Chapter 2

Programs and Programming

2.1 What Is a Program?

A program is a set of instructions for a computer. Programs receive data, carry out the instructions, and produce useful results. More precisely, the computer **executes a program** by performing its instructions, one at a time, on the supplied data. The instructions a computer is capable of carrying out are very primitive—add two numbers, move a number from here to there in memory, and so forth. Large numbers of instructions are required to carry out even the simplest task, so some programs are thousands or even millions of instructions long.

Nowadays, programs are so big and complicated that they cannot be constructed without the help of the computer itself. A C **compiler** is a computer program whose purpose is to take a desired program coded in a programming language and generate the low-level instructions for the computer from that code. A C compiler is usually used through a program development system called an *integrated development environment*, or *IDE*, that includes an editor and provides windows for viewing code, results, and other relevant information.

The process of submitting a code file to a compiler for translation is called *compilation*. The programmer's code is called **source code** (or C *code* in this case). And the compiled program is called **object code** or **machine code**. One often blurs the distinction between source code and object code by using the word *program* to refer to either. Ideally, the intended meaning will always be clear from the context.

The C *programming language* is the notation used for writing C code. The purpose of this book is to help you learn how to write programs using the C programming language and to use a C compiler and an IDE to generate the instructions that will allow the computer to carry out the actions specified by your program.

Algorithms. Many programs are based on algorithms. An **algorithm** is a method for solving a well-structured problem, much as a recipe is a step-by-step process for cooking a particular dish. The method must be completely defined, unambiguous, and effective; that is, it must be capable of being carried out in

```
To find the average of a set of numbers do the following:
    Count the numbers in the set.
    Add all of them together to get their sum.
    Divide the sum by the count.
```

Figure 2.1. A simple algorithm: Find an average.

```c
/* This simple program is where we begin. */
#include <stdio.h>
int main( void )
{
    puts( "Hail and farewell, friend!" );
    return 0;
}
```

Figure 2.2. A simple program: Hail and farewell.

a mechanical way. An algorithm must terminate; it cannot go on forever.[1] As long as these criteria are met, the algorithm may be specified in English, graphically, or by any other form of communication. Figure 2.1 is an example of a very simple algorithm specified in English.

Thousands of years ago, mathematicians and scientists invented algorithms to solve important problems such as computing areas of polygons and multiplying integers. More recently, algorithms have been developed for solving engineering, mathematical, and scientific problems such as summing a series of numbers, integrating functions, and computing trajectories. A major area of computer science focuses on the creation, analysis, and improvement of algorithms. Computer scientists have invented new algorithms for computing mathematical functions and organizing, sorting, and searching collections of data. The growing power of computer software is due largely to recently invented algorithms that solve problems better and faster.

Algorithms are sometimes explained using a mixture of English and computer language called **pseudocode** (because it looks like computer code, but is not). Figures 2.3, 2.5, and 2.7 show how the simple English statement of the average algorithm from Figure 2.1 progressively is developed into a program specification (Figure 2.3), then into pseudocode (Figure 2.5) and finally into a C program (Figure 2.7).

Algorithms are also sometimes written in C or in a similar high-level language, or as graphical representations (known as **flowcharts**) of the sequence of calculations and decisions that form the algorithm. These forms (rather than pseudocode or English) are normally used when algorithms are published in textbooks or journals.

2.1.1 The Simplest Program

A very simple, yet complete, program is shown in Figure 2.2. All it does is print a greeting on the computer's screen. Even though this program is very short, it illustrates several important facts about C programs:

- The first line in this example is called a comment. It is ignored by the C compiler and is written only to provide information for human readers.
- The second line is a preprocessor command. It instructs the C compiler to bring in the declarations of the standard input and output functions so that this program can perform an output operation.
- Every C program must have a function named `main()`. Execution begins at the first statement in `main()` and continues until the `return` statement at the end. The statements are surrounded by a pair of braces (curly brackets).
- This program is a sequence of two statements. The first is a call on the `puts()` function to display a phrase on the computer's screen (`puts()` is the name of one of the standard output functions).
- The statement `return 0;` is always written at the end of a program; it returns control to the computer's operating system after the program's work is done.
- If you enter this program into your own C system, compile it, and run it, you should see the following message on your computer's screen:

```
Hail and farewell, friend!
```

2.2 The stages of program development.

Some programs are short and can be written in a few minutes. However, most programs are larger and more complex than that, and often created by many people working for many months. Whether a program is small or large, the steps in creating it are the same:

1. **Define the problem.** The programmer must know the purpose and requirements of the program he or she is supposed to create.
2. **Design a testing process.** The programmer must figure out how to test the program and how to determine whether or not it is correct.
3. **Design a solution.** The programmer must plan a sequence of steps that starts with entering the input and ends by supplying the information that is required.
4. **Program construction.** After planning comes coding. The programmer codes each step in a computer language such as C, and enters the C code into a computer file using a text editor.
5. **Program translation.** The file of C code is processed by the C compiler. If there are no errors, the result is executable code.

