PART III

Basic Data Types

Chapter 7

Using Numeric Types

Two kinds of number representations, integer and floating point, are supported by C. The various **integer types** in C provide exact representations of the mathematical concept of "integer" but can represent values in only a limited range. The **floating-point types** in C are used to represent the mathematical type "real." They can represent real numbers over a very large range of magnitudes, but each number generally is an approximation, using a limited number of decimal places of precision.

In this chapter, we define and explain the integer and floating-point data types built into C and show how to write their literal forms and I/O formats. We discuss the range of values that can be stored in each type, how to perform reliable arithmetic computations with these values, what happens when a number is converted (or cast) from one type to another, and how to choose the proper data type for a problem.

We would like to think of numbers as integer values, not as patterns of bits in memory. This is possible most of the time when working with C because the language lets us name the numbers and compute with them symbolically. Details such as the length (in bytes) of the number and the arrangement of bits in those bytes can be ignored most of the time. However, inside the computer, the numbers *are* just bit patterns. This becomes evident when conditions such as integer overflow occur and a "correct" formula produces a wrong and meaningless answer. It also is evident when there is a mismatch between a conversion specifier in a format and the data to be written out. This section explains *how* such errors happen so that *when* they happen in your programs, you will understand what occurred.

7.1 Number Systems and Number Representation

Numbers are written using positional base notation; each digit in a number has a value equal to that digit times a place value, which is a power (positive or negative) of the base value. For example, the *decimal* (base 10) place values are the powers of 10. From the decimal point going left, these are $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1{,}000$, and so forth. From the decimal point going right, these are $10^{-1} = 0.1$, $10^{-2} = 0.01$, $10^{-3} = 0.001$, $10^{-4} = 0.0001$, and so forth. Figure 7.1 shows the place values for *binary* (base 2) and *hexadecimal* (base 16); the chart shows those places that are relevant for a short integer. Just as base-10 notation (decimal) uses 10 digit values to represent numbers, hexadecimal uses 16 digit values. The first 10 digits are 0–9; the last six are the letters A–F. [1]

7.2 Integer Types

To accommodate the widest variety of applications and computer hardware, C integers come in two varieties and up to three sizes. We refer to all of these types, collectively, as the *integer types*. However, more than

[1] The interested reader can refer to Appendix E for algorithms for converting numbers from one base to another.Figure **??** shows the decimal values of the 16 hexadecimal digits; Figure **??** shows some equivalent values in decimal and hexadecimal.

Place values for base 16 (hexadecimal) are shown on the left; base-2 (binary) place values are on the right. Each hexadecimal digit occupies the same memory space as four binary bits because $2^4 = 16$.

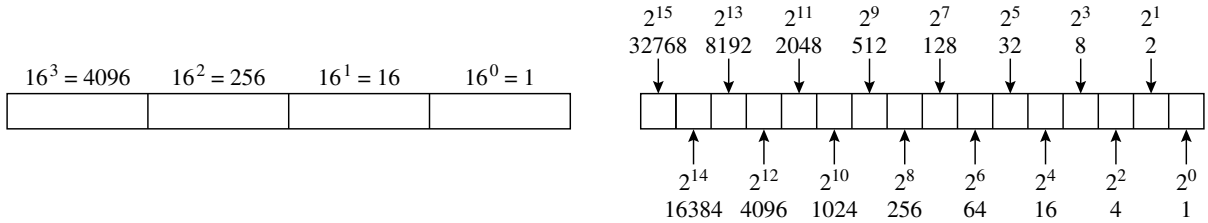


Figure 7.1. Place values.

six different names are used for these types, and many of the names can be written in more than one form. In addition, some type names have different meanings on different systems. If this sounds confusing, it is.

The full type name of an integer contains a sign specifier, a length specifier, and the keyword `int`. However, there are shorter versions of the names of all these types. Figure 7.2 lists the commonly used name, then the full name, and finally other variants.

A C programmer needs to know what the basic types are, how to write the names of the types needed, and how to input and output values of these types. He or she must also know which types are portable (this means that the type name always means more or less the same thing) and which types are not (because the meaning depends on the hardware).

7.2.1 Signed and Unsigned Integers

In C, integers come in varying lengths and in two underlying varieties: `signed` and `unsigned`. The difference is the interpretation of the leftmost bit in the number's representation. For signed numbers, this bit indicates the sign of the value. For unsigned numbers, it is an ordinary magnitude bit.

