also easily modify your original program to make the initial comment use this format.

```
1  // Program 1_9                                          e
2  /************************************/                 f
3  /* Our First Program! */
4  /************************************/
5  #include <iostream>
6  using namespace std;
7
8  /* About to start the main program */                   f
9  int main()
10 {
11    cout << "Hello, World!" << endl; // This is a single line comment   e
12 }
```

You often use comments to provide visual separation—for example, to create a block at the beginning of a program. Notice in this example that you have a few lines of just asterisks, providing a visual block of comments at the beginning of the program.

And you often will use a comment to describe what's coming next. For example, you might put a comment right before the main part of the program, just showing what that part is about to do.

This is a pretty simple program; the real work of the program is all done in just one line.

This slightly more complex program has 3 different lines of code in the main program—the program lines that begin just below the **int main** line. All 3 of these lines are nearly identical; they just print out some text. Each one prints out text on a different line. The first one says, ***Howdy, John!***. The next one says, ***Are you ready to learn C++?***, and the final one says, ***Let's get started...***.

Each output also includes an end-of-line character.

Commands are executed in sequential order. When you have a series of commands, like the 3 **cout** statements you have here, the program will handle each of them in sequence—in the order specified. So, if you consider a program as being a series of commands, then running the program is executing those commands in order. ◆

```
1  // Program 1_10
2  // A longer program, giving a greeting
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     cout << "Howdy, John!" << endl;
8     cout << "Are you ready to learn C++?" << endl;
9     cout << "Let's get started..." << endl;
10 }
```

Exercise

Write a program that will print your name and address the way you would write it on a letter. You can use multiple **cout** statements or try to put it all into one.

Click here to see the solution.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 2 and section 22.2.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 1.1–1.3.

c Ousterhout, *A Philosophy of Software Design*, chaps. 12 and 13 (re: comments).

Exercise Solution

Answers will vary.

Here are 2 possible solutions:

```
1   // Program 1_13
2   // Printing an address
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       cout << "John Keyser" << endl;
8       cout << "123 Any Street" << endl;
9       cout << "Somewhere, TX 77777" << endl;
10  }
```

OR

```
1   // Program 1_14
2   // Printing an address
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       cout << "John Keyser" << endl << "123 Any Street" << endl
8           << "Somewhere, TX 77777" << endl;
9   }
```

Click here to go back to the exercise.

// QUIZ

1 Match the following commands and syntax to their purpose.

a	`<<`	1	Comment for the remainder of a line
b	`//`	2	Comment that could span multiple lines
c	`;`	3	Designates a group of commands to be executed
d	`{ }`	4	Designates that a line of output should end
e	`/* */`	5	Designates the end of a line of code
f	`endl`	6	Designates the beginning of the program that will be executed
g	`cout`	7	Streaming operator to show direction of output
h	`main`	8	Designates an output statement

2 What C++ command(s) would be used within a program (inside the **main** section of the code) to output the following multiline poem?

```
Learning C++,
Syntax, Concepts, and Design
What fun lies ahead!
```

3 There are 8 errors in the following program. Can you find them?

```
1   / Program to debug
2   include<iostream>
3   using std;
4
5   main()
6       cout "Hello, World! << endl
7   }
```

Click here to see the answers.

// QUIZ ANSWERS

1 The matches are as follows:

- a 7
- b 1
- c 5
- d 3
- e 2
- f 4
- g 8
- h 6

2 There are multiple possible answers. Among them are the following:

```
    cout << "Learning C++" << endl << "Syntax, Concepts, and
Design" << endl << "What fun lies ahead!" << endl;
```

OR

```
    cout << "Learning C++" << endl;
    cout << "Syntax, Concepts, and Design" << endl;
    cout << "What fun lies ahead!" << endl;
```

3 Here are the 8 errors in the given code:

1. Missing a `/` in the first line (the comment should begin with `//`, not just `/`)
2. Missing a `#` before `include`
3. Missing the `namespace` between `using` and `std`
4. Missing the `int` before `main()`
5. Missing the opening `{` after `int main()`
6. Missing the `<<` after `cout`
7. Missing the closing `"` after the `World`
8. Missing the `;` at the end of the `cout` statement

Here is a correct version of the code:

```
1  // Program to debug
2  #include<iostream>
3  using namespace std;
4
5  int main() {
6      cout << "Hello, World!" << endl;
7  }
```

Click here to go back to the quiz.

01b

C++ QUICK START: With Browser or Download

IN THIS GUIDE:

// INTRODUCTION

To get the most out of this course, you're going to want to try programming yourself. To do that, you'll need to know what tools to use to write programs, how to save programs, and how to run programs on your machine.

Here are 3 ways you can use free tools to write C++ programs.

The first and easiest way is to use a web browser. You can do this from just about any computer that has a web browser—even a smartphone. Using a web browser has some limitations regardless of your computer, but it's a good option if you have an especially old computer running an old operating system. And regardless of operating system, sometimes it's more convenient.

The other 2 options involve downloading software, called an integrated development environment (IDE), to your computer. Either option includes both C++ and a first-rate set of tools for using C++.

On a PC, the recommended IDE is called Visual Studio. The free version is called the Community edition, and it will have everything you need and much more. Paid versions of Visual Studio are geared toward teams of professional developers, so it's more complicated than you want for this course.

Visual Studio is a great product that can be used by novices or by people developing the most powerful programs that exist. Visual Studio can even be used with a wide variety of other languages, though you're just using it for C++ in this course.

NOTE: The recommended IDEs are Xcode on a Mac or Visual Studio Community on a PC. (If you need to use both a Mac and a PC, you might consider other options, such as "Visual Studio Code.") Both Xcode and Visual Studio Community will require a somewhat recent version of the operating system, and some gigabytes of free disk space on your system. If you have an older computer, or do not wish to install additional software on your system, you can stick with the browser-based option.

If you happen to be using Linux, then the GCC compiler is your best option. This was probably installed automatically with Linux, but if not, you can go to gcc.gnu.org to get it.

On a Mac, the recommended IDE is called Xcode. Apple supports Xcode as an IDE to build apps for Apple products, where the default is a programming language called Swift. However, this same IDE can also develop apps and other programs in C++, and, in fact, it's widely used for C++ programming more generally, too.

C++ has been updated over the years, from C++11 to C++14 and C++17. The newer versions add more features, but for this course, C++11 will be fine.

Once you've got your software environment set up, feel free to explore the interface a bit. It's fine if there are features that don't initially make sense to you; you'll gradually learn more about the various options as you go through the course and as you develop code on your own during and after the course.

```
1  // Basic Hello, World! Program
2  #include<iostream>
3  using namespace std;
4
5  int main() {
6     cout << "Hello, World!" << endl;
7  }
```

This is a basic **Hello, World!** program. It's just a few lines of code, from the first 3 lines that begin the program, to the **cout** line that actually instructs the computer to print ***Hello, World!***, and ending with a final curly brace.

// QUICK START WITH A BROWSER

The easiest way to get started with C++ is to open a web browser on your computer. Any browser should be fine.

From your browser, you will be choosing a website that offers an online editor for C++ that is interactive.

Some of the websites that work well are shown at right, but a web search for "Online C++ Compiler" or "Interactive C++ Compiler" will turn up some others that should work fine, too.

WEBSITES WITH INTERACTIVE ONLINE EDITORS

http://cpp.sh/

https://www.onlinegdb.com/online_c++_compiler

https://repl.it/site/languages/cpp

https://www.jdoodle.com/online-compiler-c++

There's one pitfall to watch for: **Use only sites that support interactive mode**. If you're going to use a browser-based compiler that's not on the list, make sure that it supports interactive mode when executing a program.

How can you tell?

Here is some code that's interactive. It's a version of **`Hello, World!`** that also takes input. If you run this in the browser, an interactive mode will prompt you for your name, which you can then type in.

But if you only see a box, often labeled **`stdin`** (short for *standard input*), where you type input separate from where you see output, that is probably not a good sign.

On a site that's not interactive, you would have to enter all your input beforehand: It would not be possible for a user to reply to a question and have the program compute a result.

You should be able to run the program and then interact with the output window—to type in data when prompted.

Feel free to select whichever site you find easy and intuitive to use, but note that cpp.sh is especially simple and clean.

OnlineGDB is more powerful; it offers an online debugger, which many interactive sites lack.

```
1  // Interactive Hello, World! Program
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8     string username;
9     cout << "What's your name? ";
10    cin >> username;
11    cout << "Howdy, " << username << "!" << endl;
12 }
```

The steps are simple:

1. **Type a web address in your browser**.
2. **Select C++ as the language**. Some online websites handle other languages, and sometimes a different language is selected by default. You may need to select C++ from a dropdown box or open a new tab for a C++ program.
3. **Save your code in a text file**. Your code is not being saved by typical online websites, so you should expect to write and save your code in a text file and paste it into the browser. A few sites may offer you an account; such sites might let you save your code there.

Regardless of which browser-based compiler you choose, when you bring up the webpage, the first thing that you're likely to see is a box with an example program already in it. That's the main window, where you'll enter your program. There might be a variation on your **`Hello, World!`** program already there.

Start by deleting what's in the main window. Then, enter your **`Hello, World!`** program. Because the browser won't save your program, type this code into a text file and then copy and paste it into the main window, where the old program used to be.

You may notice that the code is automatically assigned several different colors for you, though some colors may be less obvious when the font size is small. These colors make it easier to see the structure of your program. The specific colors will vary from site to site, but the color coding helps highlight different pieces of your program as you write the code.

Once you have entered your program into the system, in order to see the results, press the Run, or Execute, button, located somewhere above or below the main box.

When you run the program, the output window will appear, if it wasn't there already. On some websites, this output window will be visible all the time, but in others, it only appears after running a program. This output window might be labeled Execution, or Console, or Result, or even Input.

If you typed everything correctly, the output window should show **`Hello, World!`**. If you made a mistake typing in the program, then you might see an error. It could be in the same output window, or it might be in a different window or a different tab of an existing window.

Once you get a correct output, try creating an error in the code—for example, by removing the last character from the second line (**`>`**). If you try running this, you should see a compilation error message somewhere. You may need to look at the Compilation tab if it is separate from an Execution tab.

The error will list something like **`2:18: error: missing terminating > character`**. The 2:18 is telling you that there was an error detected on line 2 in the 18th character—that's where the character was removed. The message **`missing terminating > character`** tells you that you're missing the closing angle bracket. The error message should give you clues about exactly what prevented your program from running successfully. Different websites' compilers may have different error messages.

```
1  // Hello, World! program with error
2  #include<iostream
3  using namespace std;
4
5  int main() {
6      cout << "Hello, World!" << endl;
7  }
```

Throughout the course, try writing the code for yourself and maybe even making small modifications to it as you feel comfortable. Implementing and experimenting on your own is one of the best ways you'll learn the details of how to code in C++.

There are a few limitations of programming in a browser:

- **There's no guarantee of reliability**: The website could go down briefly at any time.
- **Some browser compilers will not let you create and store your files on their computer.** On those websites, reading and writing files would not be possible.
- **There's no security or privacy.** The browser shell is not running on your machine; it's compiling and running on some other machine. You should not be entering sensitive or proprietary information into this system.

Still, a web interface is probably the simplest way to quickly try out a program and see the results. The browser is an easy way for you to do 90% of what's in this course.

Remember that any program in the browser window is not being saved anywhere, so it's up to you to make sure your code is not lost.

Whenever you have programs you type into the browser, you should also copy and paste the program into a document on your computer.

Saving in plain text is best, because that won't introduce extraneous information and formatting that a word processor might add in. If you save your file with a *.cpp* extension at the end, then that's a signal that it's C++ code.

// QUICK START WITH AN IDE

We've provided supplemental documentation at TheGreatCourses.com/CPlusPlus with even more detailed steps and tips for downloading and installing. You may be fine with the information provided here, but if you do run into trouble, please see that supplemental information for additional help.

When you're ready to move beyond browser-based programming, using an integrated development environment (IDE) will provide greater capabilities but takes a little more to set up. There are 2 main steps to follow:

1. Download and install the IDE.
2. Write and run your programs.

To use an IDE, you'll need to first download and install the IDE on your computer.

On a PC, this means going to the Visual Studio Community homepage. Click the button to start the download. This will download the installer, which you then need to run to actually perform the installation.

To download on a Mac, find Xcode in the App Store and install. When you install, you'll need to follow particular installation instructions. If given the option, specify that you want to use C++.

For the most part, just let things install in their default ways and default places. In most cases, the installation will go smoothly.

CONTINUED ON PAGE 16

WINDOWS PC with Visual Studio Community

1 START A PROJECT

a Select "Start a New Project" from the first window
OR From within Visual Studio, go to File → New → Project.

b Select Empty Project.

c Enter a Name for the project; then click Create.

2 CREATE A C++ FILE

a Go to Project → Add New Item.

b Select "C++ file" (with a .cpp extension).

c Enter a Name for the file (e.g., "HelloWorld.cpp"); then click Add.

3 WRITE YOUR PROGRAM

a Enter your **`Hello, World!`** program into the window.

4 BUILD AND RUN THE PROGRAM

a Go to Debug → "Start debugging"
OR Go to Debug → "Start without debugging"
OR Hit the F5 key
OR Click the green arrow titled Local Windows Debugger.

5 OUTPUT SHOULD APPEAR

a A window will pop up, showing the ***`Hello, World!`*** output. You can dismiss this window by pressing any key inside it.

NOTE: Visual Studio 2017 (used during the development of this course) required users to include **`system("pause");`** at the end of the code in order to keep the output window open. As of Visual Studio 2019, that line of code is no longer necessary. If you're running any earlier version of Visual Studio, put **`system("pause");`** right before the final closing curly brace in every program you write to make the output window stay up.

MAC with Xcode

1 START A PROJECT

a Select "Create a new Xcode project".

b Pick Mac OS X from the ribbon at the top of the "Choose template" box.

c Select Command Line Tool and press Next.

d Enter a name for the project (such as "HelloWorldProject").

e **IMPORTANT**: Select C++ as the type (the default is Swift).

f Choose an Organization Identifier (such as your initials); then press Next.

g Optionally, create a folder for your programs (such as "C++ project folder"); then press Create.

2 CREATE A C++ FILE

a Look at the left panel for the project folder and click on "main.cpp".

b A window will pop up with a default **`Hello, World!`** program in it.

3 WRITE YOUR PROGRAM

a Delete code displayed by default.

b Type your own **`Hello, World!`** program.

4 BUILD AND RUN THE PROGRAM

a Click on the arrow at the top left of the window
OR Go to Product → Build; then go to Product → Run.

5 OUTPUT SHOULD APPEAR

a An All Output area should appear in the lower right of the Xcode Window.

b This should have the output results: **`Hello, World!`**.

CONTINUED FROM PAGE 14

Once you have your IDE installed, it's time to try writing and running a program in the IDE. Find the IDE you installed on your system and start it up.

When you start the IDE, the first thing you'll need to do is to create a new project. The project will hold all of the C++ files that you are creating to generate your program. You can create new projects for different programs throughout this course.

Setting up a new project is a little different in Visual Studio and in Xcode.

- In Visual Studio, you will create a new project, selecting Empty Project when you have a chance.
- On a Mac, you'll need to select that it's for your operating system (Mac OS X), that you want a command-line tool, and that you want a C++ project.

In both IDEs, you'll need to pick a name for your project; make it something descriptive, such as "GreatCoursesLecture1".

Once a project is created, you'll need to have an actual C++ file that will hold your program.

In Visual Studio, you'll need to "add a new item" to your project, and that new item will be a C++ file. You get to choose the name of the C++ file.

In Xcode, the file will be provided automatically and will have the name "main.cpp"; you just find the file listed under the project folder and click on it.

Once that's done, you'll be able to enter your **`Hello, World!`** program, or whatever code you want to write. When your code is entered, it's time to compile and run.

In Visual Studio, you do this by selecting Start Debugging or Start Without Debugging from the Debugger menu. After the first time running, you'll also just be able to click the green arrow labeled Local Windows Debugger.

In Xcode, you'll just hit the Build and Run arrow near the top left.

When this happens, you should get the **`Hello, World!`** output to a window that pops up. In the Mac version, that window will appear at the bottom right of the Xcode window.

In Visual Studio, a new window will pop up, but it will disappear as soon as the program is over. To keep that window open so that you can actually see what's output, you'll want to add one small line to your code: **`system ("pause");`**. Be sure to add it as the last line before the final curly brace. This system-pause command basically says "pause the system until the user hits a key" and will keep the window open for you so that you can see the output.

Just to see what happens, you might want to also make an intentional error in your code. For example, leave the **`c`** off of the word **`cout`**. That should cause a compiler error, and you will see how your IDE identifies and reports the error to you.

Now you're ready to develop your own programs in your own IDE. ◆

02

Variables, Computations, and Input in C++

All of a program's operations basically come down to a few things: input, variables, computations, and output. You're going to learn how to write programs that touch on all of these aspects of what makes a program. And each of these fundamental tasks of a program corresponds to the main hardware components of a computer. The input and output are handled by an interface with the world. Variables are handled in memory. And computations are handled by the processor, the part of the computer that performs calculations.

IN THIS LECTURE:

// VARIABLES AND COMPUTATIONS

You've already learned how to write programs that create output. This was the console output command, **cout**, which made the computer output text to the screen.

Let's start with a program for counting calories **(Program 2_1 on page 18)**. It involves not only output, but also variables assigned to memory and calculations involving the processor.

The comment at the top of this program acts like a title, telling you that the program is **to compute calories using carbs, protein, and fat** (a).

The program starts with the same (enabling) lines. The first 2 lines help you get access to the commands you need, and the **int main** line indicates where the program should start (b).

```
1  // Program 2_1
2  // Program to compute calories using carbs, protein, and fat
3  #include<iostream>
4  using namespace std;
5
6  int main() {
7      int carb_grams;
8      int protein_grams;
9      int fat_grams;
10     carb_grams = 30;
11     protein_grams = 5;
12     fat_grams = 15;
13     int calories;
14     calories = 4 * carb_grams + 4 * protein_grams + 9 * fat_grams;
15     cout << "There are " << calories << " calories in this dish" << endl;
16 }
```

a b c d e f

The main body of the program has 4 parts:

1. variable declarations, where you give names to variables (c);
2. variable assignments, where you give values to those variables (d);
3. a single computation (to calculate calories) (e); and
4. output (f).

// VARIABLE DECLARATIONS

A computer's main working memory is made up of many locations, each of which is capable of storing just a 1 or a 0. However, programmers don't write in machine language.

So, programming languages let you take a section of that memory, of 1s and 0s, and use it for your own purposes. This section of memory is called a **variable**. Think of a variable as a box that can hold different values—that is, it can vary in what it holds.

In C++, in order to use a variable, you first have to declare it. In other words, you have to write in your program an instruction to the computer that says, "I want you to set aside a box of memory for me to use as a variable." That operation—saying that you want a particular variable—is called the **variable declaration**.

Every variable declaration needs 2 things: a **type** and a name. For example, you could declare an integer-type variable that you name **year**. The code for that would be `int year`, where `int` is short for integer—any counting number, such as 0, 1, 2, or even –10.

In the calorie-counting program, the line `int carb_grams` (7) gives the 2 pieces of information you need to declare a variable: the type of variable (`int`) and the variable name (`carb_grams`).

RULES FOR NAMING VARIABLES

- Names have to begin with a letter. Besides letters, it's also possible to start with the underscore character, but you should not do this, because it is usually reserved for special things in the compiler that come up in more advanced systems programming.
- As for the rest of the name, you can use letters, underscore characters, or numbers.
- There are a few dozen special words in C++, often called **keywords** or reserved words, that are already used for other things, so you can't use those. For example, you cannot name a variable `namespace` or `using`. There are also a few predefined identifiers, such as `main`, that would be possible to declare but that you should just avoid in your own variable names.

The **variable name** is the way you'll refer to the box of memory. This variable name is often called the identifier, because it's the way you identify which box of memory you are referring to. When you refer to `carb_grams` in your program, you'll be referring to that box of memory—and, more specifically, to the value contained in that box of memory.

Likewise, the next line of code (8) is another variable declaration. You're telling the computer: "Set aside another box of integer-type memory called `protein_grams`."

Every variable declaration sets aside a box of memory, and that box will have some value inside of it. But setting aside a box of memory that's just a bunch of 1s and 0s would be a pain to deal with—if you had to write in the 1s and 0s of binary yourself.

The way you get the ability to write variables in a form you understand is with the variable type. When you declare a variable, you first tell the computer what the type of that variable will be.

In the example program **(Program 2_1 on page 18)**, the type is `int`, which means that the programmer can think of the box as containing an integer. The computer will handle the process of converting that integer to and from the binary form. As a programmer, you just say that you want a box of memory and declare the type of data that will be in it.

Each variable declaration will have 2 parts: a variable type, followed by a variable name. In your code, there are 4 variables being declared. Each one is of type integer, and they have different names: **carb_grams**, **protein_grams**, **fat_grams**, and **calories** (which is the variable you're looking to calculate).

To avoid typing **int** over and over, you can put all of your variable declarations on one line. Just begin by telling the computer the type for all the variables on that line: **int**.

Then, you can just list each of the variables you want to declare, separated by commas.

As long as you follow the rules, choosing the names for variables is up to you. But choosing good variable names is one of the most important things you can do to make your code more understandable—to yourself and to others.

So, as you choose variable names, choose names that give a clear sense of the meaning of what that variable is intended to hold. Don't choose names that are too short or too long. Also, try to be consistent in your names: For example, if you named a variable **carb_grams**, you should pick a name like **fat_grams** rather than one like **grams_of_fat**.

What if you wanted to get more precise and use decimals for the number of grams or calories in your program?

A decimal is called a floating point because the location of the decimal floats around: Compare the decimal in 1.1 with 99.999.

To designate a variable as a floating-point type, you use **float**.

So, you'd just go through your program and replace every **int** with a **float** (g).

Once you've declared variables, whether as integers or floats, you need to be able to use them—to actually store information in that box of memory and look at it later.

```
1  // Program 2_3
2  // Using floating-point
   variables
3  #include<iostream>
4  using namespace std;
5
6  int main() {
7      float carb_grams;
8      float protein_grams;
9      float fat_grams;
10     carb_grams = 30;
g
11     protein_grams = 5;
12     fat_grams = 15;
13     float calories;
14     calories = 4 * carb_grams
   + 4 * protein_grams + 9 *
   fat_grams;
15     cout << "There are "
   << calories << " calories in
   this dish" << endl;
16 }
```

Exercise 1

Which of the following would be OK to use as variable names?

tank_capacity
namespace
something
length2
left foot
account
Success!
i
2nd_account

Click here to see the solution.

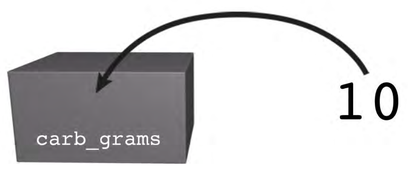
// VARIABLE ASSIGNMENTS

Let's go back to the original program for counting calories **(Program 2_1 on page 18)**. After your declarations of type and name, you can see that there are 3 variable assignments (d). Each of these lines has the same form: a variable name on the left, an equal sign, and a value on the right.

This is the general form for any assignment statement. When this code is executed, the value on the right side is placed into the variable on the left side—it's placed into the box. It replaces any other value already in that box.

```
Variable Assignment:
    <variable name> = <value>

    carb_grams = 10;
```



So, if you have a line like **carb_grams = 10**, that means that the box of memory that you call the variable **carb_grams** will have the value **10** assigned inside of it.

Let's see how the first few lines of the code work in memory.

> Assignment is written with a single equal sign, but it does not mean that 2 things are set equal to each other. It means that the thing on the right is assigned to the variable on the left. That's why it's better to say "gets" or "is assigned" instead of "equals" when you read a line of code out loud.

First, you have a variable declaration, declaring a box of memory for an integer variable named **carb_grams**.

Next, the same thing happens when you declare an integer variable for **protein_grams**.

And you declare an integer variable for **fat_grams**.

So, at this point, you have 3 different integer areas of memory with 3 different names.

Next, you assign the value 30 to **carb_grams**. This means that the box of memory for **carb_grams** now contains the number 30.

Then, you assign 5 to **protein_grams**. So, that variable now holds the value 5.

Finally, **fat_grams** gets the value 15.

When you first declare a variable, it's a good idea to set an initial value to be in the variable. It turns out that this is easy to do: You can combine the variable declaration with an assignment.

```
int carb_grams = 30;
int protein_grams = 5;
int fat_grams = 15;
```

Variables wouldn't do you much good if you couldn't ever use the value they hold. When you see a variable name on the right side of the assignment operator, you can think of what's there as meaning "the value this variable holds."

> Remember that variables can vary in value; they aren't stuck with the first value they are assigned. This is why the single equals symbol means assignment, not permanent equality. At any later time, you can assign a new value in the box, replacing the old one.

// COMPUTING CALORIES

Let's look at the computation for calories, where you are using the variables **carb_grams**, **protein_grams**, and **fat_grams**. This is a mathematical calculation, and * means multiplication (e).

In this line of code, you're adding together 4 times the value stored in **carb_grams**, 4 times the value stored in **protein_grams**, and 9 times the value stored in **fat_grams**.

So, here's the calculation the computer will be doing for you:

- **carb_grams** holds the value **30**, so 4 times 30 is 120.
- **protein_grams** holds the value **5**, so 4 times 5 is 20.
- **fat_grams** has the value **15**, so 9 times 15 is 135.

Adding all of those together gives you **275**.

Going further, you can have an output line where you're streaming to **cout** first some text, then a variable, and then some more text (f).

First, you have the text **There are**. Next, you have the variable **calories**. When you get to this variable name, what will be output is the value that's in the variable **calories**, which in this case is **275**. Then, you stream out some more text, **calories in this dish**, and finally use an end-of-line command to end the line of output text. And remember, you always need a semicolon to end the command.

The net result when you run this program will be to output one line: **There are 275 calories in this dish**.

It is your program's job to calculate the value, and you assign that value to the variable **calories**. Notice that you first calculate the value on the right side and then assign that to the variable on the left.

```
calories = 4 * carb_grams + 4 * protein_grams + 9 * fat_grams;
```

Exercise 2

Write a program that has 2 variables: **weeks** and **days**. **Initialize weeks** to **20**. Then, have your program calculate the number of days and output a statement giving this information.

Click here to see the solution.

Exercise 3

Try reading a short program. Can you figure out what values this program outputs?

```
1   // Program 2_9
2   // What does this output?
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       float x = 3.0;
8       float y;
9       float z;
10      y = x;
11      z = x * y;
12      x = 5.5;
13      cout << x << " " << y << " " <<
    z << endl;
14  }
```

Click here to see the solution.

// INCREMENTING VARIABLES

Any time you make an assignment, you want to first evaluate the code on the right side and then, once you know the value, assign the resulting value to the left side. In this example, a line has been added that you might encounter if you were adding extra butter, or other fat, to the recipe (13). It's still fat, so you don't need a new variable. Instead, the new line of code is **fat_grams = fat_grams +1**.

The way to think about this is that you first evaluate the right side, substituting in the value of any variables. In this case, you start with the value of **fat_grams** at **15**. So, the right side of this statement evaluates to **15 + 1**, or 16. You then assign that value, 16, to the variable on the left. It turns out that the variable that gets the new value is again **fat_grams**, but that's not a problem—you've already figured out the value to use on the right side.

The end result is that the value stored in the variable **fat_grams** has been increased by **1**.

That operation—where you take a value of a variable, add something to it, and then assign the end result back to the original variable—is fairly common. You often want to modify a value rather than assign a totally new value.

In fact, modifying existing variables like this is such a common operation that C++ has a shorthand way of writing this operation.

```
1   // Program 2_10
2   // Using a variable name on both sides of the assignment statement
3   #include<iostream>
4   using namespace std;
5
6   int main() {
7       int carb_grams;
8       int protein_grams;
9       int fat_grams;
10      carb_grams = 30;
11      protein_grams = 5;
12      fat_grams = 15; // First we assign a value of 15
13      fat_grams = fat_grams + 1; // Evaluate right side first, then
    assign to left
14      int calories;
15      calories = 4 * carb_grams + 4 * protein_grams + 9 * fat_grams;
16      cout << "There are " << calories << " calories in this dish"
    << endl;
17  }
```

The way this is done is with the notation **+=**. For example, when you have the **+=** set to **1**, it's doing exactly what you just saw—it takes the current value of the variable, adds the amount **1** on the right side to it, and stores the new result as the value of the variable.

```
13      fat_grams += 1; // Adds 1 to
        the value currently stored in
        variable
```

In general, the variable is increased by whatever the amount on the right side is. So, the line you see above, **fat_grams += 1**, is doing the same thing as the line from before: **fat_grams = fat_grams + 1**.

This works the same for subtraction. If you have money in the bank and withdraw $100, you have **money_in_bank -= 100.0**.

This also works for multiplication. If you have rabbits, then pretty soon you may triple your number—in other words, **num_rabbits *= 3**.

These compact **operators** can take any number, but the most common is increasing a variable by the number 1.

Think about counting: If you have a number, you usually count by 1s, increasing the number by 1 each time. You say that you're **incrementing** the variable.

An even more compact way you can express incrementing is with the **++** operator.

The value of **1999 ++** would be **2000**.

In your code, then, you can write **fat_grams++**.

So, you have 3 ways to write the same line of code, and they all do basically the same thing:

- **fat_grams = fat_grams + 1**
- **fat_grams += 1**
- **fat_grams++**

All 3 work the same: The value in the variable **fat_grams** is increased by 1. And because 1 gram of fat is 9 calories, you end up with 9 more calories than the previous result—284 instead of 275.

Now you know where the language C++ gets its name! The "++" means the next increment above the previous value. In the case of C++, it's the next language after C.

Just like there's **++**, there's also **--**, which will decrement by 1. If you used 1 less gram of fat in your recipe than usual, then you could write **fat_grams --** and would end up with 9 fewer calories—266 instead of 275.

C++ also lets you write the increment operator before the variable name, but these 2 methods don't always mean quite the same thing.

Putting **++** after the variable—for example, **x++**—means that you increment the value of the variable to something new *after* doing the parts of the command that were already there.

Here, printing **x++** gives the initial value of **1**.

```
1   // Program 2_14
2   // Program to print x++
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int x = 1;
8       cout << x++ << endl << x << endl;
9   }
```

Putting **++** before the variable—for example, **++x**—means that you increment to a new value of the variable *before* doing other parts of the command.

So, in this code, printing **++x** will print the new, updated value of **2**.

```
8       cout << ++x << endl << x << endl;
```

// HOW C++ SUPPORTS MATHEMATICAL FUNCTIONS

Let's say you have 2 variables, **a** and **b**, and they're assigned the values **12** and **3**. Notice that you can stream an **expression** like **a+b** directly to output. When this kind of line is executed, the math will be done first and the result is what's output.

In this case, outputs work as you'd expect:

- **a+b** is **15**
- **a-b** is **9**
- **a*b** is **36**
- **a/b** is **4**

How does this work when division doesn't occur evenly? For example, what happens if you divide 7 by 3?

When C++ sees division by 2 integers, it does integer division, meaning that it returns only the quotient, with no remainder. Because 3 goes into 7 twice, the result is 2.

If you want to find the remainder from division, you can use the **modulus operation,** which is represented by **%**. That will give you only the remainder—not the quotient—when one number is divided by another. So, if you combine integer division with modulus, you can get the quotient and remainder from any division operation.

In **Program 2_18**, the first **cout** line will output the quotient, **2**. The second **cout** line, where you stream out **a%b**, will output the remainder, **1**, which is the remainder of 7 divided by 3.

```
1   // Program 2_17
2   // Integer Division
3   #include <iostream>
4   using namespace std;
5
6   int main()
7   {
8      int a, b;
9      a = 7;
10     b = 3;
11     cout << a + b << endl; // Addition
12     cout << a - b << endl; // Subtraction
13     cout << a * b << endl; // Multiplication
14     cout << a / b << endl; // Division of INTEGERS (no
       remainder)
15  }
```

```
1   // Program 2_18
2   // Modulus and integer division
3   #include <iostream>
4   using namespace std;
5
6   int main()
7   {
8      int a, b;
9      a = 7;
10     b = 3;
11     cout << a / b << endl; // INTEGER Division (no
       remainder)
12     cout << a % b << endl; // MODULUS (the remainder
       from division)
13  }
```

// INPUT

Input is handled in a very similar way as output. Values are streamed in to variables. You get your input from the user typing in data into a window. This window is again referred to as the console, so the term ***cin*** is used to designate that you have input from the console.

Just like the output indicated that the stream of information should flow from variables leftward into **cout**, for input, the information will flow rightward from **cin** to the variables. So, you'll use >> to designate this; the direction of the stream of data is from **cin** to the variables.

Look at this modified calorie-counting code. First, there is a **stream operator** with output for the console, giving the user instructions on what to do (10). In this case, the user is to enter the grams of each type.

Next, you have 3 **cin** lines to indicate input that you'll get from the console (h).

The first of these indicates that the first thing the user types will be stored in the variable **carb_grams**. The second thing the user enters will go into **protein_grams**, and the third will go into **fat_grams**.

```
1  // Program 2_19
2  // Compute calories for dish based on user input
3  #include<iostream>
4  using namespace std;
5
6  int main() {
7     int carb_grams;
8     int protein_grams;
9     int fat_grams;
10    cout << "Enter the number of grams of carbohydrates,
   protein, and fat, separated by spaces." << endl;
11    cin >> carb_grams; // Read in grams of each
12    cin >> protein_grams;                                  h
13    cin >> fat_grams;
14    int calories;
15    calories = 4 * carb_grams + 4 * protein_grams + 9 *
   fat_grams;
16    cout << "There are " << calories << " calories in this
   dish" << endl;
17 }
```

In short, now the user can choose the grams of each type instead of having those values stuck on the initial values set in the code.

If you run the program and enter the values **10**, **20**, and **30**, for example, then you get an output telling you that **there are 390 calories in this dish**; that is, a dish with 10 grams of carbohydrates, 20 grams of protein, and 30 grams of fat will have 390 calories.

Just like the output streams could be combined with multiple stream operators on each line, you can combine multiple input stream operators on one line. The input line you see here is the same as the 3 lines shown previously.

```
11      cin >> carb_grams >> protein_
    grams >> fat_grams;
```

You still read the first piece of entered data into **carb_grams**, the second into **protein_grams**, and the third into **fat_grams**. Running the program gives the same output as before.

What if you wanted to ask users for carb, fat, and protein grams using 3 different questions?

You could just have 3 different **cout** output lines, each saying what to enter, as shown in **Program 2_21**. Each is followed by a separate **cin** input line, reading values in to each of the variables. ◆

```
1   // Program 2_21
2   // Prompting user for multiple inputs
3   #include<iostream>
4   using namespace std;
5
6   int main() {
7       int carb_grams;
8       int protein_grams;
9       int fat_grams;
10      cout << "Enter the number of grams of carbohydrates: ";
11      cin >> carb_grams;
12      cout << "Enter the number of grams of protein: ";
13      cin >> protein_grams;
14      cout << "Enter the number of grams of fat: ";
15      cin >> fat_grams;
16      int calories;
17      calories = 4 * carb_grams + 4 * protein_grams + 9 * fat_grams;
18      cout << "There are " << calories << " calories in this dish" << endl;
19  }
```

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 3 and sections 4.1–4.3.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 2.1–2.2.

c Ousterhout, *A Philosophy of Software Design*, chap. 14 (re: names).

Exercise 1 Solution

Good Names

Both **tank_capacity** and **length2** are valid and somewhat descriptive.

The **i** is also valid as a variable name. There are circumstances that you'll use a single letter like **i** for a variable; for example, in math, you might have a value like **x** and use **i** as a subscript or superscript.

Valid Names (but not great choices)

Naming a variable **something** tells you too little about what it's supposed to hold.

A name like **account** is not horrible, but it's not clear what value it stores: Is it the account balance, the account ID, the account name, etc.? A name like **account_balance** would be much easier to understand.

Invalid Names

You cannot use a reserved keyword like **namespace**.

You cannot have a space in a name, so **left foot** is not allowed; you need to use underscore if you want to separate words.

A punctuation mark, such as an exclamation point, is not allowed in a name, so **Success!** is out.

Names cannot start with a number, so you cannot have **2nd_account**.

Click here to go back to the exercise.

Exercise 2 Solution

Here's what that code might look like.

```
1   // Program 2_7
2   // Program to convert
    weeks to days
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int weeks = 20;
8       int days;
9       days = weeks * 7;
10      cout << "There are
    " << days << " days in
    " << weeks << " weeks."
    << endl;
11  }
```

OR

```
1   // Program 2_8
2   // Initializing
    variables in declaration
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int weeks = 20;
8       int days =
    weeks * 7;
9       cout << "There are
    " << days << " days in
    " << weeks << " weeks."
    << endl;
10  }
```

Click here to go back to the exercise.

Exercise 3 Solution

Notice that you declare 3 variables: **x**, **y**, and **z**. The **x** is initialized to **3**. You then assign **x** to **y** so that **y** also has the value 3. Next, **z** gets the value of **x** times **y**, which is **3** times **3**, or **9**. So, at this point, both **x** and **y** are **3** and **z** is **9**. Then, you assign the value **5.5** to **x**; **y** and **z** are not affected. So, when you stream the values of the 3 variables to output with the **cout** statement, you get:

5.5 3 9.

Click here to go back to the exercise.

// QUIZ

1 a How many variables are declared in the following program?

b What is the output?

```
#include<iostream>
using namespace std;

int main() {
    int score = 10;
    int dozen = 3;
    int totalnum;
    totalnum = 20 * score;
    totalnum += 12 * dozen;
    cout << "There are " << totalnum << " items."
<< endl;
}
```

2 What one or 2 lines of code could be put in this program to make it run correctly?

```
#include<iostream>
using namespace std;

int main() {
    float sidelength;
    cin >> sidelength;
    // One or 2 lines of code here
    cout << "The area of a square with side length "
<< sidelength << " is " << area << endl;
}
```

3 What is the output of the following program?

```
#include<iostream>
using namespace std;

int main() {
    int a, b, c, d, e;
    a = 31 / 3;
    b = 31 % 3;
    c = 3;
    d = c;
    c++;
    e = c;
    e-=2;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;
    cout << e << endl;
}
```

4 Imagine that you have a wall and want to know how many cans of paint will be needed to paint it. Read in the length and height of the wall and the number of square feet a can of paint will cover, and output the number of full cans of paint that you will use up when painting the wall.

Click here to see the answers.

// QUIZ ANSWERS

1 a There are 3 variables declared: **score**, **dozen**, and **totalnum**.

b The output is

```
There are 236 items.
```

Notice that **totalnum** first gets the value of **20** times **score**. Because the variable **score** has the value **10**, **totalnum** begins with the value 200. Then, the **+=** command is used to increase the amount in **totalnum** by the amount on the right side. Because the variable **dozen** has the value **3**, multiplying **12** times **dozen** gives 36. This is then added to **totalnum**, giving a value of **236**, which is output.

2 This is an example of 2 lines of code:

```
float area;
area = sidelength * sidelength;
```

This is an example of one line of code:

```
float area = sidelength * sidelength;
```

Notice that the output statement uses the variable **area**, so you must declare a variable named **area**. Because the side length is a floating-point number, **area** should be of type **float**. And to compute **area**, you can multiply the side length times itself and assign this value to **area**. The calculation and assignment can be done either as an initialization or as a separate assignment statement.

3 The output is as follows:

```
10
1
4
3
2
```

Notice that all numbers are integers. So, **31/3** is integer division, and the result assigned to **a** is the quotient, **10**.

The **%** is the modulus operation, giving the remainder after division, so the result assigned to **b** is the remainder of **31** when divided by **3**, which is 1.

Then, **c** is assigned the value **3**, and because **d** is assigned the value of **c**, it also has the value **3**. Next, **c** is incremented (using the **++** operation). This changes the value of **c** to **4**; **d** is not affected. Next, **e** is assigned the value of **c**, so **e** also has the value **4**. Finally, the value of **e** is reduced by 2, so it has the value of **2**.

Again, no other variables' values are changed. Thus, printing **c**, **d**, and **e** will print the values **4**, **3**, and **2**.

4 There are many possible solutions. Here is one of them:

```
// Calculate paint coverage
#include<iostream>
using namespace std;

int main() {
    float length, height, sqft, sqftpercan;
    cout << "Enter the length of the wall: ";
    cin >> length;
    cout << "Enter the height of the wall: ";
    cin >> height;
    sqft = length * height; // Total square feet to paint
    cout << "Enter how many square feet a can of paint covers: ";
    cin >> sqftpercan;
    float canstouse;
    canstouse = sqft / sqftpercan;
    cout << "You need " << canstouse << " cans of paint." << endl;
}
```

Click here to go back to the quiz.

03 Booleans and Conditionals in C++

Having choices opens up possibilities, and conditionals are ways to make choices in programs. They offer a way of saying: "If this set of conditions is met, then do this." These choices allow you to have much more interesting and varied programs. In order for you to have a conditional, though, you need a way of specifying what a condition is. And to do this, you need a way of saying that a value can be either true or false. A variable defined as being true or false is a **Boolean**, named for George Boole, who developed a way to simplify logic.

IN THIS LECTURE:

// BOOLEAN VARIABLES

You already know that you can have variables of an integer type and variables of a floating-point decimal type. You can also have logic variables that are of a Boolean type. In C++, you can declare these variables to be of type ***bool***, with a value of either ***true*** or ***false***.

true and *false* are defined keywords in C++.

In this example, there are 2 Boolean variables: one called **test_true** that you initialize to the value **true** and one called **test_false** that you initialize to the value **false**.

Although there are 2 keywords—***true*** and ***false***—C++ ultimately reads them as just the 2 possible values of a single bit.

When this program runs, the **true** value gets output as a **1**, and the **false** value gets output as a **0**.

```
1   // Program 3_1
2   // Illustrating Boolean values
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      bool test_true = true;
8      bool test_false = false;
9      cout << "test_true is: " << test_true << endl;
10     cout << "test_false is: " << test_false <<
    endl;
11  }
```

One way to think about this is that a Boolean variable can only take on 2 values: **`true`** or **`false`**. In a computer, a bit can also have just 2 values: **`1`** or **`0`**. So, once your code has been compiled into a set of machine instructions, those variables you declared as Booleans could be stored in just 1 bit. And you'll use a **`1`** to represent **`true`** and a **`0`** to represent **`false`**.

There are no special **`true`** or **`false`** values, just **`1`** or **`0`**. In fact, C++ interprets *any* value that's not **`0`** as **`true`**. In **`Program 3_2`**, the Boolean variable **`test_true`** is assigned the value **`2`**. But when you print this out, you see that the variable has the value **`1`**—that is, **`true`**.

The operations you can do on Boolean variables are also different. You don't add, subtract, multiply, or divide. Instead, for Booleans, there are 3 main operations: **`and`**, **`or`**, and **`not`**. These work in the same way that you might see in a logic course.

The **`not`** operation simply flips the Boolean value: **`true`** becomes **`false`**, and **`false`** becomes **`true`**.

In C++, you use **`!`** to represent **`not`**. For example, **`!raining`** means "not raining" and **`!windy`** means "not windy."

Notice that you put **`!`** *before* the Boolean value that you want to take **`not`** of.

In **`Program 3_3`**, you have 2 variables, with **`raining`** being **`true`** and **`windy`** being **`false`**. If you take **`!raining`** and **`!windy`**, you'll get the opposite values for each one—that is, **`false`** and **`true`**, respectively.

The **`and`** operation is used when you have 2 different Boolean values—say, **`x`** and **`y`**. In logic, if you say "x and y," then that statement is true

```
1   // Program 3_2
2   // Any non-zero value is interpreted as true
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       bool test_true = true;
8       bool test_false = false;
9       test_true = 2;
10      cout << "test_true is: " << test_true << endl;
11      cout << "test_false is: " << test_false << endl;
12  }
```

```
1   // Program 3_3
2   // Using ! (not) to reverse Boolean values
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       bool raining = true;
8       bool windy = false;
9       cout << "raining is: " << raining << endl;
10      cout << "windy is: " << windy << endl;
11      cout << "Not raining is: " << !raining << endl;
12      cout << "Not windy is: " << !windy << endl;
13  }
```

if, and only if, *both* "x" and "y" are true. This is *not* addition! It's not enough for one to be true; they both have to be true for the overall statement to be true.

And is represented in C++ by **&&**.

Be careful to use 2 ampersands to indicate **and**. If you use only one, that's known as a *bitwise and*, and that's not what you want.

The **or** operation again takes 2 Boolean values, but it is **true** if *either* one of the 2 is **true**. It's also **true** if both are **true**.

In C++, **or** is represented with **||**.

You'll probably find the character **|** above the backslash key on the right of your keyboard above the return or enter key.

The 3 basic Boolean operations—**and**, **or**, and **not**—can be combined to make more complex **logical operations**. And there is an order of precedence in the way that these logical operators are evaluated: Your computer will evaluate **not** first, then **and**, and then **or**.

You can specify the order with parentheses. For example, in **Program 3_6**, both of these expressions (a) are identical, and they evaluate to **true**. However, it's easier to follow the second one. Really, there's no downside to putting in parentheses, so put them in to make sure your expressions are clear.

In fact, you'll notice that even on the first line (7), there are parentheses around the whole expression.

If you leave the parentheses out, you'll get an error when you try to compile!

That's because the stream operator, **<<**, takes precedence over the **and** and **or** operators. So, the code reads as though you're trying to output just **not true** on the first line, and then it encounters an **or** that doesn't make sense. The same thing happens on the second line.

```
1  // Program 3_6
2  // Boolean order of operations
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7      cout << (!true || !false && true) << endl;
8      cout << ((!true) || ((!false) && true)) << endl;
9  }
```

a

Always enclose your Boolean expressions in parentheses to ensure you don't have any weird behavior due to order of operations.

Exercise 1

Is this overall expression **true** or **false**? Work through the expression from the innermost parentheses outward.

(((T && F) || (!(T || F))) && ((!F) && ((!F) || T)))

Click here to see the solution.

// COMPARISON OPERATORS

One of the most common ways of determining whether a Boolean is **true** or **false** is with **comparison operators**.

When you compare 2 values in C++, the result is a Boolean—either **true** or **false**.

You can assign the result of the comparison to a Boolean variable, such as by writing **Team1Winner = (Team1Points > Team2Points)**.

Or you can use a comparison in a Boolean expression, such as by writing **(heartrate > 100) || (respiration > 30)**.

Program 3_7 shows several comparisons. You have 2 variables, **Team1Points** and **Team2Points**, initialized to some values—in this case, **17** and **20**, respectively (7).

You can compare to see if **Team1Points** is less than **Team2Points** using **<** (8). The answer is **true**, and the output will show a **1**. The same is true if you use a less-than-or-equal-to (**<=**) comparison (9).

You can also make a greater-than (**>**) comparison (10) or a greater-than-or-equal-to (**>=**) comparison (11). In this case, the answers will be **false**, because **Team1Points** is not greater than, or greater than or equal to, **Team2Points**.

```
1  // Program 3_7
2  // Comparisons: less than/greater than to give a
   Boolean result
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7      int Team1points = 17, Team2points = 20;
8      cout << (Team1points < Team2points) << endl;
9      cout << (Team1points <= Team2points) << endl;
10     cout << (Team1points > Team2points) << endl;
11     cout << (Team1points >= Team2points) << endl;
12 }
```

You can assign the results of a comparison to a Boolean variable. In the snippet of code at right, there is a **water_temp** variable and Boolean variables can be created: **is_freezing** and **is_boiling**. These are set by comparing water_temp to **32°** and **212°** Fahrenheit. In this example, the temperature **35.3°** is neither, so both comparisons will return **false**, and thus both Boolean values are **false**.

SNIPPET

```
float water_temp = 35.3; // Fahrenheit

bool is_freezing = (water_temp <= 32);

bool is_boiling = water_temp >= 212;
```

You often want to see if 2 things have the same value or don't have the same value. For this, you want the equality and inequality operators:

- To check whether 2 things are equal, you use `==`.
- To check for inequality, you use `!=`.

> - Remember that a single equal sign is not saying "equals"; it's saying "assign." And if you are working with comparisons, you need to use 2 equal signs together.
> - Because there's no "not equals" symbol on the keyboard, you instead use an exclamation point to mean "not" and then the equal sign (**!=**) to mean "not equal to."

Let's say that you're operating a business and you want to give a discount to anyone under 18 or anyone who is 65 or older.

```
1   // Program 3_9
2   // Illustrating comparisons to create Boolean value
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      int age;
8      cout << "Enter your age: ";
9      cin >> age;
10     bool eligible_for_discount;
11     eligible_for_discount = ((age < 18) || (age >= 65));
12     cout << eligible_for_discount << endl;
13  }
```

After reading in an age, you can set a Boolean variable, **eligible_for_discount**, to be the result of a Boolean expression containing comparisons (11). You can check to see if the age is less than **18** or if it is greater than or equal to **65** by writing this: **eligible_for_discount = ((age < 18) || (age >= 65))**.

This Boolean variable **eligible_for_discount** will be **true** if either the age is less than **18** or the age is greater than or equal to **65**—otherwise, it will be **false**. For example, for age **10**, the program will output a **1** for **true**; however, for age **45**, you get a **0** for **false**.

// CONDITIONAL STATEMENTS

The real power of Boolean values and expressions comes when they get put into conditional statements. A common term for a conditional statement is an if-then-else statement. The simplest form is the if-then statement. A conditional statement starts with the keyword *if*.

Next, in parentheses, is a single Boolean value. The Boolean value could come in many forms. It could be a simple value, such as **true** or **false**, but the value is more likely to be given by a Boolean variable—or even a Boolean expression, made up of comparisons and Boolean operations.

```
if (true) // simple Boolean value

if (isFreezing) // Boolean variable

if ((age < 18) || (age >= 65))
// Boolean expression
```

Regardless of which way the Boolean value is specified, it needs to appear in parentheses. This Boolean is the *condition* of the *conditional statement*.

Then, after the parentheses is the command to be followed if the Boolean value is **true**; that is, the command that follows the **if** statement only gets executed if that Boolean value in the parentheses evaluates to **true**.

If there's just a single command to execute, then you can write that command, just like any other.

There are 3 common ways of writing this:

1. It could be written right after the parentheses on the same line.
2. It could be written on the next line. If this is done, it is common to indent the command anywhere from 2 to 4 spaces. Indenting is just a way of showing (to people) that the command only gets executed if the **if** statement is **true**.
3. Probably the most common option is to put curly braces around the command. The computer does read the curly braces, and what they mean is that everything inside of the curly braces counts as the thing to do if the Boolean is **true**. That way, you can have lots of commands—not just one—executed when the Boolean is **true**. When doing this, it is again common to indent each of the commands that should be executed to show that they're only encountered when the Boolean is **true**. The closing of the curly braces is commonly put on a new line, not indented.

Always put curly braces around the result of the **if** clause, even if there's only one command. If you leave off the curly braces and later add another command as part of the **if** statement, it's easy to forget to add the curly braces. Get in the habit of always using curly braces for the **if** clause to avoid problems later on.

Suppose you want to determine whether someone had a discount and give them a message that they're eligible for the discount.

You can do this with an **if** statement. You write ***if***, followed by parentheses and your Boolean expression: It will be **true** if the age is less than **18** or greater than or equal to **65**. Then, you write your commands, starting with curly braces, and then our output statement: **You're eligible for a discount!**.

If you run this program and enter a valid age, such as **10** or **70**, you'll get that message printed out. If you enter some other age, such as **45**, you won't.

```
1   // Program 3_10
2   // Illustrating an if statement
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int age;
8       cout << "Enter your age: ";
9       cin >> age;
10      if ((age < 18) || (age >= 65)) {
11          cout << "You're eligible for a discount!" << endl;
12      }
13  }
```

What if you wanted to output something different whenever the person was *not* eligible for a discount? This is where the full version of **if-then-else** comes in handy.

By adding an **else** clause, you can extend the **if** statement to also state what command should be done if the Boolean value is **false**.

Right after the command, or set of commands, in curly braces that should be executed if the Boolean is **true**, you write the word ***else***. Notice that the closing of the previous curly braces might be at the start of the **else** line.

After the **else**, you give the command you want to execute if the condition is **false**. If you want to execute more than one command, you again need to enclose them in curly braces.

Just like before, it is helpful to indent to show which commands belong to which part of the statement. Your IDE will probably indent the correct amount for you automatically, though sometimes if you write commands out of order, it won't. By indenting consistently, it'll be easy to see which commands are executed when the condition is **true** and which are executed when it's **false**.

In the example, an **else** clause is added (b), so you have an if-else statement—commonly called an if-then-else statement, even though the word *then* isn't explicitly used.

```
1   // Program 3_11
2   // Illustrating an if-else statement
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int age;
8       cout << "Enter an age: ";
9       cin >> age;
10      if ((age < 18) || (age >= 65)) {
11          cout << "You're eligible for a discount!" << endl;
12      }
13      else {                                                    b
14          cout << "Sorry, no discount is available." << endl;
15          cout << "  But our prices are so low you won't even notice!"
    << endl;
16      }
17  }
```

As with **then** commands, get in the habit of always using curly braces for the **else** statement.

The **else** clause will print out a message when the Boolean expression is **false**, such as: **Sorry, no discount is available. But our prices are so low you won't even notice!**.

Now if you run this code and enter an age that's eligible for a discount, you still get the original message. And if you enter an age that's not eligible, you get the new message instead.

Notice that it's easy to extend the case to do more than one command inside of the curly braces. In **Program 3_12**, the code has been modified to have a new floating-point variable. **Price** will store the cost for that person. If there is a discount, the person will get a **price** of just $5, while if he or she doesn't get a discount, the **price** will be $7.50. Regardless of which price is set, the **cout** command at the end will output the **price** that person will have to pay.

If you run this, you'll see that those eligible for a discount do indeed get a lower cost.

In that last set of code, notice that you ended up adding some additional lines into the set of commands as part of the **if** and **else** statements (c). That's another example of why it's good to use curly braces to group your commands together, even if you think you'll have only one command.

```
1   // Program 3_12
2   // Multiple operations within if and else clauses
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      int age;
8      cout << "Enter your age: ";
9      cin >> age;
10     float price;
11     if ((age < 18) || (age >= 65)) {
12        cout << "You're eligible for a discount!" << endl;
13        price = 5.0;
14     }
15     else {
16        cout << "Sorry, no discount is available." << endl;
17        cout << "  But our prices are so low you won't even notice!" << endl;
18        price = 7.5;
19     }
20     cout << "Your price is " << price << endl;
21  }
```

c

Exercise 2

In this modified version of the **Program 3_12**, the curly braces following the **else** statement have been left off. What do you think will happen?

```
1   // Program 3_13
2   // Illustrating an if-else statement - missing curly braces
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int age;
8       cout << "Enter your age: ";
9       cin >> age;
10      float price;
11      if ((age < 18) || (age >= 65)) {
12          cout << "You're eligible for a discount!" << endl;
13          price = 5.0;
14      }
15      else
16          cout << "Sorry, no discount is available." << endl;
17          cout << "  But our prices are so low you won't even notice!" << endl;
18          price = 7.5;
19
20      cout << "Your price is " << price << endl;
21  }
```

Click here to see the solution.

The commands you put in curly braces for an **else** clause can be just about any commands. That includes more **if** statements!

Let's look at some code used to calculate the shipping cost for a package.

First, there's weight and type of shipping. So, you have a few variables: a floating-point **package_weight** (7) and a Boolean indicator of whether it is **send_priority** shipping or not (8). In this case, **package_weight** is initialized to **5.0** and **send_priority** is initialized to **true**.

As for cost, the system is that if there is priority shipping, all packages weighing less than **3.5** have a fixed cost of **10.00**, and otherwise the cost for priority is based on weight, while all nonpriority packages just have a cheaper cost based on weight alone.

So, the code for **price** starts with an **if** statement to check whether the package is to be sent priority or not. There's also an **else** clause to handle situations when it's not priority (d).

In the case where you have priority shipping, you have another **if** statement (e). This one checks if package weight is below **3.5**; if so, it sets a fixed cost of **10.00** and otherwise assigns a cost given by weight times **3.0**.

Having one **if** statement inside of another is called nesting. It's even possible to have multiple levels of nesting.

As for the case where you don't have priority shipping, this is where the **else** clause of the outermost **if** statement comes in (19). In this case, the **price** is a lower rate of weight times **1.5**.

You should use nesting whenever the code you have makes natural sense to organize in that way—where you want to make one choice and then another.

```
1  // Program 3_14
2  // Illustrating nested if statement
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     float package_weight = 5.0;
8     bool send_priority = true;
9     float price;
10    if (send_priority) {                    d
11       if (package_weight < 3.5) {          e
12          price = 10.00;
13       }
14       else {
15          price = package_weight * 3.0;
16       }
17    }
18    else {
19       price = package_weight * 1.5;
20    }
21    cout << "Your price is " << price << endl;
22 }
```

```
1    // Program 3_15
2    // Illustrating mulitple if-else
     statements
3    #include <iostream>
4    using namespace std;
5
6    int main() {
7       int age = 10;
8       float cost;
9       if (age < 3) {
10          cost = 0.0;
11      }
```

f

```
12      else {
13          if (age <= 5) {
14              cost = 1.0;
15          }
16          else {
17              if (age <= 10) {
18                  cost = 3.0;
19              }
20              else {
21                  if (age <= 17) {
22                      cost = 5.0;
23                  }
24                  else {
25                      cost = 6.0;
26                  }
27              }
28          }
29      }
30      cout << "Your cost is " <<
     cost << endl;
31   }
```

There's another way of writing **if** statements that is common and useful. Suppose you have some sort of tiered pricing system, similar to the one from before, but you want kids under 3 to be free, those from 3 to 5 to cost $1, those from 6 to 10 to cost $3, those from 11 to 17 to cost $5, and everyone else to cost $6.

Program 3_15 shows a bunch of nested **if** statements. You start by checking whether the age is less than **3**, and if so, set a **cost** of **0.0**. Otherwise, you go to the **else** clause.

In the **else** clause, you have another **if** statement, this time checking whether the age is less than or equal to **5**. Again, you set a **cost** in that case; otherwise, you go to another **else** clause. And this continues until you handle all the cases.

When you look at this code, all those nested statements are confusing, especially when your conditions aren't that difficult.

In these instances, it's helpful to remember that in C++, the spacing and line breaks don't actually matter. Thus, you can reorganize the indentation, tabs, and organization to make this more readable.

First, notice that in this situation, you have a series of **else** clauses, each of which contains a single **if** statement (f).

This is the one situation where you should not use curly braces, and not indent the **if** statement. If you get rid of the curly braces when there is just another **if** statement inside and move that **if** up to the previous line, the code becomes much cleaner and easier to read (g).

Notice that you still have the **else if** combinations, but you organize them in a way that makes it much easier to see the different cases and what to do in each one. You can read down the list of conditionals and find the first one that evaluates to **true**. Or, if none evaluate to **true**, then you have a final **else** clause. Although this code works exactly the same as the previous code, it's much easier to follow. This way of organizing the code is called an **else if** structure, and it's a good idea to use it in cases like this, where you want to select one of several related options. ◆

g

```
12      else if (age <= 5) {
13          cost = 1.0;
14      }
15      else if (age <= 10) {
16          cost = 3.0;
17      }
18      else if (age <= 17) {
19          cost = 5.0;
20      }
21      else {
22          cost = 6.0;
23      }
```

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, section 4.4.1.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 4.1, 4.2, and 5.3.

Exercise 1 Solution

The overall expression is **`false`**.

Click here to go back to the exercise.

Exercise 2 Solution

If you run the code, you find that even people who should have gotten a discount got a message meant for those who didn't and that everyone received the same cost, $7.50. Why did that happen?

When you get to the **`else`** statement, because there were no curly braces, only the first command after the **`else`** applied to the **`else`** clause. The other lines—the second **`cout`** and setting **`price`** to **`7.5`**—will run for every situation. The fact that they're indented doesn't matter to the computer; they need to be in curly braces if they only apply to the **`else`**.

This is the type of mistake that happens easily, even to experienced programmers, when people aren't careful to put curly braces around their **`if`** clauses.

Click here to go back to the exercise.

// QUIZ

1 Answer the following questions for a Boolean variable.

a What is the type used when declaring a Boolean variable?

b If the variable has the value **`false`** and it is output, what would be output?

c If the variable has the value **`true`** and it is output, what would be output?

d If you assign 123.45 to the variable, what value will it have?

2 What are the symbols used to do each of the following?

a Compare whether 2 values are not equal to each other

b Compare whether 2 values are equal to each other

c Take the **not** of a Boolean value (convert **`true`** to **`false`** or **`false`** to **`true`**)

d Take the **and** between 2 Boolean values: something that is **`true`** if and only if both values are **`true`**

e Take the **or** between 2 Boolean values: something that is **`true`** if either or both values are **`true`**, and **`false`** otherwise.

3 What would be the value of the following Boolean expression?

```
!((true && true) || (false && true)) || (!true || (!false && true))
```

4 What is the output of the following program?

```
#include<iostream>
using namespace std;

int main() {
    int a = 3;
    int b = 4;
    int c = 3;
    if (a==b) {
        if (b!=c) {
            cout << "Output 1" << endl;
        } else {
            cout << "Output 2" << endl;
        }
    } else {
        if (b!=c) {
            cout << "Output 3" << endl;
        } else {
            cout << "Output 4" << endl;
        }
    }
}
```

5 Write a program that asks a user for a water temperature and reports whether the water is ice, freezing, liquid, boiling, or vapor.

Click here to see the answers.

// QUIZ ANSWERS

1

a `bool`

b `0`

c `1`

d **`true`** (remember: any nonzero value is treated as **`true`**)

2

a `!=`

b `==`

c `!`

d `&&`

e `||`

3 true:

```
!((true && true) || (false && true)) || (!true || (!false && true))
!((    true    ) || (    false    )) || (false || ( true && true ))
!(                true                ) || (false || (     true     ))
                 false                  || (           true           )
                                  true
```

4 `Output 3`

Notice that because the first conditional (**a==b**) is false (**a** is **3** and **b** is **4**, so they are not equal, and the comparison is **`false`**), the **`else`** clause is followed. The next step is for the nested conditional to check whether **b** is not equal to **c** (**b!=c**). Because **b** is **4** and **c** is **3**, they are not equal, so the comparison is **`true`**. So, the first clause is followed, and the line **`Output 3`** is output.

5 There are multiple possible answers. Here is one:

```
// Report phase of water
#include<iostream>
using namespace std;

int main() {
    float temperature;
    cout << "Enter a water temperature in Fahrenheit: ";
    cin >> temperature;
    if (temperature < 32) {
        cout << "Ice" << endl;
    } else if (temperature == 32) {
        cout << "Freezing" << endl;
    } else if (temperature < 212) {
        cout << "Liquid" << endl;
    } else if (temperature == 212) {
        cout << "Boiling" << endl;
    } else {
        cout << "Vapor" << endl;
    }
}
```

Click here to go back to the quiz.

04 Program Design and Writing Test Cases in C++

IN THIS LECTURE:

Although code writing is very important, there's more to making a program than just writing code. Considerations like design and testing are just as important.

// THE STRUCTURE OF A C++ PROGRAM

It's often helpful to start a program with some sort of comment—what might be called a header comment—that gives some information about the purpose of the program (1, 2).

The next line is a pound-include (**#include**) line (3). The pound sign (#) is used to designate special commands that are given to the **compiler**, which is what will translate your program into machine instructions.

When a compiler looks at a program that it needs to process, it first identifies any lines that have #. These lines are processed first as special instructions to the compiler. These are called **preprocessor commands**, because they take place before processing the rest of the code.

The **#include** is by far the most commonly used, but there are a few other preprocessor commands, all starting with #:

- **#include** statements can bring in new lines of code.
- **#define** statements can modify code.
- **#ifdef** statements (short for *if defined*) can choose lines of code.

The **#include** is a way of saying, "There is another file out there that contains some additional functions. Bring that file into this program." This gives you access to any functions in that other file.

For example, **#include <iostream>** gets you access to important input and output functionality, like **cin** and **cout**.

```
1  // Program 4_1
2  // Hello, World! program
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      cout << "Hello, World!" << endl;
9  }
```

The **#includes** that you usually begin your program with import files to give you additional functionality. Those files are usually from the C++ standard libraries, which are installed with C++ and therefore are available whenever you need them.

The next line is the **`using namespace`** command. A namespace is a way of grouping various functions together. By saying **`using namespace`**, you are indicating which group of functions you're using. In this case, that's the **`std`** (short for *standard*) namespace. A way to think of a namespace is as a special name that needs to be added to the beginning of commands.

Here is how the code would look if you had to include the namespaces directly when using each command. Notice how you have to write **`std::`** in front of **`cout`** and **`endl`** (7); this is the way to specify that those are defined in the **`std`** namespace.

```
1  // Program 4_2
2  // Hello, World! without
   "using namespace std"
3
   #include <iostream>
4
5  int main()
6  {
7      std::cout << "Hello,
   World!" << std::endl;
8  }
```

There are big projects that might require this way of working. There might even be multiple namespaces, because different groups on a project often create new namespaces to clearly distinguish their code. But it can be a pain to keep writing **`std::`** before each command for a small project where the scope of the **`using namespace std`** is clear. So, the line **`using namespace std`** lets you say, "Unless I specify otherwise, I'm using the standard namespace." And because this is a C++ command, not a command to the preprocessor, it ends with a semicolon.

In short, this is a line that saves typing and makes the rest of your code easier to read.

The next line in the code is the **`int main()`** line, which creates something called a function—a set of commands that gets executed together.

The parentheses show that it's a function, **`int`** means that it returns an integer, and **`main`** is a predefined name.

Note that **`main`** is a special function; it's the one that the compiler will recognize as the function to start with. The commands that come in the curly braces after **`main`** are the ones that will be executed first when the code is run.

Notice that **`main`** starts out with the term **`int`**. The way you can think of this is that the main program that results will have an integer value when it runs. This integer value has a meaning: If the program completes without an error, its value is 0; if there is an error encountered, it has some nonzero value. The particular value can vary based on the particular error and, for an experienced programmer, can give more information about the type of error encountered.

C++ gives you a lot of flexibility in terms of style. Unlike some other languages, C++ basically lets you space your code and indent however you want to. Sometimes, different people have different styles of formatting code. The style to use is based mainly on personal preference, unless you are part of a bigger project, where often coders try to match each others' styles for consistency.

As you develop programs, you want to do so with a style that makes it clear what the intention of your code is. And that can involve choosing descriptive variable names, using good spacing and indentation, and organizing the code in a logical manner.

// DESIGNING AND TESTING YOUR PROGRAM

How do you make sure your code is organized logically? This is where design comes in.

Usually, your program begins with some idea—what you want to be able to do. Often, this is just a vague idea at the beginning; from there, you try to get specific. You need to determine exactly what you're doing—what you expect as input to the program, what computations you want to perform, and what you want to be able to output. These are often called the program requirements.

At this point, you have 2 things you should do: determine **test cases** and design the program itself. Following program design, you'll code up your program, and as you code, you'll use the tests you determined to continually check that your code is on track. When you've written all the code and passed all the tests, you'll have achieved success—a working program.

Because of the impact that errors can have, testing is often a major part of a software development project. On professional projects, the estimates vary from about a quarter to about half the total time on a project being spent on testing!

In many ways, being able to pass tests is the most critical ability of a program. This is why some people believe that coming up with tests should be the first thing you do when developing a program. This approach is called test-driven development. If you've developed a sufficient set of tests, then having a program that can pass the tests means that the program works correctly. The only way something would be considered an error, or **bug**, is if it caused a test not to pass.

Here are some good rules of thumb for what to test:

- First, think of the boundary cases, also called edge cases or corner cases. In many programs, the edge cases are wherever there's a break or boundary of some sort. Suppose you had a test where you wanted to give a discount to people under 18 years of age. In that case, 18 would be an edge case—and so would 17. Those 2 ages would be right on the boundary and would be some of the most likely places that an incorrect computation might occur.
- Next, try to think of special cases and invalid options that users might enter. If you're writing a program that handles peoples' names, you might assume that everyone has a first, middle, and last name, each made up of letters with an initial capital and the rest lowercase. But you probably want to ensure you can handle names with apostrophes or hyphens, different capitalization, and more or fewer than 3 names. Like with edge cases, the idea is to think about what assumptions you might have made for the typical case and ensure you handle the other cases that might come up, too.
- Finally, test some of the regular cases. Usually, just a few of these are sufficient.

Think early about your tests. Sometimes, thinking about tests will help you remember things you need to include in the software design itself.

When you've thought of your tests, it's a good idea to write them down. Often, this is as simple as writing down an input and what the output should be.

Exercise

Suppose you want to write a program that handles dates. What are some of the days of the year you might want to include in your test cases?

Click here to see the solution.

Once the tests have been considered, it's time to start thinking about how you can actually develop your program.

The usual way to approach this is with **pseudocode**. Instead of writing actual code right away, first you write out steps in more general terms that describe what you want the program to do but that aren't actual code. You can write those steps of pseudocode as a series of comments; then, you can translate that pseudocode to actual code.

Suppose you want to create a program that tells you where your blood pressure numbers fall in terms of healthy or not. What are the steps you'll want the program to follow?

Many programs follow a basic pattern: getting input, performing computations, and producing output.

In this case, for input, you want to have a user type in blood pressure. For computation, you'll use the blood pressure to classify someone's level of hypertension. And you'll just output to the person what his or her category is.

In terms of categorizing blood pressure, you'll use a set of conditions based on looking at the systolic and diastolic blood pressures. You're going to want to implement these concepts in code.

PSEUDOCODE

Blood Pressure Analyzer

1. Input: Blood pressure (Systolic, Diastolic) entered from keyboard
2. Compute: Determine what category that BP puts someone into
3. Output: Print the category the person is in to the screen

Elevated blood pressure:
Diastolic < 80 and Systolic between 120 and 129

Stage 1 Hypertension:
Diastolic between 80 and 89, or Systolic between 130 and 139

Stage 2 Hypertension:
Diastolic 90 or greater, Systolic 140 or greater

Danger Zone:
Stage 2 Hypertension, but Diastolic 120 or greater, or Systolic 180 or greater

Hypotension:
Diastolic 60 or lower, Systolic 90 or lower

Because this program is relatively simple, your actual design process is not too complicated. You will have a sequence of steps: getting input, classifying the category, and producing output. Because all you're doing when classifying the category is outputting the result to the screen, you can do that at the same time as you classify the pressure.

You can write some short pseudocode to list the various steps you'll need to follow.

A good way to start when facing a programming task is to take the pseudocode steps that you know you will follow and transform them to comments.

Then, you can go back later and fill in the code between the comments. These comments serve 2 purposes: They will provide an outline that you can follow while developing code and serve as documentation for the code later so that it's clear which parts belong to which.

PSEUDOCODE

Blood Pressure Analyzer

1. Read in systolic and diastolic blood pressures
2. Compute and Output Category:
   - a. Check for Elevated blood pressure: 120<=S<=129 and D<80
   - b. Check for Stage 1 Hypertension: 130<=S<=139 or 80<=D<=89
   - c. Check for Stage 2 Hypertension: S>=140 or D>=90
   - d. Check for Danger Zone: S>=180 or D>=120
   - e. Check for Hypotension: S<=90 or D<=60
   - f. Say OK if nothing else applies

First, you outline your program, putting in the standard information. Inside of **main**, you'll enter your comments showing what needs to be done—which came from the pseudocode steps.

You now have your program design and can start filling in parts of the program. As you do so, you'll want to frequently stop, compile your code, and make sure it's running as expected.

```
1   // Program 4_6 - Stage 1: Pseudocode converted to comments
2   // Blood Pressure Analyzer
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       // Read in systolic and diastolic blood pressures
8
9           // Check for Elevated blood pressure: 120<=S<=129 and D<80
10
11          // Check for Stage 1 Hypertension: 130<=S<=139 or 80<=D<=89
12
13          // Check for Stage 2 Hypertension: 180>S>=140 or 120>D>=90
14
15          // Check for Danger Zone: S>=180 or D>=120
16
17          // Check for Hypotension: S<=90 or D<=60
18
19          // Say OK if nothing else applies
20
21  }
```

The first part to fill in is the input section—where you want to read in the blood pressure by having the user type it in. You first need a few variables to hold the 2 blood pressure values. You'll name these **systolic** and **diastolic**, and because blood pressure numbers are always reported as integers, these will be an integer type.

```
7     // Read in systolic and diastolic
      blood pressures
8        int systolic, diastolic;
9        cout << "Enter your systolic
      pressure (the larger, first
      number): ";
10       cin >> systolic;
11       cout << "Enter your diastolic
      pressure (the smaller, second
      number): ";
12       cin >> diastlic;
```

Then, you'll ask the user to enter each of these and read them in. You'll have a **cout** statement that outputs instructions to the user. Notice that there's no end line here, so when users type information, they'll do so at the end of the line of text that was just output. You read in systolic first and diastolic second.