[1]Some programs, generally called *systems programs*, are not based on algorithms and are designed to go on forever. Their purpose is to help operate the computer.

6. **Verification.** The program is run, or executed, on data values from the test plan. The program normally displays results and the programmer must verify that the results are what was expected.

In reality, many of these steps are repeated in a cycle until all errors are eliminated from the program. In the rest of this chapter, each of these steps will be described in more detail.

2.2.1 Define the Problem

Most programs are written to solve problems, but you don't start by writing program code or even by planning a solution. Before you can begin to solve a problem, you must understand exactly the nature of the problem. You need to know what data you will start with (if any) and what answers you are supposed to produce.

Suppose a teacher asked a helper to write a small program to compute the average of a student's test scores. The helper might go through these stages in defining the problem:

First version of specification.
Goal: Compute a student's exam average.
Input: A list of exam scores.
Output: The average of the scores.

We also must decide how the output will be presented. First, all output should be labeled clearly with a word or phrase that explains its meaning. Also, it is a very good idea to "echo" every input as part of the output. This lets the user confirm that the input was typed and interpreted correctly and makes it clear which answer belongs with each input set. Spacing should be used to make the answers easy to find and easy to read.

Define constants and formulas. Some applications require the use of one or more formulas and, possibly, some numeric constants. This information is part of a complete specification, so we add the formula and update the output description:

Second version of specification.
Goal: Compute a student's exam average.
Input: A list of exam scores.
Output: Echo the inputs and display the average of the scores.
Formula: average = (sum of all scores) / (number of scores)

At this stage, the programmer should realize that more information is needed and go back to the teacher to find out how many exams the class had, and whether the program should do anything about strange scores like -1 or 1000. Assume that the teacher's answers are 3 exams, and (for now) don't worry about ridiculous scores. We now have a complete problem description, which is shown in Figure 2.3.

Goal:	Compute a student's exam average.
Input:	The user will enter exam scores interactively.
Output:	Echo the inputs and print their average.
Constant:	There are 3 exam scores.
Formula:	$Average = (score1 + score2 + score3)/3.0$
Other requirements:	The answer must be accurate to at least one decimal place.

Figure 2.3. Problem specification: Find the average exam score.

Case	**Score1**	**Score2**	**Score3**	**Answer**
Easy to compute	10	20	30	20
Special cases	100	100	100	100
	0	0	0	0
Typical cases	66	5	21	30.67
	90	82	89	87

Figure 2.4. Test plan for exam average program.

2.2.2 Design a Test Plan

Verification is an essential step in program development. When a program is first compiled and run, it generally contains errors due to mistakes in planning, unexpected circumstances, or simple carelessness. Although program testing and verification take place after a program is written and compiled, a **program verification** plan should be created much earlier in the development process, as soon as the program specifications are complete. Designing a test plan at this stage helps clarify the programmer's thoughts. It often uncovers special cases that must be handled and helps create an easily verified design for the solution.

A **test plan** consists of a list of input data sets and the correct program result for each, which often must be computed by hand. This list should test all the normal and abnormal conditions that the program could encounter. To develop a test plan, we start with the problem definition. The first few data sets (if possible) should have answers that are easy to compute in one's head. These test items often can be created by looking at the formulas and constants that are part of the problem definition and choosing values for the variables that make the calculations easy. Figure 2.3 shows a test plan for the exam average problem.

Next, all special cases should be listed. The test designer identifies these by looking at the computational requirements and limitations given in the problem definition. Extreme data values at or just beyond the specified limits should be added to the plan. The next step is to analyze what could go wrong with the program and enter data sets that would be likely to cause failure. For example, some data values might result in a division by 0.

The last data sets should contain data that might be entered during normal use of the program. Since the answers to such problems generally are harder to compute by hand, we list only one or two of them. The

1. Declare names for real numbers $n1$, $n2$, $n3$, and *average*.
2. Display output titles that identify the program.
3. Instruct the user to enter three numbers.
4. Read $n1$, $n2$, and $n3$.
5. Add $n1$, $n2$, and $n3$ and divide the sum by 3.0. Store the result in *average*.
6. Display the three numbers we just read so that the user can verify that they were entered and read correctly.
7. Display the *average*.
8. Return to the system.