Why does C bother with two kinds of integers? FORTRAN, Pascal, and Java have only signed numbers. For most purposes, signed numbers are fine. Some applications, though, seem more natural using unsigned numbers. Examples include applications where the actual pattern of bits is important, negative values are meaningless or will not occur, or one needs the extra positive range of the values.

On paper or in a computer, all the bits in an unsigned number represent part of the number itself. If the number has n bits, then the leftmost bit has a place value of 2^{n-1}. Not so with a signed number because one bit must be used to represent the sign. The usual way that we represent a signed decimal number on paper is by putting a positive or negative sign in front of a value of a given magnitude. This representation, called *sign and magnitude*, was used in early computers and still is used today for floating-point numbers. However, a different representation, called *two's complement*, is used for signed integers in most modern computers. In the two's complement representation of a 2-byte signed integer, the leftmost bit position has a place value of -32768. A 1 in this position signifies a negative number. All the rest of the bit positions have positive place values, but the total is negative.

Common Name	Full Name	Other Acceptable Names	
`int`	`signed int`	`signed`	
`long`	`signed long int`	`long int`	`signed long`
`short`	`signed short int`	`short int`	`signed short`
`unsigned`	`unsigned int`		
`unsigned long`	`unsigned long int`		
`unsigned short`	`unsigned short int`		

Figure 7.2. Names for integer types.

The binary representations of several signed and unsigned integers follow. Several of these values turn up frequently during debugging, so it is useful to be able to recognize them.

$2^{15} = 32768$	$2^{14} = 16384$	$2^{13} = 8192$	$2^{12} = 4096$	$2^{11} = 2048$	$2^{10} = 1024$	$2^{9} = 512$	$2^{8} = 256$	$2^{7} = 128$	$2^{6} = 64$	$2^{5} = 32$	$2^{4} = 16$	$2^{3} = 8$	$2^{2} = 4$	$2^{1} = 2$	$2^{0} = 1$	Interpreted as a signed `short int` high-order bit = `-32768`	Interpreted as an unsigned `short int` high-order bit = `32768`
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	`32767`	`32767`
0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0	`10000`	`10000`
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	`1`	`1`
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	`0`	`0`
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	`-32768 + 32767 = -1`	`+32768 + 32767 = 65535`
1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0	`-32768 + 22768 = -10000`	`+32768 + 22768 = 55536`
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	`-32768 + 0 = -32768`	`+32768 + 0 = 32768`
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	`-32768 + 1 = -32767`	`+32768 + 1 = 32769`

Figure 7.3. Two's complement representation of integers.

Since the difference in meaning between signed and unsigned numbers depends only on the first bit, a number with a 0 in the high-order position has the same interpretation whether it is signed or unsigned. However, a number with a 1 in the high-order position is negative when interpreted as a signed integer and a very large positive number when interpreted as unsigned. Several examples of positive and negative binary two's complement representations are shown in Figure 7.3. Every unsigned 2-byte number larger than 32,767 has the same bit pattern as some negative-signed 2-byte number. For example, in most PCs, the bit patterns of 32,768 (unsigned) and −32,768 (signed) are identical.

Negation. To negate a number in two's complement, invert (complement) all of the bits and add 1 to the result. For example, the 16-bit binary representation of +27 is 00000000 00011011, so we find the representation of −27 by complementing these bits, 11111111 11100100, and then adding 1: 11111111 11100101.

You can tell whether a signed two's complement number is positive or negative by looking at the high-order bit; if it is 1, the number is negative. To find the magnitude of a positive integer, simply add up the place values that correspond to 1 bits. To find the magnitude of a negative integer, complement the bits and add 1. For example, suppose we are given the binary number 11111111 11010010. It is negative because it starts with a 1 bit. To find the magnitude we complement these bits, 00000000 00101101; add 1 using binary addition[2] , and convert to its decimal form, 00000000 00101110 $= 32+8+4+2 = 46$. So the original number is −46.

7.2.2 Short and long integers.

Integers come in two[3] lengths: `short` and `long`. On most modern machines, short integers occupy 2 bytes of memory and long integers use 4 bytes. The resulting value **representation ranges** are shown in Figure 7.4. As you read this list, keep the following facts in mind:

- The ranges of values shown in the table are the minimum required by the ISO C standard.
- On many machines the smallest negative value actually is −32,768 for `short int` and −2,147,483,648 for `long int`.
- Unsigned numbers are explained more fully in Section 15.1.