At this point, you've written a full chunk of code, so it's time to test it.

You'll create one line of output to make sure you know what you read in—just to ensure the data was read in correctly. Putting in output statements is one relatively easy way to do a quick check that variables have the right values.

In this case, you add one **cout** statement and try to compile the code to test it.

```
   cout << "You entered values
of " << systolic << " and " <<
diastolic << endl;
```

But you get an error when you compile! Your compiler might show the error information differently, but it should show something like **12:9**, which says that in the function **main**, an error was discovered on line 12 and 9 characters in. It then gives you a hint as to what the error is. In this case, it says that you're trying to use a variable you didn't declare.

It turns out that the problem was that **diastolic** was misspelled; it looked like you were trying to use some variable named **diastlic**, which didn't exist.

Fortunately, that's easy to fix, and if you run the fixed program a few times, you should get the right numbers output: You enter a number and see that same number output. So, you can remove the output line that you put in for testing and go on to the next section.

Then, you fill in the portion of code around the next comment. In this case, you are checking for elevated blood pressure. To do this, you have a conditional: You check to see that the systolic is at least 120 and no more than 129 and that the diastolic is less than 80. If this is the case, you'll output that the person has elevated blood pressure.

```
14       if ((systolic >= 120) &&
      (systolic <= 129) && (diastolic
      < 80)) {
15          // Check for Elevated blood
      pressure: 120<=S<=129 and D<80
16          cout << "You have elevated
      blood pressure." << endl;
17       }
```

After writing this, the next thing to do is test.

Because you want to handle edge cases, you should check diastolic pressures at 79 and 80 as well as systolic pressures of 120 and 129, plus a few numbers on either side of those numbers. Plus, you should check some more typical values, such as 125 over 75, 125 over 85, and 110 over 70.

Mistakes are a normal part of programming—even for the best programmers in the world. As you develop programs, don't worry that you encounter bugs and errors. Keep testing frequently as you develop your code and you'll catch your bugs sooner, making it easier to locate and then fix them.

Next, you'll fill in another comment area—in this case, the check for and then response to **stage 1 hypertension**. The situation is very similar to the earlier case, but with a different condition. It makes sense to use the else-if construction to write this code.

```
18      else if (((systolic >= 130) && (systolic <= 139))
19                   && ((diastolic <= 80) && (diastolic <= 89))) {
20              // Check for Stage 1 Hypertension: 130<=S<=139 or 80<=D<=89
21              cout << "You have stage 1 hypertension." << endl;
22          }
```

So, in the case where the previous condition failed, you now check a new condition—in this case, to look for **stage 1 hypertension**. That occurs when either systolic is in the 130 to 139 range or the diastolic is in the 80 to 89 range. Your condition will be more complex.

You've now written an else-if clause in which you use comparisons of systolic and diastolic levels to compare to the limits and a **cout** statement to print: **You have stage 1 hypertension**.

It's once again time to test the code. You can test the edge cases for both systolic and diastolic, so you can check 130 over 80 and 130 over 89, along with 139 over 70, and so on. You can also check a few in the middle.

If you enter **130** over **80**, that is in the **stage 1 hypertension** area, and your program correctly outputs that information.

Now try **130** over **89**. You don't get an answer; there must be a bug somewhere!

If you look at the code, you can see that diastolic was accidentally written as **<= 80** instead of **>= 80**. To fix this, just change **<** to **>**.

Once you make that change, test again to make sure you fixed the bug. Keep testing. Try some more average cases, such as **135** over **75**. This should be hypertension, but again, you don't get a line of output!

Looking at the code, you can see another problem in your condition: The line with **&&** means that both systolic and diastolic have to be in a certain range, when the actual result should be an **or**. Change **&&** to **||** to fix that line.

This code now seems fixed, but you should run all of your tests to make sure.

If you keep going, you can fill in the rest of the program section by section. You add additional else-if sections with more conditionals to check for the various cases. In this case, you handle the check for **stage 2 hypertension**, for being in the danger zone, and for hypotension. And in the best case, when you don't fall into any of the bad categories, you get to print out a message saying that blood pressure is in a healthy range.

As you do this, continue to test to make sure that everything is working as expected. ◆

```
18   else if (((systolic >= 130) && (systolic <= 139))
19               || ((diastolic >= 80) && (diastolic <= 89))) {
20          // Check for Stage 1 Hypertension: 130<=S<=139 or 80<=D<=89
21          cout << "You have stage 1 hypertension." << endl;
22      }
```

```cpp
// Program 4_6 - Stage 9: Final Version
// Blood Pressure Analyzer
#include <iostream>
using namespace std;

int main() {
   // Read in systolic and diastolic blood pressures
   int systolic, diastolic;
   cout << "Enter your systolic pressure (the larger, first number): ";
   cin >> systolic;
   cout << "Enter your diastolic pressure (the smaller, second number): ";
   cin >> diastolic;

   if ((systolic >= 120) && (systolic <= 129) && (diastolic < 80)) {
      // Check for Elevated blood pressure: 120<=S<=129 and D<80
      cout << "You have elevated blood pressure." << endl;
   }
   else if (((systolic >= 130) && (systolic <= 139))
         || ((diastolic >= 80) && (diastolic <= 89))) {
      // Check for Stage 1 Hypertension: 130<=S<=139 or 80<=D<=89
      cout << "You have stage 1 hypertension." << endl;
   }
   else if (((systolic >= 140) && (systolic < 180))
         || ((diastolic >= 90) && (diastolic < 120))) {
      // Check for Stage 2 Hypertension: 180>S>=140 or 120>D>=90
      cout << "You have stage 2 hypertension!" << endl;
   }
   else if ((systolic >= 180) || (diastolic >= 120)) {
      // Check for Danger Zone: S>=180 or D>=120
      cout << "Your blood pressure is in the danger zone!" << endl;
   }
   else if ((systolic <= 90) || (diastolic <= 60)) {
      // Check for Hypotension: S<=90 or D<=60
      cout << "You have hypotension." << endl;
   }
   else {
      // Say OK if nothing else applies
      cout << "Your blood pressure is in a healthy range." << endl;
   }
}
```

Blank lines, or white space, can help separate parts of a program that are conceptually different.

Except for lines starting with #—which must remain on single lines by themselves—it's possible to break up any other line and almost anywhere within each line.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 5.11 and 6.2.

b Ousterhout, *A Philosophy of Software Design*, chap. 15.

Exercise Solution

Edge cases would include the first and last days of the year: January 1 and December 31. You'd probably also want to check that you can correctly handle the first and last days of some other months, including 28-day, 30-day, and 31-day months.

There is one special case: February 29. You'd want to make sure you handle leap years correctly.

Finally, you'd want to test a few random dates spread through the year. Overall, if your program worked for all those cases, it probably works everywhere.

Click here to go back to the exercise.

// QUIZ

1 When developing a larger program, which of the 2 options in each of the following pairings should come first in the process?

a Writing code or writing tests

b Writing pseudocode or writing comments

c Running tests or writing code

d Writing code or writing comments

2 If you are writing a program that calculates the phase of water given a temperature, what are some good temperature values at which to check your code?

3 To determine whether someone is eligible for a loan and how much can be loaned, a person's assets, debts, and credit history are examined. Write the pseudocode and then the outline/framework (with comments) for a program that would collect information from a person and tell that person how much of a loan he or she can receive and at what interest rate. Note that you do not have sufficient information to actually write most of the code.

Click here to see the answers.

// QUIZ ANSWERS

1 a Writing tests. You should, generally, write tests prior to writing code. The idea is to decide what your program needs to do by describing how it should perform on various tests before actually running the tests.

b Writing pseudocode. Pseudocode is part of the design process and can be used to create many of the comments in a program.

c Trick question! You should generally run tests while you write code: Write some code, then test, write more code, then test, etc. Obviously, some code must be written before a test can be run over that part of the code.

d Generally, comments can be written before the code is written. This is especially true if the comments are taken from the pseudocode and describe the purpose of various parts of the code. But it is OK to write additional comments while writing code or after writing a piece of code.

2 Generally, you want to be sure to check your code near special points where things change. For water, this would be near the phase transition points: 32° Fahrenheit and 212° Fahrenheit. So, a thorough set of tests would probably verify that the program works at the transition points, 32 and 212; a little on either side of the transition points (31, 33, 211, and 213); and a few points in the middle ranges, such as 10, 100, and 1000.

3 There are several possibilities, but here is one:

Pseudocode:

1 Collect information on assets

2 Collect information on debts

3 Collect information on credit history

4 Calculate loan amount eligible for and interest rate

5 Report loan information to user

Code:

```
1  // Calculate loan eligibility
2  #include<iostream>
3  using namespace std;
4  int main() {
5      /* Collect Asset Information */
6      /* Collect Debt Information */
7      /* Collect Credit History Information */
8      /* Calculate loan amound and rate */
9      /* Report information to user */
10 }
```

Click here to go back to the quiz.

05 C++ Loops and Iteration

Programmers use **loops** to get the computer to do one of the things that it's best at and that people tend to dislike—mere repetition. Loops make it possible to repeatedly execute the same code over and over. There are 2 main types of loops: the basic **`while`** loop and the **`for`** loop, which is a compact way to make the **`while`** loop happen a set number of times.

IN THIS LECTURE:

// WHILE LOOPS

This code is for computing a balance from a loan where there is interest accumulated and payments made.

You begin by initializing the situation to a $1000 balance, where each payment period you incur interest of 5% and make a payment of $100 (a).

Then, you check to see if you have a positive balance (11). If you do, you incur interest, adding the interest onto the balance (12). You make a payment, reducing the balance by that payment (13). You increase a count on how many payments have been made, which is just 1 (14). And you print out what the balance is after that payment is made (15).

That's fine for computing the balance after 1 month. If you run this code, you'll see that the balance after 1 payment is $950—that's the $1000 starting balance plus $50 in interest and then less $100 in payment.

But that's only good for 1 payment. Obviously, you're going to need more than 1 payment to pay off the whole loan, so you can repeat that **`if`** statement over and over. You'd have to copy and paste that code for each payment you wanted to make. If you do this, you find that you need more than 15 payments—which is 15 **`if`** statements taking more than 100 lines of code—to pay off the whole loan.

```
1   // Program 5_1
2   // Single if statement for
    computing balance
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       float balance = 1000.0;
8       float payment = 100.0;
9       float interest = 0.05;
10      int numpayments = 0;
11      if (balance > 0.0) {
12          balance += balance *
    interest;
13          balance -= payment;
14          numpayments++;
15          cout << "Balance after " <<
    numpayments << " payment(s) is: "
    << balance << endl;
16      }
17  }
```

a

An easy way to program exactly what you want is with a **`while`** loop. It's like an **`if`** statement that keeps getting repeated as long as the **`if`** condition is true—in other words, *while* the condition is true.

You need to make only one small change to the program to get the functionality you want: Just change the word *if* to the word w*hile*. And if you run the program now, you'll get the exact same output you would've had when repeating the **`if`** statement 15 times.

```
11      if (balance > 0.0) {
```

```
11      while (balance > 0.0) {
```

When the program comes to the **`while`** statement, it first checks the condition, just like it did with the **`if`** statement. In this case, the condition is still that the balance is greater than **`0`**. As long as that's true, then everything in the curly braces after **`while`** is executed—even if something inside the braces makes the condition no longer true.

For example, even though the balance on the last payment, payment 15, becomes negative, you continue to execute the remaining lines. The balance becomes negative when you subtract the payment. But you still execute the remaining lines in the curly braces: **`numpayments++`** and the **`cout`** statement to print the balance.

After everything in the braces has been executed, you reevaluate the condition. After all, at the end of the loop, things have changed. In this example, the balance will have changed, so maybe it's no longer a positive balance. If the condition is still true, then you do everything in the braces again. This can continue indefinitely.

The part of the **`while`** loop inside the curly braces is called the body of the loop. Every time you go through the body of the loop, it is called an iteration of the loop.

When the condition is no longer true, you skip the things in the braces and go on to the next line after the **`while`** statement. If there are no lines after the loop ends, the whole program ends at that point.

Exercise

How might you create a **`while`** loop to print the numbers from **`1`** to **`10`**?

Click here to see the solution.

There is a less common variation of the **`while`** loop called the **`do while`** loop, which starts with the keyword **`do`** and puts the **`while`** condition check at the end of the loop and ends with a semicolon. Overall, the only difference this makes is that the program is now forced to go through the very first iteration. This can be helpful if you know that you need to go through the loop at least once.

A **sentinel** is a deliberately unusual value that you wouldn't normally encounter, so it's a value you can use to indicate when you're done processing "regular" data. A negative balance in a bank account could be a sentinel, assuming you're not allowed to have an overdrawn account. You can use sentinels to help you identify when it's time to break out of a loop. If a sentinel value is encountered, the loop will be skipped over.

If the condition never becomes false, then every time the condition is evaluated, it's true, so the loop just repeats forever. This is called an infinite loop.

For example, the loop in **Program 5_5** just keeps printing out consecutive numbers, starting at **1**; notice that the condition is that the count is greater than **0**, which is always true. If you run this program, it'll keep on going forever, until you do something special to stop it.

Generally, the way you stop the program will vary depending on your system. Sometimes, your IDE will have a Stop button of some sort that will cause it to stop; other times, you can hit control+C, and that will cause the program to "break" and stop. In the worst case, you might have to actually stop your entire system by closing whatever window you are running the program in.

If you are prepared to stop the infinite loop, try programming one yourself.

There's a good chance that you'll accidentally create an infinite loop at some point while you program. It's a common pitfall that's easy to make if you make a mistake writing a loop condition, so it's better to figure out early on how to break out of one when it happens.

One problem you want to be aware of is that when you're figuring out average age, you'll want to divide the sum of the ages by the number of people. If you divide them, you'll be performing integer division. Remember that this gives only the quotient between them, ignoring any remainder. To deal with this, you need to convert one or both of the numbers to a floating-point value.

You can do this by **casting a value**, which is a way of converting a value of one type into a value of another type. To do this, you have a few options: We can put the new type in parentheses in front of the value you want to cast, or you can state the new type and then put the value we want to convert from in parentheses afterward. Either way, you'll convert the value, if possible, to the other type.

For the 3 types you've seen so far (integers, floating-point values, and Boolean variables), it's possible to convert one type to either of the others, as **Program 5_9** shows. Note that there are some types you can't convert, such as strings of text and self-defined types.

```
1   // Program 5_5
2   // Infinite Loop
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      int count = 1;
8      while (count > 0) {
9          cout << count << endl;
10         count++;
11     }
12  }
```

```
1   // Program 5_9
2   // Type conversions
3   #include<iostream>
4   using namespace std;
5
6   int main() {
7      int a = 3;
8      float b = 2.6;
9      bool c = true;
10     cout << float(a) << " " << (float)a
    << endl;
11     cout << bool(a) << " " << (bool)a
    << endl;
12     cout << int(b) << " " << (int)b
    << endl;
13     cout << bool(b) << " " << (bool)b
    << endl;
14     cout << int(c) << " " << (int)c
    << endl;
15     cout << float(c) << " " << (float)c
    << endl;
16  }
```

// FOR LOOPS

Loops are useful for many aspects of computer programs. Any time you have a large amount of data to process, you're probably going to use a loop to go through all the pieces of data. But there's one particular pattern of specifying the number of loops that's very common, so there's a more compact way of writing a specific number of loops, called a **for** loop.

Program 5_12 is a slight variation on the previous loan-balance-calculation code. However, instead of the loop being based on having a positive balance, it's based on the number of payments.

This loop will go through a fixed number of payments. It has a variable, **numpayments**, that keeps track of how many payments have been made; uses the number of payments as its condition; and increments the number of payments in every iteration of the loop. Notice that the condition is that the number of payments is less than **5**.

What payments are printed out? What's the first one, and what's the last one?

The first you'll print out in this case is **after 0 payments**—in other words, the initial balance. That's because the **numpayments** variable begins at **0**, and the first thing done after the **while** condition is checked is to print out the results.

At the end of that first iteration, the new balance is calculated, and the number of payments is incremented to **1**. This is checked versus the condition, and because it's still less than **5**, you go through another iteration.

This will continue through payment **4**. Notice that after printing out the result of payment **4**, you calculate a new balance and update your **numpayments** to **5**. So, when you next check the condition of the **while** statement, it will be false; the **numpayments** is no longer less than **5**. So, you see results from **0** to **4**.

```
Balance after 0 payments is: 1000
Balance after 1 payments is: 950
Balance after 2 payments is: 897.5
Balance after 3 payments is: 842.375
Balance after 4 payments is: 784.494
```

Let's look at some of the ways this loop works. You have a variable you initialize before the loop begins. That's **numpayments**, which you set to **0** (10). You have a condition where you check something about that variable. In this case, that's **numpayments** being less than **5** (11). And you update the variable in every iteration. Here, you're incrementing **numpayments** (15).

This is a very common way to organize loops. You initialize a variable, your loop condition relies on that variable, and you update the variable with every iteration.

```
1   // Program 5_12
2   // While loop for computing
    balance for set number of
    payments
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       float balance = 1000.0;
8       float payment = 100.0;
9       float interest = 0.05;
10      int numpayments = 0;
11      while (numpayments < 5) {
12          cout << "Balance
    after " << numpayments <<
    " payments is: " << balance
    << endl;
13          balance += balance *
    interest;
14          balance -= payment;
15          numpayments++;
16      }
17  }
```

This initialize-compare-increment pattern is so common that it has its own form of loop: the **for** loop. **Program 5_12** and **Program 5_13** have exactly the same effect.

Notice that you have taken the 3 key parts from the **while** loop structure—initialization of the variable, the condition, and the variable update—and put them into one statement at the beginning of the **for** loop.

Notice also that you start your loop with the word *for* instead of *while*. The **for** loop basically packages up several parts that would otherwise be spread across a **while** loop.

The **for** loop always begins with the keyword *for* and then, like the **while** loop, has parentheses and curly braces, giving the body of the loop. The difference is what's in the parentheses. Instead of just a condition, like the **while** loop has, you actually have 3 different parts separated by semicolons:

- The initialization, which would come before the loop in the **while** loop formulation, is where you set the initial value for the variable you are using in the loop.
- The condition works just like the condition in the **while** statement.
- The update occurs at the end of each iteration of the loop.

```
1   // Program 5_13
2   // For loop for computing balance for set number of payments
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      float balance = 1000.0;
8      float payment = 100.0;
9      float interest = 0.05;
10     int numpayments;
11     for (numpayments = 0; numpayments < 5; numpayments++) {
12         cout << "Balance after " << numpayments << " payments is: " << balance
   << endl;
13         balance += balance * interest;
14         balance -= payment;
15     }
16  }
```

So, when you have a **for** statement, you first execute the initialization command, then check the condition on each iteration of the loop, and finally run the update command at the end of each iteration of the loop.

While loop

```
initialization
while (condition) {
    //Commands to do each iteration of loop
    //      (This is the body of the loop)
    // update
}
```

For loop

```
for (initialization; condition; update) {
    //Commands to do each iteration of loop
    //      (This is the body of the loop)
}
```

All that the **for** statement has to have is the condition; almost like the **while** loop, it's possible to omit the initialization and the update from inside the **for** statement.

For example, in this version of the code, you keep the initialization outside the **for** statement. You still need the semicolon to separate the different parts, but you can just leave out the first part.

```
10      int numpayments = 0;
11      for (; numpayments < 5; numpayments++) {
```

And in this version, you also omit the update part, putting the update back into the loop explicitly. So, the **for** statement in this instance is working exactly like the **while** statement, except the condition is surrounded by semicolons on both sides.

```
10      int numpayments = 0;
11      for (; numpayments < 5;) {
12          cout << "Balance after " << numpayments
    << " payments is: " << balance << endl;
13          balance += balance * interest;
14          balance -= payment;
15          numpayments++;
16      }
```

But then, there's not much point in using a **for** statement like this; you might as well use a **while** statement.

Normally, the advantage of the **for** loop is that all the information needed to understand the way the loop will behave is contained inside the **for** statement—inside the parentheses. It makes it easier to understand how the loop works if the initialization, condition, and update are all in one place.

In this very simple loop, notice that the **for** statement has a variable **i** that starts at **1** and continues while **i** is less than **4**, incrementing by 1 each time. So, the loop will output **In loop: 1**, **In loop: 2**, and **In loop: 3**.

After that third iteration, the variable **i** has the value **4**, so it's no longer less than **4**, and thus the loop won't go through another iteration. After the loop, the value of **i** is still **4**, so you finally output the line **After loop: 4**.

The point is that the variable you have controlling the loop—known as the loop control variable—still has a value afterward. Typically, that'll be the first value that caused the condition to fail.

Just like you were able to nest conditional statements, you can nest loops. This is useful when you want to

```
1  // Program 5_16
2  // Illustrating value of variable after a loop
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7      int i;
8      for (i = 1; i < 4; i++) {
9          cout << "In loop: " << i << endl;
10     }
11     cout << "After loop: " << i << endl;
12 }
```

do something like visit every entry in a table; you can loop over the rows, and then in each row, loop over the columns.

So, if you wanted to print out the times tables for the numbers 1, 2, and 3, you could use the code shown here, which has 2 loops nested with each other. The outermost loop has variable **i** iterating from **1** to **3**, and the inner loop has variable **j** iterating from **1** to **3**. For each, the code will print out the product of **i** and **j**.

So, in this case, when **i** has the value **1**, then **j** will have the values **1** through **3**. Then, **i** will have the value **2**, and **j** will again have values **1** through **3**. Finally, **i** has the value **3**, and **j** again gets the values **1** through **3**. So, there are 9 lines of output in this program.

```
1  // Program 5_17
2  // Nested for loops
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int i;
8     int j;
9     for (i = 1; i < 4; i++) {
10       for (j = 1; j < 4; j++) {
11          cout << i << " times " << j << " is " << i * j
   << endl;
12       }
13    }
14 }
```

The use of **i** and **j** for loop control variables is very common. Even though these are not exactly descriptive names, if you are going to loop through a sequence of values, it's common to use **i** to represent those values, and if you have 2 nested loops, it's common to use **j** for the inner loop. Because people expect **i** and **j** to be loop iteration indexes, this is one time when single letters have themselves become descriptive variable names.

// SCOPE OF VARIABLES

Scope applies to more than just loops; it can even apply to conditionals. But loops are the first place you generally want to use scope.

Put simply, every time you declare a variable, you get a new "box" in memory for that variable. But this variable only persists within the block of code—the section of commands defined by the curly braces. Once you leave a block of code defined by the closing curly brace, that box of memory disappears.

In **Program 5_18**, the initialization part of the **for** loop does more than just initialize the integer—it also declares it. That means that the variable being declared—in this case, **i**—only exists in that loop itself. Its scope is just that one loop.

In this case, the loop runs just like you'd expect, and it outputs the values **1**, **2**, and **3**.

```
1   // Program 5_18
2   // Declaring variable in a for loop header
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       for (int i = 1; i < 4; i++) {
8           cout << i << endl;
9       }
10  }
```


However, if you try to print out the value of **i** after the loop, then you get an error! The variable **i** is no longer defined at that point; the box of memory has been destroyed!

In **Program 5_19**, you declare the variable **j** inside a loop. If you run this, you get the number of iterations you'd expect, and just like you'd expect, the value of **i** takes on the values **1**, **2**, and **3** in the loop body and is **4** after the loop is over. But why is the value of **j** always **1**, even though there is a line, **j++**, that seems to change the value of **j**?

```
1   // Program 5_19
2   // Declaring a variable in a loop
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int i;
8       for (i = 1; i < 4; i++) {
9           int j = 1;
10          cout << "In loop: " << i << " " << j
    << endl;
11          j++;
12      }
13      cout << "After loop: " << i << endl;
14  }
```

Every time the loop body is executed, **j** gets declared again. So, you create a new box, named **j**, and initialize its value to **1**. Later in the loop body, you increment **j**, so it'll have the value **2**, but then the loop body ends. At this point, the memory is destroyed; there's no more **j** available.

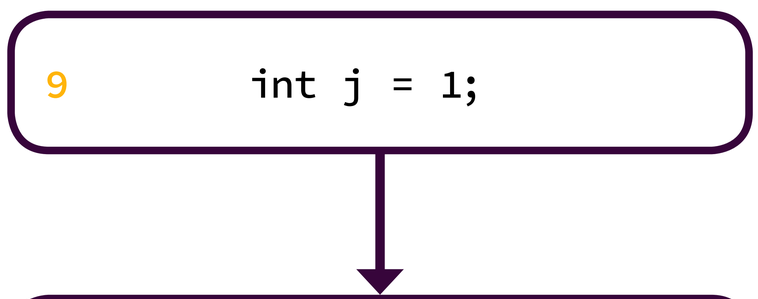
The next time you go through the loop, you again create the variable **j** and initialize it to **1**. So, every time it's printed out, the value will show up as **1**.

To get a variable to stick around from one iteration to the next inside a loop, you can declare it as a **static variable** inside the loop. If you put the word *static* in front of the declaration, it says that you want to declare a variable and you want that declaration to stick around even when it goes out of scope. So, if you come back to this same variable in this part of the code later, you'll have access to it.

```
1   // Program 5_21
2   // New variable name inside loop
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int i;
8       int j = 1;
9       for (i = 1; i < 4; i++) {
10          int j = 500;
11          cout << "In loop: " << i << " " << j << endl;
12          j++;
13      }
14      cout << "After loop: " << i << " " << j << endl;
15  }
```

Declaring a variable inside a loop sets aside new memory for that variable, even if it has the same name as a variable outside the loop. When the loop ends, that new memory is destroyed, but the original variable remains.

```
9       int j = 1;
```



```
9       static int j = 1;
```

In this case, the variable **j** is created the first time you enter the loop body. Then, each time you go through the body in the future, that line is skipped, and you don't destroy the box named **j** at the end of the loop. So, every time you encounter the **j++** line, it will increment **j**, and that new value will stick around into the next iteration.

However, that does not mean that **j** sticks around after the loop is over; it still goes back out of scope after that. If you try to access **j** after the loop, you still get an error.

A variable called **j** that you declare inside the curly braces can even have the same name as another variable declared outside the curly braces; that is, you create a new variable **j** with scope that lasts only during the loop, and once you leave the loop, the previous variable **j** is again used **(Program 5_21)**. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 4.4.2–4.4.3.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 5.4 and 5.5.

Exercise Solution

Here's one way to do it.

```
1   // Program 5_4
2   // While loop for printing 1 to 10
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int num = 1;
8       while (num <= 10) {
9           cout << num << endl;
10          num++;
11      }
12  }
```

Click here to go back to the exercise.

// QUIZ

1 For each of these cases, when would you usually use a **for** loop, and when would you use a **while** loop?

a To read in numbers until a specific number is entered

b To read in 10 numbers from a user

c To add up the numbers in a specified range

d To repeat a calculation until the value is negative

2 How would you rewrite the following code that uses a **while** loop to instead use a **for** loop?

```
1   #include<iostream>
2   using namespace std;
3
4   int main() {
5       int i;
6       i = 10;
7       while (i > 1) {
8           cout << i << endl;
9           i--;
10      }
11  }
```

3 What would be the output of the following code?

```
1   #include<iostream>
2   using namespace std;
3
4   int main() {
5       int i;
6       for(i=0;i<10;i+=2) {
7           int j = i;
8           while(j>1) {
9               j /= 2;
10              cout << i << " " << j << endl;
11          }
12      }
13  }
```

Click here to see the answers.

// QUIZ ANSWERS

1 While it may be possible to use either loop, there is a clear better choice in each case:

a **`while`**. This will repeat an unknown number of times, until some condition is reached. That is typical of a **`while`** loop.

b **`for`**. You know a specific number of times you want the loop to run, so this is generally done using a **`for`** loop.

c **`for`**. You will want to loop over each of the values in the range, and because there is a specific range, a **`for`** loop is more appropriate.

d **`while`**. You will repeat some unknown number of times until some condition is met. This is more commonly handled by a **`while`** loop.

2 Here is the most straightforward way:

```
1  #include<iostream>
2  using namespace std;
3
4  int main() {
5      int i;
6      for (i=10; i > 1; i--) {
7          cout << i << endl;
8      }
9  }
```

Remember that the initialization (**`i=10`**) is the first item in the parentheses of the **`for`** loop, the condition (**`i > 1`**) is the second term in the parentheses, and the update (**`i--`**) is the last term in the parentheses.

3 This program has 2 nested loops. The outer **`for`** loop lets variable **`i`** take on the values **`0`**, **`2`**, **`4`**, **`6`**, and **`8`** because it is initialized to **`0`**, continues only as long as **`i`** is less than **`10`**, and increases by **`2`** each time. In the inner **`while`** loop, **`j`** takes on values starting at **`i`** and then keeps dividing by **`2`**. This continues until **`j`** is **`1`** or less. Notice that the output statement occurs after **`j`** is divided by **`2`**, so generally, as long as **`j`** begins with a value greater than **`1`**, the final output value will be **`1`**. So, the overall output is:

```
2 1
4 2
4 1
6 3
6 1
8 4
8 2
8 1
```

» When **`i`** has the value **`0`**, then the inner loop is never run.

» When **`i`** has the value **`2`**, then **`j`** gets the value **`2`**; the loop is run for 1 iteration, in which **`j`** becomes **`1`**; and one line is output: **`2 1`**.

» When **`i`** has the value **`4`**, then **`j`** gets the value **`4`**, then **`2`**, then **`1`**. The values are output twice, first after **`j`** gets the value **`2`** and then again after it gets the value **`1`**.

» Likewise, when **`i`** has the value **`6`**, then **`j`** will have the value **`6`**, then **`3`**, then **`1`** (remember that this is integer division). So, again, 2 lines are output: one when **`j`** is **`3`** and one when **`j`** is **`1`**.

» Finally, when **`i`** has the value **`8`**, **`j`** will take on values **`8`**, then **`4`**, then **`2`**, then **`1`**. Thus, there will be 3 lines output, for the values of **`j`** equal to **`4`**, **`2`**, and **`1`**.

Click here to go back to the quiz.

06 Importing C++ Functions and Libraries

IN THIS LECTURE:

While the most basic programs have just a few lines of code, some of the more complex systems written in C++ require many thousands of lines of code, even when the programmers use loops and functions to make the code as compact as possible. A major system like Windows takes tens of millions of lines of code! That much code is well beyond what any one programmer is capable of writing and understanding well. One of the main ways that people collaborate to produce code is through a library.

// CODE LIBRARIES

Code libraries give you access to additional functionality; they directly give you the ability to do things in your code that you wouldn't otherwise be able to do. If you want the ability to do something special, such as perform a mathematical computation, then you can use a **library** for mathematical operations.

You've already been using an example of a library: You have been #including **`iostream`** ever since the **`Hello, World!`** program, and that's a way of giving the rest of your code access to a special library. The *io* part means *input-output*, and **`iostream`** lets you print output to the screen and read in input the user types on the keyboard.

If you did not have access to that library, you'd need to learn how the computer stores information that gets sent to the screen, how the individual keystrokes come in, and how you can use those really basic operations to do things like write text to the screen or read in a large number typed in.

With enough time and effort, you could learn to do that, but someone already did all that on your behalf by creating the library, so you don't have to mess with those details yourself.

There are a few good reasons to use libraries.

- Libraries let you separate ideas in your code, which lets you have a simple way of thinking about any one piece of code.
- Libraries are reusable. When someone else has figured out how to do a particular task, there's no sense in doing the exact same thing yourself.

// HOW CODE AND LIBRARIES ARE COMPILED IN C++

A library typically consists of 2 parts: a **header file** and an implementation file. You can think of the header file as the interface for the library. The header file basically makes a few definitions and declarations and says, "Here is what will be in the library and how it can be accessed." But it doesn't necessarily contain the details of how the things in the library are actually implemented.

When people write their own header files, the header file will typically have the extension **`.h`**. You might also see the extension **`.hpp`**, where the *pp* is added to show that the header is part of C++ (*plus plus*). For standard headers, you don't worry about any extension, because it's automatically implied. You just write **`iostream`**, not **`iostream.h`**.

The details of how the library is implemented are contained in the implementation file, which will contain regular C++ code and will have a **`.cpp`** extension. If the file is just standard C code, it might only have a **`.c`** extension. The implementation file has all of the details of how the things declared in the header file should be implemented, and will usually #include the header file at the beginning.

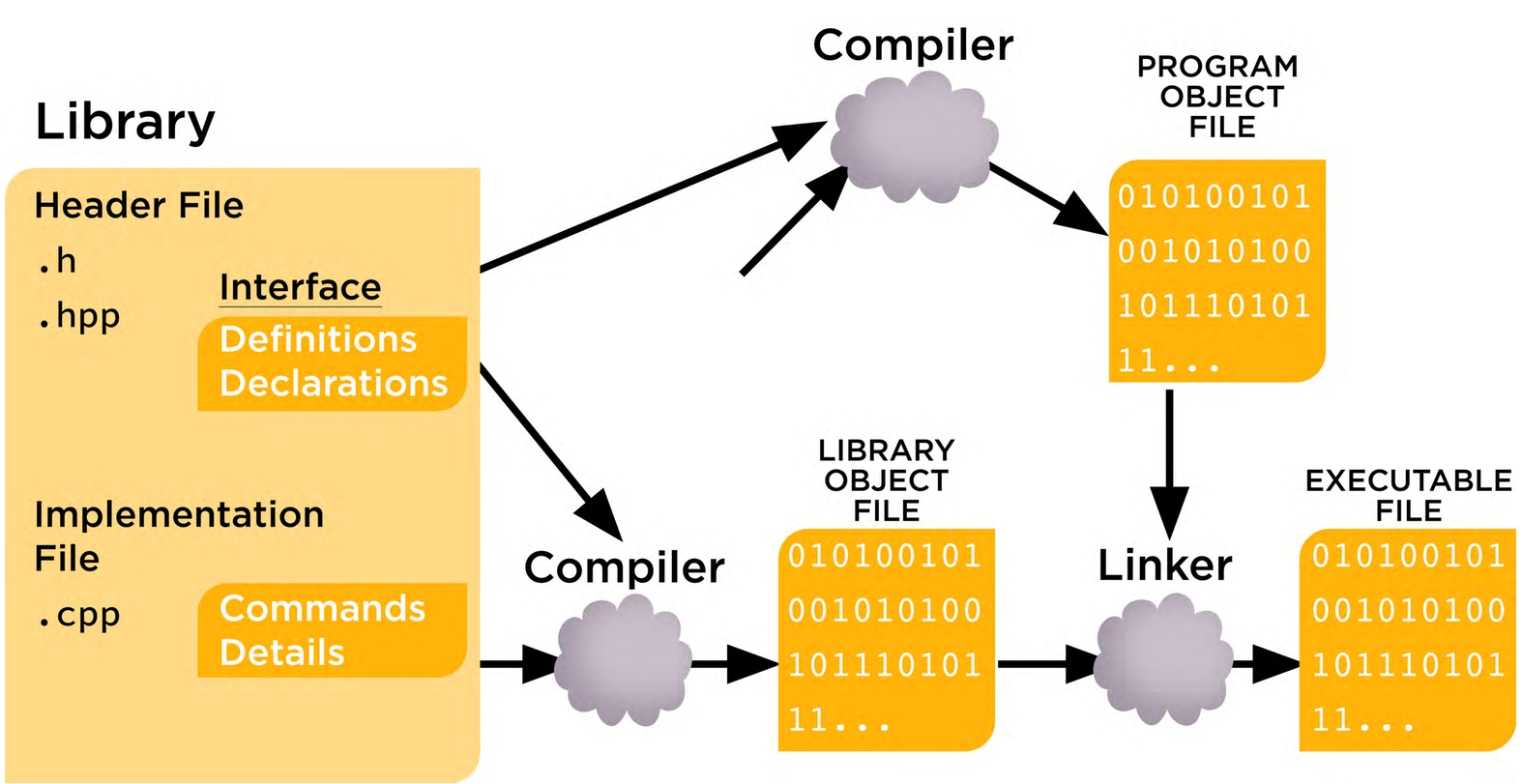
Note that a library does not actually contain a **`main`** function. The library is not going to contain code that is run as the **`main`** program by itself; it's just going to contain functions that can be used by the **`main`** program.

The library can then be compiled into an object file, which is machine instructions. But these machine instructions can't be run on their own. Because no **`main`** function was defined in the library, it doesn't have any commands that just get executed on their own; instead, these are machine instructions waiting to be used by some other program.

Next is your program—the one that you want to actually make use of the library. This is where you'll define **`main`**. It needs to know what functionality is available and how to use it. So, to get access to that library's functionality, it will #include the library's header file.

Your **`main`** program gets compiled into its own object file. But this new object file does not have the actual implementation of the stuff in the library. Those commands are in the other object file—the library's object file.

So, after all the compilation is done, there's one more step: the **linker**, which will combine the **`main`** program's object file with the library's object file to create an **executable file**. The executable file now has all the information from both the program file and from the library file, and it can thus be run on its own.



Because a program is not too useful until it is linked with the library, the linking phase is usually combined with the compiling phase. In fact, when people talk about *compiling*, they usually really mean "compile and link."

The libraries are usually already compiled well in advance. The standard C++ libraries are installed automatically when you install the C++ compiler. If you ever create your own library, you'll compile that part ahead of time, too.

Also, realize that a program can, and often does, link with more than one library. The process is the same; there are just more header files to #include.

```
1   // Program 6_3
2   // Using more than one library
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   int main() {
8      string s;
9      cin >> s;
10     cout << "You entered " << s << endl;
11  }
```

// THE C++ STANDARD LIBRARY

The first libraries to be aware of are a standard set of files that are included with every C++ installation. These libraries are collectively often called the **C++ Standard Library**, and there are around 50 of them in all. These standard libraries are the ones to rely on first when you have something you want to do. Just write a **`#include`** statement near the top of your program, putting the library name inside the angle brackets.

Because C++ basically grew out of C, C++ libraries have been inherited from C. And C was, and still is, used for several applications, so many libraries were developed for C, including some standard C libraries, such as ones for math and time. C++ still provides access to all of these libraries, which are named "c__", where the "__" is the name of the old C library.

The standard C++ libraries provide a lot of functionality for you to make use of. There are dozens of functions located in just the math library.

The website cplusplus.com gives clear explanations for lots of functions, including all the standard C++ library headers.

Try following the Reference link on the site to find more about a specific topic. Or just try a search engine; the first or second item in the search results may take you to the same location.

In the case of this code, you're going to use the **cmath** library. To do this, you'll #include **cmath**. This will take the header file, **cmath**, and put all that header file information into your code. One of the things **cmath** will give you access to is a square root function, **sqrt**, which you can use in your program.

In this case, you print out the square root of **2.0**, which is **1.41421**.

This is the first time you've really used a **function** in your code. When you use a function, there are 2 key parts: a function name and parentheses that come right after it.

In the square root example, you have a function for computing the square root, and the name is **sqrt**. After the function name there will always be a pair of parentheses.

```
1  // Program 6_4
2  // Using cmath (and iostream)
3  #include <iostream>
4  #include <cmath>
5  using namespace std;
6
7  int main() {
8     cout << sqrt(2.0) << endl;
9  }
```

Inside those parentheses, there will often be arguments, which are values that you will be giving to the function. In the case of computing a square root, you need a single number to take the square root of. So, you need to have one argument: the number to take the square root of.

When you write code and want a function to happen, you write the function name and parentheses, along with any arguments you need to give it. This is referred to as a **function call**, and it is said that you are calling or invoking the function. It means that when you get to that part of the code, you execute the code corresponding to that function. Remember, not every function needs arguments; some will perform operations without requiring arguments. But regardless, they'll still have parentheses.

If the function is designed to compute some value, that function call is replaced by the value it computes. It is said that the function has returned a value. For the square root function, you will have a floating-point value—the value of the square root—returned. Whenever you encounter a function call, if there is a return value, you can think of the function call as just being replaced by the value of the function.

Notice that the square root function call occurs in the middle of an output statement. The value that is returned from the function is what is output.

The square root function, like all functions, is not defined in C++ all by itself. To access to it, you need to get it from the **cmath** library, which includes a variety of math functions, such as a power function (**pow**), trigonometric functions, and exponential and logarithmic functions.

// RANDOM NUMBERS

Suppose you want to develop a guessing game in which the computer picks a number and the user has to try to guess the number picked.

To do this, you're going to need a function that lets you get a random number. Random number generation is actually a tricky problem—it's nearly impossible to generate truly random numbers with a deterministic computer—but the key is to generate **pseudorandom** numbers, which seem like random numbers and are just as useful for most applications.

You can turn to libraries, where people who understand pseudorandom number generation have implemented ways of getting these numbers for you.

When you have #includes, the library file you're including is in brackets formed by < and >. An alternative to this is to use quotation marks. The code works exactly the same.

Generally, the convention is to use brackets for things that are in the C++ Standard Library and quotation marks for other libraries or files that you're using.

A web search for something like "C++ random" can be the quickest way to identify a relevant library function from the standard C++ library, if one exists.

In this case, C++ has 2 standard libraries that will generate random numbers. One of them is called **random**, but it's complicated to use. An easier-to-use library is the C Standard Library, **cstdlib**, which includes the widely used function **rand**, which does not take any arguments and returns an integer. The integer will be some number between 0 and the maximum-possible random integer.

To generate a random number, you have a simple program that you begin by #including the **cstdlib** header. Then, you can use the function **rand**, with no arguments, in the code to give you an integer.

If you run **Program 6_7**, you see that a random number is generated.

You can generate more random numbers by calling the **rand** function multiple times. Every time you call **rand**, you get a new random integer. In **Program 6_8**, 3 random numbers are generated.

```
1   // Program 6_7
2   // Random Number generation
    example
3   #include <iostream>
4   #include <cstdlib>
5   using namespace std;
6
7   int main() {
8      int x = rand();
9      cout << "The random number is
    " << x << endl;
10  }
```

```
1   // Program 6_8
2   // Generating multiple random
    numbers
3   #include <iostream>
4   #include <cstdlib>
5   using namespace std;
6
7   int main() {
8      int x = rand();
9      cout << "The random number is
    " << x << endl;
10     cout << "Here is another
    random number: " << rand()
    << endl;
11     cout << "... and another: " <<
    rand() << endl;
12  }
```

With pseudorandom number generation, you have to initialize the numbers with a seed value, which can be any number, and the numbers produced seem very random—you can't tell they're related to the seed. However, if you start with the same seed, you will always get the same sequence of random numbers every time you call **rand**.

You can adjust this by setting the seed, which is done with a function called **srand**, which takes in one argument: the seed value to be used. After calling **srand**, every call to **rand** will be generated based on that seed value. If you give the same seed, you will get the same sequence of pseudorandom numbers.

Program 6_9 shows what the code will look like with a different seed value. In this case, the seed used is 3. If you run the code with different seeds, you'll get different sequences of numbers.

Notice that the sequence you get for a seed of **3** is very different from that for a seed of **4**, which would be very different from a seed of **5**, and so on.

If you want something that seems even more random, you can try to choose a seed without having to specify a number. A common way of doing this is to use the current time as the seed value. So, even though the numbers are not truly random, no one is going to be able to guess the actual digits ahead of time.

There's a library called **ctime** that will give you the current time.

In **Program 6_10**, the **ctime** library has been #included. That gives you access to a function, **time**, that will give you a current time. The **time** function takes one argument, which you should just set to **0**. The function **time** will then give you a different value at every time, based on the current clock time.

If you run this code, you'll get a different sequence of numbers every time you run it. Because the actual time in the computer is changing every second, the **time** function is giving you a different value all the time, so every time you run the program, you have a different seed.

```
1   // Program 6_9
2   // Random Number generation
    example with seed (new seed value)
3   #include <iostream>
4   #include <cstdlib>
5   using namespace std;
6
7   int main() {
8      srand(3); // Set the seed for
    the pseudorandom sequence
9      int x = rand();
10     cout << "The random number is
    " << x << endl;
11     cout << "Here is another random
    number: " << rand() << endl;
12     cout << "... and another: " <<
    rand() << endl;
13  }
```

```
1   // Program 6_10
2   // Random Number generation example with time-based seed
3   #include <iostream>
4   #include <cstdlib>
5   #include <ctime>
6   using namespace std;
7
8   int main() {
9      srand(time(0)); // Set the seed for the pseudorandom sequence
10     int x = rand();
11     cout << "The random number is " << x << endl;
12     cout << "Here is another random number: " << rand() << endl;
13     cout << "... and another: " << rand() << endl;
14  }
```

Now that you're able to generate random integers, let's say that you want to limit your integers to between 1 and 100. One way to do this is to use the modulus operator (**%**), which gives you the remainder after dividing. If you take a number modulus **100**, then you get the remainder, which will be something between 0 and 99. If you had random numbers to begin with, then you'll have random remainders.

```
1   // Program 6_11
2   // Random Number generation within range 0-99
3   #include <iostream>
4   #include <cstdlib>
5   #include <ctime>
6   using namespace std;
7
8   int main() {
9      srand(time(0)); // Set the seed for the
    pseudorandom sequence
10     int x = rand() % 100; // Modulus 100 means number
    between 0 and 99
11     cout << "The random number is " << x << endl;
12  }
```

Then, if you want a random number to be from 1 to 100 instead of from 0 to 99, you just add **1** onto the modulus.

```
10     int x = rand() % 100 + 1; // Adding 1 to change
    [0-99] to [1-100]
```

Program 6_13 is a modification of the previous code that will generate random numbers in the 1 to 100 range. If you run this several times, you'll see that you get random numbers, all in the range of 1 to 100.

```
1   // Program 6_13
2   // Multiple random numbers in range of
    1 to 100
3   #include <iostream>
4   #include <cstdlib>
5   #include <ctime>
6   using namespace std;
7
8   int main() {
9      srand(time(0)); // Set the seed for the
    pseudorandom sequence
10     int x = (rand() % 100) + 1;
11     cout << "The random number is " << x
    << endl;
12     cout << "Here is another random number: "
    << (rand() % 100) + 1 << endl;
13     cout << "... and another: " <<
    (rand() % 100) + 1 << endl;
14  }
```

People have created C++ libraries to do all sorts of things, from display 3-D graphics to perform advanced scientific computation. At least at first, it's often better to stick to more established C++ libraries, but if you're interested in finding additional libraries, one particularly good source is Boost.org. In fact, some of what started out as Boost libraries have now been incorporated into the standard C++ library! Here are some places to start: Boost Filesystem Library and Boost Math Library.

To make your guessing game, you'll want to pick a random number and then repeatedly ask a user to pick a value, telling the user if he or she is too high or too low. You'll see how long it takes for the user to guess the right answer.

Remember to start by writing some pseudocode, giving an outline of what you want the program to do.

Next, you use the pseudocode outline and convert it into comments in the code. You will make each of the steps a comment within **main**, testing the code after each step is coded. Then, you fill in the parts of the program one by one.

PSEUDOCODE

1. Generate a random number
2. Get a user's initial guess
3. Repeat until the user guesses the right value
   - a. Report if too high or too low
   - b. Get another guess
4. User guessed it—report how many tries

```
1   // Program 6_14_a
2   // Random Number Guessing Game
3   #include <iostream>
4   #include <cstdlib>
5   #include <ctime>
6   using namespace std;
7
8   int main() {
9      // Generate a random number
10
11     // Get a user's initial guess
12
13     // Repeat until the user
    guesses the right value
14
15     // Report if too high or
    too low
16
17     // Get another guess
18
19     // User guessed it - report
    how many tries.
20
21  }
```

For someone just starting, the amount of information in these libraries may feel a bit overwhelming. So, if you want to look at library functions, start with a header for something that sounds simple and just focus on one or 2 functions, trying to understand them. To get started, you might look at the libraries **string**, **complex**, **ctime**, or **stack**.

For more libraries, you can search for lists of "open source C++ libraries" available on the web. On the home page for cppreference.com, there is a link for Non-ANSI/ISO Libraries that's a list of unofficial libraries that are available for a variety of applications.

If you run this program, you'll be able to play a different game every time.

Review this program twice. First, do it with everything visible and see if you're able to follow each of the statements and commands used:

- **cstdlib** and **ctime** (a),
- the **srand** function (10),
- the **time** function (10),
- the **rand** function (11),
- the **while** loop (b),
- the **if** conditional (c),
- the input (16, 31), and
- the output (15, 23, 26, 30).

Then, run through the program again, but this time see if you can generate something similar to the code by yourself, starting from the pseudocode. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, section 8.3.

b See also www.cplusplus.com and en.cppreference.com for lists of standard libraries that can be imported.

```
1   // Program 6_14
2   // Random Number Guessing Game
3   #include <iostream>
4   #include <cstdlib>
5   #include <ctime>                    a
6   using namespace std;
7
8   int main() {
9      // Generate a random number
10     srand(time(0)); // Set the seed for the pseudorandom sequence
11     int num_to_guess = (rand() % 100) + 1;
12
13     // Get a user's initial guess
14     int user_guess;
15     cout << "Guess a number from 1 to 100: ";
16     cin >> user_guess;
17     int num_guesses = 1;
18
19     // Repeat until the user guesses the right value
20     while (user_guess != num_to_guess) {                    b
21        // Report if too high or too low
22        if (user_guess < num_to_guess) {
23           cout << "Your guess was too low." << endl;
24        }
25        else {
26           cout << "Your guess was too high." << endl;
27        }                                                    c
28
29        // Get another guess
30        cout << "Guess again: ";
31        cin >> user_guess;
32        num_guesses++;
33     }
34
35     // User guessed it - report how many tries.
36     cout << "You guessed it! It took you " << num_guesses << " tries."
   << endl;
37  }
```

// QUIZ

1 The **cmath** library includes a function **floor** that will round a number down to the next integer.

a What line of code lets you use the functions in the **cmath** library?

b What line of code would you use to assign to a variable **a** the result of rounding the variable **x** down to the next integer?

2 Write code that will simulate 10 rolls of a pair of dice. Print out the result of each roll (the sum of the numbers on the dice) on a separate line.

3 This task will require looking up information about a library online to demonstrate the process that you can generally follow to make use of other libraries.

Look up the library **<iomanip>** in an online resource (e.g., http://www.cplusplus.com/reference/iomanip/ or https://en.cppreference.com/w/cpp/header/iomanip).

On separate output lines, use the **setprecision** and **setw** functions to output the number **1234.56789** in a range of **10** characters but rounded to **4** decimal places, in a range of **9** characters rounded to **3** decimal places, and in a range of **8** characters rounded to **2** decimal places.

Note: You will need to output **fixed** to **cout** before any other output to ensure that you have a decimal representation and not scientific notation. Simply add the line **cout << fixed;** to your program. You might wish to experiment to see what happens if you do not include that line.

Click here to see the answers.

// QUIZ ANSWERS

1 a
```
#include<cmath>
```

b
```
a = floor(x);
```

Note: In addition, **cmath** includes a **ceil** function that rounds up to the next integer and **round** that rounds to the nearest integer.

2 Here is one way of writing the program:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
    srand(time(0)); // Set the pseudorandom seed
    int d1, d2;
    int i;
    for(i=0;i<10;i++) {
        d1 = rand() % 6 + 1; // Random number from 1 to 6
        d2 = rand() % 6 + 1; // Random number from 1 to 6
        cout << d1+d2 << endl; // Output the sum of the dice
    }
}
```

Notice that you include the **cstdlib** library to give you access to the **rand()** function and **ctime** to give you access to the **time** function. Then, you set a seed using the **time** value so that you get different rolls each time you run the program. You have a **for** loop that iterates 10 times.

In each, you generate one simulated roll by taking **rand()** (which generates a random integer), then taking it modulus **6** (which gives the remainder after dividing by 6—thus, a number from 0 to 5), and finally adding **1** (to make the number a random one in the range of 1 to 6). You do this for each die and output the sum.

3 a You will need to first go to the website and learn how the **setw** and **setprecision** commands are used. It is often most helpful to look at example code there.

b Notice that you will need to #include **<iomanip>**. Also notice that the commands are all in the **std** namespace (with **std::** in front of them); you can leave this off because you are writing **using namespace std;** in your programs.

c You will need to put the **setw** and **setprecision** commands in the **cout** statements. There are multiple ways you can do this, but one example is shown below, in which the **fixed** statement is used.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    float x = 1234.56789;
    cout << fixed;
    cout << setw(10) << setprecision(4) << x << endl;
    cout << setw(9) << setprecision(3) << x << endl;
    cout << setw(8) << setprecision(2) << x << endl;
}
```

Click here to go back to the quiz.

07 Arrays for Quick and Easy Data Storage

IN THIS LECTURE:

Society today generates massive amounts of data, including records of digital transactions, social media posts, and information from sensors and devices spread everywhere. In order to handle that large amount of data, we need a way to store it. And we need to be able to store not just a few variables, but dozens, or hundreds, or thousands. Fortunately, there's a solution—in fact, more than one solution—for handling as much data as you have. Two methods for storing data are **arrays**, which were part of C, and vectors, which were an improvement of C++.

// STORING VARIABLES IN MEMORY

Let's say that you want to store one variable. You declare a variable, and that creates a "box" with that variable name that's stored in memory. If you want to store a second variable, you declare a second variable, which gets a new box in memory.

If you wanted to allocate many variables—say 20 of them—it would be painful to declare 20 different variables. Instead, it would be easier to say, "I want a set of 20 variables"—and that's what an array gives you.

An array is a collection of some number of variables. All the variables will have the same type, and all will be referred to using the same name, with a separate way of identifying each individual variable.

To declare an array of variables, you still start with the type of variable you want. In **Program 7_1 on page 77**, it's an integer.

You then have the name of the array, which is **test_array**. It wouldn't be practical to name all the variables with different names, so you refer to the entire group—the entire array—by one name.

After the array name, you have a pair of square brackets, which let you specify the size of the array, or the number of elements in the array. In this case, you're setting up an array of size **20**.

In mathematics, vectors and matrices are described using square brackets. Because an array is similar to a vector or matrix in math, it makes sense that square brackets are used for them, too.

In memory, these **20** "boxes" are all set aside in one big block—one block right after another.

After the array has been declared—that is, after that long stretch of boxes has been set aside in memory—you can refer to each individual box as a separate variable. To do this, you will again use square brackets, each containing a number, after the array name. The first such box will be **0**, the second one will be **1**, etc. You refer to a box by the array name and then, in square brackets, the box number.

Why do you start the numbering with 0 instead of 1? The short answer is it's a little more efficient than starting with 1.

In **Program 7_1**, you are assigning values of **10**, **20**, and **25** to the first 3 boxes in the array. Notice that you can assign values to them, and you can refer to the values inside them, just like with other variables.

```
1  // Program 7_1
2  // Declaring an array
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int test_array[20];
8     test_array[0] = 10;
9     test_array[1] = 20;
10    test_array[2] = 25;
11    cout << test_array[0] << " " << test_
   array[1] << " " << test_array[2] << endl;
12 }
```

Generally, the whole array is referred to as the variable. In other words, even though the array is made up of many different "boxes", which you could think of as many different individual variables, you will generally be able to refer to the entire array—the entire collection of boxes—as one thing, so it makes sense to call it a variable.

The individual boxes are called the **elements**, which are the component variables of the overall array variable.

The number identifying an element is called the **index**.

To refer to an element of the array, the common phrase used is ***sub*** (which can be thought of as short for *subscript*) and then the index number.

// INDEXING INTO AN ARRAY

One of the things that makes using arrays feasible is that you can use a variable to represent the individual index.

Suppose you create a small array named **a** with **5** elements, and suppose you assign values of **10**, **20**, **30**, **40**, and **50** to them. Then, you create an integer index variable **i** that you initialize to **3**, and you print out **a[i]**.

```
int main() {
  int a[5];
  a[0]=10; a[1]=20; a[2]=30; a[3]=40; a[4]=50;
  int i = 3;
  cout << a[i] << endl;
}
```

The output is the value that you assigned to **a[3]**, which was **40**.

When you access a single element of the array—by listing the array name and then putting the particular element index in square brackets—you are said to be **indexing into the array**. In other words, you're accessing array elements using the index of an element.

The ability to index into an array by using a variable that contains the index gives you the ability to easily address all the elements of the array by looping through it.

For example, you can set all of the elements to initial values. Suppose you were playing a 4-person game and wanted people to have the same amount of money in the game to begin with—say $200 **(Program 7_3)**.

You would first create an array. You could write **int player_money[4]** (7), which would declare an array named **player_money** with **4** elements.

Then, you would loop over all the indexes (9). In this case, you can use a **for** loop to iterate over the values from **0** to **3**. You use **i** to keep track of the index.

Within that loop, you could then set **player_money[i]** to **200** (10), indicating that that player has $200. Because you loop over all **4** indexes, you have initialized all **4** elements.

Besides initializing, you can loop over the elements to print them (a).

```
1  // Program 7_3
2  // Setting all elements of
   an array
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int player_money[4];
8     int i;
9     for (i = 0; i < 4; i++) {
10       player_money[i] = 200;
11    }
12    // Show output
13    for (i = 0; i < 4; i++) {
14       cout << player_money[i]
   << endl;
15    }                          a
16 }
```

Imagine that you want to read in a series of daily temperatures, report the average, and then list all the days that have a temperature greater than the average.

To develop this program, you're going to need 3 parts, so your pseudocode will just be 3 lines.

- First, you'll need to read in all of the temperature data. Because you don't know how many temperatures you'll have, you won't know the size of the array ahead of time. You'll have to pick a size you think is plenty large. You'll also have to pick a sentinel value to let the program know when you're done entering data. In this case, that would be some temperature that no one would ever enter. When you read in temperatures, you can keep track of a count and a sum.
- Then, you use the count and the sum to compute an average.
- Once you've computed the average, you can go through the temperatures again and print out those that are larger than the average.

PSEUDOCODE

1. Read in temperature data, keeping the count and sum
2. Determine and print out average
3. Print out elements greater than the average

That pseudocode becomes comments, and then you fill in the details.

```
1   // Program 7_4
2   // Reporting temperatures greater than the average
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      // Read in temperature data, keeping sum
8      int temps[1000];
9      int counttemps = 0;
10     int currenttemp = 0;                                   b
11     int total = 0;
12     cout << "Enter a temperature: ";
13     cin >> currenttemp;                                    c
14     while (currenttemp < 200) {                                          d
15        temps[counttemps] = currenttemp;
16        total += currenttemp;
17        counttemps++;                                       e
18        cout << "Enter a temperature. Enter 200 or greater when done: ";
19        cin >> currenttemp;
20     }
21
22     // Determine and print out average
23     float average = (float)total / (float)counttemps;
24     cout << "The average temperature was " << average << endl;
25
26     // Print out elements greater than the average                    f
27     for (int i = 0; i < counttemps; i++) {                  g
28        if (temps[i] > average) {
29           cout << temps[i] << " was above the average temperature." << endl;
30        }
31     }
32  }
```

You begin by declaring an array of integers called **temps** that will hold the temperatures you read in from the user (8). You can make an array of size **1000**, which assumes that the user won't enter more than 1000 temperatures.

When choosing an array size, you want to pick one guaranteed to be big enough to hold all of your data. But you don't want to choose a size that is unnecessarily too large; the bigger the array, the more memory is taken up. If you have arrays with many millions of elements, you can actually exceed the memory in your computer!

You then set up some variables that you'll use in your program (b). The number of valid temperatures you've read in will be **counttemps**, and **currenttemp** will be used to hold the current temperature that you read in. And **total** will hold the combined sum of all temperatures. You'll use that to calculate the average later.

Next, you prompt the user to enter a temperature, which you read in to the variable **currenttemp** (c). Presumably, there will be at least one temperature, so you don't prompt the user for a sentinel value the first time. You then have a **while** loop (d) that will repeat until the user does enter a sentinel value, telling you it's time to stop. Because you're recording temperatures on Earth, you won't have temperatures like 200, so numbers 200 or greater work well as a sentinel.

Each time through the loop, you'll update several variables (e). If you are in the loop, then you had a valid temperature, **currenttemp**, entered. So, you set the next element of the array, **temps**, to this current temperature.

Notice that the array is indexed by the variable **counttemps**, which is initialized to **0**, so the first temperature you have will go into the first array element, **temps[0]**. After storing the temperature there, you add the temperature to the running total and increment **counttemps**. So, **counttemps** will hold the number of valid temperatures read so far. The elements of **temp** that will contain valid temperatures are elements **0** through **counttemps** minus 1.

Finally, still in the loop, you need to read in the next temperature, prompting the user (18). Notice that you instruct the user to enter a sentinel value when finished.

At this point, you have a loop that will read in all of your temperature values into an array, keeping track of the total number of temperatures and the sum.

The next segment of the program is just specifying what to calculate. You calculate the average by dividing the total sum of temperatures by the count of temperatures. Notice that you need to cast either the sum or the count to a float in order to get a floating-point output (23). Finally, you output the average (24).

Your last section of code should go through your array and print out the temperatures that were greater than the average.

You begin by setting up a **for** loop (f) with initialization of the variable, the condition, and the variable update. Notice that you declare a new variable **i** that will iterate through the array values and is initialized to **0** and incremented by 1 on each iteration. The comparison ensures that this continues as long as **i** is less than **counttemps**. So, your loop is **for(int i=0; i<counttemps; i++)**.

Within the loop, you have a simple comparison: You check whether the element indexed by the current value of **i** is greater than the average and output if it was greater (g). Notice that because **i** will take on every valid index of the array, this will let you examine every element of the array.

If you run this program, you are able to enter several temperature values, and then you get an average printed, along with the temperatures that were greater than that average.

// INITIALIZING AN ARRAY

You can also initialize arrays directly.

Program 7_5 has an array declaration that you might use if you were keeping track of numbers of people on various teams in a competition. The array **team_members** is declared to have size **5**. To initialize it, you can specify the initial values that you want the elements to take as a series of 5 values separated by commas and enclosed in curly braces. In this case, you initialize the array to have values **5**, **6**, **4**, **5**, and **4**.

When you run this program, the **for** loop iterates through all the members and the **cout** prints out values for each of the 5 elements in the order you specified them.

```
1   // Program 7_5
2   // Array Initialization
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7       int team_members[5] = { 5, 6, 4, 5, 4 };
8       for (int i = 0; i < 5; i++) {
9           cout << team_members[i] << endl;
10      }
11  }
```

If you have an array of size **5** but only initialize it with 2 values—suppose you don't know the size of the other teams—then the first 2 elements of the array are initialized.

```
7       int team_members[5] = { 5, 6 };
```

When you have array elements that aren't specified, they are often initialized to the value **0** by default. It's usually better to explicitly set any values that you want to be initialized so that you know what's in the memory locations that you might access.

When you print out the values of the first 5 elements, you see that the first 2 elements have the values **5** and **6**, as specified. The last 3 elements all have the value **0**.

Finally, you can declare an array without specifying the size if you initialize it with a set of values. In the code below, the array **my_array** is declared with an indeterminate size, where you don't specify anything in the square brackets. But you initialize it with a set of 5 values. So, this implicitly defines the array to be of size **5**, and you see that the values are all set correctly.

```
7       int my_array[] = { 5, 6, 4, 5, 4 };
```

On the other hand, if you just tried to declare an array with no size, you'd get a compiler error. Only when the array is initialized to something with a known size is it OK to leave the square brackets empty.

Data is often more than just a single value, and you often want to keep track of multiple types of data at the same time, where the different data values are all tied together.

The method of doing this is called **parallel arrays**. You create multiple arrays, each one holding a different type of data. And you tie these together by making sure that the same index in each array refers to the same item. So, if you have an array to store low temperature, and another to store high temperature, then element 20 in both arrays refers to the same day. You can get the low and high for that day just by using that index.

```
1    // Program 7_8
2    // Parallel Arrays
3    #include <iostream>
4    using namespace std;
5
6    int main() {
7       int year[100];
8       int month[100];
9       int day[100];
10
11      //January 1, 2000 in
     element 0
12      year[0] = 2000;
     month[0] = 1; day[0] = 1;
13      //February 12, 2007 in
     element 1
14      year[1] = 2007;
     month[1] = 2; day[1] = 12;
15   }
```

In **Program 7_8**, you have 3 parallel arrays, which are used to store a date. The 3 arrays will store a year, a month, and a day. You set the arrays to size **100**, meaning you can store up to 100 different dates. So, you can assign a date by assigning the day, month, and year to the elements of the 3 arrays, assuming they have the same index.

You can also have an array of arrays, or **multidimensional arrays**. You can declare a 2-dimensional array by declaring a variable name with 2 sets of brackets, each with a value inside.

Suppose you wanted to keep track of 5 people, each of whom had 3 bank accounts: a checking account, a savings account, and a credit card account. So, you would want an array that is **5** elements long. Each of those elements would be an array of length **3**. So, overall, you want a 2-dimensional array that is 5 by 3. You could declare this by the statement **double accounts[5][3]** as in **Program 7_9**. The name of the 2-dimensional array is **accounts**.

To access some particular person's account, you would access an element—by using the name of the account followed by 2 numbers, each in square brackets. For example, the fourth person's second account would be **accounts[3][1]**, and in this code, you assign a balance of $1000 there (9). Remember that you start counting from **0**, so the **3** indicates that it's the fourth person, and the **1** indicates that it's the second account. To set the value of the first person's third account to $50, you'd write **accounts[0][2] = 50.0** (10).

```
1    // Program 7_9
2    // A 2D array uninitialized
3    #include <iostream>
4    using namespace std;
5
6    int main() {
7       double accounts[5][3];
8
9       accounts[3][1] = 1000.0;
10      accounts[0][2] = 50.0;
11
12      int i, j;
13
14      for (i = 0; i < 5; i++) {
15          for (j = 0; j < 3; j++) {
16              cout << "Person "
     << i << ", account " << j << "
     has balance " << accounts[i][j]
     << endl;
17          }
18      }
19   }
```

h

Any time you want to access an element in this array, you should specify 2 values, each in brackets. The first element should be a value from **0** to **4**; the second element should be a value from **0** to **2**.

You can use nested loops to loop over all individuals and then for each one, over all accounts, to see the results (h).

If you do that, you see some values that you set as intended—the $50 and $1000 you just set, but there are many other random values. Remember, you never initialized those, so the information in them could be almost anything.

But you can add one more set of nested loops at the beginning of your code (i), just after declaring the array, to go through all individuals and all accounts for each individual and initialize them to **0**. Now when you run the code, all the balances come out as expected.

Using 2-dimensional arrays can be useful if you have data that comes in on a 2- dimensional grid. But multidimensional arrays can also be useful when you have something that is easy to index in 2 different ways.

```
1   // Program 7_10
2   // A 2D array initialized
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      double accounts[5][3];
8
9      int i, j;
10
11     for (i = 0; i < 5; i++) {
12         for (j = 0; j < 3; j++) {
13             accounts[i][j] = 0.0;
14         }
15     }
16
17     accounts[3][1] = 1000.0;
18     accounts[0][2] = 50.0;
19
20
21     for (i = 0; i < 5; i++) {
22         for (j = 0; j < 3; j++) {
23             cout << "Person " <<
    i << ", account " << j << " has
    balance "
24                 << accounts[i][j]
    << endl;
25         }
26     }
27  }
```

i

// ARRAY BOUNDS

Regardless of whether you use 1-dimensional or multidimensional arrays, there is one issue to be particularly careful of: array bounds.

```
1   // Program 7_11
2   // ERROR in output due to array
    out of bounds
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      int my_array[3];
8      my_array[0] = 10;
9      my_array[1] = 20;
10     my_array[2] = 30;
11     for (int i = 0; i < 4; i++) {
12         cout << my_array[i] << endl;
13     }
14  }
```

In **Program 7_11**, you have an array named **my_array** of size **3**, and you set the values of the 3 elements in the array to **10**, **20**, and **30**.

You have a loop that's designed to print out the values of the array, but instead of outputting 3 values, you make a mistake and have the loop try to print out 4 values instead. If you look at the **for** loop, the index **i** will take on values **0**, **1**, **2**, and **3**, because the condition is **i<4**.

Because **0**, **1**, and **2** are part of the array and you just set values for each, they should be fine. But what happens when you try to print **my_array[3]**?

When you run this, you might expect to encounter a compiler error or some other error at the beginning telling you that you're accessing an array element that's not there. Some compilers might be able to catch this and give a warning message, but this isn't what usually happens!

If you're lucky, the program will just print out some value like 0 when the fourth element is printed. But what actually happens is not guaranteed. You could get almost any value printed out; in fact, you might find numerous lines output at once. The program might just crash completely!

This is called an **array out-of-bounds error**.

Unfortunately, there's no simple fix for this. Array out-of-bounds errors are one of the more common sources of bugs in C and C++ programming. Vectors are an alternative to arrays that have a way to help avoid this error. Vectors are the subject of the next lecture. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, section 18.6.

b Lippman, Lajoie, and Moo, *C++ Primer*, section 3.5.

// QUIZ

1 What would be the command to do each of the following?

a Declare an array named `x` of `5` integers.

b Assign the value `10` to the 3rd element of an array `x`.

c Print the first element of the array `x`.

d Create an array, `x`, initialized with 5 elements: `10`, `20`, `30`, `40`, and `50`.

e Create a 2-dimensional array, `x`, of floating-point values with `4` rows and `6` columns.

2 Write a program that reads in `5` numbers then prints them out in reverse order.

3 Assume that you have a set of products and want to find which products have a rating that is better than the average rating. Write a program that reads in a product ID (an integer) and a rating (a floating-point number) for no more than `100` products, stopping when a product ID of `0` is read in and not counting this product. Then, print out the IDs of all products with an above-average rating.

Click here to see the answers.

// QUIZ ANSWERS

1 a `int x[5];`

b `x[2] = 10;`

Remember that array indexes start with 0, not 1. So, the 3rd element is `x[2];`.

c `cout << x[0] << endl;`

d `int x[] = {10, 20, 30, 40, 50};`

Note that you could have written `x[5]` instead of `x[]` and had the same result.

e `float x[4][6];`

2 Although there are several ways this could be done, the most straightforward is to read the numbers into an array of **5** integers and then loop through the array in reverse order:

```
#include <iostream>
using namespace std;
int main()
{
    int nums[5];
    int i;
    cout << "Enter 5 integers:" << endl;
    for(i=0;i<5;i++) {
        cin >> nums[i];
    }
    cout << "The numbers in reverse order are:"
<< endl;
    for(i=4;i>=0;i--) {
        cout << nums[i] << endl;
    }
}
```

3 This will use parallel arrays. You'll use one array for storing the product IDs and one for storing the rating. You will compute the average rating as you read in the data. Then, you will go through the list and print the ID for the products with a rating that is greater than the average:

```
#include <iostream>
using namespace std;
int main()
{
    int ID[100];
    float rating[100];
    int totalnumber = 0;
    float sumofratings = 0.0;
    int i;
    cout << "Enter a product ID and a rating,
entering an ID of 0 to stop" << endl;
    int id;
    float r;
    cin >> id >> r;
    while(id > 0) { // Repeat until ID 0 is read in
        ID[totalnumber] = id; // Store the product ID
        rating[totalnumber] = r; // Store the
product rating
        totalnumber++; // Increase count by 1
        sumofratings += r; // Sum up ratings
        cin >> id >> r; // Read in next product
    }
    float averagerating = sumofratings/totalnumber;
    cout << "The products with above average ratings
are:" << endl;
    for(i=0;i<totalnumber;i++) {
        if (rating[i] > averagerating) {
            cout << ID[i] << endl;
        }
    }
}
```

Click here to go back to the quiz.

08 Vectors for Safe and Flexible Data Storage

As you've learned, arrays are one way of handling large amounts of data. They can be efficient, but they have some issues. First, there is the possibility of arrays going out of bounds. Also, you need to know the size of the array at the time you declare it; this is a problem if you don't know how much space you need. As an alternative to arrays, C++ offers **vectors**, which address many of the annoying things about arrays. But vectors also offer a sampling of some of the other features that make C++ such a useful language, including object-oriented programming and `templates`.

IN THIS LECTURE:

// USING VECTORS

Program 8_2 on the following page creates a vector with 3 elements, which have the values **1**, **2**, and **3**, and then prints those values out.

Note that a C++ vector is not a mathematical vector.

In terms of how vectors are designed, they are, at their base, still arrays. So, the way you think of arrays in memory and how you access elements still applies, and much of the same terminology tends to be used for both vectors and arrays.

This would have been easy to do with an array, too, but this program illustrates some similarities and differences between arrays and vectors.

Notice that to use a vector, you have to #include the vector library—which is part of C++'s Standard Template Library and is included in all C++ installations. This will give you access to what is called the vector class; it is going to let you make vectors, not just arrays.

A class can be thought of as a new type—much like an integer or floating point. For now, you will use classes that are defined in the standard C++ libraries. In this case, that's the **vector** library, which defines the vector class.

To declare a vector, the line to use is **vector**; then the type, integer (**int**), inside of angle brackets (**<>**); and then a name (in this case, **Victor**) (8).