Figure 2.5. Pseudocode program sketch: Find an average.

results for these items allow the programmer to refine the appearance and readability of the output.

After the coding step is complete, the programmer should return once more to the test plan. If necessary, more test cases should be added to ensure that every line of code in the program is tested at least once.

These guidelines have been stated in general terms so that they are applicable to a wide range of program testing situations. Because of this, the guidelines are somewhat vague and abstract. More examples of test plans are given on the website for this chapter and as part of the case studies in later chapters of this text. These examples will help you gain understanding of verification techniques and learn to build test plans for your own programs.

2.2.3 Design a Solution

We are now ready to begin writing the program. We must design a series of steps to go from the data given to the desired result. The steps of most simple programs follow this pattern:

- Display a title that identifies the program.
- Read in the data needed.
- Calculate the answer.
- Print the answer.

Following this guide and the specification, we write down a series of steps that will solve the problem. Many programmers like to write a *pseudocode* version first, which is a combination of English and programming terms. Figure 2.5 gives a pseudocode version of the exam average program.

2.3 The Development Environment

A program **development environment** consists of a set of system programs that enable a programmer to create, translate, and maintain programs. Many modern commercially available compilers are part of an *integrated development environment*, which includes an editor, compiler, linker, and run-time support system including a symbolic debugger. These system programs are used through a menu-driven shell. The stages of program creation are illustrated in Figure 2.6 and described next.

2.3.1 The Text Editor

The **text editor** is a tool used to enter a program into a computer file. Text editors permit a typist to enter lines of code or data into a file and make changes and rearrange parts easily. They lack the font and style commands and control over page layout included in a word processor because these "bells and whistles" are neither useful nor desirable in a program file. If you were to use a word processor and enter your program into a normal word-processor document file, the compiler would be unable to translate the program because of the embedded formatting commands.

When you enter a text editor to create a new program or data file, its screen will show a blank document. To type a program, just start at the top and type the lines of code. This is your source code. The first time you save the program file, you must give it a name that ends in *.c*. Be sure to save your file periodically and maintain a backup copy. When the program is complete, save it again. Data files are created similarly, except that they are given names ending in *.in* or *.dat*.

Text editors come in three general varieties. The very old editors, like the UNIX ed editor, are line editors. They operate on one line of a file at a time, by line number, and are slow and awkward to use. Somewhat newer are the full-screen editors, such as UNIX vi and older versions of emacs, which allow the programmer to move a cursor around the video screen and delete, insert, or change the text under the cursor. They also can delete, copy, or move large blocks of code. Some make periodic, automatic backup copies of your file, which is a tremendous help when the power unexpectedly goes off. To use one of these editors, the programmer enters a command into the operating system interpreter that contains the name of the program to be executed, the names of the files to be used, and sometimes other information regarding software options.

Modern text editors (including the new versions of emacs permit all the necessary text editing operations to be done using a mouse and a text window with scroll bars. They often display the text in color, using different colors for C's reserved words, user-defined words, and comments.

Use the most modern text editor you can find. Scroll bars and mice make a huge difference in the ease of use. A good editor encourages you to write good programs. You will be much more willing to make needed changes and reorganize program parts if it is not much work to do so.

2.3.2 The Translator

We use programming languages to communicate with computers. However, computers cannot understand our programming languages directly; they must be translated first. When we write source code in a symbolic

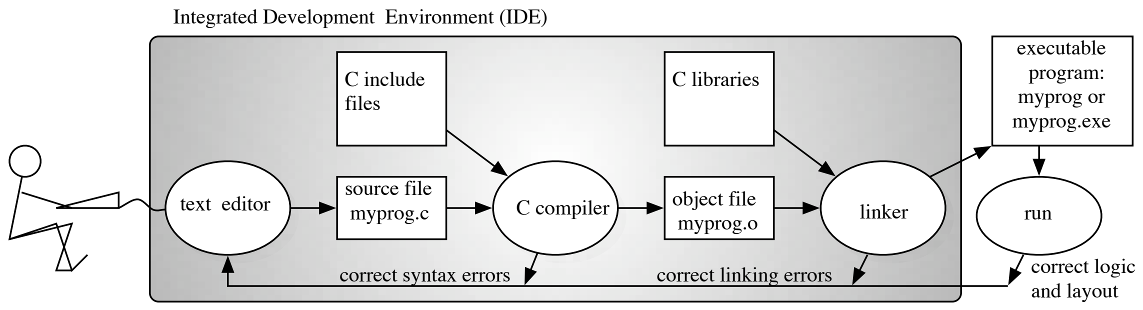


Figure 2.6. The stages of program translation.

computer language, a translator is used to analyze the code, follow the commands, find the declarations, assign memory addresses for the data objects, and determine the structure and meaning of each statement.

