Chapter 3

The Core of C

The C language is a powerful language that has been used to implement some of the world's most complex programs. However, by learning a modest number of fundamental concepts, a newcomer to C can write simple programs that do useful things. As with any language, the beginner needs a considerable amount of basic information before it is possible to read or write anything meaningful. In this chapter, we introduce some of the capabilities of C by using them in simple, but practical, programs and explaining how those programs work. These examples introduce all the concepts necessary to read and write simple programs, providing an overview of the central parts of the language. Later chapters will return to all of the topics introduced here and explain them in greater detail.

3.1 The Process of Compilation

In Chapter 2, we discussed the process of creating a C program. One of the stages in that process is compilation (Section 2.2). The compilation step can also be broken into stages, as shown in Figure 3.1. It helps a programmer to understand what these stages are, and the problems that can arise at each stage.

Lexical analysis: Identify the words and symbols used in the code. Some words are built into C and form the core of the language; these are called *keywords* and are listed in Appendix C. New words can be defined by the programmer by giving a declaration. For example, in Figure 2.7, two declarations were used to define the words `n1, n2, n3`, and `average`.

Preprocessing: Carry out the preprocessor commands, which all start with the `#` symbol on the left end of the line. These commands are used to bring sets of C library declarations into your program. In this chapter, you will see that they can also be used to define symbolic names for constants.

Stage	Purpose
1. Lexical analysis	Identify the words and symbols used in the code
2. Preprocessing	Carry out the preprocessor commands
3. Parsing	Analyze the structure of each declaration and statement, according to the official grammar of the language
4. Code generation	Select the appropriate machine operation codes and produce the actual machine instructions that will carry out the C-language program statements The object code can be saved in a file or linked with library code modules, as shown in Figure 2.6, to form an executable program.

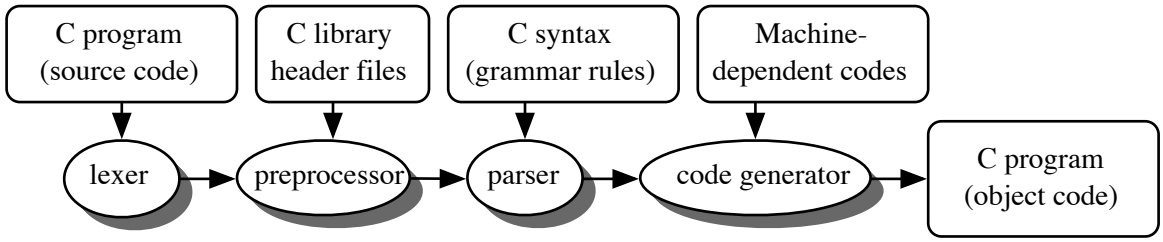


Figure 3.1. The stages of compilation.

Parsing: Analyze the structure of each declaration and statement according to the official grammar of the language. When the compiler parses your code it may discover missing punctuation, incomplete statements, bad arithmetic expressions, and various other kinds of structural errors.

Code generation: Select the appropriate machine operation codes and produce the actual machine instructions that will carry out the C-language program statements. If the compiler can parse your code correctly, and can find definitions for all of your symbols, it will generate machine code and store it in an object-code file.

3.2 The Parts of a Program

3.2.1 Terminology.

A C program is a series of comments, preprocessor commands, declarations, and function definitions. The function definitions contain further comments, statements, and possibly more declarations. Each of these program units is composed of a series of words and symbols. We define these terms very broadly in the next several paragraphs; their meaning gradually should become clearer as you look at the first several sample programs. As you read these definitions, you should refer to the diagram in Figure 3.2

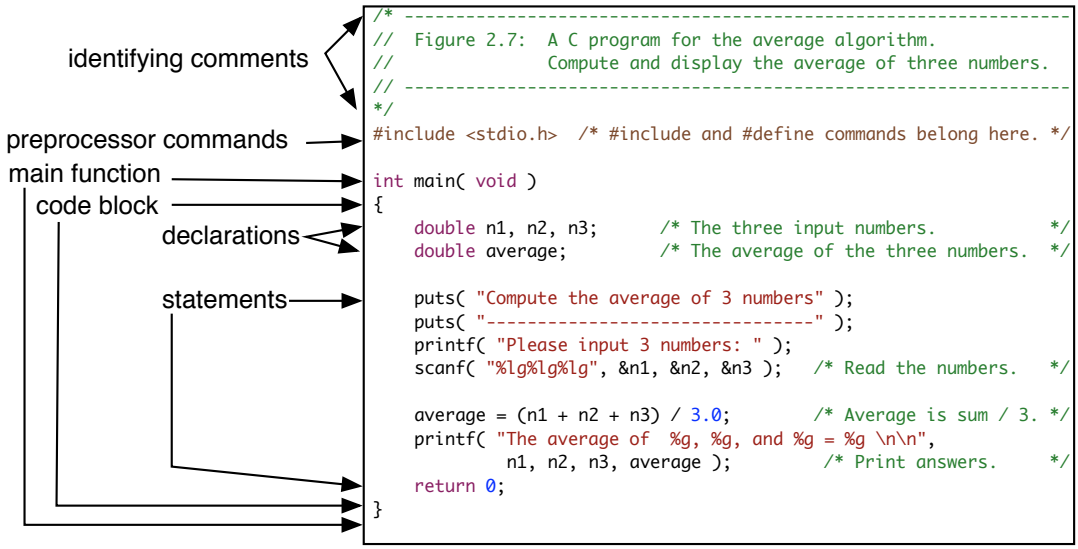


Figure 3.2. How to lay out a simple program.

Comments. At the top of a program, there should be a few lines enclosed between the symbols `/*` and `*/` that supply information about the program's purpose and its author. These lines are comments. Their purpose is to inform the human reader, not the computer, and so they are ignored by the compiler. Comments also can and should appear throughout the program, wherever they could help a reader understand the form or function of the code.

Preprocessor commands. Also, ly a series of commands at the top of a program tell the C compiler how to process the pieces of a program and where to look for essential definitions. These **preprocessor commands** all start with the symbol `#` and are handled by the C preprocessor before the compiler starts to translate the code itself, as indicated in Figure 3.1.

Words. Many of the words used to write a program are defined by the C language standard; these are known as *keywords*. A list of keywords can be found in Appendix C. Other words are defined by the programmer. These words can be grouped into categories analogous to English parts of speech; the table in Figure 3.3 lists the kinds of words in C and their English analogs.

Declarations. The purpose of a **declaration** is to introduce a new word, or **identifier,** into the program's vocabulary. It is like a declarative sentence: it gives information about the objects a program will use. In C, a declaration must be written in the program prior to any statement that uses the word it defines.

C Category	In English	Purpose
Identifiers	Nouns	Used to name objects
Data types	Adjectives	Used to describe the properties of an object
Operators	Verbs	Denote simple actions like add or multiply
Function calls	Verbs	Denote complex actions like finding a square root
Symbols	Punctuation	Symbols like a semicolon or # are used to mark the beginning or the end of a program unit
Symbols	Grouping	Pairs of parentheses, brackets, quotes, and the like are used to enclose a meaningful unit of code

Figure 3.3. Words in C.

Statements. A **statement** is like an imperative sentence; it expresses a complete thought and tells the compiler what to tell the computer to do. Just as an English sentence has a verb and a subject, a typical C statement contains one or more words that denote actions and one or several names of objects to use while doing these actions. The objects we study in this chapter are called *variables* and *constants*; the action words are assignment statements, arithmetic operations, and function calls. An entire program is like an essay, covering a topic from beginning to end.

Blocks. Sometimes, C statements are grouped together by enclosing them in curly brackets (braces). Such a group is called a *block*. Blocks are used with **control statements** to into action units that resemble paragraphs.

Putting it all together. This chapter shows how to use a subset of the C language we call the *beginner's toolbox*. It consists of the basic elements from each category: comments, preprocessor commands, declarations, objects, actions, and control statements. Each section will focus on one or a few elements and use them in a complete program. The accompanying program notes should draw your attention to the practical aspects of using each element in the given context.

The sample programs should help you gain a general familiarity with programming terminology and the C language and provide examples to guide the first few programming efforts. Each topic is revisited in more depth in later chapters. As you read this material, try to understand just the general purpose and form of each part. Then test your understanding by completing one of the skeletal programs given at the end of the chapter.

3.2.2 The main() program.

Every C program must have a function named `main()`, with a first line like the one shown here, followed by a block of code in curly brackets. Identifying comments and preprocessor commands come before `main()`. The last line of code in `main()` is `return 0`. This much never changes.

The keyword `void` appears in the parentheses after `main()`. This means that `main()` does not receive any information from the operating system. (Some complex programs do receive data that way.) The type

name `int` appears before the name `main()` to declare that `main()` will return a termination code to the operating system when the program finishes its work.

The `{` and `}` (curly bracket) symbols may be read as "begin" and "end." The brackets and everything between them is called a **program block**. For simplicity, we usually use the shorter term **block**. Every function has a block that defines its actions, and the statements in this block are different for every program. They are carried out, in the order they are written, whenever the function is executed. A simple sequential program format is described below; it has three main phases: input, calculation and output. Most short programs follow this pattern.

```
int main( void )
{
    Variable declarations, each with a comment that describes its purpose.

    An output statement that identifies the program.
    Prompts and statements that read the input data.
    Statements that perform calculations and store results.
    Statements that echo input and display results for user.
    return 0;
}
```

3.3 An Overview of Variables, Constants, and Expressions

Before we can write a program that does anything useful, we need to know how to get information into and out of a program and how to store and refer to that information within the program. This section focuses on how a program can define and name objects and how those objects can be used in computations. Several ways are introduced to write variable declarations and constant definitions. Formal rules and informal guidelines for naming these objects are discussed and informal rules for diagramming objects are presented.

3.3.1 Values and Storage Objects.

A **data value** is one piece of information. Data values come in several types; the most basic of these are numbers, letters, and strings of letters. When a program is running in a computer, it reads and writes data values, computes them, and sends them through its circuits. A computed data value might stay for a while in a CPU register, be stored into a variable in memory, or be sent to an output device. C keeps track of where data values are when they are being moved around within the computer, but you must decide when to output or store a value (assign it to a variable). C also permits you to give a symbolic name to a value.

We use the term *storage object*, or simply *object* to refer to any area of memory in which a value can be stored[1]. Objects are created and named by declarations. Each one has a definite type and an address. Many, but not all, contain values (the rest contain garbage). Some are variable, some are constant.

[1]This terminology is normal in C, and is consistent with the use of the term in C++. However, the term "object" in Java is used only for instances of a class.

3.3.2 Variables

A **variable** is an area of computer memory that has been given a name by the program and can be used to store, or remember, a value. You can visualize it as a box that can hold one data value at a time.

The programmer may choose any name (there are certain rules to follow when picking a name) for a variable, so long as it is not a keyword[2] in the language. Good programmers try to choose meaningful names that are not too long.

Each variable can contain a specific **type** of value, such as a letter or a number. In this chapter, we introduce the first three[3] data types:

- `char` is used to store characters (letters, punctuation, digits)
- `double` is used for real numbers such as 3.14 and .0056. This types can represent an immense range of numbers but does so with limited precision.
- `int` is used for integers such as 31 and 2006.

The amount of memory required for a variable depends on its type and the characteristics of the local computer system. Normally, simple variables are 1 to 8 bytes long. For example, an `int` (in many machines) uses four bytes of memory while a `double` occupies eight, and a `char` only one[4].

Declarations. A variable is created by a **declaration**. (Unlike some languages, C requires every variable name to be declared explicitly.) The declaration specifies the type of object that is needed, supplies a name for it, and directs the compiler to allocate space for it in memory. Declarations can appear at the beginning of any block of code, just after the opening `{`. Only the statements within a block can use the names declared at its beginning. We call them **local names** because they are not "visible" to other blocks.

The simplest declaration is a type name followed by a variable name, ending with a semicolon. The following declaration creates an **integer variable** named `minutes`:

```
int minutes;
```

For each name declared, the compiler will create a storage object by allocating the right amount of memory for the specified type of data. The address of this storage object becomes associated with or *bound* to the variable name. Thereafter, when the programmer refers to the name, the compiler will use the associated address. Sometimes we need to refer explicitly to the address of a variable; in those contexts, we write an ampersand (which means "address of") in front of the variable name. For example, `&minutes` means "the address of the variable `minutes`."

[2]Keywords are words such as `main()` and `void` that have preset meanings to the translator and are reserved for specific uses. These are listed in Appendix C.

[3]Other types are introduced in Chapters 7, 15, and 11,

[4]A full treatment of types `int` and `double` is given in Chapter 7 where we deal with representation, overflow, and the imprecise nature of floating point numbers.

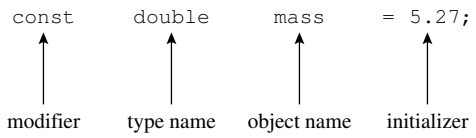


Figure 3.4. Syntax for declarations.

