1

Foundations of C++ Programming

C++ Programs: Form and Function 2

Scientists build to learn; engineers learn to build.
Fred Brooks

tem·plate (têm´plît) *n.* A pattern, …used as a guide in making something accurately …
The American Heritage Dictionary

Art is the imposing of a pattern on experience, and our aesthetic enjoyment in recognition of the pattern.
Dialogues of Alfred North Whitehead (June 10, 1943)

It is a bad plan that admits of no modification.
Publius Syrus, Maxim 469

To learn to write programs, you must write programs and you must read programs. Although this statement may not seem profound, it is a lesson that is often left unpracticed and, subsequently, unmastered. In thinking about the concepts presented in this chapter, and in practicing them in the context of writing C++ programs, you should keep the following three things in mind.

1. Programming has elements of both art and science. Just as designing a building requires both a sense of aesthetics and a knowledge of structural engineering, designing a program requires an understanding of programming aesthetics, knowledge of computer science, and practice in software engineering.
2. Use the programs provided as templates when designing and constructing programs of your own—use what's provided along with your own ingenuity. When some concept is unclear, stop to work on it and think about it before continuing. This work will involve experimenting with the programs provided. Experimenting with a program means reading, executing, testing, and modifying the program. When you experiment with a program, you can try to find its weak points and its strengths.
3. Practice.

This book is predicated on the belief that you learn best by doing new things and by studying things similar to the new things. This technique applies to learning carpentry, learning to play a musical instrument, or learning to program a computer. Not everyone can win a Grammy award and not everyone can win the Turing award,[1] but becoming adept programmers and practitioners of computer science is well within your grasp.

[1]The former is awarded for musical excellence, the latter for excellence in computer science.

Ultimately programs are a means of expressing algorithms in a form that computers execute. Before studying complicated and large programs, it's necessary to begin with the basics of what a program is, how programs are executed, and what C++ programs can do. However, understanding a program *completely* requires a great deal of experience and knowledge about C++. We'll use some simple programs to illustrate basic concepts. Try to focus on the big picture of programming; don't get bogged down by every detail. The details will eventually become clearer, and you'll master them by studying many programming examples.

2.1 Simple C++ Programs

In this section we introduce simple C++ programs to demonstrate how to use the C++ language. These programs produce **output**; they cause characters to be displayed on a computer screen. The first C++ program is *hello.cpp*, Program 2.1. This program is based on the first program in the book [KR78], written by the inventors of C; and it is the first program in [Str97], written by the inventor of C++. It doesn't convey the power of C++, but it's a tradition for C and C++ programmers to begin with this program.

All programs have names, in this case *hello*, and suffixes, in this case *.cpp*. In this book all programs have the suffix *.cpp*, which is one convention (other suffixes used include *.cc* and *.cxx*). If this program is compiled and executed, it generates the material shown in the box labeled "output."

Program 2.1 hello.cpp

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

//traditional first program
// author: Owen Astrachan, 2/27/99

int main()
{
    cout << "Hello world" << endl;
    return 0;
}
```

hello.cpp

OUTPUT