When talking about a vector, you usually say "vector of" some type, such as "vector of ints" or "vector of floats." And these can be thought of as the type of the variable.

```
1   // Program 8_2
2   // Example of a vector
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8      vector<int> Victor;
9      Victor.push_back(1);
10     Victor.push_back(2);
11     Victor.push_back(3);
12     cout << Victor[0] << " " <<
    Victor[1] << " " << Victor[2]
    << endl;
13  }
```

a

Declaring things with brackets like this is the way to handle what's called generic programming in C++. It's an example of a **template**. The way to think of templates is that you can have a very general structure—in this case, a vector—that is a template that can take on multiple different types. When you want a specific version of the vector, you need to specify that it's a vector of some type, such as a vector of ints (integers).

Unlike with an array, you don't specify a size for vectors. One of the interesting features of vectors is that they can grow in size as you need them to. Initially, when you first declare a vector like this, it'll have no elements in it. In this example, you add elements to the vector and it will increase in size as you do so.

The next 3 lines (a) show one of the most common ways of adding data to a vector. You use a function called **push_back** to add a new element at the end of the vector (in other words, it gets pushed onto the back of the vector). And that element will have whatever value is the argument.

When you get to the first line, you'll add a **1** onto the end of the vector. Because the vector started out with no elements, after that line, it will have 1 element, which will have the value **1**. The second line will add an element with value **2** onto the end of the vector. And the third **push_back** will add a third element to the vector—this one with a value of **3**.

Exercise 1

How would you declare a vector of floats named **temperatures**? Also, what do you have to #include at the top of the program?

Click here to see the solution.

Notice that there's something different about these function calls. You have the name of the vector, which is **Victor**, and then a period, and then the function call. You do not just have the function call all by itself; the name of the vector comes before the function call.

This is an example of using a **member function**—in this case, **push_back**—which "belongs" to the particular variable. The period is used to say, "The second thing is part of the first thing." Here, the **push_back** function belongs to **Victor**; the function is part of the vector named **Victor**. And this **push_back** function is going to push the value at the back of the vector, and nowhere else.

The last line in the program is an output line. You'll stream several values to **cout**: **Victor[0]**, **Victor[1]**, and **Victor[2]**, separated by spaces. Notice that you are addressing the elements of the vector using the same notation as you had for an array: an index you want in square brackets.

```cpp
// Program 8_6
// Vector size
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;
    cout << "Initial size is: " << v.size() << endl;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    cout << "Later size is: " << v.size() << endl;
}
```

You can use vectors very much like arrays. You can assign values to individual elements, access the elements using variable indexes, and so on.

Because vectors can change in size as you add on additional elements, you would like to have a way to find out the size of the vector at any given time. Fortunately, there's a function to find the size of a vector.

Let's declare a vector of ints whose name is **v**. Like the **push_back** function for adding elements to the end of a vector, the function for finding the size of a vector is a member function of the vector class—that is, it "belongs" to a vector. That means that to use it, you need to write the name of the vector, then a period, and then the name of the function. In this case, the function name is **size**, and it doesn't take any arguments.

You now have 2 output lines in which you output the initial size and the later size of the vector. In each case, to get the size of the vector **v**, you write **v.size()**. This will be the size of the vector.

Your first output command occurs right after declaring the vector. So, you get an output saying that the initial size is **0**. The second output command comes after the 3 **push_back** commands. The size of the vector has increased with each of the **push_back** commands, and when you output the size, you indeed find that the later size is **3**.

Exercise 2

Suppose you have a vector named ages and want to read in from a user a bunch of ages until the user enters a sentinel value, such as a negative number, to end the program. What would the code look like for this?

Click here to see the solution.

You can use the size to loop through all the elements of a vector, in the same way you would loop through the elements of an array.

In **Program 8_7**, you declare a vector of floats (floating-point values) and then **push_back** 3 different floating-point values (b). So, you should now have a vector of size **3**.

To output the contents of the vector, you create a **for** loop (c). You use an index **i** that will be initialized to **0** and go as long as it is less than the size of the vector. So, in this case, because the vector is size **3**, it should take on 3 values—specifically, **0**, **1**, and **2**.

For each value, you output the element of the vector, just like you would have done for an array. You write **cout << v[i] << " "** . This will output the element **v[i]**, followed by a space. Notice that you do not output an **endl**, so all elements will appear on the same line. Only at the end do you stream out an **endl** to end the line of output (15).

The output shows 3 values: the 3 values you pushed back into the vector, in the same order you pushed them.

```
1   // Program 8_7
2   // Printing all vector
    contents using size
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8       vector<float> v;
9       v.push_back(10);
10      v.push_back(20.0);
11      v.push_back(0.05);
12      for (int i = 0; i <
    v.size(); i++) {
13          cout << v[i] << " ";
14      }
15      cout << endl;
16  }
```

b

c

// VECTOR SIZE INITIALIZATION

```
1   // Program 8_8
2   // Vector size initialization
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8       vector<int> v(3);
9       cout << "Initial vector : ";
10      for (int i = 0; i < v.size(); i++) {
11          cout << v[i] << " ";
12      }
13      cout << endl;
14  }
```

Although the size of vectors can change, you can also specify a vector's initial size at the time of creation, like you were required to do for an array. To do this, you put the size you want the vector to have in parentheses right after the variable name in the declaration. Note that you use parentheses here, not square brackets, like you do for an array.

So, if you want a vector of floats named **v** of size **3**, you can declare it by writing **vector<int> v(3)** (8). This will create a vector of ints with 3 elements, which will all be initialized to **0**.

Right after the declaration, you have a set of output statements that will print the elements of the vector: There's an initial output line indicating that you're printing the initial vector (9); then a loop through all the elements of the vector (10), printing each element, separated by a space (11); and, finally, a **cout** statement ending the line of output (13). Notice that you are using the vector's size in the **for** loop by calling **v.size()**.

When you run **Program 8_8**, you see that, indeed, the vector has 3 elements and they are all initialized to the value **0**.

```
1   // Program 8_9
2   // Vector initialization
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8       vector<int> v = { 1, 2, 3 };
9       cout << "Initial vector : ";
10      for (int i = 0; i < v.size(); i++) {
11          cout << v[i] << " ";
12      }
13      cout << endl;
14  }
```

Recall that you could initialize an array by specifying an initial set of element values enclosed in curly braces. You can do the same thing with vectors when you declare them. After the declaration, you write = and then the values you want in curly braces, and now the vector is initialized with those values.

In **Program 8_9**, you declare a vector of ints named **v** and then initialize it by adding **= {1, 2, 3}**. This will cause the vector to have 3 elements, valued **1**, **2**, and **3**. Notice that you do not have to set the size of the vector explicitly; it will get the correct size when it gets the initialization. When you print out the vector, you see that, indeed, it has 3 elements, with values **1**, **2**, and **3**.

Vectors have another method that can be used to initialize them. Imagine that you wanted a large vector with 1000 elements and you wanted all of them to have some initial value, say 10. One option would be to write curly braces with 1000 **10**s inside it. That would be a huge pain to type!

For vectors, though, you have a way of initializing all of the elements to some particular value. To do this, when you declare the vector, you use parentheses after the variable name, then the number of elements, followed by a comma, and then the value that you want each of the elements to have:

```
vector<int> v(3, 10);
```

This example shows you creating a vector of size **3** with all elements initialized to **10**. Outputting the vector verifies that this, indeed, works as expected.

The following line declares a vector of floats named **accounts** with starting size **4**:

```
vector<float> accounts(4);
```

If you wanted that vector to be initialized with a balance of $100 in each account, you would change the initialization by writing the following line:

```
vector<float> accounts(4, 100.0);
```

Now you have a 2-argument version with 2 values in the parentheses. The first is still the size, **4**, and the second is the value to initialize everything to, **100**.

This is called a **constructor** function—a function that's called to initialize a variable when it's first declared. Because it's a function, there are parentheses and possibly arguments. For now, just think of the constructors as providing a nice way to initialize your vectors.

// VECTOR RESIZING

One of the nice features of vectors is that they can change in size. You've seen one way of doing that—by using the **push_back** member function. When you use that function, you add an element onto the back of the vector, thus increasing its size by 1.

Another way you can resize a vector is with a member function named **resize**. To use **resize**, you use the vector name, followed by a period, followed by a call to **resize**. You can give resize one argument, which is the new size that you want the vector to be.

When you call **resize**, if the new size is larger than the old one, then new elements are created and they are initialized with the value **0**. If the new size is smaller than the old one, then the last elements of the vector are removed.

Program 8_11 is an example of resizing a vector. After declaring a vector with the name **v**, you call **resize** on that vector, with an argument of **5**. This will cause the vector to have a size of **5** instead of **0**. And when you print out the vector elements after resizing, you see 5 elements, each with the value **0**.

```
1   // Program 8_11
2   // Vector resizing
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8      vector<int> v;
9      v.resize(5);
10     cout << "Resized vector : ";
11     for (int i = 0; i <
    v.size(); i++) {
12        cout << v[i] << " ";
13     }
14     cout << endl;
15  }
```

There's even another version of the resize function, one that takes 2 arguments **(Program 8_12)**. Just like the initializing constructor can take 1 argument, specifying just size, or 2 arguments, with the first specifying size and the second specifying an initial value for the elements, you have the same thing here.

This example has 2 parts:

- First, you create a vector with an initial size of **3**. Because you did not specify any initial values, that vector of size **3** will have 3 elements, each of which has the value **0**.
- Next, you have a command, **v.resize(5,1)**, that will resize the vector to be size **5**. And the new elements of the vector will be initialized with the value **1**. Because the original vector was of size **3**, resizing the vector will add 2 more elements onto it. Those 2 new elements—and just those 2—will be initialized with the new value **1**. So, when you print out the vector's elements, you see 5 elements, the first 3 having the value **0** and the last 2 having the value **1**.

```
1   // Program 8_12
2   // Vector resizing and
    initializing
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8      vector<int> v(3);
9      v.resize(5, 1);
10     cout << "Resized vector : ";
11     for (int i = 0; i <
    v.size(); i++) {
12        cout << v[i] << " ";
13     }
14     cout << endl;
15  }
```

// PERFORMING OUT-OF-BOUNDS CHECKS

In terms of addressing individual elements inside the overall data structure, vectors can be used just like arrays. You've seen several examples already where square brackets are used to access individual elements. In all of the loops that print out array elements in the last several examples, you had elements accessed with square brackets.

Accessing elements of a vector by using square brackets works just like with arrays—for good and bad. On the good side, it's very efficient, and you can access the next element directly. Because all the data is allocated in one continuous block of memory, it's easy for the computer to process it efficiently.

```
1    // Program 8_13
2    // Vector ERROR indexing out
     of bounds
3    #include <iostream>
4    #include <vector>
5    using namespace std;
6
7    int main() {
8       vector<int> v = { 1, 2, 3 };
9       for (int i = 0; i < 5; i++) {
10          cout << v[i] << " ";
11      }
12      cout << endl;
13   }
```

But there is a problem you have to be careful to avoid when accessing array elements: the array out-of-bounds error, which occurs when you try to access elements that have not been allocated to the array or vector.

In **Program 8_13**, you have a vector that's been initialized with 3 elements, having the values **1**, **2**, and **3**. However, in the loop that you use to print out the array elements, you are printing out 5 elements! That means you're accessing **v[3]** and **v[4]**, which don't really exist.

Just like with arrays, the code lets you do this; there's nothing stopping you from doing it. But when you run the code, who knows what results you'll get. In general, these nonexistent elements will have whatever value was last left in that memory location, and you have no guarantee what that is!

But with vectors, there is a way to avoid this error from ever occurring: Instead of accessing elements using square brackets, you can use the member function **at**.

To use this member function, you use the vector name, followed by a period, then the function call **at**, and then the number of the element you want to access in parentheses. This is basically the same code from before, but now you are using **.at** and parentheses, rather than square brackets, to access the element.

When you use **at** instead of square brackets, it will automatically check the bounds of the array. If you try to access an element that should not exist, you get an exception, which is an error code. And when you have an exception in a situation like this, the program will just stop. It won't access memory that it shouldn't, won't let you set values in memory that it shouldn't, and won't let you print out memory values that contain nonsense.

```
7    int main() {
8       vector<int> v = { 1, 2, 3 };
8       for (int i = 0; i < 5; i++) {
10          cout << v.at(i) << " ";
11      }
12      cout << endl;
13      cout << "Made it here!"
     << endl;
```

When you run this code, you get an output of just 3 elements. You have your **for** loop, in which you try to print out each element, from **0** to **4**, from a vector of size **3**. Printing out elements **0**, **1**, and **2** is fine. However, when you try to print out **v[3]**, that will be out of bounds. Because you're using the **at** function instead of square brackets, the program will stop at this point. It will exit with an exception.

It might seem weird that a program exiting is a better result than printing random data. But when you're accessing random data, who knows what will happen; instead, by using the **at** function, you guarantee a clean exit from the program. And there are ways of handling exceptions in the program so that it won't crash.

You can also use the **at** command to assign values to vector elements, just like you could with square brackets.

In **Program 8_15**, you first create a vector **v** with 5 elements, with values **1** through **5**. Using square brackets, you set the second element, **v[1]**, to **100** (9). Then, you set the fourth element, **v[3]**, to **500**, using the **at** command (10). This gives you access to the particular element of the vector, letting you assign the value there.

When you output the result of this, you see that the second and fourth elements of the vector have indeed been modified. The **at** function worked just like the square brackets. And if you had tried to set a value for an element that wasn't part of the vector, the **at** command would have thrown an exception, just like you saw when you were just printing.

Although **at** is certainly a safer way to access elements of a vector, it does have a downside: Every time you try to access an element with **at**, the computer is going to check the bounds of the array to see if you're trying to access something you shouldn't, which means that every access to an array element is going to be a little slower.

For most programmers, this is not a big deal; the program runs plenty fast, and the microseconds taken to check a value are insignificant. But for applications where performance is critical—where there are going to be millions of attempts to access array or vector elements—checking the bounds each time could be a significant factor. So, the usage of square brackets versus **at** is not consistent.

```
1  // Program 8_15
2  // Assigning elements with brackets and "at"
3  #include <iostream>
4  #include <vector>
5  using namespace std;
6
7  int main() {
8     vector<int> v = { 1, 2, 3, 4, 5 };
9     v[1] = 100;
10    v.at(3) = 500;
11    for (int i = 0; i < v.size(); i++) {
12       cout << v.at(i) << " ";
13    }
14    cout << endl;
15    cout << "Made it here!" << endl;
16 }
```

If you're interested in learning more about vectors, check out http://www.cplusplus.com/reference/vector/vector/.

// ASSIGNING VECTORS

Another feature that vectors offer that's not part of arrays is the ability to copy them.

In **Program 8_16_a**, you create 2 integer arrays of size **3** named **a** and **b**. In this case, **a** is initialized with values **1**, **2**, and **3**.

You might like to be able to make **b** have the same values as **a**—to essentially copy **a** into **b**. However, if you try to write a line **b = a**, you'll get a compilation error. You are not allowed to assign arrays like that.

Instead, if you use vectors, this works out fine **(Program 8_16)**. You create 2 vectors of ints **a** and **b**, and you initialize **a** to have 3 elements, with values **1**, **2**, and **3**. Then, you have a command, **b = a**, which copies **a** to **b**. Now **b** has 3 elements itself, and the values of the elements are the same as those of **a**. When you print out the elements of the vector **b**, you see that they have the same values as **a**.

Going forward, you'll be relying on vectors as your data structure of choice for storing large amounts of data. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 4.6, 17.2, 17.3, and 18.5.

b Lippman, Lajoie, and Moo, *C++ Primer*, section 3.3.

```
1   // Program 8_16_a
2   // ERROR - Attempting to assign arrays is not allowed
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      int a[3] = { 1, 2, 3 };
8      int b[3];
9      b = a;
10  }
```

```
1   // Program 8_16
2   // Assigning vectors
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   int main() {
8      vector<int> a = { 1, 2, 3 };
9      vector<int> b;
10     b = a;
11     for (int i = 0; i < b.size(); i++) {
12         cout << b[i] << " ";
13     }
14     cout << endl;
15  }
```

Exercise 1 Solution

```
#include<vector>

...

vector<float> temperatures;
```

Click here to go back to the exercise.

Exercise 2 Solution

```
1   // Program 8_5
2   // Reading in values and adding on to a vector indefinitely
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6   int main() {
7       vector <int> ages;
8       int age;
9       cout << "Enter someone's age. Enter a negative age to stop: ";
10      cin >> age;
11      while (age >= 0) {
12          ages.push_back(age);
13          cout << "Enter another age. Enter a negative age to stop: ";
14          cin >> age;
15      }
16  }
```

Click here to go back to the exercise.

// QUIZ

1 What would be the command to do each of the following?

a Create a vector named `v` of integers.

b Assign the value `10` to the 3rd element of a vector `v` (think of 2 ways to do this).

c Initialize a vector `v` of floating-point values to have `10` elements, each with the value `1.0`.

d Add an element with value `10` to the end of a vector `v`.

e Assign the number of elements of a vector `v` to a variable `s`.

For the next 2 questions, compare these to the solutions you obtained in the equivalent exercises for the previous lecture.

2 Write a program to read in positive integers and then output them in reverse order. You should allow *any number* of integers to be read in.

3 Assume that you have a set of products and want to find which products have a rating that is better than the average rating. Write a program that reads in a product ID (an integer) and a rating (a floating-point number) for *any number* of products, stopping when a product ID of `0` is read in and not counting this product. Then, print out the IDs of all products with an above-average rating.

Click here to see the answers.

// QUIZ ANSWERS

1 a `vector<int> v;`

b There are 2 ways to do this:

```
v[2] = 10;
v.at(2) = 10;
```

c `vector<float> v(10, 1.0);`

d `v.push_back(10);`

e `s = v.size();`

2 You will use a vector to store the integers you read in. Because you want positive integers, you can use a negative integer as a sentinel value, telling you when to stop reading in. By using a vector instead of an array, you can add as many elements as you wish, and you do this using the **push_back** command rather than reading directly into an array element. To output, you will go through the vector in reverse order, starting from element number **n-1**, where **n** is the size of the vector; the output loop is identical to how you would output an array in reverse order.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> values;
    cout << "Enter positive integers, entering a negative value to stop: "
<< endl;
    int v;
    cin >> v;
    while (v > 0) {
        values.push_back(v);
        cin >> v;
    }
    cout << "The numbers in reverse order are:" << endl;
    int n = values.size();
    int i;
    for(i=n-1;i>=0;i--) {
        cout << values[i] << endl;
    }
}
```

3 You will use parallel vectors to store the product ID and the rating, similar to how you would handle an array. However, because you have vectors, you can add as many elements as you wish to the vector using **push_back**; you do not need to declare an array of a particular size and read directly into array elements. You will compute the average rating by summing as you construct the vector and then dividing by the total number (which you can get by using the **size** command rather than having to store a value as you read in data). Finally, you loop through the vector and output as you compare to the average; this final step is identical to how you would handle the problem with an array.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> ID;
    vector<float> rating;
    float sumofratings = 0.0;
    int i;
    cout << "Enter a product ID and a rating, entering an ID of 0 to stop" << endl;
    int id;
    float r;
    cin >> id >> r;
    while(id > 0) { // Repeat until ID 0 is read in
        ID.push_back(id); // Store the product ID
        rating.push_back(r); // Store the product rating
        sumofratings += r; // Sum up ratings
        cin >> id >> r; // Read in next product
    }
    int totalnumber = ID.size();
    float averagerating = sumofratings/totalnumber;
    cout << "The products with above average ratings are:" << endl;
    for(i=0;i<totalnumber;i++) {
        if (rating[i] > averagerating) {
            cout << ID[i] << endl;
        }
    }
}
```

Click here to go back to the quiz.

09 C++ Strings for Manipulating Text

IN THIS LECTURE:

Although computers were originally developed to deal mainly with numbers, it quickly became obvious that representing text, including letters and punctuation, was just as important. You've been using text for output in many of your **cout** statements, and you've seen ways of putting text in double quotes and then streaming that text to the console. The term for text like this—a collection of letters, numbers, punctuation, spaces, etc., all strung together—is a **string**.

// STRING VARIABLES AND LITERALS

Up to now, you've been using only string **literals**—specific fixed values of some type. Just like you can have variables to let you store and manipulate numbers or Boolean values, you also would like to have variables to let you store and manipulate strings. To do this, you use a new type of variable—the string type. String variables are the main way you process and manipulate text data.

Unlike **int**, **float**, and **bool**, a string is not a built-in language type. To have access to a string type, you're going to bring in a new library, the **string** library. Once you've done that, you can declare string variables, just like you declare other variable types.

Think of a literal as being in contrast to a variable. A variable can potentially be any value of a certain type; a literal is some specific value.

If you have a variable **x** and a line of code like **x = 5**, **x** is a variable while **5** is a literal. If you stream **3.14** to **cout**, you're streaming a literal floating-point value. Or, if you assign **3.14** to a floating-point variable **f**, then when you print **f**, you're streaming a variable.

The **string** library is one of the standard C++ libraries, so to have access to it, you simply #include **string**. And everything in **string** falls in the standard namespace, so writing **using namespace std** at the beginning means you don't have to list the namespace explicitly.

Program 9_1 is a different variation on the **Hello, World!** program. Because you've #included **string**, you can declare variables to be of type string. In this example, you have declared a variable of type string named **greeting**. You then assign **greeting** a value, which in this case is the string **Hello, World!**.

```
1   // Program 9_1
2   // Declaring and printing
    a string
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int main()
8   {
9       string greeting;
10      greeting = "Hello, World!";
11      cout << greeting << endl;
12  }
```

Finally, you output **greeting**, streaming the value of the string variable to **cout**, just like you would have streamed out a string literal.

When you run the program, you see **Hello, World!** printed out, just like you did in the original version.

You can stream input into a string, just like you streamed a string to output; that is, you can use **cin** to get input from the console and stream that input into a string variable.

Program 9_2 is a simple greeting program. You #include the **string** library, and you declare a variable of type string named **username**.

After prompting for the user's name by printing **What's your name?**, you then read the user's response into the string variable, **username**. The syntax is just like streaming other variable inputs: **cin >> username;**. You can then stream the string as part of an output statement so that you can say **Howdy** back to the user, using the user's own name.

If you run this code and type in **John**, you get the message **Howdy, John!**—just what you'd hope for.

On the other hand, if you type in a full name, **John Keyser**, you still get the same response: **Howdy, John!**. You don't get **Howdy, John Keyser!**. But you know that strings can contain spaces, so why didn't you get the full name stored in the string variable here?

When you stream input into a string, the stream will just take the next individual item entered and put that item's data into the variable. Individual items are considered to be the text elements separated by white space—that is, spaces, tabs, and line breaks. So, when you typed in **John Keyser**, the streaming operation saw that as 2 different strings: one with the value **John** and one with the value **Keyser**. Only the first string, **John**, got streamed in to the variable.

```
1   // Program 9_2
2   // A greeting program
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int main()
8   {
9       string username;
10      cout << "What's your
    name? ";
11      cin >> username;
12      cout << "Howdy, " <<
    username << "!" << endl;
13  }
```

This variation on the program is designed to handle both first and last names. You have 2 string variables declared: **firstname** and **lastname**. You also adjust your prompt to ask for both first and last names.

Then, you stream in to both variables, **firstname** and **lastname**. This means that the first string entered will go into the first variable, **firstname**, and the next string entered will go into the second variable, **lastname**.

The output statement is also changed. Notice that in the streaming output, you have to explicitly output a space between the first name and last name; otherwise, they would run together.

When you run the program now, if you enter the name **John Keyser**, you get the complete greeting, **Howdy, John Keyser!**, back from the computer.

```
1   // Program 9_3
2   // A greeting program handling first and last names
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int main()
8   {
9       string firstname, lastname;
10      cout << "What's your name? (Enter first name and last name) ";
11      cin >> firstname >> lastname;
12      cout << "Howdy, " << firstname << " " << lastname << "!" << endl;
13  }
```

Exercise 1

Suppose you want to write a program that asks a user for his or her name and favorite color and reads that information in. Then, a reply is printed, telling the person that the computer likes that color, too, using both the person's name and favorite color in the output. How might you program this?

Click here to see the solution.

You can stream in data with **cin**, but what if you want to read in a longer piece of text, such as a sentence with spaces and punctuation? There is a function called **getline** that is defined within the **string** library that lets you do this.

> Remember that you can look up what is in the **string** library through online references like cplusplus.com.

To use **getline**, you make a function call: You use the function name followed by parentheses, which will contain the arguments to the function. For **getline**, we want to use 2 arguments: the source of the input (the console input **cin**) and the identifier of the variable that you want to hold the result of the function call (a variable of type string).

Let's modify the **Hello, World!** program using **getline**.

In this code, all the commands are identical to the ones from before, except the input command. Rather than streaming from **cin** to **username**, you're instead using the **getline** function. You write the command **getline(cin, username);**, which will read an entire line of input into the string **username**.

If you run this program and enter **John Keyser** as the name, the output will include the entire name, writing **Howdy, John Keyser!**.

```
1  // Program 9_5
2  // Using getline
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main() {
8     string username;
9     cout << "What is your name? ";
10    getline(cin, username);
11    cout << "Howdy, " << username << "!" << endl;
12 }
```

Exercise 2

a) How would you declare a string named **favoritefood**?

b) How would you then assign a value **pizza** to that variable?

c) How would you stream in the string from input?

d) How would you instead read an entire line into the string?

e) How would you stream that string to output?

Click here to see the solution.

// STRING OPERATIONS

In **Program 9_6**, you have 3 strings—named **s1**, **s2**, and **s3**—and you assign the string **Happy** to **s1** and the string **Birthday** to **s2**.

Then, you have a line: **s3 = s1+s2;**. In other words, you're adding together 2 strings to get a third string.

If you run this program, outputting **s3** gives you a string **HappyBirthday**, with no space between the words. Basically, the string **s3** was formed by taking the first string, **Happy**, and sticking the next string, **Birthday**, onto the end. This is called **concatenation**. You have concatenated the strings **s1** and **s2** to form **s3**.

```
1    // Program 9_6
2    // Adding strings
3    #include <iostream>
4    #include <string>
5    using namespace std;
6
7    int main()
8    {
9        string s1, s2, s3;
10       s1 = "Happy";
11       s2 = "Birthday";
12       s3 = s1 + s2;
13       cout << s3 << endl;
14   }
```

This means that the addition operation, **+**, works differently for 2 strings than it does for 2 numbers. The **+** operator is **overloaded**, which means that it has more than one meaning, depending on the types of the input.

Operator overloading is a common practice for defining behavior between different types.

> A **+** does not always mean addition. The meaning of various operators changes depending on the type of thing they are operating on.

In another variation on the previous program, instead of the strings **Happy** and **Birthday**, you assign **s1** the string **3** and **s2** the string **4**.

When you add these 2 together, the output is **34**. Why is that?

```
10       s1 = "3";
11       s2 = "4";
12       s3 = s1 + s2;
```

Keep in mind that **s1** and **s2** are strings. The **3** that is assigned to **s1** is not the number **3**; it is a string that happens to have just a single-character digit, **3**, inside of it. Likewise, **s2** is assigned the string **4**. When you add together strings, you get concatenation, so the result of adding **s1** and **s2** is to concatenate their strings. So, the result is not really the number **34**; it's the string **34**.

In **Program 9_8**, you have a string, **s**, that you initialize to the value **Happy**. You then have a line: **s += "Birthday";**.

When you run this, you'll get a concatenated string, **HappyBirthday**, with no space in between. The string **Birthday** has been **appended** onto the original string, which was **Happy**. Basically, you took the value already in the string and performed a **+** operation with the right side, putting the result back in the original variable. But **+** for strings is defined as concatenation, so the 2 strings were concatenated together and the result was placed back in **s**.

```
1    // Program 9_8
2    // Appending a string
3    #include <iostream>
4    #include <string>
5    using namespace std;
6
7    int main()
8    {
9        string s = "Happy";
10       s += "Birthday";
11       cout << s << endl;
12   }
```

So, when a variable is a string, the **+=** operation is the same as **append**, because you just append the new string onto the end of the existing one.

There are some unexpected effects of adding strings that you can run into. In **Program 9_8a**, you have a string, **s**, and you try to assign **s** the sum of 2 string literals, **Happy** and **Birthday**. But unexpectedly, you get a compiler error, basically telling you that you're not allowed to add the 2 string literals together.

For general purposes, you're not allowed to add together string literals. You can add together string variables, and you can even add a string variable and a string literal. But you cannot add 2 string literals.

```
1    // Program 9_8a
2    // ERROR - can't add string literals
3    #include <iostream>
4    #include <string>
5    using namespace std;
6
7    int main()
8    {
9       string s;
10      s = "Happy" + "Birthday";
11      cout << s << endl;
12   }
```

// CHAR-TYPE VARIABLES

A string is basically made up of an array, or a vector, of characters. A character is a letter, or digit, or punctuation, or space, etc. And a character has its own built-in type.

You don't need to #include any libraries to have access to it. The type is **char**.

Program 9_9 is a very basic program using a **char**-type variable. Notice that you don't need to #include any special libraries to use the **char**; you just have **iostream** so that you can output to the console.

You can declare a variable named **testchar** of type **char** by writing **char testchar;**. You can then assign a value to **testchar**—in this case, **a**. When **testchar** is streamed to output, **a** is printed to the screen.

In C, a string is basically an array. In C++, although there is still an array at its base, the way you use a string has many more vector-like options.

```
1    // Program 9_9
2    // char type
3    #include <iostream>
4    using namespace std;
5
6    int main() {
7       char testchar;
8       testchar = 'a';
9       cout << testchar << endl;
10   }
```

In C and C++, characters are enclosed in single quotes ('), and there can be just one character inside. Strings use double quotation marks (").

Note: The single and double quotes should be the basic, standard quotation mark. If you use a word processor to write your code, it will often "help" by changing these to curly quotes—which are not the same thing and will cause errors in your program.

ASCII Table

DEC	CHAR	DEC	CHAR	DEC	CHAR	DEC	CHAR
0	NUL (null)	32	SPACE	64	@	96	'
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

A **char** variable can take on any of the standard characters that you would type. These are called the ASCII characters, and each character you can get on a keyboard—letters, numbers, punctuation, spaces, etc.—is assigned a number.

The first 32 characters are control characters, followed by 92 printable characters, starting with punctuation marks and then numbers, then letters, first uppercase and then lowercase. In fact, if you assign a number to a **char** variable, we are meaning that it will have the corresponding ASCII value.

The backslash (\) is called an escape character, and it means that the next character will indicate what special action to take.

MOST COMMONLY NEEDED SPECIAL CHARACTERS	
\n	new line
\t	tab
\0	null
\"	quotation mark (")
\'	apostrophe (')
\\	backslash (\)
\?	question mark (?)
\a	audible bell
\b	backspace
\f	new page (form feed)
\r	carriage return
\v	vertical tab
\x...	hexadecimal value
\....	octal value
\u....	Unicode character (4 digit)
\U...	Unicode character (8 digit)

In C++, a string is basically a vector made up of **chars**. That's different from the way strings were handled in C, which basically used an array of **chars** with an extra **null** character—the ASCII 0—at the end to mark the end of the string.

Even in C++, a string literal is still treated like a C-style string. And because you can't add arrays, that's why you can't add string literals.

In both C and C++ strings, a particular character in a string can be accessed by using square brackets. And because C++ strings operate like vectors, they have all the benefits that vectors had over arrays.

For example, recall that the **at** member function allows you to access a particular character and check that you don't go out of bounds in the array.

You also have the ability to get the size of a string, just like you can get the size of a vector. For strings, it's easier to think in terms of **length** of a string rather than **size**, so a **length** command is also provided.

Strings can be compared to each other as well. You can check whether 2 strings are equal. This means every character has to be checked to see if it matches. And the same goes for checking whether 2 strings are not equal.

Comparing strings is the basis for sorting in alphabetical order. Because upper-case letters come before lower-case in ASCII, "Zebra" will come before "apple" when comparing strings.

Comparing less than or greater than is a little trickier. This really involves considering whether the ASCII characters come before or after. To compare whether one string comes before another string according to the ASCII table, the comparison will go through, character by character, to find the first character that doesn't match. If the ASCII value of the first nonmatching character is greater in one string, then that string is the "greater" string.

Strings have other member functions that are useful.

- The function **empty** returns a Boolean value that is true if, and only if, the string is empty—that is, it does not contain any characters.
- The **find** function takes another string as an argument. It returns the position where that other string first occurs in the given string.
- The substring function, **substr**, takes in 2 arguments: the first being the starting position and the second being the length. It returns a new string that is a piece of the old one. It will be the piece that starts at the starting position and is of whatever length is specified.
- The **replace** function lets you replace part of a string with another string. ◆

A full list of member functions available for strings can be found at http://www.cplusplus.com/reference/string/string/.

Exercise 1 Solution

Here's one way you might have written that program.

```
1   // Program 9_4
2   // Reading and using multiple strings
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int main()
8   {
9       string username, favoritecolor;
10      cout << "What's your name? ";
11      cin >> username;
12      cout << "What's your favorite color? ";
13      cin >> favoritecolor;
14      cout << username << ", I like the color " << favoritecolor << ", too!" << endl;
15  }
```

Click here to go back to the exercise.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 23.1–23.2.

b Lippman, Lajoie, and Moo, *C++ Primer*, section 3.2.

Exercise 2 Solution

a)
```
#include <string>
using namespace std;
string favoritefood;
```

b)
```
favoritefood = "pizza";
```

c)
```
cin >> favoritefood;
```

d)
```
getline(cin, favoritefood);
```

e)
```
cout << favoritefood;
```

Click here to go back to the exercise.

// QUIZ

1 What is the output of the following program?

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      string favoritefood = "Pizza";
7      cout << favoritefood << " is my favorite food."
   << endl;
8  }
```

2 What is the output of the following program?

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      string a = "One";
7      string b = "Two";
8      string c = "Three";
9      cout << a << b << c << endl;
10     cout << a+b+c << endl;
11 }
```

3 Assume that the string variable **a** has the value **abcdefghijklmno**. What would be the result of each of the following commands?

a `a.length()`

b `a.size()`

c `a[2]`

d `a.find("hi")`

e `a.substr(4,3)`

f `a.replace(a.begin()+3, a.begin()+8, "ABCDE")`

4 Write a program that reads in names and outputs the one that comes first alphabetically. You can assume that all names use the same capitalization pattern.

Click here to see the answers.

// QUIZ ANSWERS

1 `Pizza is my favorite food.`

2 `OneTwoThree`
`OneTwoThree`

Notice that outputting the 3 strings outputs them with no spaces in between. This appears the same as forming a new string by concatenating the 3 strings together (recall that `One + Two + Three` is `OneTwoThree`).

3

a **15**. There are 15 characters in the string.

b **15**. The length and size commands operate the same way.

c The result is the character **c**. Remember that numbering starts at 0.

d **7**. Notice that the string **hi** occurs at positions 7 and 8 and thus starts at position 7.

e **efg**. This is taking the substring starting at position **4**, which is the letter *e*, and taking **3** elements.

f **abcABCDEijklmno**. This replaces a part of the string, starting with position **3** (the *d* character) until just before position **8** (i.e., through the *h* character). The string **ABCDE** is inserted in place of the removed characters.

4 There are multiple ways to write such a program. You must first pick a way to stop the input, so you will use a sentinel string named **stop** to indicate this. Notice that when printing the quotation marks around this word, you must use a **** character beforehand. Because you can compare strings and the "smaller" string comes first alphabetically, the program will just read in strings and keep track of the smallest one seen, which is output at the end:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      string name;
7      string firstname;
8      cout << "Enter names, one per line, and
   enter \"stop\" when finished." << endl;
9      cin >> name;
10     firstname = name;
11     while (name != "stop") {
12         if (name < firstname) {
13             firstname = name;
14         }
15         cin >> name;
16     }
17     cout << "The first name alphabetically is
   " << firstname << endl;
18 }
```

Click here to go back to the quiz.

10 Files and Stream Operators in C++

Data is stored in files, which are located and saved in a long-term storage area, such as on a hard drive or solid-state drive, in flash memory, or even over a network in the cloud. Files contain data that you want to either read in to your program or write from your program. Streaming is a very general and flexible way to handle a wide variety of input and output.

IN THIS LECTURE:

// FILE STREAMING

Whenever a program is reading or writing to a file, it always involves 3 main steps:

1. Opening the file.
2. Reading from or writing to a file.
3. Closing the file.

You're going to need to use a library called **`fstream`** (which is short for *file streaming*) that gives you the tools to write to a file.

Just like you were able to **stream** console input and output using **`iostream`**, now you'll stream to and from files using **`fstream`**.

Once you have the **`fstream`** library included, you can create objects of the type **`fstream`**.

If you've been using only a browser-based C++ editor to run programs so far, now is the time to download an integrated development environment (IDE). For more details, see the C++ QUICK START or consult the supplementary material available online.

An **object** can be thought of as being like a variable, and you can basically use the words *object* and *variable* interchangeably. You use terminology like "declare an object" or "initialize an object" just like you do for other variables, such as integers.

But using the term *object* actually implies a little more than what you've been learning about with variables, because an object can store more than one piece of data and has functions that "belong" to it.

Vectors and strings are objects, and they have functions, such as **push_back** and **size** that "belong" to them. So, variables of type **fstream** are often called objects because they also include their own functions.

To create an **fstream** object, you just declare it like you declare other variables. You give the type, **fstream**, along with the object name—in this case, **my_file**.

To actually use the file, you need to link up the object name that's in your program with some file that's stored on your computer. You'll do this through an **open** command. There's a simple version of the **open** command, but in most cases, it's better to use the longer version.

To get a file to open in the simple version, you write the **fstream** object name, then a period, then **open**, then a string giving the file name in parentheses:

<fstream variable>.open(<string with file name>);.

The file name should be the name of the file in the computer itself; the one you would see if you opened a file explorer. Unless the file is in the same directory as your program, be sure to include the path information as part of the file name. In the following example line of code, the **fstream** object is called **my_file**, and the file name on the computer is **Test.dat**:

my_file.open("Test.dat");.

The longer version of the open command also asks you to specify how you want to work with that particular file—to read or to write. If you're just reading, it's straightforward to open the **my_file** object: **my_file.open("Test.dat");**. But writing to a file that's already on the computer can have a few different interpretations:

- You can leave everything in the file that's there but overwrite it one character at a time.
- You can leave everything in place and have everything you write added onto the end.
- You can delete everything that's already there and start writing to an empty file.

To specify which way to use the file, you'll use special syntax. When you open the file, you'll give not just the file name as an argument, but you'll also have a comma and then a second argument, starting with **fstream::**, which describes how you want to use the file:

my_file.open("Test.dat", fstream::2ndArgument).

There are 6 different options:

1. Read as input (**in**)
2. Output that overwrites one character at a time (**out**)
3. **Append** (**app**)
4. Truncate (**trunc**)
5. The at-end form of output (**ate**)
6. Binary (**binary**)

out will open for writing. Unless combined with **trunc** or **ate**, any existing data in the file will be overwritten.

app means that you're going to write output to the file but only after whatever's already there. Nothing gets erased or deleted.

trunc means that you should delete whatever is already in the file and start writing from scratch. You can only use **trunc** if **out** is also specified, and they are combined with a single vertical bar (**|**).

ate writes at the end of the file, usually with the same effect as for **app** but with some ways to move around in the file after you've written in one place. **ate** should also be enabled along with **out**, using **|** in between.

binary can be combined with any of the previous options using **|**. Specifying **binary** means that you'll be outputting just **1**s and **0**s directly. Binary is a more efficient way of storing data, but it means that people won't be able to open a file and look at it as text.

For the **fstream** variable **f**, to use the binary file **input.dat** for reading, we could write:

f.open("input.dat", fstream::in | fstream::binary);.

You can initialize file-stream objects when you declare them. This is just like how you initialize other variables: by setting them equal to a value when you first declare them. You do this by putting the information that you would put in the **open** function into parentheses that follow the declaration. Then, you don't need an **open** statement!

The 2 pieces of code below are basically equivalent, but the second piece of code eliminates the **open** statement to put everything on one line.

```
fstream my_file;

my_file.open("Test.dat", fstream::in);
```

```
fstream my_file("Test.dat", fstream::in);
```

When you initialize an object like this—by specifying parentheses and arguments when the object is declared—you're using a **constructor**.

Once an **fstream** object has been declared and a file has been opened, you can use that **fstream** object to stream to and from the file. This works exactly like **cin** and **cout** worked for the console, except instead of writing **cin** or **cout**, you use the name of the **fstream** object.

For example, to write the value **10** to the file linked with the **fstream** variable **f**, you could write:

```
f << 10 << endl.
```

Once you're done using a file, you should always **close** it to ensure that everything is written to it and it's left in a valid state. To close, you simply take the **fstream** object and call the function **close** that is a part of it; that is, you have the object name, then a period, then **close**, then parentheses, and a semicolon to end the line.

```
my_file.close();
```

Often, when you're reading information from a file, you'd rather read in an entire line of the file at once rather than just a single word. You can use the **getline** function to get an entire line. **getline** is a function that takes 2 parameters: an input stream and a place to hold the result. Previously, your input stream was **cin**, but now you can use the **fstream** object as your input stream. You hold the result in a string.

When reading from a file, you often want to keep processing through the whole file.

To check whether you are at the end of the file, you can call a function, **eof** (which stands for end-of-file), that belongs to the **fstream** object. This function doesn't take any arguments and returns a Boolean. If you have already hit the end of the file, then it returns **true**; otherwise, it returns **false**.

Every file on a computer has a name, followed by a period, followed by a short combination of letters called the extension of the file, such as .pdf or .cpp.

Some people think that just by changing the extension, you can change what the file actually is—but that's not the case. If you create a file, make sure that the extension accurately reflects what type it is.

Exercise 1

Suppose you have a file named **accounts.txt** and want to write data to the end of that file, keeping what's there. How could you create an object named **datafile** in your program that lets you write data to the end of your **accounts** text file?

Click here to see the solution.

```cpp
// Program 10_4
// Use of the eof function in a while loop
#include<fstream>
#include<iostream>
#include<string>
using namespace std;

int main() {
    fstream my_file;
    my_file.open("GroceryList.dat");
    string s;
    my_file >> s;
    while (!my_file.eof()) {
        cout << s << endl;
        my_file >> s;
    }
    my_file.close();
}
```

b

a

The **eof** function is helpful in a **while** loop. You can set up a loop that continues until you reach the end-of-file, where the **eof** function returns **true** (a). This loop will thus read every string from a file and output it to the console.

Note that the end-of-file is not true when you read the last thing in the file; it will still be false after reading that last item. Only after you've tried to read again and there's nothing left to read will **eof** be true.

So, in **Program 10_4**, notice that you try to read a string in from the file before you get to the **while** loop; then, you read a string again as the last thing inside the loop (b).

Whether you are checking the **while** condition for the first time or after completing an iteration of the loop, you're always trying to read a string just before checking **eof**. That way, if you fail to read a string, you'll see that **eof** is true when you check.

Notice that this **while** loop is basically a **for** loop. You have an initialization step, where you read in one string; you have a condition, making sure that **eof** is still false; and each iteration of the loop, you read in one more string.

So, you can write this code using a **for** loop (c), which describes how the loop behaves in just one statement.

Exercise 2

Write a program that will generate a file called **address.txt** containing your name and address.

Click here to see the solution.

Exercise 3

Using floating-point numbers, write a program that will read from a file named **balances.txt** and then compute and output the average of the numbers.

Click here to see the solution.

```cpp
// Program 10_5
// Use of the eof function in a for loop
#include<fstream>
#include<iostream>
#include<string>
using namespace std;

int main() {
    fstream my_file;
    my_file.open("GroceryList.dat");
    string s;
    for (my_file >> s; !my_file.eof(); my_file >> s) {
        cout << s << endl;
    }
    my_file.close();
}
```

c

// STRING STREAMING

In C++, the concept of streaming means any way of making data flow into or out of something. This means that there are other ways to stream that don't involve files or the console.

In particular, you can also stream to and from string variables. This lets you store data in a string, similar to the way you'd store data in a file but without having to actually put a file out on the operating system. And it lets you avoid nagging users for sentinel values when they've finished their input.

You can create a string by using streaming operations and write to a string like you would to a file. You can also stream in from a string, processing the data contained in a string, like you would the data from a file.

To stream in and out of strings, you'll need a new variable type called a stringstream. It's a type of variable that links a string to the streaming operations you've been using.

To declare and use this new type of variable, you'll need to #include the **sstream** library. The stringstream will provide a "link" to and store a particular string. This is much like how the file stream "links" to a particular file, but the stringstream also stores a particular string. And just like when you want to read or write a file you stream in and out of an **fstream**, when you want to read or write from a string, you stream in and out of a stringstream.

Data that you stream in and out of the stringstream will be written to or read from the string that is stored inside.

So, if you're going to stream data out of a stringstream, you need to give it some initial value. In other words, you need to tell that stringstream: "This is a starter string you should use." There's a built-in command, **str**, for specifying the string that you want to use as the default value argument.

Program 10_8 is a short program that reads in one word from a string. Notice that you #include **sstream** at the beginning (3).

The type of variable you declare is this new stringstream-type variable, and the variable name here is **wordreader** (9).

You have a variable of this new type and need to initialize it with some particular string. In this case, you initialize **wordreader** with the string **Apple Banana Cherry** (10).

Notice the pattern:

- **iostream** is for input/output with the console
- **fstream** is for files
- **sstream** is for strings

```
1   // Program 10_8
2   // Using a stringstream for input
3   #include<sstream>
4   #include<iostream>
5   #include<string>
6   using namespace std;
7
8   int main() {
9      stringstream wordreader;
10     wordreader.str("Apple Banana
    Cherry");
11     string first_word;
12     wordreader >> first_word;
13     cout << "The first string read
    was: " << first_word << endl;
14  }
```

The general form for streaming doesn't change whether you are using **cin** to get input from the console, **fstream** to get an item from a file, or **stringstream** to get an item from a string. So, suppose you want to read in some string variable named **first_word**. You can stream from **wordreader** into **first_word** (12). This will read in the first item from whatever string **wordreader** holds. In other words, the text until the first white space is read, and that is then assigned to **first_word**—in this case, **Apple**. Your output verifies that **Apple** is what has been assigned to **first_word**.

A stringstream can also be initialized with a constructor, similar to how you initialize a file. In this case, when you declare the stringstream, named **`wordreader`**, you just add parentheses with the string that you want to initialize inside the parentheses.

You can also stream output to a string. You just declare a stringstream variable and then, just like you could use **`cout`** to stream to the console, you can stream output to that variable.

Inside the stringstream variable will be a string, which will get whatever text you stream into it.

To actually get to that string, you again use the **`str`** command, but this time with no arguments. You just write **`.str()`**.

As you did for files, you can also use the end-of-file function on a stringstream.

When you've read the last character in the string, then **`eof`** will be true. Notice that that's a little different than files, where you had to actually read past the end of the file before **`eof`** would return **`true`**. But **`eof`** can still be used to go through all of a stringstream when you're processing it.

For simple programs, you could have created a string some other way—just by concatenating strings. But you'd have to manually convert your data to strings and use concatenation, rather than getting to use the stream operators, which handle a variety of data types automatically. ◆

By processing from a string, you can avoid the need to have sentinel values. There's no need to make a user enter an artificial value to end input; the user just ends the line when he or she is done typing.

Exercise 4

Write a program that

- first asks a user to enter a bunch of integers on one line, separated by spaces;
- then reads that whole line into a string; and
- then pulls out each number from that string separately, outputting them one by one.

Remember to use **`eof`** to find when you've reached the end of the stringstream.

Click here to see the solution.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 10 and sections 11.3 and 11.4.

b Lippman, Lajoie, and Moo, *C++ Primer*, chap. 8.

Exercise 1 Solution

There are a few options that all basically work the same way.

```
a    fstream datafile("accounts.txt", fstream::app);

b    fstream datafile;
     datafile.open("accounts.txt", fstream::app);

c    fstream datafile;
     datafile.open("accounts.txt", fstream::out | fstream::ate);

d    fstream datafile("accounts.txt", fstream::out | fstream::ate);
```

Click here to go back to the exercise.

Exercise 2 Solution

Here's what that code might look like.

```
1   // Program 10_6
2   // Writing an address to a file
3   #include<fstream>
4   #include<string>
5   using namespace std;
6
7   int main() {
8       fstream my_file("address.txt", fstream::out);
9       my_file << "John Keyser" << endl;
10      my_file << "123 Any St." << endl;
11      my_file << "Somewhere, TX 77777" << endl;
12      my_file.close();
13  }
```

Click here to go back to the exercise.

Exercise 3 Solution

Here's one way you could write that program.

```
1   // Program 10_7
2   // Averaging values in a file
3   #include<fstream>
4   #include<iostream>
5   #include<string>
6   using namespace std;
7
8   int main() {
9       fstream bank_file("balances.txt", fstream::in);
10      float f;
11      float total = 0.0;
12      int num_vals = 0;
13      for (bank_file >> f; !bank_file.eof(); bank_file >> f) {
14          total += f;
15          num_vals++;
16      }
17      bank_file.close();
18      cout << "The average was " << total / num_vals << endl;
19  }
```

Click here to go back to the exercise.

Exercise 4 Solution

Here's one way that might look. Notice that you don't have to bother the user for a sentinel value!

```
1   // Program 10_12
2   // Reading from a stringstream
3   #include<sstream>
4   #include<iostream>
5   #include<string>
6   using namespace std;
7
8   int main() {
9       cout << "Enter several integers on one line:" << endl;
10      string input_line;
11      getline(cin, input_line);
12      stringstream string_input(input_line);
13      int val;
14      while (!string_input.eof()) {
15          string_input >> val;
16          cout << "You entered: " << val << endl;
17      }
18  }
```

Click here to go back to the exercise.

// QUIZ

1 Answer the following questions.

a What is the library that should be included to access files, and what is the library used to read and write from a string?

b What is the variable type you use that allows you to stream to and from a file, and what type allows you to stream to and from a string?

c What 2 pieces of information are provided when opening a file?

d What is the last thing you should do with a file in your program?

e What designator do you use to write to a file by adding to the end of an existing file?

f What function gives you access directly to the string stored inside a stringstream?

2 Write a program that creates a file named **`myname.txt`**, containing your name.

3 Write a program that reads in integer scores from a file, **`scores.dat`**, and outputs the average score.

4 Write a program that reads in a series of integer scores on a single line from the console and outputs the average score.

Click here to see the answers.

// QUIZ ANSWERS

1

a **fstream** and **sstream**. These should be #included in the program if you wish to use file streaming or string streaming.

b **fstream** and **stringstream**. Notice that while **fstream** has the same name as the library that is included to gain access to it, **stringstream** does not.

c The name of the file on the computer system and the way you want to use the file you opened.

d Close it. Files should be closed when you are done using them.

e There are 2 ways: You can use **fstream::app** to append to the end of a file, or you can use **fstream::out | fstream::ate** to designate the file for output, starting at the end of the current file. The second version would technically allow you to move to an earlier point in the file and overwrite, too.

f **str()**. If you have a stringstream variable **ss**, then **ss.str()** will give you the string that is stored within it.

2

Here is one program. Notice that you #include the **fstream** library. You create an **fstream** object, **output_file**, that you open to link to the file **myname.txt** and designate for output. Next, you output one line: a name. Finally, you close the file by calling the **close** command.

```
#include<fstream>
using namespace std;

int main() {
    fstream output_file("myname.txt", fstream::out);
    output_file << "John Keyser" << endl; // Use your name
    output_file.close();
}
```

3

Here is one possible program. Notice that you first open the file, specifying that it will be for input. You set up variables to hold an individual score, the total sum of scores, and the number of values read in. Then, you have a loop: You begin by reading a value from the file into the variable **score** and do this on every iteration of the loop. This continues until you reach the end of the file. Note that you could have written this loop as a **while** loop or in other ways, too.

Within the loop, you simply update your total sum of scores and the number of scores you have. After the loop, you close the file. Finally, you print out the resulting average (the total divided by the number of elements), being sure to cast one of the variables in the computation to a floating-point value (in this case, **double**, though **float** would have also been fine) so that you perform floating-point division rather than integer division.

```
#include<fstream>
#include<iostream>
#include<string>
using namespace std;

int main() {
    fstream score_file("scores.dat", fstream::in);
    int score;
    int total = 0;
    int num_vals = 0;
    for (score_file >> score; !score_file.eof(); score_file >> score) {
        total += score;
        num_vals++;
    }
    score_file.close();
    cout << "The average was " << total / (double) num_vals << endl;
}
```

4 Here is a program; it may be helpful to compare to the previous program to see differences between files and stringstreams. You will read an entire line from input (using the **getline** command) and use this to initialize a stringstream (in this case, **ss**). You will then read integers until you have read the last value from the stringstream. Note that the **for** loop you used for files should not be used here: the **eof** function is true for files only after trying to read beyond the last element, while for stringstreams, **eof** is true once the last element has been read. Thus, it is easier to use a **while** loop for stringstreams, with each iteration reading a value and then updating the sum total and the number of scores. The average is output just as with files.

```
#include<sstream>
#include<iostream>
#include<string>
using namespace std;

int main() {
    string inputline;
    getline(cin, inputline);
    stringstream ss(inputline);
    int total = 0;
    int score;
    int num_vals = 0;
    while (!ss.eof()) {
        ss >> score;
        total += score;
        num_vals++;
    }
    cout << "The average was " << total / (double) num_vals << endl;
}
```

Click here to go back to the quiz.

11 Top-Down Design and Using a C++ Debugger

By this point in the course, you've accumulated a lot of tools—a lot of individual ways to code various things. But there's a difference between knowing how to use the individual programming tools and knowing how to bring them together to create something larger. Some useful methods for approaching bigger software programs are top-down design, incremental development, and the debugger tool provided in your integrated development environment (IDE).

IN THIS LECTURE:

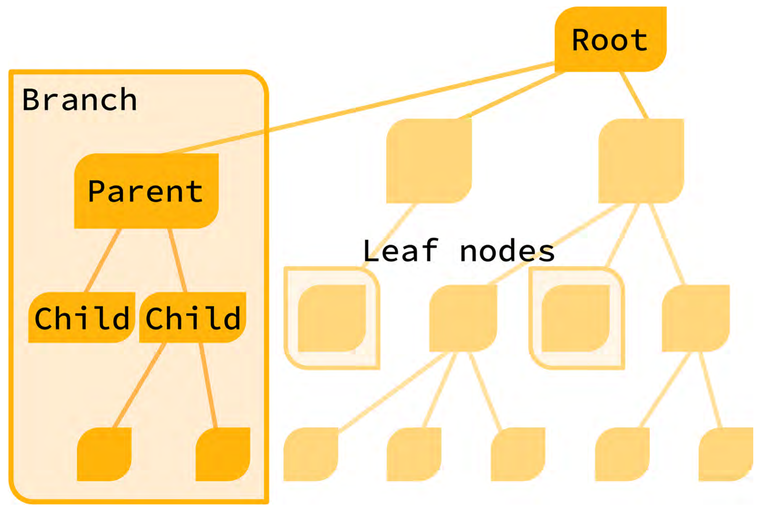
// TOP-DOWN DESIGN

The basic idea of top-down design is that you take a big task and break it into a sequence of smaller tasks, and then you take that and break it into a sequence of even smaller tasks, and so on.

A pitfall for many beginners approaching a software program is jumping right in to writing individual lines of code—the smallest unit of code there is. Instead, you need to consider the programming task as a whole and try to break it down into conceptually simpler parts. Eventually, you do get down to individual lines of code. But at first, you don't need to think about the details of the code—just the overall steps to follow.

The question programmers face when performing top-down design is exactly how to break up the task.

In computer science, this hierarchical structure as a whole is called a tree. The individual elements in the tree are called nodes. The single node at the top of the tree is the root. If 2 nodes are connected, then the one closer to the root is called the parent, and the ones farther away are the children. Every node except the root has exactly one parent, but it can have several children. For a parent, the children, as well as all of their own children, are branches. The nodes that don't have any children are called leaf nodes.



The number of levels in the tree are referred to as the depth of the tree, and the average number of children in a non-leaf node is called the branching ratio.

As you think of the design process—that is, how you decompose a problem into a hierarchy—you're often concerned with 2 questions: how deep you want the tree to be and what branching ratio you want.

There isn't a single answer to either question; instead, there are 2 rules of thumb:

1. The branching ratio (the average number of children per node) should not be so high that you cannot understand all the children of one node. In other words, one task should be broken up into a few subparts so that it's easy to understand how the larger task is broken into smaller ones.
2. The depth (the number of levels in the tree) should be deep enough that the leaf nodes are obvious—meaning that a particular task should be something that an experienced programmer could readily implement without having to understand the other parts of the program. Typically, this is just a few lines of code.

In software development, there's always more than one way to write a program. And while some ways of solving a programming problem are better or worse than others, there's rarely any one way that is truly ideal in every regard.

Imagine that you have been asked to create a program to predict electricity usage at someone's house. You have files available with data about past electricity usage as well as weather data and information about the person's schedule, and you want to get a sense of what to expect regarding future energy demand.

Let's use a top-down approach to attack this problem.

First, think of the major steps that you'll want to go through. You'll want to read in information from the data files you have; get some information about exactly what you want to predict, such as the range of dates; do your calculation to actually make the prediction; and output some results.

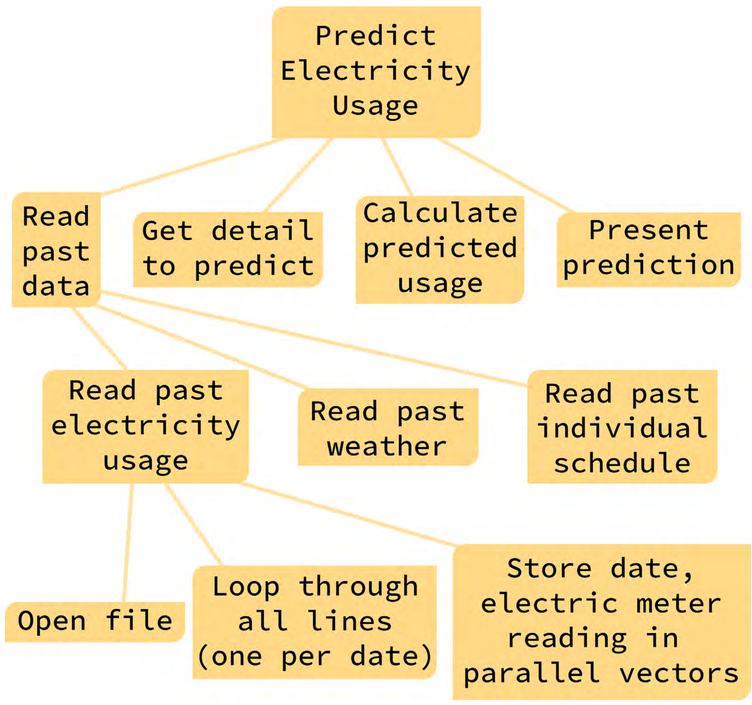
Notice that any one step is simpler—and that's the point! For any one idea, breaking it down into simpler ideas shouldn't be too difficult. Then, you can turn to each of those subtasks that you identified and expand them.

For the first subtask, reading the past data, you have information about past electricity usage, past weather data, and past schedule, so you'll want to read in the data from each of those files.

Next, you break down each of these items. Reading the electric usage file will depend on how it's stored, but suppose the data is stored in a series of lines, each with a date and an electric meter reading. The basic process that you'll want to follow is to open the file, read in each line, and then store the data—in this case, in parallel vectors: one for the date and one for the meter reading.

You keep going until you reach a level that should be pretty straightforward to code; each of your tasks should correspond to just one or a few lines of code.

You can continue to walk through the entire program at this level of detail for each part.



Once you have the program decomposed with a top-down design, then you can begin to code it. To get started, this is a great point to take the pseudocode you've been generating and convert each step into a comment, generating comments for each section. These comments will help guide your code, and they provide an explanation for the purpose of each part of the code, both during and after writing the code itself.

Considering the tree hierarchy, the root node becomes the overall comment for the program (2). The next level, the children of the root, should each get their own comment. Because these are at the higher levels of the hierarchy and will be major sections, it can make sense to designate these in some way that makes them stand out—for example, by using several asterisks (a).

Then, you can divide those into the next level down. Again, you convert each node into a new comment. In this case, you can again use a few asterisks to show that these are not the most basic levels of the hierarchy (b).

Then, you can go from there to even the next level down, converting those nodes to comments (c).

Each of these is still pretty self-explanatory. Once you've done this for your whole program, it's time to start writing code.

```
1   // Program Fragment 11_1_c
2   // Program to Predict Electricity Usage
3   #include <iostream>
4   #include <vector>
5   #include <fstream>
6   #include <string>
7   using namespace std;
8
9   int main() {
10    /***** Read Past Data *****/                          a
11    /*** Read Past Electricity Data ***/
12    /* Open File */                                       c
13    /* Loop Through All Lines */
14    /* Store Date and Meter Reading in Parallel Vectors */
15
16    /*** Read Past Weather Data ***/
17
18    /*** Read Past Individual Schedule ***/              b
19
20
21    /***** Get Detail To Predict *****/
22
23
24    /***** Calculate Predicted Usage *****/
25
26
27    /***** Present Prediction *****/
28
29  }
```

// INCREMENTAL DEVELOPMENT

When you develop software, you want to construct the software piece by piece, and as each piece is put into place, you want to ensure that you have a stable structure—something that is not yet complete but that will operate just fine for the part that's been written. You want to be able to see that each piece works and is tested as you go along. And if you ever write something that's not working, you can always take one step back and return to a stable piece of software.

Unfortunately, often when programmers—even experienced ones—construct a larger piece of software, they'll write lots of code, and only at the end, after they think they have all the pieces in place, do they try to test it. If everything actually works, it's glorious! But it's extremely rare for that to happen. Instead, somewhere along the way, they made a mistake, and the entire program falls apart. Trying to find that one mistake is difficult, and sometimes even after fixing that, there's some other mistake, and they can't even figure out if they really fixed a problem or not.

This is why you should use an incremental development approach, in which you code a little, test it until you're confident that everything up to that point is working, code a little more, test again, and so on. You never move on until you're confident in what you already have.

In this case, it's OK to design your entire program top-down and write out the comments for each part ahead of time. But when it comes to coding, you want to write code in a way that you can implement a small chunk of the program and test what you've written. Generally, this means that you'll have to write code in a sequential order, though once you start writing functions, that won't exactly have to be the case.

In this example, the first section you encounter is part of reading past data, where you first will read the past electricity data and where you first want to open the file (d). That means you declare an **fstream** variable (13) and then **open** it for input (14). Notice that you also write a line of code to **close** the file (17).

```
1   // Program Fragment 11_1_d
2   // Program to Predict Electricity Usage
3   #include <iostream>
4   #include <vector>
5   #include <fstream>
6   #include <string>
7   using namespace std;
8
9   int main() {
10     /***** Read Past Data *****/
11     /*** Read Past Electricity Data ***/
12     /* Open File */                                    d
13     fstream electricfile;
14     electricfile.open("MeterReadings.dat",
fstream::in);
15     /* Loop Through All Lines */
16     /* Store Date and Meter Reading in Parallel
Vectors */
17     electricfile.close();
18
19     /*** Read Past Weather Data ***/
20
21     /*** Read Past Individual Schedule ***/
22
23
24     /***** Get Detail To Predict *****/
25
26
27     /***** Calculate Predicted Usage *****/
28
29
30     /***** Present Prediction *****/
31
32  }
```

Incremental development is a critical part of developing large software programs. It's almost impossible to develop large programs without it.

After writing this, before going on to the next section, you want to stop and test what you have, making sure that you don't have a typo or some statement formatted incorrectly.

Once that's tested, it's time to write and check your next section—looping through all the lines of the program (e).

In this case, you'll assume that the file consists of lines, each of which is a date given by a string and a meter reading given by an integer. So, to loop through all the lines in the file, you'll need to have 2 variables, labeled **date** (16) and **meter** (17). You'll try to read them in from the file (18) and loop until you've reached the end of the file (19). Inside the loop, you'll read the next line (22).

Also inside the loop, you'll have what you want to happen each time, which is to add the date and reading to vectors (20). But you're not going to write that yet; you'll test what you have so far.

To test that you are reading in all the lines from the file, you could put in some output statements to show what you read in from each line. That's better than nothing, but another option that's available is using the debugger.

