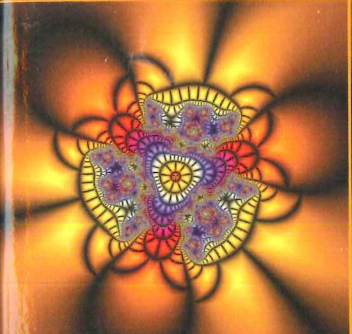


FOURTH EDITION

DATA STRUCTURES and Other Objects Using C++

Michael Main • Walter Savitch

This chapter illustrates the phases of software development. These phases occur in all software, including the small programs that you'll see in this first chapter. In subsequent chapters, you'll go beyond these small programs, applying the phases of software development to organized collections of data. These organized collections of data are called **data structures**, and the main topics of this book revolve around proven techniques for representing and manipulating such data structures.

Years from now you may be a software engineer writing large systems in a specialized area, perhaps computer graphics or artificial intelligence. Such futuristic applications will be exciting and stimulating, and within your work you will still see the phases of software development and fundamental data structures that you learn and practice now.

Here is a list of the phases of software development::

The Phases of Software Development

- Specification of the task
- Design of a solution
- Implementation (coding) of the solution
- Analysis of the solution
- Testing and debugging
- Maintenance and evolution of the system
- Obsolescence

the phases blur into each other

Do not memorize this list: Throughout the book, your practice of these phases will achieve far better familiarity than mere memorization. Also, memorizing an "official list" is misleading because it suggests that there is a single sequence of discrete steps that always occur one after another. In practice, the phases blur into each other; for instance, the analysis of a solution's efficiency may occur hand in hand with the design, before any coding. Or low-level design decisions may be postponed until the implementation phase. Also, the phases might not occur one after another. Typically there is back-and-forth travel between the phases.

Most of the work in software development does not depend on any particular programming language. Specification, design, and analysis can all be carried out with few or no ties to a particular programming language. Nevertheless, when we get down to implementation details, we do need to decide on one particular programming language. The language we use in this book is C++.

What You Should Know About C++ Before Starting This Text

OOP supports information hiding and component reuse

The C++ language was designed by Bjarne Stroustrup at AT&T Bell Laboratories as an extension of the C language, with the purpose of supporting **object-oriented programming (OOP)**—a technique that encourages important strategies of information hiding and component reuse. Throughout this book, we introduce you to important OOP principles to use in your designs and implementations.

you should already know how to write, compile, and run short C++ programs

There are many different C++ compilers that you may successfully use with this text. Ideally, the compiler should support the latest features of the ANSI/ISO C++ Standard, which we have incorporated into the text. However, there are several workarounds that can be applied to older compilers that don't fully support the standard. (See Appendix K, "Downloading the GNU Compiler Software," and Appendix E, "Dealing with Older Compilers.")

Whichever programming environment you use, you should already be comfortable writing, compiling, and running short C++ programs built with a top-down design. You should know how to use the built-in types (the number types, char, and bool), and you should be able to use arrays.

C++ Standard Library

Throughout the text, we will introduce the important roles of the C++ Standard Library, though you do not need any previous knowledge of the library. Studying the data structures of the Standard Library can help you understand trade-offs between different approaches, and can guide the design and implementation of your own data structures. When you are designing your own data structures, an approach that is compliant with the Standard Library has twofold benefits: Other programmers will understand your work more easily, and your own work will readily benefit from other pieces of the Standard Library, such as the standard searching and sorting algorithms.

The rest of this chapter will prepare you to tackle the topic of data structures in C++, using an approach that is compliant with the Standard Library. Section 1.1 focuses on a technique for specifying program behavior, and you'll also see some hints about design and implementation. Section 1.2 illustrates a particular kind of analysis: the running time analysis of a program. Section 1.3 provides some techniques for testing and debugging programs.

1.1 SPECIFICATION, DESIGN, IMPLEMENTATION

> *One begins with a list of difficult design decisions which are likely to change. Each module is then designed to hide such a decision from the others.*
>
> D. L. PARNAS
> "On the Criteria to Be Used
> in Decomposing Systems into Modules"

```
CONVERSIONS FROM -50.0 to 50.0
   Celsius        Fahrenheit
    -50.0C
    -40.0C
    -30.0C
    -20.0C
    -10.0C
      0.0C
     10.0C
     20.0C
     30.0C
     40.0C
     50.0C
```

The actual Fahrenheit temperatures will be computed and displayed on this side of the table.