```
prompt> hello
Hello world
```

Program 2.2, *hello2.cpp*, produces output identical to that of Program 2.1. We'll look at why one of these versions might be preferable as we examine the structure of C++ programs. In general, given a specific programming task there are many, many different programs that will perform the task.

Program 2.2 hello2.cpp

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

// traditional first program with user defined function
// author: Owen Astrachan, 02/27/99

void Hello()
{
    cout << "Hello world" << endl;
}

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

hello2.cpp

2.1.1 Syntax and Semantics

Programs are run by computers, not by humans. There is often much less room for error when writing programs than when writing English. In particular, you'll need to be aware of certain rules that govern the use of C++. These rules fall into two broad categories: rules of syntax and rules of semantics.

What are syntax and semantics? In English and other natural languages, syntax is the manner in which words are used to construct sentences and semantics is the meaning of the sentences. We'll see that similar definitions apply to syntax and semantics in C++ programs. Before reviewing rules for C++ programs, we'll look at some rules that govern the use and construction of English words and sentences:

- Rules of spelling:
  i before *e* except after *c* or when sounding like $\overline{a}$ as in . . . *neighbor* and *weigh*.
- Rules of grammar:
  "with *none* use the singular verb when the word means 'no one' . . . a plural verb is commonly used when *none* suggests more than one thing or person—'None are so fallible as those who are sure they're right' " [JW89].
- Rules of style:
  "Avoid the use of qualifiers. *Rather, very, little, pretty*—these are the leeches that infest the pond of prose, sucking the blood of words" [JW89].

Similar rules exist in C++. One difference between English and C++ is that the meaning, or **semantics,** of a poorly constructed English sentence can be understood although the syntax is incorrect:

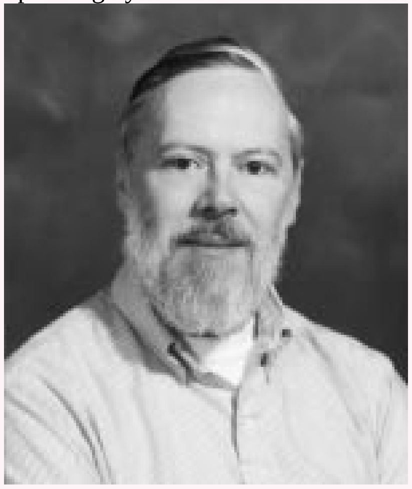
> Its inconceivable that someone can study a language and not know whether or not a kind of sentence—the ungainly ones, the misspelled ones, those that are unclear—are capable of understanding.

This sentence has at least four errors in spelling, grammar, and style; its meaning, however, is still discernible.

In general, programming languages demand more precision than do natural languages such as English. A missing semicolon might make an English sentence fall into the run-on category. A missing semicolon in a C++ program can stop the program from working at all.

Dennis Ritchie *(b. 1941)*

Dennis Ritchie developed the C programming language and codeveloped the UNIX operating system. For his work with UNIX, he shared the 1983 Turing award with the codeveloper, Ken Thompson. In his Turing address, Ritchie writes of what computer science is.

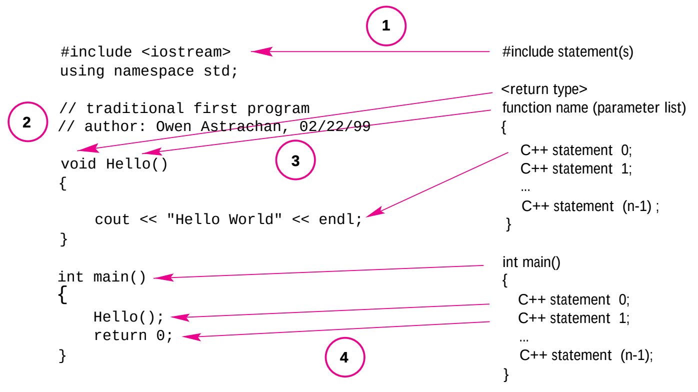


Computer science research is different from these [physics, chemistry, mathematics] more traditional disciplines. Philosophically it differs from the physical sciences because it seeks not to discover, explain, or exploit the natural world, but instead to study the properties of machines of human creation. In this it is analogous to mathematics, and indeed the "science" part of computer science is, for the most part, mathematical in spirit. But an inevitable aspect of computer science is the creation of computer programs: objects that, though intangible, are subject to commercial exchange.

Ritchie completed his doctoral dissertation in applied mathematics but didn't earn his doctorate because "I was so bored, I never turned it in." In citing the work that led to the Turing award, the selection committee mentions this:

> *The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the UNIX system has led a generation of software designers to new ways of thinking about programming.*

For more information see [Sla87, ACM87].



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

// traditional first program
// author: Owen Astrachan, 02/22/99

void Hello()
{

    cout << "Hello World" << endl;
}

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

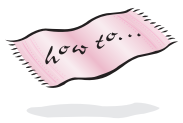
Figure 2.1 Format of a C++ program.

We'll illustrate the important syntactic details of a C++ program by studying *hello.cpp* and *hello2.cpp*, Progs. 2.1 and 2.2. We'll then extend these into a typical and general program framework. Four rules for C++ program syntax and style will also be listed. A useful tool for checking the syntax of programs is the C++ compiler, which indicates whether a program has the correct form—that is, whether the program statements are "worded correctly." You should *not* worry about memorizing the syntactic details of C++ (e.g., where semicolons go). The details of the small subset of C++ covered in this chapter will become second nature as you read and write programs.

All the C++ programs we'll study in this book have the format shown in Figure 2.1 and explained below. Although this format will be used, the spacing of each line in a program does not affect whether a program works. The amount of **white space** and the blank lines between functions help make programs easier for humans to read but do not affect how a program works. *White space* refers to the space, tab, and return keys.

1. Programs begin with the appropriate `#include` statements.[2] Each include statement provides access, via a **header** file to a **library** of useful functions. We normally think of a library as a place from which we can borrow books. A programming library consists of off-the-shelf programming tools that programmers can borrow. These tools are used by programmers to make the task of writing programs easier.
In most C++ programs it is necessary to import information from such libraries into the program. In particular, information for output (and input) is stored in the `iostream` library, accessible by including the header file `<iostream>`, as shown in Figure 2.1. If a program has no output (or input), it isn't necessary to include `<iostream>`.

[2]The # sign is read as either "sharp" or "pound"; I usually say "pound-include" when reading to myself or talking with others.



All programs that use standard C++ libraries should have `using namespace std;` after the `#include` statements. Namespaces[3] are explained in Section A.2.3 of Howto A.

2. All programs should include comments describing the purpose of the program. As programs get more complex, the comments become more intricate. For the simple programs studied in this chapter, the comments are brief. The compiler ignores comments; programmers put comments in programs for human readers. C++ comments extend from a double slash, `//`, to the end of the line. Another style of commenting permits multiline comments—any text between `/*` and `*/` is treated as a comment. It's important to remember that people read programs, so writing comments should be considered mandatory although programs will work without them.
3. Zero, one, or more **programmer-defined** functions follow the `#include` statements and comments. Program 2.2, *hello2.cpp,* has two programmer-defined functions, named `Hello` and `main`. Program 2.1, *hello.cpp,* has one programmer-defined function, named `main`. In general, a function is a way of grouping C++ statements together so that they can be referred to by a single name. The function is an abstraction used in place of the statements. As shown in Figure 2.1, each programmer-defined function consists of the function's **return type,** the function's **name,** the function's **parameter list,** and the statements that make up the function's **body.** For the function `Hello` the return type is `void`, the name of the function is `Hello`, and there is an empty parameter list. There is only one C++ statement in `Hello`.
   The return type of the function `main` is `int`. In C++, an `int` represents an integer; we'll discuss this in detail later. The name of the function is `main` and it too has an empty parameter list. There are two statements in the function body; the second statement is `return 0`. We'll also discuss the return statement in some detail later. The last statement in the function `main` of each program you write should be `return 0`.
4. Every C++ program must have exactly one function named **main.** The statements in `main` are executed first when a program is run. Some C++ compilers will generate a warning if the statement `return 0` is not included as the last statement in `main` (such statements are explained in the next chapter). It's important to spell `main` with lowercase letters. A function named `Main` is different from `main` because names are case-sensitive in C++. Finally, the return type of `main` should be specified as `int` for reasons we'll explore in Chapter 4.[4]

Since program execution begins with `main`, it is a good idea to start reading a program beginning with `main` when you are trying to understand what the program does and how it works.

[3]Compilers that support the C++ standard require `using namespace std;` but older compilers don't support namespaces. Howto A explains this in more detail.

[4]Some books use a return type of `void` for `main`. According to the C++ standard, this is not legal; the return type *must* be `int`.

Pause to Reflect

2.1 Find four errors in the ungainly sentence given above (and reproduced below) whose semantics (meaning) is understandable despite the errors.

> Its inconceivable that someone can study a language and not know whether or not a kind of sentence—the ungainly ones, the misspelled ones, those that are unclear—are capable of understanding.

Are humans better “processors” than computers because of the ability to comprehend “faulty” phrases? Explain your answer.

2.2 Find two syntax errors and one semantic error in the sentence “There is three things wrong with this sentence.”

2.3 Given the four rules for C++ programs, what is the smallest legal C++ program? (Hint: it doesn’t produce any output, so it doesn’t need a `#include` statement.)

2.4 No rules are given about using separate lines for C++ functions and statements. If `main` from Program 2.2 is changed as follows, is the program legal C++?

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

2.2 How a Program Works

Computer programs execute sequences of statements often producing some form of output. Statements are executed whether the program is written in a high- or low-level language. Determining what statements to include in a program is part of the art and science of programming. Developing algorithms and classes, and the relationship between classes, is also part of this art and science.

When you execute a program, either by typing the name of the program at a prompt or using a mouse to click on “run” in a menu, the execution starts in the function `main`. When the program is running it uses the processor of the computer; when the program is finished it returns control of the processor to the operating system. The explicit `return 0` statement in `main` makes it clear that control is returning to the operating system.

The output of Program 2.1, *hello.cpp*, results from the execution of the statement beginning `cout <<` followed by other characters, followed by other symbols. This statement is in the body of the function `main`. The characters between the double quotation marks appear on the screen exactly as they appear between the quotes in the statement. (Notice that the quotes do not appear on the screen.) If the statement

```
cout << "Hello world" << endl;
```

is changed to

```
cout << "Goodbye cruel planet" << endl;
```

then execution of the program *hello.cpp* results in the output that follows:

OUTPUT

```
Goodbye cruel planet
```

2.2.1 Flow of Control

In every C++ program, execution begins with the first statement in the function `main`. After this statement is executed, each statement in `main` is executed in turn. When the last statement has been executed, the program is done. Several uses of `<<` can be combined into a single statement as shown in *hello.cpp*, Program 2.1. Note that `endl` indicates that an end-of-line is to be output (hence "end ell"). For example, if the statement

```
cout << "Hello world" << endl;
```

is changed to

```
cout << "Goodbye" << endl << "cruel planet" << endl;
```

then execution of the program *hello.cpp* results in the output shown below, where the first `endl` forces a new line of output.

OUTPUT

```
Goodbye
cruel planet
```

This modified output could be generated by using two separate output statements:

```
cout << "Goodbye" << endl;
cout << "cruel planet" << endl;
```

Since each statement is executed one after the other, the output generated will be the same as that shown above.

In C++, statements are terminated by a semicolon. This means that a single statement can extend over several lines since the semicolon is used to determine when the statement ends.

```
cout << "Goodbye"      << endl
     << "cruel planet" << endl;
```

Just as run-on sentences in English can obscure the meaning, long statements in C++ can be hard to read. However, the output statement above that uses two lines isn't really too long; some programmers prefer it to the two statement version since it is easy to read.

Function Calls. If a statement invokes, or **calls**, a function—such as when the statement `Hello();` in `main` invokes the function `Hello` in Program 2.2—then each of the statements in the called function is executed. For example, while the statements in the function `Hello` are executing, the statements in `main` are suspended, waiting for the statements in `Hello` to finish. When all the statements in a function have executed, **control** returns to the statement after the call to the function. In Program 2.2 this is the statement `return 0` after the function call `Hello()`. When the last statement in a program has executed, control returns from `main` to the computer just as control returns from `Hello` to `main` when the last statement in `Hello` is executed. If the `return 0` statement in `main` is missing, control will still return to the computer.

In the case of Program 2.2, three statements are executed:

1. The call `Hello();` in the function `main`.
2. The statement `cout << "Hello World" << endl;` in `Hello`.
3. The statement `return 0;` in `main`.

The execution of the second statement above results in the appearance of 11 “visible” characters on the computer’s screen (note that a space is a character just as the letter *H* is a character).

Output Streams. To display output, the **standard output stream** `cout` is used. This stream is accessible in a program via the included library `<iostream>`. If this header file is not included, a program cannot make reference to the stream `cout`. You can think of an output stream as a stream of objects in the same way that a brook or a river is a stream of water. Placing objects on the output stream causes them to appear on the screen eventually just as placing a toy boat on a stream of water causes it to flow downstream. Objects are placed on the output stream using <<, the **insertion operator,** so named since it is used to insert values onto an output stream. Sometimes this operator is read as “put-to.” The word `cout` is pronounced “see-out.”

2.3 What Can Be Output?

Computers were originally developed to be number crunchers, machines used for solving large systems of equations. Not surprisingly, numbers still play a large part in programming and computer science. The output stream `cout` can be used for the output of numbers and words. In Program 2.1 characters appeared on the screen as a result of executing a statement that uses the standard output stream `cout`. Sequences of characters appearing between quotes are called **string literals.** Characters include letters a–z (and uppercase versions), numbers, symbols such as !+$%&*, and many other non-visible “characters,” such as the backspace key, the return key, and in general any key that can be typed from a computer keyboard. String literals cannot change during a program’s execution. The number 3.14159 is a **numeric literal** (it approximates the number π). In addition to string literals, it is possible to output numeric literals and arithmetic expressions.

For example, if the statement