```
1   // Program Fragment 11_1_e
2   // Program to Predict Electricity Usage
3   #include <iostream>
4   #include <vector>
5   #include <fstream>
6   #include <string>
7   using namespace std;
8
9   int main() {
10    /***** Read Past Data *****/
11    /*** Read Past Electricity Data ***/
12    /* Open File */
13    fstream electricfile;
14    electricfile.open("MeterReadings.dat", fstream::in);
15    /* Loop Through All Lines */                    e
16    string date;
17    int meter;
18    electricfile >> date >> meter;
19    while (!electricfile.eof()) {
20       /* Store Date and Meter Reading in Parallel Vectors */
21
22       electricfile >> date >> meter;
23    }
24    electricfile.close();
25    /*** Read Past Weather Data ***/
26
27    /*** Read Past Individual Schedule ***/
28
29
30    /***** Get Detail To Predict *****/
31
32
33    /***** Calculate Predicted Usage *****/
34
35
36    /***** Present Prediction *****/
37
38  }
```

// DEBUGGER TOOL

As you work to develop larger programs, having the use of an IDE can be critical to helping examine the state of the program as it progresses, making sure that the program is working as intended.

An IDE makes it easy to access and use the **debugger** tool. There are other debuggers, but by far the easiest to use are those included within an IDE. The one in Visual Studio is used here.

Unfortunately, a debugger is not a tool that takes the bugs out of your program. In fact, a debugger is a tool that lets you step through a program's instructions one at a time and examine the state of everything in the program at that point.

Debuggers have many tools within them that can help developers examine their code very closely. But there are just a few simple parts that are enough to make the debugger a really useful tool, and even if you knew every feature of the debugger, these would still account for 90% of what you'd ever do. These features are **breakpoints**, the Step Over and Step Into operations, and variable values.

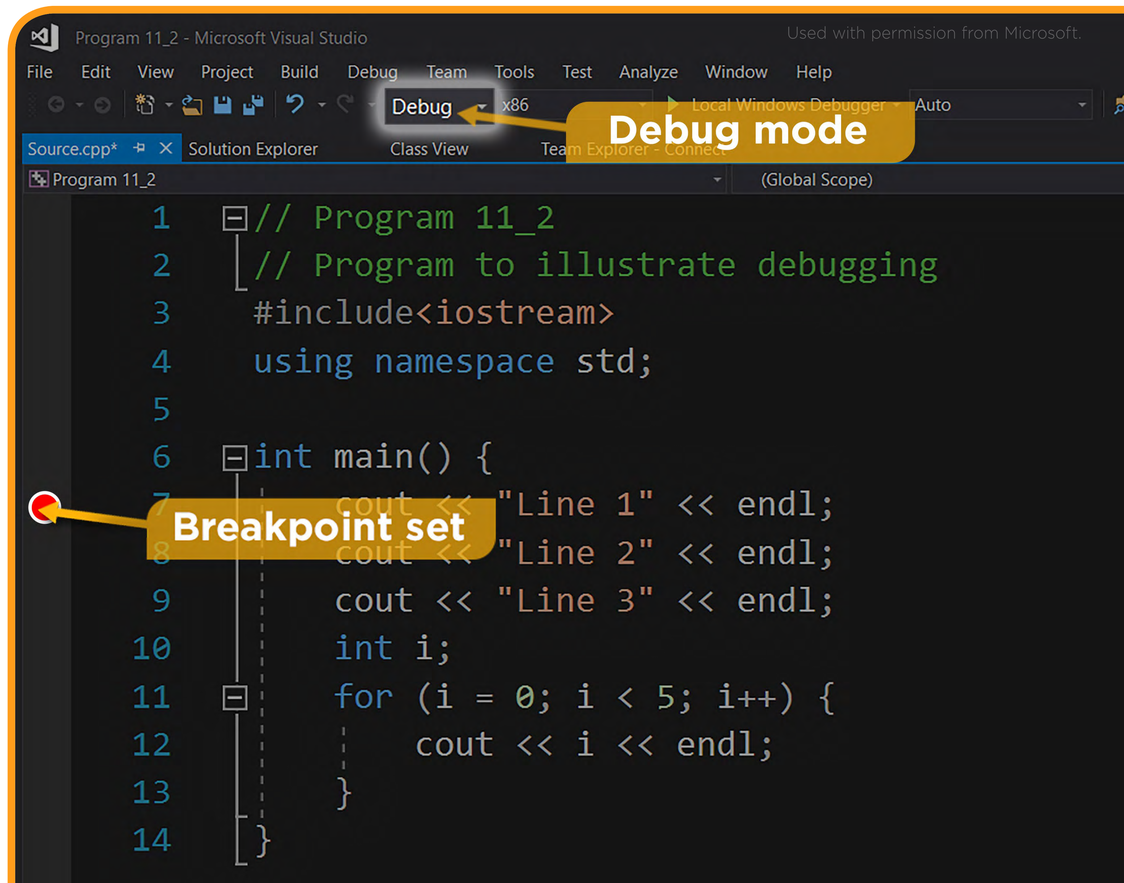
A breakpoint is a point in the code at which you want to stop the execution, usually so that you can examine the code more closely. When you run a program in the debugger, it'll run just like you'd expect until you hit a breakpoint. The execution will pause just before the line where the breakpoint is set. The breakpoint is a "break", not a line that gets executed. If you don't have a breakpoint set, the code will just run all the way through. That's what you've been seeing until now.

In Visual Studio, to set a breakpoint, you click on the area farthest to the left edge for the line where you want to stop.

If you set a breakpoint at the first **`cout`** statement, this puts a red circle at that line. (In Xcode, you click on the thin vertical area just left of the line of code and a blue marker (▬) appears to show the breakpoint.)

In order to use a debugger, the code must be compiled in what's called **debug mode**, which adds some extra stuff in automatically that lets the debugger work. But this mode also makes the code less efficient than it will be when fully debugged and then compiled into a more efficient form called **release mode**.

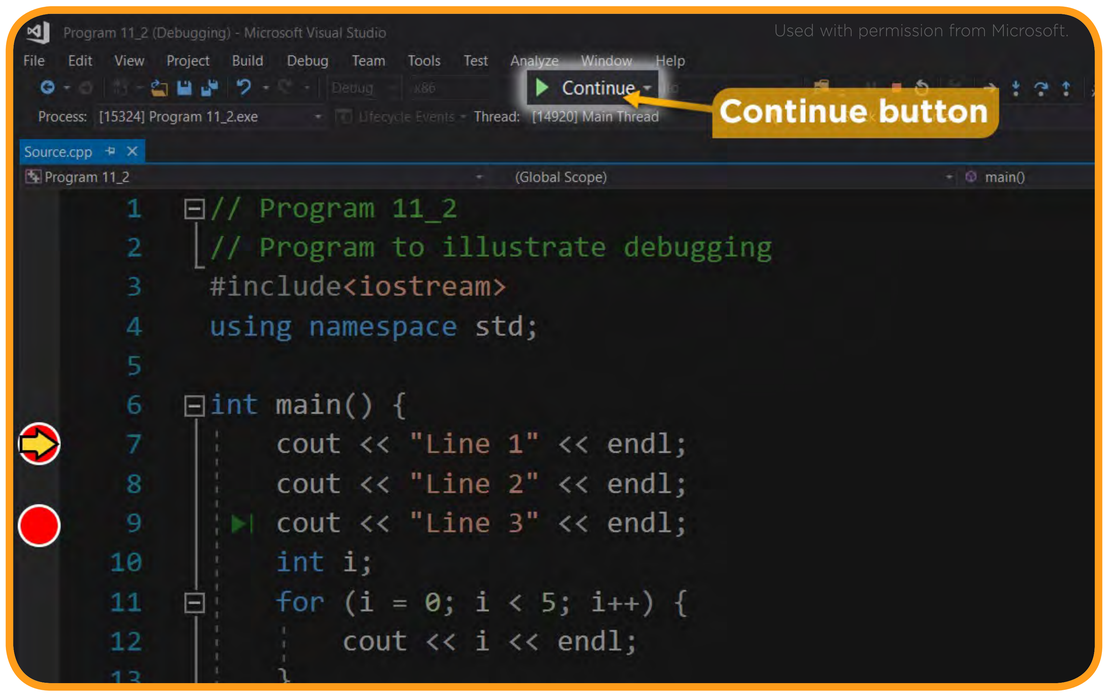
Because most development is done in debug mode, that's the default mode for compiling in both Visual Studio and Xcode.



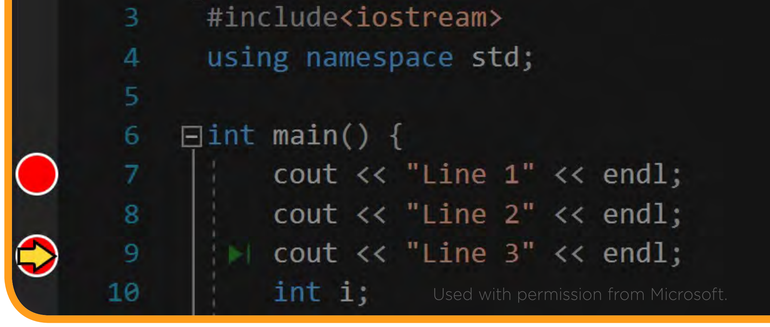
If you run the program, it will run until it gets to the breakpoint and then stop. There will be a visual indicator showing which line of code will be next to execute: a yellow arrow on top of a red circle in Visual Studio (or a highlighted green line in Xcode).

In this case, you're stopping before any lines of output were executed.

Next, set a second breakpoint at the third **cout** statement. If you hit the Continue button at this point, the program will continue executing until it hits that next breakpoint. (In Xcode, the Continue button is a rightward-facing hollow arrow (▮▶) below the code.)



After pressing the Continue button, notice that the indicator shows you're now at this new line.

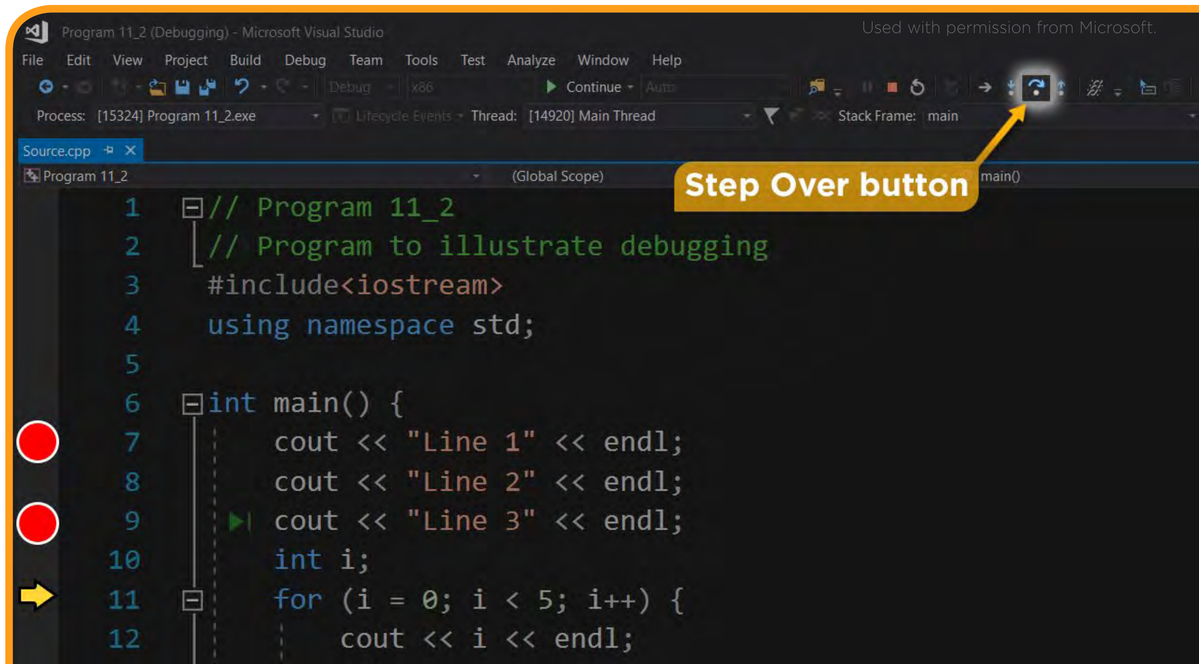


Also notice in the output area of the IDE that you've output 2 lines of code: The first 2 **cout** statements were executed.

That's one way to go through the program: to set breakpoints and continue running between them. But once you've reached a point of interest in the code, you often want to examine just one line at a time. Putting a breakpoint on every line would be one way to do this, but that's overkill.

One way you can progress through the code line by line is by using a command called Step Over. In Visual Studio, there is a button near the top called Step Over. (In Xcode, the Step Over button (⌂) is at the bottom, next to the Continue button.)

At this point, the next line to be executed is the third **cout** statement. If you push the Step Over button, notice that the marker showing which line you're about to execute progresses to the next executable line: the **for** statement. The third line was output.



KEYBOARD SHORTCUTS

	START DEBUGGING	STEP OVER
WINDOWS	F5	F10
MAC	⌘R	F6

If you push Step Over again, you are inside the **for** loop. If you push Step Over once more, you print the **i** value and go back to the top of the **for** loop, where you will check the conditional. And you can keep stepping through.

You'll notice that you go through all iterations of your loop and eventually leave once the loop condition is false.

To stop your debugging session, you can hit the square Stop button, which will stop the entire run.

Besides the Step Over command to step over the next line of code, there's a Step Into command, which doesn't just say "execute the next line of code." If that next line of code has a function call in it, then the execution goes *into* that function. That way, you'll be able to see how the function itself is working. There's also a Step Out button if you are tired of looking inside a function's inner workings and want to return to the place it was called.

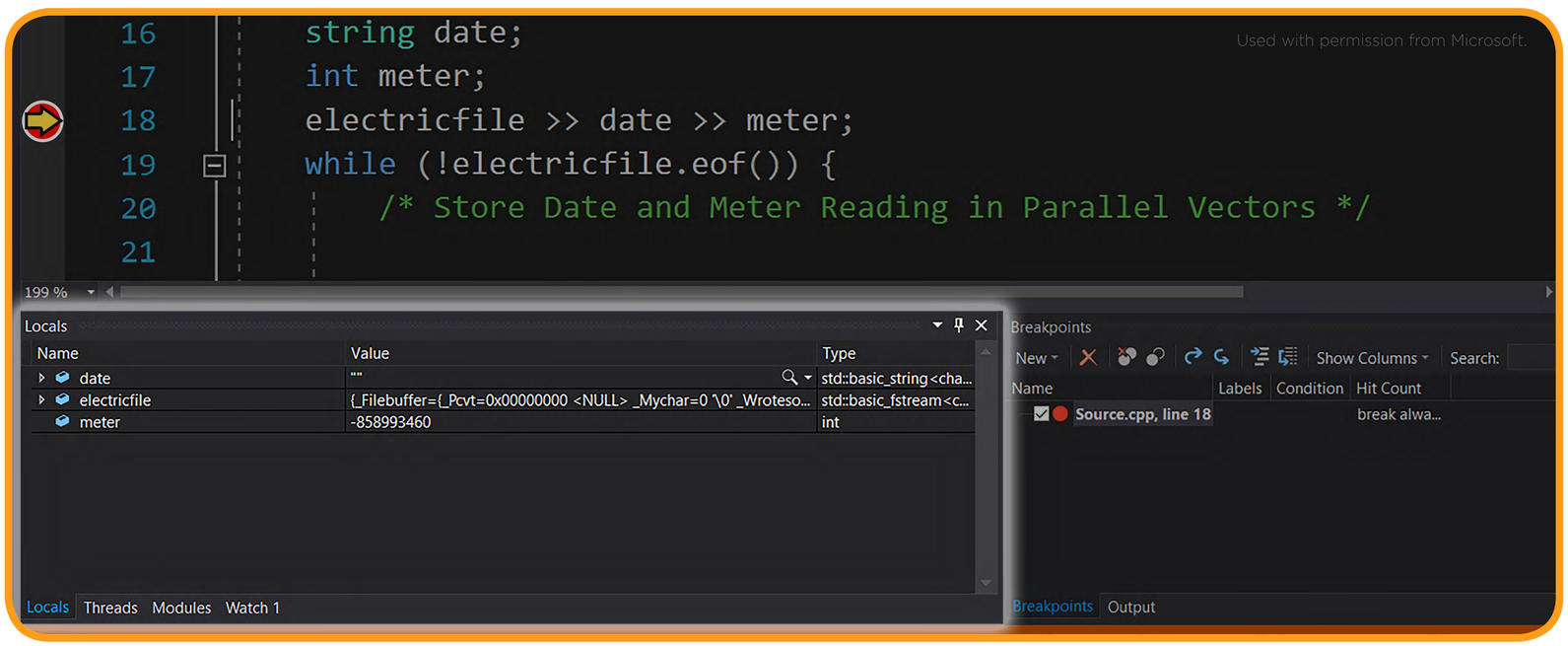
There's another key part of the debugger. When the debugger runs, there is a window at the bottom that lets you examine the values of variables.

In the electric usage program, put a breakpoint right before the first line where you stream in from **electricfile** and then start the debugger. It will execute right up to that line.

At the bottom of the screen, you should see a window that shows variable values. In Visual Studio, it will have 3 tabs titled Autos, Locals, and Watch 1, and one of them (probably Locals) will be selected. You should see 3 items listed there. In alphabetical order, they are the 3 variables you have at this point in the program: **date**, **electricfile**, and **meter**.

Next to each of them is the value of that variable. Notice that neither **date** nor **meter** is initialized to anything. The **date** should have an empty string as its value, while the **meter** could have any possible number. You haven't read anything into either one of them yet. The variable **electricfile** doesn't have a very meaningful value; it's reporting on how an **fstream** is implemented.

If you go forward one line, by pressing Step Over, you move on to the next line of the code. You also see that the values in both **date** and **meter** have been updated. The previous line read data into both of them from the file. In this case, you have a date of **1/1/2018** and a meter reading of **1024**. As you go through the code, you can keep an eye on the window to see what the values of variables are.



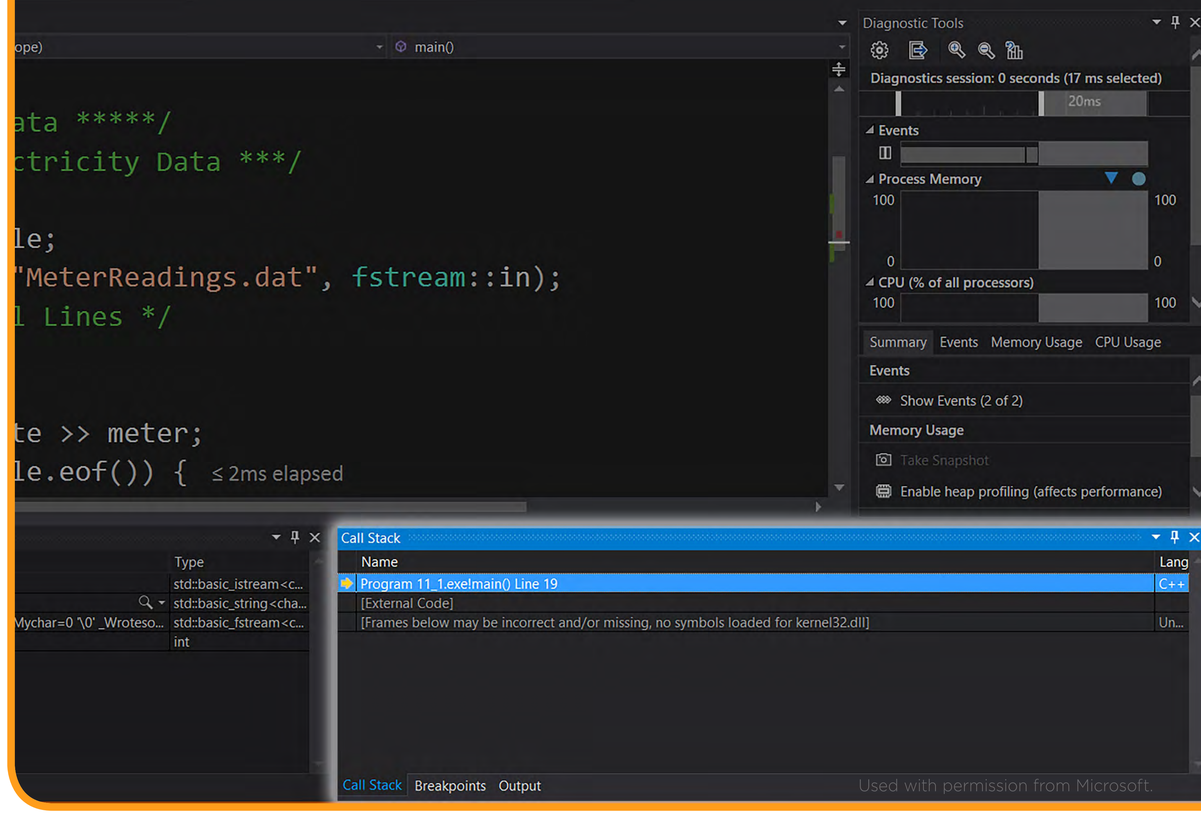
When approaching any large programming problem, start with a top-down design, build it incrementally, and use the debugger to help you verify that things are working.

One other useful item in the debugger is Call Stack. In Visual Studio, Call Stack is found in the lower right, and in Xcode, it's in the lower left, where there's a window showing the Call Stack.

If you have written your own functions and stepped into those, then the Call Stack will also show which functions have been called in order from other functions. In Visual Studio, the Call Stack will also show the line numbers.

The whole reason you're using the debugger here is to verify that your code is working as expected. The debugger will also be very useful when you do encounter bugs and need to find where they're occurring.

At this point, you could expand on the program that has been started here, first completing the full top-down design of the entire program and then incrementally building every piece of it. ◆



READINGS

a Top-down design is a long-standing software development practice and is a typical approach used in procedural programming for languages that do not support object-oriented design, such as C.

b Top-down design with functions can thus be found in several C textbooks—e.g., *Problem Solving and Program Design in C* (8th ed.) by Jeri R. Hanly and Elliot B. Koffman—or C++ textbooks with more of a build-on-C flavor—e.g., *Problem Solving, Abstraction, and Design Using C++* (6th ed.) by Frank L. Friedman and Elliot B. Koffman.

// QUIZ

1 When using a debugger, what button/command would you use to do each of the following?

a Designate a line of code to stop at during execution.

b Execute the current line of code.

c Go into the function being called in the current line of code.

d Look at the value currently stored in a variable.

2 A tree is used repeatedly in software development and in describing organizations that are formed from top-down design. What is the name used for each of the following elements of a tree?

a The single node at the top of the tree

b The single node that comes above another node in the tree

c The node(s) that come below others in the tree

d The nodes that have no nodes below them

3 Answer the following questions about performing top-down design of a program.

a When do you know that you have gone far enough—that is, broken the design down into small enough parts?

b When breaking one idea/task into several others, how do you determine what level of detail to subdivide the idea/task into?

4 If you had the following top-down design of a program, how might you set up the initial program? Note: You do not need to fill in the program, but you should set up the program so that code could then be written to make it work.

a Compute profit generated over time period.

i Gather data information.
1 Prompt user for dates of interest.
2 Read in data from user.

ii Gather records of income and expenses.
1 Get purchasing records from time period.
2 Get payroll expenses from time period.
3 Get capital/service expenses for time period.
4 Get sales records from time period.

iii Compute total income.

iv Report data to user.

Click here to see the answers.

// QUIZ ANSWERS

1

a Set a breakpoint. In an IDE, this is often done by clicking at a point in front of the line of code.

b Step Over. After executing, the debugger will go on to the next line of code in the same function or, if it was the last line of a function, leave the function and return to where it was called from.

c Step Into. If the current line has a function call in it, then the debugger will go into the function, to the first line in the function body.

d Use the Watch list. Most IDE debuggers will have a sub-window in which you can watch variables you care about. Some debuggers will have other options, such as letting you type a variable you want the value of.

2

a Root. There is a single root node to the tree. This can be thought of as the top of the hierarchy, like the CEO in an organization.

b Parent. In a tree, each node except the root has one parent. This is the element that is higher in the hierarchy, like a person's manager in an organization.

c Child/children. A node may have multiple children, one child, or none. Children are the next level down in the hierarchy, like the people directly supervised in an organization.

d Leaf. Leaf nodes are those with no children, like the people in an organization who are not managing anyone else.

3 Top-down design takes practice and experience, and there is never a single right answer. So, all approaches are just guidelines.

a You stop when a task is at a level that implementing it in code will be obvious. In other words, an experienced programmer should have no difficulty taking the lowest level of design and generating several lines of code to implement it.

b A task should be broken into a set of simpler subtasks so that each subtask is still a single coherent idea and it is easy to understand the greater task as a combination of the few subtasks.

4 The initial step to follow is to convert the outline of the program into comments. Then, in later stages, you would fill in the code corresponding to each comment. The root gives the overall purpose of the program and can be placed at the beginning of the program. The other steps will take place in **main**. In the following code, the higher levels of the hierarchy are specified with **/*** ***/** comments, while lower levels use **//** comments. The blank lines show where code will be written.

```
1   /* Compute profit generated over time period */
2   #include<iostream>
3   using namespace std;
4
5   int main() {
6       /*** Gather data information ***/
7       // Prompt user for dates of interest
8
9       // Read in data from user
10
11      /*** Gather records of income and expenses ***/
12      // Get purchasing records from time period
13
14      // Get payroll expenses from time period
15
16      // Get capital/service expenses for time period
17
18      // Get sales records from time period
19
20      /*** Compute total income ***/
21
22      /*** Report data to user. ***/
23
24  }
```

Click here to go back to the quiz.

12 Creating Your Own Functions in C++

Functions are a key part of programming. They enable you to organize your programs by breaking up ideas into individual pieces—the functions—and they can sometimes save a lot of repeated coding. You've already been using functions; you've imported functions from libraries and used functions to get input, for file operations, and with vectors. All of these functions performed some computation for you, all by just making a function call.

IN THIS LECTURE:

// FUNCTIONS AS BLACK BOXES

The easiest way to think of a **function** is as a so-called black box. You have some program, consisting of various instructions. When you want something to happen, you make a function call. You possibly send the function some input **parameters**; these are the **arguments**. Then, the function takes some action, and possibly there's a final **return** value sent back to the part of the program that called the function. But the inner workings of the function—that is, knowing how the function does what it does—are hidden from the `main` program.

On the other hand, when you write a function, you don't have to worry about how it is being used. You might be getting some input parameters, and you might know that you need to return some particular value. But you shouldn't need to worry at all about when and how that function is being called. You don't care about the function's role in the `main` program; you just need to know that the function is going to do its job.

This brings up the 2 main reasons you have functions:

1. Functions offer a conceptual separation of ideas. Functions let you write code without worrying about details. Either you're writing a function and not worrying about how it is being called or you're writing the calling program and not worrying about how the function works.
2. Using functions lets you avoid the work of writing the same code over and over again.

// CREATING YOUR OWN FUNCTIONS

Suppose you want to write a function that takes in a string and returns the number of words the string contains.

The first part of the definition of a function is the return type (a), which states what the type of the return value will be. In this example, because you're counting the number of words, you'll want to return an integer, so the type is `int`.

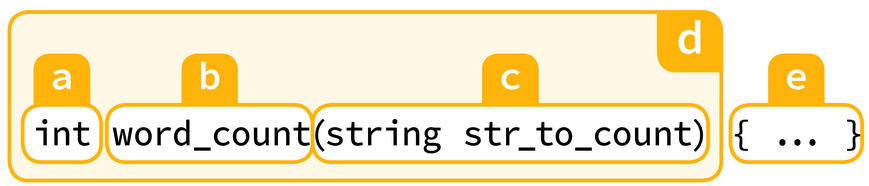
The next part of the definition is the function name (b)—the name you'll use to call the function. You're writing a function that will count words, so `word_count` seems like a reasonable name for the function.

When naming functions, you should take the same care as you do when naming variables. Use a descriptive, memorable, and consistent name.

Then, there will be a pair of parentheses, inside of which will be the parameters (c). For now, you need to specify the type of the parameter being passed in and give it a name. In this example, you are passing in a string and you are going to be counting words in that string, so the parameter is of type string and has the name `str_to_count`.

The part of a function up to this point is called the **function header** (d), which contains the return type, the name of the function, and the parentheses giving the parameter list. The function header contains all of the information needed for the black box of the function to interact with the larger program.

Finally, there is a pair of curly braces, inside of which will be the actual instructions for the function (e). This is called the **function body**. These are the actual contents of the black box—the commands run whenever the function is called. (For this initial code snippet, there are no specific commands for the function body.)



When you've made function calls, you've put values in the parentheses and called them arguments—the values that are given to the function that you want the function to use to do its thing with.

As you're now writing your own functions, you'll again have values that are specified in parentheses, but now these values are called parameters.

The parameters are directly tied to the arguments. The arguments that you put in parentheses when you call a function match up one-to-one with the parameters that you use inside the function itself. Because the arguments and parameters are linked together so closely, sometimes the terms are used interchangeably. But if you're being careful, arguments will be the values in the function call, and parameters will be the values that come inside the parentheses of the function itself.

Sending in the parameters is called passing. When you transfer a value from the calling side to the function, you say that you are "passing in" a parameter.

You've used functions that take one parameter, more than one parameter, and even no parameters. You can specify each situation—one parameter, multiple parameters, or no parameters—in a function header. You've already defined a function, **`word_count`**, that took a single parameter, a string.

If you have multiple parameters, you just separate the parameters with commas. Each of the parameters is defined the same way an individual parameter would be. But note that each parameter needs both a type and a name.

To define a function that has no parameters, you put nothing between the parentheses.

Sometimes you don't need a function to return anything; it just needs to do something on its own and not return a type of any sort. In this case, you can specify a different return type, called **`void`**.

When a program is compiled, the function definition should occur in the program before the function is actually called. Typically, functions are defined near the beginning of the file. You write the function definitions, and then later you can make calls to that function. When the compiler comes along and encounters the function call, it sees a function name, with some arguments of various types, and it matches that up to the function with that name and those parameters.

Actually, it's slightly more complicated; all the compiler really needs to know at first is the function header. In fact, there are ways you can give just the function header information and put the body of the function elsewhere. You'll learn more about this in lecture 15.

Exercise 1

For each of the following situations, what would the return type be?

1. A function that tells you whether a certain set of measurements indicates danger or not.
2. A function that prints out a warning message if there is danger.
3. A function that computes the recommended dosage of a medicine given a person's age and weight.
4. A function that will get a person's address by reading it from a file.

Click here to see the solution.

Exercise 2

Imagine that you want a function to calculate how many days it is until the next leap year day, February 29. How could you write the function header for such a function?

Click here to see the solution.

// THE FUNCTION BODY

The body of a function is the set of instructions that you want to follow when the function is called. When you see a function call, you can think of it almost as if the commands in the body of the function definition are being stuck into the code right there.

Suppose you're going to write a tic-tac-toe program and that one thing you want to be able to do is print out help for someone who doesn't know the rules.

In this case, you can define a function called **print_help** to print that information. The function doesn't need any parameters, and it doesn't need to return anything. So, the function header will be just **void print_help()** (6).

Then, you can define the body of the function. Inside of the curly braces, you just have a set of **cout** statements to output the helpful information (f). You could put these all in one **cout**, or in even more **cout** statements, if you wanted to.

Then, in the **main** program, you can make a call to the function by just writing **print_help()** (18). When this happens, the text you had in the **cout** statements is printed.

Within the body of the function, you can declare variables, just like you would in the **main** program. And you can use this variable just like you would use a variable in the **main** program.

Each of the parameters also becomes a variable inside of the function's memory. The name in the variable is the name that you used for the parameter, and the variable is initialized with whatever value was given as an argument.

To return a value from a function, the command is just **return** and then the value being returned.

As soon as the function encounters the statement **return**, the function ends, and it returns that value to the **main** program.

The **return** command will return as soon as it's encountered. So, if you have multiple lines of code and encounter a **return** statement before you've executed all of them, then you just miss some lines.

Also, if a function has a void return type, then you just write **return**—no need for a value to be returned.

Or you can just leave off the **return** statement if the return type is void; the function just automatically returns at the end.

```
1   // Program 12_1
2   // Demonstrating a function body
3   #include<iostream>
4   using namespace std;
5
6   void print_help() {
7      cout << "The object of the
    game is to get three of your
    mark in a single row,"
8           << "a single column, or
    on a diagonal." << endl;
9      cout << "One player will use
    an 'X' mark, and one wil use an
    'O' mark." << endl;
10     cout << "The players will
    take turns placing a mark in an
    empty square in an 3x3 grid."
11          << endl;
12     cout << "The first player
    to get 3 marks in the right
    combination will win,"
13          << "but if neither does
    after all 9 cells are filled, the
    game ends in a tie."
14          << endl;                f
15  }
16
17  int main() {
18     print_help();
19  }
```

// CONCEPTUAL SEPARATION

Let's figure out how you would write a function that counts the number of words in a string.

One option is to take the string and turn it into a stringstream. Then, you could read individual strings from the stringstream until you reached the end. If you counted how many times you could read a word from the stringstream, that would be the number of words in the original string.

To implement this, you'll create a stringstream (10)—in this case, named **ss**—initialized to the string that was an input parameter, **str_to_count** (8).

Then, you'll declare 2 variables. The integer **count** will keep track of how many words you read in from the stringstream, so you initialize it to **0** (11). You'll also create a string variable named **word** to hold each word you read (12).

Next, you'll have a **while** loop that continues as long as you haven't reached the end of the stringstream (g). Inside the loop, you'll read in a new **word** and increment **count**. This will keep happening as long as the loop's condition is met—that is, until **ss** has nothing else.

Finally, you **return** the **count** (18).

In the **main** program, you can call the function **word_count** with a string argument (23) and you'll get back the number of words that are in that string.

```
1   // Program 12_6
2   // A more complete function: counting words in a string
3   #include<iostream>
4   #include<string>
5   #include<sstream>
6   using namespace std;
7
8   int word_count(string str_to_count) {
9      // Turn the string into a stringstream
10     stringstream ss(str_to_count);
11     int count = 0;
12     string word;
13     // Count the number of individual words by reading from the stringstream
14     while (!ss.eof()) {
15        ss >> word;
16        count++;
17     }                                    g
18     return count;
19  }
20
21  int main() {
22     string s = "This is a string that has eight words.";
23     cout << "There are " << word_count(s) << " words in the string." << endl;
24  }
```

```
1   // Program 12_7
2   // A more complete function: counting words in a string, alternative
3   #include<iostream>
4   #include<string>
5   #include<sstream>
6   using namespace std;
7
8   int word_count(string str_to_count) {
9      // Count number of spaces
10     int count = 0;
11     int i;
12     for (i = 0; i < str_to_count.size(); i++) {
13         if (str_to_count[i] == ' ') {
14             count++;
15         }
16     }
17     // Number of words is one more than number of spaces
18     return count + 1;
19  }
20
21  int main() {
22     string s = "This is a string that has eight words.";
23     cout << "There are " << word_count(s) << " words in the string." << endl;
24  }
```

h

That isn't the only way you could implement that function. Another option would be to count the spaces between words. If you can assume that there's just one space in between each word, then the number of words in the string will be just one more than the number of spaces in the string.

To implement the function this way, you'll first declare a **count** variable (10) and a variable **i** (11).

Then, you'll loop through all the characters of the string, one by one, letting **i** take on the values from **0** to one less than the string length (h).

You'll check each character to see if it is a space (13). If so, you'll increment **count** (14).

Finally, you'd **return count+1** (18), remembering that there will be one less space than the number of words.

In terms of the **main** function that calls the **word_count** function, it is exactly the same as before (23).

There's actually a problem with this implementation. If you just count the number of spaces, you'll count extra words when you have double spaces.

You can make a small change—to verify that both the character is a space and that the one before it was not a space—to fix this (i). In this way, if there are a bunch of spaces together, only the first one counts for counting a word.

This illustrates one of the main benefits of functions: the conceptual separation. Notice that in the **main** routine, you didn't need to worry about how the function was written; you just made a call to **word_count** and expected it to give you a result. You could have 2 totally different implementations of the function, but they'd look exactly the same when they were being called.

But you do have to be careful when you write the functions that you don't make hidden assumptions—such as the single space between words—that would make the function behave differently than expected.

```
1   // Program 12_8
2   // A more complete function: counting words in a string,
3   // alternative handling multiple spaces
4   #include<iostream>
5   #include<string>
6   #include<sstream>
7   using namespace std;
8
9   int word_count(string str_to_count) {
10     // Count number of spaces
11     int count = 0;
12     int i;
13     for (i = 0; i < str_to_count.size(); i++) {
14         // Only count if not a consecutive space
15         if (str_to_count[i] == ' ' && str_to_count[i - 1] != ' ') {
16             count++;
17         }
18     }
19     // Number of words is one more than number of spaces
20     return count + 1;
21  }
22
23  int main() {
24     string s = "This is a string that has   eight words.";
25     cout << "There are " << word_count(s) << " words in the string." << endl;
26  }
```

i

Exercise 3

Write a function that takes in a vector of integers and returns the largest one.

Click here to see the solution.

// SCOPE

The last thing to keep in mind when writing your own functions is the idea of **scope**, which is actually relevant for almost all of programming. Scope refers to the range of a program in which a variable is defined. If a variable is defined in a part of a program, it's said to be "in scope"; if it's not defined in a part of a program, it's said to be "out of scope."

In the way you've usually been thinking about programs so far, a variable is in scope as soon as it's declared, and it stays that way until the end of the program. You can't use it before you declare it, and once it's declared, it can be used in the rest of the program.

But that's not actually the case. In reality, scope works the way loops work: A variable is in scope only within its own section (denoted by curly braces) of the program.

In **Program 12_ERROR_1**, notice that the variable **a** is declared inside the main body of the **main** function, so it is in scope throughout **main**. But you declare a variable, **b**, inside the body of a loop that is in scope only within that body of the loop. In fact, each iteration of the loop generates a whole new variable, because the previous one's scope ends as soon as it reaches the end of the loop. So, when you try to print out **b** from outside the loop, you'll get a compiler error that tells you the variable is out of scope.

Notice that this means that the variables defined in the **main** routine are not in scope inside the black box of another function you might create. For example, **Program 12_ERROR_2** will give an error when compiling. The variable **a** is defined in the **main** function, so variable **a** is in scope there. But when you try to use variable **a** inside another function, it will be out of scope (7).

```
1   // Program 12_ERROR_1
2   // Scope Error: b is only defined inside the
    while loop
3   #include<iostream>
4   using namespace std;
5
6   int main() {
7      int a = 3;
8      while (a > 0) {
9          int b = 10;
10         a -= b;
11     }
12     cout << b << endl;
13  }
```

```
1   // Program 12_ERROR_2
2   // Scope Error: a is only defined inside main
3   #include<iostream>
4   using namespace std;
5
6   void demo_func() {
7      cout << a << endl;
8      return;
9   }
10
11  int main() {
12     int a = 3;
13     demo_func();
14  }
```

One of the nice things about scoping is that you can reuse variable names in different places without causing conflicts.

In **Program 12_10**, notice that you have the variable **balance** declared in the **main** function that is used to refer to an account balance (13). You also have another variable named **balance** declared in the function **calculate_loan** (7), this time referring to a remaining loan balance.

What happens here is that there are 2 separate sections of memory: the **main** function's section of memory and a section of memory for **calculate_loan**, each of which has a variable named **balance**. When you are in the **main** routine, any reference to **balance** will refer to its variable **balance**, and when you're in **calculate_loan**, you'd be referring to its variable **balance**.

So, the output statements from the **main** routine always show that the value of **balance** is **75**, both before and after the function call. Meanwhile, the output statement in **calculate_loan** shows that the value of **balance** is **475**.

```
1   // Program 12_10
2   // Same variable names in different scopes
3   #include<iostream>
4   using namespace std;
5
6   float calculate_loan(float principal, float payment) {
7      float balance = principal - payment;  // Remaining balance of loan after payment
8      cout << "Inside the function, the value of the loan balance is: " << balance << endl;
9      return balance;
10  }
11
12  int main() {
13     float balance = 100.0; // Set up a variable to keep track of an account balance
14     float loan_principal = 500.0;
15     float loan_payment = 25.0;
16
17     // Now use that to pay a loan
18     balance -= loan_payment;
19     cout << "Before the function call, the value of the account balance is: "
20         << balance << endl;
21     loan_principal = calculate_loan(loan_principal, loan_payment);
22
23     cout << "After the function call, the value of the account balance is: "
24         << balance << endl;
25  }
```

You can actually declare variables outside of the **main** routine! These are called **global variables**, and they are in scope for the entire program—in every function.

```
1   // Program 12_11
2   // Use of a global variable
3   #include<iostream>
4   using namespace std;
5
6   int a;
7
8   void my_function() {
9       a++;
10  }
11
12  int main() {
13      a = 1;
14      cout << "Before the function call, the
    value of a is: " << a << endl;
15      my_function();
16      cout << "After the function call, the value
    of a is: " << a << endl;
17  }
```

That said, it's much better practice to use **local variables**; that is, only use variables declared in the current function and pass information between functions through parameters and return values. That way, you can see exactly what is going into and coming out of a function by looking at the header. ◆

```
1   // Program 12_12
2   // Use of local variables only
3   #include<iostream>
4   using namespace std;
5
6   void my_function(int& x) {
7       x++;
8   }
9
10  int main() {
11      int a = 1;
12      cout << "Before the function call, the
    value of a is: " << a << endl;
13      my_function(a);
14      cout << "After the function call, the value
    of a is: " << a << endl;
15
16  }
```

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, section 4.5.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 6.1 and 6.3.

Exercise 1 Solution

1 The function would need to return a Boolean; it is trying to tell whether something is true—there is danger—or not.

2 That would be a void return type. There's nothing to return; the function just outputs some information.

3 That would probably be a float or double return type; it's computing a numerical value and needs to return a numerical value.

4 That would probably be a string return type. The address would need to be returned, and that would likely be in a string variable.

Click here to go back to the exercise.

Exercise 2 Solution

Here's one possible solution.

```
int days_to_leap_day(int day, int month, int year)
```

Click here to go back to the exercise.

Exercise 3 Solution

Here's what that might look like.

```
1   // Program 12_9
2   // Function to find largest value in vector
3   #include<iostream>
4   #include<vector>
5   using namespace std;
6
7   int find_largest(vector<int> v) {
8       int largest = v[0];
9       for (int i = 1; i < v.size(); i++) {
10          if (v[i] > largest) largest = v[i];
11      }
12      return largest;
13  }
14
15  int main() {
16      vector<int> test_data = { 12, 35, 21, 10, 18 };
17      int large_one = find_largest(test_data);
18      cout << large_one << endl;
19  }
```

Click here to go back to the exercise.

// QUIZ

1 What is output from the following program?

```
#include<iostream>
using namespace std;

void test_func(int a) {
    cout << a*a << endl;
}

int main() {
    test_func(2);
    test_func(3);
}
```

2 What would be the function header for a function named **get_distance** that takes in 2 integer parameters and returns a floating-point value?

3 Write a function that will take in a number of weeks and a number of days and return the total number of days.

4 What is the output of the following code?

```
#include<iostream>
using namespace std;

int test_func(int value) {
    value = 10*value;
    cout << "Value in function is: " << value
<< endl;
    return value;
}

int main() {
    int value = 3;
    int argument = 2;
    int retval;
    retval = test_func(argument);
    cout << "Value is: " << value << endl;
    cout << "Argument is: " << argument << endl;
    cout << "Return Value is: " << retval << endl;
}
```

Click here to see the answers.

// QUIZ ANSWERS

1 Here is the output:

```
4
9
```

2 Here is one possibility:

```
float get_distance(int a, int b)
```

Note that the parameters could be named something besides **a** and **b**.

3 Here is one option:

```
int num_days(int weeks, int days) {
  return 7*weeks+days;
}
```

Notice that the function name indicates what it is computing (number of days), and it takes in 2 parameters: the number of **weeks** and **days**. It returns an integer. The body of the function can be just a single line: the number of weeks times **7**, plus the number of days. This value is returned from the function.

4 Here is the output:

```
Value in function is: 20
Value is: 3
Argument is: 2
Return Value is: 20
```

Notice that neither **value** nor **argument** is changed by the function call. Within the function, the variable **value** refers to a different "box" of memory than the variable **value** from the **main** program. So, when you call the function, the value of **argument** (**2**) is copied into the local variable **value** inside the function. That value is multiplied by **20**, is output, and then is returned, where it is assigned to **retval**. Neither **value** nor **argument** in the **main** program is changed.

Click here to go back to the quiz.

13 Expanding What Your Functions Can Do in C++

As you've learned, functions help you separate parts of your code, providing a break between different ideas, and they let you avoid having to think about the code for a problem you've already solved. But there are other things you can do with functions that you haven't learned how to do yet, including handling cases where functions have different parameters and modifying big pieces of data.

IN THIS LECTURE:

// OVERLOADING FUNCTIONS

This is one way to write a function that returns the average of 3 numbers.

What if you want to modify this function? What if you really wanted the average of just 2 numbers or wanted to handle averages of 4 numbers?

```
1   // Program 13_1
2   // Function review: function to average 3 numbers
3   #include <iostream>
4   using namespace std;
5
6   float average(float a, float b, float c) {
7      return (a + b + c) / 3.0;
8   }
9
10  int main() {
11     cout << "The average of 10, 35, and 17 is: " << average(10, 35, 17) << endl;
12  }
```

You can create 2 new functions, each with the same name, **average**, but with different numbers of parameters. This approach is called **function overloading**—creating more than one implementation with the same name.

You'll create a version of **average** that has just 2 parameters: **a** and **b**. Everything is basically the same; you just return the sum of the 2 parameters divided by **2.0** (a).

You'll also create a version of **average** that has 4 parameters, **a** through **d**. Again, the function is basically the same, with just one more parameter and the average computing the sum of 4 parameters divided by **4.0** (b).

When you call **average**, you can then call it with either 2, 3, or 4 parameters. And the compiler will know to call the appropriate function, whether it's the one with 2, 3, or 4 parameters.

Notice that when you overload a function, it's because it makes sense to use the same name for all the variations; you don't want to overload functions when the different variations will do very different things.

The function's header defines the function **signature**, which is the name of the function but also specifies the number and types of parameters the function by that name will take in. When there is a call to a function, the compiler will match the call to the same function signature—the one that has the right name, the right number of parameters, and the right types of parameters. The names of the parameters don't matter in the signature; it's just their number and types.

In the **Program 13_2**, you have a function named **average** that takes 2 floating-point parameters, another function named **average** that takes 3 floating-point parameters, and another function named **average** that takes 4 floating-point parameters. These function implementations are all distinct from each other.

While the signature includes the function name and the number and types of parameters, it does not include the return type. So, it's not possible to overload by having 2 functions of the same name just returning different types—for example, one returning a string and one returning a float.

However, you can convert from one type to another when a function is called. If you call a function with parameters that don't exactly match the signature but can be automatically converted to ones that match the signature—such as converting integers to floats—then the conversion is automatic. That only works if there's just one possible matching signature.

No 2 functions are allowed to have the same signature, because the compiler needs to be able to distinguish between functions.

```
1   // Program 13_2
2   // Multiple parameter options
    with different function
    signatures
3   #include <iostream>
4   using namespace std;
5
6   float average(float a,
    float b) {
7      return (a + b) / 2.0;
8   }                                  a
9
10  float average(float a, float b,
    float c) {
11     return (a + b + c) / 3.0;
12  }
13
14  float average(float a, float b,
    float c, float d) {
15     return (a + b + c + d) / 4.0;
16  }                                  b
17
18  int main() {
19     cout << "The average of 10,
    and 35 is: " << average(10, 35)
    << endl;
20     cout << "The average of 10,
    35, and 17 is: " << average(10,
    35, 17) << endl;
21     cout << "The average of
    10, 35, 17, and 21 is: " <<
    average(10, 35, 17, 21) << endl;
22  }
```

```cpp
// Program 13_3
// Multiple parameter types
#include <iostream>
using namespace std;

float average(float a, float b) {
   return (a + b) / 2.0;
}

int average(int a, int b) {
   return (a + b) / 2;
}

int main() {
   cout << "The average of 10 and 35 is: " << average(10, 35) << endl;
}
```

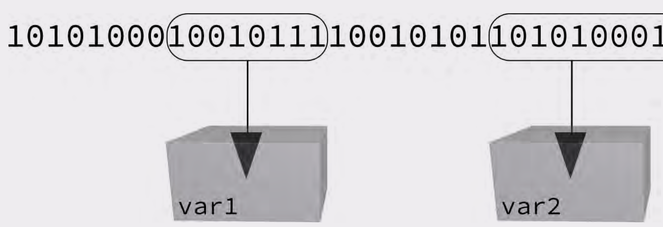
c

d

In this variation, you have the original **average** function that takes in 2 floats (c). You also create another version that takes in 2 integers. It also returns an integer, instead of a float. In this case, you'll perform integer division in the interior; because the parameters are integers and you divide by **2**, not **2.0**, you'll have an integer result (d).

If you call the function with 2 integers, the compiler will match it up with the function signature that has 2 integers. Because this is an exact match, that is the function it will use. There won't be any ambiguity because there's nothing that needs to be converted. So, in this case, a call with 2 integers, **10** and **35**, gives an integer result, **22**, rather than the floating-point result of **22.5**.

VARIABLE TYPES IN C++



For a computer, each variable is a way to interpret some bits in memory—a group of 1s and 0s that gets interpreted to mean an integer or a floating-point number, for example. Each variable also has some size of memory that it is taking up.

INTEGERS

`char`

`short int`

`int`

`long int`

`long long int`

FLOATS

`float`

`double`

`long double`

In C++, you can specify, to a degree, how many bits are used to represent numbers. The more bits that are used, the larger the number that can be represented. For integers, that means that you can represent larger integers. For floating-point numbers, that means that you can use more digits of precision and can represent both much larger and smaller numbers.

For integers, you have several options, ranging from a **`char`** (a single character) all the way up to a **`long long int`**. The exact sizes of integers aren't specified, but you do know that a **`long long int`** has at least as many bits as a **`long int`**, which has at least as many as an **`int`**, and so on. Choosing larger integers lets you represent larger numbers, but at the cost of more memory to store values.

Similarly, floats have a longer version, the **`double`**, and an even longer version, the **`long double`**.

When passing parameters, remember that the different types are actually different. And that means that if you have a function expecting one type and call it with a different type, it often needs to perform a type conversion in order to match the call to a specific signature. Some of these type conversions can be very tricky.

This is the same program from before, but now you're calling it with **10.0** and **35.0** rather than **10** and **35**.

When you compile this code, you'll probably find that you get a compiler error, saying that you're trying to make a call to a function named **average** and are giving 2 doubles as parameters.

The compiler can't figure out if it's better to convert those to floats and use the **float** version of **average** or to convert those to integers and use the **int** version of **average**. It's ambiguous what's wanted, so it won't compile the program.

Even though you thought you had called with floats, floating-point literals are treated as doubles by default.

One way to get around this is to make sure you call with floating-point numbers. Any time you write a floating-point literal, you can append an **f** onto the number to indicate that the number should be treated as a float:

```
15      cout << "The average
    of 10 and 35 is: "
    << average(10.0, 35.0)
    << endl;
```

```
15      cout << "The average
    of 10 and 35 is: " <<
    average(10.0f, 35.0f)
    << endl;
```

There are similar methods for other types; for example, adding ll after a number ensures that it's treated as a **long long int**.

Now this code will compile and run just fine.

```
1   // Program 13_3_ERROR
2   // Multiple parameter types; Error when
    trying to use floats due to double
3   #include <iostream>
4   using namespace std;
5
6   float average(float a, float b) {
7       return (a + b) / 2.0;
8   }
9
10  int average(int a, int b) {
11      return (a + b) / 2;
12  }
13
14  int main() {
15      cout << "The average of 10 and 35
    is: " << average(10.0, 35.0) << endl;
16  }
```

// SETTING DEFAULT PARAMETERS

There's one more way that you can provide functions with the same name but different parameter sets: You can specify a default value for one or more of the parameters. If, when the function is called, the corresponding argument is left out of the call, you use the **default parameter** instead.

```
1    // Program 13_5
2    // Using a Default Parameter
3    #include <iostream>
4    #include <string>
5    using namespace std;
6
7    void printmultiple(string s, int
     times = 1) {
8       for (int i = 0; i <
     times; i++) {
9             cout << s << endl;
10      }
11   }
12
13
14   int main() {
15      printmultiple("Hello", 5);
16      printmultiple("World");
17   }
```

To do this, you just set an initializing value for the appropriate parameter, assigning it some value inside the parentheses. When it is done this way, that assignment is understood by the compiler as one to be used if the corresponding parameter is not provided.

Suppose you want to create a function, called **printmultiple**, to print out a string multiple times. Let **printmultiple** take in the string to print and the number of times to print it. There's nothing to return; that is, the function does not need to return a value to the place it was called from, so you'll have a **void** return type.

If someone doesn't specify how many times to print the string, you might choose to have a default number, such as **1**. If so, your header would look something like line 7. So, there are 2 parameters: a **string** called **s** that you are taking in and an **int** called **times** (the number of times to print, which has a default value of **1**).

If you call the function with both parameters—for example, **Hello** and **5**—you'll get the string **Hello** printed out 5 times.

On the other hand, if you call it with just a string parameter, such as **World**, then **times** has the default value, so the string is printed out just once.

If there are multiple default parameters, then arguments from the function call are assigned to parameters from left to right (the first argument is assigned to the first parameter, the second to the second parameter, etc.). You can't have just some parameter in the middle be assigned a default value; once one is assigned, everything after it must also be default.

Default parameters are commonly used to control how a function works when there's a normal, or common, way to do something set as the default but you want to let the programmer have the option of selecting some other behavior if needed.

Setting default parameters is an alternative to overloading a function by specifying different signatures.

// USING REFERENCES

References allow you to share information without making needless copies.

A **reference** is a way of describing a particular variable. Normally, when you define a variable, it gets its own box of memory. If you do something to that box of memory, such as assign a value to it, it's not going to affect any other box.

With a reference, though, you don't create a new box of memory for the variable. Instead, you use an existing box of memory and say that the variable *refers* to that box of memory. Basically, your variable becomes another name for some box of memory that already exists; anything you do to that new variable must affect the original box of memory, too!

The way you specify a reference is by putting an ampersand (&) after the type when you first declare it. You then have to initialize your new variable to refer to some other variable. **Program 13_7** illustrates this.

First, you create a variable, **a**, that you initialize to a value of **1** (7).

Next, you create a variable, **b**, that is a reference to an integer. So, you write **int&** **b**. You have to make this a reference to something; you can't just leave it there. So, you initialize this to **a** (8). This means that **b** is going to refer to the exact same box of memory that is **a**. When you print out the values of **a** and **b**, both show up as **1**.

You can assign a new value, **2**, to **a** (10). When you do so, this changes the value stored in that box of memory. So, when you print out both **a** and **b** now, both show up as **2**.

The same thing happens if you assign a new value, **3**, to **b** (12). The box of memory that **b** refers to has its value changed to **3**. So, printing out the values of **a** and **b** now indicates that both have the value **3**.

Essentially, **a** and **b** are referring to the exact same thing.

The most common way that references are used is when calling functions. That's because you can declare any of the parameters in a function to be a reference. This means that the parameter will not be a copy of the argument that comes in; it'll be a reference to the argument that comes in. So, any changes to the reference parameter will be like a change to the original.

```
1   // Program 13_7
2   // Reference example
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      int a = 1;
8      int& b = a;
9      cout << a << " " << b << endl;
10     a = 2;
11     cout << a << " " << b << endl;
12     b = 3;
13     cout << a << " " << b << endl;
14  }
```

The term used to describe this is ***passing by reference***. This is distinguished from what you did previously, which is ***passing by value***. When you pass by value, you're copying the value of the argument into the new box of memory set up for the parameter. When you pass by reference, you don't copy anything; you just let the parameter refer back to a shared version of input.

Suppose you have a savings account balance and you'd like to increase the balance by some percentage to account for earning interest.

You can create a function named **increase_percentage** that will take 2 parameters: the value that you want to increase and the percentage to increase by. The key thing here is that you are going to modify the balance so that the first parameter will be a reference, not just a basic parameter. So, you will write **float& value** for the first parameter.

So, when the function is called, with arguments **savings_balance** and **10**, the first argument, **savings_balance**, is passed by reference. In the function's memory, the parameter **value** just refers to the exact same block of memory as **savings_balance** from the **main** function. The second argument, **10**, is passed by value, so a new box of memory named **percentage** is created in the function's memory area, and **10** is copied into there.

Thus, when you modify the variable named **value** in the function, you're actually modifying the same box of memory as **savings_balance** in the **main** function. After the function returns, you'll find that the value in **savings_balance** has in fact been changed.

Passing by reference makes it possible to modify a value. It also has the advantage of not having to go through the process of copying the variable.

Copying is not a big deal in most cases, but if the variable is something really large, such as an entire video file, you don't want to take the computational time and memory of going through and copying it.

The ability to not have to copy information needlessly is an advantage, so why wouldn't you always do that? Why wouldn't you always just pass by reference?

There are 2 main reasons:

1. One is if you know you want to modify the parameter in your function but don't want it affected back in the **main** program. You'd like the function to be as disconnected from where it was called as possible, so you don't want to needlessly tie things together that shouldn't be tied. Passing by value is the right solution when you want your function to be as much of a black box as possible, because you need to make a copy of that data regardless.
2. The other reason for passing by value is a little subtler: If you pass by reference, then the argument that you call with needs to be a variable. It can't be a literal—because there's no box of memory to refer to. If you try to pass in a string literal, you will get a compiler error.

```
1   // Program 13_8
2   // Passing by Reference
3   #include<iostream>
4   using namespace std;
5
6   void increase_percentage(float&
    value, float percentage) {
7      value += value * (percentage
    / 100.0);
8   }
9
10  int main() {
11     float savings_balance =
    1000.0;
12     cout << savings_balance
    << endl;
13     increase_percentage(savings_
    balance, 10);
14     cout << savings_balance
    << endl;
15  }
```

Exercise

Suppose you have a really long string and want to do a search-and-replace for some word in the string, finding every occurrence of that word and replacing it with a different word. Can you come up with a header for a function that will modify the string in that way? (Don't worry about the body of the function.)

Click here to see the solution.

There is a workaround that lets you pass a literal by reference. The trick is to add **const** in front of the reference parameter. This tells the compiler: "I don't need to copy this data, and I won't modify it. It will stay constant." This is called passing by const reference. Passing by const reference lets you avoid unnecessary copying and can help optimize code when you're dealing with large data.

Another way **const** can be used is when you have a variable that you want to assign a value to and you know that the value will never change. In that case, you can declare it to be a **const** variable when you declare and initialize it. You just write **const** in front of the type. Declaring a variable to be constant ensures that the intent of the variable—as an unchanging value—is maintained.

The use of pass by reference is also a way, in effect, of returning more than one value. Ordinarily, a function can only have one return type, but if you have multiple parameters that are passed by reference, you can use that to modify the values of each of them. In this way, a function can return multiple values—by modifying multiple arguments. ◆

When you're deciding when to overload functions, when to pass by value, and when to pass by reference, a good rule of thumb is to pass by value as a default, unless you have a good reason not to. That way, you're less likely to have errors or run into problems.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, section 8.5.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 6.2, 6.4, 6.5, and 6.6.

Exercise Solution

Here's what the header could look like. (Note that the order of parameters is up to you.)

```
1   // Program 13_9
2   // Passing by Reference - Incomplete program
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   void search_and_replace(string& main_string, string word_
    to_find, string replacement) {
8       // Manipulate main_string here
9   }
10
11  int main() {
12      string long_document;
13      // Assign something to long_document here
14      search_and_replace(long_document, "Robert", "Bob");
15  }
```

Click here to go back to the exercise.

// QUIZ

1 What do you call each of the following types of parameter passing? Assume that you call a function with argument `X` and the parameter in the function is `Y`.

a `Y` and `X` are exactly the same, but you're not allowed to change `Y`.

b `Y` gets assigned a copy of `X`, so anything done to `Y` does not change `X`.

c `Y` and `X` are exactly the same, so anything done to `Y` is also done to `X`.

2 Write a function header that would behave the same for all of the following calls:

```
float a, b, c, d;
a = compute_cost();
b = compute_cost(10.0);
c = compute_cost(10.0, 8.25);
d = compute_cost(10.0, 8.25, "Texas");
```

3 What would be the output of the following program?

```
#include<iostream>
using namespace std;

void A(int x) {
    x++;
    cout << x << endl;
}

void B(int& x) {
    x++;
    cout << x << endl;
}

int main() {
    int x = 2;
    A(x);
    cout << x << endl;
    B(x);
    cout << x << endl;
}
```

Click here to see the answers.

// QUIZ ANSWERS

1 These are all different ways you can pass parameters into a function; each has its own advantages.

a This is pass by const reference.

b This is pass by value.

c This is pass by reference.

2 You need to use default parameters here. The function is returning a value that gets assigned to a floating-point variable, so you make the return type **`float`**. The name of the function is **`compute_cost`**. There are 3 parameters, and they have to be able to take default values (**`10.0`** for the first parameter, **`8.25`** for the second, and **`Texas`** for the third). So, the parameters need to be of type **`float`**, **`float`**, and **`string`**, and the defaults need to be set appropriately; the actual parameter names don't matter, though. Here is an example:

```
    float compute_cost(float val1 = 10.0, float val2 = 8.25,
string state = "Texas")
```

3

```
3
2
3
3
```

Notice that the value of **`x`** is initially **`2`**, and when passed to **`A`**, the parameter has the value **`2`**, which is incremented to **`3`** and then output. This was a pass by value (so that **`x`** was copied into the parameter), so when you output **`x`** in the **`main`** program, its value is still **`2`**. Next, you call function **`B`**, which is a pass-by-reference call. The parameter thus refers to the same memory location as **`x`** in the **`main`** program. So, when you increment the value to **`3`**, it is printed as **`3`** in the function but is also changed to **`3`** back in the **`main`** program.

Click here to go back to the quiz.

14 Systematic Debugging, Writing Exceptions

Although everyone would like their programs to run perfectly as soon as they're written, that's never actually the case. So, every programmer needs to come to terms with the fact that a lot of programming time will be spent dealing with **errors**.

IN THIS LECTURE:

// A SYSTEMATIC APPROACH TO DEBUGGING

To avoid frustration and failure, you should approach the process of debugging methodically, following 6 main steps:

1. isolate the error,
2. narrow in on the failure point,
3. identify the problem,
4. fix the problem,
5. retest, and
6. consider similar cases where the error you've fixed might reappear elsewhere in your program.

STEP 1: ISOLATE

The first step is to isolate an error. This involves writing good test cases, which helps make sure that the code is running as expected. You want to run tests until you either cannot come up with any tests that give problems or find a test that gives a repeatable error. It's important for the error to be repeatable; you want to make sure that every time you run the program with that test, you get the same incorrect result.

Debugging is a methodical process. It's not just a matter of randomly jumping through your code to find errors and make bugs disappear.

A common version of the test case idea is to set up a **unit test**, which is a test case run for a particular function. Basically, you have some input to the function and then some output you should expect to see. When you write a function, it's a good idea to run several unit tests to make sure the function is working as expected.

This program is designed to let a user enter some text and then it will check whether that text is a palindrome or not (a). It does this by calling a function **is_palindrome** (b), which calls a function **remove_spaces** to remove spaces from the text (c). Then, it goes over the first half of the characters (21), comparing them to the corresponding character on the other side (22). If they all match, then it's a palindrome; otherwise, it's not.

Because you're focused on the **is_palindrome** function, you can consider this a unit test of the **is_palindrome** function itself. You want to give the **is_palindrome** function an input and check that it's giving the right output.

For your unit test, you'll check to see if your function works for a very simple text string: **aaa**. That should be a palindrome, but when you run the program to check it, you get the wrong answer!

```
1   // Program 14_1
2   // Example program to debug - contains an error!
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   string remove_spaces(string s) {
8      string ret = "";
9      int i;
10     for (i = 0; i < s.size(); i++) {
11        if (s[i] != ' ') {
12           ret += s[i];
13        }
14     }
15     return ret;
16  }
17
18  bool is_palindrome(string s) {
19     string no_spaces = remove_spaces(s);
20     bool could_be_palindrome = true;
21     for (int i = 0; i < no_spaces.size() / 2; i++) {
22        if (no_spaces[i] != no_spaces[no_spaces.size() - i]) {
23           could_be_palindrome = false;
24        }
25     }
26     return could_be_palindrome;
27  }
28
29  int main()
30  {
31     cout << "This will tell you whether certain text is a
   palindrome.  Enter some text:"
32        << endl;
33     string user_input;
34     getline(cin, user_input);
35     if (is_palindrome(user_input)) {
36        cout << "It is a palindrome!" << endl;
37     }
38     else {
39        cout << "It's not a palindrome." << endl;
40     }
41  }
```

c

b

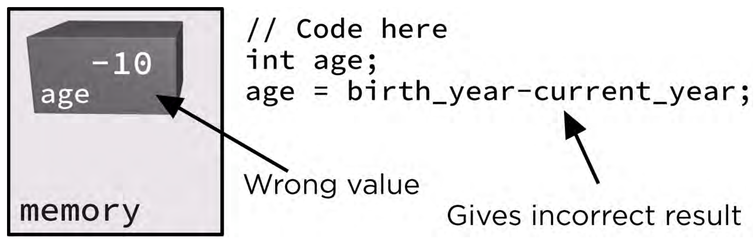
a

STEP 2: NARROW

Now that you have an error that you can use to debug your program, the next step in the debugging process is to hone in on where the failure is occurring in the code. This is where the debugger in Visual Studio (on a PC) or Xcode (on a Mac) can come in handy.

A debugger can only help you follow what's going on in the program; it's not going to solve your issues for you or even tell you where the error occurred.

Your goal is to find the first line of the program that gives a result that's incorrect. Usually, this will mean that some variable in memory has the wrong value.



If you don't have a debugger, you can always use output statements to print out data along the way.

To hone in on an error, you should start at a point in the program where you know everything is OK and one where you know things have gone wrong. In the worst case, that'll be the first and last lines of the program! But in many cases, you have a better idea—often right before calling some new function you wrote and right after it.

Then, you'll want to gradually narrow down the scope of the program where the error could be occurring. Pick another point in the program, somewhere between the last-known OK point and the first-known bad point and analyze whether the variables in memory are correct there. If everything looks good, then you can move your OK point to there. If there's a problem at that point, then move your bad point to there. The goal is to narrow down the location of the error where things first go wrong as much as possible.

In this case, you know that things are good at the beginning of the program and that there's a problem at the end, so you should check a point in the middle.

You could do this with just print statements by using a print-debug approach: inserting one line after the **remove_spaces** function has returned in order to output the result there.

```
19      string no_spaces =
    remove_spaces(s);
20      cout << "String with no
    spaces is:" << no_spaces
    << endl;
21      bool could_be_palindrome
    = true;
```

When you run the program, it appears that the data is correct to that point. This indicates that you did indeed read in the string and calling **remove_spaces** did not introduce any obvious errors. So, you can assume that up until that point, things are good.

You could likewise check the value that you are going to return right before returning. If you add another output line there, printing out the value you will return right before the return statement, you can check if the problem is before or after the return from the **is_palindrome** function.

```
26      cout << "About to return:"
    << could_be_palindrome<<endl
27      return could_be_
    palindrome;
```

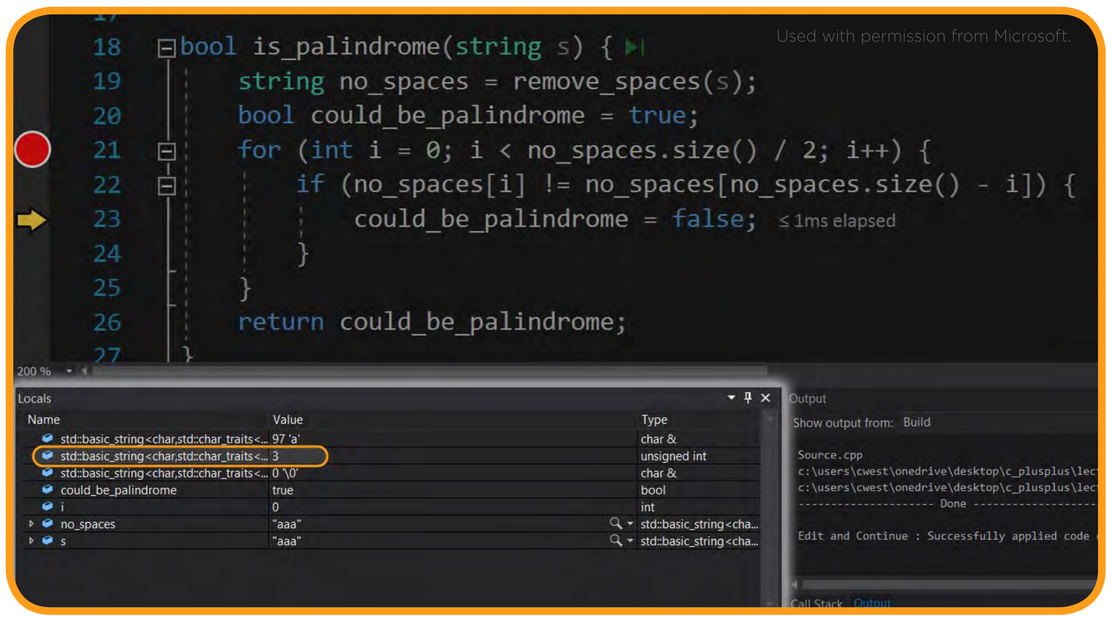
Sure enough, when you run the program, it's returning **0**, or **false**, which is not correct. So, you know that things seem good before the **for** loop and are wrong afterward. You need to hone in further.

In a debugger, you can set a breakpoint at the beginning of the loop that will make sure you stop there. When you run the debugger to there, looking at variables, **could_be_palindrome** is **true**, as it should be.

Stepping over the first line, you can check and see that **i** has the value **0**, as expected. So, things seem good so far.

Stepping over one more line, you have ended up inside the **if** statement and are about to set **could_be_palindrome** to **false**. But that's not right! You should not be getting to this point for this input.

This means that the error had to be on the previous line—the one where you are comparing **no_spaces[i]** with **no_spaces[no_spaces.size()-i]**. You can examine those values: **i** is **0**, as expected, and **no_spaces.size()-i** is **3**. But that's not right. You want to be looking at the last element of the string, but that should be element 2, not element 3.



STEP 3: IDENTIFY

Now that you know where the error is occurring, the third step is to identify the source of the error. The source is not just the line of code where things went wrong; it's what caused that line of code to be wrong. Was it a simple typo, or is it an error in the fundamental logic of the program?

The key point is that you want to understand what caused this error before you try to fix it. If it was a basic error like mistyping, then it'll be very simple to fix. But if it's a logic error in the program, you'll want to make sure you really understand what went wrong and why. There's no shortcut around this part of the debugging process.

In the palindrome program, for a string of 3 characters, you were trying to compare character 0 to character 3. The problem is that the last character would not start at the size of the string; it's actually the size minus 1, so you need to compare character 0 to character 2. So, in this case, the error was in coming up with the wrong formula to use for comparison.

STEP 4: FIX

Once you know the problem, the next step is to fix the bug. This could be simple—maybe just spelling a variable name correctly or changing a character.

It could involve rewriting a line of code. Or it could be that you were so wrong in your approach that the best fix is to throw out the whole thing and start over again!

Here, the fix for the bug is pretty simple: You just subtract 1 from the element number for the end of the string. So, when **i** is **0**, you should be comparing to **size()-1**. When **i** is **1**, you'll be comparing to **size()-2**, etc.

```
22    if (no_spaces[i] != no_spaces[no_spaces.size()-1-i]) {
```

STEP 5: RETEST

Once an error is fixed, you need to retest everything. Your goal is not just to show that the fix you made fixed the one problem you had isolated and identified. You want to make sure that you didn't break anything else in the meantime. If you've saved all your tests, now is the time to rerun them and make sure you still get correct results for everything that was previously correct—plus for the test you just fixed!

When you test the palindrome program on **aaa** now, it returns that it is a palindrome. You could check other tests: **abc** is not a palindrome; **bob** is a palindrome. The program seems to be working now.

STEP 6: CONSIDER

Before your debugging task is done, there's one final step: You should think about whether anywhere else in the code is likely to have a similar error.

Errors tend to multiply when there are different versions of the same calculation. Sometimes, people copy and paste code from one area to another and thereby duplicate any errors. Other times, if your thought process was off, you might have made the same conceptual error in other parts of the code.

The key is to spend time thinking about whether the fix you made would apply somewhere else, too. If so, fix and test that other part of your code as well.

For the palindrome program, you should think: "Were there any other places where I was assuming the string went all the way to size rather than size minus 1?" In this case, the answer is no, so you're done debugging.

```
1   // Program 14_2
2   // Example program to debug - fixed!
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   string remove_spaces(string s) {
8      string ret = "";
9      int i;
10     for (i = 0; i < s.size(); i++) {
11        if (s[i] != ' ') {
12           ret += s[i];
13        }
14     }
15     return ret;
16  }
17
18  bool is_palindrome(string s) {
19     string no_spaces = remove_spaces(s);
20     bool could_be_palindrome = true;
21     for (int i = 0; i < no_spaces.size() / 2; i++) {
22        if (no_spaces[i] != no_spaces[no_spaces.size() - 1 - i]) {
23           could_be_palindrome = false;
24        }
25     }
26     return could_be_palindrome;
27  }
28
29  int main()
30  {
31     cout << "This will tell you whether certain text is a
    palindrome.  Enter some text:"
32        << endl;
33     string user_input;
34     getline(cin, user_input);
35     if (is_palindrome(user_input)) {
36        cout << "It is a palindrome!" << endl;
37     }
38     else {
39        cout << "It's not a palindrome." << endl;
40     }
41  }
```

// TYPES AND SOURCES OF ERRORS

The errors that you encounter tend to fall into one of 3 types: syntax errors, logic errors, and runtime errors.

1. A syntax error occurs where you've written something that is not accurate code. These are usually the easiest to find and fix, because the compiler will identify that you have a mistake in your code and will usually give you a particular line of code where it detected the error.
2. A logic error occurs when there was something wrong with the thought process that went into the code, and that gets reflected in a mistake. These are some of the toughest errors to find.
3. A runtime error is due to unexpected circumstances. This is where adding **exceptions** to your program can be a big help.

Often, the source of the error could be due to circumstances outside the programmer's direct control. Perhaps you asked a user to enter a number and he or she typed a word instead of a number. The program will either give a wrong answer or just crash.

To avoid this kind of error, you could make a much more complicated process for reading in a string and then checking to see if the string is a positive integer—and if so, convert it to an integer, and so on. But doing all of this would be very painful!

Programming is full of situations, besides a user entering data of the wrong type, where you might run into results that you don't expect.

One common situation is with files: You might try to open a file for reading but the file just isn't there. Maybe the name got mistyped or the file is in a different directory. For whatever reason, it's not where you expected, and you can't open it.

For example, **Program 14_4** asks a user for a file name, reads the first element (an integer) from the file, and prints it out. If you run this program and type in the name of a non-existent file, you'll probably get some sort of nonsense printed out. Since the file did not exist, it could not be opened to read from, and so whatever should have been read and then printed out will be junk.

Trying to access a vector is another situation where what you expect might not be what you get. For example, maybe you read in a week's data but a user included only the workweek and not the weekend. What should happen if you tried to access the 6th element?

Program 14_5 is an example: The **hours_worked** vector has 5 elements, giving the number of hours worked each day. The loop, however, tries to access 7 elements.

```
1    // Program 14_4
2    // Exception example - reading a
     nonexistent file
3    #include <iostream>
4    #include <fstream>
5    #include <string>
6    using namespace std;
7
8    int main() {
9       fstream infile;
10      string filename;
11      cout << "What is the file
     name? ";
12      cin >> filename;
13      infile.open(filename,
     fstream::in);
14
15      int a;
16      infile >> a;
17      cout << "Read in: " << a
     << endl;
18   }
```

```
1    // Program 14_5
2    // Exception example - vector
     past end of range
3    #include <iostream>
4    #include <vector>
5    using namespace std;
6
7    int main() {
8       vector<float> hours_worked =
     { 7.5, 8.5, 10.0, 7.0, 7.0 };
9       for (int i = 0; i < 7; i++) {
10          cout << "On day " << i <<
     ", " << hours_worked.at(i) << "
     hours were worked."
11                << endl;
12      }
13   }
```

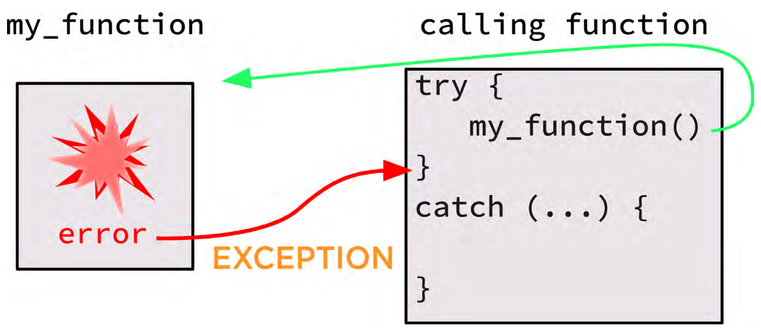
// USING EXCEPTIONS

The solution to dealing with unexpected, "exceptional" cases is through the use of exceptions. Here's how they work.

First, you have a function called from somewhere within a program. This call should occur within what's referred to as a **try** block.

Somewhere within the function that is called, you encounter an error. This is where you realize that some exceptional situation has occurred—maybe you've tried to access past the end of a vector or you had text entered instead of a number.

That function then generates an exception, which is basically a special note that something has gone wrong. In C++, the term for this idea is *throwing* an exception. In other programming languages, you might hear this called *raising* an exception.



When you throw an exception, the function immediately returns. The exception is being "thrown" back to the function that called the routine.

Back in the calling function, the exception is then caught. There is separate code that is executed to handle the exception that occurred.

If you're going to deal with exceptions, you need 2 parts of code. First, you have a **try** block, which consists of the keyword **try** followed by curly braces. Whatever code is inside the curly braces is something that you will "try" to execute.

If the code executes and no exception is thrown within that code, then you go on to whatever additional code follows. On the other hand, there could have been an exception raised.

Immediately following the **try** block will be one or more **catch** blocks—used to "catch" any exception thrown when trying to execute the code in the **try** block, including in any functions that were called from there. The **catch** block will take in a parameter, which is going to be the exception that is returned.

```
try {
  // Code to try
}
catch (exception& e) {
  // Code to handle exceptions
}
// Additional code
```

In many cases, you're going to be throwing your own exceptions. But there are a few places in standard C++ where exceptions are already thrown.

Let's see how to handle exceptions in 2 cases.

The **fstream** library supports the ability to throw an exception in the case of trying to read from a nonexistent file.

There is a special command that has to be run for the **fstream** first: the member function **exceptions**, which basically activates the ability for **fstreams** to throw exceptions. The **exceptions** function takes in as input the exceptions that are being activated.

In this case, the exception you would care about is a predefined exception called **failbit**: If the file does not open correctly, then a **failbit** exception is thrown. So, for this code, because the **fstream** is named **infile**, the command you'll write is **infile.exceptions(fstream::failbit)** (9).

Then, you have a **try** block (d), where the code you want to try to run is put inside the curly braces. In this case, the code you're trying is opening the file.

Following that is a **catch** block (e), which takes in one parameter—a reference to an exception, given the name **category** in this case.

Inside the **catch** block is the code you want to run if the exception is caught. In this case, you print out a message that there was an error opening the file. Then, you return a nonzero value. Because you're in the **main** function, **return** means that the program ends, and because it ended with a nonzero value, it indicates that there was an error along the way.

If you run this code and the file does not exist, then the **open** function you tried to call will generate an exception. And when this exception is caught, you print out **Error opening file**, and the program ends.

If there was not an exception, then you would have just skipped over the **catch** block and gone on to the remaining code in the **main** function.

```
1   // Program 14_6
2   // Exception example - reading a nonexistent file, now with
    try/catch
3   #include <iostream>
4   #include <fstream>
5   using namespace std;
6
7   int main() {
8       fstream infile;
9       infile.exceptions(fstream::failbit);
10      try {
11          infile.open("something.dat", fstream::in);
12      }                                                    d
13      catch (exception& category) {
14          cout << "Error opening file" << endl;
15          return 1;
16      }                                                    e
17      int a;
18      infile >> a;
19      cout << "Read in: " << a << endl;
20  }
```

```
1   // Program 14_7
2   // Throwing exceptions - throwing directly
3   #include <iostream>
4   #include <stdexcept>
5   using namespace std;
6
7   int process_boxes(int num_boxes) {
8       if (num_boxes < 1) {
9           throw exception();
10      }
11      // Do other stuff here
12      return 1;
13  }
14
15  int main() {
16      try {
17          process_boxes(-1);
18      }
19      catch (exception& e) {
20          cout << "Had an exception!" << endl;
21      }
22  }
```

f

When you're writing your own programs, you throw an exception simply by calling the **throw** command. You need to throw exceptions, so the syntax will be **throw exception()**. That creates an exception and throws it back to the original function.

Imagine you have some function, named **process_boxes** (7), that takes in an integer parameter. It makes no sense if that function is called with no boxes, so in that case, you might want to throw an exception.

Inside the function, you perform a check: If the number of boxes is less than 1, then you have an invalid input and you'll throw an exception (8).

To throw the exception, you just have **throw exception()** (9). An alternative is that you could create an exception variable, **exception bad_input**, and then you could throw that exception, **throw bad_input**.