As an example of software development in action, let's examine the specification, design, and implementation for a particular problem. The **specification** is a precise description of the problem; the **design** phase consists of formulating the steps to solve the problem; the **implementation** is the actual C++ code that carries out the design.

The problem we have in mind is to display a table for converting Celsius temperatures to Fahrenheit, similar to the table shown in the margin. For a small problem, a sample of the desired output is a sufficient specification. Such a sample is a good specification because it is *precise*, leaving no doubt about what the program must accomplish. The next step is to design a solution.

An **algorithm** is a set of instructions for solving a problem. An algorithm for the temperature problem will print the conversion table. During the design of the algorithm, the details of a particular programming language can be distracting, and can obscure the simplicity of a solution. Therefore, during the design we generally write in English. We use a rather corrupted kind of English that mixes in C++ when it's convenient. This mixture of English and a programming language is called **pseudocode**. When the C++ code for a step is obvious, then the pseudocode may use C++. When a step is clearer in English, then we will use English. Keep in mind that the reason for pseudocode is to improve *clarity*.

We'll use pseudocode to design a solution for the temperature problem, and we'll also use the important design technique of decomposing the problem.

Key Design Concept

Break down a task into a few subtasks; then decompose each subtask into smaller subtasks.

Design Concept: Decomposing the Problem

A good technique for designing an algorithm is to break down the problem at hand into a few subtasks, then decompose each subtask into smaller subtasks, then replace the smaller subtasks with even smaller subtasks, and so forth. Eventually the subtasks become so small that they are trivial to implement in C++ or whatever language you are using. When the algorithm is translated into C++, each subtask is implemented as a separate C++ function. In other programming languages, functions are called "methods" or "procedures," but it all boils down to the same thing: The large problem is decomposed into subtasks, and subtasks are implemented as separate pieces of your program.

For example, the temperature problem has at least two good subtasks: (1) converting a temperature from Celsius degrees to Fahrenheit, and (2) printing a line of the conversion table in the specified format. Using these subproblems, the first draft of our pseudocode might look like this:

1. Do preliminary work to open and set up the output device properly.
2. Display the labels at the top of the table.
3. For each line in the table (using variables `celsius` and `fahrenheit`):
   a. Set `celsius` equal to the next Celsius temperature of the table.
   b. `fahrenheit` = the `celsius` temperature converted to Fahrenheit.
   c. Print the Celsius and Fahrenheit values with labels on an output line.

what makes a good decomposition?

We have identified the major subtasks. But aren't there other ways to decompose the problem into subtasks? What are the aspects of a good decomposition? One primary guideline is that the subtasks should help you produce short pseudocode—no more than a page of succinct description to solve the entire problem, and ideally much less than a page. In your designs, you can also keep in mind two considerations for selecting good subtasks: the potential for code reuse, and the possibility of future changes to the program. Let's see how our subtasks embody these considerations.

code reuse

Step 1 opens an output device, making it ready for output in a particular form. This is a common operation that many programs must carry out. If we write a function for Step 1 with sufficient flexibility, we can probably reuse the function in other programs. This is an example of **code reuse**, in which a function is written with sufficient generality that it can be reused elsewhere. In fact, programmers often produce collections of related C++ functions that are made available in packages to be reused over and over with many different application programs. Later we will use the C++ Standard Library as this sort of package, and we will also write our own packages of this kind. For now, just keep in mind that the function for Step 1 should be written with some reuse in mind.

easily modified code

Decomposing problems also produces a good final program in the sense that the program is easy to understand, and subsequent maintenance and modifications are relatively easy. Our temperature program might be modified to convert to Kelvin degrees instead of Fahrenheit, or even to do a completely different conversion such as feet to meters. If the conversion task is performed by a separate function, much of the modification will be confined to this one function. Easily modified code is vital since real-world studies show that a large proportion of programmers' time is spent maintaining and modifying existing programs.