```
cout << "Hello world" << endl;
```

is changed to

```
cout << "Goodbye" << endl << "cruel planet #" << 1 + 2 << endl;
```

then execution of the program *hello.cpp* results in the output shown below.

O U T P U T

```
Goodbye
cruel planet #3
```

The arithmetic expression $1 + 2$ is evaluated and 3, the result of the evaluation, is placed on the output stream. To be more precise, each of the chunks that follow a `<<` are evaluated and cause the output stream to be modified in some way. The string literal `"Goodbye"` evaluates to itself and is placed on the output stream as seven characters. The arithmetic expression $1+2$ evaluates to 3, and the character 3 is placed on the output stream. Each `endl` begins a new line on the output stream.[5]

The C++ compiler ensures that arithmetic expressions are evaluated correctly and, with the help of the stream library, ensures that the appropriate characters are placed on the output stream.

Changing the output statement in *hello.cpp* to the statement here:

```
cout << "The radius of planet #" << 1+2
     << " is " << 6378.38 << " km," << endl
     << "which is " << 6378.38 * 0.62137 << " miles" << endl;
```

results in the output shown below. The symbol `*` is used to multiply two values, and the number 0.62137 is the number of miles in 1 kilometer.

O U T P U T

```
The radius of planet #3 is 6378.38 km
which is 3963.33 miles.
```

The capability of the output stream to handle strings, numbers, and other objects we will encounter later makes it very versatile.

[5]An `endl` also flushes the output buffer. Some programmers think it is bad programming to flush the output buffer just to begin a new line of output. The escape sequence `\n` can be used to start a new line.

Pause to Reflect

2.5 Suppose that the body of the function `Hello` is as shown here:

```
cout << "PI = " << 3.14159 << endl;
```

What appears on the screen? Would the output of

```
cout << "PI = 3.14159" << endl;
```

be the same or different?

2.6 We've noted that more than one statement may appear in a function body:

```
void Hello()
{
  cout << "PI = " << 3.14159 << endl;
  cout << "e  = " << 2.71828 << endl;
  cout << "PI*e = " << 3.14159 * 2.71828 << endl;
}
```

What appears on the screen if the function `Hello` above is executed?

2.7 If the third `cout <<` statement in the previous problem is changed to

```
cout << "PI*e = 3.14159 * 2.71828" << endl;
```

what appears on the screen? Note that this statement puts a single string literal onto the output stream (followed by an `endl`). Why is this output different from the output in the previous question?

2.8 What does the computer display when the statement

```
cout << "1 + 2 = 5" << endl;
```

is executed? Can a computer generate output that is incorrect?

2.9 What modifications need to be made to the output statement in Program 2.1 (the *hello.cpp* program) to generate the following output:

```
I think I think, therefore I think I am
```

2.10 All statements in C++ are terminated by a semicolon. Is the programmer-defined function

```
void Hello()
{
   cout << "Hello World" << endl;
}
```

a statement? Why? Is the function call `Hello()` in `main` a statement?

2.11 If the body of the function `main` of Program 2.2 is changed as shown in the following, what appears on the screen?

```
cout << "I rode the scrambler at the amusement park"
     << endl; Hello();
```

2.4 Using Functions

The flow of control in *hello.cpp*, Program 2.1, is different from *hello2.cpp*, Program 2.2. The use of the function `Hello` in *hello2.cpp* doesn't make the program better or more powerful; it just increases the number of statements that are executed: a function call as well as an output statement. In this section we'll explore programs that use functions in more powerful ways. I say *powerful* in that the resulting programs are easier to modify and are useful in more applications than when functions are not used. Using functions can make programs longer and appear to be more complicated, but sometimes more complicated programs are preferred because they are more general and are easier to modify and maintain. Using functions to group statements together is part of managing the complex task of programming.

We'll now investigate Program 2.3 (*drawhead.cpp*) in which several `cout <<` statements are used in the programmer-defined function `Head`. In *drawhead.cpp* the body of the main function consists of a call of the programmer-defined function `Head` and the statement `return 0`.

For the moment we will assume that all programmer-defined functions are constructed similarly to the manner in which the function `main` is constructed except that the word `void` is used before each function. This is precisely how the syntactic properties of functions were given in Section 2.1.1. The word `void` will be replaced with other words in later examples of programmer-defined functions; for example, we've seen that `int` is used with the function `main`.

Program 2.3 drawhead.cpp

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

// print a head, use of functions
```

```
void Head()
{
    cout << "  ||||||||||||||||  " << endl;
    cout << "  |              |  " << endl;
    cout << "  |    o    o    |  " << endl;
    cout << " _|              |_ " << endl;
    cout << "|_                _|" << endl;
    cout << "  |   |______|   |  " << endl;
    cout << "  |              |  " << endl;

}

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

drawhead.cpp

OUTPUT

```
prompt> drawhead
      ||||||||||||||||
      |              |
      |    o    o    |
     _|              |_
    |_                _|
      |   |______|   |
      |              |
```

At this point the usefulness of functions may not be apparent in the programs we've presented. In the program *parts.cpp* that appears as Program 2.4, many functions are used. If this program is run, the output is the same as the output generated when *drawhead.cpp*, Program 2.3, is run.

Program 2.4 parts.cpp

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

// procedures used to print different heads

void PartedHair()
// prints a "parted hair" scalp
{
    cout << "  |||||||///////// " << endl;
}

void Hair()
// prints a "straight-up" or "frightened" scalp
{
    cout << "  ||||||||||||||||  " << endl;
}

void Sides()
// prints sides of a head - other functions should use distance
// between sides of head here as guide in creating head parts (e.g., eyes)
{
    cout << "  |              |  " << endl;
}

void Eyes()
// prints eyes of a head (corresponding to distance in Sides)
{
    cout << "  |    o    o    |  " << endl;
}

void Ears()
// prints ears (corresponding to distance in Sides)
{
    cout << " _|              |_ " << endl;
    cout << "|_                _|" << endl;
}

void Smile()
// prints smile (corresponding to distance in Sides)
{
    cout << "  |   |______|   |  " << endl;
}

int main()
{
    Hair();
    Sides();
    Eyes();
    Ears();
    Smile();
    Sides();
    return 0;
}
```

parts.cpp

The usefulness of functions should become more apparent when the body of `main` is modified to generate new "heads." This program is longer than the previous programs and may be harder for you to understand. You should begin reading the program starting with the function `main`. Starting with `main` you can then move to reading the functions called from `main`, and the functions that these functions call, and so on. If each call of the function `Sides` in the body of `main` is replaced with two calls to `Sides`, then the new body of `main` and the output generated by the body are as shown here (as shown, `Sides` is called to add space between the eyes and the ears.)

```
int main()
{
    Hair();
    Sides(); Sides();
    Eyes();
    Sides();
    Ears();
    Smile();
    Sides(); Sides();
    return 0;
}
```

OUTPUT

```
  ||||||||||||||||
  |              |
  |              |
  |    o    o    |
 _|              |_
|_                _|
  |   |______|   |
  |              |
  |              |
```

Although *parts.cpp*, Program 2.4, is more complicated than *drawhead.cpp*, Program 2.3, it is easier to modify. Creating heads with different hairstyles or adding a nose is easier when *parts.cpp* is used, because it is clear where the changes should be made because of the names of the functions. It's also possible not only to have more than one hairstyle appear in the same program, but to change what is displayed. For example, adding a nose can be done using a function `Nose`:

```
void Nose()
// draw a mustached nose
{
    cout << "  |      O       |  " << endl;
    cout << "  |    |||||     |  " << endl;
}
```

On the other hand, the original program clearly showed what the printed head looks like; it's not necessary to run the program to see this. As you gain experience as a programmer, your judgment as to when to use functions will get better.