```
9       throw exception();
```

```
9       exception bad_input;
10      throw bad_input;
```

> If you approach debugging steadily and systematically, you will find your errors.

Either way, you're creating an exception and then throwing it. In most circumstances, it's easier to just throw the exception in one line of code that both creates the exception and throws it.

If you didn't throw an exception, then the function would continue to do whatever else it needed to.

Back in the calling function, you can try calling the routine with an invalid parameter and catch the exception being thrown back (f), which results in printing a message that there was an exception.

Note that you had to #include **stdexcept** here. You didn't do this in earlier examples because including **vector** or **fstream** automatically included basic exceptions as well. But if you don't use another library that is using exceptions already, you need to #include it yourself.

Using exceptions can help make your code much more robust; that is, your code is able to deal with a wider range of exceptional circumstances cleanly. Exactly how you deal with those circumstances can vary from case to case.

- You could just close the program, maybe after printing some error messages.
- You could prompt the user for another input, if the source of the problem was user error.
- You could set a default value that gets used if a given value is invalid. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 5.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 5.6 and 18.1.

c Ousterhout, *A Philosophy of Software Design*, chap. 10.

Rather than just **`exception`**, there are specific exception types named **`logic_error`**, **`runtime_error`**, **`range_error`**, and so on. You can have separate **`catch`** blocks for each of the different types of exceptions. To do that, you just define the parameter in the parentheses after **`catch`** as having the more specific type of exception, rather than just the general **`exception`**.

// QUIZ

1 Put the following steps of the debugging process in order.

- **a** Retest.
- **b** Identify the source of the failure.
- **c** Isolate a repeatable error.
- **d** Fix the error at its source.
- **e** Consider similar cases.
- **f** Narrow in on the location of failure.

2 Imagine that you need to make a function call to a function named **`draw_square`**, which takes no parameters and returns nothing. But it may generate an exception. Write the code you would use to call the function but, if there is an exception, to print **`Can't draw square`**.

3 Write a function named **`tester`** that takes in 2 integer parameters and throws an exception if either one is negative or otherwise returns the sum.

Click here to see the answers.

// QUIZ ANSWERS

1

c Isolate a repeatable error. You first need to have a definite error that you can reliably check.

f Narrow in on the location of failure. Find the line where you can identify that things go wrong. Remember that the location of failure is where the bug manifests but is not necessarily where the actual bug in the code occurs.

b Identify the source of the failure. Based on the location the error occurs, try to determine what the actual source of the error is. This usually requires logic, deduction, and inference.

d Fix the error at its source. When you know the error, you can fix it at its source.

a Retest. Once the error has been fixed, you need to test again to make sure everything still works—to ensure you both fixed the bug and did not create a new error.

e Consider similar cases. Because an error in one location in the program might have been repeated elsewhere, before moving on, consider whether the error might have been made in other locations as well.

2

The code will need to use **try-catch** blocks. The function call should be inside of the **try** block, and the **catch** block will be used to catch the exception, with the output line inside that block.

```
1       try {
2           draw_square();
3       }
4       catch (exception& e) {
5           cout << "Can't draw square" << endl;
6       }
```

3

Here is a function. You need to have a conditional to check whether either parameter is negative and, if so, throw an exception. Otherwise, just return the sum.

```
1   int tester(int a, int b) {
2       if ((a<0) || (b<0)) {
3           throw exception();
4       }
5       return a+b;
6   }
```

Click here to go back to the quiz.

15 Functions in Top-Down and Bottom-Up Design

IN THIS LECTURE:

Functions let you write much bigger and more complex programs than would otherwise be possible, and the power of functions is even greater when integrated with a deliberate approach to software design. Functions integrate equally well with top-down and bottom-up design approaches.

// TOP-DOWN DESIGN

With top-down design, you take a larger problem and treat it as a set of smaller problems without worrying as much about how each of the smaller problems impacts the other smaller problems. Your job with any one task is to either break it into even simpler subtasks or implement it directly.

With functions, you have a very similar motivation. A function is a way of conceptually separating one set of functionality from everything outside. This is very similar to the way you treat each of the subtasks in top-down design.

So, if you consider the tree structure you create in top-down design, each of the nodes can become, in effect, a function of its own. Instead of merely turning the hierarchical tree into a series of comments, you can turn it into a collection of functions.

Remember that because you must have a function declaration before you call the function, this means that the functions listed earlier in the code will be those far from the root of the tree. The **`main`** function will correspond to the root of the tree and will come last.

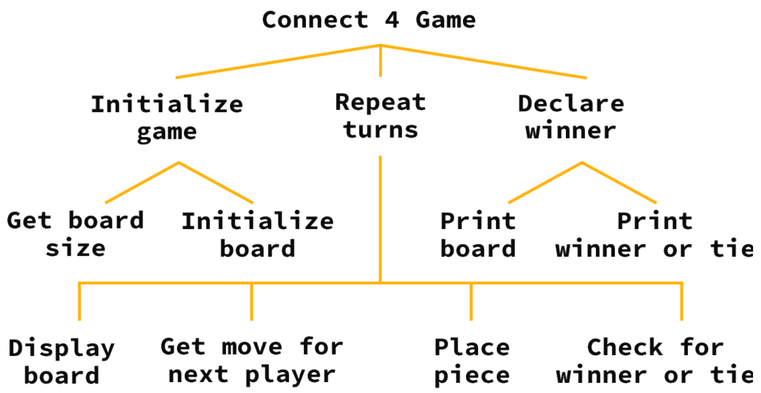
Let's see what this might look like by designing a program to play a game of Connect 4, which is played by 2 players, each with a different color piece, usually red or black. Players take turns dropping their pieces down columns of the board: When a piece is dropped, it travels as far down as it can, until it is either at the bottom or on top of another piece. The goal is to get 4 pieces in a row vertically, horizontally, or diagonally.

There are 3 main problems that a program created to play this game needs to solve. You need to

1. set up the board,
2. repeatedly take turns, and
3. determine and display the winner or note that it's a tie.

Next, you can break these problems into smaller tasks.

1. Initializing the game will involve getting the board size for the game and then initializing that board to be empty. Traditional Connect 4 is played on a board with 7 columns and 6 rows, but basically a board of any size could be used. You can get the board size from the user.
2. For each turn, you want to display the current board for the players, get the player's next move, make that move, and then check to see if someone won. These steps will happen repeatedly through the game
3. For declaring the winner, you want to print out the board one more time and then print out who the winner is.



While most of these elements are now about as simple as they can be, checking for a winner can be broken down further. You'll need to check for various win conditions: a vertical win, where someone has 4 consecutive pieces in the same column; a horizonal win, where someone has 4 consecutive pieces in the same row; and an increasing or decreasing diagonal, where a player has 4 consecutive pieces in an increasing or decreasing diagonal. If none of these are the case, then you should check to see if the board is full, in which case it's a tie game.

Notice that the process for checking for a winner is basically the same as the process for printing a winner. You have to make all the same checks in both cases. This is one of the places where you can take advantage of the fact that functions allow you to use the same code for multiple calls. You can write one set of functions that should be able to be used in both of the other functions.

You also have a display or print board node in 2 different places—both as something you do at the beginning of each turn and something you do before printing the winner. Again, you can write one function that gets called from 2 different locations, thus saving you coding time.

After you've broken down the game into a series of subtasks, you're going to turn each of the nodes from your top-down design into a function in code, with **main** as the root node. Initially, you'll just create these functions, not worrying about the return types or the parameters for each; you'll go back to adjust that as you fill in the functions.

For example, you can create functions for the first branch of the top-down hierarchy. You'll have one function to initialize the game. Under that are 2 more functions: one to get the board size and one to initialize the board. Because the function to initialize the game is higher in the hierarchy and will need to call the other 2 functions, it should come last.

You can do the same for all the branches from the beginning, or you can begin filling in these functions.

```
// Program 15_1_1
// Connect 4 Program - stage 1
#include <iostream>
#include <vector>
using namespace std;

void get_board_size() {
   /* Get size of board from
user */
}

void initialize_board() {
   /* Initialize the board
itself, to all empty squares */
}

void initialize_game() {
   /* Initialize the entire
game */
}

int main() {
}
```

Let's take the first function, **get_board_size**. Basically, you need to ask the user what size board to use and then return that size. The size will actually be 2 numbers: a number of rows and a number of columns. Because you need to return 2 values, you'll use a pass-by-reference approach.

The code for the function itself is very straightforward: You just output a few prompts and read in the number of columns and rows to use (a). These will get returned to the calling program using the pass by reference (b).

Remember that whenever you write a section of code, you should test it to make sure it's stable before moving on. When you finish a function, you want to run unit tests on it, testing that it works as it's supposed to. So, you just set up a call to the function in the **main** program and output what it returns (c). When you do this, you see that it's all working.

```
1   // Program 15_1_2
2   // Connect 4 Program - stage 2
3   #include <iostream>
4   #include <vector>
5   using namespace std;
6
7   void get_board_size(int& columns, int& rows) {
8      /* Get size of board from user */
9      cout << "How many columns should the board have? ";
10     cin >> columns;
11     cout << "How many rows should the board have? ";
12     cin >> rows;
13  }
14
15  void initialize_board() {
16     /* Initialize the board itself, to all empty squares */
17  }
18
19  void initialize_game() {
20     /* Initialize the entire game */
21  }
22
23  int main() {
24     int a, b;
25     get_board_size(a, b);
26     cout << "Returned " << a << " columns and " << b << " rows." << endl;
27  }
```

b

a

c

STUBS AND SCAFFOLDS

Notice that not all of the functions you intended to write had code. This approach—building a function that doesn't actually do anything—is somewhat common when developing programs. You might know that a program will need this function and might need to call the function from the code you're working on, but you haven't actually written that other function yet. A function that can be called and return some value of the correct type is called a **stub**.

Likewise, you sometimes need some other data or operations to be performed before you call a function, but you don't necessarily want to write all the code needed before that function you're working on is called. In this case, you can create a **scaffold**, which gets things set up so that a function you care about can be called and tested.

Now you can continue to fill in additional functions. For example, you can fill in the **initialize_board** function, which will need to take in the number of rows and columns and return a board. In this case, the board is going to be a vector of vectors. There will be one vector for each column and thus a vector of each of those. Each element of the board will be a **0** to indicate that the spot is empty (d).

Notice that the **initialize_game** function just calls the 2 other functions you created. It calls **get_board_size** to get the numbers of rows and columns and then **initialize_board** to actually create the empty board (e).

If you had wanted to do this in a different order—that is, write the **initialize_game** function first—then you'd have had to make stub functions for the **get_board_size** and **initialize_board** functions.

And continuing from there, you can fill in all the other functions corresponding to each node in the hierarchy that you created in your top-down design. You'd want to develop incrementally—to add one function's code and test it thoroughly before going on to write the next function.

The entire program is about 200 lines long—but those 200 lines are spread over a bunch of different functions. No single function is too large, and each one should be understandable on its own.

Spend some time looking at this code and seeing how it reflects the top-down design of the outlined program.

To view the entire program, go to TheGreatCourses.com/CPlusPlus.

```
1    // Program 15_1_3
2    // Connect 4 Program - stage 3
3    #include <iostream>
4    #include <vector>
5    using namespace std;
6
7    void get_board_size(int& columns, int& rows) {
8       /* Get size of board from user */
9       cout << "How many columns should the board have? ";
10      cin >> columns;
11      cout << "How many rows should the board have? ";
12      cin >> rows;
13   }
14
15   vector<vector<int> > initialize_board(int columns, int rows) {
16      /* Initialize the board itself, to all empty squares */
17      vector<int> column(rows, 0);
18      vector<vector<int> > board(columns, column);
19      return board;
20   }
21
22   vector<vector<int> > initialize_game() {
23      /* Initialize the entire game */
24      int c, r;
25      get_board_size(c, r);
26      return initialize_board(c, r);
27   }
28
29   int main() {
30      vector<vector<int> > board;
31      board = initialize_game();
32   }
```

d

e

Top-down design combines very easily with function definitions to make what would otherwise be a challenging problem much more tractable.

// BOTTOM-UP DESIGN

The idea of **bottom-up design** is to take existing things you know how to do and combine them to do something new. Then, you could use that to do something else, and so on. You work up the hierarchy, using individual functions as building blocks.

In this simple example that you might use in a word processor, you have a function, **pos_first**, that lets you find the first occurrence of some word in a string (f). And you have another function, **replace_string**, that lets you replace one section of a string with a different string (g). You can look at these functions and realize that you can use them to replace the first occurrence of one word with another word. So, you'll create a new function, **search_and_replace**, that will replace one string with another inside of that larger string (h).

```
1   // Program 15_2_1
2   // Example of bottom-up program - string processing (incomplete) -
    Stage 1
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int pos_first(string& string_to_find, string& string_to_search) {
8      /* Returns the first position of string_to_find in string_to_search,
9      or -1 if it is not in there */
10     // Code Here
11  }
12
13  void replace_string(string& string_to_modify, int start_pos, int end_pos,
14     string& string_to_insert) {
15     /* Inserts string_to_insert into string_to_modify, replacing anything
16     in positions start_pos to just BEFORE end_pos */
17     // Code Here
18  }
19
20  bool search_and_replace(string& string_to_modify, string& old_string,
21     string& new_string) {
22     /* Replaces first occurrence of old_string with new_string and returns
23     true, or returns false if it did not find old_string */
24
25  }
26
27  int main()
28  {
29     // Code Here
30  }
```

f

g

h

How might you fill in the code for **search_and_replace** based on what you have?

Here's how that function might be written. You can use the **pos_first** function to find the first position of the string you are searching for (24). If it's not found, you return **false** (i); otherwise, you identify where that string would end in the original string (28). Finally, you call **replace_string** to replace the string and return **true** (j).

And you can build up from there to create another level. You can combine the function you just had with a loop, which will let you replace every occurrence of one word in a string with a different one. You can create a new function, building up from the function you just created, that just repeatedly loops over the previously defined function until it returns **false**—that is, until it can't find the string to search for in the longer string (k).

Bottom-up design is a great way to add features to a piece of software. When you have a product that can already do one thing and then have some new idea that integrates well with it, you then can end up creating a new software product that incorporates an additional feature.

Bottom-up designs tend to focus on what is possible with what you have, rather than the top-down philosophy of how a particular problem is solved. As a result, bottom-up implementations tend to be useful in a wider range of applications. This makes the bottom-up method a good way of developing a library of related functions—a bunch of functions that you might use across a range of domains.

```
1   // Program 15_2_3
2   // Example of bottom-up program - string processing (incomplete) - Stage 3
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int pos_first(string& string_to_find, string& string_to_search) {
8      /* Returns the first position of string_to_find in string_to_search,
9      or -1 if it is not in there */
10     // Code Here
11  }
12
13  void replace_string(string& string_to_modify, int start_pos, int end_pos,
14     string& string_to_insert) {
15     /* Inserts string_to_insert into string_to_modify, replacing anything
16     in positions start_pos to just BEFORE end_pos */
17     // Code Here
18  }
19
20  bool search_and_replace(string& string_to_modify, string& old_string,
21     string& new_string) {
22     /* Replaces first occurrence of old_string with new_string and returns
23     true, or returns false if it did not find old_string */
24     int start_position = pos_first(old_string, string_to_modify);
25     if (start_position == -1) {
26         return false;
27     }
28     int end_position = start_position + old_string.size();
29     replace_string(string_to_modify, start_position, end_position, new_string);
30     return true;
31  }
32
33  void search_and_replace_all(string& string_to_modify, string& old_string,
34     string& new_string) {
35     /* Replaces every occurrence of old_string with new_string and returns
36     true, or returns false if it did not find old_string */
37     while (search_and_replace(string_to_modify, old_string, new_string)) {}
38  }
39
40  int main()
41  {
42     // Code Here
43
44  }
```

i

j

k

// BUILDING A LIBRARY

If you're going to have a library, you're going to want a **namespace**—the name and double colon that you have to put in front of certain commands—to help you separate the functions in your library from others.

You should be very familiar with the **using namespace std** that you've been putting at the beginning of every program, enabling you to use commands in the standard namespace without writing **std::** in front of everything.

Namespaces are a way of helping you distinguish one library's functions from another's. That way, if one library has a function named **get_data**, then another library can also have a function named **get_data**. They can be in different namespaces, and the 2 functions can remain distinct.

You can also create your own namespaces. To do this, you just write the keyword **namespace** and then the name you want to use for your namespace; then, in curly braces, you include all the function declarations that will be part of that namespace.

For example, in **Program 15_3**, you have a namespace named **english** (l). Inside the curly braces, you have a function that is of type **void** named **greeting** (m). This function is being defined as part of the **english** namespace.

In the **main** function, you can call the **greeting** function by writing **english::greeting()** (14). That means you are calling the function named **greeting** within the **english** namespace. When that call is made, **Hello!** is printed out.

Just like you have libraries such as **iostream** that provide useful functions, you can create your own libraries to use in a similar way. When you create your own library, you will want to put all of the new functions you create in a separate namespace to prevent any possible overlap in the naming of different functions.

When you have separate libraries and as you start wanting to write really large programs—the types that are built by teams of people or by one person working for a long time—you can begin to make use of a feature called **separate compilation**.

Suppose you want to create a library of functions. The common practice in C and C++ is to break the code for these functions into 2 separate files: a **header file** and a source file. Header files usually have a **.h** or sometimes a **.hpp** file extension, while source files have a **.cpp** designation.

The header file will contain the namespace, any variables to be defined in that library or namespace, and the function declarations—which are just the header part of the function, not including the curly braces.

The declaration is a way of saying "there will be a function with this signature" but not actually saying how that function is implemented. The actual implementation is called the definition.

Up until this point, you've been putting the definition with the declaration, as is commonly done when everything is in one file. But the header-like declaration and the body-like definition can actually be split up. If you want to try this, you'll probably need to use an IDE like Visual Studio or Xcode rather than an online browser-based compiler. (See page 172 for instructions on how to create separate source and header files in your IDE.)

```
1   // Program 15_3
2   // Namespace example
3   #include <iostream>
4   using namespace std;
5
6   namespace english {
7      void greeting() {
8          cout << "Hello!" << endl;
9      }
10  }
11
12  int main()
13  {
14      english::greeting();
15  }
```

l

m

When designing your program, using pseudocode to lay out your functions, look at all of them and search for places where you might want to reuse the code. Or a file might seem to have too many ideas in one place. Both are good times to consider breaking functions into separate files.

There are many reasons why separate compilation is useful, particularly for larger programming tasks.

- It makes it easier to reuse files in other programs; you write the code once and then just #include it in other programs.
- Dividing functions across files helps ensure that no single file is too large and that everything in one file is closely related.
- By separately compiling files, only the files that change are ever recompiled. On large projects, that can greatly improve productivity.

In fact, you could separately compile every function you write. But for beginners, this makes code a little harder to follow, and you wouldn't be able to execute such code from the browser-based development environment. The rule of thumb is to consider separate compilation when you might reuse code or to simplify your files. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, section 8.7 (for namespaces).

b Lippman, Lajoie, and Moo, *C++ Primer*, section 18.2 (for namespaces).

c Ousterhout, *A Philosophy of Software Design*, chap. 4 (describing function design) and chap. 5 (describing information hiding).

d See references from lecture 11:
Problem Solving and Program Design in C (8th ed.) by Jeri R. Hanly and Elliot B. Koffman and *Problem Solving, Abstraction, and Design Using C++* (6th ed.) by Frank L. Friedman and Elliot B. Koffman.

HOW TO CREATE SEPARATE SOURCE AND HEADER FILES IN YOUR IDE

In Visual Studio, the separate files you create are not automatically put in your project, so you will have to add files to the project when you're ready to compile them. To add your separate files to the project:

- Save the files.
- Go to the project window and select "add" and "existing item".
- Add all 3 files to the project:
  - the source file that contains **`main`**
  - the header file that contains the function **`header`**
  - the source file that contains the function **`definition`**
- Then, you can build the project: **`run`** it.

Xcode automates putting the files in the project.

- Start with the **`main.cpp`** file, with **`main`** automatically appearing.
- Go to the File menu and create a Header File.
  - Save it as **`.h`**.
  - Then save it as a C++ file, **`.cpp`**.
  - If you create the **`.cpp`** source file first, Xcode will give you the option of also creating the header file automatically.
- Build and Run the program.

// QUIZ

1 Match the idea to its name:

a	top-down	1	a way of building a program by combining existing, simpler parts
b	bottom-up	2	a way of writing a minimal function so that it can be called when testing some other code
c	separate compilation	3	a way of setting up code so that a particular function can be tested
d	stub	4	a way of grouping functions and variables with a common purpose
e	namespace	5	a way of dividing a program into multiple files
f	scaffold	6	a way of dividing a problem into simpler subproblems

2 Imagine you have the following program and want to break it up into several files, which will be compiled separately. Specifically, you would like to have: a **library** header file, a **library** implementation file, and a **main** file. Which parts of the code would end up in each of those files?

```
1   #include <iostream>
2   using namespace std;
3
4   bool update_balance(float& balance, float payment,
    float interest) {
5       balance -= payment;
6       if (balance <= 0) {
7           return true;
8       }
9       balance *= (1.0+interest/100.0);
10      return false;
11  }
12
13  int main()
14  {
15      float account_balance=1000.0;
16      float payment = 50.0;
17      float interest_rate = 3.5;
18      int numpayments = 1;
19      while(!update_balance(account_
    balance,payment,interest_rate)) {
20          numpayments++;
21          payment += 10.0;
22      }
23      cout << "It takes " << numpayments << "
    increasing payments to pay the balance in the
    account." << endl;
24  }
```

Click here to see the answers.

// QUIZ ANSWERS

1

a 6

b 1

c 5

d 2

e 4

f 3

2 Separate compilation means that functions can be placed in separate files that are compiled separately. In this case, that means that there will be the following:

» a header file, containing just the function header:

```
bool update_balance(float& balance, float payment, float interest)
```

» an implementation file, containing the function definition:

```
bool update_balance(float& balance, float payment, float interest) {
  balance -= payment;
  if (balance <= 0) {
    return true;
  }
  balance *= (1.0+interest/100.0);
  return false;
}
```

» a **main** file that would include the **main** program (it must also #include the library header file):

```
int main()
{
  float account_balance=1000.0;
  float payment = 50.0;
  float interest_rate = 3.5;
  int numpayments = 1;
  while(!update_balance(account_balance,payment,interest_rate)) {
    numpayments++;
    payment += 10.0;
  }
  cout << "It takes " << numpayments << " increasing payments to pay the balance in the account." << endl;
}
```

Click here to go back to the quiz.

16 Objects and Classes: Encapsulation in C++

IN THIS LECTURE:

Up until this point, the basic style of programming you've been learning is one you could have been doing in C, even though you've seen several C++-specific features that make things better. Called procedural programming, this style relies on functions, or procedures, to organize the computation. From now on, you'll still need everything you used for procedural programming, but now you're ready to learn one of the key advances in C++: object-oriented programming.

// OBJECT-ORIENTED PROGRAMMING

When you use **object-oriented programming**, your development is centered around the creation of what are known as classes, which present an overall way of organizing the computations in a program, where objects are just particular instances of classes.

The feature of classes that is perhaps the most useful part of object-oriented programming is **encapsulation**.

Imagine that you want to write a program that would allow you to track what happens with a vending machine—money taken in, the number of items left inside, when it's time to reorder, etc.

There were "pure" object-oriented languages before C++, and there have been languages developed after C++, such as Java, that only allow object-oriented programming. By contrast, C++ lets you use both a procedural and an object-oriented style. And compared to other languages that offer some of both, such as Python, C++ has a very complete set of object-oriented features. It enables you to program in a strict object-oriented style if you want to.

The focus in object-oriented programming is to identify objects. In the case of vending machines, any particular vending machine is an object. But you would also like your program to handle vending machines more generally. So, you could also have a vending machine **class**—a way of referring to all vending machines.

The concept of classes is important enough that when C++ was first being developed, it was originally called C with Classes.

An **object** is just one particular instance of a more general class. For a given class, you can have many actual objects.

When writing code, you can think of each class as referring to a new type—a type that's defined by the programmer.

So, when you define the class **vending_machine**, you are defining a new type of variable.

Then, you can use that new type to define any specific machine. Defining each specific machine looks like how you defined a variable; that is, defining an object of a class is just like defining a variable of a type. You'll define a class, such as **vending_machine** (a), and then you can declare variables of that type, writing things like **vending_machine lobby_soda_machine** (12).

```
1  // Program 16_1
2  // Very basic class example
3  #include <iostream>
4  using namespace std;
5
6  class vending_machine {
7     // More stuff here
8  };
9
10 int main()
11 {
12    vending_machine lobby_soda_machine;
13 }
```

a

Although each vending machine object gets declared in a way that makes it look like a single variable, in reality, each vending machine contains more variables. For example, there are variables for money, including

- the amount of money someone has put into the machine toward buying something, or the credit that has been accumulated;
- the current total amount of money collected by the machine;
- the prices for all of the items; and
- the inventory level of each item.

In addition to the data that you want to track for the vending machine, you also have actions—also known as operations—that you want a vending machine to be able to perform. These actions include

- taking in money,
- giving the customer an item,
- returning change, and
- reporting inventory levels so that it can be refilled the proper amount.

You keep track of the data with variables, and you handle the actions with functions.

Using object-oriented programming, you can put all of these variables and functions together into one package.

You can collect all the stuff related to a vending machine—all its data variables and all the functions that it needs—into one vending machine class. And you don't need to let someone outside the vending machine know everything going on inside; the inventory data, for example, can be hidden from the outside. That's because all that data and all those functions are wrapped up together in one package—a process referred to as encapsulation.

Three classes you've already been using are **vector**, **string**, and **fstream**. These were predefined classes that you accessed from the **vector** library, the **string** library, or the **fstream** library using #include.

// CREATING CLASSES

You declare a class in a similar way to how you declare a function. It'll typically be defined outside of the **main** function, closer to the top of the program.

You start with the keyword **class** and then give the name of the class—in this case, **vending_machine**. The name of the class will be the name for the new type you create.

Next, you have curly braces, which group the definition of the class all together. At the end of the curly braces will be a semicolon. Note that this is one of the few times you need to add a semicolon after curly braces.

Once you have a class, you can declare objects as instances of the class, just like you would any other variable.

In **Program 16_1** on page 176, you've declared a class named **vending_machine** by writing **class vending_machine {};**. Obviously, there's nothing in the class at this time; so far, it's useless. But this gives you the ability to declare objects of type **vending_machine**.

Remember that a class is a way of defining a new type, so classes are defined outside the **main**, or any other, function. Objects can then be declared to be of that new type. Variables are typically declared inside of functions, including **main**.

Now you can declare 2 vending machines, called **lobby_machine** and **break_room_machine**. All you do to declare them as variables is write the following:

```
vending_machine lobby_machine;
vending_machine break_room_machine;.
```

These commands will create 2 **vending_machine** objects.

Remember that classes need semicolons after the closing curly brace. Fortunately, in most cases, if you forget, the compiler will catch a missing semicolon and remind you to put it in there.

// SORTING DATA IN CLASSES

A class is going to contain, inside of itself, a set of variables and functions called **member variables** and **member functions**. The class is a way of grouping all of those variables and member functions together—encapsulating them. So, when you create a new object of that class type, you're also creating a bunch of subordinate member variables defined by that class but with values specific to each object. Basically, those sub-variables "belong" to the object as its member variables.

For vending machines, one of the things you wanted to store is the price of the item that the machine sells. So, inside of the class definition, you will create a floating-point member variable named **price**. Before you declare the member variable, you write the line **public:**. This just means that the member variable can be seen outside the class. You just declare this in a similar way as you would declare any other variable: the type, the name of the member variable, and then a semicolon. In this case, that's **float price;**.

This means that a vending machine contains a member variable, sometimes called an attribute, named **price**. Every object of type **vending_machine** will have this sub-variable inside of it.

To access this member variable—the attribute sub-variable that's included inside of an instance of the **vending_machine** class—you just use the object name, followed by a period, followed by the variable: **object.member_variable**.

So, if you've declared an instance of a vending machine, such as the variable **lobby_machine**, this is a vending machine object. To access the price member variable inside of the **lobby_machine**, you write **lobby_machine.price**.

At the beginning of **Program 16_3_a**, you have your class definition. This declares a **vending_machine** type that's available to the entire program. In this case, the class contains just one public member variable: a float variable named **price** (b).

When you execute the first line of the code in **main**, it'll create a new object named **lobby_machine** (13). That will be an instance of the class **vending_machine**, and that instance will contain its own member variable inside, named **price**.

```
1   // Program 16_3_a
2   // Vending Machine class example 1 - one member variable
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   public:
8      float price;
9   };
10
11  int main()
12  {
13     vending_machine lobby_machine;
14     vending_machine break_room_machine;
15
16     lobby_machine.price = 1.00;
17     break_room_machine.price = 2.25;
18
19     cout << "The prices are: " << lobby_machine.price << " and "
20          << break_room_machine.price << endl;
21  }
```

b

The next line will create another new object, named **break_room_machine** (14). Again, it'll be an instance of the class **vending_machine**, meaning that it will have its own variable inside named **price**.

Then, you will assign a price of $1 to the **lobby_machine**. You write **lobby_machine.price**, which means that you are referring to the variable **price** within the object **lobby_machine**. You assign that the value **1.00**, so the memory will store **1.00** in that variable in that object (16).

Likewise, when you assign a price of **2.25** to the **break_room_machine**, that will assign the value to the **price** variable in *that* object. Notice that the price is not shared between the 2 instances of **vending_machine**. Each instance—each object—gets its own **price** variable (17).

When you print out **lobby_machine.price** and **break_room_machine.price**, you are printing out the values from 2 different variables.

You can define more than one member variable within a class. So, let's create new member variables for **credit** and **money_collected** and an integer variable for **inventory** (c).

You can then give any object access to each of the member variables by writing **.price**, **.inventory**, **.credit**, and **.money_collected** (d). This works just the way that accessing the price did; you're accessing a variable that's an attribute, or member, of the specific object.

This idea of packaging data together in one large container goes back to the days of C, when the container was called a **struct** (short for *structure*). Structs are still allowed in C++, but they're now almost the same as classes; the use of struct and class are pretty much interchangeable.

What really distinguishes the C++ class from the old-style struct is the ability to have member functions.

Member functions, or methods, get declared just like member variables in the class definition. Then, to call the function, you give the name of the object, then a period, and then the function name, along with the parentheses and any parameters: **some_object.function_name(...)**.

For example, say you wanted to make your vending machine friendlier, so you defined a function called **print_hello** that will just output the word **Hello**. You define the function within the class definition, so it's a member function (e).

Then, in the **main** program, you have an instance of that class. In this case, the class is **vending_machine** and the instance is **lobby_machine** (20). So, you can write **lobby_machine.print_hello()** (22).

When this is run, **Hello** is indeed printed to the screen.

```
1   // Program 16_3_b
2   // Vending Machine class example 2 - several member variables
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   public:
8      float price;
9      float credit;
10     float money_collected;
11     int inventory;
12  };
13
14  int main()
15  {
16     vending_machine lobby_machine;
17     vending_machine break_room_machine;
18
19     lobby_machine.price = 1.00;
20     lobby_machine.inventory = 200;
21     lobby_machine.credit = 0.0;
22     lobby_machine.money_collected = 0.0;
23  }
```

c

d

```
1   // Program 16_4
2   // Vending Machine class example 3 - member function
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   public:
8      float price;
9      float credit;
10     float money_collected;
11     int inventory;
12
13     void print_hello() {
14         cout << "Hello" << endl;
15     }
16  };
17
18  int main()
19  {
20     vending_machine lobby_machine;
21
22     lobby_machine.print_hello();
23  }
```

e

Now suppose you want the machine to be able to tell workers how many items are remaining inside so that they know if it's time to restock.

Let's create a function named **number_remaining** that will return the number of items remaining.

This function does not need any parameters; you can just return an integer, the number of items in the machine. So, the function will be declared as **int number_remaining()**, with no parameters needed inside **(f)**.

```
1    // Program 16_5
2    // Vending Machine class example
     4 - alternative member function
3    #include <iostream>
4    using namespace std;
5
6    class vending_machine {
7    public:
8       float price;
9       float credit;
10      float money_collected;
11      int inventory;
12
13      int number_remaining() {
14          return inventory;
15      }
16   };
17
18   int main()
19   {
20      vending_machine lobby_
     machine;
21      lobby_machine.inventory = 250;
22
23      cout << lobby_machine.number_
     remaining() << " items remain in
     the machine." << endl;
24   }
```

f

The body of this function is really simple: **return inventory** **(14)**. Notice that because the function and the variable are part of the same class, the function can refer to the variable simply as **inventory**. When the function refers to **inventory**, it will use the value in the inventory variable for whichever object it is a part of.

In this case, when you set the inventory level to **250** in the **lobby_machine** **(21)**, then calling **lobby_machine**'s member function **number remaining** will return the value in that lobby machine: **250**.

Now let's look to **Program 16_6** to see how your vending machine will handle money. First, you need to be able to take in money. When someone puts money into the machine, that person should have a credit for that amount of money. Let's call that member function **deposit_money**.

The **deposit_money** function will again be very simple. It doesn't need to return anything; it's just a deposit. So, you declare it as **void deposit_money**.

The function will take in a floating-point number, the amount that's being deposited. So, you have **(float amount)** **(17)**.

Whatever money is deposited will go toward a credit in the machine. So, the body of the function will just increase **credit** by the amount of the deposit: **credit += amount**. **(18)**

```
1    // Program 16_6
2    // Vending Machine class example 5 -
     multiple member functions
3    #include <iostream>
4    using namespace std;
5
6    class vending_machine {
7    public:
8       float price;
9       float credit;
10      float money_collected;
11      int inventory;
12
13      int number_remaining() {
14          return inventory;
15      }
16
17      void deposit_money(float amount) {
18          credit += amount;
19          cout << "Current credit is " <<
     credit << endl;
20      }
21   };
22
23   int main()
24   {
25      vending_machine lobby_machine;
26      lobby_machine.credit = 0.0;
27
28      lobby_machine.deposit_money(0.25);
29      lobby_machine.deposit_money(0.25);
30      lobby_machine.deposit_money(0.10);
31   }
```

g

h

You'll also add an output statement to note what the current credit is after the deposit is made **(19)**.

Then, in your **main** program, you can create a **vending_machine** object and set the credit in the object to **0** **(g)**. Then, you'll deposit **0.25**, **0.25**, and **0.10** **(h)**. The credit at this point is **0.6**, as expected.

To initialize the member variables, you just write an assignment operation at the time of declaration, just like with any other variable declaration. In this code, you set both the **credit** and **money_collected** to **0.0**, a default value of **1.0** for the price, and an inventory level of **0**.

```
8       float price = 1.0;
9       float credit = 0.0;
10      float money_collected = 0.0;
11      int inventory = 0;
```

To return the change from the machine, you could write a **return_change** function. It would take whatever credit is currently in the machine, return it, and set the current credit to **0**. You output the amount of money that you're returning.

Exercise

Create a function for pulling together the specific steps a vending machine goes through when the machine is asked to vend an item.

Note that for an item to be vended, there needs to be an item in the machine and enough credit to purchase it. If there is, then you'll distribute that item, deduct the price from the credit, and return any change. Make this function return a Boolean: **true** if an item is vended and **false** otherwise.

Use the following pseudocode to write the code.

```
1   define a function in the class named vend that returns a Boolean
    a   check to see if there is enough credit to purchase an item
    b   check to see if you actually have any items in inventory
    c   handle the transaction, adjusting credit, money collected, and inventory
    d   return any change to the customer
```

Click here to see the solution.

// PUBLIC VERSUS PRIVATE

So far, you've always had **public:** at the top of your class definition—meaning that everything you declared after that in the class is a **public variable or function**, available for anyone to call from anywhere.

But you can also have **private variables and functions**, which you get by writing **private:** beforehand.

When a variable or function is private, it is accessible only to other parts within that class; that is, the functions within that same class can access the private variables and call private functions, but those outside the class cannot.

```
1     // Program 16_9_ERROR
2     // Vending Machine class example 8 - public and private members
3     #include <iostream>
4     using namespace std;
5
6     class vending_machine {
7     private:
8          float price = 1.0;
9          float credit = 0.0;
10         float money_collected = 0.0;
11         int inventory = 0;
12
13    public:
14         int number_remaining() {
15              return inventory;
16         }
17
18         void deposit_money(float amount) {
19              credit += amount;
20              cout << "Current credit is " << credit << endl;
21         }
22
23         float return_change() {
24              float amt_to_return;
25              amt_to_return = credit;
26              credit = 0;
27              cout << "Returning " << amt_to_return << " in change." << endl;
28              return amt_to_return;
29         }
30
31         bool vend() {
32              if (credit < price) {
33                   cout << "Please deposit more money" << endl;
34                   return false;
35              }
36              else if (inventory <= 0) {
37                   cout << "Sold Out." << endl;
38                   return false;
39              }
40              else {
41                   credit -= price;
42                   money_collected += price;
43                   cout << "Vending an item" << endl;
44                   inventory--;
45                   return_change();
46                   return true;
47              }
48         }
49    };
50
51    int main()
52    {
53         vending_machine lobby_machine;
54         lobby_machine.inventory = 200;
55
56         lobby_machine.deposit_money(0.25);
57         lobby_machine.deposit_money(0.25);
58         lobby_machine.deposit_money(0.60);
59
60         if (lobby_machine.vend()) {
61              cout << "We got an item!" << endl;
62         }
63         else {
64              cout << "No item for us." << endl;
65         }
66    }
```

Going back to the vending machine example, the member variables are now marked as private, with **`private:`** in front of their declarations. The member functions are still public, with **`public:`** in front of them.

When you try to run this code, you get an error, coming from the line in the **`main`** program where you try to access **`inventory`** to assign some initial inventory to the **`lobby_machine`** (54). This is not allowed; because **`inventory`** is private, you cannot access it from outside the class.

To fix this, you can move the **`public:`** line up above the **`inventory`** member variable. That will make **`inventory`** public, meaning that it's OK to access it.

If you run the code now, everything is fine. You are able to access inventory from outside the class, so the line that gave you trouble is no longer an issue.

```
7     private:
8          float price = 1.0;
9          float credit = 0.0;
10         float money_collected = 0.0;
11         int inventory = 0;
12
13    public:
14         int number_remaining() {
```

```
7     private:
8          float price = 1.0;
9          float credit = 0.0;
10         float money_collected = 0.0;
11
12    public:
13         int inventory = 0;
14         int number_remaining() {
```

Notice that you did not get a compiler error in any of the many places inside the member functions where you accessed member variables. Because you were accessing the private members from within the class, everything was OK. Only when you tried to access those members from the **`main`** function was there an issue.

The **`public`** and **`private`** designations can be used multiple times in a class definition. However, it's common practice to put the private members first and the public members after that. It's also common practice to put all the member variables (attributes) first and the member functions (methods) after that.

Besides **`public`** and **`private`** designations, there's also **`protected`**, but for the cases you've seen, **`protected`** acts just like **`private`**.

In a class definition, the default assumption is that the members are being defined as private. That's why you previously had to have **`public:`** in your code; if you had left it off, everything would have been seen as private.

It's not uncommon for all of the member variables to be private, or protected. That makes sure that nothing will mess around with the internal structure of the class unless it does so through an approved channel—one of the public functions.

But it's still common to need to adjust those variables or find out what they are.

Accessor functions let you read the value of a private member variable. Basically, they provide one of those approved channels through which information is allowed to leave the class.

This is actually the main place where there's a difference between the C++ class and the C++ struct: In a C++ class, the default is that everything is private, but in a struct, the default is that everything is public.

Mutator functions let you modify the value of a private member variable.

When you're debugging, accessor and mutator functions give you a line of code that you can set a breakpoint at so that you know when a member is being accessed. Conversely, you have no way of automatically pausing when a public member is being changed.

So, accessors and mutators help you ensure that the overall object maintains internal consistency whenever it is used. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 9.1–9.4.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 7.1 and 7.2.

Exercise Solution

The function **vend** (lines 30–47) defines the vending operation.

```
1   // Program 16_8
2   // Vending Machine class example 7 - vending an item
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   public:
8       float price = 1.0;
9       float credit = 0.0;
10      float money_collected = 0.0;
11      int inventory = 0;
12
13      int number_remaining() {
14          return inventory;
15      }
16
17      void deposit_money(float amount) {
18          credit += amount;
19          cout << "Current credit is " << credit << endl;
20      }
21
22      float return_change() {
23          float amt_to_return;
24          amt_to_return = credit;
25          credit = 0;
26          cout << "Returning " << amt_to_return << " in change." << endl;
27          return amt_to_return;
28      }
29
30      bool vend() {
31          if (credit < price) {
32              cout << "Please deposit more money" << endl;
33              return false;
34          }
35          else if (inventory <= 0) {
36              cout << "Sold Out." << endl;
37              return false;
38          }
39          else {
40              credit -= price;
41              money_collected += price;
42              cout << "Vending an item" << endl;
43              inventory--;
44              return_change();
45              return true;
46          }
47      }
48  };
49
50  int main()
51  {
52      vending_machine lobby_machine;
53      lobby_machine.inventory = 200;
54
55      lobby_machine.deposit_money(0.25);
56      lobby_machine.deposit_money(0.25);
57      lobby_machine.deposit_money(0.60);
58
59      if (lobby_machine.vend()) {
60          cout << "We got an item!" << endl;
61      }
62      else {
63          cout << "No item for us." << endl;
64      }
65  }
```

// QUIZ

1 For each of the following, determine whether the statement is true or false:

a If you do not specify whether something is public or private in a class, it is public.

b You can call a private member function from within the same class.

c You can access a public member variable from outside the class.

d A class's member variables must be either all public or all private.

e All other things being equal, it's better for member variables to be private.

2 Create a class named **message** with a single member function named **display** that, when called, will display **Hello, World!**.

3 The distance between a point (x_1, y_1) and a point (x_2, y_2) is given by the formula distance = $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Assume there is a class defined as

```
class point {
  public:
  float x;
  float y;
};
```

and that the **cmath** library is already included. Write a function that takes in 2 points and returns the distance between them.

4 What would be the output of the following code?

```
#include <iostream>
using namespace std;

class pointcounter {
    private:
    int points=0;
    int bonus=500;

    public:
    void increasepoints(int p) {
        points += p;
    }
    void givebonus() {
        points += bonus;
    }
    int number() {
        return points;
    }
};

int main()
{
    pointcounter gamescore;
    cout << "Score is now: " << gamescore.number()
<< endl;
    gamescore.increasepoints(10);
    cout << "Score is now: " << gamescore.number()
<< endl;
    gamescore.givebonus();
    cout << "Score is now: " << gamescore.number()
<< endl;
}
```

Click here to see the answers.

// QUIZ ANSWERS

1 a **False**. If you do not specify one way or another, members default to private.

b **True**. Private member variables and functions are available to anything else within the same class but nothing outside that class.

c **True**. A public member variable or function can be accessed from anything else outside the class.

d **False**. You can have some member variables public and some private.

e **True**. If member variables are private, then only the functions within the class can modify them, so the class can ensure the variables always have valid values.

2 You should declare a class by writing **class message** and then curly braces. Within the curly braces, you want to declare the member function as being **public** so that it can be called. The member function itself will be straightforward: **void** return, no parameters, and just a single output line in the body.

```
class message {
  public:
  void display() {
    cout << "Hello, World!" << endl;
  }
};
```

3 A function might be:

```
float distance(point a, point b) {
  return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}
```

Notice that there are 2 parameters, each of type **point**, named **a** and **b**. In the function, you can access the member variables of the 2 parameters by using **a.x**, **a.y**, **b.x**, and **b.y**. Note that there are other ways to write the function, too.

4 The output is:

```
Score is now: 0
Score is now: 10
Score is now: 510
```

Notice that the private variable **points** is initialized to **0**. The **number** member function will return the value of **points**, so the first output line will state that the points are initially **0**. The **increasepoints** member function increases the points by the argument value, which is **10** in this case. Thus, the next line of output is **10**. Finally, the **givebonus** member function will increase the points by **500**, increasing **points** by **500** more, so the final output line will output the value **510**.

Click here to go back to the quiz.

17 Object-Oriented Constructors and Operators

The ability to design appropriate classes may be the single most important skill in object-oriented programming. Classes allow you to encapsulate data relationships in ways that match up very nicely with many real-world problems. But how well your program works—indeed, whether it works at all—can depend crucially on how your classes, and their operations, have been set up at the start of the program. Two key tools for getting classes off to a strong start for any object-oriented program are constructors and operator overloading.

IN THIS LECTURE:

// CONSTRUCTORS

Whenever you declare a new class or object, the computer sets aside a chunk of memory. This memory will hold all of the member variables for that object. So, you want to have some way of initializing the memory—that is, the member variables—and that's where a **constructor** comes in. A constructor is a special function run at the time a new object is created. Its purpose is to initialize the object to some valid state so that the member variables all have reasonable values.

Often, the constructor you'll want to define is the **default constructor**. A default constructor that you create should be defined as a public member function of the class. Make sure to write `public:` before defining the constructor.

The constructor is a special function; there is no return type, because the only thing it is doing is initializing an object.

The word default can have 2 meanings here:

- The default constructor is what's called when you declare an object in the common way, where you just list the type and then the variable name.
- But if a user doesn't specify any constructor at all, then the compiler will automatically create its own default constructor for the class.

To ensure that your objects begin in valid states, you should always provide constructors for all of your classes.

The name of the constructor function should just be the class name. And the default constructor should take no parameters. So, you'll define the default constructor by writing the class name and then parentheses; then, inside curly braces, you'll write whatever you want to happen each time you create a new instance of that class.

In **Program 17_1**, a default constructor is added after the **public** designation so that it's publicly available. It uses the class name **vending_machine** as the name for the constructor function, followed by parentheses, in this case with nothing inside, and then an opening curly brace (14).

Inside the curly braces, you have all the default values you want to set (a). The **price**, **credit**, and **money_collected** levels are all set to **0.0**. The **inventory** level is set to a default level of **100**. Also, you'll output a message that you've created a new vending machine whenever this constructor is run (19).

In the **main** program, you declare a new vending machine object named **lobby_machine** (30). When you declare the new object, there is memory set aside for that object—basically enough memory to contain all the member variables for a **vending_machine**. And notice that when the object is declared, the constructor you defined is executed, so the default constructor will set the **price**, **credit**, **money_collected**, and **inventory** for that machine. And you'll get the output: **Created a new vending machine**.

If you also access the **inventory** using the **number_remaining** accessor function (b), you see that indeed there are 100 items in the machine. The constructor was what set that value to **100**.

Using a constructor is the preferred way to initialize member variables.

- A constructor is more explicit about what's being done at the time of declaration.
- A constructor is far more flexible. It can print out messages and perform more complex computations, including making function calls, forming more complex variables, and so on.

The constructor format is very much like a function, but without a return type. You can even take parameters in for a non-default constructor, just like you take in parameters for a regular function. Then, when declaring a new object, you include parentheses and arguments.

```
1   // Program 17_1
2   // Vending Machine class example
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   private:
8      float price;
9      float credit;
10     float money_collected;
11     int inventory;
12
13  public:
14     vending_machine() {
15        price = 0.0;
16        credit = 0.0;
17        money_collected = 0.0;
18        inventory = 100;
19        cout << "Created a new
    vending machine." << endl;
20     }
21
22     int number_remaining() {
23        return inventory;
24     }
25
26  };
27
28  int main()
29  {
30     vending_machine lobby_machine;
31
32     cout << "There are " <<
    lobby_machine.number_remaining()
    << " items." << endl;
33  }
```

a

b

```
1   // Program 17_3
2   // Vending Machine non-default constructor
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   private:
8      float price;
9      float credit;
10     float money_collected;
11     int inventory;
12
13  public:
14     vending_machine() {
15        price = 1.0;
16        credit = 0.0;
17        money_collected = 0.0;
18        inventory = 100;
19        cout << "Created a new vending machine." << endl;
20     }
21
22     vending_machine(int starting_inventory) {
23        price = 1.0;
24        credit = 0.0;
25        money_collected = 0.0;
26        inventory = starting_inventory;
27        cout << "Created a new vending machine." << endl;
28     }
29
30     int number_remaining() {
31        return inventory;
32     }
33
34  };
35
36  int main()
37  {
38     vending_machine lobby_machine(50);
39
40     cout << "There are " << lobby_machine.number_
    remaining() << " items." << endl;
41  }
```

Exercise

Write a constructor that takes in both an item price and an initial level of inventory and then call that to create a **vending_machine** that has 75 items, costing $5 each.

Click here to see the solution.

Let's see how you could define a new constructor for your **vending_machine**.

Suppose you want to be able to specify a specific starting inventory level, rather than a default level, at the time the object is allocated. In that case, you'd want a constructor that takes in one integer parameter. Remember, this is no longer the default constructor—the one with no parameters—but rather a new constructor.

Again, the name is just the class name, but now it takes in a single integer, **starting_inventory** (22). And the function itself sets up all the other values to their defaults, but sets the inventory to that parameter value that should be passed in (26).

When declaring a new **vending_machine**, you give the variable name, which in this case is again **lobby_machine**, but also have parentheses in which you have a single integer argument (38). Then, when you run this program, the new constructor—not the default constructor—will be used to initialize the object.

So, in this case, because you passed in **50** as an argument when you declared the **vending_machine**, you have an initial inventory level of 50.

You can pass in arguments that are variables just as easily as other values. In this case, you read in from the user a number of initial items that the machine should be stocked with. Then, you can declare a vending machine with that number.

Notice that this sort of initialization could not occur if variables were just set to default values by assignment statements in the class declaration.

```
36  int main()
37  {
38      cout << "How many items does the machine start wtih? ";
39      int init_number;
40      cin >> init_number;
41      vending_machine lobby_machine(init_number);
42
43      cout << "There are " << lobby_machine.number_remaining() << " items." << endl;
44  }
```

Be aware that you should not try to declare an object using the default constructor by just using parentheses with no arguments. If you're using the default constructor, just leave off the parentheses entirely. Only use parentheses when you want to use a non-default constructor or when you're declaring the constructor within the class.

// OPERATOR OVERLOADING

You've already encountered function overloading, where you can use the same function name but take different parameters to get different behavior. **Operator overloading** is similar, but you take different operands to get different behavior from an **operator**.

You've already seen examples of operator overloading. You've seen how **+** can be used to mean addition for number types or concatenation for string types. The operator has been "overloaded" to take on more than just one meaning; it has different meanings depending on the implementation.

But that's not the only place you've seen operator overloading. The streaming operators (**<<** and **>>**) are used for streaming input and output for the console, a file, or a stringstream. These operators have been defined for multiple data types.

When you have the stream operator to output a floating-point number, it behaves one way: It analyzes the number and displays the digits of the number with the decimal point shown in the correct position. When you use the same streaming operator to output a string, it behaves a different way: It analyzes the array of characters kept in the string and outputs each character in the string to the stream.

If you did not have operator overloading, you would have to have different operators for outputting integers, outputting floats, outputting strings, and so on.

Using stream operators to stream from input shows the same differences; the input is treated in a way that is specific to the type you are wanting to read in. So, you can use the same operator—the stream operator—but it is specialized to the type you need.

In fact, the stream operator can also be used with an integer on the left, where it has a totally different operation from when there is a stream on the left. With integers, the stream operator actually means "shift the bits of the integer left or right a certain number of spaces." While those shifts can be useful in making really efficient code, that operation has nothing to do with streaming input or output!

There is a wide variety of operators you can overload. Binary operators take 2 elements per operator. Examples include arithmetic operators, stream operators, comparison operators, and Boolean operators.

```
+ - * / % <<
>> > < >- <=
 == != && ||
 & | ^ [] ->
```

There are also a few unary operators, which take just a single element.

```
 - + ++
-- ! * &
```

All of these operators, and even a few less common ones, are those you can overload; you can create your own version of how they should operate for a particular class or combination of classes.

Operator overloading is more than just a clever way to put familiar characters to a new use; it can be a fundamental part of making a class more widely useful.

```
1  // Program 17_6
2  // Example of shifting bits
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int i = 1;
8     int j = i << 3; // Shifting bits 3
   spaces left
9     cout << j << endl;
10 }
```

// OVERLOADING BINARY OPERATORS

One way of overloading a binary operator involves defining an operator in a very similar way to how you define a function.

Whenever a binary operator is used, there are going to be 2 operands: one just before the operator and one immediately after. Then, there will be a result of some type. For example, a comparison operator should probably give a Boolean result.

The type of result is basically the return type of the operator **function**, which will take exactly 2 parameters: the object type of the first and second operands. The only "different" part is the function name, which will have the keyword **operator**, followed by the particular operation. And all of this gets defined just like any other function would—not as a part of any class.

```
1    // Program 17_8
2    // Operator example - multiplying lightbulb
3    #include<iostream>
4    using namespace std;
5
6    class lightbulb {
7    public:
8       int watts_used;
9       int lumens;
10      int temperature;
11
12      lightbulb() {
13          watts_used = 60;
14          lumens = 900;
15          temperature = 2700;
16      }
17   };
18
19   lightbulb operator *(lightbulb bulb, int increase_
     factor) {
20      lightbulb ans;
21      ans.watts_used = bulb.watts_used * increase_factor;
22      ans.lumens = bulb.lumens * increase_factor;
23      ans.temperature = bulb.temperature;
24      return ans;
25   }
26
27   int main() {
28      lightbulb bulba, bulbb;
29      bulbb = bulba * 2;
30      cout << bulbb.watts_used << " " << bulbb.lumens << "
     " << bulbb.temperature << endl;
31   }
```

c

d

Note that operators are not automatically commutative; in other words, you cannot switch around the order of the 2 operands. So, you could not multiply an integer times a lightbulb; you defined a lightbulb times an integer, but not the other way around.

Suppose you have a class you are using to keep track of lightbulbs. Every bulb will have 3 pieces of data: the number of watts used, the number of lumens (amount of light) produced, and the equivalent color temperature of the bulb.

You can define a class with those member variables, as you can see in the code. Notice that all 3 member variables are public in this case and that a default constructor is being used to set initial values for the 3 values (c).

Let's say that you want the ability to create a new bulb that is some factor multiple of an existing bulb. For example, if a bulb is 2 times another bulb, it should use 2 times as many watts and produce 2 times as many lumens but have the same color temperature.

Let's see how you could define a multiplication operator for the lightbulb. Your multiplication operator should take in 2 operands: a lightbulb and an integer that you are multiplying the lightbulb by. And the result should be a new lightbulb, one using that factor more watts and producing that factor more lumens. So, your operator definition should produce another lightbulb as a result, so the definition has a return type of **lightbulb**. You then would write **operator *** to show that you are defining the multiplication operator. Then, the parameters for the operator should be a lightbulb (labeled **bulb** in this case) and an integer (labeled **increase_factor**) (19).

Inside of the function, notice that you create a new lightbulb, called **ans,** short for *answer*, because it will be the answer you return from the operator. You set the watts used and the lumens as a product of those elements of **bulb** multiplied by **increase_factor**. The color temperature is just the same as **bulb** is. Then, you return **ans** (d).

In your **main** routine, you can now create 2 bulbs named **bulba** and **bulbb**, where **bulba** is initialized with the default constructor so that it has **60** watts, **900** lumens, and a **2700** color temperature. You set **bulbb** to be **bulba** times **2**. This will call the multiplication operator on a lightbulb times an integer, so **bulbb** should have twice the number of watts, or **120**, and twice the number of lumens, or **1800**, while having the same color temperature, **2700**, as **bulba**. And that's indeed what you see as a result.

If you had also wanted to be able to define an integer times a lightbulb, you'd have had to define a separate operator (e). Then, you could multiply 2 times a lightbulb (37).

```
27  lightbulb operator *(int increase_factor, lightbulb bulb) {
28     lightbulb ans;
29     ans.watts_used = bulb.watts_used * increase_factor;
30     ans.lumens = bulb.lumens * increase_factor;
31     ans.temperature = bulb.temperature;
32     return ans;
33  }
34
35  int main() {
36     lightbulb bulba, bulbb;
37     bulbb = 2 * bulba;
38     cout << bulbb.watts_used << " " << bulbb.lumens << " "
    << bulbb.temperature << endl;
39  }
```

e

Also, if you had not made the member elements public, then you couldn't have written the multiplication operator. It would not have been possible to access private member variables of the object from outside the function. Instead, you'd have had a compilation error.

So, it's more common for operators to be defined within a class. That way, you can access all the elements of the class without accessors and mutators.

In **Program 17_10**, the member values have been made private, but a **print_vals** function has been added that prints all the member variables to the console, separated by spaces (f). This will let you print the values of the member variables from outside the class.

The **main** routine is set up in a very similar way as before, with the only difference being that you're calling the **print_vals** function because the member variables of the **lightbulb** class are private (g).

```
1   // Program 17_10
2   // Operator example - defining inside a class
3   #include<iostream>
4   using namespace std;
5
6   class lightbulb {
7      int watts_used;
8      int lumens;
9      int temperature;
10
11  public:
12     lightbulb() {
13        watts_used = 60;
14        lumens = 900;
15        temperature = 2700;
16     }
17
18     lightbulb operator *(int increase_factor) {
19        lightbulb ans;
20        ans.watts_used = watts_used * increase_
    factor;
21        ans.lumens = lumens * increase_factor;
22        ans.temperature = temperature;
23        return ans;
24     }
25
26     void print_vals() {
27        cout << watts_used << " " << lumens <<
    " " << temperature << endl;
28     }
29  };
30
31  int main() {
32     lightbulb bulba, bulbb;
33     bulbb = bulba * 2;
34     bulbb.print_vals();
35  }
```

f

g

Notice how the multiplication operator has been defined inside the class (18). First, you still have the return type of **lightbulb**, just like you did when the operator was defined outside the class.

You also still have the keyword **operator** and the particular operator you are dealing with, which in this case is * for multiplication. A key difference to note is that there is only one parameter: an integer. The first parameter—in this case, a **lightbulb**—is implied because this operator is being defined inside the **lightbulb** class. When defining in a class like this, you don't specify the first parameter and will get an error if you try to.

Inside of the routine, you can access the values of the member variables for the particular lightbulb you are operating with directly. You simply write **watts_used**, **lumens**, and **temperature** to get the watts used, lumens, and temperature for this particular lightbulb.

And notice that you could set the values for the member variables of the lightbulb you are returning directly. Even though they are private members, because you are writing this routine inside of the class, you have access to those member variables, even the private ones.

Also, keep in mind that the order of operations still matters. By defining multiplication inside of the class, that only helps you multiply a lightbulb by an integer. You still could not reverse that to multiply an integer times a lightbulb.

In fact, from within the **lightbulb** class, you have to start with a lightbulb times something. From within the class, you can't ever do the reverse—define something times a lightbulb! The only way to define something times a lightbulb would be to define it outside the class.

```
18    lightbulb operator *(int increase_factor) {
19        lightbulb ans;
20        ans.watts_used = watts_used * increase_factor;
21        ans.lumens = lumens * increase_factor;
22        ans.temperature = temperature;
23        return ans;
24    }
```

// OVERLOADING UNARY OPERATORS

Let's try defining a new operation for **!**, which usually means "not" in Boolean logic.

Remember that you can define operators to mean whatever you want and that the best redefinitions typically expand an existing meaning in some way. Here, let's say that **!** defines a "dead" lightbulb—a "not" lightbulb. You'll set the watts used and lumens to **0** and you'll copy the color temperature from before so that you can remember what type each was originally.

If you define the unary operator inside the class, you have an empty parameter list. You just write **lightbulb operator !()** (30).

The function definition just sets **watts_used** and **lumens** to **0** and copies the **temperature** member variable (h).

In the **main** routine, then, you can create a lightbulb named **bulb** and then print out the values from **!bulb** and see that indeed you have the modified value (45).

```
1   // Program 17_11
2   // Operator example - defining a unary operator
3   #include<iostream>
4   using namespace std;
5
6   class lightbulb {
7      int watts_used;
8      int lumens;
9      int temperature;
10
11  public:
12     lightbulb() {
13         watts_used = 60;
14         lumens = 500;
15         temperature = 2700;
16     }
17
18     lightbulb operator *(int increase_factor) {
19         lightbulb ans;
20         ans.watts_used = watts_used * increase_factor;
21         ans.lumens = lumens * increase_factor;
22         ans.temperature = temperature;
23         return ans;
24     }
25
26     bool operator <(lightbulb& a) {
27         return (watts_used < a.watts_used);
28     }
29
30     lightbulb operator !() {
31         lightbulb ans;
32         ans.watts_used = 0;
33         ans.lumens = 0;
34         ans.temperature = temperature;
35         return ans;
36     }
37
38     void print_vals() {
39         cout << watts_used << " " << lumens << " " << temperature << endl;
40     }
41  };
42
43  int main() {
44     lightbulb bulb;
45     (!bulb).print_vals();
46  }
```

h

```
1    // Program 17_11_a
2    // ++ Operator example - prefix and postfix
3    #include<iostream>
4    using namespace std;
5
6    class elephant {
7    public:
8       float height;
9       float weight;
10
11      elephant() {
12          height = 10.0;
13          weight = 13000.0;
14      }
15
16      void operator ++() {
17          // Prefix operator - for ++elephant
18          cout << "increasing height of
     elephant" << endl;
19          height += 0.1;
20      }
21
22      void operator ++(int whatever) {
23          // Postfix operator - for elephant++
24          cout << "increasing weight of
     elephant" << endl;
25          weight += 100.0;
26      }
27
28      void print_characteristics() {
29          cout << "Height is " << height << "
     and Weight is " << weight << endl;
30      }
31   };
32
33   int main() {
34      elephant Dumbo;
35      Dumbo.print_characteristics();
36      Dumbo++;
37      Dumbo.print_characteristics();
38      ++Dumbo;
39      Dumbo.print_characteristics();
40
41   }
```

In **Program 17_11_a**, you've created 2 versions of the **++** operator: one for prefix and one for postfix. You have an **elephant** class with **height** and **weight**. The prefix operator will increase the height of the elephant, while the postfix operator will increase the weight.

For the prefix operator, you just have the operator defined with no parameters (16). For the postfix operator, you have an integer parameter included—but it never gets used (22).

If you were to define the operator outside the class, then the prefix version would have one parameter—of type **elephant**, in this case—and the postfix version would have 2 parameters: the first one being an **elephant** and the second one being an integer.

There are a few rules of thumb for when it's best to define operators—inside of the class or outside.

It usually makes sense to define an operation inside the class if you're defining

1. any unary operator, or
2. a binary operator that operates on 2 of the same type, such as a comparison of 2 objects of the same type.

This is because these are operations that are a part of only that class and no other.

On the other hand, it can make sense to define outside

- if 2 different classes are involved, and
- especially if you need to provide an operation in both forms.

Still, there's no single right answer; rather, it's a judgment call. Think about whether an operator feels more like it's part of a class or external to a class.

// FRIEND FUNCTIONS

One of the main advantages of defining a function or operator inside a class is that you get access to member variables and functions.

But even if you define functions or operators outside a class—for example, because the function or operator involves 2 different classes—there's actually a mechanism to make a function or operator still get access to the member variables. And this is by declaring the function or operator a **friend**.

To do this, you include the function or operator signature in the class definition with the designator **friend**. That way, you can define the function outside the class and still have access to member variables. **Program 17_12** is an example, using lightbulb multiplication.

In the class definition, you say: "There is going to be a friend operator that will multiply a lightbulb times an integer." The way you do this is by writing **friend lightbulb operator *(lightbulb, int);**. Notice that you don't need to give the parameter names, just the types (22).

Then, outside of the class, you can define the actual operator, this time with named parameters (25). Because this is a friend function of the **lightbulb** class, it will have access to all the private members of the **lightbulb** class. So, if the lightbulb has the parameter name **bulb**, you can access elements of the lightbulb by writing **bulb.lumens**, and so on.

You can also assign values to objects of the type **lightbulb** by writing, for example, **ans.temperature**, where **ans** is a **lightbulb** object.

```
1   // Program 17_12
2   // Friend Operator example
3   #include<iostream>
4   using namespace std;
5
6   class lightbulb {
7      int watts_used;
8      int lumens;
9      int temperature;
10
11  public:
12     lightbulb() {
13        watts_used = 60;
14        lumens = 500;
15        temperature = 2700;
16     }
17
18     void print_vals() {
19        cout << watts_used << " " << lumens << " " << temperature
    << endl;
20     }
21
22     friend lightbulb operator *(lightbulb, int);
23  };
24
25  lightbulb operator *(lightbulb bulb, int increase_factor) {
26     lightbulb ans;
27     ans.watts_used = bulb.watts_used*increase_factor;
28     ans.lumens = bulb.lumens*increase_factor;
29     ans.temperature = bulb.temperature;
30     return ans;
31  }
32
33  int main() {
34     lightbulb bulba, bulbb;
35     bulbb = bulba * 2;
36     bulbb.print_vals();
37  }
```

// OVERLOADING STREAM OPERATORS

It is very common that you'd want to stream data from an object to output it to the console, a file, or a string. Likewise, it's common that you'd want to stream data into an object. It would be nice to declare a **lightbulb** object and just be able to **cout** or **cin** it. But to do this, you have to overload the stream operations.

Remember that the syntax of a stream operation is that you have the stream on the left, then the stream operator (<< for output or >> for input), and then the thing you are wanting to stream out of or into. And the stream should be able to continue afterward.

Because the stream itself comes at the left of the list, that means that you can't write a stream operation for your new class as part of that class.

So, instead, you need to have the stream operation defined outside the class (21). It's a good idea to make this stream operator a friend operator (18), because that way you'll have access to the member variables.

You can think: "If I'm outputting to a stream, I'll have an **ostream**, and if I'm inputting, I'll have an **istream**." Both are predefined stream types, where **cin** is an example of an **istream** and **cout** is an example of an **ostream**.

You'll also need to make sure that your operator returns another stream, because that way you can stream one thing after another; the result of the operation is a stream, which then can have another operation applied, and so on ◆.

```
1   // Program 17_13
2   // Overloading Output Stream Example
3   #include<iostream>
4   using namespace std;
5
6   class lightbulb {
7      int watts_used;
8      int lumens;
9      int temperature;
10
11  public:
12     lightbulb() {
13        watts_used = 60;
14        lumens = 500;
15        temperature = 2700;
16     }
17
18     friend ostream& operator <<(ostream&, const lightbulb&);
19  };
20
21  ostream& operator <<(ostream& s, const lightbulb& b) {
22     s << b.watts_used << " " << b.lumens << " " << b.temperature;
23     return s;
24  }
25
26  int main() {
27     lightbulb bulb;
28     cout << bulb << endl;
29  }
```

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 9.4, 9.6, and 9.7.

b Lippman, Lajoie, and Moo, *C++ Primer*, section 7.5, chap. 14, and section 15.7.

Here's one way to implement that.

```
1   // Program 17_5
2   // Vending Machine constructor with multiple parameters
3   #include <iostream>
4   using namespace std;
5
6   class vending_machine {
7   private:
8       float price;
9       float credit;
10      float money_collected;
11      int inventory;
12
13  public:
14      vending_machine() {
15          price = 1.0;
16          credit = 0.0;
17          money_collected = 0.0;
18          inventory = 100;
19          cout << "Created a new vending machine." << endl;
20      }
21
22      vending_machine(int starting_inventory) {
23          price = 1.0;
24          credit = 0.0;
25          money_collected = 0.0;
26          inventory = starting_inventory;
27          cout << "Created a new vending machine." << endl;
28      }
29
30      vending_machine(float initial_price, int starting_inventory) {
31          price = initial_price;
32          credit = 0.0;
33          money_collected = 0.0;
34          inventory = starting_inventory;
35          cout << "Created a new vending machine with " << inventory
36              << " items at a cost of " << price << " each." << endl;
37      }
38
39      int number_remaining() {
40          return inventory;
41      }
42
43  };
44
45  int main()
46  {
47      vending_machine lobby_machine(50.0, 75);
48  }
```

Click here to go back to the exercise.

// QUIZ

1 Assume that you have a class called **customer** with a string member variable called **name**. What would the default constructor function be if it were setting the **name** to be **unknown**?

2 What is the output of the following code?

```
#include <iostream>
using namespace std;

class configuration {
     int setup;
     public:
     configuration () {
          setup = 1;
     }
     configuration (int a, int b) {
          setup = 2;
     }
     configuration (float a) {
          setup = 3;
     }

     void printwhich() {
          cout << setup << endl;
     }
};

int main()
{
     configuration item1(3.21);
     configuration item2;
     configuration item3(2, 4);
     item1.printwhich();
     item2.printwhich();
     item3.printwhich();
}
```

3 Given the following class that describes a point, write 2 operators:

a an addition operator that takes 2 points and returns a new point whose **x** and **y** values are the sum of the **x** and **y** values of the 2 points.

b an output streaming operator so that a point, which should appear in the form **(x, y)**, can be output.

Note that some **main** code is provided to show how the values can be used.

```
#include <iostream>
using namespace std;

class point {
     private:
     float x;
     float y;
     public:
     point () {
          x = 0.0;
          y = 0.0;
     }
     point (float a, float b) {
          x = a;
          y = b;
     }

     point operator+(point p) {
          return point(x+p.x, y+p.y);
     }

     friend ostream& operator <<(ostream&, const point&);
};

ostream& operator <<(ostream& os, const point& p) {
     os << "(" << p.x << ", " << p.y << ")";
}

int main()
{
     point p1(10.0, 10.0);
     point p2(4.0, 10.0);
     point p3;
     p3 = p1+p2;
     cout << p3 << endl;
}
```

Click here to see the answers.

// QUIZ ANSWERS

1 The constructor will be defined like a function with the name that is the same as the class—in this case, **customer**—with no return value. Because it is the default constructor, there is no parameter list. Inside the constructor, the variable **name** should be set to **unknown**.

```
customer () {
  name = "unknown";
}
```

2 The output would be:

```
3
1
2
```

Notice that the 3 variables are declared with 3 different constructors. The first variable uses the constructor that takes a floating-point parameter (and sets **setup** to **3**); the second one uses the default constructor, which takes no parameters (and sets **setup** to **1**); and the third one uses the constructor that takes 2 integer parameters (and sets **setup** to **2**).

3 a For the **+** operation, you can either define the operator within the class or as a **friend** outside the class. If you define it inside the class, it can be done like this:

```
point operator+(point p) {
  return point(x+p.x, y+p.y);
}
```

Notice that the result of the operation is a point. Because you define the operator inside the class, the first operand is assumed to be the point that the operator is a member of.

The second operand is another point (named **p**). Then, you just create a new point and return it. You use the constructor to set the point's values. Notice that the **x** value is set to be **x+p.x** and that **y** is similarly set: The **x** refers to the **x** member from the first operand, while the **p.x** is the **x** value for the second operand.

A second option would be to declare a friend function within the class:

```
friend point operator +(point, point);
```

and then define the function outside the class:

```
point operator +(point p1, point p2) {
  return point(p1.x+p2.x, p1.y+p2.y);
}
```

b For the streaming operation, you need to define the operator as a friend operator outside the class. You must first declare it to be a **friend** within the class:

```
friend ostream& operator <<(ostream&, const point&);
```

Note that the return type is an **ostream** reference. The 2 operands are an **ostream** (like **cout**) and a **point** (using pass by const reference so that you don't copy the data and can use literals in the call if you wish). Then, outside the class, you can define the function itself, just outputting to the **ostream**:

```
ostream& operator <<(ostream& os, const point& p) {
  os << "(" << p.x << ", " << p.y << ")";
}
```

Click here to go back to the quiz.

18 Dynamic Memory Allocation and Pointers

You usually think of the computer's main working memory (the memory stored in RAM) as one uniform block of memory. Items you declare get space set aside for them ahead of time. This is known as **static memory allocation**, and it's how standard variables and parameters are stored. But as you start developing more complex objects, sometimes you want to allocate new memory during the program—memory you didn't know you'd need ahead of time. That's what **dynamic memory allocation** is for.

IN THIS LECTURE:

// DEREFERENCING POINTERS

Main working memory comes in one of 2 varieties: the stack or the heap.

- The **stack** is static memory that you know and allocate as soon as you declare something. The stack stores all the variables and parameters.
- The **heap** is for dynamic memory allocation—for anything that's not known in advance and is instead allocated as needed. The heap is sometimes referred to more formally as the free store.

It's not uncommon to be uncertain as to how much memory you'll need. You use dynamic memory allocation whenever you want to store an uncertain amount of information that will keep coming in, and thus keep growing, over time. Often, classes hide this from you. The vector class hides how and where it's allocating new memory.

But vectors don't always fit the problem you have. Sometimes you need to allocate memory for yourself.

To allocate memory on the heap in this more dynamic way, you need a **pointer**, which is a variable that holds an address in memory. Usually, but not necessarily, there is some data at that point in memory, but keep in mind that the pointer variable does not store the data—just the location where the data is.

However, a pointer also defines the type of thing that should be found at the address it contains.

A pointer is declared much like any other variable, but there is some syntax to get used to.

- To create a pointer, you first list the type of the thing you're pointing to but then also include * (commonly read "star") after the type, indicating that this variable is going to contain an address where there's something of that type.
- Once you've declared a pointer, you can assign it a particular memory address. To get the memory address of a variable, you put & in front of the variable's name. Think of the memory address as being like GPS coordinates.
- Once you have a pointer declared and you've assigned a memory address, you can get the thing that it is pointing to—that is, the thing that is at that memory address—in a process known as **dereferencing the pointer**.