Syntax for declarations. The general form of a declaration, shown in Figure 3.4, contains four parts: zero or more modifiers,[5] a type name, an object name or names, and an optional initializer for each name. The parts are written in the order given, followed by a terminating semicolon, as diagrammed in Figure 3.4. The various different data types will be explored in the next few chapters. Object naming conventions are discussed shortly. An **initializer** gives a variable an initial value. This is done by using an `=` sign, followed by a value of the appropriate type.[6]

If multiple objects are declared, and possibly initialized, using a single statement, the objects are separated by commas. We have adopted a style in which most declaration lines declare one variable and have a comment explaining the use of that variable. This is a good way to write a program and make it self-documenting.

Declaration examples. Two examples of variable declarations were given in Chapter 2, Figure 2.7, which declares four `double` variables, `n1, n2, n3`and `average`. Figure 3.5 illustrates variations on the basic declaration syntax and shows how to draw diagrams of variables. The first line declares one variable of type `double`, as we have done many times already. This line tells C to allocate enough memory to store a `double` value and use that storage location whenever the name `length` is mentioned. The declaration does not put a value into the variable; it will contain whatever was left in that storage location by the program that previously ran on the computer. This value is unpredictable and meaningless in the present context. Formally, we say that it is an **undefined value**; informally, we call it **garbage**. The garbage stays there until a value is read into the variable (perhaps by `scanf()`) or stored in the variable by an assignment statement.

A variable is diagrammed, as in Figure 3.5, by drawing a box with the variable name just above the box and the current value inside it. The size of each box is proportional to the number of bytes of storage required on a representative C system. In the diagram of `length`, its undefined value is represented by a question mark. Variable diagrams, or **object diagrams**, are a concrete and visual representation of the abstractions that the program manipulates. We use them to visualize relationships among objects and to

[5]In the C standard, the properties `const` and `volatile` are called *type qualifiers* and `extern`, `static`, `register`, and `auto` are called *storage class specifiers*. For simplicity, and because the distinctions are not important here, we refer to both as *modifiers*. The modifiers are optional and, with the exception of `const`, you need not understand or use them for the time being. We use `volatile` in Chapter 15. We discuss `static` in Chapter 10 and all storage classes, in general, in Chapter 19. Additionally, the modifier `extern` is important in complex, multimodule programs and is demonstrated in Chapter 20.

[6]The rules for initializers will be given as each type is considered.

Four variables are declared here and diagrammed below.

```
int main( void )
{
    double length;            /* Length, in meters (uninitialized). */
    double weight = 1.5;      /* Weight, in kilograms (initialized to 1.5). */
    int k, m = 0;             /* One unitialized and one initialized variable. */
    char gender = 'F'         /* Use a char literal to initalize a char variable. */
    ...                       /* Rest of program goes here. */
}
```

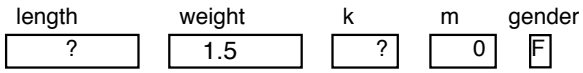


Figure 3.5. Simple declarations.

perform step-by-step traces of program execution. Object diagrams become increasingly important when we introduce compound data objects such as arrays, pointers, and data structures.

The second line in Figure 3.5 declares the variable `weight` and gives it an initial value. This shorthand notation combines the effect of a declaration and an assignment:

```
double weight;      /* Weight, in kg.  Contains garbage. */
...
weight = 1.5;       /* Now weight contains value 1.5 */
```

As we progress to more complex programs, the ability to declare and initialize a variable in one step will become increasingly important.

The third line in Figure 3.5 declares two variables, `k` and `m`, and it initializes `m`. It does not initialize `k`. To do that, another `=` sign and value would be needed prior to the comma. This omission is a common mistake of beginning programmers. Combining two declarations in this manner on one line saves space and writing effort. However, it does not provide a place to put separate comments explaining the purpose of the variables and so should be done only when the meaning of the variables is the same or they are logically related.

Caution. C has a somewhat arbitrary restriction that all of the declarations in a program block must come immediately after the `{` that follows `main( void )` and before any statements. (This restriction is relaxed in C++.)

Assignment. An assignment statement is one way to put a value into a variable. The term *variable* is used because a program can change the value stored in a variable by assigning a new value to it. An **assignment**

begins with the name of the variable that is to receive the value. This is followed by an = sign and a value to be stored (or an arithmetic expression that computes a value, see later). As an example,

```
minutes = 10;    /* Store value 10 in variable "minutes". */
```

3.3.3 Constants and Literals

Literals. In addition to variables, most programs contain **constants**, which represent values that do not change. A **literal constant** is one that is written, literally, in your code; in the assignment statement `radius = diameter/2.0; 2.0` is a literal `double`. For each type of value in C, there is a way to write a literal constant. These will be explained as we go through the various types. Here is a table of literal formats for the three types we have covered

C Syntax for Literal Constants

Type	Examples	Notes
`double`	`3.1, .045`	A string of digits with a decimal point.
`double`	`4.5E-02`	.045 written in scientific notation.
`int`	`32767`	A string of digits with an optional sign; no decimal point, no commas.
`char`	'A' or '%'	A single character enclosed in single quotes.

Symbolic constants. A **symbolic constant** is a constant to which we have given a name. There are two ways in C to create a symbolic constant: a `#define` command and a `const` declaration. Physical constants such as π often are given symbolic names. We can define the constant `PI` as

```
#define PI 3.1416
```

The `#define` commands usually are placed after the `#include` commands at the top of a program. It is sound programming practice to use `#define` to name constants rather than to write literal constants in a program, especially if they are used more than once in the code. This practice makes a program easier to debug because it is easier to locate and correct one constant definition than many occurrences of a literal number.

Note three important syntactic differences between `#define` commands and assignment statements: (1) the `#define` command does not end in a semicolon while the assignment statement does; (2) there is no = sign between the defined constant name and its value, whereas there always is an = sign between the variable and the value given to it; and (3) you cannot use the name of a constant on the left side of an assignment statement, only a variable.

Often, using symbolic names rather than literal constants saves trouble and prevents errors. This is true especially if the literal constant has many digits of precision, like a value for π, or if the constant might be changed in the future. Well-chosen descriptive names also make the program easier to read. For example, we might wish to use the temperature constant, `-273.15`, which is absolute zero in degrees Celsius. A name like `ABS0_C` is more meaningful than a string of digits.

Using `#define`. A `#define` command is generally used to define a physical constant such as `PI`, `GRAVITY` (see Figure 3.7) or `LITR_GAL` (see Figure 3.18), or to give a name to an arbitrarily defined value that is used throughout a program but might have to be changed at some future time. (The first example of such usage is the loop limit `N` in Figure 6.14.) We use a `#define` for an arbitrary constant so that changing the constant is easy if change becomes necessary. Changing one `#define` is easier than changing several references to the constant value, and finding the `#define` at the top of the file is easier than searching throughout the code for copies of a literal constant.

When you use a name that was defined with `#define`, you actually are putting a literal into your program. Commands that start with `#` in C are handled as a separate part of the language by a part of the compiler, called the *preprocessor*, that looks at (and possibly modifies) every line of source code before the main part of the compiler gets it. The preprocessor identifies the `#define` commands and uses them to build a table of defined terms, with their meanings. Thereafter, each time you refer to a defined symbol, the preprocessor removes it from your source code and replaces it by its meaning. The result is the same as if you wrote the meaning, not the symbol in your source code. For example, suppose a program contained these two lines of source code before preprocessing:

```
#define PI  3.1415927
area = PI * radius * radius;
```

After preprocessing, the `#define` is gone. In some compilers, you can instruct the compiler to stop after preprocessing and write out the resulting code. If you were to look at the preceding assignment statement in that code, it would look like this:

```
area = 3.1415927 * radius * radius;
```

Note the following things about a `#define` command:

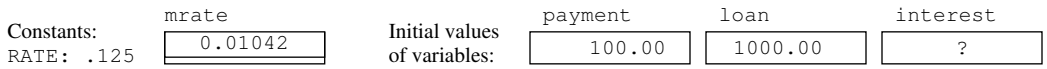
- *No* `=` sign appears between the constant's name and its value.
- The line does *not* end in a semicolon, and you do *not* write the type of the name. (The C translator will deduce the type from the literal.)
- In this example, `PI` is a `double` constant because it has a decimal point.

The C preprocessor always has been a source of confusion and program errors. However, these difficulties are caused by advanced features of the preprocessor, not by simple constant definitions like those shown here. A beginning programmer can use `#define` to name constants with no difficulty. The programmer usually is unaware of the substitutions made by the preprocessor and does not see the version with the literal constants.

The `const` qualifier. You also can create a constant by writing the `const` *modifier* at the beginning of a variable declaration. This creates an object that is like a variable in every way except that you cannot change its value (so it is not really like a variable at all).[7]. A `const` declaration is like a variable declaration except that it starts with the keyword `const` and it must have an initializer.

[7] A `const` variable has an address. This becomes important in advanced programs and in C++.

```
#define RATE .125                /* Annual interest rate. */
const double mrate = RATE/12;    /* Monthly interest rate. */
double payment = 100.00;         /* Monthly payment. */
double loan = 1000.00;           /* Remaining unpaid principle amount. */
double interest;                 /* Current month's interest. */
```




```
interest = mrate * loan ;
loan = loan + interest - payment;
```

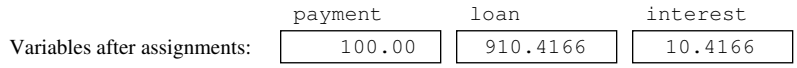


Figure 3.6. Using constants.

A symbolic name is one way to clarify the purpose of a constant and the meaning of the statement that uses the constant. That C provides two different ways to give a symbolic name to a constant value (`#define` and `const`) might seem strange. C does so because each kind of constant can be used to do some advanced things that the other cannot. We recommend the following guidelines for defining constants:[8]

- Use `#define` to name constants of simple built-in types.
- Use `const` to define anything that depends on another constant.

This usage is illustrated in Figure 3.6. The lines in this figure are excerpted from a program that computes a payment table for a loan. The program uses `#define` for the annual interest rate, because while it is constant for this particular loan, it is likely to change for future loans. By placing the `#define` at the top of the program, we make it easy to locate and edit the interest rate at that future time. The monthly rate is one-twelfth of the annual rate; we declare it as a `const double` initialized to `RATE/12`. We use `const` rather than `#define` because the definition involves a computation.

The loan amount and the monthly interest are defined as variables, because they decrease each month. The monthly payment normally will be $100 but we do not define it as a constant because the loan payment on the final month will be smaller.

Following the declarations in the figure are diagrams for the objects declared. The variables `payment`, `loan`, and `interest` are diagrammed as variable boxes. The striped box indicates that `mrate` is a constant variable and cannot be changed. No box is drawn for `RATE` because it is implemented as a literal. The lower line in the diagram shows the changes in the variables produced by the two assignment statements.

[8] These guidelines are consistent with the advanced uses of constants and with usage in C++.

3.3.4 Names and Identifiers

As a programmer works on the problem specification and begins writing code to solve a problem, he or she must analyze what variables are needed and invent names for them. No two programmers will do this in exactly the same way. They probably will choose different names, since naming is wholly arbitrary. They might even use different types of variables or a different number of variables, since the same goals can often be accomplished in many ways. In this section, we introduce guidelines and formal syntactic rules for naming objects.

The technical term for a name is an **identifier**. You can use almost any name for an object, subject to the following constraints. These are absolute rules about names:

1. It must begin with a letter or underscore.
2. The rest of the name must be made of letters, digits, and underscores.
3. You cannot use C keywords such as `double` and `while` to name your own objects. You also should avoid the names of functions and constants in the C library, such as `sin()` and `scanf()`.
4. C is case sensitive, so `Volume` and `volume` are different names.
5. Some compilers limit names to 31 characters. Very old C compilers use only the first eight characters of a name.

These are guidelines for names:

1. Use one-letter names such as `x, t`, or `v` to conform to standard engineering and scientific notation. Writing `d = r * t` is better for our purposes than writing the lengthier `distance = rate * time`. Otherwise, avoid single-letter names.
2. When you have two similar quantities, such as two time instances, you might call them `t1` and `t2`. Otherwise, avoid using such similar names.
3. Use names of moderate length. Most names should be between 2 and 12 letters long.
4. Avoid names that look like numbers; `O`, `l`, and `I` are very bad names.
5. Use underscores to make compound names easier to read: `tot_vol` or `total_volume` is clearer than `totalvolume`.
6. Try to invent meaningful names; `x_coord` and `y_coord` are better names than `var1` and `var2`.
7. Do not use names that are very similar, such as `metric_distance` and `meter_distance` or `my_var` and `my_varr`.