In order for a problem decomposition to produce easily modified code, the functions that you write need to be genuinely separated from one another. An analogy can help explain the notion of "genuinely separated." Suppose you are moving a bag of gold coins to a safe hiding place. If the bag is too heavy to carry, you might divide the coins into three smaller bags and carry the bags one by one. Unless you are a character in a comedy, you would not try to carry all three bags at once. That would defeat the purpose of dividing the coins into three groups. This strategy works only if you carry the bags one at a time. Something similar happens in problem decomposition. If you divide your programming task into three subtasks and solve these subtasks by writing three functions, then you have

traded one hard problem for three easier problems. Your total job has become easier—provided that you design the functions separately. When you are working on one function, you should not worry about how the other functions perform their jobs. But the functions do interact. So when you are designing one function, you need to know something about what the other functions do. The trick is to know *only as much as you need, but no more*. This is called **information hiding**. One technique for incorporating information hiding involves specifying your functions' behavior using *preconditions* and *postconditions*.

Preconditions and Postconditions

When you write a complete function definition, you specify how the function performs its computation. However, when you are using a function, you only need to think about *what* the function does. You need not think about *how* the function does its work. For example, suppose you are writing the temperature conversion program and you are told that a function is available for you to use, as described here:

```
// Convert a Celsius temperature c to Fahrenheit degrees
double celsius_to_fahrenheit(double c);
```

Your program might have a *double* variable called `celsius` that contains a Celsius temperature. Knowing this description, you can confidently write the following statement to convert the temperature to Fahrenheit degrees, storing the result in a *double* variable called `fahrenheit`:

```
fahrenheit = celsius_to_fahrenheit(celsius);
```

When you use the `celsius_to_fahrenheit` function, you do not need to know the details of how the function carries out its work. You need to know *what* the function does, but you do not need to know *how* the task is accomplished.

procedural abstraction

When we pretend that we do not know how a function is implemented, we are using a form of information hiding called **procedural abstraction**. This technique simplifies your reasoning by abstracting away irrelevant details—that is, by hiding the irrelevant details. When programming in C++, it might make more sense to call it "functional abstraction," since you are abstracting away irrelevant details about how a function works. However, the term *procedure* is a more general term than *function*. Computer scientists use the term procedure for any sequence of instructions, and so they use the term *procedural abstraction*. Procedural abstraction can be a powerful tool. It simplifies your reasoning by allowing you to consider functions one at a time rather than all together.

To make procedural abstraction work for us, we need some techniques for documenting what a function does without indicating how the function works. We could just write a short comment as we did for `celsius_to_fahrenheit`. However, the short comment is a bit incomplete—for instance, the comment doesn't indicate what happens if the parameter `c` is smaller than the lowest Celsius temperature (–273.15°C, which is **absolute zero** for Celsius temperatures).

precondition and postcondition

For better completeness and consistency, we will follow a fixed format that always has two pieces of information called the *precondition* and the *postcondition* of the function, described here:

> **Preconditions and Postconditions**
>
> A **precondition** is a statement giving the condition that is required to be true when a function is called. The function is not guaranteed to perform as it should unless the precondition is true.
>
> A **postcondition** is a statement describing what will be true when a function call is completed. If the function is correct and the precondition was true when the function was called, then the function will complete, and the postcondition will be true when the function call is completed.

For example, a precondition/postcondition for the `celsius_to_fahrenheit` function is shown here:

```
double celsius_to_fahrenheit(double c);
// Precondition: c is a Celsius temperature no less than
// absolute zero (-273.15).
// Postcondition: The return value is the temperature c
// converted to Fahrenheit degrees.
```

comments in C++

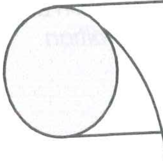
This format of comments might be new to you: The characters // indicate the start of a comment that extends to the end of the current line. The other form of C++ comments, starting with /* and continuing until */, is also permitted.

specify the precondition and postcondition when you write the function's prototype