```
1   // Program 18_1
2   // Pointer example using *
3   #include <iostream>
4   using namespace std;
5
6   int main()
7   {
8      int x = 3;
9      int* y;
10     y = &x;
11     cout << *y << endl;
12     *y = 5;
13     cout << x << endl;
14  }
```

There are a few ways to dereference pointers. The most basic way is to use * in front of the pointer name.

In **Program 18_1**, you create an integer, x, that gets assigned the value 3 (8). This integer is a static variable.

Then, you create a new variable, y, that is a pointer to an integer (9). The pointer is not set to anything initially.

You assign y the address of x (10). In other words, y points to x.

When you output *y, then you are outputting the value that y is pointing to. In this case, the integer where it's pointing has the value 3, so you output 3 (11).

In the next line, you are assigning a value, 5, to the integer pointed to by y. So, the memory location that y is pointing to will get the value 5. This is the same memory location that is used by x (12).

So, when you then print out the value of x, you see that x has the value 5 (13).

Another way you can dereference a pointer is to use an array operator and access element 0.

The reason this works is because an array is actually a pointer! The array variable holds the address where the first item of the array is stored.

When you write [0], you are getting the value stored at the place the pointer is referring to. And if you write [1], you mean the thing stored one position in memory past the place the pointer is referring to.

So, **Program 18_2** works identically to **Program 18_1**. The only difference is that instead of writing *y, you write y[0] (11, 12).

If you have a pointer to an object—but only to an object—there's a third way you can dereference pointers. You can access public member variables or public member functions by using ->, which looks like an arrow, so it's known as the arrow operation.

This is the same as if you dereferenced the pointer and then used a dot to access the member variable or function.

```
1   // Program 18_2
2   // Pointer example using [0]
3   #include <iostream>
4   using namespace std;
5
6   int main()
7   {
8      int x = 3;
9      int* y;
10     y = &x;
11     cout << y[0] << endl;
12     y[0] = 5;
13     cout << x << endl;
14  }
```

POINTER ERRORS

Pointers are notorious for how many errors they can cause in programs. Pointer errors can be easy to make and really difficult to find when debugging. Yet there are cases when you do need to use pointers:

- when you are reading in an unknown amount of data, or
- when you need a lot of memory but the operating system, as it often does, has limited the amount that can be used on the stack.

Then, you need to allocate memory dynamically.

// DYNAMIC MEMORY ALLOCATION

Now that you have pointers, you can start to perform dynamic memory allocation.

The **new** command creates a new object on the heap and returns a pointer to that new object. To use the **new** command, you just write *new* followed by the type that you want to allocate. If there is a second constructor defined, in addition to the default constructor, you can invoke it by putting parentheses afterward, with the arguments you want to pass to the second constructor in parentheses.

If you want to allocate several of an object, you allocate an array on the heap. When you allocate an array, then after the variable name, you just put brackets with the size of the array you want inside.

When the array form of **new** is used, the default constructor is called for each of the elements.

Data stays on the heap until it is explicitly deallocated. Once you are done with whatever memory you allocated, you need to explicitly free it. Unlike the stack memory that is automatically freed when a function call completes, heap memory stays allocated until the end of the program—unless you free it.

To free memory on the heap, you use the **delete** command. You just write *delete* and give a pointer to the thing you want to delete. The allocated object will then be deleted from memory.

If you allocated an array, you have to instead use **delete[]** to delete the entire array that was allocated.

One other thing that's useful when setting up a default variable is a null pointer.

Sometimes, you have a pointer but don't want it to point to anything in particular; you just have it to use sometime later. In this case, you can set the pointer value to be the value **NULL**. The null pointer does not point to any usable memory location, but it functions as a special value you can have the program check when no value is what is needed.

// A GAME OF 20 QUESTIONS

Let's create a 20 Questions type of game, in which you ask someone up to 20 true/false questions to try to guess what he or she is thinking of. For this game, you'll want to create a tree structure where every internal node is a question, and if the answer is *true*, you will head down one side of the hierarchy, and if the answer is *false*, you'll head down the other.

One way to do this is to generate a **binary tree**, where every node has at most 2 children. Each of the leaf nodes in your game tree will be a guess—the thing you hope the person is thinking of.

If your program guesses incorrectly, and thereby loses, then the program will ask the person playing to volunteer a question that teaches it how to distinguish its guess from the person's item. That will let your game "learn" so that it can play better the next time. You'll create a new question node—which you'll allocate on the heap—and another new node for the new guess. In other words, the previous, wrong guess will be one child and the new node will be the other child of the question node.

During a game, the program might first ask **`Is it alive?`**. If the answer is **`No`**, then it asks **`Can you hold it in your hand?`**. If the answer is **`Yes`**, then the program might guess: **`Is it a rock?`**.

If that is not the correct answer and the correct answer was **`a pencil`**, the program would then ask the user **`What is a question to distinguish a rock from a pencil?`** and use the user's response, which might be **`Do you use it to write?`**, to update the tree.

You'll have a class named **`node`**, because each object that you create will be one node of the tree. The node is going to have 4 member variables (a).

The first of these is a Boolean, **`is_answer`**, which will be **`true`** if the node is an answer node and **`false`** if it's a question node.

Next, you'll have a string named **`text`**. For an answer node, this will be the thing you are going to guess. For a question node, it will be the question you ask to distinguish between the 2 items.

Finally, you'll have 2 pointers to more nodes: a pointer to the **`false_answer`** and a pointer to the **`true_answer`**. For a question node, one of these will be a pointer to the node that you should go to next if the answer to the question is **`Yes`**, and the other will be the one to go to if the answer is **`No`**.

You'll create 2 constructors. The default constructor (b) will create an answer node with the **`text`** value set to **`rock`**. So, it'll set **`is_answer`** to **`true`** and **`text`** to **`rock`**.

Both the **`false_answer`** pointer and the **`true_answer`** pointer will be set to **`NULL`**. This is an answer node, so there are no children and therefore nothing to point to.

The second constructor (c) will take in a string as a parameter. It'll again create an answer node such that **`is_answer`** is **`true`** and both pointers are **`NULL`**.

```
1   // Program 18_5
2   // 20 Questions Game
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   class node {
8       bool is_answer; // True if
    an "answer" node, False if a
    "question" node
9       string text; // The answer
    for an answer node, the question
    otherwise
10      node* false_answer;
11      node* true_answer;
12
13  public:
14      node() {
15          is_answer = true;
16          text = "rock";
17          false_answer = NULL;
18          true_answer = NULL;
19      }
20
21      node(string s) {
22          is_answer = true;
23          text = s;
24          false_answer = NULL;
25          true_answer = NULL;
26      }
27
```

a

b

c

...

```
...

28      bool ask_question() {
29          if (is_answer) {
30              // This is an answer node.
31              // Guess the answer, and if it's wrong, generate new nodes
32              cout << "OK, I'm ready to guess: is it " << text << "? (Answer Yes or No): ";
33              string answer;
34              cin >> answer;
35              if ((answer == "Yes") || (answer == "yes") || (answer == "y") ||
36                  (answer == "Y")) {
37                  return true;
38              }
39              else {
40                  cout << "What were you thinking of? ";
41                  cin >> answer;
42                  cout << "Help me learn.  What is a question that would help me distinguish "
43                      << text << " from " << answer << "? " << endl;
44                  cout << " (The answer to the question should be Yes for " << answer
45                      << " and No for " << text << ".)" << endl;
46                  string question;
47                  getline(cin, question);
48                  getline(cin, question);

...
```

d

e

f

g

h

The **main** part of the class will be the **ask_question** member function. This is going to be the function to call to have the computer ask the question in that node—either trying to guess the answer or asking a question to narrow down choices.

The **ask_question** function will return **true** if the computer wins the game—the function will guess the answer—or return **false** if it doesn't win the game. The function will have 2 parts: one option if it's a node with a final guess and one if it's a node to ask another question.

If it is a node with a final guess, then you'll make a guess of what you think the player is thinking of. You output your guess, which is the **text** member variable (d).

You then read an answer from the user (e). If the answer is **Yes**, it means you guessed the correct thing, so you just return **true**, and the game is over (f). If your guess was not correct, you need to update your decision tree. You first find out what item the person was thinking of. This is going to become a new answer node (g).

You next ask the user to give you a question to distinguish your incorrect guess from the thing the user was thinking of (h). This question gets read in as an entire line, because it is not just one word. You use the **getline** command to read the question in.

Remember from lecture 9 that **getline** first reads any leftover carriage return that might have come after streaming in a variable as an entire line, so you have to call **getline** twice: once to get the new line that came from pressing Enter after typing the name of the thing the user was thinking of, and another line will read in the new thing the user wrote.

```
...
49                  true_answer = new node(answer);
50                  false_answer = new node(text);
51                  text = question;                    i
52                  is_answer = false;
53                  return false;                       j
54              }
55          }
56          else {
57              // This is a question node.  Ask the question, get the answer, and
    go to a child
58              cout << text;
59              cout << " (Answer Yes or No): ";        k
60              string answer;
61              cin >> answer;
62              if ((answer == "Yes") || (answer == "yes") || (answer == "y") ||
63                  (answer == "Y")) {
64                  return true_answer->ask_question();
65              }
66              else {                                  l
67                  return false_answer->ask_question();
68              }
69          }
70      }
71  };
72
                                                    ...
```

Following this, you're going to create 2 new children nodes (i). Both of these will be answer nodes. You use the **new** command to create a new node, using the constructor to give the answer text to use for each. The **new** command allocates a new node on the heap, and the constructor initializes that node appropriately. One of the nodes will have the new item as an answer, and the other one will have the existing answer.

Then, you update the current node (j). Instead of this node being an **answer** node, it will be a **question** node, so you change the text to be the question that you just read in and set **is_answer** to **false**. This designates that it's now a question node that can be used to ask the question and go to one of the 2 answer nodes: either the new one you just learned of or the answer that you previously had in this node. So, next time you play the game, instead of guessing the wrong answer, you'll instead ask a question to distinguish your answers.

Finally, you return **false** (53). You did not guess the thing the person was thinking of, so you did not win the game.

On the other hand, you need to handle the case where you have a question node. In this case, the operation is relatively simple. You print out the question and get a **Yes** or **No** answer from the user (k).

If the user answered **Yes**, then you call **ask_question** on the node pointed to by the **true_answer** pointer. Otherwise, you call **ask_question** on the node pointed to by the **false_answer** pointer. In either case, you just return whatever the result of that node is (l).

Basically, you go down to the next level of the tree and again ask a question there—either making a guess if it's an answer node or asking another question if it's a question node. This will continue until you eventually reach a leaf of the tree, where you'll have an answer node.

This final operation is an example of **recursion**, which refers to a function calling itself. In this case, the member function **ask_question** was calling **ask_question** again, though this time on a different node. Once you are at a node, you treat the remaining tree just like you would the original tree; in other words, for any one question node, the subtrees are themselves trees.

The only thing left is the **main** program that actually runs your game. You start the game by creating a node, with the default constructor, named **firstnode** (75), which will create an answer node with the guess **rock**. So, the game will always start the first time with the computer guessing that the user is thinking of a rock.

Then, you will repeatedly ask the user if he or she wants to play the game, and as long as the user keeps answering **Yes**, you play another round (m).

Each time, to play the game, you just call **ask_question** on **firstnode** (82). Depending on whether you get a **true** or a **false** returned, you either celebrate winning the game (83) or sulk that you've lost (86).

Assuming the computer doesn't win, the more you play the game, the bigger the tree will get—that is, the more your program will "learn." In principle, you could go on until the computer runs out of memory; in practice, the computer will eventually win or you'll run out of time to play. Dynamic allocation of memory gives you the potential to expand without limit.

```
...

73   int main()
74   {
75      node firstnode;
76
77      cout << "Would you like to play 20 questions? ";
78      string answer;
79      cin >> answer;
80      while ((answer == "Yes") || (answer == "yes") || (answer == "y") ||
81          (answer == "Y")) {
82          if (firstnode.ask_question()) {
83              cout << endl << "I won!!!" << endl;
84          }
85          else {
86              cout << endl << "OK, I guess I lost..." << endl;
87          }
88          cout << "Would you like to play again? ";
89          cin >> answer;
90      }
91   }
```

m

Dynamic memory allocation is something you need to keep in mind if

- you need to allocate an amount of memory that you cannot predict ahead of time, or
- the memory you need has a special structure, such as the binary tree used in the 20 Questions game.

// DESTRUCTOR FUNCTIONS

Destructors are functions that are called when it is time for an object to be removed from memory. The destructor function is meant to "clean up" when it's time to get rid of an object. In code, a destructor is declared just like constructor functions, except that a destructor starts with a tilde (~), after which comes the class name and parentheses with no parameters.

```
~node() {
    if (!is_answer) {
      delete false_answer;
      delete true_answer;
    }
}
```

In particular, the destructor is the place to delete previously allocated dynamic memory. For example, your 20 Questions program keeps all the memory you allocated until the end of the program.

It is always good practice to free up any allocated memory in the same object that created it. This means that when it's time to delete a node, you also need to delete its children branches, so you should define a destructor in this case.

The destructor will only need to call **`delete`** on the children if it is a question node, because answer nodes don't have any children, so you first check to see if it's a question node. If so, you'll call **`delete`** on each of the children. The destructor will then be called on *those* nodes, which will call the destructors on their children, and so on, all the way through the tree.

// VECTORS: AN ALTERNATIVE TO DYNAMIC MEMORY ALLOCATION

Vectors are a way of indirectly allocating data on the heap without having to use pointers or the allocating and destructor commands for **`new`** and **`delete`**. All that work is hidden inside the vector class.

When a vector is first created, there is enough space allocated on the heap to hold all the data in the initialization, plus a little more. As you keep adding elements onto a vector, eventually the memory that was allocated gets used up.

But when the vector is finally "full," where it's used all the memory that was already allocated, it then goes through a process to get more. It allocates double the amount of memory, copies all the old data into this new memory, and then deletes the old memory. Now it has twice as much memory to work with, so it should take even longer before that memory fills up. This can keep going as long as more data keeps being added to a vector.

If you find yourself wondering whether you should use a vector or write your own dynamic memory allocation, then use the vector! With a vector, you're still using the heap, but it's a little easier to use, and you're much less likely to run into major errors or problems that people tend to encounter when using pointers. ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 17.4, 17.5, and 17.9.

b Lippman, Lajoie, and Moo, *C++ Primer*, chap. 12.

// QUIZ

1 Are the following true or false?

a Dynamic memory is sometimes called the stack.

b A "normal" variable declaration, such as `int a;`, is an example of dynamic memory allocation.

c Some classes hide dynamic memory allocation from the user.

d Dynamic memory allocation should be used as often as static memory allocation.

2 Which one/ones of the following is/are a good reason you might want a destructor for a class?

a You have private member variables.

b Your class dynamically allocated memory.

c You want to notify other objects that this is being destroyed.

3 Assume you have a class **testclass**, defined below:

```
class testclass {
  public:
  int x;

  testclass() {
    x=0;
  }
  testclass(int a) {
    x = a;
  }

  void print() {
    cout << x << endl;
  }
};
```

How would you do each of the following?

a Declare a pointer to a **testclass** object, named **p**.

b Dynamically allocate a **testclass** object and assign it to **p** using the default constructor (there are at least 2 possible solutions).

c Dynamically allocate a **testclass** object with the parameter **5** and assign it to **p**.

d Dynamically allocate an array of 5 **testclass** objects and assign the array to **p**.

e Call the **print** function for the object pointed to by **p** (there are at least 3 possible solutions).

f Destroy the object(s) pointed to by **p**.

Click here to see the answers.

// QUIZ ANSWERS

1 a False. Dynamic memory is also known as the heap or the free store; the stack is used to refer to static memory.

b False. Those "normal" declarations are for variables you know exist at compile time. They are allocated in static memory; dynamic memory is allocated when you use **new** to set aside memory during a program's execution that was not necessarily known ahead of time.

c True. The vector class is one example. It is performing dynamic memory allocation to grow as needed when additional items are pushed onto the back of the vector.

d False. Dynamic memory allocation has the potential for creating numerous bugs, security flaws, and general program instability. It is better to use static memory when feasible and to use dynamic memory allocation when you need to. An example of when you might need to is when you are dealing with an indeterminate amount of data.

2 a No. This is not a good reason on its own. Whether variables are private or public does not really matter as far as whether a destructor is needed.

b Yes. This is typically a main reason a destructor is used. A destructor will typically free the memory that was dynamically allocated for that object so that it does not stay allocated when the object is deleted.

c Yes. This is a reason to implement a destructor. If other objects are pointing to this object, then when it is deleted, they could have a bad pointer. The destructor might include code that tells those objects that this one is being deleted.

3 a
```
testclass* p;
```
The * indicates that the new variable will be a pointer to something of that type.

b Here are 2 options:
```
p = new testclass;
  OR
p = new testclass();
```
In either case, the default constructor is called.

c
```
p = new testclass(5);
```
By giving arguments when allocating the object, the corresponding constructor is called for that object.

d
```
p = new testclass[5];
```
This allocates an array of 5 objects, each initialized with the default constructor.

e Here are 3 equivalent options:
```
p->print();
(*p).print();
p[0].print();
```
In each case, the pointer is dereferenced to get the object it is pointing to and then the function is called for that object.

f There are 2 ways to do this. If there is just a single object pointed to by **p** (i.e., only one object was allocated), then you call the following:

```
delete p;
```

Otherwise, if you allocated an array of objects, you should instead call the following:

```
delete[] p;
```

The **delete** command deallocates the memory that was assigned to that object after calling the destructor (if there is one; in this case, there is not).

Click here to go back to the quiz.

19 Object-Oriented Programming with Inheritance

The most important idea of object-oriented programming is encapsulation, which is the wrapping of data and functions together in one tidy package. Another powerful idea is **inheritance**, which is the idea that you should be able to create classes that can inherit this encapsulation of member variables and functions from another class. In short, you can use an abstraction you already have to create one or more new abstractions. This creates a hierarchy: Objects can inherit from a "parent," and the relationships among various inheriting objects can form a tree structure.

IN THIS LECTURE:

// INHERITANCE

Inheritance offers several benefits.

- It prevents having to rewrite the same code over and over. An ancestor can pass on properties to all of its offspring. In terms of coding, this can save you a lot of time and work.
- It reduces the number of places that bugs could be introduced to the program, so you reduce debugging time.
- It enables polymorphism, which you'll learn about in the next lecture.

In practice, there are a few ways that inheritance works.

From a top-down viewpoint, object-oriented inheritance can be thought of as a way of taking a more general idea and dividing it into more specialized ideas. But inheritance can also work from a bottom-up viewpoint, where you start with the offspring and decide what sort of ancestor they have in common. Whether top-down or bottom-up, the end result is a similar arrangement of classes.

Inheritance plays a powerful role in object-oriented design. It lets you define something a single time in one class and then use it in multiple other classes. It means that you can define a new class, and rather than writing every piece of it from scratch, you can start with everything already defined in another class.

Inheritance also creates a hierarchy of classes—a relationship between ancestors and descendants.

TERMINOLOGY

There are different terminologies that are sometimes used to describe the relationships between different classes in a class hierarchy.

Following the terminology used for trees, you may refer to a *parent* class and *child* class. One parent can have several children, also known as *offspring*.

A *superclass* is the class above a point in the hierarchy, and a *subclass* is a class below a point in the hierarchy. A *superclass* encompasses many different classes, while a *subclass* is a narrower description of a class.

The *base* class is effectively the parent, and the *derived* class is the child.

For the most part, if you hear the terms *parent class*, *superclass*, or *base class*, they probably mean the same thing. And if you hear the terms *child class*, *subclass*, or *derived class*, they probably mean the same thing.

You've already seen a few areas in which inheritance is used to define classes. One important area involves streams. You've seen streaming to and from the console, to and from files, and to and from strings. All of these are basically derived classes that inherit from a more general **stream** base class.

A second area of inheritance you've already encountered is **exceptions**, which are used to cause a function to exit when an exceptional case is encountered. There is a very general exception class and then more specialized child classes, each of which has even more specific classes.

In this example, you have a general class, the **product**. Products contain floating-point **member variables** for both a wholesale cost and a retail cost (a).

Then, there is a separate class created for a more specific product: the **cup**. The cup has 2 member variables of its own: a **volume** that the cup holds and a **color** for the cup, represented as a float and a string, respectively (b).

The key difference here from what you've seen before is that the cup is also defined to be a child of the product. You see this by the **: public product** placed right after the class name. Because this is the case, the cup inherits the member variables from the **product** class.

```
1   // Program 19_1
2   // Inheritance Example
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   class product {
8   public:
9      float wholesale_cost;
10     float retail_cost;
11  };                                  a
12
13  class cup : public product {
14  public:
15     float volume;
16     string color;
17  };                                  b
18
19  int main() {
20     cup plastic_cup;
21     plastic_cup.wholesale_cost = 0.10;
22     plastic_cup.retail_cost = 0.30;
23     plastic_cup.volume = 16.0;
24     plastic_cup.color = "red";       c
25
26       cout << "The plastic cup costs "
    << plastic_cup.wholesale_cost
27           << " to purchase and sells for
    " << plastic_cup.retail_cost << endl;
28  }
```

Then, in the **main** routine, you can define a **cup** object, named **plastic_cup**. This will let you set not just the volume and color, but also the wholesale and retail costs for the cup. You do this by assigning values to **plastic_cup.wholesale_cost** and **plastic_cup.retail_cost**, the same as you would the other member variables, **plastic_cup.volume** and **plastic_cup.color** (c).

To keep things simple, let's assume that everything is public.

When creating a class, if you want to inherit from some other class, you declare the class as usual, but you have a colon, an access statement like **public** or **private**, and then the name of the parent class.

In **Program 19_2a**, because you wanted to create a **cup** class that was derived from the **product** class, you would write **class cup : public product** and then the class definition in curly braces (13).

You can also inherit in a larger class hierarchy. For example, say you wanted to further distinguish your cups so that some of them were specifically measuring cups. You could make a new class, a **measuring_cup** class. It would inherit from **cup**, so you'd write **class measuring_cup : public cup** in the declaration (19).

That means that it inherits the features of the **cup** class, which therefore means it also inherits the features of the **product** class. It can also define its own member variables and functions. In this case, you can define 2 new variables for the **measuring_cup**: a Boolean to note whether it is in metric or English units and an integer number of gradations (d). Thus, a **measuring_cup** actually has access to 6 different member variables: the 2 that it defined for measuring, the 2 that were defined in the **cup** class, and the 2 that were defined in the **product** class (e).

```
1   // Program 19_2_a
2   // Inheritance Example - multiple levels of inheritance
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   class product {
8   public:
9      float wholesale_cost;
10     float retail_cost;
11  };
12
13  class cup : public product {
14  public:
15     float volume;
16     string color;
17  };
18
19  class measuring_cup : public cup {
20  public:
21     bool metricunits;
22     int num_gradations;
23  };
24
25  int main() {
26     measuring_cup plastic_cup;
27     plastic_cup.wholesale_cost = 0.10;
28     plastic_cup.retail_cost = 0.30;
29     plastic_cup.volume = 1000.0;
30     plastic_cup.color = "clear";
31     plastic_cup.metricunits = true;
32     plastic_cup.num_gradations = 10;
33
34     cout << "The plastic cup costs " << plastic_cup.wholesale_cost
35         << " to purchase and sells for " << plastic_cup.retail_cost << endl;
36     cout << "It is " << plastic_cup.color << " in color and has a volume of "
37         << plastic_cup.volume << endl;
38     if (plastic_cup.metricunits) {
39         cout << "It has a total of " << plastic_cup.num_gradations
40             << " markings in metric units." << endl;
41     }
42     else {
43         cout << "It has a total of " << plastic_cup.num_gradations
44             << " markings, in English units." << endl;
45     }
46  }
```

d

e

```
1   // Program 19_3
2   // Inheritance Example - inheriting member functions
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   class product {
8   public:
9      float wholesale_cost;
10     float retail_cost;
11
12     float profit_per_unit() {
13         return retail_cost - wholesale_cost;
14     }
15  };
16
17  class plate : public product {
18  public:
19     float diameter;
20  };
21
22  class cup : public product {
23  public:
24     float volume;
25     string color;
26  };
27
28  class measuring_cup : public cup {
29  public:
30     bool metricunits;
31     int num_gradations;
32  };
33
34  class drinking_cup : public cup {
35  public:
36     bool has_lid;
37  };
38
39  int main() {
40     measuring_cup plastic_cup;
41     plastic_cup.wholesale_cost = 0.10;
42     plastic_cup.retail_cost = 0.30;
43     plastic_cup.volume = 1000.0;
44     plastic_cup.color = "clear";
45     plastic_cup.metricunits = true;
46     plastic_cup.num_gradations - 10;
47
48     cout << "The plastic cup costs " << plastic_cup.wholesale_cost
49         << " to purchase and sells for " << plastic_cup.retail_cost << endl;
50     cout << "We make a profit of " << plastic_cup.profit_per_unit()
51         << " on each one." << endl;
52  }
```

g

f

h

And, just as you'd expect from a hierarchy, you can have more than one derived class from the same base class. For example, in addition to a **measuring_cup**, you could define a **drinking_cup**. In **Program 19_3**, it's been defined as a subclass of **cup**, with one more member variable: a Boolean denoting whether it has a lid or not (f).

You're not confined to inheriting just member variables; you also inherit member functions.

Say your product class had a member function named **profit_per_unit**. It would be a simple function, just returning the difference between the retail and wholesale prices (g).

Then, if you have an object of some derived type, such as a **measuring_cup**, you can call that function on the **measuring_cup**, just like you would for the base class. In this example, you are able to have a **measuring_cup** object named **plastic_cup**, and you call **plastic_cup.profit_per_unit** to get the profit (h).

Inheritance only goes one way—from the parent to the child. A superclass does not have access to the member variables and functions of the subclasses, and sibling classes don't have access to each other's variables.

Exercise 1

Let's say that in addition to cups, you wanted to sell plates. Create a new type of product, a plate, that includes a diameter measure.

Click here to see the solution.

// THE PROTECTED CATEGORY

Up until this point, the examples have had all of the member variables be public. But, as discussed when encapsulation was introduced, the idea of **information hiding**—of not allowing outsiders access to more information than they need—is important to creating good classes. It helps ensure that a class's member variables are only modified in approved ways, so you declare some member variables as **private** and some as **public**.

When it comes to inheritance, there's a third category that's in between public and private: **protected**.

As before, things that are public will get inherited by all of the descendants and be public members of all those descendants. Any code can call those member functions or access those member variables directly.

The private member properties are accessible only to the class they are declared in, and no others. Not even the descendants of a class can access those member properties that are considered private. So, although the descendants still inherit those variables and functions—that is, they still allocate space for those member variables and can have values stored there—they can't actually call the private functions or read the private variables defined in an ancestor.

This is the reason for the protected category. Protected member properties are accessible to the class they're declared in and to the descendants, but nothing outside of the class. In effect, this is like a private member property for everything following in the inheritance tree.

Let's say you create a **bank_account** class that can be used to keep track of different bank accounts. You'll have a base class called **bank_account** and 2 derived classes, **checking_account** and **savings_account**.

The **bank_account** class is declared just like any other class (7). It's not inheriting from anything.

Within the **bank_account** class will be 3 groups of member properties. First, you have private member properties (i), which in this case are the owner's name, address, and account number. Because these are private, that means that the derived classes, which will be **checking_account** and **savings_account**, will have access to those member variables but won't be able to access them directly. The only way they'll be able to access these variables is indirectly, through a member function inherited from **bank_account**. No other classes outside **bank_account** will be able to access them directly, either.

```
1    // Program 19_4
2    // Inheritance Example: Public/
     Protected/Private
3    #include<iostream>
4    #include<string>
5    using namespace std;
6
7    class bank_account {
8    private:                          i
9       string owner_name;
10      string owner_address;
11      long long int account_number;
12
```

...

You also have a couple members declared **protected** (j). There is a protected variable: a double-precision floating-point number named **balance**. There's also a protected function: a **set_account** function that sets the account number and the balance.

Because the **balance** variable and the **set_account** function are protected, any subclasses you might define under **bank_account** will be able to access them; that is, the subclasses will be able to modify the **balance** directly and call the **set_account** function. But no other classes—and no code outside of the **bank_account** subclasses—will be able to access protected variables or functions. No outside code can call **set_account**, and no outside code can look at or modify the **balance**.

Then, you have a few public functions (k): One lets the account name and address be changed, and the other prints out the account information. Notice that because both of these are public, they can be called from anywhere. Any other code, from any class or function, can call either of these 2 public functions.

Now that your **bank_account** class is all set up, you can turn to subclasses. Suppose you have a **checking_account** class and a **savings_account** class, each of which inherits from **bank_account**, as specified with the **: public bank_account** designation (l).

First, let's look at a way to implement the **checking_account**, which has a few public functions. One of these, called **begin_account**, is used to indicate that a new account is starting. It just calls **set_account**. Because **begin_account** is part of the **checking_account** class and the **checking_account** inherited from **bank_account**, it is able to call the **set_account** function that was a protected function of **bank_account** (m).

The **checking_account** also has functions to deposit and withdraw money. Each of these refers to the protected **balance** variable that is inherited from **bank_account**. Again, because this class inherits from **bank_account**, it can access the protected member properties (n).

```
...
13  protected:                                                    j
14     double balance;
15
16     void set_account(long long int acctnum, double
   startbalance = 0.0) {
17         account_number = acctnum;
18         balance = startbalance;
19     }
20
21  public:                                                       k
22     void update_owner(string name, string address) {
23         owner_name = name;
24         owner_address = address;
25     }
26
27     void print_account_info() {
28         cout << "Account: " << account_number << endl;
29         cout << "Current Balance : " << balance
   << endl;
30         cout << "Owner: " << owner_name << endl;
31         cout << "Address: " << owner_address << endl;
32     }
33  };
34
35  class checking_account : public bank_account {              l
36  public:
37     void begin_account(long long int num,                    m
   double amt) {
38         set_account(num, amt);
39     }
40
41     void deposit(double amt) {                               n
42         balance += amt;
43     }
44
45     double withdraw(double amt) {
46         if (balance >= amt) {
47             balance -= amt;
48             return amt;
49         }
50         else {
51             return 0;
52         }
53     }
54  };
55
56  class savings_account : public bank_account {
...
```

In the **savings_account** class, there's a private member variable, **interest_rate** (58). Remember that for classes, if you don't declare it as public, private, or protected, it's assumed to be private. But just to be especially clear, let's explicitly declare it private by putting **private:** beforehand.

You also have 2 public member functions that you might want to invoke from outside the class: **begin_account** and **generate_interest**. Like the **checking_account**, these access the protected function **set_account** and the protected variable **balance** that are inherited from the **bank_account** (o).

If you then, in your **main** code, want to work with some individual's accounts, you can declare both a **checking_account** and a **savings_account** object for that person. In this case, you can have accounts named **Holmes_checking** and **Holmes_savings**, and you can call any of the public member functions on them. So, you can call **begin_account** defined in the **checking_account** or **savings_account** class. You can call **deposit** and **withdraw** on the **checking_account** or **generate_interest** on the **savings_account**. Or you can call **update_owner** and **print_account**, functions that were defined in the base class **bank_account**.

```
...

57  private:
58      double interest_rate;
59
60  public:
61      void begin_account(long long int num, double amt, double rate) {
62          set_account(num, amt);
63          interest_rate = rate;
64      }
65
66      void generate_interest() {
67          balance = balance + interest_rate * balance;
68      }
69  };
70
71  int main() {
72      checking_account Holmes_checking;
73      savings_account Holmes_savings;
74      Holmes_checking.begin_account(11122233, 100.0);
75      Holmes_checking.update_owner("Sherlock Holmes", "221B Baker St., London");
76      Holmes_savings.begin_account(99988877, 500.0, 0.03);
77      Holmes_savings.update_owner("Sherlock Holmes", "221B Baker St., London");
78
79      Holmes_checking.deposit(50.0);
80      Holmes_checking.withdraw(25.0);
81      Holmes_savings.generate_interest();
82
83      Holmes_checking.print_account_info();
84      Holmes_savings.print_account_info();
85  }
```

o

All of this runs fine. Within any class, you are accessing only functions and variables that were either defined in that class or were public or protected properties from an ancestor. From outside of the classes, you are only accessing public properties of the class.

But there are several things you would not be allowed to do. You could not access a private member variable of the parent class, and you could not access any private member function from outside the class.

// CONSTRUCTORS WITH INHERITANCE

```
1   // Program 19_5
2   // Inheritance Example: Constructors
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   class bank_account {
8   private:
9      string owner_name;
10     string owner_address;
11     long long int account_number;
12
13  protected:
14     double balance;
15
16     void set_account(long long int acctnum, double startbalance
    = 0.0) {
17        account_number = acctnum;
18        balance = startbalance;
19     }
20
21  public:
22     bank_account() {                                    p
23        owner_name = "";
24        owner_address = "";
25        account_number = 0;
26        balance = 0.0;
27     }
28
29     bank_account(string name, string address, long long int num,
    double bal) {
30        owner_name = name;
31        owner_address = address;
32        account_number = num;
33        balance = bal;
34     }                                                   q
35
```

...

When using inheritance, constructors for the most part work much like all the constructors you've seen before, but it can be tricky to handle constructors when you're inheriting some of your properties and don't have direct access to them.

The way to think of this is that when you call a constructor in a derived class, you need to first have a constructor for the base class. After all, the derived class is only adding additional stuff onto whatever was in the base class. So, first you need to have the base class set up, and then you can add on whatever additional things are needed for the derived class.

The way this is done is that when a constructor is defined, you put a colon after the constructor declaration, giving the constructor to use for the base class.

For example, let's define 2 constructors for the base **bank_account**. First, you'll have a default constructor that basically sets everything to empty strings or to **0** (p). Then, you'll have a constructor that takes in all the various parameters needed to initialize all the member variables to specific values. In this case, there are 4 parameters: for the owner's name, owner's address, account number, and starting balance. You use these parameters to initialize the various member variables of the **bank_account** class. All of this is just like you would have had for any other constructor (q).

```
...
    void update_owner(string name, string address) {
        owner_name = name;
        owner_address = address;
    }

    void print_account_info() {
        cout << "Account: " << account_number << endl;
        cout << "Current Balance : " << balance << endl;
        cout << "Owner: " << owner_name << endl;
        cout << "Address: " << owner_address << endl;
    }
};

class checking_account : public bank_account {
public:

    checking_account() : bank_account() {
    }                                                    r

    checking_account(string name, string address, long long int num,
        double bal) : bank_account(name, address, num, bal) {
    }                                                    s


    void begin_account(long long int num, double amt) {
        set_account(num, amt);
    }

    void deposit(double amt) {
        balance += amt;
    }

    double withdraw(double amt) {
        if (balance >= amt) {
            balance -= amt;
            return amt;
        }
        else {
            return 0;
        }
    }
};

...
```

Now let's look at the constructors for the derived classes, the checking and savings accounts. For a **checking_account**, there are no new member variables. So, the **checking_account** constructor doesn't need to do anything more than call the base class **bank_account** constructor. You define 2 constructors for checking accounts: one default one and one with the 4 parameters needed to initialize a **bank_account**.

For the default constructor, you write **checking_account() : bank_account()** and then a few curly braces with nothing inside (r). That **: bank_account** with no arguments in the parentheses means that when this default **checking_account** constructor is called, it will first call the **bank_account** constructor with no arguments. In other words, it calls the default **bank_account** constructor first. There's nothing in the curly braces, so nothing more is done.

Likewise, for the other constructor, you'll call the **bank_account** constructor with 4 parameters (s). Again, there won't be anything in the curly braces. You just define the **checking_account** constructor to take in 4 parameters and then write **: bank_account** and, in the parentheses, pass on those same 4 parameters as arguments to the constructor. This will call the non-default **bank_account** constructor with those arguments and then do whatever is in the curly braces, which in this case is nothing.

```
...
79  class savings_account : public bank_
    account {
80     double interest_rate;
81
82  public:
83     savings_account() : bank_account() {
84        interest_rate = 0.0;
85     }                                          t
86
87     savings_account(string name, string
    address, long long int num, double bal,
88        double rate) : bank_account(name,
    address, num, bal) {
89        interest_rate = rate;
90     }                                          u
91
92     void begin_account(long long int
    num, double amt, double rate) {
93        set_account(num, amt);
94        interest_rate = rate;
95     }
96
97     void generate_interest() {
98        balance = balance + interest_rate
    * balance;
99     }
100 };
101
102 int main() {
103    checking_account my_checking("John
    Keyser", "123 Any St., Anytown, TX
    77777",
104       11122233, 100.0);
105    savings_account my_savings("John
    Keyser", "123 Any St., Anytown, TX
    77777",
106       99988877, 500.0, 0.03);
107
108    my_checking.print_account_info();
109    my_savings.print_account_info();        v
110 }
```

With the **savings_account**, there is one more member variable. You'll again define 2 **savings_account** constructors, just like for the **checking_account**. Everything will be the same about these, except that the member variable **interest_rate** will need to be set in each constructor. The default constructor just sets this member variable to **0.0** after calling the default **bank_account** constructor (t). The other constructor takes in 5 parameters and passes 4 of those on as arguments to the **bank_account** constructor. Then, it uses the fifth parameter to initialize the **interest_rate**, inside the curly braces (u).

> When you are defining a constructor in a derived child class, you probably need to make use of the constructor in the parent class to fully initialize an object. In particular, if the parent has any private member variables, the way you'll initialize those is by using the parent's constructor, because the derived class can't access those variables itself.

You can see that if you declare instances of the **checking_account** and **savings_account** variables with the non-default constructors, you do get everything initialized correctly (v). ◆

Exercise 2

Say you owned a store called Everything Wrists that sold wrist accessories, such as bracelets, watches, fitness trackers, and rubber wristbands. You want to keep track of your products, so you'll put together some classes that can be used to describe products.

Think of the classes you might want, including which ones might inherit from others. What data might each class contain? How might you go about implementing a few of those classes?

Click here to see the solution.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chaps. 12 and 13.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 15.1 and 15.2.

Exercise 1 Solution

Here's one way that could have been implemented.

```
class plate: public product {
    public:
    float diameter;
};
```

Click here to go back to the exercise.

Exercise 2 Solution

Here's one option. You could come up with lots of data and many ways to organize this.

```
// Program 19_6
// Everything Wrists store example
#include <iostream>
#include <string>
using namespace std;

class accessory {
public:
    float cost;
    float sale_price;

    // Default constructor
    accessory() {
        cost = 0.0;
        sale_price = 0.0;
    }

    // Non-default constructor
    accessory(float c, float p) {
        cost = c;
        sale_price = p;
    }
};

class rubber_wristband : public accessory {
public:
    string color;
    string message;

    // Default constructor
    rubber_wristband() : accessory() {
        color = "";
        message = "";
    }

    // Non-default constructor
    rubber_wristband(float c, float p, string col, string mess) : accessory(c, p) {
        color = col;
        message = mess;
    }
};

int main()
{
    rubber_wristband birthday_band(0.20, 2.00, "Yellow", "It's My Birthday!");
    rubber_wristband bridges_band(0.20, 0.50, "Black",
        "Support Bridges - your life depends on them!");
}
```

Click here to go back to the exercise.

// QUIZ

1 Imagine you have 3 different classes:

Class A
Class B, which inherits from class A
Class C, not related to classes A or B

a If the member of Class A is private:
   i Is it accessible from Class B?
   ii Is it accessible from Class C?

b If the member of Class A is protected:
   i Is it accessible from Class B?
   ii Is it accessible from Class C?

c If the member of Class A is public:
   i Is it accessible from Class B?
   ii Is it accessible from Class C?

2 Imagine you want the following classes, each with a set of member variables, as listed. How might you use inheritance to group these together in a hierarchy of classes? In other words, what new classes might you create that would allow you to group all of these together in one hierarchy?

- Shopping Center: Appraised Value, Square Feet, Rental Price per Square Foot, Length of street frontage
- Restaurant: Appraised Value, Square Feet, Rental Price per Square Foot, Kitchen area, Dining area
- Office Building: Appraised Value, Square Feet, Rental Price per Square Foot, Number of exterior entries
- Industrial Facility: Appraised Value, Square Feet, Rental Price per Square Foot, Number of loading docks
- Single-Family House: Appraised Value, Square Feet, Number of people who can live there, Yard Size, Number of bedrooms and bathrooms
- Multiplex: Appraised Value, Square Feet, Number of people who can live there, Yard Size, Number of separate units
- Apartment: Appraised Value, Square Feet, Number of people who can live there, Monthly Rent
- Condo: Appraised Value, Square Feet, Number of people who can live there, Monthly Maintenance Fee

3 Imagine you are given a class, **airplane**, that includes a member variable, **weight**, as shown below.

```
class airplane {
  public:
  float weight;
};
```

a How would you create a class, **jet**, that is a subclass of **airplane** and also defines a new member variable, **numengines**?

b Now assume that you have the following variables declared:

```
jet commercialplane;
airplane privateplane;
```

Which of the following commands are valid?

```
commercialplane.weight = 10000.0;
commercialplane.numengines = 4;
privateplane.weight = 3000.0;
privateplane.numengines = 4;
```

// QUIZ ANSWERS

1 Remember that private members are not accessible outside the class, protected members are accessible to any descendants, and public members are accessible to anyone.

a **i** No
ii No

b **i** Yes
ii No

c **i** Yes
ii Yes

2 There is more than one way to organize this. Basically, if there is something common between various items, that can be pulled out into a superclass.

For example, because all 8 classes have member variables **Appraised Value** and **Square Feet**, these can be pulled into a superclass that you might call **Real Estate**; all other classes would be descended from this.

Below that, 4 of the classes have **Rental Price per Square Foot** as a member variable, so you could create a new class, **Commercial**, containing a **Rental Price per Square Foot** member variable that those 4 are then derived from.

Likewise, 4 other classes could be grouped under a **Residential** class, with a member variable **Number of people who can live there**.

Finally, both **Single-Family House** and **Multiplex** have a **Yard Size** member variable, so these could both be grouped under a **Standalone House** class.

Note that it is not necessary to divide things this way, but it is one option.

- **Real Estate** (a base class) - **Appraised Value**, **Square Feet**
- **Commercial**: **Derived from Real Estate - Rental Price per Square Foot**
- **Residential**: **Derived from Real Estate - Number of people who can live there**
- **Shopping Center**: **Derived from Commercial - Length of street frontage**
- **Restaurant**: **Derived from Commercial - Kitchen area**, **Dining area**
- **Office Building**: **Derived from Commercial - Number of exterior entries**
- **Industrial Facility**: **Derived from Commercial - Number of loading docks**
- **Standalone House**: **Derived from Residential - Yard Size**
- **Single-Family House**: **Derived from Standalone House - Number of bedrooms and bathrooms**
- **Multiplex**: **Derived from Standalone House - Number of separate units**
- **Apartment**: **Derived from Residential - Monthly Rent**
- **Condo**: **Derived from Residential - Monthly Maintenance Fee**

3 a To define this, you need to specify that **jet** is inheriting from **airplane** by including **: public airplane** when declaring **jet**. Then, inside the class, you need to declare only the new variable, **numengines**.

```
class jet : public airplane {
  public:
  int numengines;
};
```

b These commands are valid:

```
commercialplane.weight = 10000.0;
commercialplane.numengines = 4;
privateplane.weight = 3000.0;
```

And this one is not:

```
privateplane.numengines = 4;
```

Because **jet** inherits from **airplaine**, a **jet** has both **numengines** and **weight** member variables, so both assignments to **commercialplane** are valid. However, an **airplane** does not have a **numengines** member variable, so because **privateplane** is an **airplane** object, and not a **jet** object, it is invalid to try to assign to that member variable.

Also, note that since all variables were **public**, they are accessible from outside the class. If any of the variables were private or protected, then the corresponding command would not have been allowed.

Click here to go back to the quiz.

20 Object-Oriented Programming with Polymorphism

IN THIS LECTURE:

One of the key principles of object-oriented design is **polymorphism**, which refers to the idea that a class can take on many different shapes. To get at this concept, you need a class hierarchy with inheritance—basically, a superclass that's specialized into multiple subclasses. The idea is that the superclass, the one higher in the class hierarchy, can take on the particular form of any one of its subclasses; that is, the superclass can have many different shapes, each of which is an example of the overall class.

// A CLASS HIERARCHY

The important part of polymorphism is that you can define operations at the highest level of the hierarchy where it makes sense—basically, you define an operation on the top-level superclass as much as possible. Then, the actual implementation of that operation can be defined at any point along the hierarchy. Each of the subclasses could potentially have a different way of performing that operation, but they all perform it. When you refer to some class, then anything that is of that class, including any descendant classes, could meet the requirements.

Inheritance can save coding by inheriting features from a base class rather than having to repeat them in all the derived classes. But probably an even more important feature of inheritance is that it supports the ability to let classes lower in the hierarchy meet the requirements defined higher in the hierarchy.

The idea of a class hierarchy means that when a function is written to use some class, that same function can also use any of its descendent classes. However, things don't work the other way around: You can't require a more specialized class and pass in the more generalized superclass.

In this code, which uses the hierarchy from the previous lecture, you define a base **product** class and derive **cup** and **plate** subclasses from that (a). Then, from **cup**, you derive **measuring_cup** and **drinking_cup** (b). In this case, every member is public. The **product** class has a wholesale price (8)and a retail price (9).

Let's say you wanted to create a function that put items on sale by 10%. You might want a function that gives you the sale price. So, you can write a function called **sale_price** that returns a floating-point number giving the sale price. It will have one parameter, which will be a product. And it will return 90% of the product's retail price (c).

Then, you can create very specific objects in your code. In this case, you have a **plastic_cup** that's an instance of a **measuring_cup** and a **coffee_cup** that's an instance of a **drinking_cup** (d). Because both the **measuring_cup** and the **drinking_cup** are derived from the **product** class, you can call the **sale_price** function on both of them (54, 55) without difficulty!

This is one of the most straightforward ways to use polymorphism. When you inherit from a base class, you automatically have access to everything that base class has. In other words, the descendants from a base class have all the member variables and functions from the base class, so any descendant can be used in place of that base class.

This makes it possible to extend an existing class to provide more functionality. You can add on some additional features that you might want and keep everything about the class that you already liked.

```
1   // Program 20_1
2   // Inheritance Example - using function on base class
3   #include<iostream>
4   using namespace std;
5
6   class product {
7   public:
8       float wholesale_cost;
9       float retail_cost;
10
11      float profit_per_item() {
12          return retail_cost - wholesale_cost;
13      }
14  };
15
16  class plate : public product {
17  public:
18      float diameter;
19  };
20
21  class cup : public product {
22  public:
23      float volume;
24      string color;
25  };
26
27  class measuring_cup : public cup {
28  public:
29      bool metricunits;
30      int num_gradations;
31  };
32
33  class drinking_cup : public cup {
34  public:
35      bool has_lid;
36  };
37
38  float sale_price(product p) {
39      return p.retail_cost * 0.9;
40  }
41
42  int main() {
43      measuring_cup plastic_cup;
44      plastic_cup.wholesale_cost = 0.10;
45      plastic_cup.retail_cost = 0.30;
46      drinking_cup coffee_cup;
47      coffee_cup.wholesale_cost = 0.08;
48      coffee_cup.retail_cost = 0.40;
49
50      cout << "The plastic cup costs " << plastic_cup.wholesale_cost
51           << " to purchase and sells for " << plastic_cup.retail_cost << endl;
52      cout << "The coffee cup costs " << coffee_cup.wholesale_cost
53           << " to purchase and sells for " << coffee_cup.retail_cost << endl;
54      cout << "The sale prices will be " << sale_price(plastic_cup)
55           << " and " << sale_price(coffee_cup) << " respectively." << endl;
56  }
```

a

b

c

d

Here's a situation in which you can extend an existing class. Recall that exceptions are used for getting out of functions when you have errors. There are different types of exceptions: There is a base **exception** class, and then there are a lot of classes that are derived from that. Some are derived directly from **exception**, such as **logic_error**, and some are further derived from there, such as **out_of_range** being derived from **logic_error**.

However, the functionality of **try-catch** statements and exception throwing is designed to work with the general **exception** class. So, all the various specific exceptions can be used in any functions that take an exception as input, because they all inherit from **exception**.

You can even create your own exceptions! Suppose you want to create a new version of a **logic_error**, one that you can use to report when there is a bad value. And say you want to keep track of that value within your exception itself.

In **Program 20_2**, you've defined a new class (e), **bad_value_exception**, that inherits from **logic_error**. The class does a few things. First, it has a private member integer variable, **value**, which will be used to store the bad value that was used (f). It has a constructor that takes in an integer and a string, with the string having a default value of the empty string so that the constructor could be called with just an integer. The constructor uses the string to initialize the **logic_error** base class, and it uses the integer parameter to set the **value** member variable (g). Finally, your new **bad_value_exception** class has a public member function, **printval**, that prints that stored value (h). Basically, this class is now your own special exception that you can throw when you want to and use how you want.

Next, you have a function defined, **something_to_do**, that has one input parameter. For some values of the input parameter, those less than **3**, it will print a message, and for others it will throw your **bad_value_exception** (i).

In the **main** function, you have a **try** block where you call the **something_to_do** function with 3 different parameters (j). The first call will be a good value, so it will just print an output statement normally; no exception is thrown. The second call, though, causes a **bad_value_exception** to be thrown. So, in your **catch** block (k), you have a **catch** defined to take a bad_**value_exception**, and for that you use the **printval** function to print the value.

When this is run, you get one line of output from the **something_to_do** function and another line printed from the **printval** as part of the exception catching.

```
1   // Program 20_2
2   // Defining our own exception
3   #include<iostream>
4   #include<exception>
5   #include<stdexcept>
6   using namespace std;
7
8   class bad_value_exception :
    public logic_error {                    e
9   private:
10     int value;                           f
11
12  public:
13     bad_value_exception(int n,
    string s = "") : logic_error(s) {
14        value = n;
15     }                                    g
16
17     void printval() {
18        cout << "The number "
    << value << " was used when it
    wasn't allowed." << endl;
19     }                                    h
20  };
21
22  void something_to_do(int x) {
23     // Print the value if it's
    less than 3, throw an exception
    otherwise
24     if (x > 3) {
25        throw(bad_value_
    exception(x, "Bad Value"));
26     }
27     else {
28        cout << x << " is a good
    number." << endl;
29     }                                    i
30  }
31
32  int main() {
33     try {
34        something_to_do(1);
35        something_to_do(5);
36        something_to_do(10);
37     }                                    j
38     catch (bad_value_
    exception e) {
39        e.printval();
40     }                                    k
41  }
```

You did not have to make **bad_value_exception** derive from **exception**; you can throw and catch things other than exceptions. But if you modified your code slightly to make it more general—where instead of catching just a **bad_value_exception**, you catch a more general exception and output a general statement—you see that this still works for your **bad_value_exception**.

```
38      catch (bad_value_exception e) {
39          e.printval();
40      }
```

```
38      catch (exception e) {
39          cout << "Encountered an
    exception!" << endl;
40      }
```

In other words, because you made your new class derive from **logic_eror**, which is itself derived from **exception**, you can use it in all the ways that you would have used an exception. Plus, if you ever want the special functionality that **logic_eror** could provide, you have that, too!

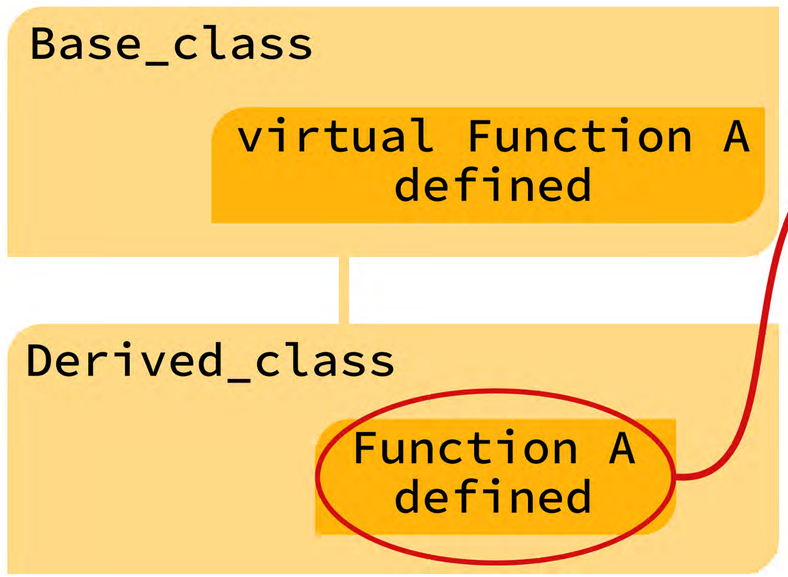
Polymorphism can do even more than just let you use inherited classes. You also have the ability to create member functions of classes that operate differently but can be called the same way.

Here is the general idea of how polymorphism works. A base class declares some function and possibly even provides an implementation—a full definition—for that function. Then, a class that is derived from that base class creates a slightly different version of the same function.

Suppose you have a base class and a derived class. Say that there is a function, **A**, defined in the base class and that the derived class also has a function, **A**, defined.

Then, imagine that you have code that defines a function, **fun**, that takes in a **base_class** as a parameter and that then calls the function **A** on that parameter.

When you call the function **fun** with an instance of **base_class**, then it should call the function **A** that's defined in the base class. On the other hand, if you call **fun** with an instance of **derived_class**, then it should call the function **A** that's defined in the derived class.




```
void fun(Base_class& B) {
    B.A();
}

Base_class x;
Derived_class y;
fun(x);
fun(y);
```

```
1    // Program 20_4
2    // Not Quite Polymorphism Example
3    #include<iostream>
4    #include<string>
5    using namespace std;
6
7    class car {
8    protected:
9       float base_price;
10      string model_name;                    l
11
12   public:
13      car() {
14          base_price = 20000.0;
15          model_name = "Generic1";
16      }
17
18      car(float price, string name) {
19          base_price = price;
20          model_name = name;
21      }                                     n
22
23      void print_info() {
24          cout << "The model " << model_name << " costs " << base_price << endl;
25      }
26
27      float price() {
28          return base_price;
29      }
30   };
31
32   class SC300 : public car {
33   protected:
34      bool performance_package;
35      bool entertainment_package;
36      bool safety_enhancements;             m
37
38   public:
39      SC300(bool p, bool e, bool s) : car(35000.0, "SC300") {
40          performance_package = p;
41          entertainment_package = e;
42          safety_enhancements = s;
43      }                                     o
44
```

...

Imagine that you're a car dealer. You'll have a general class to keep track of cars, and then from that, you'll derive classes that represent the more specific cars.

In this case, you have a base **car** class that contains 2 member variables: **base_price** and **model_name** (l).

You'll also have a class for the **SC300** model car (m), which inherits from the **car** class. That class will have 3 member variables that are Booleans indicating whether the car has various add-on features specific to that car, such as a **performance** or **entertainment** package.

Each of these classes will also have constructors defined. For the **car** class, there will be both a default constructor for a generic model car and a constructor allowing the base price and model name to be set (n).

For the **SC300** class, the constructor will take parameters specifying whether or not the various options were ordered. It will call the **car** constructor with the base price for the **SC300** and the model name for the **SC300** and then will set the appropriate Booleans (o).

```
...
```

```
45      void print_info() {
46          cout << "The model " << model_name;
47          float total_price = base_price;
48          if (performance_package) {
49              cout << ", with the performance package";
50              total_price += 3000.0;
51          }
52          if (entertainment_package) {
53              cout << ", with the entertainment package";
54              total_price += 1200.0;
55          }
56          if (safety_enhancements) {
57              cout << ", with safety enhancements";
58              total_price += 2100.0;
59          }
60          cout << " costs " << total_price << endl;
61      }
62
63      float price() {
64          float total_price = base_price;
65          if (performance_package) {
66              total_price += 3000.0;
67          }
68          if (entertainment_package) {
69              total_price += 1200.0;
70          }
71          if (safety_enhancements) {
72              total_price += 2100.0;
73          }
74          return total_price;
75      }
76  };
77
78  int main() {
79      car x;
80      x.print_info();
81      SC300 y(true, false, true);
82      y.print_info();
83  }
```

p

q

Both the base **car** class and the derived **SC300** class will have 2 functions: one giving the **price** (27, 63) and one that prints information about the vehicle (23, 45). For the **car** class, the price is just the base price, and the only thing to print is the model name and price. For the **SC300**, the price is determined based on which optional packages are included, and the output will be more complicated.

If you create a generic car, using the default constructor for the **car** class, and call the **print_info** function for that object, it will call the version of **print_info** defined in the **car** class (p). If you instead create an **SC300** car and call **print_info** for it, you will be calling the **print_info** function defined in the **SC300** class—not the one defined in the **car** class (q).

This is still not "full" polymorphism. All you are doing here is letting subclasses do some redefining of functions; a complete version of polymorphism comes from the superclass, where a single function from the superclass is created that takes on different forms in the subclasses.

// VIRTUAL FUNCTIONS

Virtual functions get defined differently—that is, take on a different shape—in all the child classes.

This is a modified version that does, in fact, demonstrate polymorphism. Notice that this code is the same as the previous code, except for a few changes:

- In the base class, **car**, you now have the word *virtual* in front of your 2 member functions that you want to override in the base class (r). The **virtual** is needed to say that "this function will be a polymorphic function."
- You have defined a new function, called **print_car**. This function takes in a **car** parameter—as a reference—and calls **print_info** for that car (s).
- The **main** code is also modified so that you are calling the **print_car** function, passing in a **car** and an **SC300** rather than calling the member functions for each (27, 63).

When you run this code, you indeed see that the specific versions of **print_info** are called. In other words, when you call **print_car** with a **car** object, it will end up calling the **print_info** command defined in the **car** class. And when you call **print_car** with an **SC300** object, it will end up calling the **print_info** command defined in the **SC300** class. This is exactly what you want from polymorphism: the ability to define a different version of a function for different classes. This way, when you call **print_car**, you get the most descriptive output you can.

```
1   // Program 20_5
2   // Polymorphism Example
3   #include<iostream>
4   #include<string>
5   using namespace std;
6
7   class car {
8   protected:
9       float base_price;
10      string model_name;
11
12  public:
13      car() {
14          base_price = 20000.0;
15          model_name = "Generic1";
16      }
17
18      car(float price, string name) {
19          base_price = price;
20          model_name = name;
21      }
22
23      virtual void print_info() {
24          cout << "The model " << model_
    name << " costs " << base_price << endl;
25      }
26
27      virtual float price() {
28          return base_price;
29      }
30  };
31
32  class SC300 : public car {
33  protected:
34      bool performance_package;
35      bool entertainment_package;
36      bool safety_enhancements;
37
38  public:
39      SC300(bool p, bool e, bool s) :
    car(35000.0, "SC300") {
40          performance_package = p;
41          entertainment_package = e;
42          safety_enhancements = s;
43      }
44
45      void print_info() {
46          cout << "The model " <<
    model_name;
47          float total_price = base_price;
48          if (performance_package) {
49              cout << ", with the
    performance package";
50              total_price += 3000.0;
51          }
52          if (entertainment_package) {
53              cout << ", with the
    entertainment package";
54              total_price += 1200.0;
55          }
56          if (safety_enhancements) {
57              cout << ", with safety
    enhancements";
58              total_price += 2100.0;
59          }
60          cout << " costs " << total_
    price << endl;
61      }
62
63      float price() {
64          float total_price = base_price;
65          if (performance_package) {
66              total_price += 3000.0;
67          }
68          if (entertainment_package) {
69              total_price += 1200.0;
70          }
71          if (safety_enhancements) {
72              total_price += 2100.0;
73          }
74          return total_price;
75      }
76  };
77
78  void print_car(car& c) {
79      c.print_info();
80  }
81
82  int main() {
83      car x;
84      print_car(x);
85      SC300 y(true, false, true);
86      print_car(y);
87  }
```

r

s

If you remove the **virtual** from in front of the declarations in the base class, then when you run the code, the output when you pass in the **SC300** object is the **print_info** result for the **car** class—not for the **SC300** class! You know this because you're not seeing the printout of the various options, and the price is only the base price, not the price with options.

You no longer have a polymorphic function. Instead, the calling function, **print_car**, says "I have a **car** object, so I will call the **car** object's **print_info** function." Without the **virtual** statement, it won't look to see if this function has been overridden by a subclass.

It's a different situation than previously in **Program 20_4**, when you were calling **print_info** directly on an object **(q)**. There, you knew that you were calling the function from an **SC300** object. So, you'd first look at the **SC300** definition to see if **print_info** is defined there: If it is, like it was in **Program 20_4**, you would have used that, and only if it was not defined would you look at the parent to see if it had **print_info**, and so on.

In this case, though, you are in a function where you think you have a **car** object **(t)**. So, the compiler is going to look at the **car** class and find a **print_info** function defined and use that. It isn't going to look further to see if that function has been overridden because it was not declared a virtual function.

Note that subclasses don't have to redefine the function if it's virtual; they just have the option of redefining it if they want to.

Let's look at one other variation, with the **virtual** back on these functions so that they're indeed polymorphic and so that the subclasses can redefine the function if they so choose.

But now you'll change the **print_car** function to take in a parameter by value, instead of by reference; that is, instead of the parameter being **car& c**, you just have **car c**.

```
23      virtual void print_info() {
24          cout << "The model " <<
    model_name << " costs " << base_
    price << endl;
25      }
26
27      virtual float price() {
28          return base_price;
29      }
...     }
78  void print_car(car& c) {
79      c.print_info();
80  }
```

t

If there is a function that you want to allow to be polymorphic—that is, that you want to allow a subclass to override and provide a more detailed description of—then you should always make sure it is declared to be a virtual function.

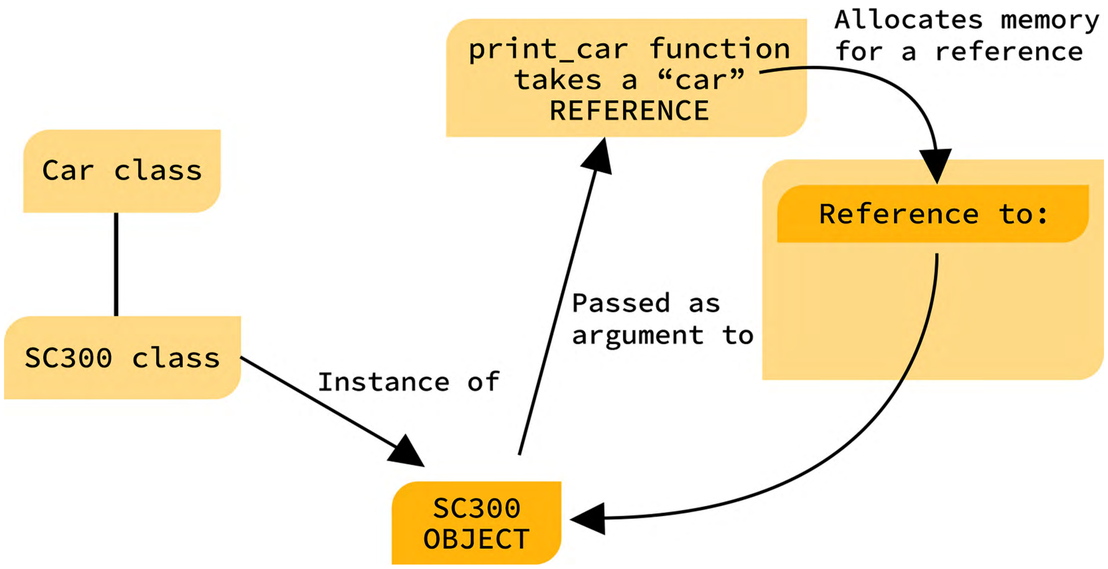
```
23      virtual void print_info() {
24          cout << "The model " <<
    model_name << " costs " << base_
    price << endl;
25      }
26
27      virtual float price() {
28          return base_price;
29      }
...     }
78  void print_car(car c) {
79      c.print_info();
80  }
```

u

Again, the output shows that the **print_info** function that is called is the one for the **car** class, not the one for the **SC300** class. But it looked like you declared this as virtual. What happened?

In the preceding case, you create an **SC300** object in your **main** code and pass this to the **print_car** function using a pass by reference. When this function is called, the memory for the parameter has a reference stored. This is basically a pointer, and it has a reference to the object that was passed in. No data is copied. Because the **SC300** class is derived from the **car** class, it is OK to pass in the **SC300** as a reference.

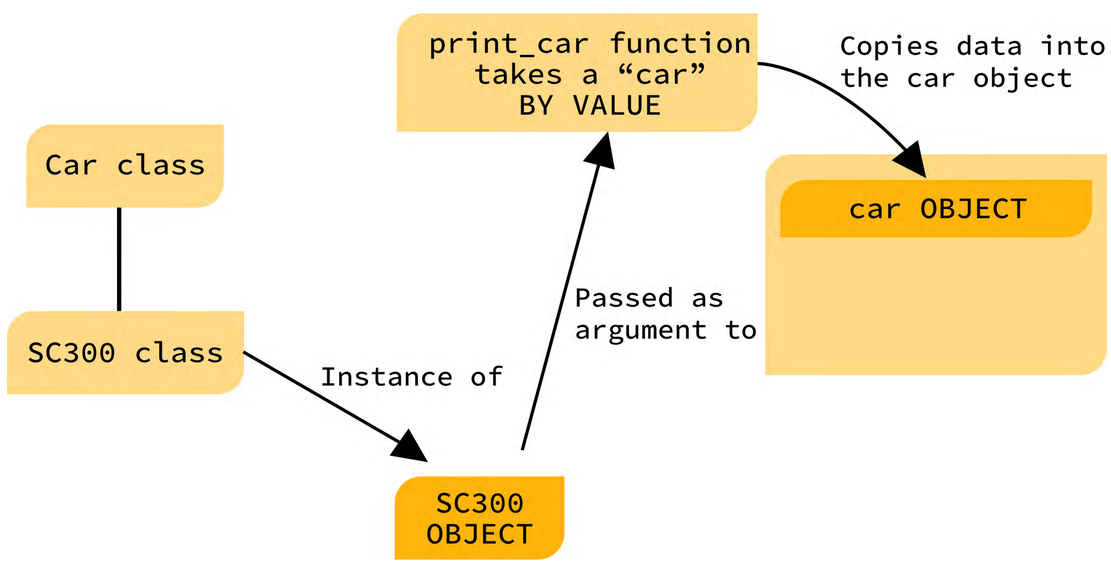
So, when you call **print_info** within the **print_car** function, you go to that reference and call **print_info** on it. Because the thing it is referring to is the **SC300** object that was created before, it will try to call the **print_info** command for the **SC300** class. Because **print_info** was declared to be virtual, it will look for the most specific version of that function, which is the one defined in the **SC300** class.



On the other hand, when you pass by value, you still have an **SC300** defined in the **main** program that you call **print_car** on, but now it is a parameter that is passed by value.

When the function is called, there is space set aside in memory for a **car** object. Then, the values from the argument are copied into the **car** object that has been allocated. In this case, that's the **base_price** and the **model_name**, which were defined in the **car** class, but not the various Booleans that indicated which packages were selected.

Thus, the thing that's sitting there in memory is a **car**—not an **SC300**. It has no memory of having originally come from an **SC300**; all it knows is that it's a **car**. As a result, when you call **print_info** on that object, it's going to use the **print_info** command defined for the **car** class, not the one defined for the **SC300** class.



// PURE VIRTUAL FUNCTIONS

It's possible for the base class to declare a virtual function yet provide no definition. This is known as a **pure virtual function**, and it has some major implications for the class itself.

In this code, you are providing classes that can be used to calculate shipping costs for a package. To keep things simple, the class will have no member variables and just one member function, **shipping_cost**.

You have one base class, **shipping_company** (6), which will be a class that the other classes inherit from. In the **shipping_company** class, you declare one member function, **shipping_cost** (8). Notice that the **shipping_cost** function is declared virtual, as expected, and it has a return value, **float**, and takes in a single **float** parameter. Notice that the parameter does not have to have a name declared, because you are not actually defining the function here. Instead, this is going to be a pure virtual function, and to designate that, instead of providing the function definition—that is, instead of the body enclosed in curly braces—you write **=0;**.

A pure virtual function will have to be defined in the derived classes. In this case, you have 3 different specific shipping companies. Each of these will define its own version of the **shipping_cost** function.

```
1   // Program 20_8
2   // Pure virtual function
3   #include<iostream>
4   using namespace std;
5
6   class shipping_company {
7   public:
8       virtual float shipping_cost(float) = 0;
9   };
10
11  class VQT : public shipping_company {
12  public:
13      float shipping_cost(float weight) {
14          return 3.0 + 1.5*weight;
15      }
16  };
17
18  class NatFast : public shipping_company {
19  public:
20      float shipping_cost(float weight) {
21          return 2.0*weight;
22      }
23  };
24
25  class GovernmentPost : public shipping_
    company {
26  public:
27      float shipping_cost(float weight) {
28          if (weight < 2.0) {
29              return 4.0;
30          }
31          else if (weight < 4.0) {
32              return 8.0;
33          }
34          else if (weight < 6.0) {
35              return 11.0;
36          }
37          else {
38              return 2.0*weight;
39          }
40      }
41  };
42
43  shipping_company& select_company() {
44      int option;
45      cout << "Which shipping company do you
    use? " <<
46          "Enter 1 for VQT, 2 for NatFast, 3
    for Government Post: ";
47      cin >> option;
48      if (option == 1) {
49          return *new VQT();
50      }
51      else if (option == 2) {
52          return *new NatFast();
53      }
54      else {
55          return *new GovernmentPost();
56      }
57  }
58
59  int main() {
60      float w;
61      cout << "What is the package weight? ";
62      cin >> w;
63      shipping_company& x = select_company();
64
65      cout << "Your package will cost " <<
    x.shipping_cost(w) << " to ship." << endl;
66  }
```

The key thing to realize is that the base class, **shipping_company**, declared a pure virtual function with no implementation. Only in the derived classes were actual definitions for that function given.

When you have pure virtual functions, it's important to realize that you can't have an instance of just the base class. In the example, it doesn't make any sense to have just a shipping company; there's no way to know what the shipping cost would be for some generic shipping company. Instead, you need to have a specific shipping company.

A class that has pure virtual functions is an **abstract class**. It's abstract in the sense that it defines a general idea, like **shipping_company**, that can't refer to any specific instance of the class, but rather just defines some more general, abstract representation.

You can't create an object out of an abstract class, though you can have references to an abstract class. Classes that do provide the implementation details—the lower-level derived classes that you can create objects from—are sometimes called **concrete classes**.

So, notice that if you take the same prior definitions and change the **main** routine to try to create an instance of a **shipping_company**, you end up with an error telling you that you can't declare an abstract type. ◆

```
// Error when trying to instantiate an abstract class
...
int main() {
  shipping_company s;
}
```

Exercise

Create an abstract **shape** class that has an **area** function. Then, create 2 specific, concrete classes—**square** and **circle**—that inherit from the general **shape** class.

Click here to see the solution.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, sections 14.2–14.3.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 15.3–15.4.

Exercise Solution

Here's one way that this could be implemented.