3.3.5 Arithmetic and Formulas

Arithmetic formulas, which are called **expressions**, are written in C in a notation very much like standard mathematical notation, using the **operators** `+` (add), `-` (subtract), `*` (multiply), and `/` (divide). Normal

mathematical operator **precedence** is supported; that is, multiplication and division will be performed before addition and subtraction. As in mathematics, parentheses may be used for grouping[9]. These operators are combined with variable names (such as `radius`), constants (such as `PI`), or **literal** values such as 3.14 or 10 to form expressions. The result of an expression can be used for output or it can be stored in memory by using assignment. For example, the following statement computes the area of a circle with radius `r` and stores the result in the variable `area`:

```
area = 3.1416 * r * r;
```

Using a constant definition (`#define PI 3.1416`) permits us to rewrite the area formula thus:

```
area = PI * r * r;
```

3.4 Simple Input and Output

A call on an input function is one way to put a value in a variable; we call an output function when we want to see the value stored there. The input and output facilities in C are some of the most complex parts of the language, yet we need to use them as part of even the simplest programs. The best way to start is to learn to do input and output in a few simple ways. In this chapter, we focus on the elementary use of just three I/O functions:

Function name	Meaning of name	Purpose of function
`puts()`	Put string and newline	To write a message and a newline on the screen
`printf()`	Print output using a format	To write messages and data values.
`scanf()`	Scan input using a format	To read a data value from the keyboard

These functions (and many others) are in the standard input/output library (`stdio`), which is "added" to the program when you write `#include <stdio.h>`. Several examples given in this chapter use these functions with the hope that you can successfully imitate them.[10]

Streams and buffers. Input and output in C are handled using **streams**. An *input stream* is a device-independent connection between a program and a specified source of input data; an *output stream* connects a program to a destination for data. In this book, we use streams connected to data files or devices such as the computer's keyboard and monitor screen. Three streams are predefined in standard C: one for input (`stdin`), one for output (`stdout`), and one for error comments (`stderr`). The standard output stream is directed by default to the operator's video screen but can be redirected to a printer or a file.[11] The standard

[9] Many more operators and the details of operator precedence and associativity are given in Chapter 4

[10] A note about notation: In writing about C functions, we often use the name of a function separately, outside the context of program code. At these times, it is customary to write an empty pair of parentheses after the function name. The parentheses remind us that we are talking about a function rather than some other kind of entity. In a program, they distinguish a function call (with parentheses) and a reference to a function (without).

[11] The complexities of streams will be explored in Chapter 14.

input stream normally is connected to the keyboard but also can be redirected to get information from a file.

Input and output devices are designed to handle chunks of data. For example, a keyboard delivers an entire line of characters to the computer when you hit Enter, and a hard disk is organized into clusters that store 1,000 or more characters. When we read from or write to a disk, the entire cluster is read or written. On the other hand, programs typically read data only a few numbers at a time and write data one number or one line at a time. To bridge the gap between the needs of the program and the characteristics of the hardware, C uses buffers. Every stream has its own **buffer**, an area of main memory large enough to hold a quantity of input or output data appropriate for the stream's device.

When the program uses `scanf()` to read data, the input comes out of the buffer for the `stdin` stream. If that buffer is empty, the system will stop and wait for the user to enter more data. If a user types more data than are called for, the extra input remains in the input buffer until the next use of `scanf()`.

Similarly, when the program uses `printf()` to produce output, that output goes to the output buffer and stays there until the program prints a newline character (denoted by `\n`) or until the program stops sending output and switches to reading input. This permits a programmer to build an output line one number at a time, until it is complete, then display the entire line. However, the most common use of `printf()` is to print an entire line at one time.

3.4.1 Formats.

Some input and output functions, like `puts()`, read or write a single value of a fixed type. A call on `puts()`, which outputs a single string, is very simple: We write the message enclosed in double quotes inside the parentheses following the function name. For example, we might write this lines as the first statement in a program that computes square roots:

```
puts( "Compute the square root of a number." );
```

Other input and output functions, including `scanf()` and `printf()`, can read or write a list of values of varying types. Using these functions is more complex than using `puts()` because two kinds of things may be written inside the parentheses. First comes a **format** string, which describes the form of the data. It describes how many items will be read or written and the type of each item. Following that is either a list of addresses for the input data (for `scanf()`) or a list of expressions whose results are to be printed (for `printf()`). The complete set of rules for formats is long and detailed.[12] Fortunately it is not necessary to understand formats fully in order to use them. In this chapter, we show how to write simple formats for three types of data.[13]

A format is a string (a series of characters enclosed in double quotes) that describes the form and quantity of the data to be processed. Input and output formats are quite different and will be described separately. Both, however, contain conversion specifiers. The **conversion specifiers** in the format tell the input or output function how many data items to process and what type of data is stored in each item. Each

[12] These rules can be found in any standard reference manual, such as S. Harbison and G. Steele, *C: A Reference Manual*, 4th ed (Englewood Cliffs, NJ: Prentice-Hall, 1995).

[13] As other types of data are introduced in Chapters 7 through 11, more details about formats will be presented.

conversion specifier starts with a percent sign and ends with a code letter(s) that represents the type of the data.

Data Type	Input Specifier	Output Specifier	Notes
`char`	`" %c"`	`"%c"`	Type a space between the quote and the `%` sign for input.
`int`	`"%i"`	`"%i"`	`"%d"` also works.
`double`	`"%lg"`	`"%g"`	Use `%lg` for input, but only `%g"` (without the letter `l`) for output.

Using input formats. An input format is a series of conversion specifiers enclosed in quotes; for example, the format string `"%i"` could be used to read one integer value and `"%i%i"` could be used to read two integers that are separated by spaces on the input line. In a call on `scanf()`, the format is written first, followed by a list of the addresses of the variables that will receive the data after they are read. Here are two complete calls on `scanf()`:

```
scanf( "%i", &minutes );
scanf( " %c%i%lg", &gender, &age, &weight );
```

The first line tells `scanf()` to read one integer value and store it at the memory location allotted to the variable named `minutes`. The second line tells `scanf()` to read a character an integer, and a real number. The character will be stored at the address of the variable named `gender`, the integer in the variable named `age`, and the real number in the variable named `weight`.

Using output formats. Output formats contain both conversion specifiers and words that the programmer wishes to see interspersed within the data. Therefore, output formats are longer and more complex than input formats. An appropriate output format to print the data just read might look like this:

```
"Gender: %c  Age: %i  Weight: %g\n"
```

The words and the spaces are written exactly the way they should appear on the screen. The `%c`, `%i` and `%g` tell `printf()` where to insert the data values in the sentence. The `\n` represents a newline character. We need it with `printf()` to cause the output cursor to go to a new line. (The `\n` is not needed with `puts()`.) Hence, most `printf()` formats end in a newline character. Make this a habit: *Use* `\n` *at the end of every format string to send the information to your screen immediately and prepare for the next line.*[14]

Following the format string in a call on `printf()` is the list of variables or expressions whose values we want to write (do not use the ampersand for output). Exactly one item should appear in this list for each `%` in the format. A complete call on `printf()` might look like this:

```
printf( "Gender: %c  Age: %i  Weight: %g\n", gender, age, weight );
```

[14]The exception is a format string used to display a user prompt. These normally end in a colon and a space so that the screen cursor does not move to a new line and the user types the input on the same line as the prompt.

This tells `printf()` to print the values stored in the variables gender, age, and weight. These values will appear in the output after labels that tell the meaning of each number. The output from this line might be

```
Gender: M  Age: 21  Weight: 178.5
```

3.5 A Program with Calculations

Now you know how to define constants and variables, how to get letters and numbers into and out of the computer's memory, and how to do simple calculations. You know, in general, the form of a program. In this section, we combine all these elements in a simple program in Figure 3.7 to illustrate variable declarations, assignments, input, output, and calculations. In this program, some parts of the code are boxed. Program notes corresponding to these boxes are given below.

The problem. A grapefruit is dropped from the top of a very tall building[15]. It has no initial velocity since it is dropped, not thrown. The force of gravity accelerates the fruit. Determine the velocity of the fruit and the distance it has fallen after `t` seconds. The time, `t`, is read as an input.

Notes on Figure 3.7: Grapefruits and gravity.

Introductory comments.

- The first two lines of "Grapefruits and Gravity" are a comment block. The first line of such a block starts with `/*` and the last line ends with `*/` Other lines in the block do not need any special mark at the beginning, but using ** or `//` (as shown here) makes an attractive heading.
- Good programmers put comments at the top of a program to identify the program, its author, and the date or version number. Comments are also written throughout the code to explain the purpose of each group of statements. The compiler does not use the comments—we write them for the benefit of the humans who will read the code.

First box: preprocessor commands.

- Any line that starts with a `#`, called a *preprocessor command*, is handled by the preprocessor before the compiler translates the code.
- In this program, as in most, we ask the preprocessor to bring in the file `stdio.h` and include the contents of that file as part of the program. This is the header (basically, the table of contents) for the standard I/O library. This command makes the standard input-output library functions available for use.
- We use a `#define` command to give a symbolic name to the constant for the acceleration of gravity at sea level.

[15] This problem will serve as the basis for a series of example programs in the following sections.

```
/* Determine the velocity and distance traveled by a grapefruit
// with no initial velocity after being dropped from a building.
*/

#include <stdio.h>
#define GRAVITY 9.81         /* gravitational acceleration (m/s^2)     */

int main( void )
{
    double t;   /* elapsed time during fall (s)                        */
    double y;   /* distance of fall (m)                                */
    double v;   /* final velocity (m/s)                                */

    printf( "\nWelcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " );  /* prompt for the time, in seconds. */
    scanf ( "%lg", &t );           /* keyboard input for time          */

    y = .5 * GRAVITY * t * t;      /* calculate distance of the fall   */
    v = GRAVITY * t;               /* velocity of grapefruit at impact */

    printf( "    Time of fall = %g seconds \n", t );
    printf( "    Distance of fall = %g meters \n", y );
    printf( "    Velocity of the object = %g m/s \n", v );

    return 0;

}
```

Figure 3.7. Grapefruits and gravity.

Second box: the declarations.

- Declarations are used to define the names of variables that will be used in statements later in the program.
- We declare three `double` variables named `t, y`, and `v`. This instructs the compiler to allocate enough space in memory to store three real values and use the appropriate location every time we refer to `t, y`, or `v`. On most common machines, 8 bytes of memory will be allocated for each variable.
- Note that `t, y`, and `v` all are variables of type `double`. When you have several variables of the same type, you also declare them on separate lines, as shown, or you may declare them all on one line like this: `double t, y, v;`. In this problem, we declare one variable per line so that there is space for a comment that explains the meaning or purpose of each variable. This is good programming practice.

Third box: printing an output title.

- A well-designed program displays a title and a greeting message so that the user knows that execution has started.
- We could use either `puts()` or `printf()` to print the title, but if we use `printf()` we must end the string with a newline character if we want the output cursor to move down to the next line.
- Here, we print a 3-line title, using three lines of code. Where we break the code, we end the line with a quote mark and start the next with indentation and a quote mark.
- Program execution begins with the first box following the declarations, proceeding sequentially through the other boxes to the end of the code. This is studied more fully in the next section.

Fourth box: user input.

- A **prompt** is a message displayed on the video screen that tells the user what to do. A program must display a prompt whenever it needs input from the human user.
- In this prompt, we ask the user to type in the time it took the grapefruit to hit the ground. We *do not* put a newline character at the end of the message because we want the input to appear on the same line as the prompt. We *do* leave a space after the colon to make the output easy to read. The user sees

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 10
```

- In the call on `scanf()`, we send two pieces of information to the `scanf()` function: the format string and the address of the variable to receive the input. This causes the system to scan the `stdin` input stream to find and read a value for `t` (time, in seconds). This value may be entered with or without a decimal point.
- The `%` sign in the format is the beginning of a conversion specifier. It tells us to read one item and specifies the type of that item. We use `%lg`, for "long general-format real," to read a value of type `double`. (The letter between the `%` and the `g` is a lowercase *letter* `l`, not a numeral 1.)
- After the format comes the address for storing the input data. In this case, the data will be stored at `&t`, the address of the variable `t`.

Fifth box: calculations.

- The symbol `=` is used to store a value into a variable. Here we store the result of the calculation `.5 * GRAVITY * t * t`, which is the standard equation for distance traveled under the influence of a constant force after a given time, in the variable named `y`.
- The calculation for height, `y`, uses several `*` operators. The `*` means "multiply." When there is a series of `*` operators, they are executed left to right.
- The second formula is standard for computing the terminal velocity of an object with no initial velocity under the influence of a constant force.

Sixth box: the output.

- First, the input value (time) is echoed onto the video screen to convince the user that the calculations were done with the correct value.
- Two `printf()` statements write the answers to the screen. We use a `%g` specifier to write a `double` value. Note that this is different from a `scanf()` format, where we need `%lg` for a `double`.
- The `printf()` function leaves the output cursor on the same output line unless you put a newline character, `\n`, in the format string. In this case, we use a `\n` to put the velocity value on a different line from the distance.
- This would be a typical output:

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 10
   Time of fall = 10 seconds
   Distance of fall = 490.5 meters
   Velocity of the object = 98.1 m/s

Gravity has exited with status 0.
```

Last box: termination.