[2]Adding binary numbers is similar to adding decimal values, except that a carry is generated when the sum for a bit position is 2 or greater, rather than 10 or greater, as with decimal numbers. The carry values are represented in binary and may carry over into more than one position as they can in decimal addition.

[3]Or three lengths, if you count type `char` (discussed in Chapter 8), which actually is a very short integer type.

Data Type	Names of Constant Limits	Range
`int`	`INT_MIN...INT_MAX`	Same as either `long` or `short`
`short int`	`SHRT_MIN...SHRT_MAX`	$-32{,}767 \ldots 32{,}767$
`long int`	`LONG_MIN...LONG_MAX`	$-2{,}147{,}483{,}647 \ldots 2{,}147{,}483{,}647$
`unsigned int`	`0...UINT_MAX`	Same as `unsigned long` or `short`
`unsigned short`	`0...USHRT_MAX`	$0 \ldots 65{,}535$
`unsigned long`	`0...ULONG_MAX`	$0 \ldots 4{,}294{,}967{,}295$

Figure 7.4. ISO C integer representations.

- On PCs, `int` usually is the same as `short int`. On workstations and larger machines, it is the same as `long int`.
- The constants `INT_MIN`, `INT_MAX`, and the like are defined in every C implementation in the header file `limits.h`. This header file is required by the C standard, but its contents can be different from one installation to the next. It lists all of the hardware-dependent system parameters that relate to integer data types, including the largest and smallest values of each data type supported by the local system.

The type `int` is tricky. It is defined by the C standard as "not longer than `long` and not shorter than `short`." The intention is that `int` should be the same as either `long` or `short`, whichever is handled more efficiently by the hardware. Therefore, many C systems on Intel 80x86 machines implement type `int` as `short`.[4] We refer to this as the **2-byte `int` model**. Larger machines implement `int` as `long`, which we refer to as the **4-byte `int` model**.

The potential changes in the limits of an `int`, shown in Figure 7.4, can make writing portable code a nightmare for the inexperienced person. Therefore, it might seem a good idea to avoid type `int` altogether and use only `short` and `long`. However, this is impractical, because the integer functions in the C libraries are written to use `int` arguments and return `int` results. The responsible programmer simply must be aware of the situation, make no assumptions if possible, and use `short` and `long` when it is important.

Integer literals. An **integer literal** constant does not contain a sign or a decimal point. If a number is preceded by a `-` sign or a `+` sign, the sign is interpreted as a unary operator, not as a part of the number. When you write a literal, you may add a type specifier, `L`, `U`, or `UL` on the end to indicate that you need a `long`, an `unsigned`, or an `unsigned long` value, respectively. (This letter is not the same as the conversion specifier of an I/O format.) If you do not include such a type code, the compiler will choose between `int`, `unsigned`, `long`, and `unsigned long`, whichever is the shortest representation that has a range large enough for your number. Figure 7.5 shows examples of various types of integer literals. Note that no commas are allowed in any of the literals.

7.3 Floating-Point Types in C

In traditional **scientific notation**, a real number, N, is represented by a signed **mantissa**, m, multiplied by a base, b, raised to some signed **exponent**, x; that is,

$$N = \pm m \times b^{\pm x}$$

For example, we might write 1.4142×10^{-2}. A floating-point number is represented similarly inside the computer by a sign bit, a mantissa, and a signed exponent.

7.3.1 Representation of Real Numbers

Each of the components of a real number is stored in some binary format. The IEEE (Institute for Electrical and Electronic Engineers) established a standard for floating-point representation and arithmetic that has

[4] However, the Gnu C compiler running under the Linux operating system on the same machine implements type `int` as `long`.

In this table, we assume that the type `int` is the same length as `short`, and that the maximum representable `int` is 32,767.

Literal	Type	Reason for Type
0	`int`	Number is less than 32,767
200	`int`	Number is less than 32,767
255U	`unsigned`	It uses the `U` code
255L	`long`	It uses the `L` code
255UL	`unsigned long`	It uses the `UL` code
32767	`int`	Largest possible 2-byte `int`
32767L	`long`	It uses the `L` code
32768	`unsigned`	Too large for a 2-byte `signed int`
65536	`long`	Too large for a 2-byte `unsigned int`
3000000000	`unsigned long`	3 billion, too large for `long`
6000000000	Compile-time error	6 billion, too large for any integer type

Figure 7.5. Integer literals in base 10.

been carefully designed to give predictable results with as much precision as possible. Most scientists doing serious numerical computations use systems that implement the IEEE standard. Figure 7.6 shows the way that the number -10 is represented according to the IEEE standard for four-byte real numbers. This representation uses 32 bits divided into three fields to represent a real value. The base, $b = 2$, is not represented explicitly; it is built into the computer's floating-point hardware.

