How to think like a computer scientist

Allen B. Downey

C++ Version, First Edition

Augmented by Peter Walsh for CSCI 160
Computing Science Department
Malaspina University-College
Fall 2005

How to think like a computer scientist

C++ Version, First Edition

The LaTeX source for this book, and more information about the Open Source Textbook project, is available from

`http://www.cs.colby.edu/~downey/ost`

or by writing to Allen B. Downey, 5850 Mayflower Hill, Waterville, ME 04901.

The GNU General Public License is available from www.gnu.org or by writing to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

This book was typeset by the author using LaTeX and dvips, which are both free, open-source programs.

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem solving**. By that, I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

1.1 What is a programming language?

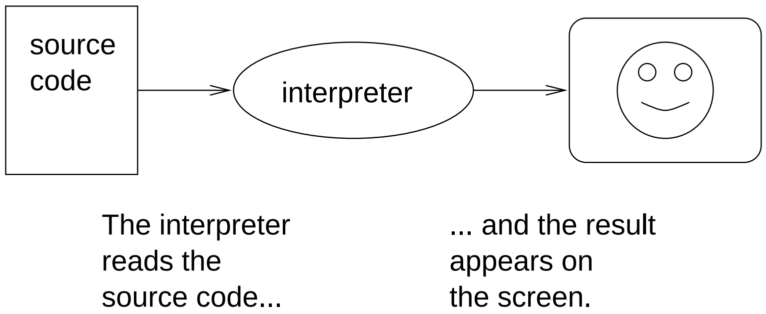
In CSCI 160, we use the **high level programming language** C++. Other high-level languages you might have heard of are Java, C and FORTRAN.

As you might infer from the name "high-level language," there are also **low level languages**. Loosely-speaking, computers can only execute programs written in machine language (the lowest of the low-level languages). Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is *much* easier to program in a high-level language; by "easier" I mean that the program takes less time to write, it's shorter and easier to read, and it's more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few special applications.

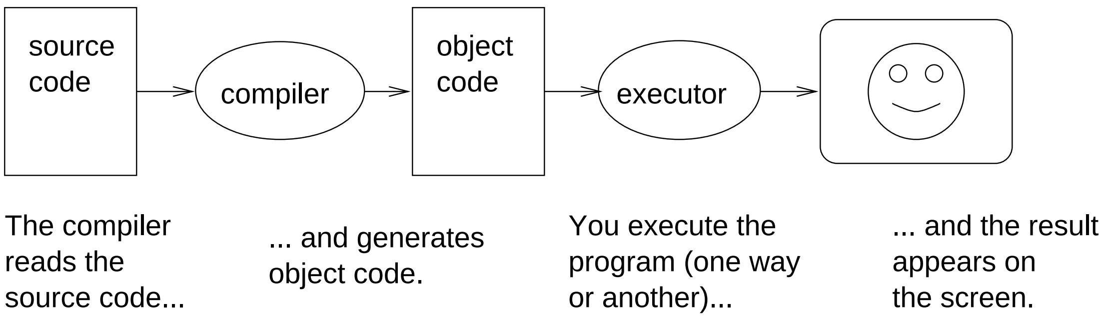
There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.



A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

As an example, suppose you write a program in C++. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named `hello.C`, where "hello" is an arbitrary name you make up, and the suffix `.C` is a convention that indicates that the file contains C++ source code.

Then, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named `hello.o` to contain the object code, or `hello` (no suffix) to contain the executable.



The next step is to run the program, which requires some kind of executor. The role of the executor is to load the program (copy it from disk into memory) and make the computer start executing the program.

Although this process may seem complicated, the good news is that in most programming environments (sometimes called development environments), these steps are automated for you. Usually you will only have to write a program and type a single command to compile and run it. On the other hand, it is useful to know what the steps are that are happening in the background, so that if something goes wrong you can figure out what it is.

In CSCI 160, we work in a Unix environment. You are free to choose from a number of different text editors. We will use the GNU C++ compiler `g++` and the Unix program development utility `make`.

1.2 What is a program?

A program is a sequence of instructions (or commands, or statements) that when executed, reads input data and writes output information. To this end, all programs have three fundamental components *viz.*, an input-output component, a control component and a data processing component. The input-output component is responsible for reading the input data and writing the output information. The control component dictates what data transformations are necessary to convert the input data into the output information. These three fundamental components can be found in all programs no matter what the programming language employed. They can be further characterized as follows:

- **Input Output**

  input get data from the keyboard, or a file, or some other device

  output put information on the screen or send information to a file or other device

- **Data Processing**

  arithmetic perform operations like addition and subtraction

  logical perform operations like logical AND and logical OR

- **Control**

  sequence execute a sequence of instructions

  decision check for certain conditions and execute the appropriate sequence of instructions

  iteration execute a sequence of instructions repeatedly

The critical thing to remember is that all programs, no matter how complex, are controlled using combinations of sequence, decision and iteration. Master the use of these three constructs and you are well on your way to conqueroring CSCI 160.

1.3 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.3.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in C++ than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program.

For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

1.3.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you should acquire from working with this book is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. Conan Doyle's *The Sign of Four*).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (from *The Linux Users' Guide* Beta Version 1).

In later chapters I will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

> **Programming languages are formal languages that have been designed to express computations**

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3+3=6$ is a syntactically correct mathematical statement, but $3 = +6\$$ is not. Also, H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with `3=+6$` is that `$` is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement `3=+6$` is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax and semantics—there are many differences.

ambiguity Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally the first program people write in a new language is called "Hello, World." because all it does is print the words "Hello, World." In C++, this program looks like this:

```
// File: hello.C
// Author: Peter Walsh CSCI 160 Fall 2005
// Behaviour: Outputs "Hello, World." to the screen (stdout)

#include <iostream>
using namespace std;

int main()
{
        cout << "Hello, World." << endl;
        return 0;
}
```

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, C++ does reasonably well. Even so, this simple program contains several features that are hard to explain to beginning programmers. For now, we will attempt to gain a high level understanding of the program and gloss over some of the more "difficult" details.

The lines that begin with `//` are comments. A comment is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a `//`, it ignores everything from there until the end of the line.

When the program runs, it starts by executing the first statement in `main` and it continues, in order, until it gets to the last statement, and then it quits.

There is no limit to the number of statements that can be in `main`, but the example contains only one. It is a basic **output** statement, meaning that it outputs or displays a message on the screen.

`cout` is a special object provided by the system to allow you to send output to the screen. The symbol `<<` is an **operator** that you apply to `cout` and a string, and that causes the string to be displayed.

endl is a special symbol that represents the end of a line. When you send an **endl** to **cout**, it causes the cursor to move to the next line of the display. The next time you output something, the new text appears on the next line.

Like all statements, the output statement ends with a semi-colon (;).

There are a few other things you should notice about the syntax of this program. First, C++ uses squiggly-braces ({ and }) to group things together. In this case, the output statement is enclosed in squiggly-braces, indicating that it is *inside* the definition of **main**. Also, notice that the statement is indented, which helps to show visually which lines are inside the definition.