- It is sound programming practice to display a termination comment. This leaves no doubt in the mind of the user that all program actions have been completed normally.
- At the top of every program, we write `int main( void )`. This tells the operating system to expect to receive a termination code from your program.
- The last line in every program should be a `return 0;` statement. The zero is a termination code that tells the system that termination was normal.

3.6 The Flow of Control

All the examples we have considered so far execute the code (instructions) line by line from beginning to end. However, in most practical programs, it is necessary to follow alternative paths of execution, depending

This is a flow diagram of the program in Figure 2.7. It illustrates simple straight-line control and the way we begin and end the diagram of a function.

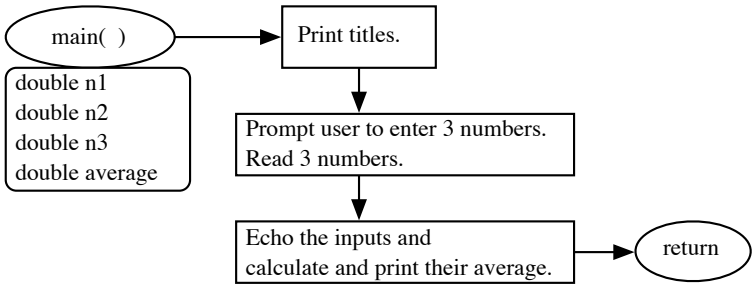


Figure 3.8. A complete flow diagram of a simple program.

on the data values and other conditions. For example, suppose you want to calculate the velocity of an object, as in Figure 3.7, but you want to prohibit cases that make no physical sense; that is, negative values of time. Two potential courses of action could be taken: (1) Do the calculation and display the answer or (2) comment on the illegal input data and skip the calculation. Another reason we need nonsequential execution is to enable a program to analyze many data values by repeating one block of code.

When a C program is executed, action starts at the first statement following the declarations. For example, in Figure 3.7, execution starts with the `printf()` statement. From there, execution proceeds to the next statement and the next, in order, until it reaches the `return` statement at the program, at which point control returns to the operating system. This is called **simple sequential execution**. *Control statements* are used to create conditional branching and repetitive paths of execution in a program. *Flow diagrams* are used as graphical illustrations of the execution paths that these control statements create.

A **flow diagram** is an unambiguous, complete diagram of all possible sequences in which the program statements might be executed. We use flow diagrams to illustrate the flow of control through the statements of a program or part of a program. This is not so necessary for simple sequential execution. However, a two-dimensional graphic representation can greatly aid comprehension when control statements are used to implement more complex sequencing. A few basic rules for flow diagrams follow and are illustrated in Figure 3.8:

1. A flow diagram for a complete program begins with an oval "start" box at the top and ends with an oval return box at the bottom.
2. Declarations need not be diagrammed unless they contain an initializer (used to give the variable a starting value and discussed more in the next chapter). It is probably clearer, though, if you include one round-cornered box below the start oval that lists the variables declared within the function.
3. The purpose of each statement is written in the appropriate kind of box, each depending on the nature of the action. Since the purpose of the diagram is to clarify the logic of the function, we use English

This is a flow diagram of the program in Figure 3.7.

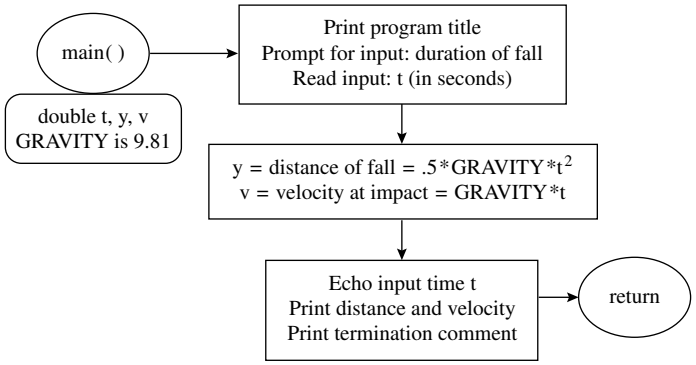


Figure 3.9. Diagram of the grapefruits and gravity program.

or pseudocode, not C, to describe the actions of the program. Such a diagram could be translated into languages other than C.

4. Simple assignment statements and function calls are written in rectangular boxes. Several of these may be written in the same box if they are sequential and relate to each other.
5. Arrows connect boxes in the sequence in which they will be executed, from start to finish. In the diagram of a complete function, no arrow is left dangling in space and no box is left unattached. All arrow heads must end at a box of some sort, except in the diagram of a program fragment, where the beginning and ending arrows might be left unattached.
6. The diagrams are laid out so that flow generally moves down or to the right. However, you may change this convention if it simplifies your layout or makes it clearer.
7. No arrow ever branches spontaneously. Every tail has exactly one head.

As another example, Figure 3.9 shows the diagram of the program in Figure 3.7. The graph consists of five nodes connected by arrows that indicate the flow of control during execution:

1. A start oval at the top, attached to a box listing the variable declarations.
2. A rectangular box listing statements that print a title and prompt for and read the input.
3. A box that calculates the values for distance and velocity using appropriate formulas.
4. A box that echoes the input and outputs the answers.
5. A return oval at the bottom that terminates the program.

Note that these boxes correspond roughly to the boxed units of code in the program.

3.7 Asking Questions: Conditional Statements

A large part of the power of a computer is the ability to take different actions in different situations. For this purpose, all computers have instructions that do various kinds of conditional branching. These instructions are represented in C by the `if` and `if...else` statements.

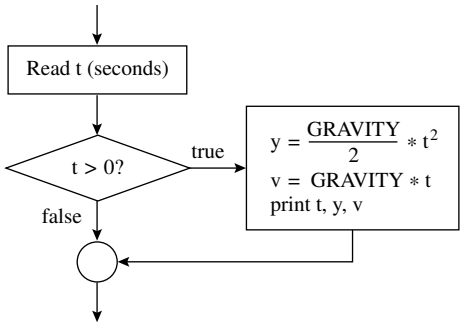
3.7.1 The Simple `if` Statement

In a simple `if` statement, the keyword `if` is followed by an expression in parentheses, called the *condition*, followed by a single statement or a block of statements in curly brackets, called the *true clause*. At run time, the expression is evaluated and its result is interpreted as either true or false. If true, the block of statements following the condition is executed. If false, that block of statements is skipped. Execution continues with the rest of the program following the conditional.

This control pattern is illustrated by the program in Figure 3.10. We use a simple `if` statement here to test whether the input data make sense (the input is a valid time value). If so, we process the data; if not, we do nothing.

Notes on Figure 3.10. Asking a question.

1. The outer box contains the entire simple `if` statement, which consists of
   - The keyword `if`.
   - The condition `(t > 0)`.
   - A block (inner box) containing two assignment statements and three `printf()` statements.
2. As shown in the flow diagram, below, there is only one control path into the `if` unit and one path out. Control branches at the `if` condition but rejoins immediately below the true clause.



3. We use an `if` statement here to test whether the input data makes sense (the time is positive). If so, we follow the `true` path in the diagram and process the data. if not, we follow the `false` path in the diagram and do nothing.

This solves the same problem as Figure 3.7, except that we screen out invalid inputs.

```
/* ------------------------------------------------------------------------
// Determine the velocity and distance traveled by a grapefruit
// with no initial velocity after being dropped from a building.
*/
#include <stdio.h>
#define GRAVITY  9.81    /* gravitational acceleration (m/s^2) */

int main( void )
{
    double t;            /* elapsed time during fall (s) */
    double y;            /* distance of fall (m) */
    double v;            /* final velocity (m/s) */

    printf( " \n\n Welcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " );  /* prompt for the time, in seconds. */
    scanf( "%lg", &t );            /* keyboard input for time */

    if (t > 0) {                   /* check for valid data.  */

        y = .5 * GRAVITY * t * t;  /* calculate distance of the fall */
        v = GRAVITY * t;           /* velocity of grapefruit at impact */
        printf( "    Time of fall = %g seconds \n",  t );
        printf( "    Distance of fall = %g meters \n",  y );
        printf( "    Velocity of the object = %g m/s \n",  v );
    }
    return 0;
}
```

Figure 3.10. Asking a question.

In either case, the next statement executed will be the final `puts()`. A sample output from the `false` path follows. Note that the user gets no answers at all and no explanation of why. This is not a good human–machine interface; it will be improved in the next program example.

```
 Welcome.
 Calculate the height from which a grapefruit fell
 given the number of seconds that it was falling.

 Input seconds:  -1

Gravity has exited with status 0.
```

4. The inner box (the true clause) shows what happens when `t`, the time, is positive: we execute the statements that calculate and print the answers. A sample output from this path might be

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 10
   Time of fall = 10 seconds
   Distance of fall = 490.5 meters
   Velocity of the object = 98.1 m/s

Gravity has exited with status 0.
```

3.7.2 The `if...else` Statement

An `if...else` statement is like a simple `if` statement but more powerful because it lets us specify an alternative block of statements to execute when the condition is false. It consists of

The keyword `if`.

An expression in parentheses, called the *condition*.

A statement or block of statements, called the *true clause*.

The keyword `else`.

A statement or block of statements, called the *false clause*.

At run time, the condition will be evaluated and either the true or the false clause will be executed (the other will be skipped), depending on the result.

The syntactic difference between an `if...else` and a simple `if` statement is that the true clause of an `if...else` statement is followed immediately by the keyword `else`, while the true clause of a simple `if` statement is not. There must not be a semicolon before the `else`.

One common use of the `if...else` statement is to validate input data, it often is necessary to perform more than one test. For example, one might need to test two inputs or test whether an input lies between minimum and maximum acceptable values. To do this, we can use a series of `if...else` statements, as demonstrated in Figure 3.11, which is an extension of the program in Figure 3.10. In this program, we make two tests using two `if...else` statements in a row. We ensure that the input both is positive and does not exceed a reasonable limit, in this case 60 seconds. Using `if...else` in this manner, we can test as many criteria as needed.

The flow diagram corresponding to this new version of the program is given in Figure 3.12. In this diagram, you can see the column of diamond shapes typical of a chain of `if` statements. The `true` clause actions form another sequence to the right of the tests in the diamonds, and the final `false` clause terminates the diamond sequence. All the paths come together in the bubble before the termination message.

This solves the same problem as Figure 3.10, except that we limit valid inputs to be less than 1 minute.

```
#include <stdio.h>
#define GRAVITY  9.81        /* Gravitational acceleration (m/s^2) */

#define MAX     60           /* Upper bound on time of fall. */

int main( void )
{
    sdouble t;     /* elapsed time during fall (s) */
    double y;      /* distance of fall (m) */
    double v;      /* final velocity (m/s) */

    printf( " \n\n Welcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " );   /* prompt for the time, in seconds. */
    scanf( "%lg", &t );             /* keyboard input for time */

    if (t < 0) {                    /* check for negative input */
        printf( " Error: time must be positive.\n\n" );
    }

    else if (t > MAX) {             /* Is time value too big?  */
        printf( " Error: time must be <= %i seconds.\n\n", MAX );
    }

    else {  /* Input is valid; calculate distance and velocity */
        y = .5 * GRAVITY * t * t;  /* calculate distance of the fall */
        v = GRAVITY * t;           /* velocity of grapefruit at impact */
        printf( "    Time of fall = %g seconds \n", t );
        printf( "    Distance of fall = %g meters \n", y );
        printf( "    Velocity of the object = %g m/s \n", v );
    }