The mantissa is represented using the sign and magnitude format. The sign of the mantissa (which is also the sign of the number) is encoded in a single bit, bit 31, in a manner similar to that used for integers: a 0 for positive numbers or a 1 for negative values. The magnitude of the mantissa is separated from the sign, as indicated in Figure 7.6, and occupies the right end of the number.

After every calculation, the mantissa of the result is *normalized*; that means it is returned to the form $1.XX\ldots X$, where each X represents a one or a zero. The mantissa is shifted right or left so that exactly one nonzero bit remains to the left of the decimal point. The exponent is adjusted appropriately; that is, incremented (or decremented) by 1 each time the mantissa is shifted left (or right) by one bit position. In this way, the significant information "floats" to the left end of the mantissa. A number always is normalized before it is stored in a memory variable. Since all mantissas follow this rule, the 1 and the decimal point do not need to be stored explicitly; they are built into the hardware instead. Thus the 23 bits in the mantissa of an IEEE real number are used to store the 23 X bits. For example, in Figure 7.6, the value 0.25, or 0.01 in binary, is stored in the mantissa bits and the leading 1 is recreated by the hardware when the value is brought from memory into a register.

The number -10 is shown in binary IEEE format for type `float`.

	sign	exponent		mantissa
$-10.0 = -1.25 \times 2^3 =$	1	1000001 0	1.	0100000 00000000 00000000
	31	30 ... 23		22 ... 0

- The sign bit is 1, indicating a negative number
- The exponent 10000010 is represented in excess 127 notation, which means that we must subtract 127 from the binary number shown to get the true exponent: $130 - 127 = 3$
- The mantissa is 1.01000..., which means $1 + 1/4 = 1.25$

Figure 7.6. Binary representation of reals.

These are the minimum value ranges for the IEEE floating-point types. The names given in this table are the ones defined by the C standard.

Type Name	Digits of Precision	Name of C Constant	Minimum Value Range Required by IEEE Standard
`float`	6	±`FLT_MIN`...±`FLT_MAX`	±1.175E−38 ... ±3.402E+38
`double`	15	±`DBL_MIN`...±`DBL_MAX`	±2.225E−308 ... ±1.797E+308

Figure 7.7. IEEE floating-point types.

When we add or subtract real numbers on paper, the first step is to line up the decimal points of the numbers. A computer using floating-point arithmetic must start with a corresponding operation, denormalization. To add or subtract two numbers with different exponents, the bits in the mantissa of the operand with the smaller exponent must be **denormalized**: The mantissa bits are shifted rightward and the exponent is increased by 1 for each shifted bit position. The shifting process ends when the exponent equals the exponent of the larger operand. We call this number representation *floating point* because the computer hardware automatically "floats" the mantissa to the appropriate position for each addition or subtraction operation.

The precision of a floating-point number is the number of digits that are mathematically correct. Precision directly depends on the number of bits used to store the mantissa and the amount of error that may accumulate due to round-off during computations. Typically a calculation is performed using a few additional bits beyond the lengths of the original operands. The final result then must be rounded off to return it to the length of the operands involved. The different floating-point types use different numbers of bits in the mantissa to achieve different levels of precision. Typical limits are given in Figure 7.7.

The exponent is represented in bits 23 ... 30, which are between the sign and the mantissa. Each time the mantissa is shifted right or left, the exponent is adjusted to preserve the value of the number. A shift of one bit position causes the exponent to be increased or decreased by 1. In the IEEE standard real format, the exponent is stored in *excess 127 notation*. Here, the entire 8 bits are treated as a positive value, then the excess value, 127, is subtracted from the 8-bit value to determine the true exponent. In Figure 7.6, $127 + 3 = 130$, which is the value stored in the eight exponent bits. While this format may be complicated to understand, it has advantages in developing hardware to do quick comparisons and calculations with real numbers.

C has a great variety of integer types. Fortunately, the set of floating-point types is not so extensive. There are only three: `float`, `double`, and `long double`. We use the terms *float* or *floating point* to refer to a variable of any of these types. There are no unsigned floating-point types. The only real difference among the types is the number of bits used in the representation, which directly affects the range and number of possible digits of precision. Figure 7.7 shows the minimum properties of the `float` and `double` types defined by the IEEE standard,[5] which many C implementations support.

