3 Program Design and Implementation

GIGO—Garbage In, Garbage Out
Common computer aphorism

GIGO—Garbage In, Gospel Out
New Hacker's Dictionary

Civilization advances by extending the number of important operations which we can perform without thinking about them.
Alfred North Whitehead
An Introduction to Mathematics

The memory of all that— No, no! They can't take that away from me.
Ira Gershwin
They Can't Take That Away from Me

The song-writing and head-drawing programs in Chapter 2 generated the same output for all executions unless the programs were modified and recompiled. These programs do not respond to a user of the program at **run time,** meaning while the programs are running or executing. The solutions to many programming problems require input from program users during execution. Therefore, we must be able to write programs that process input during execution. A typical framework for many computer programs is one that divides a program's execution into three stages.

1. Input—information is provided to the program.
2. Process—the information is processed.
3. Output—the program displays the results of processing the input.

This **input/process/output (IPO)** model of programming is used in the simple programs we'll study in this chapter as well as in million-line programs that forecast the weather and predict stock market fluctuations. Breaking a program into parts, implementing the parts separately, and then combining the parts into a working program is a good method for developing programs. This is often called **divide and conquer;** the program is divided into pieces, each piece is implemented, or "conquered," and the final program results from combining the conquered pieces. We'll employ divide and conquer together with iterative enhancement when designing classes and programs.

3.1 The Input Phase of Computation

In this chapter we'll discuss how the user can input values that are used in a program. These input values can be strings like the name of an animal or the noise the animal makes, as we'll see in Program 3.1, *macinput.cpp*. The input values can also be numbers like the price and diameter of a pizza as we'll see in Program 3.5, *pizza.cpp*.

Two runs of a modified version of Program 2.8, *oldmac2.cpp* are in the following output box. Input entered by the user (you) is shown in a bold-italic font. The computing environment displays `prompt>` as a cue to the user to enter a command—in this case, the name of a program. Prompts may be different in other computing environments. You may be using a programming environment in which the program is run using a menu-driven system rather than a command-line prompt, but we'll use the prompt to show the name of the program generating the output.

OUTPUT

```
prompt> macinput

Enter the name of an animal: cow
Enter noise that a cow makes: moo

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!

prompt> macinput

Enter the name of an animal: hen
Enter noise that a cow makes: cluck

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

Each run of the program produces different output according to the words you enter. If the function `main` in Program 2.8 is modified as shown in the code segment in Program 3.1, the modified program generates the runs shown above.

Program 3.1 macinput.cpp

```
//  see program oldmac2.cpp for function Verse and #includes


int main()
{
    string animal;
    string noise;

    cout << "Enter the name of an animal: ";
    cin >> animal;

    cout << "Enter noise that a " << animal << " makes: ";
    cin >> noise;

    cout << endl;
    Verse(animal,noise);
    return 0;
}
```

macinput.cpp

3.1.1 The Input Stream, `cin`

We'll investigate each statement in `main` of Program 3.1 When you run the program, you enter information and the program reacts to that information by printing a verse of Old MacDonald's Farm that corresponds to what you enter. In C++, information you enter comes from the input stream `cin` (pronounced "cee-in"). Just as the output stream, `cout`, generates output, the input stream accepts input values used in a program. In the run of Program 3.1, the output statement

```
cout << "Enter the name of an animal: "
```

is *not* followed by an `endl`. As a result, your input appears on the same line as the words that prompt you to enter an animal's name.

When you enter input, it is taken from the input stream using the **extraction** operator, `>>` (sometimes read as "takes-from"). When the input is taken, it must be stored someplace. Program **variables,** in this case `animal` and `noise,` are used as a place for storing values.

3.1.2 Variables

The following statements from Program 3.1 **define** two `string` variables, named *animal* and *noise*.

```
string animal;
string noise;
```

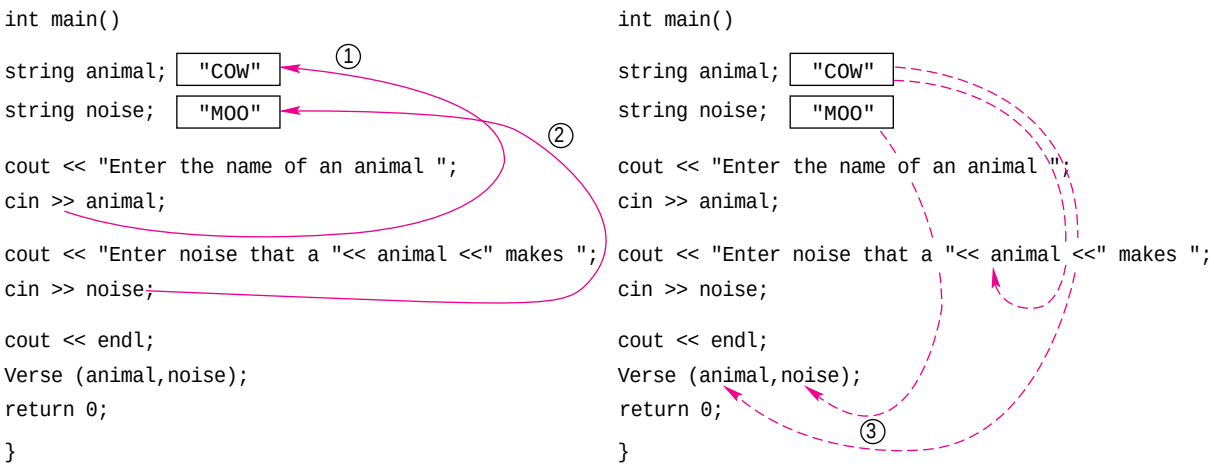


Figure 3.1 Using variables andstreams for input.

These variables are represented in Figure 3.1 as boxes that store the variable values in computer memory. The value stored in a variable can be used just as the values stored in a function's formal parameters can be used within the function (see Figure 2.3). Parameters and variables are similar; each has a name such as `animal` or `noise` and an associated storage location. Parameters are given initial values, or **initialized,** by calling a function and passing an argument. Variables are often initialized by accepting input from the user.

Variables in a C++ program must be **defined** before they can be used. Sometimes the terms *allocate* and *create* are used instead of *define.* Sometimes the word **object** is used instead of *variable.* You should think of *variable* and *object* as synonyms. Just as all formal parameters have a type or class, all variables in C++ have a type or class that determines what kinds of operations can be performed with the variable. The variable `animal` has the type or class `string`. In this book we'll define each variable in a separate statement as was done in Program 3.1. It's possible to define more than one variable in a single statement. For example, the following statement defines two `string` variables.

Syntax: variable definition

type name; **OR**
type $name_1$, $name_2$,…, $name_k$;

```
string animal,noise;
```

In the run of Program 3.1 diagrammed in Figure 3.1, values taken from the input stream are stored in a variable's memory location. The variable `animal` gets a value in the statement labeled 1; the variable `noise` gets a value in the statement labeled 2. The value of `animal` is used to prompt the user; this is shown by the dashed arrow. The arrow labeled 3 shows the values of both variables used as arguments to the function

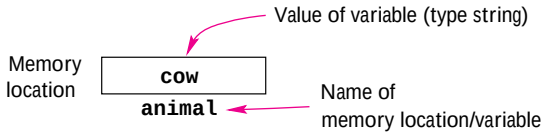


Figure 3.2 Variables as named memory locations.

`Verse`. In the interactive C++ environments used in the study of this book, the user must almost always press the return (enter) key before an input statement completes execution and stores the entered value in `animal`. This allows the user to make corrections (using arrow keys or a mouse, for example) before the final value is stored in memory.

An often-used metaphor associates a variable with a mailbox. Mailboxes usually have names associated with them (either 206 Main Street, or the Smith residence) and offer a place in which things can be stored. Perhaps a more appropriate metaphor associates variables with dorm rooms.[1] For example, a room in a fraternity or sorority house (say, ΨΥ or ΔΔΔ) can be occupied by any member of the fraternity or sorority but *not* by members of other residential groups.[2] The occupant of the room may change just as the value of a variable may change, but the type of the occupant remains the same, just as a variable's type remains fixed once it is defined. Thus we think of variables as named memory storage locations capable of storing a specific type of object. In the foregoing example the name of one storage location is `animal` and the type of object that can be stored in it is a `string`; for example, the value `cow` can be stored as shown in Figure 3.2.

In C++, variables can be defined anywhere, but they must be defined before they're used. Some programmers prefer to define all variables immediately after a left brace, {. Others define variables just before they're first used. (We'll have occasion to use both styles of definition.) When all variables are defined at the beginning of a function, it is easy to find a variable when reading code. Thus when one variable is used in many places, this style makes it easier to find the definition than searching for the variable's first use. Another version of the code in Program 3.1 is shown in the following block of code with an alternate style of variable definition:

```
int main()
{
    cout << "Enter the name of an animal ";
    string animal;
    cin >> animal;

    cout << "Enter noise that a " << animal << " makes ";
    string noise;
```

[1] This was suggested by Deganit Armon.

[2] The room could certainly not be occupied by independents or members of the opposite sex except in the case of co-ed living groups.

```
    cin >> noise;

    cout << endl;
    Verse(animal,noise);
    return 0;
}
```

Before the statement `cin >> animal` in Program 3.1 is executed, the contents of the memory location associated with the variable `animal` are undefined. You can think of an undefined value as garbage. Displaying an undefined value probably won't cause any trouble, but it might not make any sense. In more complex programs, accessing an undefined value can cause a program to crash.

Program Tip 3.1: When a variable is defined give it a value. Every variable must be given a value before being used for the first time in an expression or an output statement, or as an argument in a function call.

One way of doing this is to define variables just before they're used for the first time; that way you won't define lots of variables at the beginning of a function and then use one before it has been given a value. Alternatively, you can define all variables at the beginning of a function and program carefully.

Pause to Reflect

3.1 If you run Program 3.1, *macinput.cpp*, and enter *baah* for the name of the animal and *sheep* for the noise, what is the output? What happens if you enter *dog* for the name of the animal and *bow wow* for the noise (you probably need to run the program to find the answer)? What if *bow-wow* is entered for the noise?

3.2 Why is there no `endl` in the statement prompting for the name of an animal and why is there a space after the ell in `animal`?

```
cout << "Enter the name of an animal ";
```

3.3 Write a function *main* for Program 2.5 (the Happy Birthday program) that prompts the user for the name of a person for whom the song will be "sung."

3.4 Add statements to the birthday program as modified in the previous exercise to prompt the user for how old she is, and print a message about the age after the song is printed.

3.5 What happens if the statement `cin >> noise;` is removed from Program 3.1 and the program is run?

John Kemeny *(1926–1992)*

John Kemeny, with Thomas Kurtz, invented the programming language BASIC (Beginner's All-purpose Symbolic Instruction Code). The language was designed to be simple to use but as powerful as FORTRAN, one of the languages with which it competed when first developed in 1964. BASIC went on to become the world's most popular programming language.

Kemeny was a research assistant to Albert Einstein before taking a job at Dartmouth College. At Dartmouth he was an early visionary in bringing computers to everyone. Kemeny and Kurtz developed the Dartmouth Time Sharing System, which allowed hundreds of users to use the same computer "simultaneously." Kemeny was an inspiring teacher. While serving as president of Dartmouth College he still taught at least one math course each year. With a cigarette in a holder and a distinct, but very understandable, Hungarian accent, Kemeny was a model of clarity and organization in the classroom.

In a book published in 1959, Kemeny wrote the following, comparing computer calculations with the human brain. It's interesting that his words are still relevant more than 35 years later.

> *When we inspect one of the present mechanical brains we are overwhelmed by its size and its apparent complexity. But this is a somewhat misleading first impression. None of these machines compare with the human brain in complexity or in efficiency. It is true that we cannot match the speed or reliability of the computer in multiplying two ten-digit numbers, but, after all, that is its primary purpose, not ours. There are many tasks that we carry out as a matter of course that we would have no idea how to mechanize.*

For more information see [Sla87, AA85]

3.2 Processing Numbers

All the examples we've studied so far have used strings. Although many programs manipulate strings and text, numbers are used extensively in computing and programming. In this section we'll discuss how to use numbers for input, processing, and output. As we'll see, the syntax for the input and output of numbers is the same as for strings, but processing numbers requires a new set of symbols based on those you learned for

ordinary math.

We'll start with a simple example, but we'll build towards the programming knowledge we need to write a program that will help us determine what size pizza is the best bargain. Just as printing "Hello World" is often used as a first program, programs that convert temperature from Fahrenheit to Celsius are commonly used to illustrate the use of numeric literals and variables in C++ programs[3]. Program 3.2 shows how this is done. The program shows two different types of numeric values and how these values are used in doing arithmetic in C++ programs.

Program 3.2 fahrcels.cpp

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

// illustrates i/o of ints and doubles
// illustrates arithmetic operations

int main()
{
    int ifahr;
    double dfahr;

    cout << "enter a Fahrenheit temperature ";
    cin >> ifahr;
    cout << ifahr << " = "
         << (ifahr - 32) * 5/9
         << " Celsius" << endl;

    cout << "enter another temperature ";
    cin >> dfahr;
    cout << dfahr << " = "
         << (dfahr - 32.0) * 5/9
         << " Celsius" << endl;

    return 0;
}
```

fahrcels.cpp

OUTPUT

```
prompt> fahrcels
enter a Fahrenheit temperature 40
40 = 4 Celsius
enter another temperature 40
40 = 4.44444 Celsius
```

[3]Note, however, that using a computer program to convert a single temperature is probably overkill. This program is used to study the types `int` and `double` rather than for its intrinsic worth.

Two variables are defined in Program 3.2, `ifahr` and `dfahr`. The type of `ifahr` is `int` which represents an integer in C++, what we think of mathematically as a value from the set of numbers $\{\ldots -3, -2, -1, 0, 1, 2, 3 \ldots\}$. The type of `dfahr` is `double` which represents in C++ what is called a **floating-point** number in computer science and a *real number* in mathematics. Floating-point numbers have a decimal point; examples include $\sqrt{17}$, 3.14159, and 2.0. In Program 3.2 the input stream `cin` extracts an integer value entered by the user with the statement `cin >> ifahr` and stores the entered value in the variable `ifahr`. A floating-point number entered by the user is extracted and stored in the variable `dfahr` by the statement `cin >> dfahr`. Except for the name of the variable, both these statements are identical in form to the statements in Program 3.1 that accepted strings entered by the user. When writing programs using numbers, the type `double` should be used for all variables and calculations that might have decimal points[4]. The type `int` should be used whenever integers, or numbers without decimal points, are appropriate.

3.2.1 Numeric Data

Although there is no largest integer in mathematics, the finite memory of a computer limits the largest and smallest `int` values in C++. On computers using 16-bit compilers, the values of an `int` can range from −32,768 to 32,767. When more modern 32-bit compilers are used, the typical range of `int` values is −2,147,483,648 to 2,147,483,647. You shouldn't try to remember these numbers, you should remember that there are limits. The smaller range of `int` values is really too small to do many calculations. For example, the number of seconds in a day is 86,400, far exceeding the value that can be stored in an `int` using C++ on most 16-bit compilers. To alleviate this problem the type **long int** should be used instead of `int`. The variable `ifahr` could be defined to use this modified `long int` type as `long int ifahr`. The type `long int` is usually abbreviated simply as `long`. This makes `long secs;` a definition for a variable `secs`.