    return 0;
}
```

Figure 3.11. Testing the limits.

This is a diagram of the program from Figure 3.11.

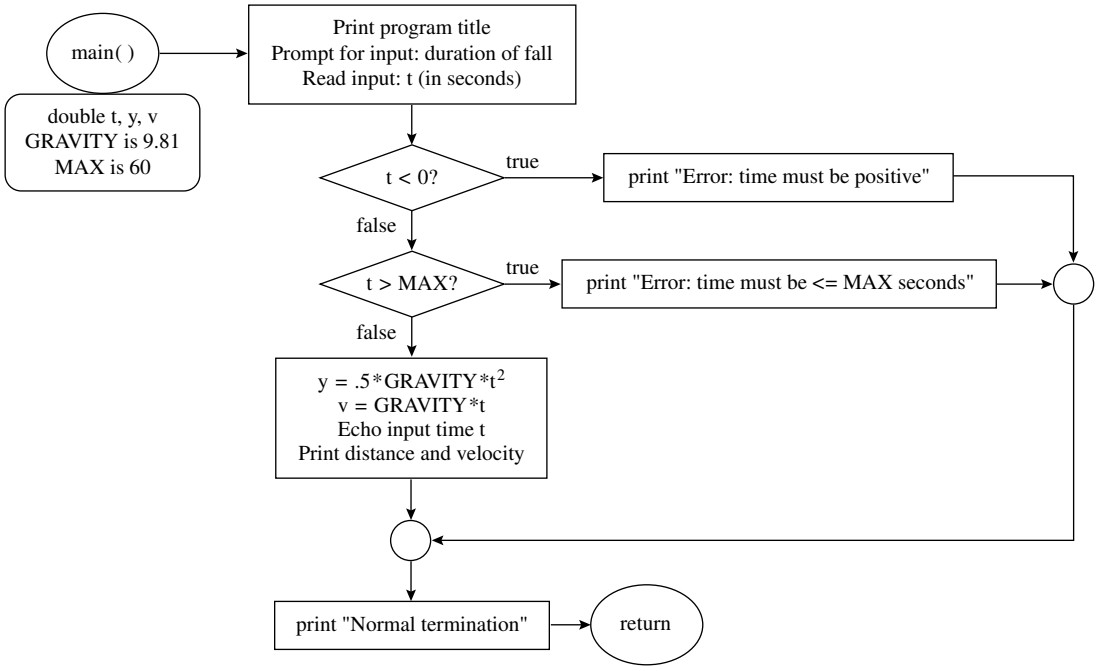


Figure 3.12. Flow diagram for Testing the limits.

Notes on Figure 3.11: Testing the limits.

First box: maximum time.

- This program checks for inputs that are too large as well as those that are negative.
- Since the upper limit is an arbitrary number, it might be necessary to change it in the future. To make such changes easy, we define this limit, `MAX`, at the top of the program and use the symbolic name in the code.

Large outer box: the `if...else` ***chain.***

- Before computations are made, we inspect the data for two errors (first two inner boxes). We skip the remaining tests and the calculations if either error is discovered. This method of error handling avoids doing a computation with meaningless data, such as the negative value in the previous version of the program.
- If no errors are found, we execute the code in the third inner box.

First inner box: negative input values.

- As in the prior example, the input must be positive to correspond to physical reality. The first `if` statement handles this.
- The `true` clause of this `if` statement prints an error comment. Following that, control goes to the `puts()` statement at the bottom of the program, skipping over the `else if` test and the `else` clause.

Second inner box: input values that are too large.

- The elapsed time also must be reasonable. We compare the input value to a maximum allowable value, as defined in the problem specification (60 seconds in this case). The second `if` statement handles this.
- The `true` clause of this `if` statement prints an error comment. Then control skips the `else` clause and goes to the `puts()` at the bottom of the program.
- Here is a sample of the program's error handling:

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 100
Error: time must be <= 60 seconds.
```

Third inner box: the computation and output.

- If the input seems valid, we calculate and print the distance and velocity using the given formulas. Then we echo the input and print the answers.
- Here is the output (without the titles) from a run with valid data:

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 20
   Time of fall = 20 seconds
   Distance of fall = 1962 meters
   Velocity of the object = 196.2 m/s
```

3.7.3 Which `if` Should Be Used?

We looked at three different ways to use `if` statements. These three control patterns are appropriate for different kinds of applications.

The simple `if` statement. The primary uses of the simple `if` statement:

- Processing a subset of the data items. Sometimes a collection of data contains some items that are relevant to the current process and others that are not of interest. We need to process the relevant items and ignore the others. For example, suppose that a file contains data from a whole census but the programmer is interested only in people over the age of 18. The entire file would need to be read but only a subset of the data items would be selected and processed.

- Handling data items that require extra work. Sometimes a few items in a data set require extra processing. For instance, a cash register program must compute the sales tax on taxable items but not on every item. It must test whether each product is taxable and, if so, compute and add the tax. If not, nothing happens.
- Testing for an error condition or goal condition within a loop and leaving the loop if that condition is found. This use of `if` with `break` will be discussed in Chapter 6.

The `if...else` statement. The primary applications of the `if...else` statement:

- Choosing one from a series of alternatives. Sometimes two or more alternative actions are possible and one must be selected. In this case, we often use a series of `if...else` statements to test a series of conditions and select the action corresponding to the first test whose result is `true`.

  For example, consider a program that computes the roots of a quadratic equation: $ax^2 + bx + c = 0$. If a is zero, the equation is linear, and should be solved by the method for linear equations. If both a and b are zero, the equation is degenerate and has no roots. Otherwise, the equation is quadratic. In this case, the program must first calculate the "discriminant" of the equation; namely, $d = b^2 - 4ac$. If this value is positive, the roots of the equation are real numbers; if it is negative, the roots are complex. Both cases are valid but require different processing and different output statements. An `if...else` statement would be used to test the discriminant. Then the `true` clause would execute the code for one kind of roots and the `false` clause would contain the code for the other.

  The program for solving quadratic equations is on the website: look for a link labelled QuadRoots. This multi-brancheddecision pattern will also be incorporated into programs in later chapters; one example is shown in Figure 13.28.

- Normal processing versus nonfatal error handling. In this control pattern, an input value is read and tested for legality. If it would cause a run-time error, an error message is given and the input is not processed in the usual way. This control pattern is illustrated by the outer box and diagram in Figure 3.11.
- Validating a series of inputs. We can use a series of `if...else` statements to test a series of inputs. The `true` clause of each statement would print an error comment, and the `false` clause of the last statement would process the validated data. This decision pattern is incorporated into Figure 3.11.

3.7.4 Options for Syntax and Layout (Optional Topic)

Normally each part of an `if` statement is written on a separate line and all the lines are indented except the `if`, the `else`, and the closing curly bracket. However, a C compiler does not care how you lay out your code.

With brackets the code is spread out:

```
if (age > 18) {
    adults = adults + 1;         /* Count the adults. */
}
else {
    kids = kids + 1;             /* Count the children. */
}
```

Without brackets the code is more compact:

```
if (age > 18)    adults = adults + 1;  /* Count the adults. */
else             kids = kids + 1;      /* Count the children. */
```

Figure 3.13. The `if` statement with and without curly brackets.

Indentation. Inconsistent or missing indentation does not cause any trouble, it simply makes the program hard for a human being to read. Since the compiler determines the structure of the statement solely from the punctuation (semicolons and brackets), an extra or omitted semicolon can completely change the meaning of the statement. Therefore, consistent style is important, both to make programs easier to modify and to help avoid punctuation errors.

The condition. The condition in parentheses after the keyword `if` can be any expression; it does not need to be a comparison. Whatever the expression, it will be evaluated. A zero result will be treated as false and a nonzero result will be treated as true. (Note, therefore, that any number except 0 is considered to be *true*.) The name of a variable, or even a call on an input function, is a legal (and commonly used) kind of condition.

Very local variables. Technically, we could declare variables inside any block, even one that is part of an `if` statement. However, this is not an appropriate style for a simple program.

Curly brackets. A `true` or `false` clause can consist of either a single statement or a block of statements enclosed in curly brackets (braces). If it consists of a single statement, the curly brackets `{` and `}` may be omitted. This can make the code shorter and clearer if the resulting clause fits entirely on the same line as the keyword `if` or `else`. Figure 3.13 illustrates this issue. In it, the same `if` statement is written with and without brackets. The first version is preferred by many experts, even though the brackets are not required. However, others feel that the second version, without brackets, is easier to read because it is written on one line as a single, complete thought. In either case, consistency makes code easier to read; a program becomes visually confusing if one part of an `if` statement has `{` and `}` and the other does not.

3.8 Loops and Repetition

The `if` statement lets us make choices. We test a condition and execute one block of code or another based on the outcome. Another kind of control structure is the loop, which lets us execute a block of statements any number of times (zero or more). Conditionals and loops are two of the three fundamental control structures in a programming language.[16] Loops are important because they allow us to write code once and have a program execute it many times, each time with different data. The more times a computation must be done, the more we gain from writing a loop to do it, rather than doing it by hand or writing the same formula over and over in a program.

C provides three types of loop statements that repeatedly execute a block of code: `while`, `do`, and `for`. The `while` statement is the most basic and is introduced first, in Figure 3.14.[17]

3.8.1 A Counting Loop

Every loop has a set of actions to repeat, called the **loop body**, and a loop test to determine whether to repeat those actions again or end the repetition. In this chapter, we study loops based on counting. In these loops, a variable is set to an initial value, then increased or decreased each time around the loop until it reaches a goal value. We call this variable a **loop counter** because it counts the repetitions and ends the loop when we have repeated it enough times. Our next example, shown in Figure 3.14, introduces a counting loop implemented using a `while` statement. It is a very simple program whose purpose is to make the repetition visible. The corresponding flow diagram used for `while` loops follows the code in the figure.

Notes on Figure 3.14: Countdown.

First box: the loop variable.
A counter is an integer variable used to count some quantity such as the number of repetitions of a loop.

Second box: the loop.

- Before entering a `while` loop, we must initialize the variable that will be used to control it. Here we scan an initial value into `days_left`, the counter variable.
- A `while` statement has three parts, in the following order: the keyword `while`, a condition in parentheses, and a body. The loop body consists of either a single statement or a block of statements enclosed in curly brackets, as shown in the program.
- To execute a `while` loop, first evaluate the condition. If the result is true, the loop body will be executed once and the condition will be retested.
- This sequence of execution is illustrated by the flow diagram at the bottom of Figure 3.14. In the diagram, the `while` loop is an actual closed loop. Control will go around this loop until the condition becomes false, at which point control will pass out of the loop. In this case, control will go to the `puts()`

[16] The third basic control structure, functions, will be introduced in Chapter 5.
[17] The other two loop statements will be presented in Chapter 6.

This program demonstrates the concept of a counting loop. After inputting N, a number of days, it counts downward from N to 0.

```
#include <stdio.h>

int main( void )
{
    int days_left;                      /* The loop counter. */

    printf( " How many days are there until the exam? " );
    scanf( "%i", &days_left);           /* Initialize the loop counter. */

    while (days_left > 0)   {           /* Count downward from 20 to 1. */
        printf( " Days left: %i. Study now.\n", days_left );
        days_left = days_left - 1;      /* Decrement the counter. */
    }

    puts( " This is it!  I hope you are ready." );
    return 0;
}
```

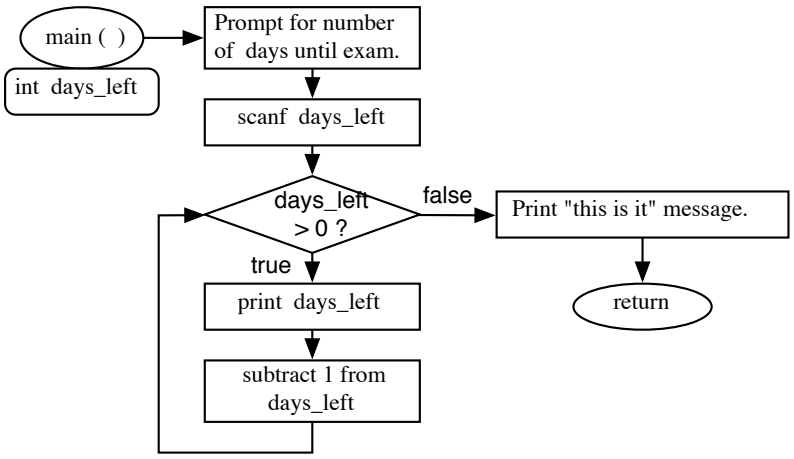


Figure 3.14. Countdown.

statement. The same diamond shape used to represent a test in an `if` statement is used to represent the loop termination test. The presence of a closed cyclic path in the diagram shows that this is a loop, rather than a conditional control statement.

- The statement `days_left = days_left - 1` tells us to use the old value of `days_left` to compute a new one. Read this expression as "`days_left gets days_left minus 1`" or "`days_left becomes days_left minus 1`". (Do not call this assignment operation *equals*, or you are likely to become confused with a test for equality.) In detail, this statement means
  - Fetch the current value of `days_left`.
  - Subtract 1 from it.
  - Store the result back into the variable `days_left`.
- Each time we execute the body of the loop, the value stored in `days_left` will decrease by 1.
- Sample output:

```
How many days are there until the exam? 4
Days left: 4. Study now.
Days left: 3. Study now.
Days left: 2. Study now.
Days left: 1. Study now.
This is it!  I hope you are ready.
```

- Control will leave the loop when the condition becomes false; that is, the first time that `days_left` is tested after its value reaches 0. This will happen before displaying the 0.

Third box: after the loop.
After leaving the loop, control goes to the statements that follow it. The call on `puts()` displays a message at the bottom of the output. The `return` statement returns to the operating system with a code of 0, indicating successful termination.

3.8.2 An Input Data Validation Loop

Interactive computer users are prone to errors; it is very difficult to hit the right keys all the time. It is common to lean on the keyboard or to hold a key down too long and type 991 instead of 91. A crash-proof program must be prepared for bad input, sometimes called *garbage input*. A much-repeated saying is "garbage in, garbage out" (GIGO); that is, the results of a program cannot be meaningful if the input was erroneous or illegal. Although it is impossible to identify all bad input, some kinds can be identified by comparing the data to the range of expected, legal data values. When an illegal data item is identified, the program can quit or ask for re-entry of the data.

A good interactive program uses input prompts to let the user know what specific inputs or what kinds of inputs are acceptable and what to do to end the process. If an input clearly is faulty, the program should (at a minimum) refuse to process it. Preferably, it should explain the error and give the user as many chances as needed to enter good data. One way to do this is the data validation loop.