Pause to Reflect

2.12 If you replace the call to the function `Hair` by a call to the function `PartedHair`, what kind of picture is output? How can one of the hair functions be modified to generate a flat head with no hair on it? What picture results if the call `Hair()` is replaced by `Smile()`?

2.13 Design a new function named `Bald` that gives the drawn head the appearance of baldness (perhaps a few tufts of hair on the side are appropriate).

2.14 Modify the function `Smile` so that the face either frowns or shows no emotion. Change the name of the function appropriately.

2.15 What functions should be changed to produce the head shown below? Modify the program to draw such a head.

```
  |||||||/////////
  |              |
  |   ___  ___   |
  |---|o|--|o|---|
  |   ---  ---   |
 _|              |_
| _              _|
  |   |______|   |
  |              |
```

2.5 Functions with Parameters

Program 2.4, *parts.cpp,* showed that the use of programmer-defined functions enabled the program to be more versatile than the program *drawhead.cpp*. Nevertheless, the program had to be re-edited and recompiled to produce new "heads." In this section and the next chapter, you will learn design methods that allow programs to be useful in more contexts.

2.5.1 What Is a Parameter?

In all the programs studied so far, the insertion operator << has been more useful than any programmer-defined function. The insertion operator is versatile because it can be used to write *any message* to the screen. Any sequence of characters between quotes (recall that such a sequence is termed a string literal) and any arithmetic expression can be inserted onto an output stream using the << operator.

- `cout << "Hello world" << endl;`
- `cout << "Goodbye cruel planet" << endl;`
- `cout << "  |||||||/////////  " << endl;`
- `cout << "The square of 10 is " << 10*10 << endl;`

The << operator can be used to output various things just as the addition operator + can be used to add them. It's possible to write programmer-defined functions that have this same kind of versatility. You've probably used a calculator with a square root button: $\sqrt{\ }$. When you find the square root of a number using this button, you're invoking the square root function with an **argument**. In the mathematical expression $\sqrt{101}$, the 101 is the argument of the square root function. Functions that take arguments are called **parameterized functions.** The parameters serve as a means of controlling what the functions do—setting a different parameter results in a different outcome just as $\sqrt{101}$ has a different value than $\sqrt{157}$. The words *parameter* and *argument* are synonyms in this context.

To see how parameters are useful in making functions more general, consider an (admittedly somewhat loose) analogy to a CD player. It is conceivable that one might put a CD of Gershwin's *Rhapsody in Blue* in such a machine and then glue the machine shut. From that point on, the machine becomes a "Gershwin player" rather than a CD player. One can also purchase a "weather box," which is a radio permanently tuned to a weather information service. Although interesting for determining whether to carry an umbrella, the weather box is less general-purpose than a normal (tunable) radio in the same way that the Gershwin player is less versatile than a normal CD player. In the same sense, the << operator is more versatile than the `Head` function in Program 2.3, which always draws the same head.

Functions with parameters are more versatile than functions without parameters although there are times when both kinds of function are useful. Functions that receive parameters must receive the correct kind of parameter or they will not execute properly (often such functions will not compile). Continuing with the CD analogy, suppose that you turn on a CD player with no CD in it. Obviously nothing will be played. Similarly, if it were possible to put a cassette tape into a CD player without damaging the player, the CD player would not be able to play the cassette. Finally, if a 2.5-inch mini-CD is forced into a normal CD player, still nothing is played. The point of this example is that the "parameterized" CD player must be used properly—the appropriate "parameter" (a CD, not a cassette or mini-CD) must be used if the player is to function as intended.

2.5.2 An Example of Parameterization: Happy Birthday

Suppose you are faced with the unenviable task of writing a program that displays the song "Happy Birthday" to a set of quintuplets named Grace, Alan, John, Ada, and Blaise.[6] In designing the program, we employ a concept called **iterative enhancement**, whereby a rough draft of the program is repeatedly refined until the desired program is finished.

A naive, first attempt with this problem might consist of 24 `cout <<` statements; that is, 5 "verses" $\times$ 4 lines per verse $+$ 4 blank lines (note that there is one blank line between each verse so that there is one less blank line than there are verses). Such a program would yield the desired output—but even without much programming experience this solution should be unappealing to you. Indeed, the effort required to generate a new

[6]Coincidentally, these are the first names of five pioneers in computer science: Grace Hopper, Alan Turing, John von Neumann, Ada Lovelace, and Blaise Pascal.

verse in such a program is the same as the effort required to generate a verse in the original program. Nevertheless, such a program has at least one important merit: it is easy to make work. Even though "cut-and-paste" techniques are available in most text editors, it is very likely that you will introduce typos using this approach.

We want to develop a program that mirrors the way people sing "Happy Birthday." You don't think of a special song *BirthdayLaura* to sing to a friend Laura and *Birthday-Dave* for a friend Dave. You use one song and fill in (with a parameter!) the name of the person who has the birthday.

OUTPUT

```
Happy birthday to you
Happy birthday to you
Happy birthday dear
Happy birthday to you

Happy birthday to you
Happy birthday to you
Happy birthday dear
Happy birthday to you

Happy birthday to you
Happy birthday to you
Happy birthday dear
Happy birthday to you

Happy birthday to you
Happy birthday to you
Happy birthday dear
Happy birthday to you

Happy birthday to you
Happy birthday to you
Happy birthday dear
Happy birthday to you
```

Program 2.5 bday.cpp

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

// first attempt at birthday singing
```

```
void Sing()
{
    cout << "Happy birthday to you" << endl;
    cout << "Happy birthday to you" << endl;
    cout << "Happy birthday dear  " << endl;
    cout << "Happy birthday to you" << endl;
    cout << endl;
}

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

bday.cpp

We need to print five copies of the song. We will design a function named `Sing` whose purpose is to generate the birthday song for each of the quintuplets. Initially we will leave the name of the quintuplet out of the function so that five songs are printed, but no names appear in the songs. Once this program works, we'll use parameters to add a name to each song. This technique of writing a preliminary version, then modifying it to lead to a better version, is one that is employed throughout the book. It is the heart of the concept of iterative enhancement.

The first pass at a solution is *bday.cpp*, Program 2.5. Execution of this program yields a sequence of printed verses close to the desired output, but the name of each person whose birthday is being celebrated is missing. One possibility is to use five different functions (`SingGrace`, `SingAlan`, etc.), one function for each verse, but this isn't really any better than just using 24 `cout` statements. We need to parameterize the function `Sing` so that it is versatile enough to provide a song for each quintuplet. This is done in Program 2.6, which generates exactly the output required. Note that the statement

```
cout << "Happy birthday dear " << endl;
```

from Program 2.5 has been replaced with

```
cout << "Happy birthday dear " << person << endl;
```

Program 2.6 bday2.cpp

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

#include <string>

// working birthday program

void Sing(string person)
{
    cout << "Happy birthday to you" << endl;
    cout << "Happy birthday to you" << endl;
```

```
        cout << "Happy birthday dear " << person << endl;
        cout << "Happy birthday to you" << endl;
        cout << endl;
    }

    int main()
    {
        Sing("Grace");
        Sing("Alan");
        Sing("John");
        Sing("Ada");
        Sing("Blaise");
        return 0;
    }
```

bday2.cpp

This statement can be spread over several lines without affecting its behavior.