There are two kinds of translators: compilers and interpreters. After analyzing the source code, a compiler generates machine instructions that will carry out the meaning of the program at a later time. At **compile time**, it plans what memory locations will be needed and creates an object file that contains machine-language instructions to allocate this memory and carry out the actions of the program. Later, the object code is linked with the library, loaded into memory, and run. At **run time**, memory is allocated and information is stored in it. Then, the machine instructions are executed using these data.

An interpreter performs the translation and execution as a single step. After determining the meaning of each command, the interpreter carries it out. Some languages, such as BASIC and Java, normally are interpreted. Others, such as FORTRAN and C, normally are compiled. In general, compiled code is more efficient than interpreted code.

2.3.3 Linkers

An object file is not executable code. Before you can run your program, the linker (another system program) must be called to link it with the precompiled code in the C system libraries. Although the two processes *can* be done separately, linking is normally done automatically when compilation is successful. The result of linking is a **load module** or **executable file**; that is, a file that is ready to load into the computer's memory and execute. The programmer should give this file, which will be used to run the program, a convenient and meaningful name. In some systems, the name automatically will be the same as the name of the source file except that it ends in *.exe*. In UNIX, however, the name of the load module is *a.out* unless the user provides a better name as part of the compile command.

2.4 Source Code Construction

You have specified the whole project and designed an algorithm to do the computation. Now you are ready to write the actual code for your program. In the old days, when computers were scarce and expensive, programmers wrote out their code, in detail, on paper so that everything was complete before approaching the computer. This ensured a minimum amount of computer time was consumed. Today, computer time is no longer scarce, so experienced programmers often skip the paper-copy phase; they approach the computer with a detailed sketch of the code but without actually writing it in longhand. Others, especially beginners, still prefer to write the entire program on paper before sitting down at the computer. This permits them to separate the process of writing the code from the difficulties of dealing with a computer and a text editor. It also is helpful in getting "the big picture"; that is, seeing how all the parts of the program fit together. However, whether you are a beginner or an expert, it is important to have at least a well-developed and well-specified program sketch when you begin entering code into the computer. Programming without one is difficult and failure prone, because without a good plan, it is hard to know what to do and what order to do it in.

The steps in code construction are:

1. Create a project and a source file in a proper subdirectory.
2. Write or copy C code into the source file.
3. Build (compile, then link) the C code.
4. Correct compilation and linker errors and rebuild.
5. Run and test the executable program.

This text presents two methods for coding a program: Start from scratch or adapt an existing program to meet the new specification. The first several chapters rely primarily on adapting the programs given as examples in the text. In Chapter 9, after the fundamental concepts of programming have been covered, we discuss the process of building a program from scratch, using a specification.

Here, we continue with the exam average problem. An algorithm was given in Figure 2.1; a specification is given in Figure 2.3 a test plan in Figure 2.4 and a pseudocode program sketch in Figure 2.5. Now we are ready to write C code.

2.4.1 Create a Source File

To begin, you should create a new subdirectory (folder) on your disk for use only with this program. By the time a program is finished, it will have several component files (source code, backup, error file, object code, executable version, data, and output). Your life will be much simpler if you keep all the parts of a program together and separate them from all other programs.

Once you have made a directory for the new project, you are ready to create a new source code file in that directory. This can be done by copying and renaming an existing program you intend to modify or by starting with a blank document and storing it under your new program name. Then either type in your

```
#include <stdio.h>
int main( void )
{
    double n1, n2, n3;      /* The three input numbers.           */
    double average;         /* The average of the three numbers.  */

    puts( "Compute the average of 3 numbers" );
    puts( "--------------------------------" );
    printf( "Please input 3 numbers: " );
    scanf( "%lg%lg%lg", &n1, &n2, &n3 );   /* Read the numbers.   */

    average = (n1 + n2 + n3) / 3.0;        /* Average is sum / 3. */
    printf( "The average of  %g, %g, and %g = %g \n\n",
            n1, n2, n3, average );         /* Print answers.      */
    return 0;
}
```

Sample output:

```
Compute the average of 3 numbers
--------------------------------
Please input 3 numbers: 21 5 66
The average of  21, 5, and 66 = 30.6667
```

Figure 2.7. A C program.

new source code or modify the existing code. When you are finished, print out a copy of your file and look carefully at what you have done. (If your printer is inconvenient or produces output of poor quality, you may find it easier to examine the code on the video screen.) Mark any errors you find and use the text editor to correct them.