```
/* --------------- Compute the average speed of your car on a trip. */
#include <stdio.h>
int main( void )
{
    int begin_miles, end_miles   /* Odometer readings.              */
    int miles;                   /* Total miles traveled.           */
    double hours, minutes        /* Duration of trip.               */
    double speed;                /* Average miles per hour for trip.*/
    puts( "\n Miles Per Hour Computation \n" );

    printf( " Odometer reading at beginning of trip: " );
    scanf( "%i", &begin_miles );
    while (begin_miles < 0) {
        printf( " Re-enter; odometer reading must be positive: " );
        scanf( "%i", &begin_miles );
    }

    printf( " Odometer reading at end of trip: " );
    scanf( "%i", &end_miles );
    while (end_miles <= begin_miles) {
        printf( " Re-enter; input must be > previous reading: " );
        scanf( "%i", &end_miles );
    }

    printf( " Duration of trip in hours and minutes: " );
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + (minutes / 60);
    while (hours < 0.0) {
        printf( " Re-enter; hours and minutes must be >= 0.\n" );
        scanf( "%lg%lg", &hours, &minutes );
        hours = hours + (minutes / 60);
    }

    miles = end_miles - begin_miles;
    speed = miles / hours;
    printf( " Average speed was %g \n", speed );

    return 0;
}
```

Figure 3.15. Input validation using `while`.

A **data validation loop** (shown in Figure 3.15) prompts the user to enter a particular data item. It then reads the item and checks whether the input meets criteria for a reasonable or legal value. If so, the loop exits; if not, the user is informed of the error and reprompted. This continues until the user enters an acceptable value.

Notes on Figure 3.15: Input validation using `while`.

First box: A valid odometer reading.

- Before the beginning of a `while` data validation loop, the program must prompt the user for input and read it.
- The `while` loop tests the input. If it is good, the loop body will be skipped; otherwise, control enters the loop. Numbers entered must be small enough to be stored in an integer variable.
- The loop must print an error comment that indicates what is wrong with the input, reprompt the user, and read another input.
- Note that we need two prompts and two `scanf()` statements for this control pattern: one before the loop and another inside the loop.

Second and third boxes: validating the other input.

- Data validation loops all follow a pattern similar to the first loop. The major difference between the first two loops is that the mileage read by the first loop is used in the validation test of the second.
- In the third loop, a computation must be made before the data can be tested. Like the `scanf()` statement, this computation must be written twice, once before entering the loop and again at the end of the loop.

The fourth box: correct data. The box produced this output when correct data were supplied:

```
Miles Per Hour Computation

Odometer reading at beginning of trip:  061234
Odometer reading at end of trip:  061475
Duration of trip in hours and minutes:  4 51
Average speed was 49.6907
```

Faulty data. Here are the results of supplying two kinds of invalid data (greeting and closing comments have been omitted):

```
Odometer reading at beginning of trip: -1
Re-enter; odometer reading must be positive: 061234
Odometer reading at end of trip: 061521
Duration of trip in hours and minutes: 5 28
Average speed was 52.5
```

```
Odometer reading at beginning of trip: 023498
Odometer reading at end of trip: 022222
Re-enter; input must be > previous reading: -32222
Re-enter; input must be > previous reading: 032222
Duration of trip in hours and minutes: 148 43
Average speed was 58.6618
```

3.9 An Application

We have analyzed several small programs; now it is time to show how to start with a problem and synthesize a program to solve it. We will create a program for Joe Smith, the owner of a gas station in Niagara Falls, New York. Joe advertises that his prices are cheaper than those of his competitor, Betty, across the border in Niagara Falls, Ontario. To be sure that his claim is true, he computes the U.S. equivalent of Betty's rates daily. Joe's employee reads Betty's pump prices on the way to work each day, and Joe uses his Internet connection to look up the current exchange rate (U.S. dollars per Canadian dollar). Canadian gas is priced in Canadian dollars per liter. U.S. gas is priced in U.S. dollars per gallon. The conversion is too complicated for Joe to do accurately in his head. Figure 3.16 defines the problem and specifies the scope and properties of the desired solution. We will write a solution step by step. As we go along, we will write a comment for every declaration and any part of the code we defer to a later step.

Step 1. Writing the specification.
Sometimes you will be given a specification, like that in Figure 3.16, and you can begin to plan your strategy based on it. Much of the time, though, you will be given only a general description, as in the previous paragraph. You can fill in many of the details of the specification directly, but you might need to look up in a reference book things like constants or formulas, and you may need to decide the level of accuracy to maintain in your calculations. Until you have completed this step, you will be wasting time by trying to jump into writing the program.

Step 2. Creating a test plan.
Before beginning to write the program, we plan how we will test it. The first test case should be something that can be computed in one's head. We note that one of the simplest computations will occur when the exchange rate is 1.0. Then, if the price per liter is the same as the number of gallons per liter, the price per gallon should be $1.00. We enter this set of numbers as the first line of the test plan in Figure 3.17. We enter the inverse case as the second line of the table: For a price of $1.00 Canadian per liter, the U.S. price should be the same as the conversion factor for liters per gallon.

As a third test case, we enter an unacceptable conversion rate; we expect to see an error comment in response. We also must test the program's response to an invalid gas price, so we add a line with a negative price. This is called "black-box testing": the test values are drawn from the specification or from general knowledge of the kinds of data that often cause trouble. They could be chosen by someone with no knowledge of the code itself. (The code could be inside a black box.)

1. **Problem scope:** Write a short program for Joe that will compute the price per gallon for one grade of gas, in U.S. funds, that is equivalent to Betty's price for that grade of gas.
2. **Inputs:** (1) The current exchange rate, in U.S. dollars per Canadian dollar. This rate varies daily. (2) The Canadian prices per liter for one grade of gasoline.
3. **Constants:** The number of liters in 1 gallon = 3.78544
4. **Formula:**
$$\frac{\$\_US}{gallon} = \frac{\$\_Canadian}{liter} * \frac{liters}{gallon} * \frac{\$\_US}{\$\_Canadian}$$
5. **Output required:** Echo the inputs and print the equivalent U.S. price.
6. **Computational requirements:** All the inputs will be real numbers.
7. **Limitations:** The exchange rate and the price for gas should be positive. If the user enters an incorrect input, an error message should be displayed and another opportunity given to enter correct input.

Figure 3.16. Problem specification: Gas prices.

Rate	Can.$/liter	U.S.$/gallon
1.0	$0.26417	$1.00
1.0	$1.00	$3.78544
−0.001	$1.00	Error
1.0	−$0.87	Error
0.7412	$0.55	$1.543173

Figure 3.17. Test plan: Gas prices.

Finally, we enter a typical conversion rate and a typical price per liter, expecting to see an answer that is consistent with real prices. We use a hand calculator to compute the correct answer. We now have five lines in our test plan, which is enough for a simple program that tests for acceptable inputs.

Step 3. Starting the program.
First, we write the parts that remain the same from program to program, that is, the `#include` command and the first and last lines of `main()` with the greeting message and termination code. The dots in the code represent the unfinished parts of the program that will be filled in by later coding steps.

```
#include <stdio.h>
...                /* Space for #defines. */
int main( void )
{
    ...            /* Space for declarations. */
    puts( "\n Gas Price Conversion Program \n" );
    ...            /* Input statements.  */
    ...            /* Computations.  */
    ...            /* Output statements.  */
    return 0;
}
```

Step 4. Reading the data.
The exchange rate is a number with decimal places, so we declare a `double` variable to store it and put the declaration at the top of `main()`:

```
double US_per_Can;   /* Exchange rate, $_US / $_Canadian */
```

We decide to use a data validation loop to prompt for and read the current exchange rate, so we write down the parts of a `while` validation loop that always are the same, modifying the loop test, prompts, formats, and variable names, as appropriate, for our current application. This code goes into the `main()` program in the second spot marked by the dots.

```
printf( " Enter the exchange rate, $US per $Can: " );
scanf( "%lg", &US_per_Can );

while (US_per_Can < 0.0) {
    printf( " Re-enter; rate must be positive: " );
    scanf( "%lg", &US_per_Can );
}
```

When writing the calls on `scanf()`, remember to use `%lg` in the format for type `double` and put the ampersand before the variable name.

Next, we must read and validate the Canadian gas price. We declare a variable with a name that reminds us that the input is the price in Canadian dollars for a liter. We also remember to declare a variable for the price in U.S. dollars.

```
double C_liter;         /* Canadian dollars per liter */
double D_gallon;        /* US dollars per gallon */
```

Now we write another data validation loop, modifying the loop test, prompts, formats, and variable names, as needed. We write it in the program after the first loop.

```
/* ------------------------------------------------------------------
// Compute the equivalent prices for Canadian gas and U.S. gas
*/
#include <stdio.h>
#define LITR_GAL   3.78544

int main( void )
{
    double US_per_Can;        /* Exchange rate, $_US / $_Canadian */
    double C_liter;           /* Canadian dollars per liter */
    double D_gallon;          /* US dollars per gallon */

    puts( "\n Gas Price Conversion Program \n" );
    printf( " Enter the exchange rate, $US per $Can: " );
    scanf( "%lg", &US_per_Can );
    while (US_per_Can < 0.0 ) {
        printf( " Re-enter; rate must be positive.\n " );
        scanf( "%lg", &US_per_Can );
    }

    printf( " Canadian price per liter: " );
    scanf( "%lg", &C_liter );
    while (C_liter < 0.0 ) {
        printf( " Re-enter; price must be positive: " );
        scanf( "%lg", &C_liter );
    }

    D_gallon = C_liter * LITR_GAL * US_per_Can;
    printf( "\n Canada: $%g  USA: $%g \n", C_liter, D_gallon );

    return 0;
}
```

Figure 3.18. Problem solution: Gas prices.

```
printf( " Canadian price per liter: " );
scanf( "%lg", &C_liter );

while (C_liter < 0.0) {
    printf( " Re-enter; price must be positive: " );
    scanf( "%lg", &C_liter );
}
```

Step 5. Converting the gasoline price.
We defined a constant for liters per gallon at the very top of the program. Now we are ready to compute the price per gallon. Remember that we do not use an = sign or a semicolon in a `#define` command:

```
#define LITR_GAL   3.78544
```

We check the conversion formula given in the specification, making sure that the units do cancel out and leave us with dollars per gallon. Then we write the code for it and a `printf()` statement to print the answers:

```
D_gallon = C_liter * LITR_GAL * US_per_Can;
printf( "\n Canada: $%g  USA: $%g \n", C_liter, D_gallon );
```

Step 6. Testing the completed program.
The finished program is shown in Figure 3.18. Now we run the program and enter the first data set from the test plan. The results are

```
Gas Price Conversion Program

Enter the exchange rate, $US per $Can: 1.0
Canadian price per liter: .26417

Canada: $0.26417   USA: $1
```

Here is a test run using ordinary data (the last line in the test plan):

```
Gas Price Conversion Program

Enter the exchange rate, $US per $Can: .7412
Canadian price per liter: .55


Canada: $0.55  USA: $1.54317
```

Finally, we run the program twice again to test the error handling (the greeting message has been omitted):

```
Enter the exchange rate, $US per $Can:  -0.001
Re-enter; rate must be positive:  1.001
Canadian price per liter:  1.00

Canada: $1  USA: $3.78923
----------------------------------------
Enter the exchange rate, $US per $Can: 1.0
Canadian price per liter: -.087
Re-enter; price must be positive: 2.30