Program 3.3 shows the limitations of the type `int`. The first run after the program listing is generated using a 32-bit compiler. The same run on a computer using a 16-bit compiler generates a much different set of results, as shown.

Program 3.3 daysecs.cpp

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

// converts days to seconds
// illustrates integer overflow

int main()
{
    int days;
```

[4]The type `float` can also be used for floating-point numbers. We will not use this type, since most standard mathematical functions use `double` values. Using the type `float` will almost certainly lead to errors in any serious mathematical calculations.

```
    cout << "how many days: ";
    cin >> days;
    cout << days << " days = "
         << days*24*60*60
         << " seconds" << endl;

    return 0;
}
```

daysecs.cpp

OUTPUT

```
prompt> daysecs
how many days: 31
31 days = 2678400 seconds
prompt> daysecs
how many days: 365
365 days = 31536000 seconds
prompt> daysecs
how many days: 13870
13870 days = 1198368000 seconds
```

OUTPUT

run using a 16-bit compiler

```
prompt> daysecs
how many days: 31
31 days = -8576 seconds
prompt> daysecs
how many days: 365
365 days = 13184 seconds
prompt> daysecs
how many days: 13870
13870 days = -23296 seconds
```

If the definition `int days` is changed to `long days`, then the runs will be the same on both kinds of computers.

Program Tip 3.2: Use `long` (`long int`) rather than `int` if you are using a 16-bit compiler. This will help ensure that the output of any program you write using integer arithmetic is correct.

It's also possible to use the type `double` instead of either `int` or `long int`. In mathematics, real numbers can have an infinite number of digits after a decimal point. For example, $1/3 = 0.333333\ldots$ and $\sqrt{2} = 1.41421356237\ldots$, where there is no pattern to the digits in the square root of two. Data represented using `double` values are approximations since it's not possible to have an infinite number of digits. When the definition of `days` is changed to `double days` the program generates the same results with 16- or 32-bit compilers.

OUTPUT

```
prompt> daysecs
how many days: 31
31 days = 2.6784e+06 seconds
prompt> daysecs
how many days: 365
365 days = 3.1536e+07 seconds
prompt> daysecs
how many days: 13870
13870 days = 1.19837e+09 seconds
```

The output in this run is shown using **exponent,** or **scientific,** notation. The expression `2.6784e+06` is equivalent to 2,678,400. The `e+06` means "multiply by 10^6." The same run results if the definition `int days` is used, but the output statement is changed as shown below.

```
cout << days*24.0*60*60 << " seconds" << endl;
```

We'll explore why this is the case in the next section. In Howto B you can see examples that show how to format numeric output so that, for example, the number of digits after the decimal place can be specified in your programs.

3.2.2 Arithmetic Operators

Although the output statements in *fahrcels.cpp*, Program 3.2, are the same except for the name of the variable storing the Fahrenheit temperature, the actual values output by the statements are different. This is because arithmetic performed using `int` values behaves differently than arithmetic performed using `double` values. An **operator,** such as `+`, is used to perform some kind of computation. Operators combine **operands** as in $15 + 3$; the operands are 15 and 3. An **expression** is a sentence composed of operands

Table 3.1 The Arithmetic Operators

Symbol	Meaning	Example
*	multiplication	`3*5*x`
/	division	`5.2/1.5`
%	mod/remainder	`7 % 2`
+	addition	`12 + x`
-	subtraction	`35 - y`

and operators, as in $(X - 32) * 5/9$. In this expression, X, 32, 5, and 9 are operands. The symbols `-`, *, and / are operators.

To understand why different output is generated by the following two expressions when the same value is entered for both `ifahr` and `dfahr`, we'll need to explore how arithmetic expressions are evaluated and how evaluation depends on the types of the operands.

```
(ifahr - 32) * 5/9          (dfahr - 32.0) * 5/9
```

The division operator `/` yields results that depend on the types of its operands. For example, what is `7/2`? In mathematics the answer is 3.5, but in C++ the answer is 3. This is because division of two integer quantities (in this case, the literals 7 and 2) is defined to yield an integer. The value of `7.0/2` is 3.5 because division of `double` values yields a `double`. When an operator has more than one use, the operator is **overloaded.** In this case the division operator is overloaded since it works differently with `double` values than with `int` values.

The arithmetic operators available in C++ are shown in Table 3.1. Most should be familiar to you from your study of mathematics except, perhaps, for the modulus operator, %. The modulus operator % yields the remainder when one integer is divided by another. For example, executing the statement

```
cout << "47 divides 1347 " << 1347/47 << " times, "
     << "with remainder " << 1347 % 47 << endl;
```

would generate the following output because $1347 = 28 * 47 + 31$.

OUTPUT

```
47 divides 1347 28 times, with remainder 31
```

In general the result of `p % q` (read this as "p mod q") for two integers should be a value `r` with the property that `p = x*q + r` where $x = p/q$. The % operator is often used to determine if one integer divides another—that is, divides with no remainder, as in $4/2$ or $27/9$. If `x % y = 0`, there is a remainder of zero when `x` is divided by `y`,

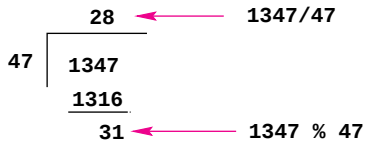


Figure 3.3 Using the modulus operator.

indicating that `y` evenly divides `x`. The following examples illustrate several uses of the modulus operator. A calculation showing the modulus operator and how it relates to remainders is diagrammed in Figure 3.3.

```
25 % 5 = 0          13 % 2 = 1          4 % 3 = 1
25 % 6 = 1          13 % 3 = 1          4 % 4 = 0
48 % 8 = 0          13 % 4 = 1          4 % 5 = 4
48 % 9 = 3          13 % 5 = 3          5 % 4 = 1
```

If either `p` or `q` is negative, however, the value calculated may be different on different systems.

Program Tip 3.3: Avoid negative values when using the % operator, or check the documentation of the programming environment you use. In theory, the result of a modulus operator should be positive since it is a remainder. In practice the result is usually negative and not the result you expect when writing code. The C++ standard requires that `a = ((a/b) * b) + (a % b)`.

3.2.3 Evaluating Expressions

The following rules are used for evaluating arithmetic expressions in C++ (these are standard rules of arithmetic as well):

1. Evaluate all parenthesized expressions first, with nested expressions evaluated "inside-out."
2. Evaluate expressions according to **operator precedence:** evaluate `*`, `/`, and `%` before `+` and `-`.
3. Evaluate operators with the same precedence left to right—this is called left-to-right **associativity.**

We'll use these rules to evaluate the expression `(ifahr - 32) * 5/9` when `ifahr` has the value 40 (as in the output of Program 3.2). Tables showing precedence rules and associativity of all C++ operators are given in Howto A, see Table A.4.

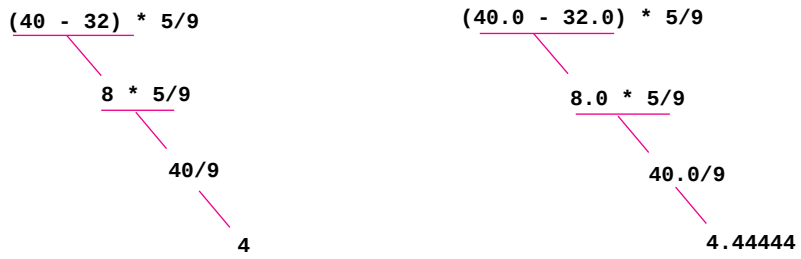


Figure 3.4 Evaluating arithmetic expressions.

- Evaluate `(ifahr - 32)` first; this is 40 − 32, which is 8. (This is rule 1 above: evaluate parenthesized expressions first).
- The expression is now `8 * 5/9` and `*` and `/` have equal precedence so are evaluated left to right (rule 3 above). This yields 40/9, which is 4.

In the last step above 40/9 evaluates to 4. This is because in integer division any fractional part is truncated, or removed. Thus although $40/9 = 4.444\ldots$ mathematically, the fractional part .444 . . . is truncated, leaving 4.

At this point it may be slightly mysterious why Program 3.2 prints 4.44444 when the expression `(dfahr - 32.0) * 5/9` is evaluated. The subexpression `(dfahr - 32.0)` evaluates to the real number 8.0 rather than the integer 8. The expression `(dfahr - 32)` would evaluate to 8.0 as well because subtracting an `int` from a `double` results in a `double` value. Similarly, the expression `8.0 * 5/9` evaluates to 40.0/9, which is 4.44444, because when `/` is used with `double` values or a mixed combination of `double` and `int` values, the result is a double. The evaluation of both expressions from Program 3.2 is diagrammed in Figure 3.4.

This means that if the first `cout <<` statement in Program 3.2 is modified so that the 5 is replaced by 5.0, as in `(ifahr - 32) * 5.0/9`, then the expression will evaluate to 4.44444 when 40 is entered as the value of `ifahr` because 5.0 is a `double` whereas 5 is an `int`.

Program 3.4 express.cpp

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

// illustrates problems with evaluating
// arithmetic expressions

int main()
{
    double dfahr;
```

```
    cout << "enter a Fahrenheit temperature ";
    cin >> dfahr;
    cout << dfahr << " = "
         << 5/9 * (dfahr - 32.0)
         << " Celsius" << endl;

    return 0;
}
```