As was previously mentioned, the C++ compiler is a real stickler for syntax. If you make any errors when you type in the program, chances are that it will not compile successfully. For example, if you misspell **iostream**, you might get an error message like the following:

```
hello.C:5:20: oistream: No such file or directory
hello.C: In function 'int main()':
hello.C:10: 'cout' undeclared (first use this function)
hello.C:10: (Each undeclared identifier is reported only
             once for each function it appears in.)
hello.C:10: 'endl' undeclared (first use this function)
```

There is a lot of information in this message, but it is presented in a dense format that is not easy to interpret. A more friendly compiler might say something like:

> "On line 5 of the source code file named hello.C, you tried to include a header file named oistream.h. I didn't find anything with that name, but I did find something named iostream.h. Is that what you meant, by any chance?"

Unfortunately, few compilers are so accommodating. The compiler is not really very smart, and in most cases the error message you get will be only a hint about what is wrong. It will take some time to gain facility at interpreting compiler messages.

Nevertheless, the compiler can be a useful tool for learning the syntax rules of a language.

1.6 Program verification

Once you have written and debugged a program, how would you know if the program is *correct*? We will refer to a "correct" program as being verified. Program verification is the task of establishing that program is implemented to specification. This begs the question, who writes and determines that the specification is correct? Where CSCI 160 is concerned, we will assume that the specifications (written in natural language) are correct and unambiguous. Consequently, our focus is on program verification rather that specification verification.

Program verification is composed of inspection and and testing. Inspection is based on peer review by small teams of programmers and is widely used in industry. In essence, one team member argues that the code is implemented to specification (or otherwise).

Program testing is where the program is executed with test input and the output is checked for correctness (as given in the specification).

E. W. Dijkstra noted that "Testing can only be used to show the presence of bugs, but never their absence." (from Structured Programming *Proc. Conf. NATO Science Committee, 1969* . Consequently, we can not use testing alone to guarantee that a program is correct. The best we can do is argue that a program is implemented to specification using testing and inspection in combination.

In CSCI 160, we will inspect programs for certain correctness criteria and we will use a testing tool that allows us to apply test cases to programs and automatically check for correct output.

1.7 What is computer programming?

The art and science of computer programming is fundamentally concerned with problem solving. An algorithm is a sequence of precise instructions that solves a problem. Problems-solving is programming-language independent and algorithms can be stated in natural-language using combinations of sequence, decision and iteration. This is very good news since it means you can develop algorithms that solve assignment problems without knowing any programming language. Once you have developed an algorithm, it is relatively easy to implement it in a high-level programming language like C++.

With regard to CSCI 160, initially we will focus on C++ syntax/semantic issues and implement simple algorithms. Later in the term, we will work on more complex problems and consequently, more complex algorithms.

> **It is critical you seperate algorithm development from code development else, you will eventually become overwhelmed with the complexity of dealing with them both at the same time.**

1.8 What is software engineering

Solo programming is where one programmer designes and develops small programs (less that 5000 lines). Large software systems are designed and developed by teams of computer, business, artistic and scientific professionals. Managing large projects involves many more organizational difficulties than the small single-person projects. Software engineering is the study of processes by which large projects can be designed and developed. Software engineering is characterized by multi-person multi-version programming.

In CSCI 160, you will develop programs on you own. With an eye to the future, we will employ techniques from software engineering where appropriate. Consequently, you will primarily be engaged in solo programming but in a software engineering context.

1.9 Glossary

problemsolving The process of formulating a problem, finding a solution, and expressing the solution.

highlevel language A programming language like C++ that is designed to be easy for humans to read and write.

lowlevel language A programming language that is designed to be easy for a computer to execute. Also called "machine language" or "assembly language."

portability A property of a program that can run on more than one kind of computer.

formal language Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language Any of the languages people speak that have evolved naturally.

interpret To execute a program in a high-level language by translating it one line at a time.

compile To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code A program in a high-level language, before being compiled.

object code The output of the compiler, after translating the program.

executable Another name for object code that is ready to be executed.

algorithm A general process for solving a category of problems.

bug An error in a program.

syntax The structure of a program.

semantics The meaning of a program.

parse To examine a program and analyze the syntactic structure.

syntax error An error in a program that makes it impossible to parse (and therefore impossible to compile).

runtime error An error in a program that makes it fail at run-time.

logical error An error in a program that makes it do something other than what the programmer intended.

debugging The process of finding and removing any of the three kinds of errors.

software engineering Multi-person multi-version programming.

Chapter 2

Software Design Methodology

There are many different ways to develop a "correct program". Some programmers can think and type at the same time. They simply develop algorithms in their head and translate them into working programs as they type. For the novice programmer, this approach almost always leads to disaster eventually. A more systematic approach greatly improves you chances of success. For CSCI 160, we will employ a five step methodology [1] as follows:

- **Problem analysis** determine the input and output for the program and record it in a I/O chart.
- **Modular design** divide the problem into sub-problems (modules) and record your design in a structure chart.
- **Detailed module design** flesh-out the design of each module in psudo-code.
- **Code production** translate the pseudo code to C++ code.
- **Verication**

  Testing pseudo code perform walk through tests on your pseudo-code and record the tests in a test-set.

  Testing production code augment the test-set to include implementation test-cases and trace the execution of the production code on the tests from the test-set.

  Testing executable code apply the tests from the test-set to the executable code and determine if the program's output is correct.

[1] It is worth noting that not all steps are required all the time. Conversely, on occasion we may need to add further steps to achieve some stated goal.

Inspection develop, record and inspect for production-code inspection-criteria.

Chapter 3

Variables and types

3.1 More output

As I mentioned in the last chapter, you can put as many statements as you want in `main`. For example, to output more than one line:

```
#include <iostream.h>

// main: generate some simple output

void main ()
{
  cout << "Hello, world." << endl;     // output one line
  cout << "How are you?" << endl;      // output another
}
```

As you can see, it is legal to put comments at the end of a line, as well as on a line by themselves.

The phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

Often it is useful to display the output from multiple output statements all on one line. You can do this by leaving out the first `endl`:

```
void main ()
{
  cout << "Goodbye, ";
  cout << "cruel world!" << endl;
}
```

In this case the output appears on a single line as `Goodbye, cruel world!`. Notice that there is a space between the word "Goodbye," and the second quotation mark. This space appears in the output, so it affects the behavior of the program.

Spaces that appear outside of quotation marks generally do not affect the behavior of the program. For example, I could have written:

```
void main ()
{
cout<<"Goodbye, ";
cout<<"cruel world!"<<endl;
}
```

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program's behavior either, so I could have written:

```
void main(){cout<<"Goodbye, ";cout<<"cruel world!"<<endl;}
```

That would work, too, although you have probably noticed that the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate syntax errors.

3.2 Values

A value is one of the fundamental things—like a letter or a number—that a program manipulates. The only values we have manipulated so far are the string values we have been outputting, like `"Hello, world."`. You (and the compiler) can identify string values because they are enclosed in quotation marks.

There are other kinds of values, including integers and characters. An integer is a whole number like 1 or 17. You can output integer values the same way you output strings:

```
  cout << 17 << endl;
```

A character value is a letter or digit or punctuation mark enclosed in single quotes, like `'a'` or `'5'`. You can output character values the same way:

```
  cout << '}' << endl;
```

This example outputs a single close squiggly-brace on a line by itself.

It is easy to confuse different types of values, like `"5"`, `'5'` and `5`, but if you pay attention to the punctuation, it should be clear that the first is a string, the second is a character and the third is an integer. The reason this distinction is important should become clear soon.