2.4.2 Coding.

With a firm idea of what the program must do, we can proceed to writing the steps in the C language. Figure 2.7 gives the C code that corresponds to the pseudocode in Figure 2.5. The notes, below, give a quick guide to the meaning of the C statements. Do not try to understand the C code at this time; all will be explained more fully in Chapter 3. This goal in this chapter is only to understand the process of program development, coding, translation, and execution.

1. Like all programs, this one starts with a command to include the declarations from the C input/output libary.
2. Next is the line that declares the beginning of the main program. That is followed by a pair of curly brackets that enclose the body of the program (declarations and instructions).

3. The first two lines of the program body are declarations that create and name space to store four numbers, $n1$, $n2$, $n3$, and *average*.
4. The words enclosed in `/* ... */` on the right end of these lines are comments. They are not part of the C code, but are written to help the reader understand the code.
5. The next two lines print the program title and a line of dashes to make the title look nice. Compare the code to the output, at the bottom of the Figure.
6. The `printf()` statement Instructs the user to enter three numbers at the keyboard. This kind of instruction is called a *user prompt* or, simply, a *prompt*.
7. The `scanf()` statement reads three numbers from the keyboard and stores them in $n1$, $n2$, and $n3$. This ends the input phase of the program, and we leave a blank line in the code.
8. Calculations in C are written much as they are in mathematics. The next line calculates the average and stores it for future use.
9. The next two lines are a `printf()` statement that displays the three input numbers and the average. The first line of this statement has a quoted string called a *format* that shows how we want the answers to be laid out. Each `%g` marks a spot where a number should be inserted. The second line lists the numbers that we want to insert in those spots.
10. The last line of code returns control to the system.

2.4.3 Translate the Program and Correct Errors

Before you can run your program, it must be compiled (translated into machine language) and linked with the previously compiled library code modules.

Compiling. When we use the compiler, we tell it the name of a C source file to translate. It runs and produces various kinds of feedback. If the job was done perfectly, the compiler will display some indication of success and create a new object file, which contains the machine code instructions that correspond to the C code in the source file. However, it is rare that all the planning, design, coding, and typing are correct on the first try. Human beings are prone to making mistakes and compilers are completely intolerant of errors, even very small ones. Spelling, grammar, and punctuation must be perfect or the program cannot be translated. The result is that the compiler will stop in the middle of a translation with an error comment (or a list of them). The programmer must locate the error, fix it, and use the compiler again.

Correcting compile-time errors. When we first entered the average program, we made typographical errors in two lines near the end. The code typed in looked like this:

```
average = (n1 + n2 + n3) / 3.0
print( "The average of  %g, %g, and %g = %g \n\n",
        n1, n2, n3, average );
```

The compiler responded with these error comments:

```
Compiling mean.c (2 errors)
    mean.c:18: illegal statement, missing ‘;’ after ‘3.0’
    mean.c:19: parse error before "print"
```

The problem is very clear: the line that computes the average needs a semicolon. This was corrected and the code was recompilied.

Interpreting error comments. Error comments differ from system to system. Language designers try to make these comments clear and helpful. However, especially with C, identifying the source of the error is sometimes guesswork, and the comments are often obscure or even misleading. For example, the comment `undefined symbol` might indicate a misspelling of a symbol that is defined, and `illegal symbol line 90` might mean that the semicolon at the end of line 89 is missing. Worse yet, a missing punctuation mark might not be detected for many lines, and an extra one might not be detected at all—the program just will not work right.

If there is a list of error comments, do not try to remember them all. Transfer the error comments to Notepad or to a new file and either print it or use a part of your screen to display it[2]. Then fix the first several problems. Very often, one small error near the beginning of a program can cause the compiler to become confused and produce dozens of false error comments. After fixing a small number of errors, recompile and get a fresh error listing. Often, many of the errors will just go away.

It takes some practice to be able to interpret the meaning of error comments. Sometimes beginners interpret them literally and change the wrong thing. When this happens, a nearly correct program can get worse and worse until it becomes really hard to fix. Anyone who is not sure of the meaning of an error comment should seek advice from a more experienced person.

Correcting linking errors. The linking process also may fail and generate an error comment. When a **linking error** happens, it usually means that the name of something in the library has been misspelled or that the linker cannot find the system libraries. The programmer should check the calls on the specified library function for correctness. If the compiler is newly installed, a knowledgable person should check that it was installed correctly. When we tried to link our program, the linker produced this error comment:

After fixing the semicolon error in our program, the compiler finished successfully, but the linker did not:

```
Compiling mean.c (1 warning)
    mean.c:19: warning: implicit declaration of function ‘print’
Linking /mean/build/mean (1 error, 1 warning)
    ld: warning prebinding disabled because of undefined symbols
    ld: Undefined symbols:
_print
```

The linker could not find a function named `print()` in the standard library. I changed it to `printf()` This time, both compilation and linking were successful and we were ready to run the program.