Canada: $2.3  USA: $8.70651
```

3.10 What You Should Remember

3.10.1 Major Concepts

The C language contains many facilities not mentioned yet, and those that have been introduced can be used in many more ways than demonstrated here. It can take years for a programmer to become truly expert in this language. In the face of this complexity, a beginner copes by starting with simple applications, mastering the basic concepts, and learning only the most important details. In this chapter, we have taken a preliminary look at the most basic and important elements of the C language and how they can be combined into a simple but complete program. These are grouped into related areas and summarized.

- **Language elements:**
  The C language contains elements (called commands, operators, and functions) that can be translated to groups of instructions built into a computer's hardware. It provides names or symbols for actions such as *add* (`+`) and *compare* (`==`) but not for complex activities like *solve this equation* or abstract activities such as *think*.
- Overall program structure:
  - An `#include` command is needed at the top of your program to allow your program to use system library functions.
  - A program must have a `main()` function.
  - The `main()` function starts with a series of declarations.
  - A series of statements follows the declarations.
  - A program should start with a statement that prints a greeting and gives instructions for the user.
- Types, objects, and declarations:
  - A program uses the computer's memory to create abstract models of real-world objects such as people, buildings, or numbers. Declarations are used to create and name these objects and may also give them initial values.
  - Declarations are grouped at the top of a program block.
  - Variables and `const` variables are objects; their names are used like nouns in English as the subjects and objects of actions.
  - An object has a name, a location, and a value. The compiler assigns the location for the object. We can give it a value by initializing it in the declaration or by assigning a value to it later. An object that has not been given a value is said to be *uninitialized* or to contain *garbage*.
  - Every object has a type. Types are like adjectives in English: The type of an object describes its properties and how it may be used. The three basic data types seen so far are `int`, `char`, and `double`.
  - A constant object must be initialized in the declaration and its value cannot be changed.
  - A `#define` command can be written at the top of a program to create a symbolic name for a literal constant.

- Simple statements:
  - The programmer combines operations and functions into a series of sentence-like statements that describe the actions to be carried out and the order in which they must happen.
  - Each statement tells the computer what to do next and what variables and constants to use in the process.
  - When a program is executed, the instructions in the program are run in order, from beginning to end. That sequential order can be modified by control instructions that allow for choices and repetition of blocks of statements.
  - The `scanf()` statements perform input. They let a human being communicate with a computer program. If a program requires the user to enter data, the input statement should be preceded by an output statement that displays a user prompt.
  - The `puts()` and `printf()` statements perform output. These statements let a computer program communicate with a human being.
  - An assignment statement can perform a calculation and store the result in a variable so that it can be used later. In general, calculations follow the basic rules of mathematics.
- Compound statements:
  - Statements can be grouped into blocks with curly brackets.
  - The simple `if` statement is a conditional control statement. It has a condition and one block of code that is executed when the condition is true.
  - The `if...else` statement is a conditional control statement that has a condition and two blocks of code, a true clause and a false clause. When an `if...else` statement is executed, the condition is tested first and this determines which block of code is executed.
  - The `while` statement is a looping control statement. It has a condition and one block of code. The condition is tested first, and if the condition is true, the block is executed. Then the condition is retested. Execution and testing are repeated as long as the condition remains true.
  - The counting loops seen so far require that a counter variable be initialized prior to the loop and updated in some manner each time through the loop.

3.10.2 Programming Style

- A comment should follow each variable declaration to explain the purpose of the variable.
- Input data should be checked for validity: Garbage input causes garbage output.
- When writing your own programs, it often helps to model your work after a sample program that does a similar task. This makes it easier to find a combination of input, calculation, output, and control statements that are consistent with each other and work gracefully together.
- Indentation is important for readability. A programmer should adopt a meaningful indentation style and follow it consistently.

- Line up the words `if` and `else` with the `}` brackets that close each clause in the same column. Indent all the statements in each clause. This assures a neat appearance and helps a reader find the end of the clause.[18]
- If the clause to be executed in one part of an `if` statement is short and the other part is long, put the short clause first. This helps the reader see the whole picture easily. For example, suppose the program must test an input value to determine whether it is in the legal range. If it is legal, several statements will be used to process it. If it is illegal, the program will print an error comment and terminate execution, which takes only two lines of code. This program should be written with the error clause first (immediately following the `if` test) and the normal processing following the `else`.
- Where possible, avoid writing the same statement twice. This makes the program clearer and easier to debug. For example, do not put the same statement in both clauses of an `if` statement; put it before the `if` or after it.
- If both the `true` clause and the `false` clause are single statements, the entire `if` statement can be written on two lines without curly brackets (`{` and `}`) to begin and end the clauses. If either clause is longer than one line, both clauses should be written with curly brackets.

3.10.3 Sticky Points and Common Errors

- When you compile a program and get compile-time error comments, look at the first one first. One small error early in the program can produce dozens of error comments; fixing that single error often will make many comments go away.
- If you misspell a word, it becomes a different word in the eyes of the compiler. This is the first thing to check when you do not understand a compile-time or link-time error comment.
- An extra semicolon after the condition in an `if` or `while` statement will end the statement, and the code that should be within the `if` or `while` statement will be outside it. For example, suppose the `while` statement in the countdown program (Figure 3.14) were written incorrectly:

```
m = ITERATIONS;
while (m > 0);
{   printf( " %i.  \n", m );
    m = m - 1;
}
```

  The programmer will expect to see 20 lines printed on the page, with the first line numbered 20 and the last numbered 1. Instead, the semicolon ends the loop, which therefore has no body at all. The update line `m = m - 1;` is outside the loop and cannot be reached. The program will become an infinite loop because nothing *within the loop* will decrement the loop variable, `m`.

[18] This layout scheme is advocated by *Recommended C Style and Coding Standards*, guidelines published in 1994 by experts at Bell Laboratories.

- A missing semicolon will not be discovered until the compiler begins working on the next line. It will tell you that there is an error, but give the wrong line number. Always check the previous line if you get a puzzling error comment about syntax.
- Quotation marks, curly brackets, and comment-begin and -end marks come in pairs. If the second mark of a pair is omitted, the compiler will interpret all of the program up to the next closing mark as part of the comment or quote. It will produce odd and unpredictable error comments.
- If the output seems to make no sense, the first thing to check is whether the declared type of each variable on the output list matches the conversion specification used to print it. An error anywhere in an output format can affect everything after that on the line. If this does not correct the problem, add more diagnostic printouts to your program to display every data item calculated or read as input. If the input values are correct and the calculated values are wrong, check your formulas for precedence errors and check your function calls for errors.
- If the input values are wrong when you echo them, make sure you have ampersands before the names of the variables in the `scanf()` statement. Check also for type errors in the conversion specifiers in the format.

3.10.4 New and Revisited Vocabulary

These important terms and concepts were presented in this chapter:

lexical analysis	precision	assignment (=)
preprocessor command	modifier	expression
program block	identifier	operators (+, −, ∗, /)
declaration	local name	precedence
statement	undefined value	sequential execution
keyword	garbage	control statement
literal constant	object diagram	condition
symbolic constant	string	loop test
constant variable	prompt	loop body
variable	stream	loop counter
data value	buffer	data validation loop
data type	format	flow diagram
initializer	conversion specifier	black-box testing

The following C keywords and functions were presented in this chapter:

`#include`	`while` loop	`double`
`#define`	`\n` (newline character)	`int`
`main()`	`&` (address of)	`char`
`return` statement	`<stdio.h>`	`const`
`{...}` (block)	`stdin`	`scanf()`
`/*...*/` (comment)	`stdout`	`puts()`
`if...else` statement	`stderr`	`printf()`

3.10.5 Where to Find More Information

- The complete set of standard C keywords is given in Appendix C.
- A full discussion of types `int` and `double` is given in Chapter 7 where we deal with representation, overflow, and the imprecise nature of floating point numbers.
- Additional simple and compound types are introduced in Chapters 10 through 14.
- Storage classes are introduced and used as follows:
  - `const`: Introduced here and used hereafter.
  - `volatile`: Introduced and used in Chapter 13.
  - `auto`: Introduced in Chapter 19, used throughout.
  - `extern`: Introduced in Chapter 19, used in Chapter 20.
  - `static`: Introduced in Chapter 19, used in Chapter 20.
  - `register`: Introduced in Chapter 19, not used in a program.
- Many more operators and the details of operator precedence and associativity are given in Chapter 4.
- Program design, with pseudocode and top-down development, is revisited in Chapters 5, 6, and 9.
- Operators associated with pointers are given in Chapter 11, those for bit manipulation are given in Chapter 15, those for accessing a `struct` in chapter 13, and the conditional operator is explained in Appendix D.
- The third basic control structure, functions, will be introduced in Chapter 5
- The other two loop statements will be presented in Chapter 6.
- As other types of data are introduced in Chapters 7 through 11, more details about formats will be presented.
- The complexities of streams will be explored in Chapter 14.

3.11 Exercises

3.11.1 Self-Test Exercises

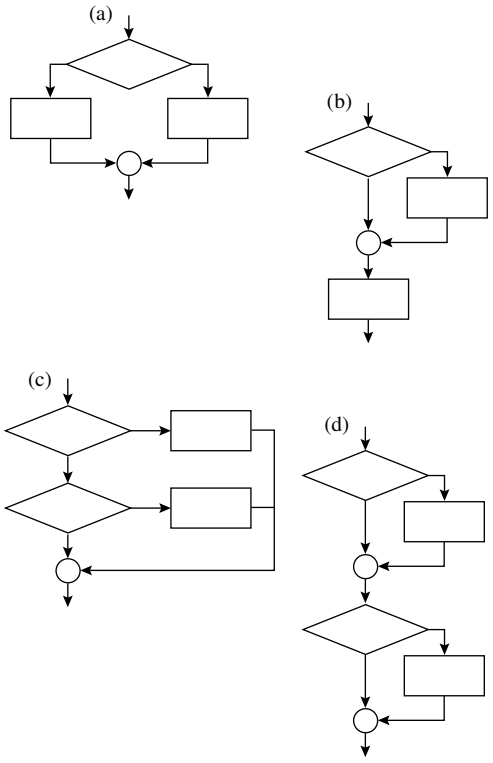
1. Four code fragments and four flow diagrams follow. Note that each diagram has two action boxes and one or two question boxes. All four represent distinct patterns of logic. Match each code fragment to the corresponding flow diagram and show how the code fits into the boxes.

(1)
```
if (radius < 0) {
volume = 0;
}
if (height < 0) {
volume = 0;
```

(2)
```
if (radius < 0) {
volume = 0;
}
else if (height < 0) {
volume = 0;
}
```

(3)
```
if (t < 1) {
v = t;
}
else {
v = 1;
}
```

(4)
```
if (rad > 100)
puts( "Too big" );
area = PI * rad * rad;
```



2. What is wrong with each of the following declarations?

   (a) `int d; a = 5;`
   (b) `doubel h;`
   (c) `int h = 2.5;`
   (d) `const double g;`
   (e) `character middle_initial;`
   (f) `double h = 2.0 * x;`
   (g) `integer k = 0;`
   (h) `char gender = "M";`

3. What conversion specifier do you use in an output format for an `int` variable? For a `double` variable? For a character?
4. Find the error in each of the following declarations.

   (a) `integer count;`
   (b) `real weight ;`
   (c) `int k; count;`
   (d) `character gender;`
   (e) `duble age;`

5. What happens when you omit the ampersand (address of) before a variable name in a `scanf()` statement? To find out, delete an ampersand in one of the sample programs. Try to compile and run the resulting program.
6. What happens when you type an ampersand before a variable name in a `printf()` statement? Try it.
7. What happens when you type a comment-begin mark but forget to type (or mistype) the matching comment-end mark? To find out, delete a comment-end mark in one of the sample programs and try to compile the result.
8. What happens when you type a semicolon after the closing parenthesis in a simple `if` statement? Try it.
9. What is wrong with each of the following `if` statements? They are supposed to identify and print out the middle value of three `double` values, `x`, `y`, and `z`.

   (a)
```
if (x < y < z) printf( "y=%g", y );
else if (y < x < z) printf( "x=%g", x );
else printf( "z=%g", z );
```

   (b)
```
if (x < y)
    if (y < z) printf( "y=%g", y );
    if (z < y) printf( "z=%g", z );
else
    if (x < z) printf( "x=%g", x );
    if (z < x) printf( "z=%g", z );
```

   (c)
```
if (x > y)
{   if (x < z);
    printf( "x=%g", x );
    else printf( "z=%g", z );
}
else
{   if (y < z);
    printf( "y=%g", y );
    else printf( "z=%g", z );
}
```

3.11.2 Using Pencil and Paper

1. What does your compiler do when you misspell a keyword such as `while` or `else`? What happens when you misspell a function name such as `main()` or `scanf()`? Try it.
2. What happens when you type a semicolon after the closing parenthesis in a `while` statement? Try it.
3. What conversion specifier do you use in an input format for an `int` variable? For a `double` variable? For a character variable?
4. Find the error in each of the following preprocessor commands.

   (a) `#include <stdio>`
   (b) `#define NORMAL = 98.6`
   (c) `#include stdio.h`
   (d) `#define TOP 1,000`
   (e) `#define LOOPS 10;`
   (f) `#include <studio.h>`

5. Given the declarations on the first three lines that follow, find the error in each of the following statements:

```
int age, count;
double price, weight;
char gender;
```

   (a) `scanf( "%g", &price );`
   (b) `scanf( "%c", &gender );`
   (c) `scanf( "%d", &weight );`
   (d) `printf( "%i", &count );`
   (e) `printf( "%lg", price );`
   (f) `printf( "%c", gendre );`

6. Draw a flow diagram of the following program and use your diagram to trace its execution. What is the output?