An IEEE `float` needs at least 8 bits for the exponent and 24 for the mantissa. Because of this limited number of bits, many values cannot be stored as type `float` with adequate precision for common numerical computations. An IEEE `double` uses 11 bits for the exponent and 53 for the mantissa, increasing both the range of exponents and the precision. This is a minimum standard; the actual range of real values supported by hardware may be greater than this standard requires, due to slight variations in the way the hardware and software treat floating-point numbers.[6] The actual precision and exponent range for a local implementation can be found in the local version of the file `float.h`. The third floating-point type, `long double`, is new in ISO C and identical to `double` on most systems. The standards do not define a minimum length for it, although 12 bytes are used on some systems. Since `double` is sufficient for most calculations and few machines have the specialized hardware to support `long double`, the longer type is rarely used. All the computations in this book will use the standard `float` and `double` types.

Floating-point literals. In traditional mathematical notation, we write real numbers in one of two ways. The simplest notation is a series of digits containing a decimal point, like 672.01. The other notation, called

[5]The ISO C standard is less demanding than the IEEE standard.

[6]In our C system, both the range and the precision are slightly larger than the IEEE standard requires.

Literal	Type	Size in Common Implementations
3.14	double	8 bytes
1.05792e+05	double	8 bytes
65536E-4f	float	4 bytes
1.01F	float	4 bytes
.02l	long double	8, 10, or 12 bytes
171.L	long double	8, 10, or 12 bytes

Figure 7.8. Floating-point literal examples

base-10 scientific notation, uses a base-10 exponent in conjunction with a mantissa: we write 672.01 as 6.7201×10^2.

In C, real literals also can be written in either decimal or a variant of scientific notation: we write `6.7201E+02` instead of 6.7201×10^2. A numeric literal that contains either a decimal point or an exponent is interpreted as one of the floating-point types; the default type is `double`. (A literal number that has no decimal point and no exponent is an integer.) There are several rules for writing literal constants of floating-point types:

1. You may write the number in everyday decimal notation: Any number with a decimal point is a floating-point literal. The number may start or end with the decimal point; for example, `1.0`, `0.1`, `1.` `.1416`, or `120.1`,.

2. You may use scientific notation. When you do so, write a mantissa part followed by an exponent part; for example, `4.50E+6`. The mantissa part follows the rules for decimal notation, except that it is not necessary to write a decimal point. Examples of legal mantissas are `3.1416`, `341.0`, `.123`, and `89`.

   The exponent part has a letter followed by an optional sign and then a number.

   - The letter can be `E` or `e`.
   - The sign can be `+`, `-`, or it can be omitted (in which case, `+` is assumed).
   - The exponent number is an integer of one to three digits in the proper ranges, as given in Figure 7.7. If your system does not follow the standard, the ranges may be different.

3. Following the literal value a **floating-point type-specifier** may be used, just as for integers. (This letter is not the same as the conversion specifier of an I/O format.) The specifiers `f` and `F` designate a `float`, while `l` and `L` designate a `long double`. If a floating-point literal has no type specifier, it is a `double`.

Figure 7.8 shows some examples of floating-point literals and the actual number of bytes used to store them.

7.4 Reading and Writing Numbers

Two factors must be considered when choosing a format for reading a number: the type of the variable in which the value will be stored and the way the value appears in the input. Similarly, when printing a number, its type and the desired output format must be considered. In previous examples, we used only a few of the many possible formats for numeric input and output, which we now discuss.

7.4.1 Integer Input

Each type of value requires a different **I/O conversion specifier** in the format string. An integer conversion specifier starts with a `%` sign, followed by an optional field-width specifier (output only), and a code for the

Context	Conversion	Meaning and Use
`scanf()`	`%d`	Read a base-10 (decimal) signed integer (traditional C and ISO C)
	`%i`	Read a decimal or hexadecimal signed integer (ISO C only)
	`%u`	Read a decimal unsigned integer (traditional C and ISO C)
	`%hi` or `%hd` or `%hu`	Use a leading `h` for `short int`
	`%li` or `%ld` or `%lu`	Use a leading `l` for `long int`
`printf()`	`%d`	Print a signed integer in base 10
	`%i`	Same as `%d` for output
	`%u`	Print an unsigned integer in base 10
	`%hi` or `%hd` or `%hu`	Use a leading `h` for `short int`
	`%li` or `%ld` or `%lu`	Use a leading `l` for `long int`

Note: The code for short integers is `h` instead of `s`, because `s` is used for strings (see Chapter 12).