[2]In some systems, using the print-screen button is convenient

This diagram shows how the constant values used in the program of Figure 2.7 might be stored in main memory. Later chapters will explain why.

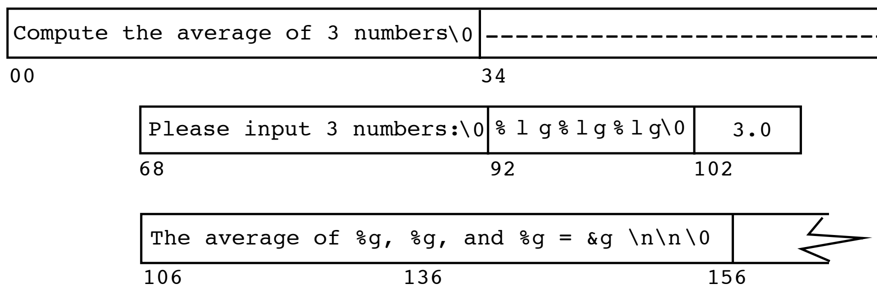

Compute the average of 3 numbers\0
--------------------------------\0
00
34
67
Please input 3 numbers:\0
% l g % l g % l g\0
3.0
68
92
102
The average of %g, %g, and %g = &g \n\n\0
106
136
156

Figure 2.8. Memory for constants.

2.5 Program Execution and Testing

2.5.1 Execute the Program

When a program has been successfully compiled and linked, the testing phase begins. An integrated development environment usually provides a icon or a RUN button to run the program; if so, just click on it. If not, you can start execution by clicking the mouse on the program's icon or typing the program's name on a command line. The operating system will respond by loading the executable program into the computer's memory. The memory will then contain the program instructions, any constants or quoted strings in the program, and space for the program's data. Figure 2.8 shows how the constants and data might be laid out in the memory for the program in Figure 2.7.

When loading is complete, the operating system transfers control to the first line of the program. At this time, you should begin to see the output produced by your program. If it needs input, type the input and hit the Enter key. When your program is finished, control will return to the system. When the program's instructions are executed, the computer's devices perform input and output and its logic circuits perform calculations. These actions produce data and results that are stored in the memory. In an integrated development environment, the system screen may disappear while your program is running and appear again when it is finished.

When a program is running, the contents of the memory cells change every time a `read` or `store` instruction is executed. In some systems, a program can be run with a debugger. A **debugger** is another program that runs the program for you. As shown in Figure 2.9, you communicate with the debugger, and it interprets your program instructions one at a time, step by step. You can ask to see the results of computations and the contents of selected parts of memory after each step and, thus, monitor how the process is proceeding. Figure 2.10 shows, step by step, how the values stored in the memory change when the program in Figure 2.7 is executed.

A program can be tested by running it directly (left) or through an on-line debugger (right).

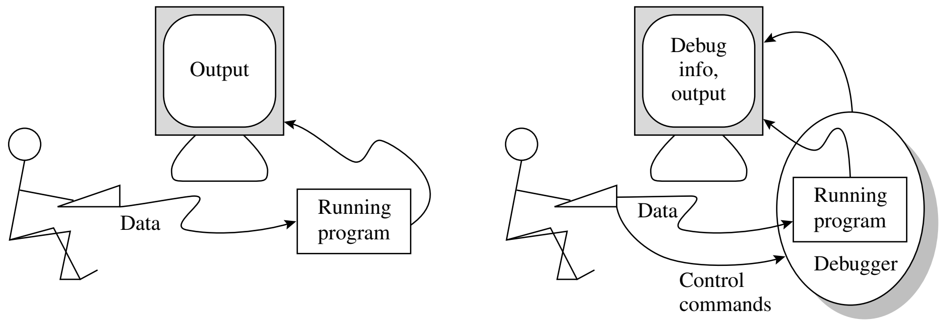
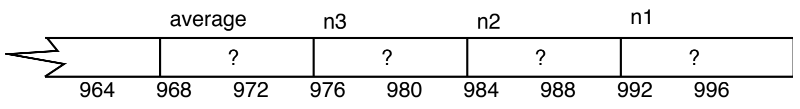


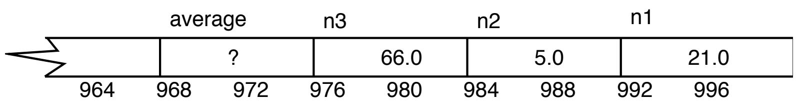
Figure 2.9. Testing a program.