3.3 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a named location that stores a value.

Just as there are different types of values (integer, character, etc.), there are different types of variables. When you create a new variable, you have to declare what type it is. For example, the character type in C++ is called `char`. The following statement creates a new variable named `fred` that has type `char`.

```
char fred;
```

This kind of statement is called a **declaration**.

The type of a variable determines what kind of values it can store. A `char` variable can contain characters, and it should come as no surprise that `int` variables can store integers.

There are several types in C++ that can store string values, but we are going to skip that for now (see Chapter).

To create an integer variable, the syntax is

```
int bob;
```

where `bob` is the arbitrary name you made up for the variable. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
char firstLetter;
char lastLetter;
int hour, minute;
```

you could probably make a good guess at what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `second` are both integers (`int` type).

3.4 Assignment

Now that we have created some variables, we would like to store values in them. We do that with an **assignment statement**.

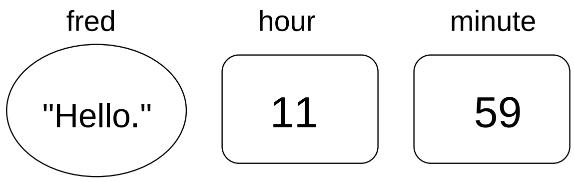
```
firstLetter = 'a';   // give firstLetter the value 'a'
hour = 11;           // assign the value 11 to hour
minute = 59;         // set minute to 59
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.

- When you make an assignment to a variable, you give it a value.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This kind of figure is called a **state diagram** because is shows what state each of the variables is in (you can think of it as the variable's "state of mind"). This diagram shows the effect of the three assignment statements:



I sometimes use different shapes to indicate different variable types. These shapes should help remind you that one of the rules in C++ is that a variable has to have the same type as the value you assign it. For example, you cannot store a string in an `int` variable. The following statement generates a compiler error.

```
int hour;
hour = "Hello.";       // WRONG !!
```

This rule is sometimes a source of confusion, because there are many ways that you can convert values from one type to another, and C++ sometimes converts things automatically. But for now you should remember that as a general rule variables and values have the same type, and we'll talk about special cases later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, the string `"123"`, which is made up of the characters `1`, `2` and `3`, is not the same thing as the *number* `123`. This assignment is illegal:

```
minute = "59";         // WRONG!
```

3.5 Outputting variables

You can output the value of a variable using the same commands we used to output simple values.

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

cout << "The current time is ";
```

```
cout << hour;
cout << colon;
cout << minute;
cout << endl;
```

This program creates two integer variables named **hour** and **minute**, and a character variable named **colon**. It assigns appropriate values to each of the variables and then uses a series of output statements to generate the following:

```
The current time is 11:59
```

When we talk about "outputting a variable," we mean outputting the *value* of the variable. To output the *name* of a variable, you have to put it in quotes. For example: `cout << "hour";`

As we have seen before, you can include more than one value in a single output statement, which can make the previous program more concise:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

cout << "The current time is " << hour << colon << minute << endl;
```

On one line, this program outputs a string, two integers, a character, and the special value `endl`. Very impressive!

3.6 Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are reserved in C++ because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include `int`, `char`, `void`, `endl` and many more.

The complete list of keywords is included in the C++ Standard, which is the official language definition adopted by the the International Organization for Standardization (ISO) on September 1, 1998. You can download a copy electronically from

    http://www.ansi.org/

Rather than memorize the list, I would suggest that you take advantage of a feature provided in many development environments: code highlighting. As you type, different parts of your program should appear in different colors. For example, keywords might be blue, strings red, and other code black. If you type a variable name and it turns blue, watch out! You might get some strange behavior from the compiler.

3.7 Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in C++ do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is `+`.

The following are all legal C++ expressions whose meaning is more or less obvious:

```
1+1         hour-1        hour*60 + minute     minute/60
```

Expressions can contain both variables names and integer values. In each case the name of the variable is replaced with its value before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
  int hour, minute;
  hour = 11;
  minute = 59;
  cout << "Number of minutes since midnight: ";
  cout << hour*60 + minute << endl;
  cout << "Fraction of the hour that has passed: ";
  cout << minute/60 << endl;
```

would generate the following output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is what we expected, but the second line is odd. The value of the variable `minute` is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that C++ is performing **integer division**.

When both of the **operands** are integers (operands are the things operators operate on), the result must also be an integer, and by definition integer division always rounds *down*, even in cases like this where the next integer is so close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
  cout << "Percentage of the hour that has passed: ";
  cout << minute*100/60 << endl;
```

The result is:

```
Percentage of the hour that has passed: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values. We'll get to that in the next chapter.

3.8 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but just to get you started:

- Multiplication and division happen before addition and subtraction. So `2*3-1` yields 5, not 4, and `2/3-1` yields -1, not 1 (remember that in integer division `2/3` is 0).
- If the operators have the same precedence they are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding `5900/60`, which in turn yields `98`. If the operations had gone from right to left, the result would be `59*1` which is `59`, which is wrong.
- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so `2 * (3-1)` is 4. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

3.9 Operators for characters

Interestingly, the same mathematical operations that work on integers also work on characters. For example,

```
char letter;
letter = 'a' + 1;
cout << letter << endl;
```

outputs the letter `b`. Although it is syntactically legal to multiply characters, it is almost never useful to do it.

Earlier I said that you can only assign integer values to integer variables and character values to character variables, but that is not completely true. In some cases, C++ converts automatically between types. For example, the following is legal.

```
int number;
number = 'a';
cout << number << endl;
```

The result is 97, which is the number that is used internally by C++ to represent the letter `'a'`. However, it is generally a good idea to treat characters as characters, and integers as integers, and only convert from one to the other if there is a good reason.

Automatic type conversion is an example of a common problem in designing a programming language, which is that there is a conflict between **formalism**, which is the requirement that formal languages should have simple rules with few exceptions, and **convenience**, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers, who are spared from rigorous but unwieldy formalism, but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

3.10 Composition

So far we have looked at the elements of a programming language—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to multiply integers and we know how to output values; it turns out we can do both at the same time:

```
cout << 17 * 3;
```

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the output, but the point is that any expression, involving numbers, characters, and variables, can be used inside an output statement. We've already seen one example:

```
cout << hour*60 + minute << endl;
```

You can also put arbitrary expressions on the right-hand side of an assignment statement:

```
int percentage;
percentage = (minute * 100) / 60;
```

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

WARNING: There are limits on where you can use certain expressions; most notably, the left-hand side of an assignment statement has to be a *variable* name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values. So the following is illegal: `minute+1 = hour;`.

3.11 Glossary

variable A named storage location for values. All variables have a type, which determines which values it can store.

value A letter, or number, or other thing that can be stored in a variable.

type A set of values. The types we have seen are integers (`int` in C++) and characters (`char` in C++).

keyword A reserved word that is used by the compiler to parse programs. Examples we have seen include `int`, `void` and `endl`.

statement A line of code that represents a command or action. So far, the statements we have seen are declarations, assignments, and output statements.

declaration A statement that creates a new variable and determines its type.

assignment A statement that assigns a value to a variable.

expression A combination of variables, operators and values that represents a single result value. Expressions also have types, as determined by their operators and operands.

operator A special symbol that represents a simple computation like addition or multiplication.

operand One of the values on which an operator operates.

precedence The order in which operations are evaluated.

composition The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

Chapter 4

Function

4.1 Floating-point

In the last chapter we had some problems dealing with numbers that were not integers. We worked around the problem by measuring percentages instead of fractions, but a more general solution is to use floating-point numbers, which can represent fractions as well as integers. In C++, there are two floating-point types, called `float` and `double`. In this book we will use `double`s exclusively.

You can create floating-point variables and assign values to them using the same syntax we used for the other types. For example:

```
double pi;
pi = 3.14159;
```

It is also legal to declare a variable and assign a value to it at the same time:

```
int x = 1;
char quit = 'q';
double pi = 3.14159;
```

In fact, this syntax is quite common. A combined declaration and assignment is sometimes called an **initialization**.

Although floating-point numbers are useful, they are often a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value `1`, is that an integer, a floating-point number, or both?

Strictly speaking, C++ distinguishes the integer value `1` from the floating-point value `1.0`, even though they seem to be the same number. They belong to different types, and strictly speaking, you are not allowed to make assignments between types. For example, the following is illegal

```
int x = 1.1;
```

because the variable on the left is an `int` and the value on the right is a `double`. But it is easy to forget this rule, especially because there are places where C++ automatically converts from one type to another. For example,

```
double y = 1;
```

should technically not be legal, but C++ allows it by converting the `int` to a `double` automatically. This leniency is convenient, but it can cause problems; for example:

```
double y = 1 / 3;
```

You might expect the variable `y` to be given the value `0.333333`, which is a legal floating-point value, but in fact it will get the value `0.0`. The reason is that the expression on the right appears to be the ratio of two integers, so C++ does *integer* division, which yields the integer value `0`. Converted to floating-point, the result is `0.0`.

One way to solve this problem (once you figure out what it is) is to make the right-hand side a floating-point expression:

```
double y = 1.0 / 3.0;
```

This sets `y` to `0.333333`, as expected.

All the operations we have seen—addition, subtraction, multiplication, and division—work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

4.2 Converting from `double` to `int`

As I mentioned, C++ converts `int`s to `double`s automatically if necessary, because no information is lost in the translation. On the other hand, going from a `double` to an `int` requires rounding off. C++ doesn't perform this operation automatically, in order to make sure that you, as the programmer, are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **typecast**. Typecasting is so called because it allows you to take a value that belongs to one type and "cast" it into another type (in the sense of molding or reforming, not throwing).

The syntax for typecasting is like the syntax for a function call. For example:

```
double pi = 3.14159;
int x = int (pi);
```

The `int` function returns an integer, so `x` gets the value 3. Converting to an integer always rounds down, even if the fraction part is 0.99999999.

For every type in C++, there is a corresponding function that typecasts its argument to the appropriate type.

4.3 Math functions

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (if x happens to be 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

C++ provides a set of built-in functions that includes most of the mathematical operations you can think of. The math functions are invoked using a syntax that is similar to mathematical notation:

```
double log = log (17.0);
double angle = 1.5;
double height = sin (angle);
```

The first example sets `log` to the logarithm of 17, base e. There is also a function called `log10` that takes logarithms base 10.

The second example finds the sine of the value of the variable `angle`. C++ assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π.

If you don't happen to know π to 15 digits, you can calculate it using the `acos` function. The arccosine (or inverse cosine) of -1 is π, because the cosine of π is -1.

```
double pi = acos(-1.0);
double degrees = 90;
double angle = degrees * 2 * pi / 360.0;
```

Before you can use any of the math functions, you have to include the math **header le** . Header files contain information the compiler needs about functions that are defined outside your program. For example, in the "Hello, world!" program we included a header file named `iostream.h` using an **include** statement:

```
#include <iostream.h>
```

`iostream.h` contains information about input and output (I/O) streams, including the object named `cout`.

Similarly, the math header file contains information about the math functions. You can include it at the beginning of your program along with `iostream.h`:

```
#include <math.h>
```

4.4 Composition

Just as with mathematical functions, C++ functions can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
double x = cos (angle + pi/2);
```

This statement takes the value of `pi`, divides it by two and adds the result to the value of `angle`. The sum is then passed as an argument to the `cos` function.

You can also take the result of one function and pass it as an argument to another:

```
double x = exp (log (10.0));
```

This statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

4.5 Adding new functions

So far we have only been using the functions that are built into C++, but it is also possible to add new functions. Actually, we have already seen one function definition: `main`. The function named `main` is special because it indicates where the execution of the program begins, but the syntax for `main` is the same as for any other function definition:

```
void NAME ( LIST OF PARAMETERS ) {
  STATEMENTS
}
```

You can make up any name you want for your function, except that you can't call it `main` or any other C++ keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or **call**) the new function.

`main` doesn't take any parameters, as indicated by the empty parentheses `()` in it's definition. The first couple of functions we are going to write also have no parameters, so the syntax looks like this:

```
void newLine () {
  cout << endl;
}
```

This function is named `newLine`; it contains only a single statement, which outputs a newline character, represented by the special value `endl`.

In `main` we can call this new function using syntax that is similar to the way we call the built-in C++ commands:

```
void main ()
{
  cout << "First Line." << endl;
  newLine ();
  cout << "Second Line." << endl;
}
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

```
void main ()
{
  cout << "First Line." << endl;
  newLine ();
  newLine ();
  newLine ();
  cout << "Second Line." << endl;
}
```

Or we could write a new function, named `threeLine`, that prints three new lines:

```
void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

void main ()
{
  cout << "First Line." << endl;
  threeLine ();
  cout << "Second Line." << endl;
}
```

You should notice a few things about this program:

- You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.

- You can have one function call another function. In this case, `main` calls `threeLine` and `threeLine` calls `newLine`. Again, this is common and useful.

- In `threeLine` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new function gives you an opportunity to give a name to a group of statements. Functions can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code. Which is clearer, `newLine` or `cout << endl`?

2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `threeLine` three times. How would you print 27 new lines?

4.6 Definitions and uses

Pulling together all the code fragments from the previous section, the whole program looks like this:

```
#include <iostream.h>

void newLine ()
{
  cout << endl;
}

void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

void main ()
{
  cout << "First Line." << endl;
  threeLine ();
  cout << "Second Line." << endl;
}
```

This program contains three function definitions: `newLine`, `threeLine`, and `main`.

Inside the definition of `main`, there is a statement that uses or calls `threeLine`. Similarly, `threeLine` calls `newLine` three times. Notice that the definition of each function appears above the place where it is used.

This is necessary in C++; the definition of a function must appear before (above) the first use of the function. You should try compiling this program with the functions in a different order and see what error messages you get.

4.7 Programs with multiple functions

When you look at a class definition that contains several functions, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `main`, regardless of where it is in the program (often it is at the bottom). Statements are executed one at a time, in order, until you reach a function call. Function calls are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the called function, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one function can call another. Thus, while we are in the middle of `main`, we might have to go off and execute the statements in `threeLine`. But while we are executing `threeLine`, we get interrupted three times to go off and execute `newLine`.

Fortunately, C++ is adept at keeping track of where it is, so each time `newLine` completes, the program picks up where it left off in `threeLine`, and eventually gets back to `main` so the program can terminate.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

4.8 Parameters and arguments

Some of the built-in functions we have used have **parameters**, which are values that you provide to let the function do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a `double` value as a parameter.

Some functions take more than one parameter, like `pow`, which takes two `doubles`, the base and the exponent.

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a class definition, the parameter list indicates the type of each parameter. For example:

```
void printTwice (char phil) {
  cout << phil << phil << endl;
}
```

This function takes a single parameter, named phil, that has type char. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice, followed by a newline. I chose the name phil to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than phil.

In order to call this function, we have to provide a char. For example, we might have a main function like this:

```
void main () {
  printTwice ('a');
}
```

The char value you provide is called an **argument**, and we say that the argument is **passed** to the function. In this case the value 'a' is passed as an argument to printTwice where it will get printed twice.

Alternatively, if we had a char variable, we could use it as an argument instead:

```
void main () {
  char argument = 'b';
  printTwice (argument);
}
```

Notice something very important here: the name of the variable we pass as an argument (argument) has nothing to do with the name of the parameter (phil). Let me say that again:

> **The name of the variable we pass as an argument has noth ing to do with the name of the parameter**

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the character 'b').

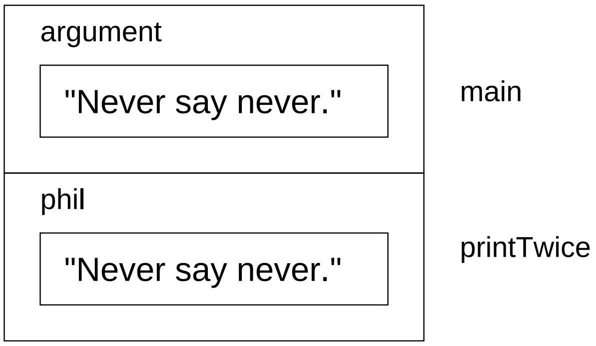
The value you provide as an argument must have the same type as the parameter of the function you call. This rule is important, but it is sometimes confusing because C++ sometimes converts arguments from one type to another automatically. For now you should learn the general rule, and we will deal with exceptions later.

4.9 Parameters and variables are local

Parameters and variables only exist inside their own functions. Within the confines of main, there is no such thing as phil. If you try to use it, the compiler will complain. Similarly, inside printTwice there is no such thing as argument.

Variables like this are said to be **local**. In order to keep track of parameters and local variables, it is useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but the variables are contained in larger boxes that indicate which function they belong to.

For example, the state diagram for `printTwice` looks like this:



Whenever a function is called, it creates a new **instance** of that function. Each instance of a function contains the parameters and local variables for that function. In the diagram an instance of a function is represented by a box with the name of the function on the outside and the variables and parameters inside.

In the example, `main` has one local variable, `argument`, and no parameters. `printTwice` has no local variables and one parameter, named `phil`.

4.10 Functions with multiple parameters

The syntax for declaring and invoking functions with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
void printTime (int hour, int minute) {
  cout << hour;
  cout << ":";
  cout << minute;
}
```

It might be tempting to write `(int hour, minute)`, but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments. The following is wrong!

```
  int hour = 11;
  int minute = 59;
  printTime (int hour, int minute);   // WRONG!
```

In this case, the compiler can tell the type of `hour` and `minute` by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is `printTime (hour, minute)`.

4.11 Functions with results

You might have noticed by now that some of the functions we are using, like the math functions, yield results. Other functions, like `newLine`, perform an action but don't return a value. That raises some questions:

- What happens if you call a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a function without a result as part of an expression, like `newLine() + 7`?
- Can we write functions that yield results, or are we stuck with things like `newLine` and `printTwice`?

The answer to the third question is "yes, you can write functions that return values," and we'll do it in a couple of chapters. I will leave it up to you to answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in C++, a good way to find out is to ask the compiler.

4.12 Glossary

oatingpoint A type of variable (or value) that can contain fractions as well as integers. There are a few floating-point types in C++; the one we use in this book is `double`.

initialization A statement that declares a new variable and assigns a value to it at the same time.

function A named sequence of statements that performs some useful function. Functions may or may not take parameters, and may or may not produce a result.

parameter A piece of information you provide in order to call a function. Parameters are like variables in the sense that they contain values and have types.

argument A value that you provide when you call a function. This value must have the same type as the corresponding parameter.

call Cause a function to be executed.

Chapter 5

Conditionals and recursion

5.1 The modulus operator

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In C++, the modulus operator is a percent sign, `%`. The syntax is exactly the same as for other operators:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y`.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, `x % 10` yields the rightmost digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

5.2 Conditional execution

In order to write useful programs, we almost always need the ability to check certain conditions and change the behavior of the program accordingly. **Condi tional statements** give us this ability. The simplest form is the `if` statement:

```
if (x > 0) {
  cout << "x is positive" << endl;
}
```

The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the `comparison operators`:

```
x == y               // x equals y
x != y               // x is not equal to y
x > y                // x is greater than y
x < y                // x is less than y
x >= y               // x is greater than or equal to y
x <= y               // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax C++ uses is a little different from mathematical symbols like $=$, $\neq$ and $\leq$. A common error is to use a single `=` instead of a double `==`. Remember that `=` is the assignment operator, and `==` is a comparison operator. Also, there is no such thing as `=<` or `=>`.

The two sides of a condition operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`. Unfortunately, at this point you can't compare `String`s at all! There is a way to compare `String`s, but we won't get to it for a couple of chapters.

5.3 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
  if (x%2 == 0) {
    cout << "x is even" << endl;
  } else {
    cout << "x is odd" << endl;
  }
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code displays a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to "wrap" this code up in a function, as follows:

```
void printParity (int x) {
  if (x%2 == 0) {
    cout << "x is even" << endl;
  } else {
    cout << "x is odd" << endl;
  }
}
```

Now you have a function named `printParity` that will display an appropriate message for any integer you care to provide. In `main` you would call this function as follows:

```
printParity (17);
```

Always remember that when you *call* a function, you do not have to declare the types of the arguments you provide. C++ can figure out what type they are. You should resist the temptation to write things like:

```
int number = 17;
printParity (int number);         // WRONG!!!
```

5.4 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `if`s and `else`s:

```
if (x > 0) {
  cout << "x is positive" << endl;
} else if (x < 0) {
  cout << "x is negative" << endl;
} else {
  cout << "x is zero" << endl;
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-braces lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

5.5 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if (x == 0) {
  cout << "x is zero" << endl;
} else {
  if (x > 0) {
    cout << "x is positive" << endl;
  } else {
    cout << "x is negative" << endl;
  }
}
```

There is now an outer conditional that contains two branches. The first branch contains a simple output statement, but the second branch contains another `if` statement, which has two branches of its own. Fortunately, those two branches are both output statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

5.6 The `return` statement

The `return` statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
#include <math.h>

void printLogarithm (double x) {
  if (x <= 0.0) {
    cout << "Positive numbers only, please." << endl;
    return;
  }

  double result = log (x);
  cout << "The log of x is " << result);
}
```

This defines a function named `printLogarithm` that takes a `double` named `x` as a parameter. The first thing it does is check whether `x` is less than or equal to zero, in which case it displays an error message and then uses `return` to exit the function. The flow of execution immediately returns to the caller and the remaining lines of the function are not executed.