Preconditions and postconditions are more than a way to summarize a function's actions. Stating these conditions should be the first step in designing any function. Before you start to think about algorithms and C++ code for a function, you should write out the function's **prototype**, which consists of the function's return type, name, and parameter list, all followed by a semicolon. As you are writing the prototype, you should also write the precondition and postcondition as comments. If you later discover that your specification cannot be realized in a reasonable way, you may need to back up and rethink what the function should do.

programming teams

Preconditions and postconditions are even more important when a group of programmers work together. In team situations, one programmer often does not know how a function written by another programmer works and, in fact, sharing knowledge about how a function works can be counterproductive. Instead, the precondition and postcondition provide all the interaction that's needed. In effect, the precondition/postcondition pair forms a contract between the programmer who uses a function and the programmer who writes that function. To aid the explanation of this "contract," we'll give these two programmers names.



the precondition/ postcondition contract

Judy is the head of a programming team that is writing a large piece of software. Jervis is one of her programmers, who writes various functions for Judy to use in large programs. If Judy and Jervis were lawyers, the contract might look like the scroll shown in the margin. As a programmer, the contract tells them precisely what the function does. It states that if Judy makes sure that the precondition is met when the function is called, then Jervis ensures that the function returns with the postcondition satisfied.

Using Functions Provided by Other Programmers

The programmers that you work with may or may not use the words "precondition" and "postcondition" to describe their functions, but they will provide and expect information about what a function does. For example, consider this function that sets up the standard output device (`cout`) to print numbers:

```
void setup_cout_fractions(int fraction_digits);
// Precondition: fraction_digits is not negative.
// Postcondition: All double or float numbers printed to cout will now be
// rounded to the specified number of digits on the right of the decimal point.
```

If you are curious about the `setup_cout_fractions` implementation, you can read Appendix F, which provides some input/output ideas for C++ programming. But even without the knowledge of how Jervis writes the function, we can write a program that uses his function. For example, the temperature program, shown in Figure 1.1, follows our pseudocode, using `setup_cout_fractions` and `celcius_to_fahrenheit`. In Chapter 2, we will see how the actual functions such as `setup_cout_fractions` do not need to appear in the same file as the main program, providing an even stronger separation between the use of a function and its implementation. Next, we discuss a few other implementation issues that may be new to you.

Implementation Issues for the ANSI/ISO C++ Standard

This section concludes with some implementation issues for the temperature program from Figure 1.1. Some of these issues may be new if you haven't previously used the ANSI/ISO C++ Standard.

C++ FEATURE

THE STANDARD LIBRARY AND THE STANDARD NAMESPACE

During the late 1990s, the American National Standards Institute (ANSI) and the International Standards Organization (ISO) developed C++ compiler requirements called the ANSI/ISO C++ Standard. The standard aids programmers in writing portable code that can be compiled and run with many different compilers on different machines. Part of the standard is the C++ Standard Library. Each facility in the Standard Library provides a group of declared constants, data types, and functions supporting particular activities such as input/output or mathematical functions.

In 1999, C++ compilers began to provide the full C++ Standard Library. To use one of the library facilities, a program places an "include directive" at the top of the file that uses the facility. For example, for a program to use the usual C++ input/output facilities, the program should use the include directive:

```
#include <iostream>
```

This gives the program access to most of the C++ input/output facilities. Some additional input/output items require a second include directive:

```
#include <iomanip>
```

A discussion of the input/output facilities from <iostream> and <iomanip> is given in Appendix F.

FIGURE 1.1 The Temperature Conversion Program

A Program

See the C++ Feature, "The Standard Library and the Standard Namespace."

```
// File: temperature.cxx
// This conversion program illustrates some implementation techniques.
#include <cassert>      // Provides assert function
#include <cstdlib>      // Provides EXIT_SUCCESS
#include <iomanip>      // Provides setw function for setting output width
#include <iostream>     // Provides cout
using namespace std;    // Allows all Standard Library items to be used

double celsius_to_fahrenheit(double c)
// Precondition: c is a Celsius temperature no less than absolute zero (-273.15).
// Postcondition: The return value is the temperature c converted to Fahrenheit degrees.
{
    const double MINIMUM_CELSIUS = -273.15; // Absolute zero in Celsius degrees

    assert(c >= MINIMUM_CELSIUS);
    return (9.0 / 5.0) * c + 32;
}
```