This diagram shows how the values stored in main memory change when we run the program from Figure 2.7. A `?` indicates that any garbage value might exist at that location.

When the program is ready to run, the memory area for variables might look like this:



After returning from `scanf()`, with inputs of 21, 5, and 66,



After computing the formula and storing the answer,

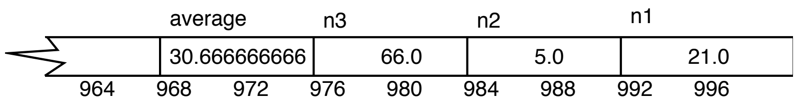


Figure 2.10. Memory during program execution.

A debugger can be a powerful tool for figuring out why and how an error occurs. Even though the C language is fully standardized, C debuggers are not. Each one is different. Because of this, it is helpful to learn other ways to monitor the progress of a program. Such techniques are presented in the next section.

2.5.2 Test and Verify

Many errors are caught by the compiler, some by the linker. When a program finally does compile and link without warnings, though, the process of finding the **logical errors** begins. The primary method for detecting logical errors is to test the program on varied input data and look carefully at the answers it prints. They might or might not be correct. Sometimes a program operates correctly on some data sets and incorrectly on others. Sometimes an unusual data condition will cause a program to crash or begin running forever. In any case, *you must verify the correctness of your answers.*

You now are ready to use your testing plan. Run your program and enter each of the data sets in your plan, verifying each time that the computer got the same answer you got by hand calculation. Compare the correct answer on your test plan to the answer the computer has printed. Are they the same? We tested our program using the plan in Figure 2.4; the output from our fourth test is shown here:

```
Compute the average of 3 numbers
--------------------------------
Please input 3 numbers: 21 5 66
The average of  21, 5, and 66 = 30.6667
```

This answer (and the others) matched the expected answers, so we have some confidence that the program is correct. If the answer is not correct, why not? Where is the error—in the test plan or in the program code? If the test plan has the correct answer, you must analyze the code and find the reason for the error; that is, you must find the logical error, better known as a *program bug*. The useful debugging method of using printouts is described next. Another technique (parse trees) is given in Section 4.7.

Debugging: locating and correcting errors. As you start writing your own programs, you must learn how to find and eliminate programming errors. Debugging printouts are a powerful technique for locating a logical error in a program. When your program computes nonsense answers, add debugging printouts until you discover the first point at which the intermediate results are wrong. This shows you where the problem is. With that information (and possibly some expert help), you can deduce why the code is wrong and how to fix it. In an integrated development environment, the debugger can be used to provide similar information.

Beginners tend to form wrong theories about the causes of errors and change things that were right. Sometimes this process continues for hours until a program that originally was close to correct becomes a random mess of damaging patches. The best defense against this is to be sure you understand the reason for the error before you change anything. If you do not understand it, ask for help. Be sure to save a backup copy of your program when you modify it. This can be helpful if you need to undo a "correction."

Finishing the test. Whether you find an error or a needed improvement, the next step is to figure out how to fix it and start the edit–compile–link–test process all over again. Eventually the output for your first test case will be correct and acceptably readable. Then the whole testing cycle must begin again with the other items on the test plan: Enter a data set, inspect the results critically, and fix any problems that become evident. When the last test item produces correct, readable answers, you are done.

Polishing the presentation. Now that we are sure the answers are correct, we look at the output with a critical eye. Is it clear and easy to read? Should it be improved? Is everything spelled correctly? Does the spacing make sense? Small changes sometimes make big improvements in readability.

2.6 What You Should Remember

2.6.1 Major Concepts

Facts about programs.

1. **Translation.** Programs are written in a computer language and converted to machine code by a translator, called a *compiler*.
2. **Syntax.** Programs must conform exactly to the rules for spelling, grammar, and punctuation prescribed by the language. Not every piece of source code describes a valid program. To be valid, every word used in the source code must be defined and the words must be arranged according to strict rules.
3. **Errors.** An invalid program can contain three kinds of errors: compilation errors, linking errors, and run-time errors. Invalid program code results in **compilation errors** or **linking errors** when translated. A runtime error happens when the machine malfunctions or the program tries do do an illegal operation, such as dividing by zero.
4. **Semantics.** A program, as a whole, has a *meaning*. The meaning of a C program is its effect when it is translated and executed.[3] Ideally, the meaning of a program is what the programmer intended. If not, we say that the program contains a bug, or **logic error**.

The stages of program development.

1. Problem definition.
2. Design of a testing process.
3. Design of a solution.
4. Program construction.
5. Program translation.