I used a floating-point value on the right side of the condition because there is a floating-point variable on the left.

Remember that any time you want to use one a function from the math library, you have to include the header file `math.h`.

5.7 Recursion

I mentioned in the last chapter that it is legal for one function to call another, and we have seen several examples of that. I neglected to mention that it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following function:

```
void countdown (int n) {
  if (n == 0) {
    cout << "Blastoff!" << endl;
  } else {
    cout << n << endl;
    countdown (n-1);
  }
}
```

The name of the function is `countdown` and it takes a single integer as a parameter. If the parameter is zero, it outputs the word "Blastoff." Otherwise, it outputs the parameter and then calls a function named `countdown`—itself—passing `n-1` as an argument.

What happens if we call this function like this:

```
void main ()
{
  countdown (3);
}
```

The execution of `countdown` begins with `n=3`, and since `n` is not zero, it outputs the value 3, and then calls itself...

> The execution of `countdown` begins with `n=2`, and since `n` is not zero, it outputs the value 2, and then calls itself...
>
> > The execution of `countdown` begins with `n=1`, and since `n` is not zero, it outputs the value 1, and then calls itself...
> >
> > > The execution of `countdown` begins with `n=0`, and since `n` is zero, it outputs the word "Blastoff!" and then returns.
> >
> > The countdown that got `n=1` returns.
>
> The countdown that got `n=2` returns.

The countdown that got `n=3` returns.

And then you're back in `main` (what a trip). So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the functions `newLine` and `threeLine`.

```
void newLine () {
  cout << endl;
}

void threeLine () {
  newLine ();  newLine ();  newLine ();
}
```

Although these work, they would not be much help if I wanted to output 2 newlines, or 106. A better alternative would be

```
void nLines (int n) {
  if (n > 0) {
    cout << endl;
    nLines (n-1);
  }
}
```

This program is similar to `countdown`; as long as `n` is greater than zero, it outputs one newline, and then calls itself to output `n-1` additional newlines. Thus, the total number of newlines is `1 + (n-1)`, which usually comes out to roughly `n`.

The process of a function calling itself is called **recursion**, and such functions are said to be **recursive**.

5.8 Infinite recursion

In the examples in the previous section, notice that each time the functions get called recursively, the argument gets smaller by one, so eventually it gets to zero. When the argument is zero, the function returns immediately, *without making any recursive calls*. This case—when the function completes without making a recursive call—is called the **base case**.

If a recursion never reaches a base case, it will go on making recursive calls forever and the program will never terminate. This is known as **innite re cursion**, and it is generally not considered a good idea.

In most programming environments, a program with an infinite recursion will not really run forever. Eventually, something will break and the program will report an error. This is the first example we have seen of a run-time error (an error that does not appear until you run the program).

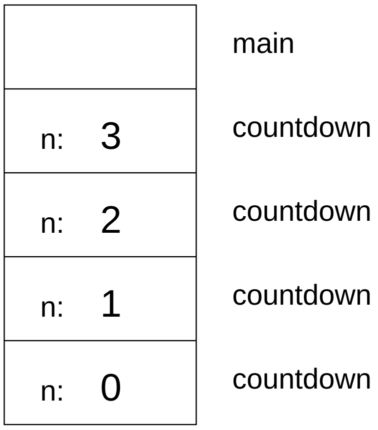
You should write a small program that recurses forever and run it to see what happens.

5.9 Stack diagrams for recursive functions

In the previous chapter we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can make it easier to interpret a recursive function.

Remember that every time a function gets called it creates a new instance that contains the function's local variables and parameters.

This figure shows a stack diagram for countdown, called with `n = 3`:



There is one instance of `main` and four instances of `countdown`, each with a different value for the parameter `n`. The bottom of the stack, `countdown` with `n=0` is the base case. It does not make a recursive call, so there are no more instances of `countdown`.

The instance of `main` is empty because `main` does not have any parameters or local variables. As an exercise, draw a stack diagram for `nLines`, invoked with the parameter `n=4`.

5.10 Glossary

modulus An operator that works on integers and yields the remainder when one number is divided by another. In C++ it is denoted with a percent sign (`%`).

conditional A block of statements that may or may not be executed depending on some condition.

chaining A way of joining several conditional statements in sequence.

nesting Putting a conditional statement inside one or both branches of another conditional statement.

recursion The process of calling the same function you are currently executing.

innite recursion A function that calls itself recursively without every reaching the base case. Eventually an infinite recursion will cause a run-time error.

Chapter 6

Fruitful functions

6.1 Return values

Some of the built-in functions we have used, like the math functions, have produced results. That is, the effect of calling the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
  double e = exp (1.0);
  double height = radius * sin (angle);
```

But so far all the functions we have written have been **void** functions; that is, functions that return no value. When you call a void function, it is typically on a line by itself, with no assignment:

```
  nLines (3);
  countdown (n-1);
```

In this chapter, we are going to write functions that return things, which I will refer to as **fruitful** functions, for want of a better name. The first example is `area`, which takes a `double` as a parameter, and returns the area of a circle with the given radius:

```
double area (double radius) {
  double pi = acos (-1.0);
  double area = pi * radius * radius;
  return area;
}
```

The first thing you should notice is that the beginning of the function definition is different. Instead of `void`, which indicates a void function, we see `double`, which indicates that the return value from this function will have type `double`.

Also, notice that the last line is an alternate form of the `return` statement that includes a return value. This statement means, "return immediately from this function and use the following expression as a return value." The expression you provide can be arbitrarily complicated, so we could have written this function more concisely:

```
double area (double radius) {
  return acos(-1.0) * radius * radius;
}
```

On the other hand, **temporary** variables like `area` often make debugging easier. In either case, the type of the expression in the `return` statement must match the return type of the function. In other words, when you declare that the return type is `double`, you are making a promise that this function will eventually produce a `double`. If you try to `return` with no expression, or an expression with the wrong type, the compiler will take you to task.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Since these returns statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one `return` statement in a function, you should keep in mind that as soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a `return` statement, or any place else where it can never be executed, is called **dead code**. Some compilers warn you if part of your code is dead.

If you put return statements inside a conditional, then you have to guarantee that *every possible path* through the program hits a return statement. For example:

```
double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else if (x > 0) {
    return x;
  }                          // WRONG!!
}
```

This program is not correct because if `x` happens to be 0, then neither condition will be true and the function will end without hitting a return statement. Unfortunately, many C++ compilers do not catch this error. As a result, the program may compile and run, but the return value when `x==0` could be anything, and will probably be different in different environments.

By now you are probably sick of seeing compiler errors, but as you gain more experience, you will realize that the only thing worse than getting a compiler error is *not* getting a compiler error when your program is wrong.

Here's the kind of thing that's likely to happen: you test `absoluteValue` with several values of `x` and it seems to work correctly. Then you give your program to someone else and they run it in another environment. It fails in some mysterious way, and it takes days of debugging to discover that the problem is an incorrect implementation of `absoluteValue`. If only the compiler had warned you!

From now on, if the compiler points out an error in your program, you should not blame the compiler. Rather, you should thank the compiler for finding your error and sparing you days of debugging. Some compilers have an option that tells them to be extra strict and report all the errors they can find. You should turn this option on all the time.

As an aside, you should know that there is a function in the math library called `fabs` that calculates the absolute value of a `double`—correctly.

6.2 Program development

At this point you should be able to look at complete C++ functions and tell what they do. But it may not be clear yet how to go about writing them. I am going to suggest one technique that I call **incremental development**.

As an example, imagine you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the usual definition,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{6.1}$$

The first step is to consider what a `distance` function should look like in C++. In other words, what are the inputs (parameters) and what is the output (return value).

In this case, the two points are the parameters, and it is natural to represent them using four `double`s. The return value is the distance, which will have type `double`.

Already we can write an outline of the function:

```
double distance (double x1, double y1, double x2, double y2) {
  return 0.0;
}
```

The `return` statement is a placekeeper so that the function will compile and return something, even though it is not the right answer. At this stage the function doesn't do anything useful, but it is worthwhile to try compiling it so we can identify any syntax errors before we make it more complicated.

In order to test the new function, we have to call it with sample values. Somewhere in `main` I would add:

```
  double dist = distance (1.0, 2.0, 4.0, 6.0);
  cout << dist << endl;
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result will be 5 (the hypotenuse of a 3-4-5 triangle). When you are testing a function, it is useful to know the right answer.

Once we have checked the syntax of the function definition, we can start adding lines of code one at a time. After each incremental change, we recompile and run the program. That way, at any point we know exactly where the error must be—in the last line we added.

The next step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. I will store those values in temporary variables named `dx` and `dy`.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  cout << "dx is " << dx << endl;
  cout << "dy is " << dy << endl;
  return 0.0;
}
```

I added output statements that will let me check the intermediate values before proceeding. As I mentioned, I already know that they should be 3.0 and 4.0.

When the function is finished I will remove the output statements. Code like that is called **scaolding**, because it is helpful for building the program, but it is not part of the final product. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square `dx` and `dy`. We could use the `pow` function, but it is simpler and faster to just multiply each term by itself.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  cout << "dsquared is " << dsquared;
  return 0.0;
}
```

Again, I would compile and run the program at this stage and check the intermediate value (which should be 25.0).

Finally, we can use the `sqrt` function to compute and return the result.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  double result = sqrt (dsquared);
  return result;
}
```

Then in `main`, we should output and check the value of the result.

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this incremental development process can save you a lot of debugging time.

The key aspects of the process are:

- Start with a working program and make small, incremental changes. At any point, if there is an error, you will know exactly where it is.
- Use temporary variables to hold intermediate values so you can output and check them.
- Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

6.3 Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we have a function, `distance`, that does that.

```
  double radius = distance (xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius, and return it.

```
  double result = area (radius);
  return result;
```

Wrapping that all up in a function, we get:

```
double fred (double xc, double yc, double xp, double yp) {
  double radius = distance (xc, yc, xp, yp);
  double result = area (radius);
  return result;
}
```

The name of this function is `fred`, which may seem odd. I will explain why in the next section.

The temporary variables `radius` and `area` are useful for development and debugging, but once the program is working we can make it more concise by composing the function calls:

```
double fred (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

6.4 Overloading

In the previous section you might have noticed that `fred` and `area` perform similar functions—finding the area of a circle—but take different parameters. For `area`, we have to provide the radius; for `fred` we provide two points.

If two functions do the same thing, it is natural to give them the same name. In other words, it would make more sense if `fred` were called `area`.

Having more than one function with the same name, which is called **over loading**, is legal in C++ *as long as each version takes different parameters.* So we can go ahead and rename `fred`:

```
double area (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

This looks like a recursive function, but it is not. Actually, this version of `area` is calling the other version. When you call an overloaded function, C++ knows which version you want by looking at the arguments that you provide. If you write:

```
    double x = area (3.0);
```

C++ goes looking for a function named `area` that takes a `double` as an argument, and so it uses the first version. If you write

```
    double x = area (1.0, 2.0, 4.0, 6.0);
```

C++ uses the second version of `area`.

Many of the built-in C++ commands are overloaded, meaning that there are different versions that accept different numbers or types of parameters.

Although overloading is a useful feature, it should be used with caution. You might get yourself nicely confused if you are trying to debug one version of a function while accidently calling a different one.

Actually, that reminds me of one of the cardinal rules of debugging: **make sure that the version of the program you are looking at is the version of the program that is running** Some time you may find yourself making one change after another in your program, and seeing the same thing every time you run it. This is a warning sign that for one reason or another you are not running the version of the program you think you are. To check, stick in an output statement (it doesn't matter what it says) and make sure the behavior of the program changes accordingly.

6.5 Boolean values

The types we have seen so far are pretty big. There are a lot of integers in the world, and even more floating-point numbers. By comparison, the set of characters is pretty small. Well, there is another type in C++ that is even smaller. It is called **boolean**, and the only values in it are `true` and `false`.

Without thinking about it, we have been using boolean values for the last couple of chapters. The condition inside an `if` statement or a `while` statement is a boolean expression. Also, the result of a comparison operator is a boolean value. For example:

```
if (x == 5) {
  // do something
}
```

The operator `==` compares two integers and produces a boolean value.

The values `true` and `false` are keywords in C++, and can be used anywhere a boolean expression is called for. For example,

```
while (true) {
  // loop forever
}
```

is a standard idiom for a loop that should run forever (or until it reaches a `return` or `break` statement).

6.6 Boolean variables

As usual, for every type of value, there is a corresponding type of variable. In C++ the boolean type is called **bool**. Boolean variables work just like the other types:

```
bool fred;
fred = true;
bool testResult = false;
```

The first line is a simple variable declaration; the second line is an assignment, and the third line is a combination of a declaration and as assignment, called an initialization.

As I mentioned, the result of a comparison operator is a boolean, so you can store it in a `bool` variable

```
bool evenFlag = (n%2 == 0);      // true if n is even
bool positiveFlag = (x > 0);     // true if x is positive
```

and then use it as part of a conditional statement later

```
if (evenFlag) {
  cout << "n was even when I checked it" << endl;
}
```

A variable used in this way is called a **ag**, since it flags the presence or absence of some condition.

6.7 Logical operators

There are three **logical operators** in C++: AND, OR and NOT, which are denoted by the symbols `&&`, `||` and `!`. The semantics (meaning) of these operators is similar to their meaning in English. For example `x > 0 && x < 10` is true only if `x` is greater than zero AND less than 10.

`evenFlag || n%3 == 0` is true if *either* of the conditions is true, that is, if `evenFlag` is true OR the number is divisible by 3.

Finally, the NOT operator has the effect of negating or inverting a bool expression, so `!evenFlag` is true if `evenFlag` is false; that is, if the number is odd.

Logical operators often provide a way to simplify nested conditional statements. For example, how would you write the following code using a single conditional?

```
if (x > 0) {
  if (x < 10) {
    cout << "x is a positive single digit." << endl;
  }
}
```

6.8 Bool functions

Functions can return `bool` values just like any other type, which is often convenient for hiding complicated tests inside functions. For example:

```
bool isSingleDigit (int x)
{
  if (x >= 0 && x < 10) {
    return true;
  } else {
    return false;
  }
}
```

The name of this function is `isSingleDigit`. It is common to give boolean functions names that sound like yes/no questions. The return type is `bool`, which means that every return statement has to provide a `bool` expression.