```
1   // Program 20_9
2   // Abstract shape class with pure virtual area function
3   #include<iostream>
4   using namespace std;
5
6   class shape {
7   public:
8       virtual float area() = 0;
9   };
10
11  class square : public shape {
12  private:
13      float side_length;
14
15  public:
16      square(float s) {
17          side_length = s;
18      }
19
20      float area() {
21          return side_length * side_length;
22      }
23  };
24
25  class circle : public shape {
26  private:
27      float radius;
28
29  public:
30      circle(float r) {
31          radius = r;
32      }
33
34      float area() {
35          return radius * radius*3.14159;
36      }
37  };
38
39  int main() {
40      square s(3.0);
41      circle c(3.0);
42      cout << "Areas are: " << s.area() << " and " << c.area() << endl;
43  }
```

Click here to go back to the exercise.

// QUIZ

1 Are the following true or false regarding a pure virtual function?

a You can provide a default definition for the pure virtual function to use in case it is not defined in a subclass.

b It is defined using a line where the virtual function header is provided and then `= 0;` is added.

c The class where a pure virtual function is declared is an abstract class.

d Only one subclass can instantiate the pure virtual function.

2 How would you create a new type of exception, called an **`insufficient_storage`** exception, that stored (in a public member variable) an integer **`amount_short`**?

3 What would be the output of the following code? (Be careful to look at how each function is defined and exactly what is called in each function call.)

```
#include<iostream>
#include<string>
using namespace std;

class airplane {
    public:
    float weight;
    string name;

    void print_name() {
        cout << "For the airplane " << name << ", ";
    }

    virtual void print_info() {
        cout << "weight is: " << weight << endl;
    }
};

class jet : public airplane {
    public:
    int numengines;

    void print_name() {
        cout << "For the jet " << name << ", ";
    }

    void print_info() {
        cout << "weight is: " << weight << " and there are ";
        cout << numengines << " engines"<< endl;
    }
};

void print_plane_data(airplane a) {
    a.print_name();
    a.print_info();
}

void print_plane_data2(airplane& a) {
    a.print_name();
    a.print_info();
}

int main()
{
    airplane privateplane;
    jet commercialplane;
    privateplane.name = "A123";
    privateplane.weight = 3000.0;
    commercialplane.name = "B456";
    commercialplane.weight = 10000.0;
    commercialplane.numengines = 4;
    privateplane.print_name();
    privateplane.print_info();
    commercialplane.print_name();
    commercialplane.print_info();
    print_plane_data(commercialplane);
    print_plane_data2(commercialplane);
}
```

Click here to see the answers.

// QUIZ ANSWERS

1 Remember that a pure virtual function is declared in an abstract class and must be instantiated in a subclass.

a False. You cannot provide a default version of the pure virtual function; if this were the case, it would not be a pure virtual function, but just a regular virtual function.

b True. You define a pure virtual function by adding `=0;` to the header. For example,

```
virtual bool is_valid() = 0;
```

declares a pure virtual function named **is_valid** that takes no parameters and returns a Boolean.

c True. When a class has a pure virtual function, it cannot be instantiated; you cannot have an object of that class because the function is not defined. This is called an abstract class.

d False. The pure virtual function can be instantiated in any and all subclasses. Every subclass must have some definition (inherited or defined in that class) for it to be instantiated.

2 To create such an exception, you would need to have #included **<exception>** and **<stdexcept>**. Then, the class definition just inherits from **exception**:

```
    class insufficient_storage :
public exception {
      public:
      int amount_short;
    };
```

Then, **insufficient_storage** could be used wherever an exception was used.

3 Here is the output:

```
    For the airplane A123, weight
is: 3000
    For the jet B456, weight is:
10000 and there are 4 engines
    For the airplane B456, weight
is: 10000
    For the airplane B456, weight
is: 10000 and there are 4 engines
```

Notice several things about how the classes are arranged.

- The **airplane** class defines **print_name** as a non-virtual function. The **jet** class then also defines **print_name**, with slightly different output. This means that only **jet** objects or references will look at the **jet** version of the function and that any time there is an **airplane**, including a reference to an **airplane**, the **airplane** version will be used.
- So, when **commercialplane** calls **print_name** directly, then the **jet** version is used. But when **commercialplane** is passed to either **print_plane_data** or **print_plane_data2**, it is treated as an **airplane**, not a **jet**, so the **airplane** version is used.
- The **print_plane_data** function takes an **airplane** parameter by value. Thus, the local parameter, **a**, is always an airplane, regardless of what subclass might have been passed in. There is a new **airplane** object created in memory, which is set based on the argument to the function. Thus, only the **airplane** versions of functions (both **print_name** and **print_info**) are called.
- The **print_plane_data2** function, however, takes an **airplane** parameter by reference. Thus, when you pass in **commercialplane**, there is nothing copied; instead, you have a reference to the full **jet** object. Although the **print_name** (which was not virtual) function still uses the **airplane** version, the **print_info** function is virtual, so the version of whichever **print_info** is defined for that object is used. In this case, that is the **jet** version.

Click here to go back to the quiz.

21 Using Classes to Build a Game Engine in C++

What you've been learning about object-oriented program is all going to come together now as you are walked through a development process for designing a game engine that can be used for more than one game through the use of classes.

IN THIS LECTURE:

// DESIGNING CLASSES

You want to create a system that lets you play 2-person board games on a computer. In lecture 15, you developed a Connect 4 game; in this lecture, you want to build on your top-down design but use object-oriented principles. In fact, you'll do this in a class that can handle all kinds of 2-person games—not just Connect 4, but checkers, chess, or Othello.

For this implementation, the focus will be on 2 different games: Connect 4 and Othello, known more generally as Reversi.

As is the case with any approach to software development, the first thing you should do when faced with a programming problem is stop and think.

In Reversi, players take turns placing either black or white pieces on an 8-by-8 board. The game starts with 2 white and 2 black pieces already placed in the middle 4 squares.

If a black piece is placed, it has to be placed next to some white piece—beside, above, below, or diagonally. If the new black piece causes any white pieces to be newly bracketed by a pair of black pieces—in a row, column, or diagonal—then that causes all those newly bracketed pieces to switch color to black.

The winner is the one with the most of his or her color piece on the board after the board is filled up.

For object-oriented design, your first thought should be to consider what the objects of interest are. From there, you can figure out what classes you need for those objects.

In a very general sense, you'll probably need

- some sort of class for a game in general to keep track of whose turn it is, and so on;
- some gameboard class to hold the current state of the board; and
- some way to describe what a move will be, where the one thing all the games will have in common is 2 players taking turns.

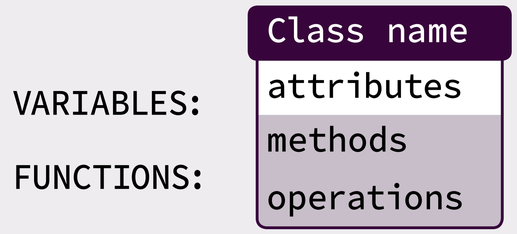
Those are all abstract ideas, so they're likely to end up as abstract classes. You'll also need some more concrete versions of those same classes: the specific gameboards for Connect 4 and for Reversi and the specific moves for Connect 4 and Reversi. These sound like subclasses.

Once you've thought about classes, you can start to look for hierarchies. In this case, there are a few obvious relationships. The abstract **gameboard** should be a parent to the concrete Connect 4 gameboard and Reversi gameboard. Likewise, the abstract **gamemove** should be a parent to a Connect 4 move and a Reversi move.

Once you have a general idea of the hierarchy, you need to think about exactly the member variables and member functions each class should have, as well as which should be public, protected, and private. For your abstract classes, you'll need to think about virtual functions.

To keep track of what you're doing, you can use **Unified Modeling Language** (UML), which is a notation system developed by object-oriented programmers for creating blueprints in the software industry.

In UML, when you describe a class, you use a rectangle with the class name in a box at the top, followed by 2 other boxes: one giving the member variables, or attributes, and the box at the bottom giving the member functions, also called methods or operations.



There are commonly used abbreviations for designating whether elements of a class are public (+), private (–), or protected (#).

Italics are used for classes that are abstract and for virtual functions.

Start at the top and think about the **game** class. Overall, a **game** needs to have a **gameboard**, which is another class, to keep track of the current state of the board, called **board**.

The **gameboard** is not something that you want things outside the game messing with directly; you want to make sure that only valid moves are made, that players don't make 2 moves in a row, and so on. Because you want to moderate any actions that a program running a game can take, this should be a private variable of the **game** class—meaning that only the **game** class can access it directly, and others (whether a player or a computer) can only access it through functions the **game** class provides.

A **gameboard** is going to be an abstract class, which means you'll have to have concrete **gameboard** classes, such as a Connect 4 gameboard or a Reversi gameboard.

To store an abstract **gameboard** in the **game** class, you'll use a pointer to a **gameboard**.

Remember that it's OK to have a pointer to the parent type, even if the actual object is one of the child types.

In UML, you'll write **- board : gameboard*** in the box designated for variables. The **-** notes that it's a private member variable. The **board** is the name of the variable, and the **gameboard*** shows that the variable is going to be a pointer to a **gameboard**.

Likewise, you'll need to be able to take moves in the game. Like **gameboard**, a **gamemove** is going to be an abstract class. So, you will have a **gamemove***, a pointer that is named **turn** for keeping track of moves.

You'll also need to keep track of which player moves next. So, you'll have a private member variable named **playerturn** that's an integer. Perhaps you also want to keep track of how many turns have occurred in the game. A more complex **game** class might keep other information.

Notice that for each UML entry, you'll have a corresponding line of code. The UML is not code itself, though, and the order of terms is different. For example:

- **- board : gameboard*** in UML becomes **gameboard* board;** in code

- **- turn : gamemove*** in UML becomes **gamemove* turn;** in code

In UML, you were starting with what's obvious for humans: **board**, **turn**, **playerturn**, **numturn**, etc. But in C++, you are starting with the type—the information that's more important for computers.

You also need to consider what functions a **game** class needs to provide.

You can use your UML diagram to organize this. The functions you want the class to provide will go in the last box. For each of them, you'll think about the function, what name you want it to have, and then what parameter types it takes in and returns. Finally, you'll consider whether you want the function to be public, private, or protected.

You'll need a constructor, and it will take in 2 parameters: one for the **gameboard** and one for the **gamemove**. And constructors need to be public. So, in your UML diagram, you'll write **+**, indicating that it's public; then **game**, because it's a constructor; and then 2 parameters in the parentheses: **board**, which is of type **gameboard**, and **turn**, which is of type **gamemmove**.

The function to check for a winner won't take any parameters, and it'll return an integer, giving an indication of which player, if either, won. This is a function that you want to call from the **game** class itself, rather than from outside the class, so you'll make it a protected function. Using UML notation, you'd write **# check_winner () : int**.

You will likely also want other functions—for taking a turn or declaring a winner—that will also be protected. These would all be functions that take no parameters and don't return anything, so you'd write **# take_turn ()** and **# declare_winner ()**.

Finally, you'll want a public function that actually starts the game and keeps it going until the game's over. You'll call that **start_game** and thus write **+ start_game ()** for it.

Game

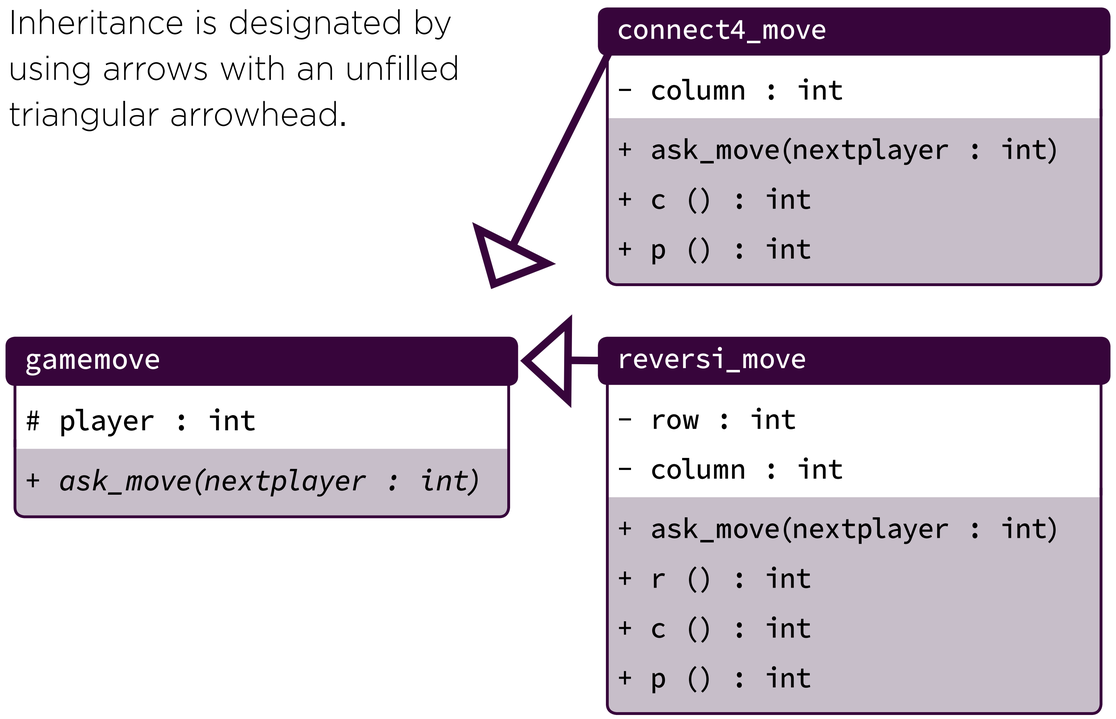
```
- board : gameboard*
- turn : gamemove*
- playerturn : int
- numturns : int
+ game(board : gameboard, turn : gamemove)
# check_winner () : int
# take_turn ()
# declare_winner ()
+ start_game ()
```

Let's look at another class, the **gamemove** class, which will be an abstract class. A move is associated with a player, so your **gamemove** class will have a member variable, **player**. You'll make it a protected variable, so all classes that inherit from it will have access to this variable. It'll also have a pure virtual function, **ask_move**, that will ask the player for a move. It'll take in one integer parameter: the player whose turn it is to move. Because this is a virtual function, the UML convention is to write it in italics.

You also have your 2 derived classes, **connect4_move** and **reversi_move**. These inherit from the **gamemove** class. Inheritance is designated by using arrows with an unfilled triangular arrowhead.

Each specific move is a little different. A **connect4_move** will need to store a column—the column that a piece will be dropped into. A **reversi_move** will need to store both a row and a column.

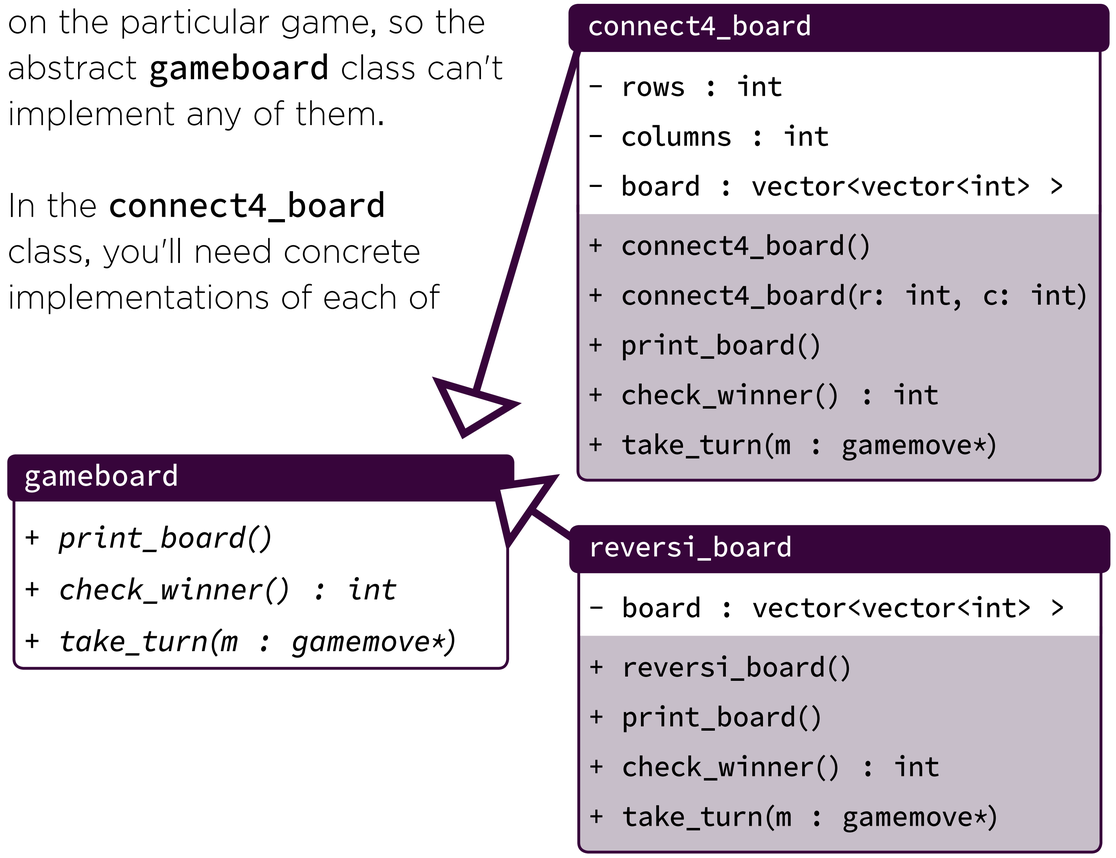
Each of the derived classes will also need to provide a concrete implementation of the **ask_move** function. Plus, you'll need accessor functions for a move—functions to set a move or that will let some other class find out what the position of the move is and which player made it. You'll use **r** for **row**, **c** for **column**, and **p** for **player** to provide this information.



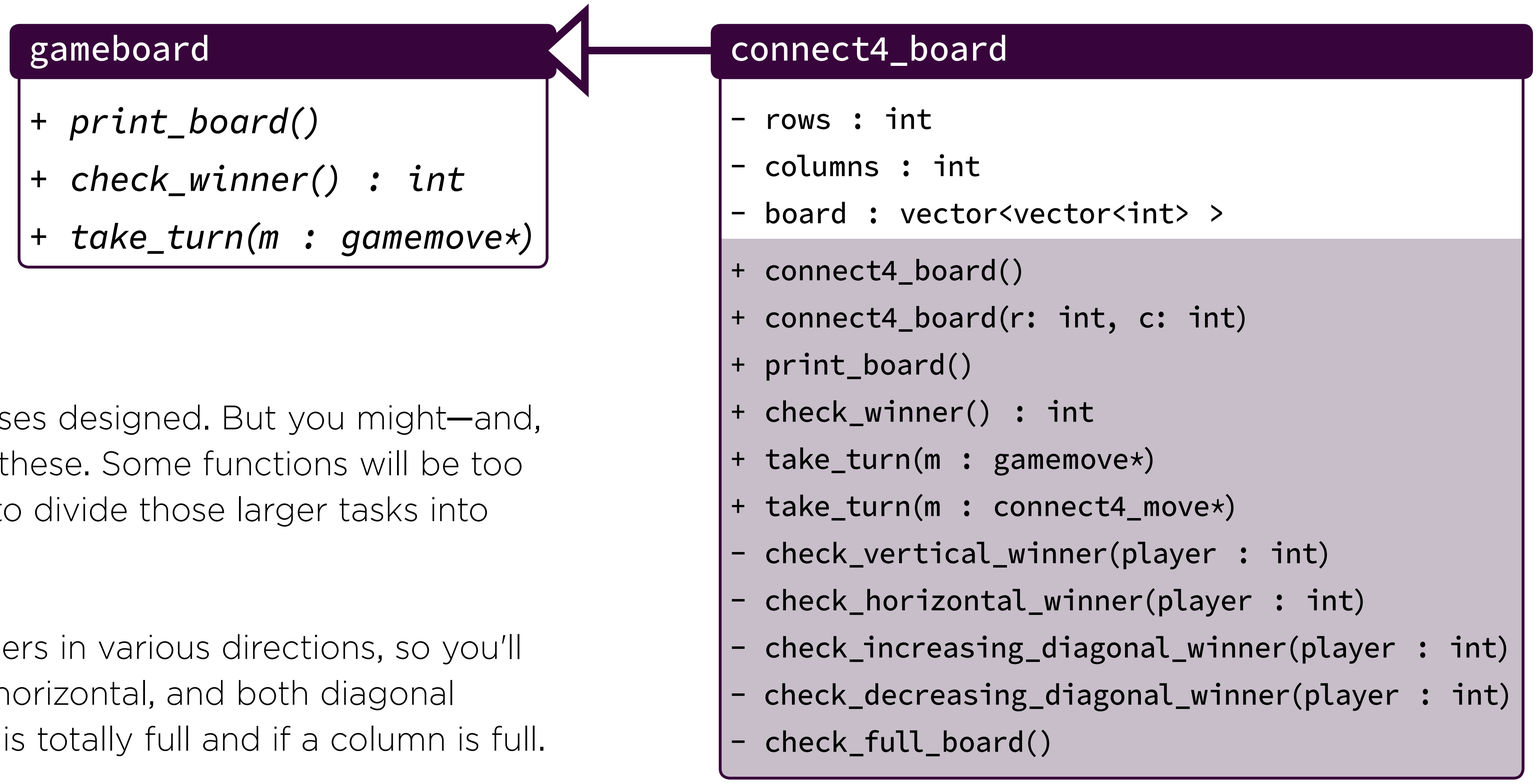
Likewise, you need to define the classes for the **gameboard**, as well as its derived classes, **connect4_board** and **reversi_board**. The **gameboard** will be an abstract class, and it needs to have a few public functions that will be pure virtual functions. It should be able to print a gameboard, so you'll declare a **print_board** function. It should be able to check a board for a winner, returning an integer indicating who won or if it's a tie. And it should be able to make a move if it receives a move of the appropriate type. All of these functions will vary depending on the particular game, so the abstract **gameboard** class can't implement any of them.

In the **connect4_board** class, you'll need concrete implementations of each of the virtual functions that were declared in the **gameboard** class. You'll also need to store information about the board itself. You'll want to store the number of rows and columns in a gameboard, because it can potentially vary. You'll use a vector of vectors to store the information at each point on the board.

Finally, you'll also have a few constructors: one for the default-size Connect 4 board and one for a board where you specify the number of rows and columns specifically.



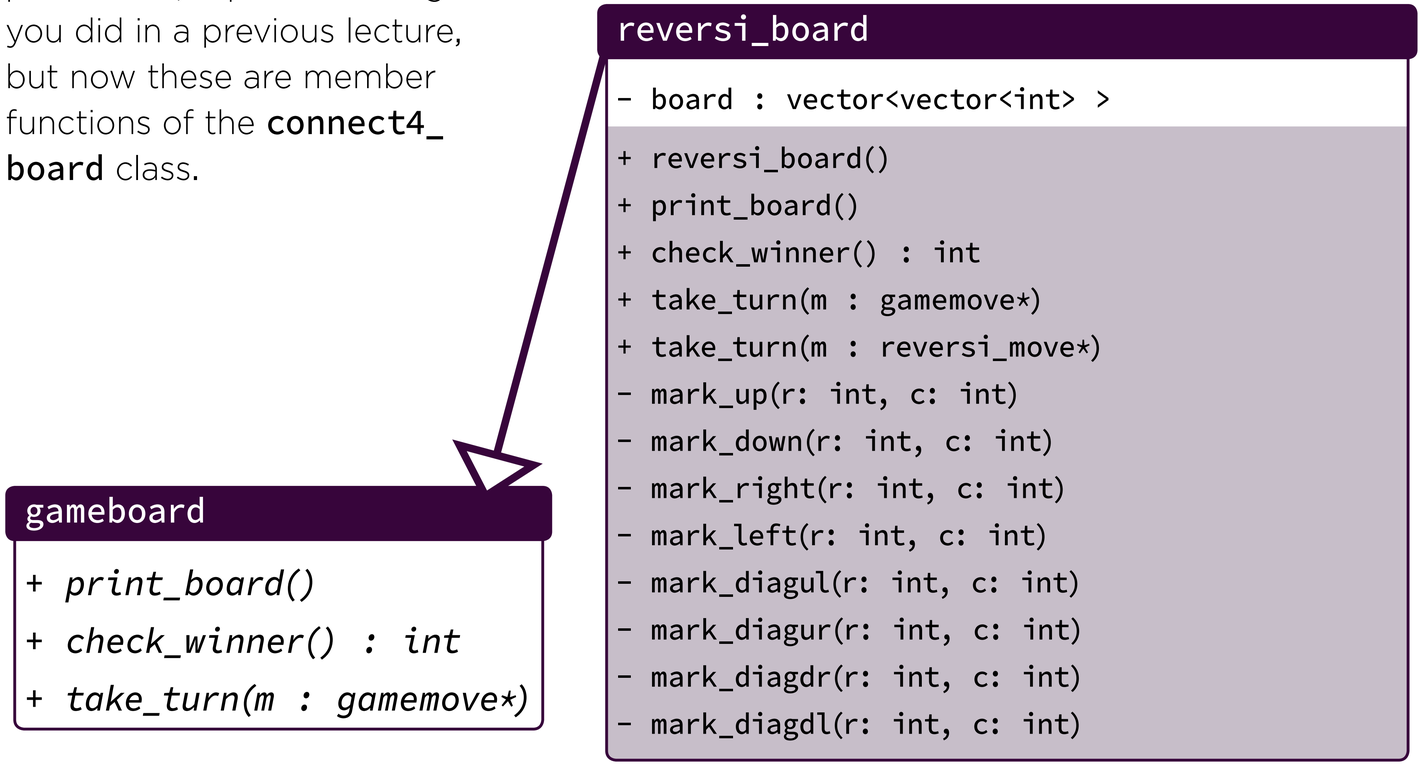
In the **reversi_board** class, just the one default constructor is all you need, because the board will always be 8 by 8 in size. Again, you need to store the board information with a vector of vectors. And you need concrete implementations of the virtual functions declared in the parent class.



At this point, you basically have your entire set of classes designed. But you might—and, in fact, will—end up wanting more functions than just these. Some functions will be too large to do everything in one function, so you'll want to divide those larger tasks into smaller ones using top-down design principles.

For the **connect4_board**, you need to check for winners in various directions, so you'll include functions to check for winners in the vertical, horizontal, and both diagonal directions. You'll also have a check to see if the board is totally full and if a column is full.

All of this is similar to the procedural, top-down design you did in a previous lecture, but now these are member functions of the **connect4_board** class.



Likewise, for a **reversi_board**, you're going to want some additional functions to help you change the board. In particular, when you place a piece, you might need to flip pieces in any of 8 directions: up, down, left, right, and along 4 diagonals. So, you'll create 8 additional functions to help mark the board in those 8 directions after placing a piece.

Realistically, when you're designing a set of classes, you might not know every function that you'll need right at the beginning. That's OK. You might need to change your class design as you build, but hopefully you'll have the major parts of the design in place.

// CODING YOUR DESIGN

With your design finished, you now want to start coding everything.

A good place to begin is with the implementation of your **game** class, because it's the most fundamental of the classes.

Start by comparing the class design that you came up with in UML with what those same steps look like in code. The member variables would be declared as a **gameboard** pointer, **board**; a **gamemove** pointer, **turn**; an integer, **playerturn**; and an integer, **numturns**.

```
private:
gameboard* board;
gamemove* turn;
int playerturn;
int numturns;
```

The next thing in your design were the member functions of the class, starting with your constructor, which is going to take as input a **gameboard** and a **gamemove**. These 2 classes are abstract and do not have a specific value, so they have to be passed in by reference. You write **gameboard& b** to get a reference to the **gameboard** and **gamemove& m** to get a reference to the **gamemove**. Inside the constructor, you just get a reference to the **gameboard**—in other words, a pointer to the **gameboard**—and assign it to the **board** member variable. You do the same for the **gamemove**, assigning it to the **turn** member variable.

```
game(gameboard& b, gamemove& m) {
  board = &b;
  turn = &m;
}
```

The full code is several hundred lines, so only some key specific parts of the program are highlighted here.

The full program is available for download at www.TheGreatCourses.com/CPlusPlus.

The next function in your design was a **check_winner** function, which was protected. You're implementing **check_winner** within the **gameboard** class, because the gameboard itself determines who the winner is. So, your code just calls the **check_winner** function in the **gameboard** you have stored.

```
int check_winner() {
  return board->check_winner();
}
```

Your next routine to look at is the **take_turn** routine, which should take a turn in the game. This means that you want to first have the board printed out to the screen, so you call the **print_board** member function of your **board**. You then want to get the move that the next player wants. You have a member variable, **playerturn**, that should say whose turn it is, so you pass this as an argument to the **ask_move** function of your **turn** member variable. That will call **ask_move** in whatever **gamemove** the **turn** variable is pointing to. Finally, you'll need to update the board based on that move, so you'll call **take_turn** on the board, passing in the move you just got to the **gameboard**.

```
void take_turn() {
  board->print_board();
  turn->ask_move(playerturn);
  board->take_turn(turn);
}
```

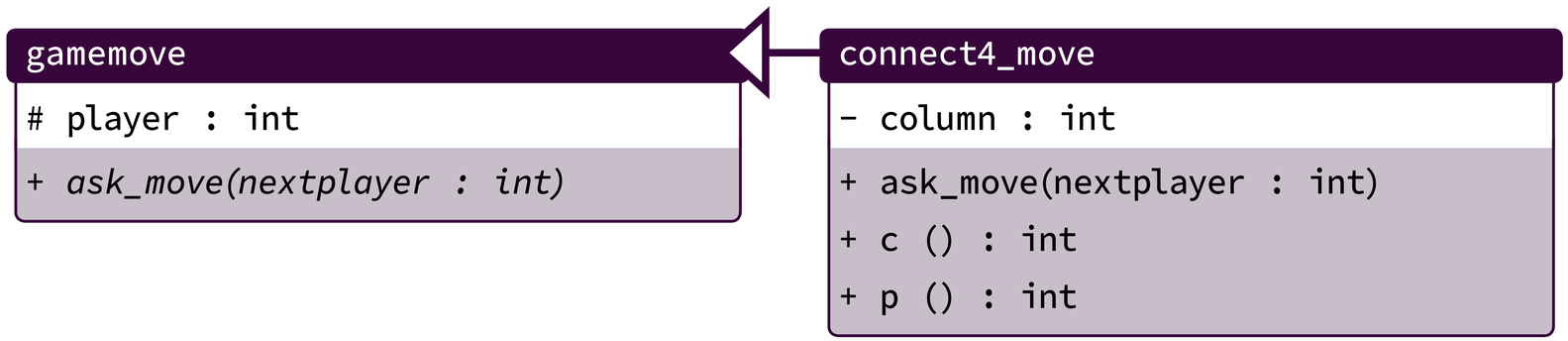
Your **declare_winner** function is several lines of code, but it's really just checking the various possible endgame situations. You determine winners based on the board status, so you just call **check_winner** on the board itself to find out who the winner is and print out the final board and a message about the winner.

```
  void declare_winner() {
    int result = check_winner();
    board->print_board();
    cout << "The game ended after "
<< numturns << " turns." << endl;
    if (result == 1) {
      cout << "Player 1 wins!
Congratulations!" << endl;
    } else if (result == 2) {
      cout << "Player 2 wins!
Congratulations!" << endl;
    } else if (result == 3) {
      cout << "The game has ended in
a tie." << endl;
    } else if (result == 0) {
      cout << "The game ended before
it was completed." << endl;
    }
  }
```

Your final function is **start_game**. This public function is probably the main one inside the **game** class. It's the function that actually makes the game run. In this function, you initialize your member variables to **0**. You then enter a loop that will continue as long as the **check_winner** function in the **gameboard** class reports that the game is not over yet. Within the loop, you first take a turn, calling **take_turn** that you just defined. After that turn is taken, you switch which player's turn it is and increment the number of turns. When the loop finally finishes—that is, when the **check_winner** function says that the game is over—then you call your **declare_winner** routine to print the result.

```
void start_game() {
  numturns = 0;
  playerturn = 1;
  while (check_winner() == 0) {
    take_turn();
    if (playerturn == 1) {
      playerturn = 2;
    } else {
      playerturn = 1;
    }
    numturns++;
  }
  declare_winner();
}
```

After the **game** class, the next thing you defined is the abstract base class **gamemove** and its concrete derived classes, **connect4_move** and **reversi_move**.



As the UML shows, your base class has just one protected member variable, the integer **player**, and one pure virtual function, **ask_move**.

The **ask_move** function takes in an integer parameter and doesn't return anything; therefore, it has a **void** return type. And because it's a pure virtual function, you write **=0** at the end.

```
class gamemove {
  protected:
  int player;
  public:
  virtual void ask_move(int) = 0;
};
```

Notice that the **connect4_move** class is a derived class from **gamemove**. There is one private member variable: an integer giving the column into which you are supposed to place a piece.

The **connect4_move** provides a definition for the **ask_move** function, which was a virtual member of **gamemove**. This takes in the player number as a parameter and uses that to set the protected member variable, **player**, that was inherited from the **gamemove** class.

It reads in the column number from the user, storing it in the **column** variable and adjusting to start numbering at 0 instead of 1.

Finally, you have the accessor functions, **c** and **p**, that return the column for the move and the player who made the move.

```
class connect4_move : public
gamemove {
  private:
  int column;

  public:
  void ask_move(int nextplayer) {
    player = nextplayer;
    cout << "Player " << nextplayer
<< ", which column do you want to
place your piece in? ";
    cin >> column;
    column--; // Adjust to make
first column be 0
  }
  int c() {
    return column;
  }
  int p() {
    return player;
  }
};
```

Implementing the **reversi_move** class is a similar process. There are some small differences here, highlighting the unique differences between the 2 games: The Reversi move needs both a row and a column, so there are 2 member variables stored and 2 pieces of information asked for from the user.

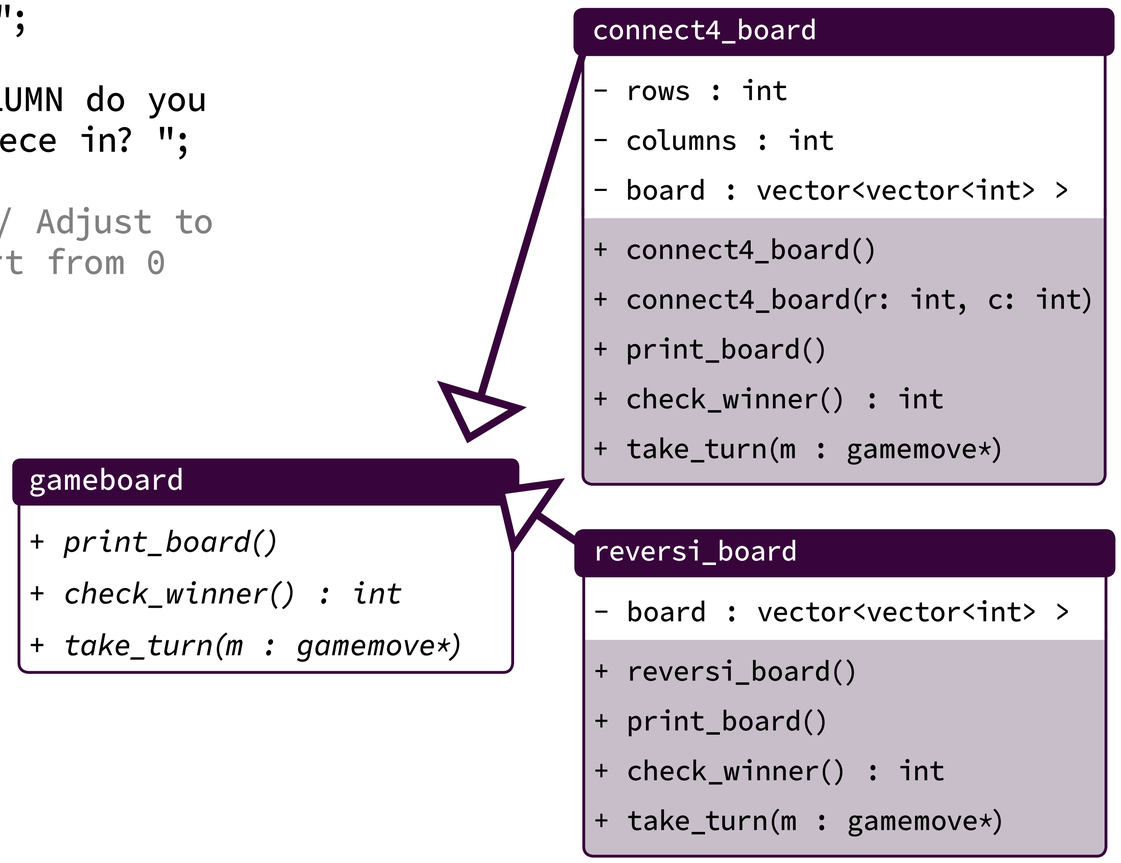
```
class reversi_move : public
gamemove{
  private:
  int row;
  int column;

  public:
  void ask_move(int nextplayer) {
    player = nextplayer;
    cout << "Player " << nextplayer
<< ", which ROW do you want to
place your piece in? ";
    cin >> row;
    cout << "Which COLUMN do you
want to place your piece in? ";
    cin >> column;
    row--; column--; // Adjust to
make the numbers start from 0
  }
  int r() {
    return row;
  }
  int c() {
    return column;
  }
  int p() {
    return player;
  }
};
```

But it's clear that both the **connect4_move** and the **reversi_move** classes are closely related, and they both implement the necessary functions needed from a **gamemove**.

Your remaining class implementation is the **gameboard** class.

First, your UML diagram gives the design of your classes. Your **gameboard** class needs to have 3 member functions. You saw all of these getting called in your **game** class implementation. The actual concrete classes will implement those functions, as well as provide constructors and member variables to hold the particular board needed for that game.



So, implementing the **gameboard** class itself is just a matter of declaring a class and creating a few pure virtual functions. Because you're not providing any implementation of those functions in this class, there's nothing more to define.

```
class gameboard {
  public:
  virtual void print_board()=0;
  virtual int check_winner()=0;
  virtual void take_turn(gamemove*)=0;
};
```

Let's consider one of the concrete implementations. The **connect4_board** implements the board for playing Connect 4. The basic operations provided here mirror those of the Connect 4 game that was developed in lecture 15. In fact, the code is almost identical in many places, with the only differences being that you can use member variables for the **board** and the number of rows and columns, rather than passing information through parameters.

Notice that you have a routine, **take_turn**, that has a parameter that's a **connect4_move** pointer. The routine itself is pretty straightforward; it uses the accessor functions you provided in the **connect4_move** class to get the player who made the move and the column and updates the board appropriately.

```
  void take_turn(connect4_move* m) {
    int last_empty = 0;
    int column = m->c();
    int player = m->p();
    while ((last_empty < rows) && (board[column][last_
empty] == 0)) {
      last_empty++;
    }
    last_empty--;
    board[column][last_empty] = player;
  }
```

But according to the definition of your virtual function, you are not supposed to be passing in a **connect4_move** pointer; it's supposed to be a **gamemove** pointer! However, a **gamemove** on its own doesn't have the specific member functions you need, such as getting the column number. So, what you need is a **connect4_move** pointer and what you have to implement is a **gamemove** pointer.

To get from the **gamemove** pointer to the **connect4_move** pointer, you can simply convert your **gamemove** pointer to a **connect4_move** pointer, just like you would do other type conversions: You put the new type in parentheses in front of the previous one. Then, everything can be called like you'd hope.

```
void take_turn(gamemove* m) {
  take_turn((connect4_move*)m);
}
```

You could also look at the implementation of the **reversi_board** class. Some parts of Reversi are easier than Connect 4 while others are more complex.

Notice that like the **connect4_board** case, you have to convert a **gamemove** pointer into a **reversi_move** pointer to get access to the particular move functions.

```
void take_turn(gamemove* m) {
  take_turn((reversi_move*)m);
}
```

Also like the **connect4_board** case, you defined some extra helper functions to keep code to a manageable size, because the process of looking at adjacent pieces to the left, right, up, down, and on all 4 diagonals is a lot of code to put into one place.

So, you create 8 different helper functions, each of which is defined as a private member function. And you can then call those from your **take_turn** function.

```
mark_up(m->r(), m->c());
mark_down(m->r(), m->c());
mark_left(m->r(), m->c());
mark_right(m->r(), m->c());
mark_diagul(m->r(), m->c());
mark_diagur(m->r(), m->c());
mark_diagdr(m->r(), m->c());
mark_diagdl(m->r(), m->c());
```

The final piece of your program is the **main** program. You just ask a user which of the 2 games to play. Depending on the game, you declare either a **connnect4_move** and **connect4_board** or a **reversi_move** and **reversi_board**. You then create a game, using that board and move as arguments to the constructor. And you start the game by simply calling **start_game** on that game.

```
int main() {
  cout << "I can play Connect 4
or Reversi.  Which do you want to
play?" << endl;
  cout << "Enter 1 for Connect4,
anything else for Reversi: ";
  int choice;
  cin >> choice;
  if (choice == 1) {
    connect4_move m;
    connect4_board b;
    game g(b, m);
    g.start_game();
  }
  else {
    reversi_move m;
    reversi_board b;
    game g(b, m);
    g.start_game();
  }
}
```

And with this, you can actually play the game. ◆

READINGS

a Object-oriented design is a topic all its own with books written just about it. One of the classic books that is still widely followed is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

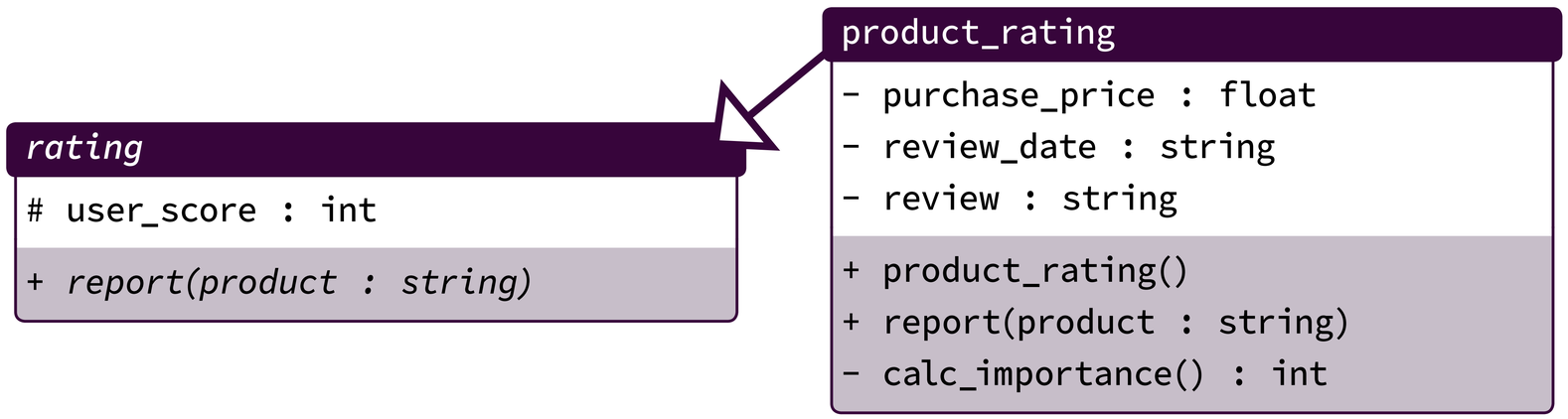
b There are many uses of UML beyond class design, and class descriptions can be more complex than what was shown in this lecture. For a more detailed overview, see www.uml-diagrams.org.

c Ousterhout, *A Philosophy of Software Design*, chaps. 6–9. (Though not specifically object-oriented, the design principles are useful at this level.)

// QUIZ

1 Assume you have the UML diagram given here:

Implement the classes that are shown. Note that you do not need to fill in the actual function behavior, but you should set up the functions (with empty bodies) appropriately.



Click here to see the answer.

// QUIZ ANSWER

1 Here is the corresponding code:

```
class rating {
    protected:
    int user_score;

    public:
    virtual void report(string) = 0;
};

class product_rating : rating {
    private:
    float purchase_price;
    string review_date;
    string review;

    int calc_importance() {
        // Code for calc_importance
    }

    public:
    product_rating() {
        // Base constructor
    }

    void report(string product) {
        // Instatiation of virtual function
    }
};
```

Notice that the first class, **rating**, is an abstract class, with a pure virtual function, **report**. It also has a protected member variable, the integer **user_score**. This corresponds to the UML diagram: # indicates a protected variable while + indicates that the pure virtual function is public.

Likewise, for the **product_rating** class, there are 3 private member variables (designated in the UML by –). The member variable names and types also correspond to the UML. Then, there are 3 member functions defined. One of these is the constructor. It is the default constructor, so it takes no parameters. It is a public member. The actual instantiation of the pure virtual function is also given, and again it is a public member function. Finally, one private member function, **calc_importance**, is also defined. It takes no parameters and returns an integer, as the UML diagram indicates.

Click here to go back to the quiz.

22 C++ Templates, Containers, and the STL

When you have an idea that's so general that it's not really tied down by any single class hierarchy, that's when you turn to generic programming. While procedural programming orients the program around functions and object-oriented programming orients the program around classes (and their objects), generic programming orients programs around very general concepts of data structures and **algorithms**. You can think of generic programming as a way of defining classes and functions that are not tied to any specific data type. Instead of using a data type, it uses a template.

IN THIS LECTURE:

// TEMPLATES AND CONTAINERS

Basically, **templates** are a way of saying that a class's member variables, or a function's parameters, are not of a specific type but instead are of some generic template type. So, rather than the class or function working only for one specific type, it can be used for any types that match some template. When you actually want to use a specific class or function, you then specify the specific type that the template will take on.

For example, the vector is a templated class. It lets you store a list of any type of object, but then to use it, you have to create a vector that stores some particular type. So, although vectors have been referred to as a class, they are really a more general container capable of holding a variety of unrelated classes.

```
vector<int> v1;
vector<float> v2;
vector<string> v3;
vector<my_class> v4;
```

In ordinary English, the term *generic* often means "not special" or even "not as good," but in programming, the term *generic* means that you are "generalizing" some concept or idea.

The angle brackets, as used when declaring vectors, are used to specify what particular type an instance of a class or function will take on. The type that's specified in the angle brackets is the type that replaces the template type throughout the class or function.

There are several very general tasks and ideas—things that are very widely applicable—that work well with templates.

The most common of those general concepts are made available to programmers through the Standard Template Library (STL), which is a library—a collection of templated classes and functions that can be included into the program. It's standard in the sense that there is a clearly defined set of what is expected to be provided in the library. And it uses templates, meaning that it provides generic container structures and algorithms.

A programmer who is not using the STL is missing a huge chunk of the advantages that C++ provides.

The STL can save a lot of time in coding by providing efficient implementations of data structures and algorithms you'd want to use. And the STL is simple to use; #include gets you access to a generic structure or the algorithms it provides.

The STL is likely all you'll need for generic programming; you will probably rarely find that you want to write your own templated classes or functions.

The vector class is an example of a container, and it's just one of several generic containers that are provided in the STL.

Containers are classes that collect several instances of some type together in a way that allows certain operations to be performed very efficiently.

Different containers will have different operations—different member functions—that they support well.

The common theme is that they are templated classes, so they can work with many different data types; you specify the type the container will use when you declare a specific container.

For example, vectors are very good for keeping elements in order; the order that elements of the vector are in is the order in which they got pushed onto the back of the vector. It's also very easy to add or remove elements at the end:

- **push_back** adds a new element to the end of the vector.
- **pop_back** removes an element from the end of a vector.

Vectors are also great at letting you access an element at a specific point in the vector. This is referred to as random access because you can access any point given at random; you don't have to access them in a particular order. But a vector is not very good at inserting a new element at the beginning, or somewhere in the middle, of an existing vector.

Here's a short menu of the most useful containers that are in the STL.

Containers	Container Adaptors
vector	stack
deque	queue
list	priority queue
forward list	

You can order from this menu in one of 2 ways:

- You can call for a container that's implemented in a fundamentally unique way. Such containers include vectors, deques, lists, and forward lists.
- There are also container adaptors, which include stacks, queues, and priority queues. You can think of a container adaptor as something that's not a new container on its own, but it makes use of the other containers to provide some set of functions.

In practice, you use containers and container adaptors in very similar ways. Either way, all you do to order from the menus is #include, for example, **stack**.

// STACKS

If you want to keep track of a set of actions you took so that you can undo back to any point, you can do this using a very basic container adaptor that provides one of the most basic operations needed in computing. This is the **stack**, which is what's called a LIFO (last in, first out) structure. It means that the last thing you put on the stack is the first thing that comes out of the stack.

In C++, you can declare a stack of some particular type and then perform one of a few operations. There are 4 main operations that you need to do to get just about everything you need to with a stack:

- **push** adds a new element to the top of the stack.
- **top** is used to see what element is on top of the stack.
- **pop** removes the top element from the stack (whatever it is).
- **empty** returns a Boolean value that tells you whether or not the stack is empty.

Like most classes in the STL, there are also a few other functions defined, such as a **size** function giving the size of the stack, but these 4 are the only ones you need on a regular basis.

In this **Program 22_1**, notice that you #include **stack** at the beginning of the program (4).

Then, in your **main** program, you declare a stack, **s**. Because the stack is a generic container, you have to give it a particular type—in this case, that's the integer type. So, you write **stack** and then **int** in angle brackets, and then you give the name of the stack variable, which in this case is **s** (8).

> Remember that the angle brackets are the way of specifying a particular type—the type that will replace the template in the class definition.

```
1   // Program 22_1
2   // Stack example
3   #include<iostream>
4   #include<stack>
5   using namespace std;
6
7   int main() {
8       stack<int> s;
9       s.push(10);
10      s.push(15);   a
11      s.push(20);
12      while (!s.empty()) {
13          cout << s.top() << endl;
14          s.pop();
15      }
16  }
```

You start by pushing 3 values onto the stack: **10**, **15**, and **20**; that is, you use the member function **push** and pass in the values **10**, then **15**, and then **20** as arguments. This will create a stack that should have **10** first so that it's on the bottom, then **15** on top of that, and then **20** on top of that (a).

Next is a loop that continues as long as the stack is not empty. You use **!s.empty()** as the condition for the **while** loop (12).

For each iteration of the **while** loop, you do 2 things. First, you print out whatever is on top, using the **top** member function to have access to that top element (13). Then, you pop off the top element, reducing the size of the stack by 1, using the **pop** command (14).

And if you run this code, you indeed see the numbers printed off in reverse order.

// QUEUES

An alternative container adaptor gives you a FIFO (first in, first out) data structure. This is called a **queue**, and it's very much like the stack, but instead of the last thing in being the one you can access, it's the first thing in. Queues are used to represent any sort of process where the first thing into the line is the first thing you want out.

It has 4 main operations, just like the stack did:

- **push** adds an element to the end of the queue.
- **front** tells you which element is in the front of the line (analogous to **top** from the stack commands).
- **pop** removes the element at the front of the queue.
- **empty** again tells you if the queue is empty or not.

In **Program 22_2**, notice that now you #include **queue** at the beginning (4).

In the **main** body of the code, you declare a queue, where the type that you want the queue to hold is an integer. So, you write **queue<int>** and then the name of the queue, which in this case is just **q** (8).

You can push the numbers **10**, **15**, and **20** into the queue (b), just like you pushed into the stack, and you can have a loop that continues as long as the queue isn't empty (12).

Inside the loop, you'll print the element at the front by using the **front** member function and outputting **q.front()** (13). Then, you remove that front element by calling **pop** (14).

And you see that the elements came off in the same order you put them in.

In addition to the vector, stack, and queue, another container is the **deque** (short for *doubly ended queue*).

A deque works a lot like a vector. It's just that instead of only having **push_back** and **pop_back**, it has **push_front** and **pop_front**, too. It has basically all the other operations that a vector has: You can get the size using **size** or make it empty using **clear**. You can access elements using square brackets with the element number or using the **at** command so that you can guarantee you're not reading or writing past the end of the allocated memory.

```
1   // Program 22_2
2   // Queue example
3   #include<iostream>
4   #include<queue>
5   using namespace std;
6
7   int main() {
8       queue<int> q;
9       q.push(10);
10      q.push(15);      b
11      q.push(20);
12      while (!q.empty()) {
13          cout << q.front() << endl;
14          q.pop();
15      }
16  }
```

But it's less efficient than a vector, and there is rarely an instance where you need all the extra functionality that a deque provides.

// LISTS AND ITERATORS

While these 4 containers work well for adding and removing new elements at the ends, if you want to insert or delete elements at arbitrary points, then a container called a **list** is particularly useful.

But to understand how to use lists, you first need to understand **iterators**.

You've encountered the idea of iterating through a vector. You usually have some index, such as `i`, that you put in a `for` loop and let take values from 0 to 1 less than the vector length. So, you can access element 0, then 1, then 2, and so on, until the end of the vector.

An iterator is a more general form of that index. It works like a pointer to the element in a container.

To use an iterator, you would start at the beginning of the vector and then access element after element until you reached the end. For a container, you can get a `begin` iterator, which points to the first element. You can increment the iterator so that it goes to the next element, then the one after that, and so on, all the way through. Finally, you can stop when you reach the end, which will be after the last element in the container.

An iterator is a new type that you can use for your variables. When you declare an iterator, you first give the container (such as a vector of integers or a list of floats) that it will iterate over, then `::`, and then the keyword `iterator`. This defines the type of the variable you're declaring.

For example, in **Program 22_4**, you declare a vector of integers where you push 3 values onto the back: `10`, `20`, and `30` (c).

Next, you create an iterator that can iterate over a vector of integers, named `my_iterator`. So, you write `vector<int>::iterator my_iterator` (12). Again, you can think of this as being like a pointer to an element of a vector of integers.

You can initialize this by assigning the iterator to the beginning of the container. For your vector, and for many other containers, the member function `begin` will return an iterator at the first element in the container. In the code, you write `my_iterator = v.begin()` (13).

There's also an `end` function associated with most containers, and it indicates a value after the last element in the container. In other words, there's not an element at the "end"; it is not pointing to the last element, but instead to some part of memory that you don't care about, other than to know that you're no longer in the container. So, in this code, you're going to loop as long as you're not at the end. So, you write `while (my_iterator != v.end())` (14).

```
1   // Program 22_4
2   // Iterator example
3   #include<iostream>
4   #include<vector>
5   using namespace std;
6
7   int main() {
8     vector<int> v;
9     v.push_back(10);
10    v.push_back(20);
11    v.push_back(30);
12    vector<int>::iterator my_
    iterator;
13    my_iterator = v.begin();
14    while (my_iterator !=
    v.end()) {
15        cout << "This iterator
    refers to: " << *my_iterator
    << endl;
16        my_iterator++;
17    }
18  }
```

C

Inside the loop, 2 key things happen:

- You dereference the iterator—that is, like a pointer, you get the value of the thing the iterator is referring to. In this case, you output it. So, the first time through the loop, you'll be looking at the first element of the vector, which is **10**, so **10** is output.

In very rare cases, iterators can't be dereferenced.

- The other thing that happens is something that is guaranteed to be possible for every iterator: an increment. This is the **++** you would use for incrementing an index value. It's OK to write **++** before or after the iterator; this moves the iterator to the next element in the container.

So, in this case, you have a loop that causes the iterator to go to every element in the container, and you print it out.

This way of iterating through a container like a vector is very common, so it's not unusual to see a for statement that starts from the beginning of the container and iterates through to the end. And because the iterator is usually only defined for that loop, as you loop over the elements of the container, it's common to put the declaration into the for loop itself.

There's a feature in C++ that will let you avoid writing the long declaration **vector<int>::iterator**. You can just use **auto**, and the compiler will automatically infer the type of the variable needed.

You can actually use **auto** in other contexts, too, to get the type of a variable automatically rather than having to specify it. It still has a fixed, single type, but it saves the programmer work.

In **Program 22_7**, you create a vector with 3 elements: **10**, **20**, and **30** (8). You then declare an iterator named **iter** (9). You set this to **begin** and increment it once so that **iter** is now at the second element in the vector (d). You call an **insert** command for the vector, passing in **iter** and the value you want to insert. So, this value, **40**, gets inserted before the second element (12).

When you print out the contents of the vector, you see that the elements are in order, as expected.

There's a similar process for erasing data from a container.

Iterators are important for specifying routines to insert or erase data in a container.

Inserting and deleting are not very efficient for containers like vectors and deques. Instead, there's another container that is much better at handling insertion and deletion: the list, and its related form, the forward list.

A C++ list is referred to by computer scientists as a linked list. Each element exists in a single place in memory, not necessarily contiguous with the next element in the list. Each element has a pointer to the next element in the list. And a doubly linked list also has a pointer back to the previous element.

So, if you are at one element, it's easy to get to the next one, and in a doubly linked list, it's also easy to go to the previous one. And you can easily see how an iterator would go through the list from beginning to end.

```
1   // Program 22_7
2   // Insertion example on a vector
3   #include<iostream>
4   #include<vector>
5   using namespace std;
6
7   int main() {
8      vector<int> v = { 10,
    20, 30 };
9      vector<int>::iterator iter;
10     iter = v.begin();   // Iterator
    is at first element
11     iter++;   // Now at second
    element
12     v.insert(iter, 40);
13     for (int i = 0; i <
    v.size(); i++) {
14         cout << v[i] << endl;
15     }
16  }
```

d

But a list does not allow you to jump directly to an element somewhere in the middle.

The advantage to lists is that you can insert into them very easily. If you inserted something into a vector, you'd have to potentially copy huge amounts of information as you basically slide the existing elements down one spot.

With a list, though, all you have to do is take the new thing to add and update a few pointers. Likewise, merging 2 lists is a very quick task, while vectors or deques would require copying one of the 2 lists.

When you type **list**, you get a doubly linked list. A **forward_list** gets you a singly linked list and will allow pushing only on the front of a list. And insertions take place after the element that an iterator is referring to.

Suppose you want to have a list of instructions for making a peanut butter and jelly sandwich. Notice in **Program 22_8** that you #include **list** at the beginning (5). **List** is templated, so you create a **list**, named **instructions**, of type **string** (9).

You can add elements to the front or back of a list using **push_back** and **push_front**, just like you could with a deque. In this case, there are 3 steps of making the sandwich (e).

```
1   // Program 22_8
2   // List example, including insertion
3   #include<iostream>
4   #include<string>
5   #include<list>
6   using namespace std;
7
8   int main() {
9      list<string> instructions;
10     instructions.push_back("Get bread");
11     instructions.push_back("Put on jelly");
12     instructions.push_back("Put two halves together");
13     list<string>::iterator insert_spot;
14     insert_spot = instructions.begin();  // Refers to first element
15     insert_spot++;  // Refers to second element
16     instructions.insert(insert_spot, "Put on peanut butter");
17     for (auto iter = instructions.begin(); iter != instructions.end(); iter++) {
18         cout << *iter << endl;
19     }
20  }
```

e

f

g

If you left out an important step, such as adding peanut butter, you can use insertion to add the element in. You can create an iterator that you increment to refer to the second element (f). Then, calling **insert** at that point will insert a new element just before the second element (16).

And you can use an iterator to loop through and print out the resulting elements in order. Notice that the instruction you put in the second spot—**Put on peanut butter**—appears in the correct place (g).

The forward_list is very similar, but you can only call **push_front**, or insert after an element. In this code, you push your instructions in reverse order because they're getting pushed onto the front of the list (h).

Then, you have an iterator that is set to the second element (i). When you call **insert_after** at that point, you insert your instruction after the second element (16).

So, printing the whole list gives you your instructions. Notice that now the peanut butter is getting put on after the jelly, because the insertion would always come after the element you were pointing to.

Iterators can also make your use of vectors and data structures more powerful by using the **for** loop in a new way. If you have a class that can be iterated through, you can form a special **for** loop. Basically, you can set up a **for** loop that has a variable that takes on the value of each element in a container. The **for** loop will iterate from beginning to end, and the variable will take on each value in succession.

The syntax for this is to write **for** and then, in parentheses, 2 items separated by a colon. The first of these is the variable type and the variable name. When you use this type of function, the variable is only going to be in scope during the loop, so it is not a restriction to make it be declared only in the **for** statement.

Note that you can use **auto** for the type if you want. However, if you're going to want to manipulate the elements of the container, you should make the type a reference.

After the type and variable name, there's a single colon and then the name of the container that you want to iterate through.

```
1   // Program 22_9
2   // Forward_list example, including insertion
3   #include<iostream>
4   #include<string>
5   #include<forward_list>
6   using namespace std;
7
8   int main() {
9      forward_list<string> instructions;
10     instructions.push_front("Put two halves together");
11     instructions.push_front("Put on jelly");                    h
12     instructions.push_front("Get bread");
13     forward_list<string>::iterator insert_spot;
14     insert_spot = instructions.begin();  // Refers to first element   i
15     insert_spot++;  //Refers to second element
16     instructions.insert_after(insert_spot, "Put on peanut butter");
17     for (auto iter = instructions.begin(); iter != instructions.end(); iter++) {
18         cout << *iter << endl;
19     }
20  }
```

In **Program 22_10** is your list with the steps to make a peanut butter and jelly sandwich.

In line 19, you have a **for** statement. The container, the list, is made up of strings, so you are going to have a new variable—**step**, in this case—that is of a **string** type. You're going to want to manipulate elements of **instructions**, so you declare **step** to be a string reference, not just a string. Then, there is a colon followed by the name of the container, **instructions**.

Within the loop, you can do anything you want to do with **step** and it will refer to the elements of the list itself. So, in this case, you are adding the string **Step:** to the beginning of each element (16). On the first iteration of the loop, then, the value of the first element of **instructions** has **Step:** added to the front. The loop ends up adding this to all elements.

Then, you can have another loop through the elements of **instructions**, this time to print out the results. You set up your loop—**for(auto step : instructions)**—which says you're going to iterate through the entire container **instructions**, and **step** is going to refer to the element of **instructions** on each such iteration (19).

Inside the loop, when you print out the values, you see that they were indeed changed by the previous loop (20). ◆

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 20.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 3.4 and 9.1–9.3.

```
1   // Program 22_10
2   // Example of for-each statement
3   #include<iostream>
4   #include<string>
5   #include<list>
6   using namespace std;
7
8   int main() {
9      list<string> instructions;
10     instructions.push_back("Get
    bread");
11     instructions.push_back("Put on
    peanut butter");
12     instructions.push_back("Put on
    jelly");
13     instructions.push_back("Put
    two halves together");
14
15     for (string& step :
    instructions) {
16        step = "Step: " + step;
17     }
18
19     for (auto step :
    instructions) {
20        cout << step << endl;
21     }
22  }
```

Exercise

Which container is the best to use in each of the following scenarios?

1. You're running a business and want to process orders in the same order they arrive in.
2. You have a long sequence of directions, and you want to be able to examine any step in detail.
3. You're keeping track of movies you want to see in order from the ones you're most interested in to least interested in. You'll want to be able to add new movies that come out or cross off movies once you've watched them.

Click here to see the solution.

Exercise Solution

1 This is perfect for a queue, which has the FIFO property.

2 The sequence of directions would work well with a vector, which lets you easily access any point in the middle.

3 This is good for a list, which lets you easily add or remove movies at any point in the middle.

Click here to go back to the exercise.

// QUIZ

1 Which container would best be used in each of the following situations?

a You have several phone messages coming in and want to handle them in the order they were received.

b You have a phone bill arriving each month and want to be able to look up any month's bill.

c You are planning a road trip and have listed several stops along the way but want to be able to add a new one partway through.

d You are having a conversation and keep going off on new topics. You want to keep track of the conversation so that once you finish one topic, you can return to the previous one—and then when that is done, the one before that, etc.

2 Write a program to keep track of the order of people who arrive to meet with you. You should repeatedly ask for a name, or, if the user types next, you should print the name of the next person who should be met. An example of a run of this might be as follows:

```
John
James
next
The next person to see is: John
Joseph
next
The next person to see is: James
```

3 Write a function that takes in a vector of integers and then uses an iterator to print out each element, one per line.

Click here to see the answers.

// QUIZ ANSWERS

1 There may be more than one type that is appropriate for each situation, but the following are the most natural containers for the given task:

a Queue. This is an example of a FIFO ordering, which is what queues are best used for.

b Vector. The vector is good for adding new items at the end and for random access—directly accessing any point in the middle.

c List. Lists keep items in sequence but make it easy to add to (or delete from) the middle. Practically, though, a vector would work about as well here, unless the list of stops was extremely long.

d Stack. The idea here is that as the conversation continues, if there is a new topic, it is added to the stack. If the topic is finished, then it is popped from the stack, and you return to the previous topic.

2 Here is one way to implement such a program:

```
1   #include<iostream>
2   #include<queue>
3   #include<string>
4   using namespace std;
5
6   int main () {
7       queue<string> names;
8       string s;
9       cout << "Enter a name, or
type \"next\" to get the next
person to see" << endl;
10      while(true) {
11          cin >> s;
12          if (s=="next") {
13              cout << "The next
person to see is: " << names.
front() << endl;
14              names.pop();
15          } else {
16              names.push(s);
17          }
18      }
19  }
```

Notice that a queue is used to keep track of the names. The loop continues indefinitely, and in each iteration of the loop, you read in a string. If it is next, then you print the item at the front of the queue (and then remove that item). Otherwise, you just add the name onto the queue.

3 Here is one way to do this:

```
    void print_elements(vector<int> v) {
      vector<int>::iterator i;
      for(i=v.begin(); i !=
v.end(); i++) {
        cout << *i << endl;
      }
    }
```

Notice that you declare an iterator for the vector. Then, you loop from the **begin()** to the **end()** of the vector, incrementing the iterator each time. You output the element of the vector by dereferencing the pointer. There are several other ways the same goal could be achieved, though.

Click here to go back to the quiz.

23 C++ Associative Containers and Algorithms

IN THIS LECTURE:

The Standard Template Library (STL) provides a wide variety of data structures and algorithms that you can use to write powerful and efficient code in C++. You've seen previously a set of unitary containers and container adaptors that you can use to hold individual elements one by one. But there are also containers—called **associative containers**—that bring, or associate, 2 or more types of elements together. Also in the STL are algorithms, which provide an even more powerful set of standard tools for processing data.

// CONTAINERS

/* PAIRS */

The simplest of the associative containers is the **pair**. There are many times that you want a very simple structure that just joins 2 things together. For example, you might be keeping track of people and want a person to have an age and a name. The pair provides a quick and easy way of joining 2 items into one structure.

A pair takes 2 templated types: the type of the first item and the type of the second item. When there are 2 or more types needed for a template, they are separated by commas within the angle brackets.

To declare the pair of age and name, you'll want an integer and a string. You'll write **pair<int, string>** and then the name of the pair, which in this case is **p** (8).

```
1    // Program 23_1
2    // Pair example
3    #include<iostream>
4    #include<string>
5    using namespace std;
6
7    int main() {
8       pair<int, string> p;
9       p.first = 45;
10      p.second = "John";
11      cout << p.second << " is " <<
     p.first << " years old." << endl;
12   }
```

To access the first element of a pair, you use **.first**. And the second element is just **.second**. So, to assign a value to the pair, you can write **p.first =** and **p.second =**. And to access the values in the pair, you again use **first** and **second**: **cout << p.second << " is " << p.first << " years old." << endl**.

You could also make your own class that combined 2 specific types, and one advantage of doing that over a pair is that the class name would give an indication of what the intention of the class is. You can get around this by using a typedef, which is a way of creating a new name for a type. The syntax is to write **typedef**, then the actual type, and then the name you want to use to refer to that type.

For example, if you didn't like using **int** and wanted to call integers numbers, you could write **typedef int number** and then you could declare things of type **number**—they'd still be integers.

So, in **Program 23_2**, you could create a new type, **person**, that is just a pair of an integer and a string by writing **typedef pair<int, string> person**. This then lets you declare variables of type **person**, and they'll be integer-string pairs (7).

While this lets you have a more descriptive name overall, accessing the member variables is still going to be done with **first** and **second** (12), and there's not an easy way to extend this if you ever decided you wanted something different.

Another command available for a pair is **make_pair**, which takes in 2 arguments and returns a pair of those 2 arguments. It can be a useful way of setting the elements in one statement rather than with 2 separate assignments (11).

```
1    // Program 23_2
2    // Using typedef to create a new
     name for a type
3    #include<iostream>
4    #include<string>
5    using namespace std;
6
7    typedef pair<int, string> person;
8
9    int main() {
10      person p;
11      p = make_pair(45, "John");
12      cout << p.second << " is " <<
     p.first << " years old." << endl;
13   }
```

/* TUPLES */

If you want to group more than 2 items together, you can use a **tuple**, which lets you group an arbitrary number of different types of objects together. You just list those types in the angle brackets. You also have a **make_tuple** command, which is similar to **make_pair**.

The downside of a tuple is that you don't have a simple **first** and **second** to access individual elements. Instead, there's a function named **get** that has a really odd form: You have to put the element of the tuple you want in angle brackets after **get**. In the parentheses afterward, you put the tuple name. That will give you access to that element of the tuple.

```
1    // Program 23_3
2    // Tuple example
3    #include<iostream>
4    #include<string>
5    #include<tuple>
6    using namespace std;
7
8    int main() {
9       // Tuple will hold name, team,
     age, height, weight
10      tuple<string, string, int,
     double, double> c;
11      c = make_tuple("James Smith",
     "Cubs", 22, 73.5, 182.1);
12      cout << get<0>(c) << " is " <<
     get<3>(c) << " inches tall and
     weighs "
13           << get<4>(c) << " pounds."
     << endl;
14   }
```

Suppose you're running a business where you're letting people buy on credit and you want to keep track of their name, how many purchases they make, and how much they owe you.

In this case, you'd want to create a tuple storing 3 types: a string for the name, an integer for the number of purchases, and a double that can hold decimal values for the amount owed.

The tuple type can be named **customer** to make it more descriptive (8). Then, you can use this type, **customer**, to declare a variable, **c**.

You'll have a **make_tuple** command that you can use to make an instance of that tuple type. Here, the tuple says that John made 3 purchases and owes $100 (12).

You can then use a **cout** statement to access the elements. To access the first element, you write **get<0>(c)**, which outputs **John**. Then, **get<1>(c)** gives you **3** purchases, and **get<2>(c)** gives you **100.0** in dollars (a).

The **get** function can also assign values to the elements or manipulate them. Because it returns a reference, you can do anything to the result of **get** that you would have done to the element. In this code, you're incrementing the second element, indicating one more purchase, and increasing the third element by $50 (b).

```
1   // Program 23_4
2   // Tuple example - extended
3   #include<iostream>
4   #include<string>
5   #include<tuple>
6   using namespace std;
7
8   typedef tuple<string, int,
    double> customer;
9
10  int main() {
11     customer c;
12     c = make_tuple("John", 3,
    100.0);
13     cout << get<0>(c) << "
    has made " << get<1>(c) << "
    purchases and owes $"
14           << get<2>(c) << endl;
15     get<1>(c)++;
16     get<2>(c) += 50.0;
17     cout << get<0>(c) << "
    has made " << get<1>(c) << "
    purchases and owes $"
18           << get<2>(c) << endl;
19  }
```

a

b

/* PRIORITY QUEUES */

The **priority queue** only takes in one type, so it's not associative, but it's often used with associative containers like pairs or tuples.

The idea of a priority queue is that you want to be able to insert items into the container and then pull them out in order from greatest to smallest. The values of a priority queue are the priority, or how important that element is. So, each time, the element at the front of the queue should be the one with the greatest priority. Each time you pull an element out of the queue, it will be the highest-priority element in the container.

To create an instance of a priority queue, you give a single data type. For the priority queue to work, this data type must have a less-than comparison operator defined; this is how elements are compared to find which is largest. If you want to have a priority queue of your own type, make sure it has **<** defined.

When using **priority_queue** with a pair or a tuple, the comparison is based on comparing the first element, and, if they are equal, then the second. For a tuple, this process continues with the remaining elements. Either way, the comparison is already defined so that you don't have to explicitly create a less-than comparison.

Pairs are especially common when using priority queues, because you often need to have 2 pieces of information: some value that you are using to determine priority and the actual object associated with that value.

Suppose you wanted to have a priority queue where you were pulling people out from oldest to youngest. Then, you could associate a pair where the first element was **age** and the second element was the **person**. The largest age will always get pulled out.

You can use the following when writing code:

- **push** adds elements to the priority queue.
- **top** accesses the highest-priority element.
- **pop** removes the top element.
- **size** gets the size of the priority queue.
- **empty** returns whether the priority queue is empty or not.