```
#include <stdio.h>
int main( void )
{
    int k, m;
    k = 0;
    m = 1;
    while (k <= 3) {
        k = k + 1;
        m = m * 3;
    }
    printf( "k = %i m = %i \n", k, m );
    return 0;
}
```

7. In the following program, circle each error and show how to correct it:

```
#include (stdio.h)
int main (void)
{
    integer k;
    double x, y, z
    printf( Enter an integer: );
    scanf( "%i", k );
    printf( "Enter a double: );
    scanf( "%g", &X );
    printf( "Enter two integers: );
    scanf( "%lg", &y, &z );
    printf( "k= %i X= %g \n", &k, &x );
    printf( "y= %i z= %g \n" y, z );
}
```

8. Draw a flow diagram for each set of statements, then trace their execution and show the output. Use these values for the variables: `w=3, x=1, y=2, z=0`.

   (a) `if (x < y) z=1; if (x > w) z=2;`
   `printf(" %i\n", z);`

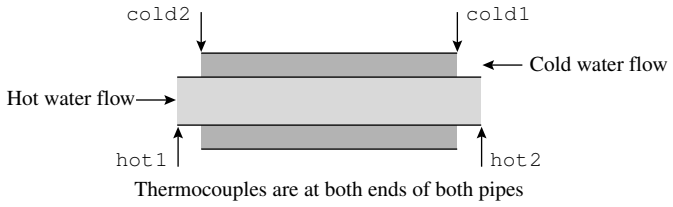
   (b) `if (x < y) z=1; else if (x > w) z=2;`
   `printf(" %i\n", z);`

   (c) `if (w < x) z=1; else if (w > y) z=2; else z=3;`
   `printf(" %i \n", z);`

3.11.3 Using the Computer

1. Heat transfer.

   Complete the following program for heat transfer that already has been designed and partially coded. The program analyzes the results from a heat transfer experiment in which hot water flows through a pipe and cold water flows in the opposite direction through a surrounding, concentric pipe. Thermocouples are placed at the beginning and end of both pipes to measure the temperature of the water coming in and going out, as diagrammed here.



This is a standard engineering calculation. To calculate the heat transfer, we need the average hot temperature, the average cold temperature, and the change in temperature from input to output for both hot and cold pipes.

Your job is to start with the incomplete program in Figure 3.19, fill in the boxes according to the instructions that follow in each box, and make the program work. Use the following test plan.

Test Plan	**Data**				**Answers**			
	Inlets		**Outlets**		**Means**		**Differences**	
Test Objective	**Hot**	**Cold**	**Hot**	**Cold**	**Hot**	**Cold**	**Hot**	**Cold**
Easy to calculate	100	0	50	50	75	25	−50	50
Another legal input	120	35	100	50	110	42.5	−20	15

(a) Get a copy of the file `fex_heat.c`, which contains the partially completed program in Figure 3.19. Write program statements to implement the actions described in each of the comment boxes. Delete the existing comments and add informative ones of your own.

(b) Compile your code and test it using the test plan given above. Prepare proper documentation for the project including the source code, output from running the program on the test data, and hand calculations that prove the output is correct.

2. Temperature Conversion.

Complete a program for temperature conversion that has been designed and partially coded. The program will read in a temperature in degrees Fahrenheit, convert it into the equivalent temperature in degrees Celsius, and display this result. Your job is to start with the incomplete program in Figure 3.20 fill in the boxes according to the instructions given below in each box, and make the program work.

(a) Make an appropriate test plan for your program following the layout scheme below.

Test objective	Input Fahrenheit	Output Celsius
Easy-to-calculate input		
Another legal input		
Minimum legal input		
Out-of-range input		

(b) Start coding with a copy of the file `fex_temp.c`, which contains the partially completed program in Figure 3.20. Following the instructions given in each comment box, write program statements to implement the actions described. Delete the existing comments and add informative ones of your own. Compile and run your code, testing it according to the first line of your plan. Check the answer. If it is correct, print it out and go on to the rest of your test plan. If it is incorrect, fix it and test it again until it is correct. When the answers are correct, print out the program and its output on these tests, and hand them in with the test plan.

```
/* This program will compute heat characteristics of a concentric-pipe */
/* heat exchanger, including mean hot and cold temperatures, and the   */
/* differences in temperatures of the ends of the pipes.               */

#include <stdio.h>
int main( void )
{
    double hot1;        /* hot inlet temperature */
    double hot2;        /* hot outlet temperature */
    double cold1;       /* cold inlet temperature */
    double cold2;       /* cold outlet temperature */
    double mean_hot;    /* average of hot temperatures */
    double mean_cold;   /* average of cold temperatures */
    double dthot;       /* difference in hot temperatures */
    double dtcold;      /* difference in cold temperatures */

    puts( "Heat Transfer Experiment" );

    Prompt user to enter four inlet and outlet temperatures.
    Use a separate prompt for each temperature you read.
    Use scanf() to read each and store in the appropriate variable.

    Calculate the mean of the hot temperatures (their sum divided by 2)
        and the mean of the cold temperatures.  Save each value in an
        appropriate variable.
    Calculate dthot (difference in hot temperatures = the hot outlet
        temperature minus the hot inlet temperature) and dtcold.
        Again save these in the appropriate variables.
    Put comments on these lines describing the calculations.

    Write printf() statements to echo the four input temperatures.
    Write printf() statements to print the four calculated numbers.
    Label each output clearly, so we can tell what it means.
    Use \n and spaces in your formats to make the output easy to read.

    return 0;
}
```

Figure 3.19. Sketch for the heat transfer program.

```
/* ------------------------------------------------------------------- */
/* This program will convert temperatures from degrees Fahrenheit into  */
/* degrees Celsius.  Input temperature must be above absolute zero*/
#include <stdio.h>

Define a constant for absolute zero in Fahrenheit (-459.67).

int main( void )
{
    double fahr;        /* Temperature in Fahrenheit degrees */
    double cels;        /* Temperature in Celsius degrees    */
    puts( "\n Temperature Conversion Program" );

    Prompt the user to enter a temperature in degrees Fahrenheit.
    Use scanf() to read it and store it in the appropriate variable.
    Write a printf() statement that will echo the input.


    Test whether the input temperature is less than absolute zero.
    If so, print an error comment.
    If not, calculate the temperature in degrees Celsius according
    to the formula C = 5.0/9.0 * (F - 32.0) and print the answer.
    Label the output clearly.
    Use newlines and spaces in your formats to make the output
    easy to read.

    return 0;
}
```

Figure 3.20. Sketch for the temperature conversion program.

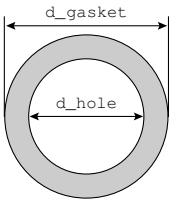
3. Sales Tax.

   The text website contains a complete program to solve the problem described below. However, the program will not compile and may have logical errors. Download the program and debug it, according to your specification. Hand in the debugged program and its output.

   Given the cost of a purchase, and a character 'T' for taxable or 'N' for non-taxable, compute the sales tax and total price. Let the tax rate be 6%.

(a) **Problem scope:** Write a program that will calculate the surface area of a ring gasket.

(b) **Inputs:** The outer diameter of the gasket, `d_gasket`, and the diameter of the hole in the center of the gasket, `d_hole`. Both diameters should be given in centimeters.



Formula:

$$\texttt{area} = \frac{\pi \times (\texttt{d\_gasket}^2 - \texttt{d\_hole}^2)}{4}$$

(c) **Limitations:** The value of `d_gasket` must be nonnegative. The ratio of `d_hole` to `d_gasket` must be greater than 0.3 and less than 0.9.

(d) **Constant:** $\pi = 3.14159265$

(e) **Output required:** The surface area of the gasket. (Echo the inputs also.)

(f) **Computational requirements:** All the inputs will be real numbers.

Figure 3.21. Problem specification: Gasket area.

4. Gasket area.

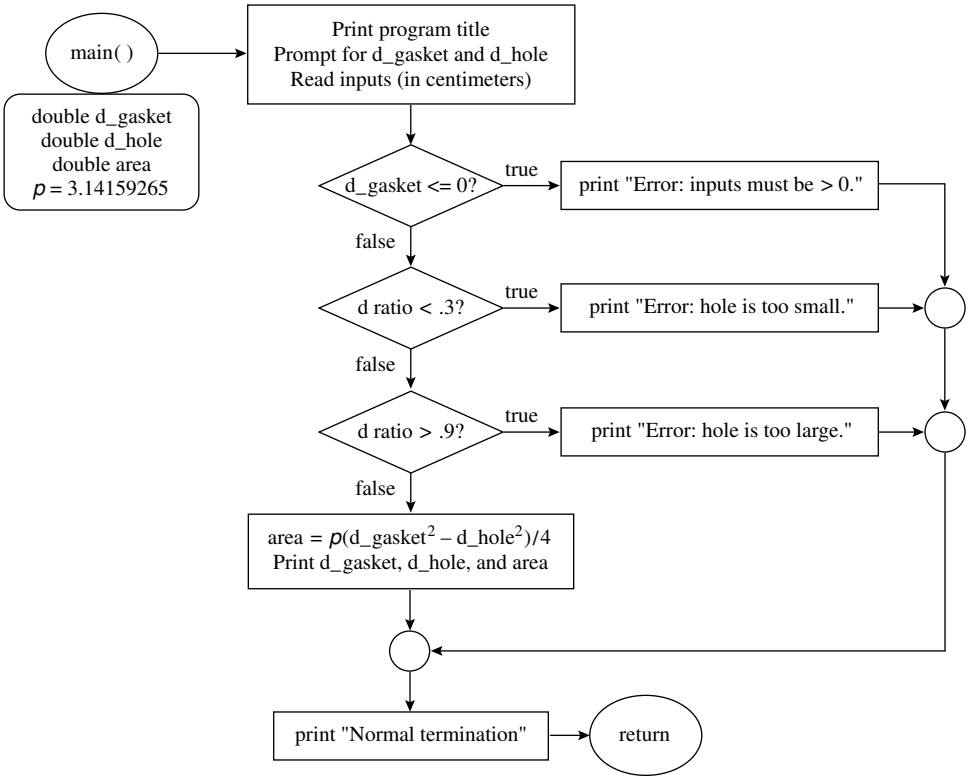
   A specification for a program that computes the area of a ring gasket is given in Figure 3.21. From this, the flow diagram in Figure 3.22 was constructed.

   (a) Develop a test plan for this program based on the problem specification in Figure 3.21.
   (b) Write a program based on the algorithm in Figure 3.22.
   (c) Make sure all of your prompts and output labels are clear and easy to read.
   (d) Test your program using the test plan you devised.
   (e) Turn in the source code and output results from your test.

5. Skyscraper.

   The text website contains a complete program to solve the problem described below. However, the specification and test plan are missing and the program will not compile and may have logical errors. Write a specification and test plan, then download the program and debug it, according to your specification. Hand in your work plus the debugged program and its output.

   Assume that the ground floor of a skyscraper has 12-foot ceilings, while other floors of the building have 8-foot ceilings. Also, the thickness in between every floor is 3 feet. On top of the building is a 20-foot flagpole. If the building has N stories altogether, and N is given as an input to your program, calculate the height of a blinking red light at the top of the flagpole and print out this height.



[.7]

Figure 3.22. Flow diagram for the gasket area program.

6. Weight conversion.

   Write a program that will read a weight in pounds and convert it to grams. Print both the original weight and the converted value. There are 454 grams in a pound. Design and carry out a test plan for this program.

7. Distance conversion.

   Convert a distance from miles to kilometers. There are 5,280 feet per mile, 12 inches per foot, 2.54 centimeters per inch, and 100,000 centimeters per kilometer.

8. More Temperature Conversion. Revise the specification, test plan and program from problem 4 so that the user can enter a temperature in either Celsius or Fahrenheit, and the program will convert it to the other system. The inputs should be a temperature and a character, "C" for Celsius or "F" for Fahrenheit. Use an `if` statement to test which letter was entered, then perform the appropriate conversion. Echo the

input and label the output properly. The formula for conversion from Celsius to Fahrenheit is

$$Fahrenheit = Celsius \times \frac{9}{5} + 32$$

9. Meaningful change.

   Write a program that will input the cost of an item from the user and output the amount of change due if the customer pays for the purchase with a $20 bill. What kinds of problems might this program have? (Think about unexpected inputs.) Design a test plan to detect these problems. Use an `if` statement to validate your data to prevent meaningless outputs.

10. A snow job.

    Your snow blower clears a swath 2 feet wide. Given the length and width of your rectangular driveway, calculate how many feet you will need to walk to clear away all the snow. Be sure to include the steps you take when you turn the snow blower around. Write a program that contains validation loops for the two input dimensions, echoes the input, and prints out the calculated distance.

11. Plusses and minuses.

    Your program will be used as part of a quality control system in a factory that makes metal rods of precise lengths. The current batch of rods is supposed to be 10 cm long. An automated measuring device measures the length of each rod as it comes off the production line. These measurements are automatically sent to a computer as input. Some rods are slightly shorter and some slightly longer than the target length. Your program must read the measurements (use `scanf()`). If the rod is too short, add it to a `short_total`; otherwise, add it to a `long_total`. Also, count it with either the `short_counter` or the `long_counter`. After each batch of 20 rods, print out your totals, counters, and the average length of the rods in the batch.