(continued)

(FIGURE 1.1 continued)

```
void setup_cout_fractions(int fraction_digits)
// Precondition: fraction_digits is not negative.
// Postcondition: All double or float numbers printed to cout will now be rounded to the
// specified digits on the right of the decimal point.
{
    assert(fraction_digits > 0);
    cout.precision(fraction_digits);
    cout.setf(ios::fixed, ios::floatfield);
    if (fraction_digits == 0)
        cout.unsetf(ios::showpoint);
    else
        cout.setf(ios::showpoint);
}

int main( )
{
    const char   HEADING1[]  = "   Celsius"; // Heading for table's 1st column
    const char   HEADING2[]  = "Fahrenheit"; // Heading for table's 2nd column
    const char   LABEL1      =          'C'; // Label for numbers in 1st column
    const char   LABEL2      =          'F'; // Label for numbers in 2nd column
    const double TABLE_BEGIN =        -50.0; // The table's first Celsius temp.
    const double TABLE_END   =         50.0; // The table's final Celsius temp.
    const double TABLE_STEP  =         10.0; // Increment between temperatures
    const int    WIDTH       =            9; // Number chars in output numbers
    const int    DIGITS      =            1; // Number digits right of decimal pt

    double value1;  // A value from the table's first column
    double value2;  // A value from the table's second column

    // Set up the output for fractions and print the table headings.
    setup_cout_fractions(DIGITS);
    cout << "CONVERSIONS from " << TABLE_BEGIN << " to " << TABLE_END << endl;
    cout << HEADING1 << "  " << HEADING2 << endl;

    // Each iteration of the loop prints one line of the table.
    for (value1 = TABLE_BEGIN; value1 <= TABLE_END; value1 += TABLE_STEP)
    {
        value2 = celsius_to_fahrenheit(value1);
        cout << setw(WIDTH) << value1 << LABEL1 << "  ";
        cout << setw(WIDTH) << value2 << LABEL2 << endl;
    }

    return EXIT_SUCCESS;
}
```

See the Programming Tip, "Use Assert to Check Preconditions," on page 12.

See the Programming Tip, "Use Declared Constants," on page 11.

See the Programming Tip, "Use EXIT_SUCCESS in a Main Program," on page 14.

www.cs.colorado.edu/~main/chapter1/temperature.cxx WWW

Older Names for the Header Files

The files iostream and iomanip are examples of C++ header files. Older C++ compilers used slightly different names for header files. For example, older compilers used iostream.h instead of simply iostream. In most cases, the new C++ header file names are the same as the old file names with the ".h" removed, and newer compilers will still allow the older names.

In addition to the C++ header files, the C++ Standard includes a collection of header files from the original C language. Two examples are the C Standard Library <stdlib.h> and the assert facility <assert.h>. These original names can still be used in a C++ program, or you can use the new C++ header file names, which are constructed by removing the ".h" and putting the letter "c" at the front of the name (such as <cstdlib> and <cassert>).

A discussion of <cstdlib> and <cassert> is given as part of Appendix G, "Selected Library Functions."

The Standard Namespace

There is one difference between using old header file names (such as <iostream.h> or <stdlib.h>) and the new names (such as <iostream> or <cstdlib>). All of the items in the new header files are part of a feature called the **standard namespace**, also called **std**. For now, when you use one of the new header files, your program should also have this statement after the include directives:

```
using namespace std;
```

This statement is a global namespace directive, which allows your program to use all items from the standard namespace. Chapter 2 discusses alternatives to the global namespace directive, and also shows how to create your own namespaces to avoid conflicts between the names that occur in different pieces of a program.

PROGRAMMING TIP

USE DECLARED CONSTANTS

Throughout the temperature program, there are several declarations of the form:

```
const double TABLE_BEGIN = -50.0;
```

This is a declaration of a *double* number called TABLE_BEGIN, which is given an initial value of –50.0. The keyword *const*, appearing before the declaration, makes TABLE_BEGIN more than just an ordinary declaration. It is a **declared constant**, which means that its value will never be changed while the program is running. A common programming style is to use all capital letters for any declared constant. This makes it easy to identify such values within a program.