express.cpp

OUTPUT

```
prompt> express
enter a Fahrenheit temperature 40.0
40.0 = 0 Celsius
prompt> express
enter a Fahrenheit temperature 37.33
37.33 = 0 Celsius
```

Often arithmetic is done by specialized circuitry built to add, multiply, and do other arithmetic operations. The circuitry for `int` operations is different from the circuitry for `double` operations, reflecting the different methods used for multiplying integers and reals. When numbers of different types are combined in an arithmetic operation, one circuit must be used. Thus when `8.0 * 5` is evaluated, the 5 is **converted** to a `double` (and the double circuitry would be used). Sometimes the word **promoted** is used instead of *converted*.

Stumbling Block

Pitfalls with evaluating expressions. Because arithmetic operators are overloaded and because we're not used to thinking of arithmetic as performed by computers, some expressions yield results that don't meet our expectations. Referring to Program 3.4, we see that in the run of *express.cpp* the answer is 0, because the value of the expression `5/9` is 0 since integer division is used. It might be a better idea to use 5.0 and 9.0 since the resulting expression should use `double` operators. If an arithmetic expression looks correct to you but it yields results that are not correct, be sure that you've used parentheses properly, that you've taken `double` and `int` operators into account, and that you have accounted for operator precedence.

Pause to Reflect

3.6 If the output expressions in Program 3.2 are changed so that subexpressions are enclosed in parentheses as shown, why do both statements print zero?

```
(ifahr - 32) * (5/9)
```

3.7 What is printed if parentheses are not used in either of the expressions in Program 3.2?

```
ifahr - 32 * 5/9
```

3.8 If the expression using `ifahr` is changed as shown what will the output be if the user enters 40? Why?

```
(ifahr - 32.0) * 5/9
```

3.9 What modifications are needed to change Program 3.2 so that it converts degrees Celsius to Fahrenheit rather than vice versa?

3.10 If *daysecs.cpp*, Program 3.3, is used with the definition `long day`, but the output is changed to `cout << 24*60*60*days << endl`, then the program behavior with a 16-bit compiler changes as shown here. The output is incorrect. Explain why the change in the output statement makes a difference.

OUTPUT

```
prompt> daysecs
how many days: 31
31 days = 646784 seconds
prompt> daysecs
how many days: 365
365 days = 7615360 seconds
```

3.11 The quadratic formula, which gives the roots of a quadratic equation, is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The roots of $2x^2 - 8x + 6$ should be 3 and 1, where $a = 2$, $b = -8$, $c = 6$, and $\sqrt{b^2 - 4ac} = 4$. Explain why the statements below print 12 and 4 instead of 3 and 1.

```
cout << (8 + 4)/2*2 << endl;
cout << (8 - 4)/2*2 << endl;
```

3.2.4 The type char

The individual letters used to construct C++ `string` values are called characters; the type `char` is used to represent characters. The type `char` is actually an integral type, in many ways `char` values act like `int` values. This means it's possible to add `char` values, though we'll avoid using characters as though they were integers. We'll use

`char`s almost exclusively as a way to build `string` values, and we'll study how to do this in later chapters.

A `char` does print differently than an integer, otherwise it can be used like an integer. Single quotes (apostrophes) are used to indicate `char` values, e.g., `'a'`, `'!'`, and `'Z'` are all valid C++ characters.

3.3 Case Study: Pizza Slices

In this section we'll look at one program in some detail. The program uses the types `int` and `double` in calculating several statistics about different sizes of pizza. The program might be used, for example, to determine whether a 10-inch pizza selling for $10.95 is a better buy than a 14-inch pizza selling for $14.95.

3.3.1 Pizza Statistics

Pizzas can be ordered in several sizes. Some pizza parlors cut all pizzas into eight slices, whereas others get more slices out of larger pizza pies. In many situations it would be useful to know what size pie offers the best deal in terms of cost per slice or cost per square inch of pizza. Program 3.5, *pizza.cpp*, provides a first attempt at a program for determining information about pizza prices.

Program 3.5 pizza.cpp

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

// find the price of one slice of pizza
// and the price per square inch

void SlicePrice(int radius, double price)
// compute pizza statistics
{
    // assume all pizzas have 8 slices

    cout << "sq in/slice = ";
    cout << 3.14159*radius*radius/8 << endl;

    cout << "one slice: $" << price/8 << endl;
    cout << "$" << price/(3.14159*radius*radius);
    cout << " per sq. inch" << endl;
}

int main()
{
    int radius;
    double price;
    cout << "enter radius of pizza ";
    cin >> radius;
```

```
    cout << "enter price of pizza ";
    cin >> price;

    SlicePrice(radius,price);

    return 0;
}
```

pizza.cpp

OUTPUT

```
prompt> pizza
enter radius of pizza 8
enter price of pizza 9.95
sq in/slice = 25.1327
one slice: $1.24375
$0.0494873 per sq. inch

prompt> pizza
enter radius of pizza 10
enter price of pizza 11.95
sq in/slice = 39.2699
one slice: $1.49375
$0.0380381 per sq. inch
```

The function `SlicePrice` is used for both the processing and the output steps of computation in *pizza.cpp*. The input steps take place in `main`. Numbers entered by the user are stored in the variables `radius` and `price` defined in `main`. The values of these variables are sent as **arguments** to `SlicePrice` for processing. This is diagrammed in Figure 3.5.

If the order of the arguments in the call `SlicePrice(radius,price)` is changed to `SlicePrice(price,radius)`, the compiler issues a warning:

```
pizza.cpp: In function ‘int main()’:
pizza.cpp:30: warning: ‘double’ used for argument 1 of
            ‘SlicePrice(int, double)’
```

It’s not generally possible to pass a `double` value to an `int` parameter without losing part of the value, so the compiler issues a warning. For example, passing an argument of 11.95 to the parameter `radius` results in a value of 11 for the parameter because `double` values are truncated when stored as integers. This is called **narrowing**. Until we discuss how to convert values of one type to another type, you should be sure that the type of an argument matches the type of the corresponding formal parameter. Since

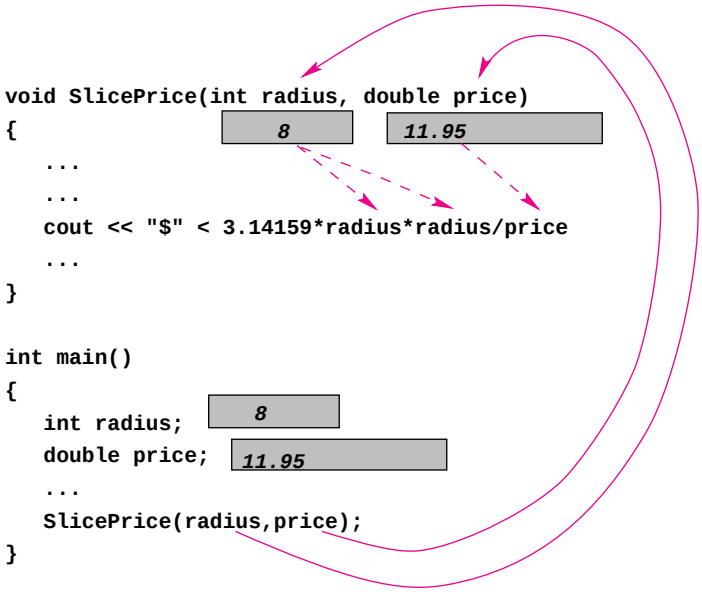


Figure 3.5 Passing arguments.

different types may use different amounts of storage and may have different internal representations in the computer, it is a good idea to ensure that types match properly.

Program Tip 3.4: Pay attention to compiler warnings. When the compiler issues a warning, interpret the warning as an indication that your program is not correct. Although the program may still compile and execute, the warning indicates that something isn't proper with your program.

The area of a circle is given by the formula $\pi \times r^2$, where r is the radius of the circle. In `SlicePrice` the formula determines the number of square inches in a slice and the price per square inch; the parentheses used to compute the price per square inch are necessary.

```
cout << "$" << price/(3.14159*radius*radius)
```

If parentheses are not used, the rules for evaluating expressions lead to a value of $380.381 per square inch for a 10-inch pizza costing $11.95. The value of `price/3.14159` is multiplied by 10 twice—the operators `/` and `*` have equal precedence and are evaluated from left to right. In the exercises you'll modify this program so that a user can enter the number of slices as well as other information. Such changes make the program useful in more settings.

3.4 Classes and Types: An Introduction

The types `int` and `double` are built-in types in C++, whereas `string` is a class. In object-oriented programming terminology, user-defined types are often called **classes.** Although some people make a distinction between the terms *type* and *class*, we'll treat them as synonyms. The term *class* is apt as indicated by the definition below from the *American Heritage Dictionary:*

> **class** 1. A set, collection, group, or configuration containing members having or thought to have at least one attribute in common

All variables of type, or class, `string` share certain attributes that determine how they can be used in C++ programs. As we've seen in several examples, the types `int` and `double` represent numbers with different attributes. In the discussion that follows, I'll sometimes use the word **object** instead of the word *variable.* You should think of these as synonyms. The use of classes in object-oriented programming gives programmers the ability to write programs using off-the-shelf components. In this section we'll examine a programmer-defined class that simulates a computer-guided hot-air balloon as shown in Program 3.6; the graphical output is shown as Figure 3.6.

Program 3.6 gfly.cpp

```
#include <iostream>
using namespace std;
#include "gballoon.h"

// auto-pilot guided balloon ascends, cruises, descends

int main()
{
    Balloon b(MAROON);
    int rise;               // how high to fly     (meters)
    int duration;           // how long to cruise  (seconds)

    cout << "Welcome to the windbag emporium." << endl;
    cout << "You'll rise up, cruise a while, then descend." << endl;
    cout << "How high (in meters) do you want to rise: ";
    cin >> rise;
    cout << "How long (in seconds) do you want to cruise: ";
    cin >> duration;

    b.Ascend(rise);       // ascend to specified height
    b.Cruise(duration);   // cruise for specified time-steps
    b.Descend(0);         // come to earth
    WaitForReturn();      // pause to see graphics window
    return 0;
}
```

gfly.cpp

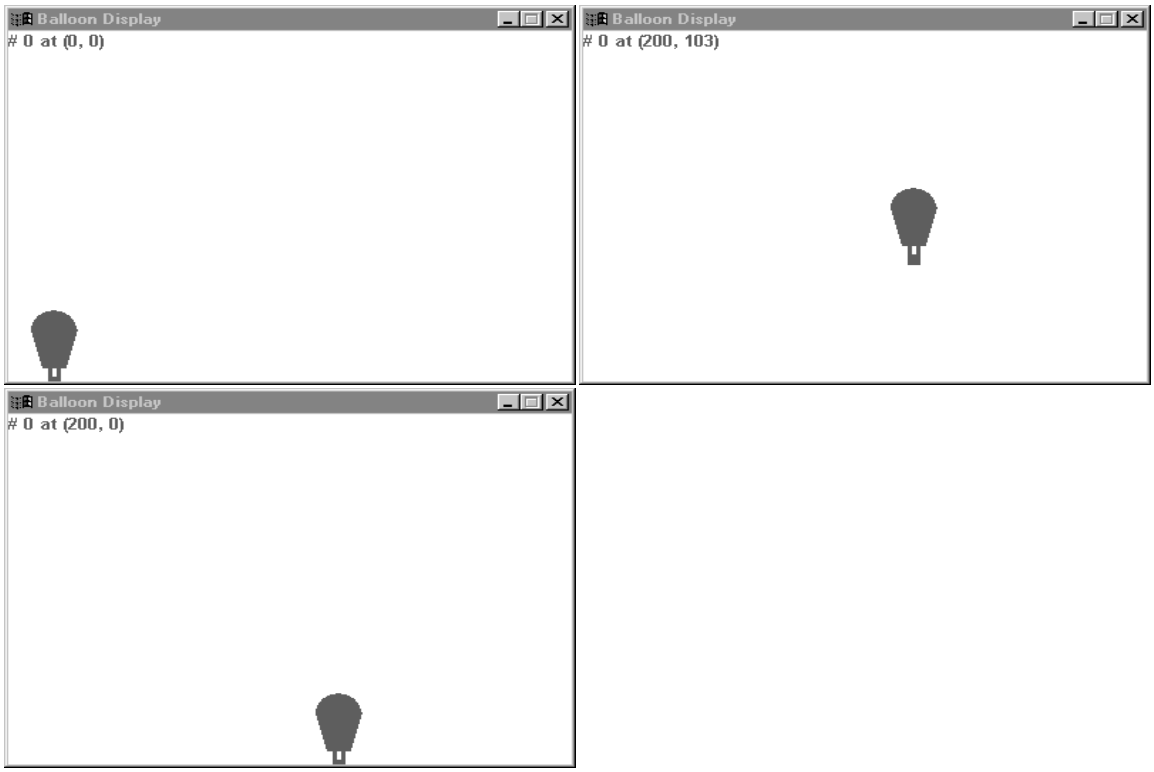


Figure 3.6 Screendumps from a run of *gfly.cpp*; rise to 100 m., cruise for 200 secs.

You won't know all the details of how the simulated balloon works, but you'll still be able to write a program that guides the balloon. This is also part of object-oriented programming: using classes without knowing exactly how the classes are implemented, that is, without knowing about the code used "behind the scenes." Just as many people drive cars without understanding exactly what a spark plug does or what a carburetor is, programmers can use classes without knowing the details of how the classes are written.

A fundamental property of a class is that its behavior is defined by the functions by which objects of the class are manipulated. Knowing about these functions should be enough to write programs using the objects; intimate knowledge of how the class is implemented is not necessary. This should make sense since you've worked with `double` variables without knowledge of how `double` numbers are stored in a computer.

In *gfly.cpp*, an object (variable) `b` of type, or class, `Balloon` is defined and used to simulate a hot-air balloon rising, cruising for a specified duration, and then descending to earth. Running this program causes both a **graphics window** and a **console window** to appear on your screen. The console window is the window we've been using in all our programs so far. It is the window in which output is displayed and in which you

enter input when running a program. The graphics window shows the balloons actually moving across part of the computer screen. The run below shows part of the text output that appears in the console window. Snapshots of the graphics window at the beginning, middle (before the balloon descends), and end of the run are shown in Figure 3.6.

Clearly there is something going on behind the scenes since the statements in Program 3.6 do not appear to be able to generate the output shown. In subsequent chapters we'll study how the `Balloon` class works; at this point we'll concentrate on understanding the three function calls in Program 3.6.

OUTPUT

Part of a run, the balloon rises and travels for seven seconds