[3]In some ways, the meaning of a C program depends on the hardware on which it is executed. Although it always will produce the same results when executed on the same machine with the same data, the results might be different when executed on a different kind of computer.

6. Verification.

2.6.2 The moral of this story.

If you wish to avoid grief in your programming, you should take to heart these proverbs:

1. It is harder than it seems it ought to be.
2. The program itself is like the tip of an iceberg: It rests on top of a lot of invisible work and it will not float without that work.
3. Attention to detail pays off. Compilers expect perfection.
4. Debugging seldom is an easy process, and if a program is not well-structured, it can become a nightmare.
5. Every development step skipped is hours wasted. (Beginners rarely understand this.)
6. You cannot start a program the day before it is due and finish it on time.

2.6.3 Vocabulary

These are the most important terms and concepts presented or discussed in this chapter.

program	development environment	compile time
algorithm	text editor	compilation error
pseudocode	compiler	linking error
test plan	linker	execute a program
source code	debugger	run time
object code	debugging printouts	run-time error
executable file	statement	logical error
load module	declaration	program verification

2.7 Exercises

2.7.1 Self-Test Exercises

1. Use a dictionary to find the meaning and pronounciation of the word "pseudo". Use that definition to explain the meaning of "pseudocode".
2. Explain why a programmer must use a text editor, not a word processor.
3. Explain why a beginner usually fails to finish a program on time when he or she starts it the day before it is due.
4. True or false: A program is ready to turn in when it compiles, runs, and produces results. Please explain your answer.
5. True or false: When a program finishes running, it returns control to the operating system. Please explain your answer.

6. Explain the difference between

   (a) an algorithm and a program.
   (b) a command and a declaration.
   (c) a compiler and an interpreter.
   (d) source code and object code
   (e) an object code file and a load module.

2.7.2 Using Pencil and Paper

1. A test plan is a list of data inputs and corresponding outputs that can be used to test a program. This list should start with input values that are easy to check in your head, then include input values that are special cases and values that might cause trouble, perhaps by being too big or too small.

   (a) Following the example in Figure 2.3, create a problem specification for a program to convert Fahrenheit temperatures to Celsius using the formula

   $$Celsius = (Fahrenheit - 32.0) * \frac{5.0}{9.0}$$

   Specify the following aspects: scope, input or inputs required, output required, formulas, constants, computational requirements, and limits (if any are necessary) on the range of legal inputs.
   (b) Write a test plan for this algorithm. Include inputs that can be checked in your head, those that will test any limits or special cases for this problem, and typical input values. For each input, list the correct output.
   (c) Using English or pseudocode, write an algorithm for this problem. Attempt to verify the correctness of your algorithm using the test plan. (Do not attempt to actually write the program in C.)

2. Explain the difference between

   (a) pseudocode and source code.
   (b) a text editor and a word processor.
   (c) compile time and run time.
   (d) program verification and program execution.

3. Given the short C program that follows,

   (a) Make a list of the memory variables in this program.
   (b) Which lines of code contain operations that change the contents of memory? What are those operations?
   (c) Which lines of code cause things to be displayed on the computer's screen?

```
int main( void )
{
    double square;
    double number;

    printf( "Enter a number:" );
    scanf( "%lg", &number );
    square = number * number;
    printf( "The square of %g is %g\n", number, square );
    return 0;
}
```

2.7.3 Using the Computer

1. Decide which computer system you will use for these exercises. If it is a shared system, find out how to create an account for yourself.
   (a) Create an account and a personal directory for your work.
   (b) Find out how to create a subdirectory on your system. Create one called `info`.
   (c) You will use a text editor to type in your programs and data files. Some C systems have a built-in text editor; others do not. Find out what text editor you will be using and how to access it. Create a text file (not a program) containing your name, address, and telephone number on separate lines. Next, write the brand of computer you are using and the name of the text editor. Then write a paragraph that describes your past experience with computers. Save this file in your `info` directory and email it to your teacher.
   (d) Find out how to print a file on your system. Print out and turn in the file you created in (c).

2. Given the short C program that follows,
   (a) Make a list of the memory variables in this program.
   (b) Which lines of code contain operations that change the contents of memory? What are those operations?

```
int main( void )
{
    double base;          /* Input variable. */
    double height;        /* Input variable. */
    double area;          /* To be calculated. */

    printf( "Enter base and height of triangle: " );
    scanf( "%lg", &base );
    scanf( "%lg", &height );
    area = base * height / 2.0;
    printf( "The area of the triangle is %g\n", area );
    return 0;
}
```

3. Write a simple program that will output your name, phone number, E-mail address, and academic major on separate lines.