There are several advantages to defining TABLE_BEGIN as a declared constant, rather than using the number –50.0 directly in the program. Using the name TABLE_BEGIN makes it easy to understand the purpose of the constant. Moreover, once a constant has been declared, it can be used throughout the program. For

example, our program uses TABLE_BEGIN twice (once when printing the heading at the top, and once to determine a beginning value used in the for-loop).

Using declared constants also makes it easier to alter a program. For example, we may decide to alter the program so that the table starts at –100.0 instead of –50.0. This change is accomplished by finding the declared constant (TABLE_BEGIN), changing its initial value to –100.0, then recompiling the program. By changing the initial value, all occurrences of TABLE_BEGIN will have the new value.

To increase clarity and to ease alterations, some programmers use declared constants for *all* fixed values in a program. As rules go, this is a reasonable one. However, there is another side to the issue. Well-known formulas may be more easily recognized in their original form (using numbers directly rather than artificially introduced names). For example, the conversion from Celsius to Fahrenheit is recognizable as $F = \frac{9}{5}C + 32$. Thus, Figure 1.1 uses the return statement shown here:

```
return (9.0/5.0) * c + 32;
```

This return statement is clearer and less error-prone than a version that uses declared constants for the values $\frac{9}{5}$ and 32.

CLARIFYING THE CONST KEYWORD
Part 1: Declared Constants

1. DECLARED CONSTANTS
2. CONSTANT MEMBER FUNCTIONS: PAGE 38
3. CONST REFERENCE PARAMETERS: PAGE 72
4. STATIC MEMBER CONSTANTS: PAGE 104
5. CONST ITERATORS: PAGE 144
6. CONST PARAMETERS THAT ARE POINTERS OR ARRAYS: PAGE 171
7. THE CONST KEYWORD WITH A POINTER TO A NODE, AND THE NEED FOR TWO VERSIONS OF SOME MEMBER FUNCTIONS: PAGE 227

For programmers who implement data structures, the C++ keyword *const* has several uses that must be coordinated with each other. Because of potential confusion between the different uses, we'll clarify each use when we first use it in an example.

You can use the keyword *const* in front of any variable declaration. This indicates that the program is not allowed to change the variable's value.

Syntax:

```
const   <Data type>   <Variable name>  =  <Value> ;
```

Examples:

```
const double TABLE_BEGIN = 50.0;
const char LABEL1 = 'C';
```

PROGRAMMING TIP
USE ASSERT TO CHECK A PRECONDITION

Consider the function celsius_to_fahrenheit from the temperature program. The function has a precondition, requiring its parameter to be no less than absolute

zero (because lower temperatures have no physical meaning). The programmer who uses the function is always responsible for ensuring that the precondition is valid. But, what if a programmer uses the function and the precondition is not valid? This is a programming error, similar to other errors, such as accidentally dividing by zero or attempting to use an array element beyond the array's bounds.

In a perfect world, such programming errors would never occur: No program would ever attempt to divide by zero, or access an array beyond its bounds, or call a function with an invalid precondition. Of course, programmers aren't perfect; both novice and experienced programmers make errors. During program development, functions should be designed to help programmers find errors as easily as possible. As part of this effort, the first action of a function should be to check that its precondition is valid. If the precondition fails, then the function prints a message and either halts the entire program, or performs some other error actions before returning.

At first glance, this approach may seem harsh. Why stop the whole program? It's just a little invalid data! But think back to programs you have written. Did you ever make an error such as accessing an array beyond its bounds, perhaps writing x[42] when the last valid location was x[41]? When this happens, a program won't always stop immediately; instead the program can continue computing with corrupted data, eventually producing a crash long after the actual error, or just silently producing a wrong answer. Difficult debugging work is sometimes needed to track down the actual location of the error. Testing and debugging is easier if a program produces an error message at the earliest detection of invalid data.