```
prompt> gfly
Welcome to the windbag emporium.
You'll rise up, cruise a while, then descend.
How high (in meters) do you want to rise: 100
How long (in seconds) do you want to cruise: 200

balloon #0 at (0, 0)   **** rising to 50 meters
balloon #0 at (0, 0)   burn
balloon #0 at (0, 10)  burn
balloon #0 at (0, 20)  burn
balloon #0 at (0, 30)  burn
balloon #0 at (0, 40)  burn

balloon #0 at (0, 50)  ***** Cruise at 50 m. for 100 secs.
balloon #0 at (0, 50)  wind-shear -1
balloon #0 at (1, 49)
balloon #0 at (2, 49)
balloon #0 at (3, 49)  wind-shear -4
balloon #0 at (4, 45)  wind-shear -1  too low!  burn
balloon #0 at (5, 54)  wind-shear -3
balloon #0 at (6, 51)
balloon #0 at (7, 51)
```

3.4.1 Member Functions

We have studied several programs with user-defined functions. In *macinput.cpp*, Program 3.1, the function `Verse` has two `string` parameters. In *pizza.cpp*, Program 3.5, the function `SlicePrice` has one `int` parameter and one `double` parameter. In Program 3.6, *gfly.cpp*, three function calls are made: *Ascend*, *Cruise*, and *Descend*. Together, these functions define the behavior of a `Balloon` object. You can't affect a

balloon or change how it behaves except by using these three functions to access the balloon. These functions are applied to the object *b* as indicated by the "dot" syntax as in

```
b.Ascend(rise);
```

which is read as "*b dot ascend rise*." These functions are referred to as **member functions** in C++. In other object-oriented languages, functions that are used to manipulate objects of a given class are often called **methods.** In this example, the object `b` invokes its member function `Ascend` with `rise` as the argument.

Note that definitions of these functions do *not* appear in the text of Program 3.6 before they are called. The prototypes for these functions are made accessible by the statement

```
#include "gballoon.h"
```

which causes the information in the **header file** *gballoon.h* to be included in Program 3.6. The header file is an interface to the class `Balloon`. Sometimes an **interface diagram** is used to summarize how a class is accessed. The diagram shown in Figure 3.7 is modeled after diagrams used by Grady Booch [Boo91]. Detailed information on the `Balloon` class and all other classes that are provided for use with this book is found in Howto G.

Each member function[5] is shown in an oval, and the name of the class is shown in a rectangle. Details about the member function prototypes as well as partial specification for what the functions do are found in the header file. The interface diagram serves as a reminder of what the names of the member functions are.

3.4.2 Reading Programs

One skill you should begin to learn is how to read a program and the supporting documentation for the program. Rich Pattis, the author of *Karel the Robot*, argues that you should read a program carefully, not like a book but like a contract you desperately want to break. The idea is that you must pay close attention to the "fine print" and not just read for plot or characterization. Sometimes such minute perusal is essential, but it is often possible to gain a general understanding without such scrutiny.

The header file *gballoon.h* is partially shown in Program 3.7. The private section of the code is not shown here, but is available if you look at the header file with the code that comes with this book.

Program 3.7 gballoonx.h

```
#ifndef _GBALLOON_H
#define _GBALLOON_H
```

[5]We will not use the functions `GetAltitude` and `GetLocation` now but will return to them in a later chapter.

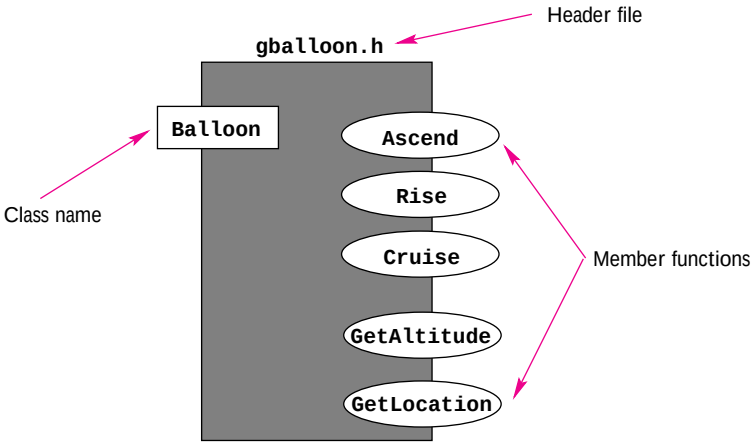


Figure 3.7 Interface diagram for `Balloon` class

```
// class for balloon manipulation using simulated auto pilot
// (based on an idea of Dave Reed) graphics version 3/22/99
//
// Ascend: rise to specified height in a sequence of burns
//         Each burn raises the altitude by 10 meters
//
// Cruise: cruise for specified time-steps/seconds
//         Random wind-shear can cause balloon to rise and fall,
//         but vents and burns keep ballon within 5 m. of start altitude
//
// Descend:   descend to specified height in sequence of vents
//            Each vent drops the balloon 10 m. or to ground if < 10 m.
//
// int GetAltitude: returns altitude (in meters)  (y-coord)
// int GetLocation: returns how many time steps/secs elapsed (x-coord)

#include "canvas.h"
#include "utils.h"

class Balloon
{
  public:
    Balloon();                    // use default color (gold)
    Balloon(color c);             // balloon of specified color

    void Ascend  (int height);    // ascend so altitude >= parameter
    void Descend (int height);    // descend so altitude <= parameter
    void Cruise  (int steps);     // cruise for parameter time-steps
    int GetAltitude() const;      // returns height above ground (y-coord)
    int GetLocation() const;      // returns # time-steps (x-coord)
  private:
```