```
cout << "Happy birthday dear "
     << person
     << endl;
```

Because only one `endl` is used in the output statement, only one line of output is written.

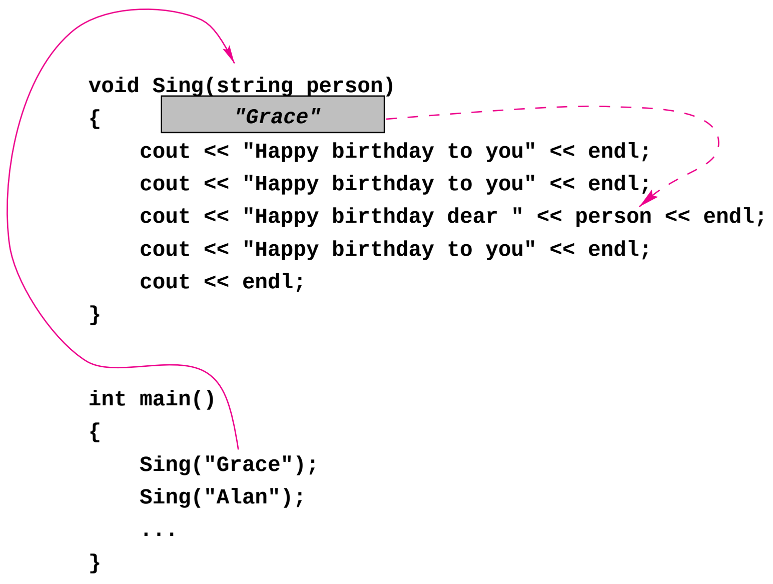
2.5.3 Passing Parameters

When the function call `Sing("Grace")` in `main` is executed, the string literal `"Grace"` is the argument **passed** to the `string` parameter *person*. This is diagrammed by the solid arrow in Figure 2.2. When the statement

```
 cout << "Happy birthday dear " << person << endl"
```

in`Sing` is executed, the parameter `person` is replaced by its value, as indicated by the dashed arrow in Figure 2.2. In this case the value is the string literal `"Grace"`. Since `person` is not between quotes, it is not a string literal. As shown in Figure 2.2, the parameter `person` is represented by a box. When the function `Sing` is called, the value that is passed to the parameter is stored in this box. Then each statement in the function is executed sequentially. After the last statement, `cout << endl`, executes, control returns from the function `Sing` to the statement that follows the function call `Sing("Grace")`. This means that the statement `Sing("Alan")` is executed next. This call passes the argument `"Alan"`, which is stored in the box associated with the parameter `person`.

If you review the format of a C++ function given in Figure 2.1, you'll see that each function has a parameter list. In a parameter list, each parameter must include the name of the parameter and the **type** of the parameter. In the definition of `Sing` the parameter is given the name *person*. The parameter has the type `string`. Recall that a string is any sequence of characters and that string literals occur between double quotes. All parameters must have an indication as to their structure—that is, what type of thing the parameter is. A parameter's type determines what kinds of things can be done with the parameter in a C++ program.



```
void Sing(string person)
{
    cout << "Happy birthday to you" << endl;
    cout << "Happy birthday to you" << endl;
    cout << "Happy birthday dear " << person << endl;
    cout << "Happy birthday to you" << endl;
    cout << endl;
}

int main()
{
    Sing("Grace");
    Sing("Alan");
    ...
}
```

Figure 2.2 Parameter passing.

The type `string` is not a built-in type in standard C++ but is made accessible by using the appropriate `#include` directive:

`#include<string>`

at the top of the program. Some older compilers do not support the standard string type. Information is given in howto C about `tstring`, an implementation of strings that can be used with older compilers. Include directives are necessary to provide information to the compiler about different types, objects, and classes used in a program, such as output streams and strings. Standard include files found in all C++ programming environments are indicated using angle brackets, as in `#include <iostream>`. Include files that are supplied by the user rather than by the system are indicated using double quotes, as in `#include "tstring.h"`.[7]

There is a vocabulary associated with all programming languages. Mastering this vocabulary is part of mastering programming and computer science. To be precise about explanations involving parameterized functions, I will use the word **parameter** to refer to usage within a function and in the function header (e.g., *person*). I will use the word **argument** to refer to what is passed to the function (e.g., *"Grace"* in the call `Sing("Grace")`.) Another method for differentiating between these two is to call the argument an **actual parameter** and to use the term **formal parameter** to refer to the

[7]The C++ standard uses header files that do not have a .h suffix, such as, iostream rather than iostream.h. We use the .h suffix for header files associated with code supplied with this book.

parameter in the function header. Here we use the adjective *formal* because the form (or type — as in `string`) of the parameter is given in the function header.

We must distinguish between the occurrence of *person* in the statement `cout << ...` and the occurrence of the string literal `"Happy birthday dear "`. Since *person* does not appear in quotes, the value of the parameter *person* is printed. If the statement `cout << "person"` was used rather than `cout << person`, the use of quotes would cause the string literal *person* to appear on the screen.

```
Happy birthday to you
Happy birthday to you
Happy birthday dear person
Happy birthday to you
```

The use of the parameter's name causes the value of the parameter to appear on the screen. The value of the parameter is different for each call of the function *Sing*. The parameter is a **variable** capable of representing values in different contexts just as the variable x can represent different values in the equation $y = 5 \cdot x + 3$.

Pause to Reflect

2.16 In the following sequence of program statements, is the string literal `"Me"` an argument or a parameter? Is it an actual parameter?

```
cout << "  A Verse for My Ego" << endl;
Sing("Me");
```

2.17 What happens with your compiler if the statement `Sing("Grace")` is changed to `Sing(Grace)`? Why?

2.18 What modifications should be made to Program 2.6 to generate a song for a person named *Bjarne*?

2.19 What modifications should be made to Program 2.6 so that each song emphasizes the personalized line by ending it with three exclamation points?

```
Happy birthday dear Bjarne !!!
```

2.20 What happens if the name of the formal parameter *person* is changed to *celebrant* in the function `Sing`? Does it need to be changed everywhere it appears?

2.21 What call of function `Sing` would generate a verse with the line shown here?

```
Happy birthday dear Mr. President
```

2.22 What is the purpose of the final statement `cout << endl;` in function `Sing` in the birthday programs?

2.23 What is a minimal change to the Happy Birthday program that will cause each verse (about one person) to be printed three times before the next verse is printed three times (rather than once each) for a total of 15 verses? What is a minimal change that will cause all five verses (for all five people) to be printed, then all five printed again, and then all five printed again for a different ordering of 15 verses?

2.24 It is possible to write the Happy Birthday program so that the body of the function `Sing` consists of a single statement. What is that statement? Can you make one statement as readable as several?

Ada Lovelace *(1816–1853)*

Ada Lovelace, daughter of the poet Lord Byron, had a significant impact in publicizing the work of Charles Babbage. Babbage's designs for two computers, the Difference Engine and the Analytical Engine,came more than a century before the first electronic computers were built but anticipated many of the features of modern computers.

Lovelace was tutored by the British mathematician Augustus De Morgan. She is characterized as *"an attractive and charming flirt, an accomplished musician, and a passionate believer in physical exercise. She combined these last two interests by practicing her violin as she marched around the family billiard table for exercise."* [McC79] Lovelace translated an account of Babbage's work into English. Her translation, and the accompanying notes, are credited with making Babbage's work accessible. Of Babbage's computer she wrote, "It would weave algebraic patterns the way the Jacquard loom weaved patterns in textiles."

Lovelace was instrumental in popularizing Babbage's work, but she was not one of the first programmers as is sometimes said. The programming language Ada is named for Ada Lovelace. For more information see [McC79, Gol93, Asp90].

2.6 Functions with Several Parameters

In this section we will investigate functions with more than one parameter. As a simple example, we'll use the children's song *Old MacDonald*, partially reproduced below. We would like to write a C++ program to generate this output.

OUTPUT

```
Old MacDonald had a farm, Ee-igh, Ee-igh, oh!
And on his farm he had a cow, Ee-igh, Ee-igh, oh!
With a moo moo here
And a moo moo there
Here a moo, there a moo, everywhere a moo moo
Old MacDonald had a farm, Ee-igh, Ee-igh, oh!