Figure 7.9. Integer conversion specifications.

type of value to be read or written. Figure 7.9 summarizes the options available for signed[7] integers. The use of `%hi` and `%li` will be illustrated by the program in Figure 7.28.

The `%i` code is new in ISO C,[8] supplementing the traditional `%d`. With `%i`, input numbers can be entered in either decimal or hexadecimal notation (see Chapter 15), whereas `%d` works only for decimal (base-10) numbers. A representational error[9] will occur if an input value has more digits than the input variable can store. The faulty input will be accepted, but only a portion of it will be stored in the variable. The result is a meaningless number that will look like garbage when it is printed. When a program's output clearly is wrong, it always is a good idea to echo the input on which it was based. Sometimes, this uncovers an inappropriate input format or a variable too short to store the required range of values.

7.4.2 Integer Output

For output, the `%d` and `%i` conversion codes can be used interchangeably. We use `%i` in this text because it is more mnemonic for "integer" and therefore less confusing for beginners.

When designing the output of a program, the most important things to consider are that the information be printed correctly and labeled clearly. Sometimes, however, spacing and alignment are important factors in making the output clear and readable. We can control these factors by writing a **field-width** specification (an integer) in the output format between the `%` and the conversion code (`i`, `li`, or `hi`). For example, `%10i` means that the output value is an `int` and the printed form should fill 10 columns, while `%4hi` means that the output value is a `short int` and the printed form should fill 4 columns. If the given width is wider than necessary, spaces will be inserted to the left of the printed value. To print the number at the left edge of the field, a minus sign is written between the `%` and the field width, as in `%-10i`. The remainder of the field is filled with blanks.[10] If the width is omitted from a conversion specifier or if the given width is too small, C will use as many columns as are required to contain the information and no spaces will be inserted on either end (therefore, the effective default field width is 1). Using a field-width specifier allows us to make neat columns of numbers, as will be illustrated by the program in Figure 7.28.

Positive or negative? If an unsigned integer has a bit in the high-order position and we try to print it in a `%i` format instead of a `%u` format, the result will have a negative sign and the magnitude may even be small. Unfortunately, most programmers eventually make this careless mistake. The short program in Figure 7.10 illustrates what can happen when an inappropriate conversion specifier is used. Two unsigned numbers are printed, first properly, then with a signed format.

[7] Input and output for unsigned numbers will be discussed in Chapter 15, which deals with hexadecimal notation and bit-level computation on unsigned numbers.

[8] Older compilers may not support `%i`.

[9] Other sources of representational error will be discussed in Section 7.6.

[10] A format string may specify a nonblank character to use as a filler.

```
#include <stdio.h>

int main( void )
{
    short unsigned hui = 33000;
    long unsigned lui= 4200000000;

    printf( "%hu is a short unsigned int\n", hui);
    printf( "    printed in hi it is: %hi\n\n", hui );

    printf( "%lu is a long unsigned int\n", lui);
    printf( "    printed in li it is: %hi\n\n", lui );
}
```

Figure 7.10. Incorrect conversions produce garbabe output.

```
33000 is a short unsigned int
    printed in hi it is: -32536

4200000000 is a long unsigned int
    printed in li it is: -5632
```

In both cases, it is easy to see that the output is garbage because it has a negative sign. However, using a different constant, the output is even more confusing; it is still wrong but there is no negative sign to give us a clue.

```
3000000000 is a long unsigned int
    printed in li it is: 24064
```

7.4.3 Floating-Point Input

In a floating-point literal constant (Figure 7.8), a letter (called the **type specifier**) is written on the end to tell the compiler whether to translate the constant as a `float`, a `double`, or a `long double` value. An input format must contain this same information, so that `scanf()` will know how many bytes to use when storing the input value. In a `scanf()` format, the input conversion specifier for type `float` is `%g`; for `double`, it is `%lg`; and for `long double`, it is `%Lg`. Figure 7.11 summarizes the basic conversion specifiers for real numbers. For input, all the basic specifiers `%g`, `%f`, and `%e` have the same meaning and can be used interchangeably, although the current convention is to use `%g`.

The actual input value may be of large or small magnitude, contain a decimal point or not, and be any number of decimal digits long. The number will be converted to a floating-point value using the number of bytes appropriate for the local `C` translator. (Commonly, this is 4 bytes for `%g`, 8 bytes for `%lg`, and 8 bytes or more for `%Lg`.) However, sometimes, the number stored in the variable is not exactly the same as the input given. This happens whenever the input, when converted to binary floating-point notation, has more digits of precision (possibly infinitely repeating) than the variable can store. In this case, only the most significant digits are retained, giving the closest possible approximation.