The code itself is straightforward, although it is a bit longer than it needs to be. Remember that the expression `x >= 0 && x < 10` has type `bool`, so there is nothing wrong with returning it directly, and avoiding the `if` statement altogether:

```
bool isSingleDigit (int x)
{
  return (x >= 0 && x < 10);
}
```

In `main` you can call this function in the usual ways:

```
  cout << isSingleDigit (2) << endl;
  bool bigFlag = !isSingleDigit (17);
```

The first line outputs the value `true` because 2 is a single-digit number. Unfortunately, when C++ outputs `bool`s, it does not display the words `true` and `false`, but rather the integers `1` and `0`.[1]

The second line assigns the value `true` to `bigFlag` only if 17 is *not* a single-digit number.

The most common use of `bool` functions is inside conditional statements

```
  if (isSingleDigit (x)) {
    cout << "x is little" << endl;
  } else {
    cout << "x is big" << endl;
  }
```

6.9 Returning from main

Now that we have functions that return values, I can let you in on a secret. `main` is not really supposed to be a `void` function. It's supposed to return an integer:

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

The usual return value from `main` is 0, which indicates that the program succeeded at whatever it was supposed to do. If something goes wrong, it is common to return -1, or some other value that indicates what kind of error occurred.

[1] There is a way to fix that using the `boolalpha` flag, but it is too hideous to mention.

Of course, you might wonder who this value gets returned to, since we never call `main` ourselves. It turns out that when the system executes a program, it starts by calling `main` in pretty much the same way it calls all the other functions.

There are even some parameters that are passed to `main` by the system, but we are not going to deal with them for a little while.

6.10 More recursion

So far we have only learned a small subset of C++, but you might be interested to know that this subset is now a **complete** programming language, by which I mean that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features we have used so far (actually, we would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a non-trivial exercise first accomplished by Alan Turing, one of the first computer scientists (well, some would argue that he was a mathematician, but a lot of the early computer scientists started as mathematicians). Accordingly, it is known as the Turing thesis. If you take a course on the Theory of Computation, you will have a chance to see the proof.

To give you an idea of what you can do with the tools we have learned so far, we'll evaluate a few recursively-defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is typically not very useful:

frabjuous an adjective used to describe something that is frabjuous.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the mathematical function **factorial**, you might get something like:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

(Factorial is usually denoted with the symbol !, which is not to be confused with the C++ logical operator `!` which means NOT.) This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of $n-1$. So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, we get 3! equal to 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a C++ program to evaluate it. The first step is to decide what the parameters are for this function, and what the return type is. With a little thought, you should conclude that factorial takes an integer as a parameter and returns an integer:

```
int factorial (int n)
{
}
```

If the argument happens to be zero, all we have to do is return 1:

```
int factorial (int n)
{
  if (n == 0) {
    return 1;
  }
}
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n-1$, and then multiply it by n.

```
int factorial (int n)
{
  if (n == 0) {
    return 1;
  } else {
    int recurse = factorial (n-1);
    int result = n * recurse;
    return result;
  }
}
```

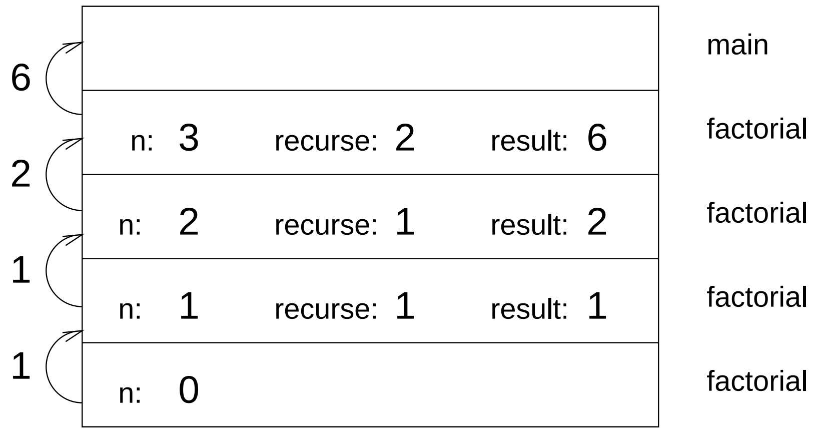
If we look at the flow of execution for this program, it is similar to `nLines` from the previous chapter. If we call `factorial` with the value 3:

Since 3 is not zero, we take the second branch and calculate the factorial of $n-1$...

> Since 2 is not zero, we take the second branch and calculate the factorial of $n-1$...
>
> > Since 1 is not zero, we take the second branch and calculate the factorial of $n-1$...
> >
> > > Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive calls.
> >
> > The return value (1) gets multiplied by `n`, which is 1, and the result is returned.
>
> The return value (1) gets multiplied by `n`, which is 2, and the result is returned.

The return value (2) gets multiplied by `n`, which is 3, and the result, 6, is returned to `main`, or whoever called `factorial (3)`.

Here is what the stack diagram looks like for this sequence of function calls:



The return values are shown being passed back up the stack.

Notice that in the last instance of `factorial`, the local variables `recurse` and `result` do not exist because when `n=0` the branch that creates them does not execute.

6.11 Leap of faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, it can quickly become labarynthine. An alternative is what I call the "leap of faith." When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `cos` or `exp`, you don't examine the implementations of those functions. You just assume that they work, because the people who wrote the built-in libraries were good programmers.

Well, the same is true when you call one of your own functions. For example, in Section 6.8 we wrote a function called `isSingleDigit` that determines whether a number is between 0 and 9. Once we have convinced ourselves that this function is correct—by testing and examination of the code—we can use the function without ever looking at the code again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should *assume* that the recursive call works (yields the correct result), and then ask yourself, "Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

Of course, it is a bit strange to assume that the function works correctly when you have not even finished writing it, but that's why it's called a leap of faith!

6.12 One more example

In the previous example I used temporary variables to spell out the steps, and to make the code easier to debug, but I could have saved a few lines:

```
int factorial (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial (n-1);
  }
}
```

From now on I will tend to use the more concise version, but I recommend that you use the more explicit version while you are developing code. When you have it working, you can tighten it up, if you are feeling inspired.

After `factorial`, the classic example of a recursively-defined mathematical function is `fibonacci`, which has the following definition:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);$$

Translated into C++, this is

```
int fibonacci (int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci (n-1) + fibonacci (n-2);
  }
}
```

If you try to follow the flow of execution here, even for fairly small values of `n`, your head explodes. But according to the leap of faith, if we assume that the two recursive calls (yes, you can make two recursive calls) work correctly, then it is clear that we get the right result by adding them together.

6.13 Glossary

return type The type of value a function returns.

return value The value provided as the result of a function call.

dead code Part of a program that can never be executed, often because it appears after a `return` statement.

scaolding Code that is used during program development but is not part of the final version.

void A special return type that indicates a void function; that is, one that does not return a value.

overloading Having more than one function with the same name but different parameters. When you call an overloaded function, C++ knows which version to use by looking at the arguments you provide.

boolean A value or variable that can take on one of two states, often called *true* and *false*. In C++, boolean values can be stored in a variable type called `bool`.

ag A variable (usually type `bool`) that records a condition or status information.

comparison operator An operator that compares two values and produces a boolean that indicates the relationship between the operands.

logical operator An operator that combines boolean values in order to test compound conditions.