```
    void Burn();
    void Vent();
    int myAltitude;
    int mySteps;        // ... see gballoon.h for details
};
#endif
```

gballoonx.h

There are three important details of this header file.

1. Comments provide users and readers of the header file with an explanation of what the member functions do.
2. Member functions are declared in the **public** section of a class definition and may be called by a user of the class as is shown in Program 3.6. We'll discuss the special member functions `Balloon` later. The other functions, also shown in the interface diagram in Figure 3.7, each have prototypes showing they take one `int` parameter except for `GetAltitude` and `GetLocation`.
3. Functions and data in the **private** section are *not* accessible to a user of the class. As a programmer using the class, you may glance at the private section, but the compiler will prevent your program from accessing what's in the private section. Definitions in the private section are part of the class's implementation, *not* part of the class's interface. As a user, or **client,** of the class, your only concern should be with the interface, or public section.

3.4.3 Private and Public

The declaration of the class `Balloon` in Program 3.7 shows that the parts of the class are divided into two sections: the private and the public sections. The public section is how an object of a class appears to the world, how the object behaves. Objects are manipulated in programs like Program 3.6, *gfly.cpp*, by calling the object's public member functions. Private member functions[6] exist only to help implement the public functions. Imagine a company that publishes a list of its company phone numbers. For security reasons, some numbers are accessible only by those calling from within the company building. An outsider can get a copy of the company phonebook and see the inaccessible phone numbers, but the company switchboard will allow only calls from within the building to go through to the inaccessible numbers.

In general, the designation of what should be private and what should be public is a difficult task. At this point, the key concept is that you access a class is via its public functions. Some consider it a drawback of C++ that information in the private section can be seen and read, but many languages suffer from the same problem. To make things simple, you should think of the private section as invisible until you begin to design your own classes.

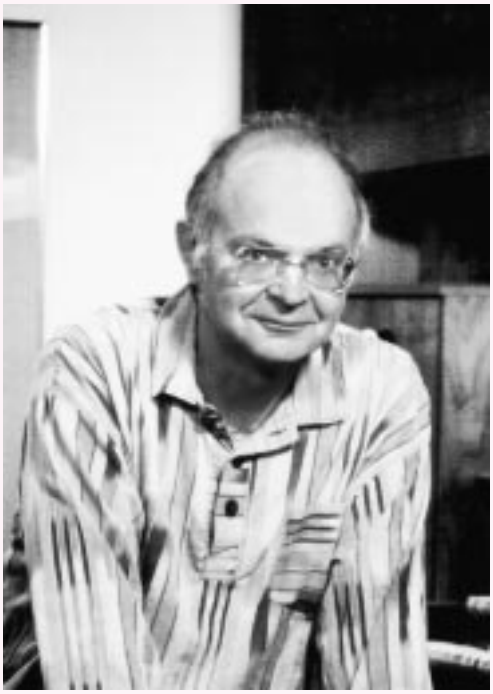
There are often variables in the private section. These variables, such as `myAltitude`, define an object's **state**—the information that determines the object's characteristics. In

[6]Prototypes for private member functions like `Burn` and `AdjustAltitude` are visible in the full listing of *gballoon.h*, but are not shown in the partial listing of *gballoonx.h* in Program 3.7.

the case of a `Balloon` object, the altitude of the balloon, represented by the `int` variable `myAltitude`, is part of this state. Knowledge of the private section isn't necessary to understand how to use `Balloon` objects.

Donald Knuth *(b. 1938)*

Donald Knuth is perhaps the best-known computer scientist and is certainly the foremost scholar of the field. His interests are wide-ranging, from organ playing to word games to typography. His first publication was for *MAD* magazine, and his most famous is the three-volume set *The Art of Computer Programming.*



In 1974 Knuth won the Turing award for "major contributions to the analysis of algorithms and the design of programming languages." In his Turing award address he says: *The chief goal of my work as educator and author is to help people learn how to write beautiful programs. My feeling is that when we prepare a program, it can be like composing poetry or music; as Andrei Ershov has said, programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and to establish a system of consistent rules.*

In discussing what makes a program "good," Knuth says:

In the first place, it's especially good to have a program that works correctly. Secondly it is often good to have a program that won't be hard to change, when the time for adaptation arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language.

Of computer programming Knuth says:

We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

For more information see [Sla87, AA85, ACM87].

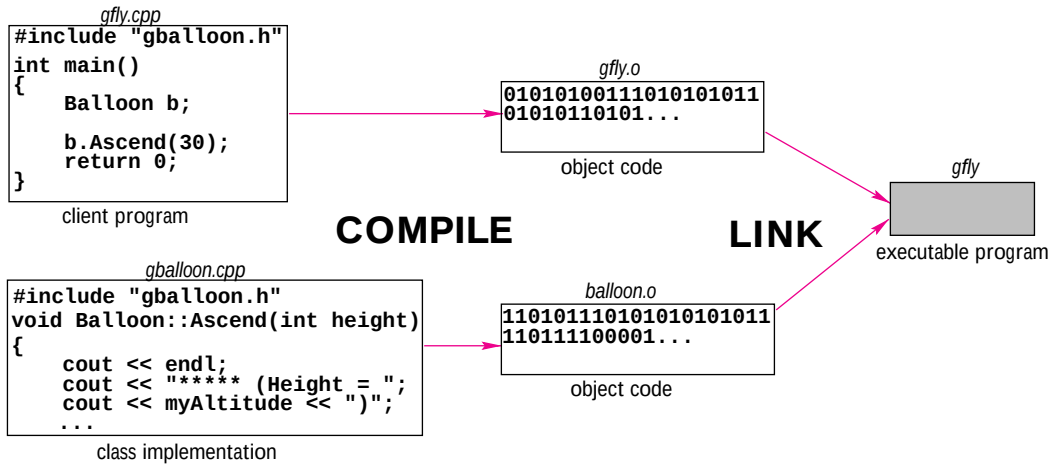


Figure 3.8 Compiling and linking.

The public section describes the interface to an object, that is, what a client or user needs to know to manipulate the object. In a car, the brake pedal is the interface to the braking system. Pressing the pedal causes the car to stop, regardless of whether antilock brakes, disc brakes, or drum brakes are used. In general, the public interface provides "buttons" and "levers" that a user can push and pull to manipulate the object as well as dials that can be used to read information about the object state.

All header files we'll use in this book will have statements similar to the `#ifndef _BALLOON_H` statement and others that begin with the # sign, as shown in *balloon.h*. For the moment we'll ignore the purpose of these statements; they're necessary but are not important to the discussion at this point. The `ifndef` statement makes it impossible to include the same header file more than once in the same program. We'll see why this is important when programs get more complex.

3.5 Compiling and Linking

In Chapter 1 we discussed the differences between **source code,** written in a high-level language like C++, and machine code, written in a low-level language specific to one kind of computer. The compiler translates the source code into machine code. Another step is almost always necessary in making an executable program. Code from libraries needs to be **linked** together with the machine code to form an executable program. For example, when the header file `<iostream>` is used, the code that implements input and output streams must be linked. When the header file `"balloon.h"` is used, the code that implements the balloon class must be linked. This process of compiling and linking is illustrated in Figure 3.8.

The compiler translates source code into machine, or **object,** code. The word *object*

here has nothing to do with object-oriented programming; think of it as a synonym for *machine code.* In some environments object code has a `.o` extension; in other environments it has a `.obj` extension. The different object files are linked together to form an executable program. Sometimes you may not be aware that linking is taking place. But when you develop more complex programs, you'll encounter errors in the linking stage. For example, if you try to compile *gfly.cpp,* Program 3.6, you *must* link-in the code that implements the balloon class. The implementation of the class is declared in *gballoon.h* and is found in the file *balloon.cpp.* The corresponding object code, as translated by the compiler, is in *balloon.o.* It's often convenient to group several object files together in a code **library.** The library can be automatically linked into your programs so that you don't need to take steps to do this yourself.

Pause to Reflect

3.12 Some pizza parlors cut larger pies into more pieces than small pies: a small pie might have 8 pieces, a medium pie 10 pieces, and a large pie 12 pieces. Modify the function `SlicePrice` so that the number of slices is a parameter. The function should have three parameters instead of two. How would the function `main` in Program 3.5 change to accommodate the new `SlicePrice`?

3.13 In *pizza.cpp,* what changes are necessary to allow the user to enter the diameter of a pizza instead of the radius?

3.14 Based on the descriptions of the member functions given in the header file `balloonx.h` (Program 3.7), why is different output generated when Program 3.6 is run with the same input values? (Run the program and see if the results are similar to those shown above.)

3.15 What would the function `main` look like of a program that defines a `Balloon` object, causes the balloon to ascend to 40 meters, cruises for 10 time-steps, ascends to 80 meters, cruises for 20 time-steps, then descends to earth?

3.16 What do you think happens if the following two statements are the only statements in a modified version of Program 3.6?