```
1    // Program 23_5
2    // Priority Queue example
3    #include<iostream>
4    #include<string>
5    #include<queue>
6    using namespace std;
7
8    typedef pair<int, string> person;
9
10   int main() {
11      priority_queue<person> pq;
12      // Could have been priority_queue<pair<int, string> > pq;
13      pq.push(make_pair(18, "Jack"));
14      pq.push(make_pair(16, "Jill"));
15      pq.push(make_pair(19, "Joe"));
16      pq.push(make_pair(17, "Jessica"));
17      cout << "The oldest person in the group is: " << pq.top().second << endl;
18      pq.pop();
19      cout << "The next oldest person in the group is: " << pq.top().second << endl;
20   }
```

C

Consider **Program 23_5**. A priority queue is contained within the **queue** library, so you #include **queue** (5). You use **typedef** to create a **person** type that consists of an integer and string pair (8). The integer will be the age, and the string will be the person's name.

You then declare a **priority_queue** of type **person**. Because the first element of a **person** is the age, the highest-priority element in the queue will be the person with the largest age. The priority queue is named **pq** (11).

Next, you insert 4 different elements into **pq**. For each, you call **pq.push**, and then in the parentheses, you give a **person**—that is, an integer-string pair. To create this pair, you use the **make_pair** function, passing in one integer and one string every time you call it. Overall, you'll insert **Jack** at age **18**, **Jill** at age **16**, **Joe** at age **19**, and **Jessica** at age **17** (c).

Next, you look at the first element in the **priority_queue**. This is the element that is the largest, so in this case, it's the one with the greatest age. You write **pq.top()** to get the oldest person in the group. Then, using **.second**, you get the name of that person. This is what is output. In this case, **Joe**, who is 19, is first in the **priority_queue**, so **Joe** is output (17).

After popping the first element off of the **priority_queue** (18)—that is, after removing **Joe**—you can again look at the **top** element. This time, it's the oldest of the remaining elements, **Jack** (19).

The **map** is a way to associate one value, called a key, with another of any type. Maps come in 2 varieties: the standard **map** and the **unordered_map**. The difference between them is that a **map** can be iterated through in order from least to greatest, while an **unordered_map** cannot. Generally, this means that an **unordered_map** is faster to use.

A map uses square brackets to index specific values. This is similar to how a vector or an array uses square brackets to access one element of the array. There, the index that you can put in that square bracket runs from 0 to one less than the size of the vector or array.

But with a map, you can use almost any value as the index—not just an integer from 0 to some other integer. This index is just the key. So, if you wanted to store the number of bills of different currency denominations, you could use **1**, **5**, **10**, **20**, and **100** as your keys. Or you could use a string as the key so that the indices are **cat** and **dog**, for example.

Say you're running an animal shelter and want to keep track of how many of each type of animal are in the shelter. You'll use a map for that, so you #include **map** at the beginning (5).

Maps are templated with 2 types: first the type of the index and then the type of value that the index maps to. In this case, you'll want to have an animal name as your index, so the index will be a string. And you're keeping track of how many of each animal there are, so you'll have an integer value stored for each of them. So, you declare your map variable, **numanimals**, by writing **map<string,int> numanimals** (9).

Now you can assign values to this map. For example, if you want to note that there are 12 cats in the shelter, you can write the map name, **numanimals**, followed by the string **cat** inside of square brackets, and assign the value **12** to that. Likewise, you can assign values for **dog** and **rabbit** (d).

In this program, you then ask a user which animal is of interest (e). You first check the count for that animal (17). If the count is not 0—that is, if it's 1—then you will output the value stored in **numanimals** by outputting **numanimals[whichanimal]**, where **whichanimal** is the string the person entered (18). If the count was 0, it means you have no record for that animal, so you output a short message that the animal is not in the shelter (21).

If a user enters an animal that you have data for, such as **dog**, you'll get an output that there are 23 dogs in the shelter.

```
1   // Program 23_6
2   // Map example - animal shelter
3   #include<iostream>
4   #include<string>
5   #include<map>
6   using namespace std;
7
8   int main() {
9      map<string, int> numanimals;
10     numanimals["cat"] = 12;
11     numanimals["dog"] = 23;
12     numanimals["rabbit"] = 2;
13
14     string whichanimal;
15     cout << "Which animal do you want a count of? ";
16     cin >> whichanimal;
17     if (numanimals.count(whichanimal) > 0) {
18         cout << "There are " << numanimals[whichanimal] << " in the shelter." << endl;
19     }
20     else {
21         cout << "That animal is not in the shelter." << endl;
22     }
23  }
```

d

e

Sometimes you'll hear a map referred to as a dictionary, because it provides a way to associate an "entry"—that is, the key—with its "definition," or value.

Maps come with a variety of predefined member functions, including the following:

- **`at`** refers to an element (like a vector).
- **`begin`** gets the iterator at the very beginning.
- **`end`** gets the iterator just after the end.
- **`count`** returns either 0 or 1, noting whether that index appears.
- **`find`** gives an iterator to some particular element.

Exercise 1

Suppose you wanted to create a dictionary with words and definitions using the map structure. How would you go about declaring that and then putting an entry in?

Click here to see the solution.

// TEMPLATED FUNCTIONS

In addition to containers, the STL also offers a set of standalone functions that can be called with a wide range of argument types. These are called **templated functions**, because they are not just defined for a single type or set of types but can take on a whole range of types.

These templated functions are all contained in the STL **`algorithm`** library, so to use any of them, you just #include **`algorithm`**.

An **algorithm** can be thought of as a set of steps to follow to handle data or accomplish some other task.

The STL **`algorithm`** library contains more than 80 different templated functions, each implementing some algorithm. Templated functions can take a range of various types as input parameters. The STL **`algorithm`** library's functions fall into categories ranging from simple minimum and maximum computation to modifying sequences.

The point of the **`algorithm`** library is not to provide functions you couldn't create on your own, but instead to provide reliable, efficient implementations of commonly used algorithms that can be applied across a wide variety of data types.

Three of the most common templated functions in the STL are **`find`**, **`sort`**, and **`lower_bound`**.

Algorithms describe the critical functionality of most major computer programs, from operating systems to search engines—all the types of programs that C++ tends to be used for.

/* FIND */

Often, **find** is used to find all values in some container. It works by searching through the container and examining every element one by one to see if a value is present.

find takes in 3 parameters: a starting iterator, an ending iterator, and a value to search for. It returns an iterator: either one pointing to the first occurrence that matches the value being searched for, or, if the value is not in the container, then it returns the ending iterator.

Suppose you have a list of ages of participants in some study and want to allow a user to check whether certain ages participated. Notice that **algorithm** is #included (5).

In this case, you create a vector of integers, named **ages**, with the ages of the participants. Notice that some ages, such as 21 and 22, are repeated (9).

Then, you have an infinite loop where you continually ask the user what age to search for (f).

In the loop, you use the **find** function to try to find the particular age the person entered. You pass in **ages.begin**, which is an iterator pointing to the beginning of the **ages** vector, and **ages.end**, which is an iterator pointing just past the end of the **ages** vector. The third parameter is the value you're searching for—in this case, the integer the user entered (17).

find will return an iterator. If the value the user entered is found, then the iterator will not be pointing to the end iterator (18). So, you can just verify that the age was found, and for proof, you print the value that the iterator is pointing to (19). If the value was not found, then the iterator will point to the end iterator, so you output that the value was not found (22).

```
1   // Program 23_8
2   // Find example
3   #include<iostream>
4   #include<vector>
5   #include<algorithm>
6   using namespace std;
7
8   int main() {
9      vector<int> ages = { 23, 21, 18, 22, 21, 19, 20, 19, 27, 22 };
10     vector<int>::iterator findval;
11     int agetofind;
12
13     cout << "I have a list of ages of participants." << endl;
14     while (true) {
15         cout << "What age do you want to find? ";
16         cin >> agetofind;
17         findval = find(ages.begin(), ages.end(), agetofind);
18         if (findval != ages.end()) {
19             cout << "Found : " << *findval << endl;
20         }
21         else {
22             cout << agetofind << " not found" << endl;
23         }
24     }
25  }
```

f

/* SORT */

Sorting can be useful not only for ordering things in a neat way, but also as a way of making subsequent computations faster. If you'll be doing several searches for elements, then having the elements in a sorted order lets you use a binary search.

The STL provides a **sort** function that will take a container like a vector or list and put it in sorted order. **sort** takes in a starting iterator and an ending iterator and then sorts everything in between.

Say you have a vector, **v**, with several elements in basically random order (9). For later comparison, you print out the elements of the original vector in order, showing that they were in fact in some unsorted order (g).

Then, you sort the vector by calling the **sort** algorithm, passing in **v.begin** and **v.end** as your starting and ending iterators (15).

After that, if you print out the elements of **v**, you see that they are indeed now in sorted order (h).

```
1   // Program 23_11
2   // Sorting example
3   #include<iostream>
4   #include<vector>
5   #include<algorithm>
6   using namespace std;
7
8   int main() {
9      vector<int> v = { 4, 5, 9, 1, 15, 12, 3, 5, 7, 11, 14, 2, 9 };
10     cout << "Before sorting: ";
11     for (auto& iter : v) {
12         cout << iter << " ";
13     }
14     cout << endl;                                   g
15     sort(v.begin(), v.end());
16     cout << "After sorting: ";
17     for (auto& iter : v) {
18         cout << iter << " ";
19     }
20     cout << endl;                                   h
21  }
```

/* LOWER_BOUND */

STL also provides a **binary search** function called **lower_bound**, which repeatedly searches over some range by examining the middle and determining whether the thing being looked for is in the upper or lower half of the range.

For example, **Program 23_12** works just like the code you used for **find**, but you have to first sort the vector (15). Then, you call **lower_bound** just like you called **find** (16).

If the search finds the element, then you get back an iterator pointing to the first occurrence. If it doesn't find the element, then it points to the next-largest element it finds. In this example, if you search for the value **25**, it doesn't find that but instead returns that it found **27**—the next-largest value in the list. ◆

For a large list—say one with a million elements—binary search with **lower_bound** will be much faster than an **iterative** search with **find**. So, the up-front cost of sorting the elements one time first can easily be outweighed by the time savings on the later searches.

```
1    // Program 23_12
2    // Lower_bound example - using binary search
3    #include<iostream>
4    #include<vector>
5    #include<algorithm>
6    using namespace std;
7
8    int main() {
9       vector<int> ages = { 23, 21, 18, 22, 21, 19, 20, 19, 27, 22 };
10      vector<int>::iterator findval;
11      int agetofind;
12
13      cout << "What age do you want to find? ";
14      cin >> agetofind;
15      sort(ages.begin(), ages.end());
16      findval = lower_bound(ages.begin(), ages.end(), agetofind);
17      if (findval != ages.end()) {
18          cout << "Found : " << *findval << endl;
19      }
20      else {
21          cout << agetofind << " not found" << endl;
22      }
23   }
```

Exercise 2

Suppose you want to read in a bunch of peoples' names and salaries and then print a list of the people from lowest to highest salary. To read in the information, imagine that you just prompt users to enter their name and salary on the console repeatedly. Because people won't have a negative salary, ask the user to indicate that he or she is done by entering a sentinel value, such as a negative value for the salary.

How might you write this code?

Click here to see the solution.

Exercise 1 Solution

Here's a solution.

```
1   // Program 23_7
2   // Map example - dictionary
3   #include<iostream>
4   #include<string>
5   #include<map>
6   using namespace std;
7
8   int main() {
9       map<string, string> dictionary;
10      dictionary["rock"] = "An object that breaks scissors";
11      dictionary["paper"] = "An object that covers rocks";
12      dictionary["scissors"] = "An object that cuts paper";
13
14      for (auto& iter : dictionary) {
15          cout << iter.first << ": " << iter.second << endl;
16      }
17  }
```

Notice that when you run this, all the dictionary elements are there, but they appear in a different order than the order they were assigned—specifically, they're iterated through in alphabetical order. This is because you've used a map. If you took this same exact code and replaced **map** with **unordered_map**, you'd still get the elements printed, but they could be in a random order.

Click here to go back to the exercise.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 20.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 3.4 and 9.1–9.3.

Exercise 2 Solution

Here's one solution.

```
1   // Program 23_13
2   // Sorting Pairs
3   #include<iostream>
4   #include<vector>
5   #include<string>
6   #include<algorithm>
7   using namespace std;
8
9   int main() {
10      vector<pair<double, string> > salaries;
11      cout << "Enter a person's name and salary, one name
    and then one salary per line."
12           << endl;
13      cout << "Enter a negative salary to stop." << endl;
14      string personname;
15      double sal;
16      cin >> personname >> sal;
17      while (sal >= 0.0) {
18          salaries.push_back(make_pair(sal, personname));
19          cin >> personname >> sal;
20      }
21      sort(salaries.begin(), salaries.end());
22      for (auto& iter : salaries) {
23          cout << iter.first << " " << iter.second << endl;
24      }
25  }
```

Click here to go back to the exercise.

// QUIZ

1 Imagine that you have a map **`account`** that maps a name (represented as a string) to an account number (represented as an integer).

   a How would you declare this map variable?

   b Given a name stored in a string variable, **`s`**, how would you get the account number?

2 Write a program to read in first and last names and then output them in alphabetical order.

3 Match each STL container or algorithm to its purpose.

a	`find`	1	given 2 iterators in a container, reorder the elements from smallest to largest in that range
b	`tuple`	2	search through a container from beginning to end, returning an iterator to the first occurrence found equal to the given value
c	`map`	3	search through a sorted container using a binary search, returning an iterator to the first element equal to or greater than the given value
d	`lower_bound`	4	associate one value (the key) to another (the value); also called a dictionary
e	`priority_queue`	5	group 2 elements together into a single container
f	`sort`	6	group an arbitrary number of elements together into a single container
g	`pair`	7	a container for elements that allows the largest one to be removed

Click here to see the answers.

// QUIZ ANSWERS

1 a To declare a map, you need to first write `#include<map>`. The declaration needs to specify 2 types in the template: the type of the key (a string, in this case) and the type of the value (an integer, in this case). Then, the declaration would be:

```
map<string, int> account;
```

b To access an element of the map, use the `[]` operator, giving the key inside the `[]`. That will return the appropriate value—in this case, the integer `account`.

```
account[s]
```

2 Here is one option:

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

int main() {
    vector<pair<string, string> > names;
    string firstname, lastname;
    cout << "Enter names (first, then last), or enter \"done\" when finished." << endl;
    cin >> firstname;
    while(firstname != "done") {
        cin >> lastname;
        names.push_back(make_pair(lastname,firstname));
        cin >> firstname;
    }
    sort(names.begin(), names.end());
    for (auto p : names) {
        cout << p.second << " " << p.first << endl;
    }
}
```

Notice first that you have a container of pairs. Each name will be stored as a pair (last name and then first name). You will store these in a vector. The first loop just reads in first and last names, forms pairs of them, and adds them to the vector. Then, you use the `sort` algorithm to sort the vector. This will sort the pairs from smallest to largest; that is, it will sort the names from first alphabetically (by last name) to last alphabetically. If the last name (the first element of the pair) is the same, then it will compare first names (the second element of the pair). Finally, you use an iterator to go through the names and print them out in order.

3 a 2. The `find` algorithm uses a linear search over a (possibly unordered) container.

b 6. Tuples group arbitrary numbers of elements but can result in awkward code.

c 4. Maps are the way of letting you look up a value for a given key.

d 3. The `lower_bound` algorithm can only be used after the container is sorted.

e 7. Priority queues have elements pushed in any order, but the top element is the largest one.

f 1. Sorting is often used to make subsequent operations more efficient.

g 5. The pair groups 2 elements, which are accessed through the first and second members.

Click here to go back to the quiz.

24 Artificial Intelligence Algorithm for a Game

A pinnacle achievement for any programmer is **artificial intelligence** (AI), which means making a computer act as though it's making intelligent choices and decisions. Game playing has long been a fundamental area of research in AI. The idea is that if you can get a computer to play a game well, where the domain and options are more clearly defined and limited, then you might have a sturdy basis for extending those ideas to a more general context.

IN THIS LECTURE:

// AI GAME PLAYING

You can develop a basic approach to AI game playing to get a computer to play a game of Connect 4 with you intelligently. To do this, you create new algorithms—that is, steps to solve a problem—to increase the artificial intelligence of your program.

Start by thinking about how you'd want a computer to make a choice for what a good move would be.

At any one point in time, the computer can look at the board and see that it has a few possible moves. Each of those moves will lead to a new board, and then it's the opponent's turn. A not-very-intelligent computer would do something like choose the first open column, or pick a column at random, or even just pick the single move that looked best without considering what the opponent might do.

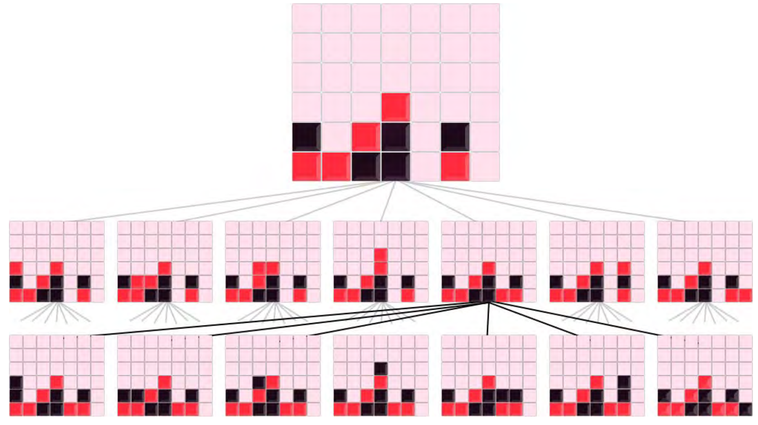
But a slightly smarter computer would not only consider what move looked best for a solo player, but then what move the opponent could make after that. So, if the computer made possible move number 1, then the opponent would have some set of possible moves to make. If the computer assumes the opponent makes the best of those moves, then the computer can reasonably think: "If I make move number 1, then here is where I'll be at the next point I can make a move." Similarly, the computer could evaluate what the opponent's moves might be for every possible move along the way.

For Connect 4, there are 7 columns, so there are typically 7 moves possible at each turn. So, if a computer had to evaluate each move it could make and each move an opponent could make, that would be 7 times 7, or 49, different moves.

An even more intelligent computer would look more than just 2 moves ahead—it would look 3 or 4 or more moves ahead! And the best game-playing AIs will do exactly that: They'll look several moves ahead. But notice how quickly the number of possibilities increases. In Connect 4, for every one further move ahead a computer looks, there are 7 times as many possibilities to consider! Looking 6 moves ahead would be 7^6, or more than 100,000, possibilities!

One of the advantages of C++ is that it tends to result in really efficient code, and the practical effect of this is that a C++ program might be able to look a little further ahead, in the same amount of time, compared with a program in some other language. And looking even one more level ahead can be all the difference in which AI is best.

You can take this basic idea of looking ahead and turn it into an algorithm for an AI playing a computer game. The algorithm will be a generic, abstract description of what to do, and it will be turned into a function, or other code, that is an actual, concrete implementation of the algorithm. So, you'll start by writing the algorithm in pseudocode and then turn it into actual C++ code.



Notice that this outlined structure follows the definition of a tree—the same type used in top-down design. The root of the tree is the current gameboard, and each node's children are the possible moves.

// DEVELOPING ALGORITHMS

To develop the algorithm, you'll follow an approach that is similar to the top-down design approach, but you want to try to keep the problem as narrow as possible.

The purpose of the algorithm is to let a computer determine the best move it can make in a game. With that goal in mind, you start by considering what your input is: What is known at each point during the game? So, the input will be the current state of the game: the current gameboard and an indication of which player's turn it is.

Next, you consider the output you want from the algorithm. Clearly, you want to output a move, the best one the algorithm could determine.

Specifying the input and output clearly is an important part of algorithm design—just as important as the steps of the algorithm itself. Algorithms are often implemented as functions; the steps of the algorithm become the code in the function. You can approach this algorithm design in a similar way that you would approach top-down program design, looking at the overall challenge and breaking it into simpler parts.

If you're faced with making a choice, you have 3 basic steps:

1. Make a list of possible valid moves. For Connect 4, each player has at most 7 moves—any column that's not full.
2. Evaluate each of those moves to get some measure of how good or bad each one is.
3. Determine which of those moves is best and return the result.

The trickiest part will be the middle step of evaluating each move. That'll have 2 parts: first try a move (that is, update the gameboard to

see what it would look like if that move was made), and then determine how good or bad of a state that move puts you in.

The idea here is that you want to think ahead as many moves as possible. You could generate all those possible moves at once, but that becomes really excessive in terms of memory you'd need. Instead, you'll treat the board as one of 2 cases: Either you don't want to look ahead any more moves and just want to see how good the current board position is, or you want to generate possible moves and choose what's best among them.

In the event that you're willing to look forward a little more, then you need to start this whole process again but from the opponent's point of view; that is, you want to go through this same algorithm but now starting from the opponent's moves.

This brings up the idea of **recursion**, which can be thought of as a function calling itself or an algorithm using itself to solve a problem. And that's what you're going to do here. You're going to determine the best move by making a move and then determining the best move.

Recursion needs to have a stopping point or it could go on forever, taking up too much time or computer memory. The ending condition you'll use here is to keep track of how many levels of recursion you've had; that is, you want to think about how many moves ahead you want the computer to consider. So, you're going to take a new input: the level of recursion you're at. Every time you make a recursive call—that is, every time you consider the best move—you'll do so for one more level of recursion than you were at previously. And if you reach your maximum, then you'll stop making recursive calls. Once you've reached the maximum level of recursion, you want to just evaluate a board to get a score.

When you are going to evaluate several different moves via recursion, it's not enough to just know which move is the next one to take—you also want to know how good or not that move is. So, you'll want to return not just the best move, but also the score for that move; in other words, your return information has increased. You use this score that you get from the recursive call to rank the moves and determine which one is best.

Evaluating a board to get a score—that is, to determine whether player 1 or player 2 is winning and by how much—can be a tricky problem that requires an algorithm of its own. What follows is a very simple algorithm, but it still turns out to perform OK!

In Connect 4, players take alternate turns, and no player develops an advantage in number of pieces.

Here's your board evaluation:

- If the game is tied, then the value should be 0.
- If player 1 seems to be winning, then you'll give a negative score—the more negative, the more that player 1 is winning.
- If player 2 seems to be winning, you'll give a positive score: The larger the score is, the more player 2 has an advantage.
- If one player or the other actually wins, you should have a really negative or really positive score.

The input to this algorithm is just the current board state. The output of the algorithm will be a score for that board state. If the board indicates that player 1 has won—that is, if player 1 gets 4 of his or her pieces in a row, called a connect 4—then you return a very small number, like −1000. If the board indicates that player 2 has won, then you return a very large number, like 1000. If you know that the game has ended in a tie, then you evaluate the board as a 0.

In your winner-or-loser evaluation function, if you don't have a winning board, then you'll just give a random value to the board. Let's say that's between a range of −5 to 5. This is basically just saying "make a random move." Notice that this is still an algorithm, but the only information it's using from the board is whether or not the game is over. But it's actually enough for your program to work.

// FROM ALGORITHMS TO IMPLEMENTATION

The full program is available for download at www.TheGreatCourses.com/CPlusPlus.

To go from algorithms to implementation, you begin by modifying the Connect 4 classes that were developed in lecture 21. A few changes were made to simplify things: There's only one game—Connect 4—and there's no inheritance or virtual functions in this version.

There are 3 main classes:

- **game**, which has member functions to start a game, take a turn, and declare a winner.
- **connect4_board**, which stores a Connect 4 board, including any pieces already in place. This class can check for a winner, print the board to the screen, check whether columns are filled, and update itself.
- **connect4_move**, which is a basic class that holds a move: the column to place a piece in and which player is placing that piece. It has some basic accessor functions to set and read moves or to ask a player to enter a move.

And 2 small changes were made to make this program an AI program:

- You no longer ask which game to play, and instead, when starting the game, you ask whether the computer should be player 1 or player 2, keeping this information as a private variable.
- Then, when taking a turn, if it's the player's turn, you get the move from the player, as usual. But if it's the computer's turn, then you will call the **get_best_move** function, which will return a pair: the score for the best move and which column the best move is in. So, you just take the second element of the pair and make the move there.

So, there are 2 algorithms that need to be implemented: a board evaluation algorithm and **get_best_move**. These are the core parts of the AI algorithm in the code. Because the board evaluation algorithm will want to make use of board information, it makes sense for it to be a member function of the **connect4_board** class. Meanwhile, **get_best_move** doesn't need any information about the board—that's what the board evaluation function does—so it should be a standalone function.

Here's one way to implement a **get_board_score** function. First, notice that your routine has access to the board state because it's a member function of the **connect4_board** class. Thus, it doesn't have to bring in the board as a parameter; it already has access to it. It will return an integer, the score for the board. You already had a routine, **check_winner**, that would indicate whether the game was won by player 1 or player 2, or was a tie, or was none of those. So, you first call that function to see if the board had a winner.

```
   // Returns negative value for player 1 advantage,
positive for player 2 advantage
   int get_board_score() {
      int winner = check_winner();
      if (winner == 1) return -1000;
      if (winner == 2) return 1000;
      if (winner == 3) return 0;
      // No one has won yet, so see who is "closer"
      return rand()%11 - 5;
   }
```

If the game was won by player 1 or player 2 or was a tie, then you return either **-1000**, **1000**, or **0**, respectively.

If the game is not over, then you return a random value between −5 and 5, as the algorithm described. To do this, you can generate a random number from 0 to 10 and subtract 5.

Remember that to generate a number from 0 to 10, you use the **rand** function to generate a random integer and take it modulo 11.

The random value you assign just means that boards where no one has won are evaluated randomly. But the fact that the algorithm does identify winning and losing positions means that it still provides value to the look-ahead process of finding the best move.

In other words, even though you have a board evaluation function that only evaluates whether someone has won or not, it will still be enough to give you a decent AI. And it gives you something to improve later!

Next, you need to implement the **get_best_move** function, using the algorithm described previously.

The snippet at right shows one way to implement it. Notice that it takes in all the algorithm input as parameters: the person whose turn it is, the current board state, and the current depth of recursion.

Instead of the recursion levels increasing, you'll instead be counting them down. So, the initial call would set some level of recursion that's wanted—5 or 6 is reasonable—and then each time a new move is generated, you decrease that number by 1 until you're at level 0.

Your function follows the same form you outlined in the algorithm pseudocode. You first make a list of all moves. To do this, you create a vector of pairs: Each pair will have a score for the move and then which column it is in. Initially, you just go through all 7 columns on the board and check if each is full. If it's not full, then you add that column as a potential move, though you don't have a score evaluation for that move yet.

```
pair<int, int> get_best_move(int player_to_move, connect4_board b, int
depthtogo) {
   int i;
   // Make list of moves
   vector<pair<int, int> > possible_moves;
   for (i = 0; i < 7; i++) {
      if (!b.isfullcol(i)) {
         // This is a possible move
         possible_moves.push_back(make_pair(0, i));
      }
   }
   // Evaluate each move
   for (auto& m : possible_moves) {
      connect4_board newboard = b;
      connect4_move nextmove;
      pair<int, int> mv;
      nextmove.set_move(player_to_move, m.second);
      newboard.take_turn(nextmove);
      if ((depthtogo == 0) || (newboard.check_winner())) {
         // Don't look ahead any more - just see how good this board is by itself
         m.first = newboard.get_board_score();
      }
      else {
         // Look ahead another level
         mv = get_best_move(3-player_to_move, newboard, depthtogo - 1);
         m.first = mv.first;
      }
   }
```

The next section of code evaluates the moves. You go through the list of moves you just identified. Notice that you're using an iterator to go through the list; the value **m** will take on each of the moves. You can see it's a reference because of the **&**, and because it's a reference, it's referring to the actual move.

For that move, you create a board, starting with the old one, which was the parameter **b**, and then take a turn using that particular column.

Then, for each move, you have one of 2 cases:

- If you've hit the limits of your recursion—that is, if the depth parameter is now 0 or if you've reached a winning condition for one player or the other—then you just set the board score.
- If you need to continue evaluating moves, you can make a recursive call. To do this, you call the same function you're in: **get_best_move**. The player number needs to switch. Notice that the next player to move is just 3 minus the current **player_to_move**, so if the current player is 1, the next player is 2, and vice versa. So, you just pass in **3-player_to_move** to get the next player number.

  You also pass in the new board that you generated after making this move that you're wanting to evaluate. Finally, you pass in a recursion depth level, 1 less than whatever came in. Notice that what is returned is a pair that includes the best score you could get from that next possible move, so you associate that returned score with the current move. You don't care about the other part of the recursion—the actual move that would yield that score. The only time that matters is at the very end, when you get your final move out.

The final part of your algorithm is to find the best move and return. To find the best move, you'll use the **sort** algorithm from the STL to sort all of your moves from smallest to largest. Because a move is a pair, the first part of which is the score, that means the STL algorithm will sort all possible moves from the lowest score to the highest.

Remember that for player 1, a low board score means that he or she is winning. So, if the current player is player 1, then the player would like to choose the move that gives the lowest score. Thus, after sorting, you just return the first element of the vector. On the other hand, if it's player 2's turn, then you want the largest score, so you return the last element of the vector.

If you run this, you'll find that the AI is surprisingly competent. You'll almost certainly still be able to beat it, but it'll make sure to try to block you from getting a Connect 4, and it'll grab an opportunity to get one itself if the chance is available. And it looks ahead several moves, so if it sees a combination of moves that could help it win, or prevent you from winning, it'll make those moves.

```
   // Select best move
   sort(possible_moves.begin(), possible_moves.end());
   if (player_to_move == 1) {
      // Player 1's move, so want smallest value
      return possible_moves[0];
   }
   else {
      // Player 2's move, so largest score is best
      return possible_moves[possible_moves.size() - 1];
   }
}
```

// IMPROVING YOUR ALGORITHMS

You could try to improve on these algorithms. The board evaluation function is key to how well the AI performs, so that's an obvious candidate for improvement, because all you've done so far is identify whether there's a winner and otherwise make a random move.

One option for improvement would be to look for all cases in the program where one player or the other is getting close to a connect 4.

For example, you might look at all lines of 4 in the board in every direction—vertical, horizontal, and diagonal—similar to the routines to check for winners, but check whether one player has 3 of the 4 pieces needed for that option and the fourth space needed is still open. That means that the player has a chance to get a connect 4 if he or she can get that one space. So, give the player 100 points. In other words, if it's player 1 who has 3 of 4, then subtract 100 points from the board score, and if it's player 2, then add 100 points to the board score.

And you could do the same thing with even fewer pieces, looking for places where a player has 2 spaces needed and the other 2 are empty. In this case, they're a little farther away, so maybe just add or subtract 10 points.

Adding this gives you an AI that's now awfully hard to beat! ◆

```
int check_vertical_advantage() {
   /* Check the board to see if either player has 2
or 3 of 4 needed */
   int i, j, k;
   int advantage = 0;
   for (i = 0; i < columns; i++) {
      // Can loop over all but last 3 rows, comparing
4 at a time
      for (j = 0; j < rows - 3; j++) {
         int count1 = 0;
         int count2 = 0;
         for (k = 0; k < 4; k++) {
            if (board[i][j + k] == 1) {
               count1++;
            }
            else if (board[i][j + k] == 2) {
               count2++;
            }
         }
         if ((count1 == 3) && (count2 == 0)) {
            advantage -= 100;
         }
         else if ((count1 == 0) && (count2 == 3)) {
            advantage += 100;
         }
         else if ((count1 == 2) && (count2 == 0)) {
            advantage -= 10;
         }
         else if ((count1 == 0) && (count2 == 2)) {
            advantage += 10;
         }
      }
   }
   return advantage;
}
```

WHAT'S NEXT?

There are several directions you could go to take your programming skills to the next level.

- You could learn more about algorithms and data structures in general.
- You could explore a particular application area, including learning one of the many libraries that give you tools to approach applications.
- You could use your foundation of C++ to learn almost any other language you might want, such as C#, Java, or JavaScript.

READINGS

a Stroustrup, *Programming Principles and Practice Using C++*, chap. 20.

b Lippman, Lajoie, and Moo, *C++ Primer*, sections 3.4 and 9.1–9.3.

// QUIZ

1 Lecture 21 presented a program that would allow the game Reversi (Othello) to be played. Now imagine that you wanted to build an AI for Othello, similar to the one you built for Connect 4. The process of looking ahead by several moves would be no different for the Reversi game. However, you would need a different board evaluation function as well as different functions to list the available moves.

For the board evaluation function for a Reversi game:

a What would be a straightforward board evaluation for Reversi? Remember, you want a computation that would give a low score if player 1 is winning and a high score if player 2 is winning.

b Write the board evaluation function that you would use. The name should be **`get_board_score`**, and it would return an integer. It should be a member of the **`reversi_board`** class:

```
class reversi_board {
  private:
  vector<vector<int> > board;
  // Other code including board
evaluation function here
};
```

Recall that the board is 8 by 8 in size and that the elements are **`0`** if there is no piece on the square or **`1`** or **`2`** if player 1 or 2 has his or her piece on the square.

Note: As an exercise, you might want to try on your own to adapt the AI for playing Connect 4 to instead play Reversi. You will need to update the earlier Reversi game functions similar to the way the Connect 4 functions were updated and introduce new routines for finding all valid moves and the board evaluation.

Click here to see the answer.

// QUIZ ANSWER

1 a For a board evaluation function, an easy method is to just take the number of player 2's pieces and subtract the number of player 1's pieces. When player 2 is doing better (has more pieces), the score will be higher, and when player 2 is doing worse (has fewer pieces), the score will be lower.

Note that there are other ways you could improve board evaluation if you know game strategy well. For example, corners are generally more valuable than other squares, because they cannot be flipped. And edge squares, especially those 2 away from a corner, are usually slightly more valuable to hold than other squares.

b Here is one possible implementation. Note that this would be a public member of the **reversi_board** class. So, it will have access to the **board** member variable.

```
int get_board_score() {
  int i, j;
  int totalscore = 0;
  for(i=0;i<8;i++) {
    for(j=0;j<8;j++) {
      if (board[i][j] == 1) {
        totalscore -= 1;
      } else if (board[i][j] == 2) {
        totalscore += 1;
      }
    }
  }
  return totalscore;
}
```

The function just loops over all the squares of the board and keeps track of a counter that is decreased for each occurrence of player 1's piece and increased for each occurrence of player 2's piece.

Click here to go back to the quiz.

Glossary

abstract class

A class that contains a pure virtual function. It defines a general category of classes from which other classes can be derived. Instances of an abstract class cannot be created. (L20)

accessor function

Provides an approved channel to read information from a class. Allows other functions to access private member variables. Also provides a useful place to set a breakpoint when debugging. (L16)

algorithm

A precise set of rules and steps to follow to accomplish some task. Often described by pseudocode, which is then converted into actual code. (L22, L23, L24)

append

To add on to the end of an existing item. Appending is commonly used when writing to files, to add additional data to the end of the file, or to add one string to the end of another. (L9, L10)

arguments

Values given in a function call; each argument parallels a **parameter** inside the function itself, and the parameter takes on the value of the argument within the function. (L12)

array

A contiguous block of memory containing several variables of the same type. The array is allocated at once, and individual variables (**elements**) are accessed by an **index**. (L7) The more modern and C++ approach is to use a **vector**.

array out-of-bounds error

A bug arising from attempting to access an element of an array that is outside the block of memory allocated. This happens when the array index is negative or larger than the maximum size of the array. (L7) This can be a major security flaw.

artificial intelligence

Programming a computer to act as though it is making intelligent choices and decisions. (L24)

associative container

A container that associates 2 or more data types together. Examples include pairs, tuples, and maps. (L23)

attribute

See **member variable**.

binary search

A process for searching through a sorted list of values by examining the halfway point and then searching through one of the 2 remaining halves. (L23) Contrast with **iterative search**.

binary tree

A data tree where every node has at most 2 children. (L18)

Boolean

A true or false value used in conditions and logical expressions. A Boolean can be a specific true or false **value**; a Boolean variable can take on a true or false value; a Boolean **expression** can evaluate to true or false using **logical operators**. (L3)

Boolean operator

See **logical operator**.

bottom-up design

Creating larger, more complex programs by combining simpler pieces. Often, new functions are created by combining calls to existing functions. (L15)

breakpoint

When debugging, a point at which execution of the program stops. (L11)

bug

A mistake in a program that causes it to fail to compile, crash when running, or produce incorrect output. Debugging a program is a common part of programming. (L4, L14)

C++ Standard Library

A standard set of around 50 library files included with every C++ installation. Libraries used in this course include, among others, **`iostream`**, **`cmath`**, **`cstdlib`**, **`ctime`**, **`vector`**, **`string`**, **`fstream`**, **`stringstream`**, and **`algorithm`**. (L6)

call (aka function call)

A command that causes a function to be executed using any arguments specified. Functions are called by writing their name followed by parentheses that contain any arguments to give to the function. (L6)

casting a value

See **type casting**.

class

A programming structure that combines data (**member variables**) with operations (**member functions**). Each class is like a new type of variable defined by the programmer. (L16)

comparison operator

An operator for comparing 2 different values, the result of which is a Boolean. Common comparisons are greater than (`>`), greater than or equal to (`>=`), less than (`<`), less than or equal to (`<=`), equal to (`==`), and not equal to (`!=`). (L3)

compile

The process of turning code into machine instructions. Code is compiled either into debug mode or release mode. (L4) See also **preprocessor commands** and **separate compilation**.

concatenation

Combining 2 strings one after the other; the result of addition of strings. (L9)

concrete class

A class that provides implementation details for all defined functions; you can create objects from concrete classes. (L20)

console

The default source for input and output of a program, reached with the command **cout**. (L1)

constant reference

See **passing by reference**.

constructor

A function, defined within a class, called to initialize a variable (**object**) when it is first declared. (L8, L10, L17)

container

A templated class that can be used to store and manipulate data of various types. Examples include vectors, lists, queues, and priority queues. The STL defines several containers. (L22)

debug

To remove bugs from a program. This can be done systematically by identifying a repeatable error, narrowing in on the location of the bug, isolating the reason for the failure, fixing the bug, retesting the fixed code, and examining any similar parts of code. (L4, L11, L14)

debug mode

A compiler setting that adds in extra machine instructions that enable a debugger to be run on the code. (L11) Contrast with **release mode**.

debugger

A tool that helps analyze a program, allowing a programmer to step through a program line by line and examine the memory while the program is running. (L1, L11)

declaration of a variable

Statement declaring a variable type, followed by a variable name. This allocates memory for the variable and associates the variable name with that part of memory. (L2)

default constructor

A constructor that does not take parameters. The default constructor is used when an object is declared with no parameters. If a class does not define a default constructor, one will be automatically created for it. (L17)

default parameter

When a function's parameter is given a value to take in case the function is called without a corresponding argument for that parameter. Only the rightmost parameter(s) in the list can be default. (L13)

dereferencing a pointer

Accessing the thing stored at the memory address kept in a pointer. Syntax is `*y` or `y[0]` (for an object) or `y->` (for an object's member variable or function). (L18)

destructor

A special function that is called when it is time for a function or class to be removed from memory. (L18).

dynamic memory allocation

Allocating new memory while the program is running in a location known as the **heap** or the free store (L18). Dynamic memory allocation is useful when the memory needed might not be known in advance. Contrast with **static memory allocation**.

element

One particular variable within an array or vector of variables. An array or vector will usually consist of many elements, which can be accessed by an index into the array. (L7)

encapsulation

An object-oriented programming approach for wrapping all related data and functions together in one package. A **class** is a way of encapsulating **member functions** and **member variables**. (L16)

error

Any problem encountered when writing or executing a program. This can include bugs like syntax errors and logic errors as well as runtime errors (such as array out-of-bounds errors) that can be caught with exceptions. (L14)

exception

Causes a function to exit when an exceptional case, typically a runtime error, is encountered. Referred to as "throwing an exception" or "raising an exception." (L14; L19)

executable file

A file of machine instructions created by a **linker** that has all the needed information from program files and from library files and thus can be run on its own. (L6)

execute

To cause a computer to perform some set of machine instructions; can refer to a program, a function, or any part of code. (L1)

expression

A combination of literals, variables, and function calls combined with operators that evaluates to some value. (L2)

friend function/operator

Workaround to give a function or operator defined outside a class access to the member variables defined inside that class. Works by placing a signature for the function inside the class definition. (L17)

function

A group of operations and commands that can be executed as a group; a way of conceptually separating one set of functionality from everything outside. Functions can be called within other parts of a program. Defining a function requires a **function header** and a **function body**. (L6, L12) See also **member function** (including **accessor function** and **mutator function**) and **stub function**.

function body

The set of commands that is executed when a function is called. The body is defined within curly braces following the function header. (L12)

function call

See **call**.

function header

Contains all of the information needed for the black box of the function to interact with the larger program: the return type, the name of the function, and the parameter list (contained in parentheses). (L12)

function overloading

Using the same function name but taking different parameters to get different behavior. (L13) Compare with operator overloading. An alternative to function overloading is using **default parameters**.

generic programming

See **template**.

global variable

A variable whose scope is the whole program. Global variables are generally discouraged, because understanding their value requires understanding the entire program, rather than just part of it. (L12) Contrast with **local variable**.

header file

A file, usually with a **`.h`** extension, that describes the interface for a library. Header files are included (via **`#include`**) in order to gain access to the functions, classes, and variables provided by a library. (L6, L15)

heap

Memory that holds data not known ahead of time but assigned by **dynamic memory allocation** during execution (L18). Contrast with **stack**.

identifier

See **variable name**.

index

An integer that identifies a particular element of an array or a vector. The first element has index 0, the second element has index 1, and so on. (L7)

indexing into an array/vector

Accessing elements of the array/vector using a variable that contains the index of an element. This is done by specifying the array/vector name, followed by square brackets containing the index of the desired element. (L7, L8)

information hiding

A programming technique for managing complexity of code in which details of how a function (or class) works is hidden from the portion of code using that function (or class). The code calling a function is unaware of how the function itself works, and a function is unaware of how it is being used by the code calling it. (L16, L19)

inheritance

A hierarchical approach in object-oriented programming to allow new offspring classes (aka child classes, derived classes, or subclasses) to make use of (aka inherit) member variables or functions already defined in an existing ancestor class (aka parent class, base class, or superclass). (L19)

initialize

Specifying a value for a variable or object when it is first declared. Objects can be initialized using a constructor. (L2)

integrated development environment (IDE)

A computer program that makes developing code easier. This typically includes a text editor to write code, a graphical interface to allow code to be compiled and executed easily, a debugger, and other tools. Free IDEs for C++ include Microsoft's Visual Studio Community and Apple's Xcode. (L1, L11)

iterative search

A process for searching through a list of values by examining each element one by one. The list is generally not in sorted order. (L23)

iterator

A more general form of an index. Iterators are a general way of accessing elements in general containers, which might not have an integer index. (L22)

library

A set of classes, functions, and variables written externally to a program. A library can be accessed by including its header file in a program. (L6) See also **C++ Standard Library**.

linker

A program run automatically after the compiler that combines the machine instructions from a compiled program with machine instructions from libraries. (L6)

list

A data structure in which items are kept in order one after the other. Items can be inserted or removed in the middle of a list efficiently but cannot be accessed by an index. (L22)

literal

A specific value written in code. Literals can be assigned to variables or can be used in an expression. (L9) Contrast with **variable**.

local variable

A variable whose scope is the current function. It can be accessed within that function but not outside it. (L12) Contrast with **global variable**.

logical operator (aka Boolean operator)

An operator used to perform logical operations on Booleans, such as **and** (`&&`), **or** (`||`), and **not** (`!`). (L3)

loop

A programming construct for repeating a set of commands. A `while` loop repeats as long as a condition is true. A `for` loop makes it easier to understand how the loop works by putting initialization, condition, and update all in one place. (L5)

map (aka dictionary)

An associative container that pairs a key, or entry, with a definition, or other value. (L23)

member function (aka method or operation)

A function defined as part of a class. Can be public, private, or protected. (L8; L16)

member variable (aka attribute)

A variable defined as part of a class. Can be public, private, or protected. (L16; L19)

memory leak

An error in programs caused by dynamically allocating memory in a way that reassigns a pointer but leaves the memory allocated with no remaining way to access or free it. (L18)

modulus operation

Gives the remainder after division. Expressed using the `%` operator. (L2, L6)

multidimensional array

An array of arrays. (L7)

mutator function

A member function of a class that can be used to modify a member variable. Mutator functions can provide good places to set breakpoints during debugging. (L16)

namespace

A way of helping you distinguish one library's functions from another's, even if both libraries use the same function names. (L15)

object

A particular instance of a **class**. An object is effectively a variable whose type is the class. An object contains variables and functions that belong to it. (L10, L16)

object-oriented programming (OOP)

A method of software development that is centered on creating classes and objects. Principles of encapsulation, inheritance, and polymorphism are commonly used. (L1, L16, L17, L20)

operator

A symbol used to perform an operation on one or 2 literals or variables. A unary operator (such as `-`, `++`, or `!`) will operate on just one element. A binary operator (such as `+`, `-`, `*`, or `/`) will have 2 elements, or operands: one just before the operator and one immediately afterward. (L2, L17)

operator overloading

Method for defining different behavior of operators depending on the types of the operands. For example, `+` can mean addition for numbers or concatenation for strings. Operators can be overloaded to define behavior for new classes. (L9, L17). Compare with **function overloading**.

output

The result of a program, generally as indicated by what is printed in the console. (L1)

overloading

Providing different implementations of a function or operator depending on the type(s) it is working with. (L9, L13, L17) See also **function overloading** and **operator overloading**.

pair

An associative container that allows 2 different data types to be easily combined into a single structure. (L23)

parallel arrays

A method of keeping track of multiple values for the same entity by using multiple arrays or vectors. The index indicates which element is referred to, and the values for that entity can be determined by indexing into each of the arrays using the same index. (L7) An alternative to using a **class**.

parameter

Values passed into a function when called. Parameters are specified in the function header and behave like variables within the function. Each parameter corresponds to one argument, and the corresponding argument provides the initial value of the parameter. (L12)

passing by reference

When a function is called, the parameter becomes a reference to the actual argument; no copying of values is performed. Changing the parameter value thus changes the value in the argument itself (L13). You have to give a variable argument, not a specific value—unless passing by constant reference, which can be a reference to a literal or a variable but does not allow any modification.

passing by value

When a function is called, making a copy of the argument and setting the parameter to that value. (L13)

pointer

A variable type that is not actually storing a value but, rather, the memory location at which some value is stored. The pointer thus points to an actual object. The type of variable or object being pointed to must be stated when the pointer is declared. Pointers must be dereferenced to access the values (or functions) in the actual variable or object. (L18) A pointer is similar to a **reference**, but pointers can change values (point to new memory locations) and must be dereferenced. A pointer can let you treat anything in an inheritance hierarchy as an example of the base type.

polymorphism

Object-oriented programming technique for creating member functions that can operate differently for each subclass yet all be called in the same way. In this way, a superclass can take on many different shapes, depending on which subclass implementation is used. (L20) See also **inheritance**.

preprocessor commands

Commands to be performed before code is actually compiled. Includes importing header files or defining values. (L4)

print

To send output to the console or any other output device. (L1)

priority queue

A data structure used to order items based on some priority. Items are pushed into the queue, and the highest-priority item can be popped from the front of the queue. (L23)

private member variable/function

A function or variable defined within a class or struct that is accessible only in that class and not in any subclasses or outside the class. This is the default for members of a class. (L16)

procedural programming

A form of programming in which computation is divided into a number of functions (also called procedures), each of which is like a small program of its own, performing a particular task. (L1) Contrast with **object-oriented programming**.

protected member variable/function

A function or variable defined within a class or struct that is accessible in that class or any subclasses but not outside the class hierarchy. (L19)

pseudocode

A summary of the general steps that should be taken in a program or algorithm. Pseudocode is not code in a programming language, but a programmer can easily convert it to actual code. (L4)

pseudorandom

Generating values that seem random, though are actually being generated through some nonrandom process. Often a (perhaps unknowable) seed value is used to begin the pseudorandom generation process. (L6) Available via the `<cstdlib>` library in the C++ Standard Library.

public member variable/function

A function or variable defined within a class or struct that is accessible anywhere, including to commands outside the class. This is the default for members of a struct. (L16)

pure virtual function

A virtual function that is not defined in the base class. A class containing a pure virtual function is an abstract class and cannot be instantiated. Pure virtual functions must have an implementation defined in a subclass. (L20)

queue

A data structure supporting first-in-first-out behavior. Items are pushed on the back of the queue and popped off the front. (L22)

recursion

When a function calls itself. (L18, L24)

reference

A variable name that is used to refer to some other variable. A reference is a pointer that does not have to be explicitly dereferenced. References are most commonly seen when passing parameter values by reference, where a function's parameter names are used to refer to the actual variable specified in the argument. (L13)

release mode

A compiler setting that produces more-efficient code; debuggers cannot be used on code compiled in release mode. (L11) Contrast with **debug mode**.

return

A value that a function computes and is returned to the place where the function was called, replacing the function call in the calling location. Each function has a return type, specifying the type of the value that is returned. A function that simply performs actions without producing a value that needs to replace the function call does not need to return anything, and the return type `void` is used. Referred to as "returning a value." (L12).

scaffold

A portion of code used to set up testing of a function. The scaffold does not need to be a complete program but just needs to set up variables and data so that a function can be called. (L15) Compare with **stub function**.

scope

The range of a program in which a variable is defined. Variables are said to be either "in scope" or "out of scope." (L5, L12)

sentinel value

An unusual value so far outside the range of expected values that its appearance can signal the program that it is time for a loop or other process to end. Sentinel values are used when a number of iterations is not known or there is not another way to predict or otherwise test when input or data should end. (L5)

separate compilation

Breaking a program into different parts (e.g., a header file and a source file), each of which is compiled on its own. The complete program requires linking the separately compiled parts together. This is used to simplify individual files or to enable reuse of code, as in a library. (L15)

signature

The information needed for a function to be called. This includes not only the function name but also the number and types of parameters. Function signatures must be unique and are how the compiler determines which function should be executed by a function call. (L13)

stack

1. Memory that holds all of the variables you declare and all those you pass as parameters. (L18) Contrast with **heap**.

2. A data structure that supports last-in-first-out behavior. Items are pushed on top of the stack and popped off the top. (L22)

Standard Template Library (STL)

A library provided by default with C++ that contains several commonly used templated data structures and algorithms. (L22, L23)

static memory allocation

Memory that is known will be needed in a function or program at the time the program is written and is set aside in a part of memory known as the stack. The memory is automatically allocated when a program is run or a function is called. (L18) Contrast with **dynamic memory allocation**.

static variable

A variable declared in a way that it persists (within a loop or function) beyond when it would otherwise fall out of scope. (L5)

stream

A source of input or a destination for output. Streams can include the console, files, or strings. (L1, L10)

stream operator

An operator that is used to direct input from a stream (`>>`) or output to a stream (`<<`) (L1, L2, L10)

string

An ordered grouping or collection of characters—letters, numbers, punctuation, spaces, etc. Strings can be either specific values, known as string literals and specified within quotation marks, or variables of the `string` type, accessed in C++ via `#include <string>`. (L9)

struct

A container, equivalent to a **class**, for packaging data together. Developed in C, structs had only public member variables. In C++, structs are identical to classes, except that the default is that all members are public. Short for *structure*. (L16)

stub function

A short implementation of a function that does not work correctly but can be used for testing a larger program before the function has been fully implemented. Stub functions can be called and should return some valid result so that code containing a function call can be tested and debugged. (L15) Compare with **scaffold**.

template

A C++ technique for implementing generic programming. Templates allow a general class, data structure, or function to be defined, where the specific type of an element of the data structure or a parameter of the function is determined when the variable is declared or the function is called. For example, a vector is a templated data structure, and the specific type of a vector is determined when the vector is declared (e.g., `vector<int>` for a vector of integers). (L8, L22, L23)

test case

A specific set of input and expected output for a function, program, or portion of code. Test cases are used to determine if code is working as expected and are an important part of debugging. (L4) See also **unit test**.

`try-catch` blocks

A set of programming commands used to handle exceptions. Code that should be run but that may have exceptions is contained in a `try` block, and then one or more `catch` blocks are used to determine what happens for various types of exceptions. (L14)

tuple

An associative data structure that allows multiple values of various types to be easily combined into a single structure. (L23)

type

The way that information in memory should be interpreted. Each variable or value in a program has a particular type. In C++, types for variables must be specified when they are declared and cannot change. Common types are `int` (integer), `float` (floating-point number), `bool` (Boolean), and `string`, but a type can be any class. (L2)

type casting (aka casting a value)

Converting one variable type into another. This can be done by specifying the new type in parentheses in front of the variable or value or by stating the new type and the value or variable to convert in parentheses. (L5)

Unified Modeling Language (UML)

Developed in the 1990s and standardized in 2005 to aid the design of object-oriented software. UML uses a standardized set of notation and graphical indicators to describe class structures, their relationships, and a wide variety of other program behavior. (L21)

unit test

A set of test cases for a particular function used to determine if it is implemented correctly. Consists of input parameters and expected return values. (L14)

value

A specific number or state. Can be expressed as a literal or can be the result of an expression or the current memory in a variable. Contrast with **variable**. See also **passing by value**.

variable

A box of memory defined by a type (such as `int`) and a name (aka an identifier). Variables can take on different values throughout a program. (L2) See also **declaration of a variable**, **global variable**, and **local variable**.

variable name (aka identifier)

The term used while programming to refer to a box of memory that is a variable. Good names help programmers identify the purpose of a variable. (L2) Compare with **signature**.

vector

A templated data structure that is commonly used to store numerous values of the same type. Elements of a vector can be accessed using an index. Vectors are similar to arrays but can grow in size. (L8) Compare with **array**.

virtual function

Object-oriented programming technique for defining functions in a base class that can be defined differently in each derived class. (L20) See also **pure virtual function**.

C++ Syntax

This short intro to C++ syntax is only a starting point for identifying the meaning for various symbols, keywords, and common commands.

// SYMBOLS

Symbols in C++ often have more than one predefined meaning (see table below), including some meanings not listed here. In addition, many symbols can be given additional meanings by users (see **overloading** in the Glossary).

Comments and Preprocessor Commands

// **comment**

Remaining line; all characters are ignored until a new line is encountered in the code.

/* **begin comment**

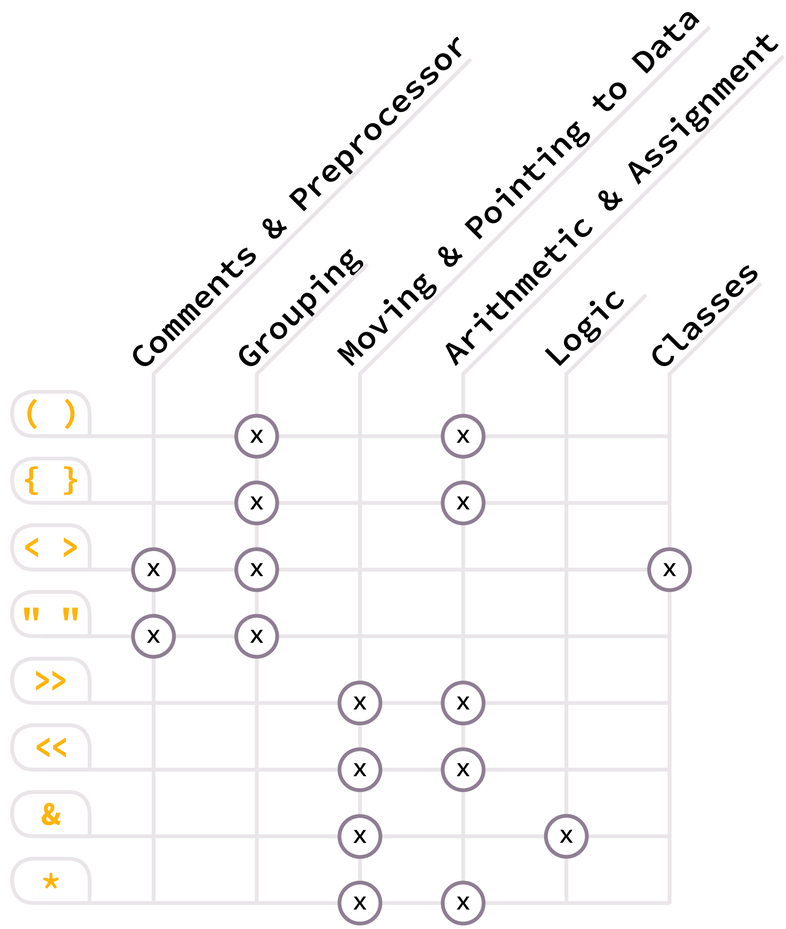
All characters are ignored until an end comment (***/**) is encountered.

***/** **end comment**

Used to end a comment that was begun with **/***.

**pound**

Indicates that the line is an instruction to the preprocessor to do something before compiling, rather than a line of C++ code to be compiled. Common uses are **`#include`** and **`#define`**.



Grouping Symbols

; semicolon

Designates the end of a statement of code; similar to a period in ordinary English.

( ) parentheses

1. Used to contain lists of arguments in function calls. Must always appear after the function name when calling a function, even if there are no arguments (e.g., `f()`);

2. Surround the list of parameters in a function header (e.g., `int f(int x, int y)`);

3. Surround the Boolean used in conditionals and the loop conditions in loops (e.g., `if (x>2)`);

4. Used to show precedence of operations in an expression (as in mathematics) (e.g., `3*(4+5)`);

5. Contains a type when typecasting a variable (e.g., to convert an integer variable `x` to a floating-point variable, you can write `(float) x`).

{ } curly braces

1. Encloses a block of code, grouping the commands inside of it. This is used to denote the extents of a function body, a loop body, the results from a conditional, a class definition, `try-catch` blocks, etc.

2. Specifies a set of indexed values for initializing a vector or array (e.g., `int x[4] = {10, 20, 30, 40}`).

< > angle brackets

1. With the `#include` preprocessor command, encloses the names of libraries to use from the C++ Standard Library (e.g., `#include<iostream>`);

2. When defining templates, used to specify the types that the template will take on (e.g., `vector<int> v`).

" " quotes

1. Used to enclose a string literal, including when it specifies information such as a file name (e.g., "This is a string.").

2. With the `#include` preprocessor command, encloses the name of a library or user-created file to bring in (e.g., `#include "myclass.h"`).

' ' single quotes

Used to enclose a single character (e.g., `'a'` or `'\n'`).

Moving Data and Pointing to Data

>>

1. Stream operator used to direct data from an input stream;
2. Right-shift operator for numerical data that shifts bits to the right.

<<

1. Stream operator used to direct data to an output stream;
2. Left-shift operator for numerical data that shifts bits to the left.

= **equal sign**

The assignment operator. Assigns the value on the right side to the variable on the left. Note that this does not compare values for equality.

[ ] **square brackets**

1. Used when declaring an array to designate the size of an array (e.g., `int x[100]`) or in an array parameter to show that the parameter will be an array (e.g., `int f(int[] a, int n)`);
2. Used to provide an index to an element of an array, vector, or similar container (e.g., `a[3]`).

& **ampersand**

1. Designates a variable or parameter as a reference. Used when passing by reference, (e.g. `void f(int &x)` indicates parameter `x` is passed by reference).
2. Generates a pointer to a particular value or variable (e.g., `&x` is a pointer to the variable `x`, and if `x` is an integer, it can be assigned to a pointer variable whose type is a pointer to an integer (i.e., `int*`)).
3. A bitwise **and** operation. This operates on a bit-by-bit basis and is not the same as the logical **and** operation that is typically used in conditionals.

Arithmetic and Assignment

+ plus

1. Addition operator for numerical data;
2. Concatenation operator for strings.

− minus

Subtraction or negation operator for numerical data.

* asterisk

1. Multiplication operator for numerical data;
2. When declaring a pointer, it is placed after the type of the thing being pointed to (e.g., `int* x` declares the variable `x` to be a pointer to an integer;
3. For pointers, it dereferences the pointer to obtain the value at the memory location stored in the pointer (e.g., `*x` is the object/variable/value at the memory location indicated by the pointer `x`).

/ (forward) slash

Division operator for numerical data. For integer data types, this gives the quotient of the result without the remainder.

% percent sign

Modulus operator. For integer data, this gives the remainder from division.

++ increment operator

Causes an integer value to increase by 1.

−− decrement operator

Causes an integer value to decrease by 1.

+= increase operation

Causes the variable on the left to have the `+` operation applied to itself with the value on the right (e.g., `x += 3` increases the value of `x` by `3`).

−=, *=, /=, %=, etc.

Variations of the `+=` operator but applying other operations.

Comparison Operators

> greater-than comparison operator

< less-than comparison operator

>= greater-than-or-equal-to comparison operator

(Note that => is not valid.)

<= less-than-or-equal-to comparison operator

(Note that =< is not valid.)

== equality operator

An operator that compares 2 values for equality. The operator is true if the 2 values are the same.

!= inequality operator

An operator that compares 2 values for inequality. The operator is true if the 2 values are different.

Logic Operators

! exclamation point

A logical **not** operation. Converts a Boolean value to the opposite value.

&& logical **and** operator

True if both operands are true; false otherwise.

| vertical bar

A bitwise **or** operation. This operates on a bit-by-bit basis and is not the same as the logical **or** operation that is typically used in conditionals.

|| logical **or** operator

True if either (or both) operands are true; false otherwise.

Special Characters

\ backslash

An escape character that is used to specify special characters. The character following the backslash indicates what the character should be (e.g., `\n` indicates a new-line character, and `\t` indicates a tab character; `\\` indicates a single backslash character, `\"` indicates a quotation mark, and `\'` indicates a single quote).

Operators for Classes and Objects

. period

Access a member variable or function (e.g., `x.y` will be the member variable `y` in object `x`, and `x.y()` will be the member function `y` in object `x`).

-> dereference and access member variable or function

For a pointer, the object at the location has its member function called (e.g., `x->y()` will call the member function `y()` for the object at location `x`, and `x->y` will refer to the member variable `y` from the object at memory location `x`).

~ tilde

Used to designate the destructor when defining a class.

// PREDEFINED KEYWORDS

A keyword is a word defined in the C++ language itself. These are often automatically recognized by your IDE and presented to you in a color distinct from the surrounding code. The list here includes C++ keywords that are commonly used but is not complete (see the Predefined Variable Types list below, for example). For a more complete list, consult an online reference such as cppreference.com.

`break`

When encountered inside of a loop, immediately stops the execution of the loop and jumps to the first statement following the loop.

`catch`

Designates the commands to be executed in response to an exception generated in a preceding `try` block.

`class`

Used to define a class. The general format is `class class_name {/* class description */};`.

`const`

Indicates that a variable will not change values; particularly useful for passing by const reference.

`continue`

When encountered inside of a loop, immediately stops the execution of that iteration of the loop and begins the next iteration (if any).

`delete`

Frees up dynamically allocated memory that was previously allocated with **new**.

`else`

Designates the commands to follow if an `if` condition evaluates to false.

`extern`

Indicates that the definition for a function or class will be provided outside of the current file.

`for`

Loop designator that provides a compact way of specifying all loop control in one place. The keyword is followed by parentheses containing 3 clauses separated by semicolons: an initialization clause executed before the first iteration, a conditional to check at the beginning of each possible iteration, and an update clause to indicate what should change with each iteration. Alternatively, the parentheses can contain variable and container to iterate over, separated by a colon; the variable will take on each value from the container.

`friend`

Designates that a function defined outside a class should have access to a class's members as though it were part of the class.

`if`

Identifies the start of a condition. The keyword is followed by parentheses containing a Boolean (the condition), and following that are any commands to execute if the condition is true and, optionally, an **`else`** clause.

`namespace`

Designates a namespace to be used when creating classes or functions or, when coupled with **`using`**, when referring to a class or function.

`new`

Allocates new memory dynamically (on the heap).

`operator`

Used to designate an operator that will be overloaded.

`private`

Indicates that subsequent class member variables and functions should be private (accessible within that class only).

`protected`

Indicates that subsequent class member variables and functions should be protected (accessible to a class and its derived classes only).

`public`

Indicates that subsequent class member variables and functions should be public (accessible to anything).

`return`

Specifies the value to return from a function.

`static`

Indicates a variable that should persist across multiple calls to a function. Static variables will maintain their value from the previous call to a function on subsequent calls rather than allocating a new local variable with each function call.

`struct`

An alternative to a class. Traditionally used only for grouping member variables, it now provides the same functionality as a class.

`try`

Designates a section of code that should be executed but that might generate exceptions. Exceptions are handled by code in a subsequent **`catch`** block.

`typedef`

Defines a new name to be used for a type; allows a user to use more compact and meaningful type names.

`using`

Indicates a namespace to be used by default so that namespaces do not need to be specified explicitly for every command.

`virtual`

Indicates that a function in a class can be defined more specifically in derived (children) classes.

`void`

A type indicating no information; commonly used to specify the return type for functions that do not need to return a value.

`while`

Loop designator. The keyword is followed by a conditional in parentheses and then by commands to be executed repeatedly, as long as the conditional evaluates to true at the beginning of the loop.

// PREDEFINED COMMANDS

Predefined commands are commonly used commands (e.g., as part of the C++ Standard Library) that are not actually part of the C++ language itself.

`begin`

A member function of a container returning an iterator pointing to the first element in a container. Calling **`.begin()`** on the container will return the iterator.

`cin`

A source (obtained by using **`#include<iostream>`**) for streaming console input (input typed in by the user).

`cout`

A destination (obtained by using **`#include<iostream>`**) for streaming console output (output to be printed in the console window).

`end`

A member function of a container returning an iterator pointing to the position just after the last element of a container. Calling **`.end()`** will return the iterator. Typically, you use this by incrementing the iterator until it equals **`.end()`**, indicating that you have reached the end of the container.

`eof`

Member function for finding whether you have encountered the end of a file. Only when you have tried to read something past the end of the file will **`.eof()`** return true.

`failbit`

A particular exception that is thrown when a file is opened incorrectly.

`find`

An algorithm (obtained by using **`#include<algorithm>`**) for performing a linear search through a container (such as an array or a vector). It takes in a beginning and ending iterator and an element value to look for. It returns an iterator to the first position where a matching element is found or the ending iterator if nothing is found. Likewise, there is a **`.find()`** member function of the string class for finding a substring within a string (the substring is the only parameter, returning **`string::npos`** if nothing is found).

`getline`

A function that takes 2 parameters: an input stream and a string variable. A line (until an end-of-line character, such as new line: `\n`) is read in from the input stream and stored in the string variable.

`lower_bound`

An algorithm (obtained by using `#include<algorithm>`) for performing a binary search on a sorted array or vector. It takes in a beginning and ending iterator and an element value to look for. It returns an iterator to the first item that is greater than or equal to the item being searched for.

`sort`

An algorithm (obtained by using `#include<algorithm>`) for sorting vectors, arrays, or other containers. It takes in a beginning and ending iterator and sorts the elements from the beginning one to just before the ending one.

`string::npos`

A value used to compare to in order to find the end of a string. When searching in a string, if an item being searched for is not found, the location will be reported as `string::npos`, indicating that it was not within the string.

// PREDEFINED VARIABLE TYPES

These are some of the most commonly used data types. Some are built-in C++ keywords, and others are part of the C++ Standard Library and must be included to be used. You can also define your own types using classes.

`auto`

Automatically determines the data type for the variable based on how the variable is first assigned a value. This is useful for avoiding long type descriptions when the type of the variable will be obvious, but in most cases, it is better to explicitly declare individual types. C++ keyword.

`bool`

Boolean. The value 0 is false; any other number is interpreted as true. C++ keyword.

`char`

A single character. Can also be interpreted as a very small integer. Pronounced "char" or "care." C++ keyword.

`double`

Floating-point decimal number that has double the precision of the float. C++ keyword.

`float`

Floating-point decimal number (standard length). C++ keyword.

`fstream`

A file stream, used for reading from or writing to files. Must `#include<fstream>`.

`int`

Integer. Any non-decimal number over the range ... –2, –1, 0, 1, 2 ... . C++ keyword.

`long` or `long long`

Integer using more precision and thus capable of representing larger numbers than standard `int`. C++ keyword.

`string`

A sequence of characters strung together. Must `#include<string>`.

`stringstream`

A string stream, used for reading from a string or writing to a string, similar to how you read and write to the console or to files. Must `#include<sstream>`.

`void`

A null data type, usually used to indicate that a function has no return value or as a way of describing a generic pointer, where the type of the data stored in memory is not known. C++ keyword.

// CONTAINER TYPES

Containers are data structures that hold several variables of the same type. Each container supports certain types of operations on its elements. They are included in the C++ Standard Template Library (STL). Each of these is a templated class and is declared by specifying one or more variable types that it will contain. These are some of the most commonly used containers, but the STL contains several more. Each of these containers is accessed using some `#include` command for the preprocessor.

`deque<type>`

Doubly ended queue. Can easily add to or remove from beginning or end. Must `#include<deque>`.

`forward_list<type>`

Stores data in a (singly linked) list. Compared to a list, it is somewhat more efficient for most operations, but it is highly inefficient to access the end of the list or to go backward through the list. Must `#include<forward_list>`.

`list<type>`

Stores data in a (doubly linked) list. Easy to add or delete at any point in the list, but no indexing is supported. Must `#include<list>`.

`map<type1, type2>`

Stores key-value data pairs. The first type is the key and is used as a unique index for storing data; the second type is the value that is associated with each particular key. Must `#include<map>`.

`pair<type1, type2>`

Stores pairs of 2 data elements. Provides a simple way of grouping 2 data values of different types without creating a new class. Can `#include<utility>`, but this is automatically included in most other STL libraries, too.

`priority_queue<type>`

Allows data to be inserted and the largest element to be pulled out. The type is often a pair or other user-defined class. Must `#include<queue>`.

`queue<type>`

Supports adding elements and pulling out elements in the order they were added. Must `#include<queue>`.

`stack<type>`

Supports adding elements and pulling out the most recently added element. Must `#include<stack>`.

`tuple<type1, type2, ..., typeN>`

Stores data in a tuple, allowing a simple structure for combining an arbitrary number of data elements of possibly different types in a single structure without creating a new class. Must `#include<tuple>`.

`unordered_map<type1, type2>`

A version of the map that does not keep the elements in any particular order. More efficient than the map for operations but cannot give an ordered list of all elements. Must `#include<unordered_map>`.

`vector<type>`

Stores data in a sequence that can be directly indexed and can be added on to at the end. Must `#include<vector>`.

Bibliography

A variety of C++ books are available, but many of them assume that the reader is already familiar with some other programming language, and few are truly aimed at novice programmers. C++ books tend to fall into 2 categories. Some will basically present C programming and then present the additional features that C++ provides as an add-on. A few books will present a C++-centric view of programming and develop all of the material but skip some of the more C-like features that C++ still provides. The references recommended here fall into the second category, while this course as a whole aims to fall in between these 2 ends of the spectrum.

In addition to books that tend to describe how to use C++, there are also language references, which can be considered as being closer to a dictionary or encyclopedia and are useful to look up details of language syntax and behavior, standard library features, etc. Although there are printed language references, for general purposes, the online sources work just as well and just as easily and tend to stay very current.

The first 2 references in the following list are both recommended, and while both may move somewhat quickly for novice programmers, they are both designed to be useful for those first learning to program. The third book is a recommendation for those who already know programming and just want an idea of the particular way C++ is used. Next are 2 online references, which are mainly recommended for language reference. Finally, a book on general software design is recommended, though it will be most useful to those already familiar with programming.

Stroustrup, Bjarne. *Programming Principles and Practice Using C++*. 2nd ed. Addison-Wesley Professional, 2014.

> This is a book written to be a college textbook by the creator of C++. It gives a thorough coverage of the language and includes information about software development, not just the language itself. It is best used if you are willing to read through entire chapters in a linear order, rather than jumping to a particular topic, because some of the topics build on earlier discussions and sometimes earlier code is intentionally presented with errors or inadvisable approaches that are fixed later.

Lippman, Stanley B., Josée Lajoie, and Barbara Moo. *C++ Primer*. 5th ed. Addison-Wesley Professional, 2012.

> This book provides an extensive coverage of C++, including many smaller details of the language. It provides suggested exercises in most sections to help you practice the topics covered. Compared to the previous book, it tends to have greater focus on the details of how C++ works and less emphasis on general programming methodology. Note that this edition does not cover C++ features beyond C++11, but most programmers will not need the features of the newer releases of C++.

Stroustrup, Bjarne. *A Tour of C++*. 2nd ed. Addison-Wesley Professional, 2018.

> This is a very short book that gives a brief overview of the features and syntax of C++. It will be valuable to those who are already familiar with another programming language and are interested in learning about some of the highlights and key ideas in C++. It is not a complete reference, nor is it a good book for someone learning to program, but will be helpful to experienced programmers.

www.cplusplus.com.

> This website includes a good language reference; a reference for all the standard C++ libraries, including the STL; and some more basic tutorials and articles. There is also an online discussion forum.

en.cppreference.com.

> This website is just a language and standard library reference, but it is very extensive and complete. It includes features from the most recent editions of C++ as well as planned future releases of C++ (clearly labeled).

Ousterhout, John. *A Philosophy of Software Design*. Yaknyam Press, 2018.

> This book describes the process of software design without regard to one specific language and mostly without reference to any particular paradigm. It is filled with extremely useful advice and is highly recommended for those who are experienced programmers. Novice programmers or those with little experience may not find the suggestions contained in it as meaningful but can still get wise advice about design from the book.