Old MacDonald had a farm, Ee-igh, Ee-igh, oh!
And on his farm he had a pig, Ee-igh, Ee-igh, oh!
With a oink oink here
And a oink oink there
Here a oink, there a oink, everywhere a oink oink
Old MacDonald had a farm, Ee-igh, Ee-igh, oh!
```

As always, we will strive to design a general program, useful in writing about, for example, ducks quacking, hens clucking, or horses neighing. In designing the program we first look for similarities and differences in the verses to determine what parts of the verses should be parameterized. We'll ignore for now the ungrammatical construct of *a oink*. The only differences in the two verses are the name of the animal, cow and pig, and the noise the animal makes, moo and oink, respectively. Accordingly, we design two functions: one to "sing" about an animal and another to "sing" about the animal's sounds, in Program 2.7, *oldmac1.cpp*.

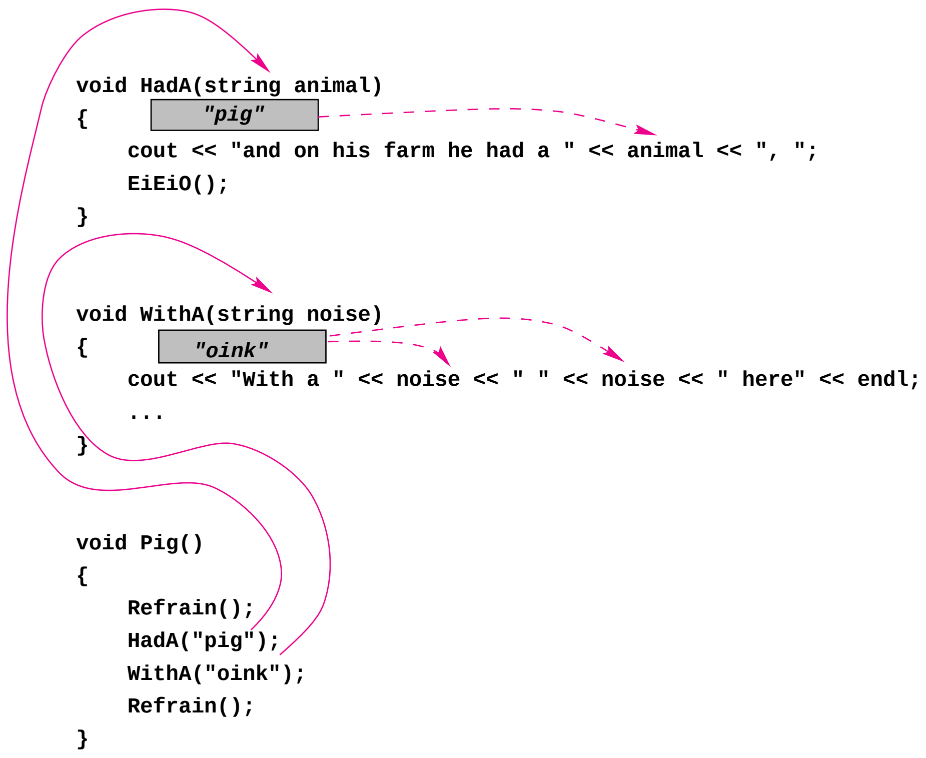
This program produces the desired output but is cumbersome in many respects. To generate a new verse (e.g., about a quacking duck) we must write a new function and call it. In contrast, in the happy-birthday-generating program (Program 2.6), a new verse could be constructed by a new call rather than by writing a new function and calling it. Also notice that the flow of control in Program 2.7 is more complex than in Program 2.6. We'll look carefully at what happens when the function call `Pig()` in `main` is executed.

Program 2.7 oldmac1.cpp

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

// working version of old macdonald, single parameter procedures

void EiEio()
{
    cout << "Ee-igh, Ee-igh, oh!" << endl;
}

void Refrain()
{
    cout << "Old MacDonald had a farm, ";
    EiEio();
}

void HadA(string animal)
{
    cout << "And on his farm he had a " << animal << ", ";
    EiEio();
}

void WithA(string noise)
// the principal part of a verse
{
    cout << "With a " << noise << " " << noise << " here" << endl;
    cout << "And a " << noise << " " << noise << " there" << endl;

    cout << "Here a " << noise << ", "
         << "there a " << noise << ", "
         << " everywhere a " << noise << " " << noise << endl;
}

void Pig()
{
    Refrain();
    HadA("pig");
    WithA("oink");
    Refrain();
}

void Cow()
{
    Refrain();
    HadA("cow");
    WithA("moo");
    Refrain();
}

int main()
{
    Cow();
    cout << endl;
    Pig();
    return 0;
}
```

oldmac1.cpp

There are four statements in the body of the function `Pig`. The first statement, the function call `Refrain()`, results in two lines being printed (note that `Refrain` calls the function `EiEiO`). When `Refrain` finishes executing, control returns to the statement following the function call `Refrain()`; this is the second statement in `Pig`, the function call `HadA("pig")`. The argument `"pig"` is passed to the (formal)



```
void HadA(string animal)
{
    cout << "and on his farm he had a " << animal << ", ";
    EiEiO();
}

void WithA(string noise)
{
    cout << "With a " << noise << " " << noise << " here" << endl;
    ...
}

void Pig()
{
    Refrain();
    HadA("pig");
    WithA("oink");
    Refrain();
}
```

Figure 2.3 Passing arguments in Old MacDonald.

parameter `animal` and then statements in the function `HadA` are executed. When the function `HadA` finishes, control returns to the third statement in `Pig`, the function call `WithA("oink")`. As shown in Figure 2.3 this results in passing the argument `"oink"`, which is stored as the value of the parameter `noise`. After all statements in the body of `WithA` have executed, the flow of control continues with the final statement in the body of the function `Pig`, another function call `Refrain()`. After this call finishes executing, `Pig` has finished and the flow of control continues with the statement following the call of `Pig()` in the main function. This is the statement `return 0` and the program finishes execution.

This program works, but it needs to be redesigned to be used more easily. This redesign process is another stage in program development. Often a programmer redesigns a working program to make it "better" in some way. In extreme cases a program that works is thrown out because it can be easier to redesign the program from scratch (using ideas learned during the original design) rather than trying to modify a program. Often writing the first program is necessary to get the good ideas used in subsequent programs.

In this case we want to dispense with the need to construct a new function rather than just a function call. To do this we will combine the functionality of the functions

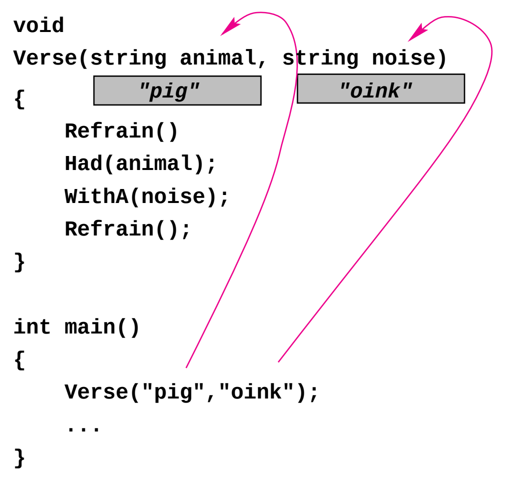
HadA and WithA into a new function Verse. When writing a program, you should look for similarities in code segments. The bodies of the functions in Pig and Cow have the same pattern:

```
Refrain()
call to HadA(...)
call to WithA(...)
Refrain()
```

Incorporating this pattern into the function Verse, rather than repeating the pattern elsewhere in the program, yields a more versatile program. In general, a programmer-defined function can have any number of parameters, but once written this number is fixed. The final version of this program, Program 2.8, is shorter and more versatile than the first version, Program 2.7 By looking for a way to combine the functionality of functions HadA and WithA, we modified a program and generated a better one. Often as versatility goes up so does length. When the length of a program decreases as its versatility increases, we're on the right track.