```
b.Ascend(50);
b.Ascend(30);
```

What would happen if these statements are reversed (first ascend to 30 meters, then to 50)?

3.6 Chapter Review

In this chapter we studied the input/process/output model of computation and how input is performed in C++ programs. We also studied numeric types and operations and a user-defined class, `Balloon`. The importance of reading programs and documentation in order to be able to modify and write programs was stressed.

- Input is accomplished in C++ using the extraction operator, `>>`, and the standard input stream, `cin`. These are accessible by including `<iostream>`.
- Variables are memory locations with a name, a value, and a type. Variables must be defined before being used in C++. Variables can be defined anywhere in a program in C++, but we'll define most variables at the beginning of the function in which they're used.
- Numeric data represent different kinds of numbers in C++. We'll use two types for numeric data: `int` for integers and `double` for floating-point numbers (real numbers, in mathematics). If you're using a microcomputer, you should use the type `long` (`long int`) instead of `int` for quantities over 5,000.
- Operators are used to form arithmetic expressions. The standard math operators in C++ are `+ - * / %`. In order to write correct arithmetic expressions you must understand operator precedence rules and the rules of expression evaluation.
- Conversion takes place when an `int` value is converted to a corresponding `double` value when arithmetic is done using both types together.
- The type `char` represents characters, characters are used to construct strings. In C++ characters are indicated by single quotes, e.g., `'y'` and `'Y'`.
- Classes are types, but are defined by a programmer rather than being built into the language like `int` and `double`. The interface to a class is accessible by including the right header file.
- Member functions manipulate or operate on objects. Only member functions defined in the public section of a class definition can be used in a client program.
- A class is divided into two sections, the private section and the public section. Programs that use the class access the class by the public member functions.
- Executable programs are created by compiling source code into object code and linking different object files together. Sometimes object files are stored together in a code library.

3.7 Exercises

3.1 Write a program that prompts the user for a first name and a last name and that prints a greeting for that person. For example:

OUTPUT

```
enter first name Owen
enter last name Astrachan
Hello Owen, you have an interesting last name: Astrachan.
```

3.2 Write a program that prompts the user for a quantity expressed in British thermal units (BTU) and that converts it to Joules. The relationship between these two units of

measure is given by: 1 BTU = 1054.8 Joules.

3.3 Write a program that prompts the user for a quantity expressed in knots and that converts it to miles per hour. The relationship needed is that 1 knot = 101.269 ft/min (and that 5,280 ft = 1 mile).

3.4 Write a program using the operators `/` and `%` that prompts the user for a number of seconds and then determines how many hours, minutes, and seconds this represents. For example, 20,000 seconds represents 5 hours, 33 minutes, and 20 seconds.

3.5 Write a program that prints three verses of the classic song "One hundred bottles of _____ on the wall" as shown here.

OUTPUT

```
56 bottles of cola on the wall
56 bottles of cola
If one of those bottles should happen to fall
55 bottles of cola on the wall
```

The function's **prototype** is

```
void BottleVerse(string beverage, int howMany)
```

The first parameter, `beverage`, is a string representing the kind of beverage for which a song will be printed. The second parameter, `howMany`, is not a string but is a C++ integer. The advantage of using an `int` rather than a `string` is that arithmetic operations can be performed on `int`s. For example, given the function `BottleVerse` shown below, the function call `BottleVerse("cola",56)` would generate the abbreviated verse shown here.

```
void BottleVerse(string beverage, int howMany)
{
  cout << howMany << " bottles of "
       << beverage << ", ";
  cout << "one fell, " << howMany - 1
       << " exist" << endl;
}
```

OUTPUT

```
56 bottles of cola, one fell, 55 exist
```

Note how the `string` parameter is used to indicate the specific kind of beverage for which a verse is to be printed. The `int` parameter is used to specify how many bottles

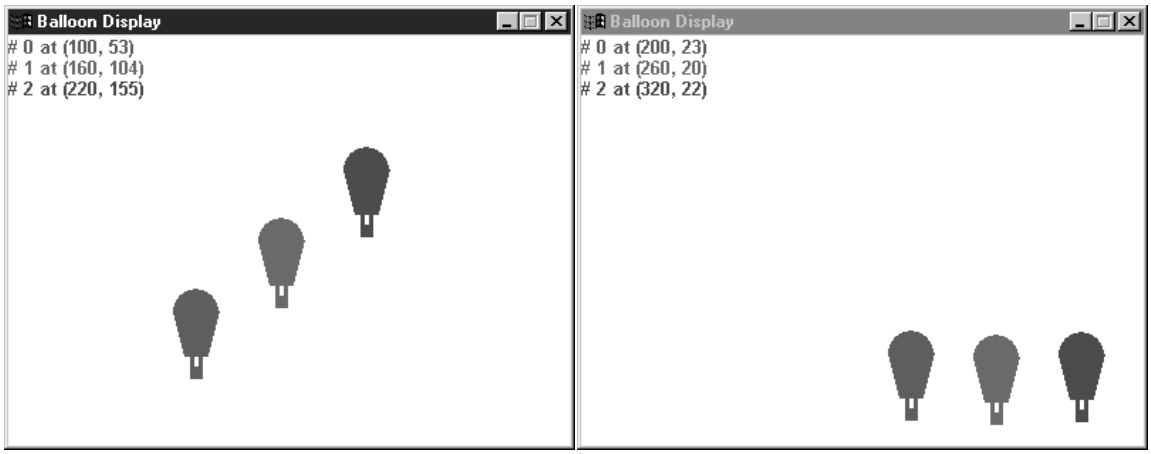


Figure 3.9 Screendumps from a run of *gfly2.cpp*.

are "in use." Note that because `int` parameters support arithmetic operations, the expression `howMany - 1` is always 1 less than the just-printed number of bottles. In the program you write, three verses should be printed. The number of bottles in the first verse can be any integer. Each subsequent verse should use 1 bottle less than the previous verse. The user should be prompted for the kind of beverage used in the song.

3.6 Write a program that calculates pizza statistics, but takes both the number of slices into account and the thickness of the pizza. The user should be prompted for both quantities.

3.7 Write a program that can be used as a simplistic trip planner. Prompt the user for the number of car passengers, the length of the trip in miles, the capacity of the fuel tank in gallons, the price of gas, and the miles per gallon that the car gets. The program should calculate the number of tanks of gas needed, the total price of the gas needed, and the price per passenger if the cost is split evenly.

3.8 Write a program that uses a variable of type `Balloon` that performs the following sequence of actions:

1. Prompt the user for an initial altitude and a number of time steps.
2. Cause the balloon to ascend to the specified altitude, then cruise for the specified time steps.
3. Cause the balloon to descend to half the altitude it initially ascended to, then cruise again for the specified time steps.
4. Cause the balloon to descend to earth (height = 0).

3.9 The program *gfly2.cpp* is shown on the next page as Program 3.8. Several different balloons are used in the same program. A screendump is shown in Figure 3.9. Modify the program so the user is prompted for how high the first balloon should rise. The

other two balloons should rise to heights two and three times as high, respectively. The user should be prompted for how far the balloons cruise. The balloons should cruise for one-third this distance, then the function `WaitForReturn` should be called so that the user can see the balloons paused in flight. Repeat this last step twice so that the balloons all fly for the specified time, but in three stages.

Program 3.8 gfly2.cpp

```
#include <iostream>
using namespace std;
#include "gballoon.h"

// illustrates graphical balloon class
// auto-pilot guided balloon ascends, cruises, descends

int main()
{

    Balloon b1(MAROON);
    Balloon b2(RED);
    Balloon b3(BLUE);

    WaitForReturn();

    b1.Ascend(50);  b1.Cruise(100);
    b2.Ascend(100); b2.Cruise(160);
    b3.Ascend(150); b3.Cruise(220);

    WaitForReturn();

    b1.Descend(20); b1.Cruise(100);
    b2.Descend(20); b2.Cruise(100);
    b3.Descend(20); b3.Cruise(100);

    WaitForReturn();
    return 0;
}
```

gfly2.cpp