7.4.4 Floating-Point Output

Output formats for real numbers are more complex than those of integers because they have to control not only the field width but also the form of the output and the number of significant digits printed. There are three basic choices of conversions: `%f`, `%e`, and `%g`. The `%f` conversion prints the value in ordinary decimal form, the `%e` conversion prints it in scientific notation, and the `%g` conversion tries to choose the "best" way to present the number. This may be similar to `%f`, `%e`, or even `%i`, depending on the size of the number relative to the specified field width and precision. Whatever precision is specified and whatever conversion code is used, floating-point numbers will be rounded to the last position printed.[11]

[11] Note that this is different from the rule for converting a real number to an integer, which will be discussed later in this chapter. During type conversion, the number is truncated, not rounded.

Context	Conversion	Meaning and Usage
`scanf()`	`%g`, `%f`, or `%e`	Read a number and store in a `float` variable
	`%lg`, `%lf`, or `%le`	Read a number and store in a `double` variable
	`%Lg`, `%Lf`, or `%Le`	Read a number and store in a `long double` variable
`printf()`	`%f`	Print a `float` or a `double` in decimal format
	`%e`	Print a `float` or a `double` in exponential format
	`%g`	Print a `float` or a `double` in general format

Figure 7.11. Basic floating-point conversion specifications.

All three kinds of conversion specifiers (`%f`, `%e`, and `%g`) can include two additional specifications: the total field width (as described for an integer) and a precision specifier. These two numbers are written between the `%` and the letter, separated by a period, as in `%10.3f`. In addition, either the total field width or the precision specifier can be used alone, as in `%10f` or `%.3f`. The default precision is 6, and the default field width is 1 (as it is for integers). If the field is wider than necessary, the unused portion will be filled with blank spaces.

The `%f` and `%e` conversions. For the `%f` and `%e` conversions, the precision specifier is the number of digits that will be printed after the decimal point. When using the `%f` conversion, numbers are printed in ordinary decimal notation. For example, `%10.3f` means a field 10 spaces wide, with a decimal point in the seventh place, followed by three digits. An example is given in Figure 7.12.

In a `%e` specification, the mantissa is *normalized* so that it has exactly one decimal digit before the decimal point, and the last four or five columns of the output field are occupied by an exponent (an example is given in Figure 7.13). For a specification such as `%.3e`, one digit is printed before the decimal point and three are printed after it, so a total of four significant digits will be printed.

The `%g` conversion tries to be smart. The result of a `%g` conversion can look like an integer or the result of either a `%f` or `%e` conversion. The precision specifier determines the maximum number of significant digits that will be printed. The `printf()` function first converts the binary numeric value to decimal form, then it uses the following rules to decide which output format to use. Here, assume that, for a number N, with D digits before the decimal point, the precision specifier is S.

- If $D == S$, the value will be rounded to the nearest integer and printed as an integer (an example is given in Figure 7.14).
- If $D > S$, the number will be printed in exponential format, with one digit before the decimal point and $S - 1$ digits after it (an example is given in Figure 7.15).

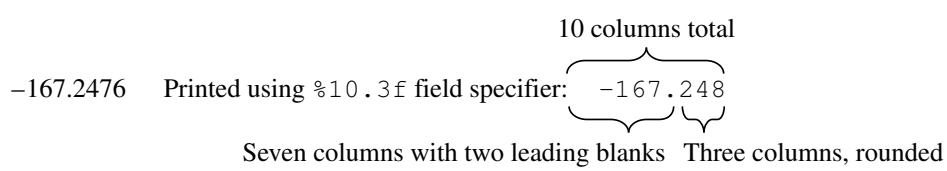


Figure 7.12. The `%f` output conversion.

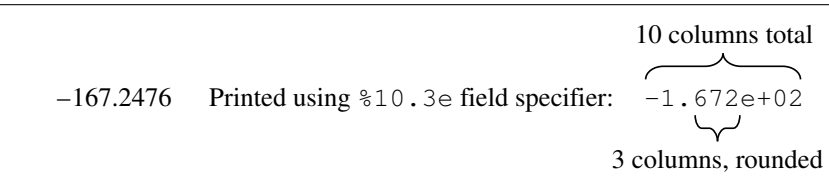


Figure 7.13. The `%e` output conversion.