Program 2.8 oldmac2.cpp

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

// working version of old macdonald, functions with more than one parameter

void EiEio()
{
    cout << "Ee-igh, Ee-igh, oh!" << endl;
}

void Refrain()
{
    cout << "Old MacDonald had a farm, ";
    EiEio();
}

void HadA(string animal)
{
    cout << "And on his farm he had a " << animal << ", ";
    EiEio();
}

void WithA(string noise)
// the principal part of a verse
{
    cout << "With a " << noise << " " << noise << " here" << endl;
    cout << "And a " << noise << " " << noise << " there" << endl;

    cout << "Here a " << noise << ", "
```



```
void
Verse(string animal, string noise)
{
    Refrain()
    Had(animal);
    WithA(noise);
    Refrain();
}

int main()
{
    Verse("pig","oink");
    ...
}
```

Figure 2.4 Passing multiple arguments.

```
             << "there a " << noise << ", "
             << " everywhere a " << noise << " " << noise << endl;
}

void Verse(string animal, string noise)
{
    Refrain();
    HadA(animal);
    WithA(noise);
    Refrain();
}

int main()
{
    Verse("pig","oink");
    cout << endl;
    Verse("cow","moo");
    return 0;
}
```

oldmac2.cpp

I will sometimes use the word *elegant* as a desirable program trait. Program 2.8 is elegant compared to Program 2.7 because it is easily modified to generate new verses.

Note that it is the order in which arguments are passed to a function that determines their use, not the actual values of the arguments or the names of the parameters. This is diagrammed in Figure 2.4.

In particular, the names of the parameters have nothing to do with their purpose. If *animal* is replaced everywhere it occurs in Program 2.8 with *vegetable*, the program will produce exactly the same output. Furthermore, it is the order of the parameters in the function header and the corresponding order of the arguments in the function call that determines what the output is. In particular, the function call

```
Verse("cluck","hen");
```

would generate a verse with the lines shown below since the value of the parameter *animal* will be the string literal `"cluck"`.

O U T P U T

```
And on his farm he had a cluck, Ee-igh, Ee-igh, oh!
With a hen hen here
And a hen hen there
Here a hen, there a hen, everywhere a hen hen
```

The importance of the *order of the arguments and parameters* and the lack of importance of the names of parameters often leads to confusion. Although the use of such parameter names as *param1* and *param2* (or, even worse, *x* and *y*) might at first glance seem to be a method of avoiding such confusion, the use of parameter names that correspond roughly to their purpose is far more useful as the programs and functions we study get more complex. In general, parameters should be named according to the purpose just as functions are named. Guidelines for using lowercase and uppercase characters are provided at the end of this chapter.

Pause to Reflect

2.25 Write a function for use in Program 2.7 that produces output for a gobbling turkey. The function should be invoked by the call `Turkey`, which appears in the body of the function `main`.

2.26 Is it useful to have a separate function `EiEiO`?

2.27 How would the same effect of the function `Turkey` be achieved in Program 2.8?

2.28 If the order of the parameters of the function `Verse` is reversed so that the header is

```
void Verse(string noise, string animal)
```

but no changes are made in the body of `Verse`, then what changes (if any) must be made in the calls to `Verse` so that the output does not change?

2.29 What happens if the statement `Verse("pig","cluck");` is included in the function `main`?

2.30 The statement `Verse("lamb");` will not compile. Why?

2.31 What happens if you include the statement `Verse("owl",2)` in the function `main`? What happens if you include the statement `Verse("owl",2+2)`?

Stumbling Block

You must be careful organizing programs that use functions. Although we have not discussed the order in which functions appear in a program, the order is important to a degree. Program 2.9 is designed to print a two-line message. As written, it will not compile.

Program 2.9 order.cpp

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

// order of procedures is important

void Hi (string name)
{
    cout << "Hi " << name << endl;
    Greetings();
}

void Greetings()
{
    cout << "Things are happening inside this computer" << endl;
}

int main()
{
    Hi("Fred");
    return 0;
}
```

order.cpp

When this program is compiled using the g++ compiler, the compilation fails with the following error messages.

```
order.cpp: In function ‘void  Hi (class string)’:
order.cpp:10: warning: implicit declaration of function
                ‘Greetings’
undefined reference to ’Greetings’
collect2: ld returned 1 exit status
```

With the Turbo C++ compiler the compilation fails, with the following error message.

```
Error order.cpp 8:
Function ’Greetings’ should have a prototype
```

O U T P U T

```
Hi Fred
Things are happening inside this computer
```

(intended output, but the program doesn't run)

These messages are generated because the function `Greetings` is called from the function `Hi` but occurs physically after `Hi` in Program 2.9. In general, functions must appear (be defined) in a program before they are called.

This requirement that functions appear before they're called is too restrictive. Fortunately, there is an alternative to placing an entire function before it's called. It's possible to put information about the function before it's called rather than the function itself. This information is called the **signature** of a function, often referred to as the function's **prototype.** Rather than requiring that an entire function appear before it is called, only the prototype need appear. The prototype indicates the order and type of the function's parameters as well as the function's return type. All the functions we have studied so far have a `void` return type, but we'll see in the next chapter that functions (such as square root) can return `double` values, `int` values, `string` values, and so on. The return type, the function name, and the type and order of each parameter together constitute the prototype. The names of the parameters are not part of the prototype, but I always include the parameter names because names are useful in thinking and talking about functions.

Syntax: function prototype

return type
function name `(` *type param-name, type param-name, …*`);`

For example, The prototype for the function `Hi` is

```
void Hi (string name);
```

The prototype of the `Verse` function in Program 2.8 is different:

```
void Verse (string animal, string noise);
```

Just as arguments and parameters must match, so must a function call match the function's prototype. In the call to `Greetings` made from `Hi`, the compiler doesn't know the prototype for `Greetings`. If a function header appears physically before any call of the function, then a prototype is not needed. However, in larger programs it can be necessary to include prototypes for functions at the beginning of a program. In either case the compiler sees a function header or prototype before a function call so that the matching of arguments to parameters can be checked by the compiler.

The function `main` has a return type of `int` and the default return type in C++ is `int`. Thus the error messages generated by the g++ compiler warn of an "implicit declaration" of the function `Greetings`, meaning that the default return of an integer is assumed. Since there is no function `Greetings` with such a return type, the "undefined reference" message is generated.

The error message generated by the Turbo C++ compiler is more informative and indicates that a prototype is missing. Program 2.10 has function prototypes. Note that the prototype for function `Hi` is not necessary since the function appears before it is called. Some programmers include prototypes for all functions, regardless of whether the prototypes are necessary. In this book we use prototypes when necessary but won't include them otherwise.

Program 2.10 order2.cpp

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

// illustrates function prototypes

void Hi(string);
void Greetings();

void Hi (string name)
{
    cout << "Hi " << name << endl;
    Greetings();
}

void Greetings()
{
    cout << "Things are happening inside this computer" << endl;
}