The assert facility is a good approach to detecting invalid data at an early point. To use assert, the program includes this directive:

```
#include <cassert>
```

(Older compilers may use <assert.h> instead.) The primary item in the cassert facility is called assert, which is used like a function with one argument. The argument is usually a true-false expression. The expression is evaluated. If the result is true, then no action is taken. But if the result is false, then an error message is printed, and the program is halted. These checks are called **assertions**. For example, the celsius_to_fahrenheit function uses this assertion:

```
assert(c >= MINIMUM_CELSIUS);
```

If the expression (c >= MINIMUM_CELSIUS) is true, then c is valid and the assertion takes no action. On the other hand, if the expression is false, then the precondition has been violated, so a message is printed and the program is halted.

After testing and debugging is complete, the programmer has the option of turning off all assertion checks to speed up the program. Assertions can be turned off by placing this statement immediately before the program's include directives:

```
#define NDEBUG
```

Programming Tip

Use EXIT_SUCCESS in a Main Program

When the temperature program finishes, it executes the statement:

```
return EXIT_SUCCESS;
```

This return statement ends the main program and also sends the value of the constant EXIT_SUCCESS back to your computer's operating system. The operating system is the software that is responsible for running all programs on your computer. Although you may not realize it, the operating system is able to take further actions based on the return value from a main program. For example, the return value of EXIT_SUCCESS tells the operating system that the program ended normally, and the operating system can then proceed with its next task. Other return values tell the operating system about abnormal terminations such as problems opening files or running out of memory. The EXIT_SUCCESS constant is defined in `cstdlib` (or `stdlib.h`). For most operating systems, this constant is defined as zero (which is why you may have used `return 0` in other programming).

By the way, a program can also return another constant, EXIT_FAILURE, as a simple way of indicating non-normal completion.

C++ Feature

Exception Handling

The C++ language provides built-in support for handling unusual situations, known as "exceptions," which may occur during the execution of your program. Exception handling is commonly used to handle run-time errors. It is a very good alternative to traditional techniques of error handling, which are often inadequate, error-prone, and ad hoc. Once you have a program working for the core situation where things always go as planned, you can use the C++ exception handling facilities to add code for unusual cases. Please refer to Appendix L for more information about these facilities. In order to focus on data structures, formal exception handling is not incorporated into the examples in this book.

Self-Test Exercises for Section 1.1

Each section of this book finishes with a few self-test exercises. Answers to these exercises are given at the end of each chapter.

1. What are two considerations for selecting good subtasks?
2. What are the elements of a C++ function prototype?
3. This exercise refers to a function that Jervis has written for *you* to use. The prototype and precondition/postcondition contract are shown at the top of the next page.

```
int date_check(int year, int month, int day);
// Precondition: The three parameters are a legal year, month, and
// day of the month.
// Postcondition: If the given date has been reached on or before today,
// then the function returns 0. Otherwise, the value returned is the number
// of days until the given date will occur.
```

Suppose you call the function `date_check(2009, 7, 29)`. What is the return value if today is July 22, 2009? What if today is July 30, 2009? What about February 1, 2010?

4. Write an assert statement that checks whether the `month` variable in the function `date_check` is a valid integer.
5. One of the libraries is the `<cmath>` facility, which contains a function with this prototype:

   ```
   double sqrt(double x);
   ```

   The function returns the square root of `x`. Write a reasonable precondition and postcondition for this function, and compare your answer to the solution at the end of the chapter.
6. Write the include directive that must appear before using the `sqrt` function from Self-Test Exercise 5.
7. Write the *`using`* statement that must appear before using any of the items from the C++ Standard Library.
8. Write a program to print a conversion table from feet to meters. Use the temperature conversion program as the starting point (available online at www.cs.colorado.edu/~main/chapter1/temperature.cxx).
9. Why is it a good idea to stop a program at the earliest point when invalid data is detected?
10. What is the easiest way to turn off all assertion checking in a program?

1.2 RUNNING TIME ANALYSIS

Time analysis consists of reasoning about an algorithm's speed. *Does the algorithm work fast enough for my needs? How much longer does the method take when the input gets larger? Which of several different methods is fastest?* We'll discuss these issues in this section. An example will help start the discussion.

The Stair-Counting Problem

Suppose that you and your friend Judy are standing at the top of the Eiffel Tower. As you gaze out over the French landscape, Judy turns to you and says, "I wonder how many steps there are to the bottom?" You, of course, are the ever-accommodating host, so you reply, "I'm not sure . . . but I'll find out." We'll