int main()
{
    Hi("Fred");
    return 0;
}
```

order2.cpp

2.7 Program Style

The style of indentation used in the programs in this chapter is used in all programs in the book. In particular, each statement within a function or program body is indented four spaces. As programs get more complex in subsequent chapters, the use of a consistent indentation scheme will become more important in ensuring ease of understanding. I recommend that you use the indentation scheme displayed in the programs here. If you adopt a different scheme you must use it consistently.

Indentation is necessary for human readers of the programs you write. The C++ compiler is quite capable of compiling programs that have no indentation, have multiple statements per line (instead of one statement per line as we have seen so far), and that have function names like `He553323xlo3`.

2.7.1 Identifiers

The names of functions, parameters, and variables are **identifiers**—a means of referral both for program designers and for the compiler. Examples of identifiers include *Hello*, *person*, and *Sing*. Just as good indentation can make a program easier to read, I recommend the use of identifiers that indicate to some degree the purpose of the item being labeled by the identifier. As noted above, the use of *animal* is much more informative than *param1* in conveying the purpose of the parameter to which the label applies. In C++ an identifier consists of any sequence of letters, numbers, and the underscore character (_). Identifiers may not begin with a number. And identifiers are *case-sensitive* (lower- and uppercase letters): the identifier *verse* is different from the identifier *Verse*. Although some compilers limit the number of characters in an identifier, the C++ standard specifies that identifiers can be arbitrarily long.

Traditionally, C programmers use the underscore character as a way of making identifiers easier to read. Rather than the identifier `partedhair`, one would use `parted_hair`. Some recent studies indicate that using upper- and lowercase letters to differentiate the parts of an identifier can make them easier to read. In this book I adopt the convention that all programmer-defined functions and types[8] begin with an uppercase letter. Uppercase letters are also used to separate subwords in an identifier, such as, `PartedHair` rather than `parted_hair`. Parameters (and later variables) begin with lowercase letters although uppercase letters may be used to delimit subwords in identifiers. For example, a parameter for a large power of ten might be `largeTenPower`. Note that the identifier begins with a lowercase letter, which signifies that it is a parameter or a variable. You may decide that `large_ten_power` is more readable. As long as you adopt a consistent naming convention, you shouldn't feel bound by conventions I employ in the code here.

In many C++ implementations identifiers containing a double underscore (__) are used in the libraries that supply code (such as `<iostream>`), and therefore identifiers in your programs must avoid double underscores. In addition, differentiating between single and double underscores: (_ and __) is difficult.

Finally, some words have special meanings in C++ and cannot be used as identifiers. We will encounter most of these **keywords,** or **reserved** words, as we study C++. A list of keywords is provided in Table 2.1.

2.8 Chapter Review

In this chapter we studied the form of C++ programs, how a program executes, and how functions can make programs easier to modify and use. We studied programs that displayed songs having repetitive verses so that an efficient use of functions would reduce

[8] The type `string` used in this chapter is not built into C++ but is supplied as a standard type. In C++, however, it's possible to use programmer-defined types just like built-in types.

Table 2.1 C++ keywords

asm	default	for	private	struct	unsigned
auto	delete	friend	protected	switch	using
bool	do	goto	public	template	virtual
break	double	if	register	this	void
case	dynamic_cast	inline	reinterpret_cast	throw	volatile
catch	else	int	return	true	wchar_t
char	enum	long	short	try	while
class	explicit	mutable	signed	typedef	
const	extern	namespace	sizeof	typeid	
const_cast	false	new	static	typename	
continue	float	operator	static_cast	union	

our programming efforts. At the same time, the verses had sufficient variation to make the use of parameters necessary in order to develop clean and elegant programs—programs that appeal to your emerging sense of programming style.

- C++ programs have a specific form:
  - `#include` statements to access libraries
  - comments about the program
  - programmer-defined functions
  - one function named `main`
- Libraries make "off-the-shelf" programming components accessible to programmers. System library names are enclosed between < and >, as in `<iostream>`. Libraries that are part of this book and nonsystem libraries are enclosed in double quotation marks, as in `"tstring.h"`.
- Output is generated using the insertion operator, `<<`, and the standard output stream, `cout`. These are accessible by including the proper header file `<iostream>`.
- Strings are sequences of characters. The type `string` is not a built-in type but is accessible via the header file `<string>`.
- Functions group related statements together so that the statements can be executed together, by calling the function.
- Parameters facilitate passing information between functions. The value passed is an *argument.* The "box" that stores the value in the function is a *parameter.*
- Iterative enhancement is a design process by which a program is developed in stages. Each stage is both an enhancement and a refinement of a working program.
- In designing programs, look for patterns of repeated code that can be combined into a parameterized function to avoid code duplication, as we did in `Verse` of Program 2.8.
- Prototypes are function signatures that convey to the compiler information that is used to determine if a function call is correctly formed.

- Identifiers are names of functions, variables, and parameters. Identifiers should indicate the purpose of what they name. Your programs will be more readable if you are consistent in capitalization and underscores in identifiers.

2.9 Exercises

2.1 Add a function `Neck` to *parts.cpp*, Program 2.4 to generate output similar to that shown below.

OUTPUT

```
  ||||||||||||||||
  |   o      o   |
 _|              |_
|_                _|
  |              |
  |   |______|   |
  |_____    _____|
       |    |
```

2.2 Modify the appropriate functions in Program 2.4 to display the head shown below.

OUTPUT

```
||||||||||||||||
|  __      __  |
| !  ! __ !  ! |
| !o !/  \!o ! |
| !__!    !__! |
|              |
|    ///|\\\   |
 \            /
  \     o    /
   \________/
```

2.3 Write a program whose output is the text of *hello.cpp*, Program 2.1. Note that the output is a program!

OUTPUT

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

int main()
{
    cout << "Hello world" << endl;
    return 0;
}
```

To display the character " you'll need to use an **escape sequence.** An escape sequence is a backslash \ followed by one character. The two-character escape sequence represents a single character; the escape sequence \" is used to print one quotation mark. The statement

```
cout << "\"Hello\" " << endl;
```

can be used to print the characters "Hello" on the screen, including the quotation marks! Be sure to comment your program-writing program appropriately.

2.4 A popular song performed by KC and the Sunshine Band repeats many verses using the words "That's the way Uh-huh Uh-huh I like it Uh-huh Uh-huh," as shown below.

OUTPUT

```
That's the way
Uh-huh Uh-huh
I like it
Uh-huh Uh-huh

That's the way
Uh-huh Uh-huh
I like it
Uh-huh Uh-huh
```

Write a program that generates four choruses of the song.

2.5 Write a program that generates the verses of a children's song shown below. Don't worry about the ungrammatical qualities inherent in the use of "goes" and "go" in your first attempt at writing the program. You should include a function with two parameters capable of generating any of the verses when the appropriate arguments are passed. Strive to make your program "elegant."

OUTPUT

```
The wheel on the bus goes round round round
round round round
round round round
The wheel on the bus goes round round round
All through the town

The wipers on the bus goes swish swish swish
swish swish swish
swish swish swish
The wipers on the bus goes swish swish swish
All through the town

The horn on the bus goes beep beep beep
beep beep beep
beep beep beep
The horn on the bus goes beep beep beep
All through the town

The money on the bus goes clink clink clink
clink clink clink
clink clink clink
The money on the bus goes clink clink clink
All through the town
```

Is it possible to generate a verse of the song based on the lines

```
The driver on the bus goes move on back
move on back
move on back
```

with small modifications? How many parameters would the `Verse` function of such a song have?

2.6 Consider the song about an old woman with an insatiable appetite, one version of which is partially reproduced in the following.

OUTPUT

```
There was an old lady who swallowed a fly
I don’t know why she swallowed a fly
Perhaps she’ll die.

There was an old lady who swallowed a spider
That wiggled and jiggled and tiggled inside her
She swallowed the spider to catch the fly
I don’t know why she swallowed a fly
Perhaps she’ll die.

There was an old lady who swallowed a bird
How absurd to swallow a bird
She swallowed the bird to catch the spider
That wiggled and jiggled and tiggled inside her
She swallowed the spider to catch the fly
I don’t know why she swallowed a fly
Perhaps she’ll die.
```

This song may be difficult to generate via a program using just the predefined output stream `cout`, the operator `<<`, and programmer-defined parameterized functions. Write such a program or sketch its solution and indicate why it might be difficult to write a program for which it is easy to add new animals while maintaining program elegance. You might think about adding a verse about a cat (imagine that!) that swallows the bird.

2.7 In a song made famous by Bill Haley and the Comets, the chorus is

```
One, two, three o’clock, four o’clock rock
Five, six, seven o’clock, eight o’clock rock
Nine, ten, eleven o’clock, twelve o’clock rock
We’re going to rock around the clock tonight
```

Rather than using words to represent time, you are to use numbers and write a program that will print the chorus above but with the line

```
1, 2, 3  o’clock, 4  o’clock rock
```

as the first line of the chorus. Your program should be useful in creating a chorus that could be used with military time; i.e., another chorus might end thus:

```
21, 22, 23 o’clock, 24 o’clock rock
We’re going to rock around the clock tonight
```

You should use the arithmetic operator + where appropriate and strive to make your program as succinct as possible, calling functions with different parameters rather than writing similar statements.