–167.2476 Printed using `%10.3g` field specifier: `-167`

10 columns total with three digits of precision.

Figure 7.14. Sometimes %g output looks like an integer.

- If $D < S$ and the exponent is less than -4, the number will be printed in exponential format, with one digit before the decimal point and $S - 1$ digits after it (an example is given in Figure 7.16).
- If $D < S$ and the exponent is -4 or greater, the number will be printed in decimal format with D digits before the decimal point and $S - D$ digits after it (an example is given in Figure 7.17).

In all four cases, the precision specifier determines the *total* number of significant digits printed, including any nonzero digits before the decimal point. Therefore, `%.3g` will print one less significant digit than `%.3e`, which always prints three digits after the decimal point. Also, `%.3g` may print several digits fewer than `%.3f`.

Finally, the `%g` conversion strips off any trailing zeros or decimal point that the other two formats will print. Therefore, the number of places printed after the decimal point is irregular. This leads to an important rule: The `%g` conversion is not appropriate for printing tables. Usually `%f` is used for tables, unless the values are of very large magnitude.

7.4.5 One Number may Appear in Many Ways

Figure 7.18 shows how the input values of 32.1786594, 2.3, and 12345678 might look if they were read into a `float` variable, and then printed in a variety of formats. Each input was read using `scanf()` with the `%g` specifier. The actual converted value stored in a `float` variable is shown beneath that. Output of these values was produced by `printf()`, using the conversion formats shown.

Notes on Figure 7.18. Output conversion specifiers. When examining the various results, note the following details:

First line. This line shows the values entered from the keyboard. In the first column, we input more than the six or seven significant digits that a `float` variable can store; the result is that the last digits of the internal value (on the next line) are only an approximation of the input.

Second line. This line shows the actual values stored in three `float` variables. Due to the limited number of bits, the first two values cannot be represented exactly inside the computer. This may not seem surprising for the first value, since a `float` has only six digits of precision, but even the value of 2.3 is not represented

10 columns total

–1672.476 Printed using `%10.3g` field specifier: `-1.67e+03`

Three digits of precision, rounded

Figure 7.15. Sometimes %g looks like %e.

10 columns total

-.000016724 printed using %10.3g field specifier: -1.67e-05

3 digits of precision, rounded

Figure 7.16. For tiny numbers, %g looks like %e.

−167.2476	Printed using `%10g` field specifier: (The default precision = 6)	`-167.248`
	Seven columns with two leading blanks	Three columns, rounded

Figure 7.17. Sometimes `%g` looks like `%f`.

exactly. This is because, just as there are repeating fractions in the decimal system (like 1/7), when certain decimal values are converted into their binary representation, the result is a repeating binary fraction. The stored internal value of 2.3 is the result of **truncating** this repeating bit sequence. By chance, even though it is more than six digits long, the third value, 12345678, could be represented exactly. Even though the stated level of precision is six decimal digits, longer numbers sometimes can be represented exactly, while some shorter ones can only be approximated.

Main portion of table.

1. All output values are rounded to the last place that is printed.
2. An output too wide for the field is printed anyway, it just overflows its boundary, as in some of the values in the last column.
3. The **default output precision is six decimal places**, so you get six digits after the decimal point with `%f` and `%e` unless you ask for more or fewer. With `%g`, you get a maximum of six significant digits.
4. The `%g` conversion specifier works similar to `%f` for numbers that are not too large or too small. The primary differences are that trailing zeros and trailing decimal points will not be printed and that the precision specifies significant digits, not actual digits after the decimal point. For very large and very small numbers, `%g` works almost like `%e` except that one fewer significant digit will be printed. Therefore, the number 12345678 printed in `%.3e` becomes 1.235e+07, but printed in `%.3g`, it is 1.23e+07.

The programs in Figures 7.21 and 7.28 illustrate some ways in which format specifiers can be used to achieve desired output results. To get a good sense of what the C language does with different floating-point types, experiment with various input values and changing the formats in these programs.

Alternate output conversion specifiers. Some compilers will accept `%lg` (or `%lf` or `%le`) in a `printf()` format for type `double`. However, `%g` is correct according to the standard and it is poor style to get in the habit of using nonstandard features. The standard is clear on this issue. All `float` values are converted to type `double` when they are passed to the standard library functions, including `printf()`. By the time `printf()` receives the `float` value, it has become a `double`. So `%g` is used with `printf()` when printing both `double` and `float` values.

Input at keyboard	%g	32.1786594	2.3	12345678
Internal bit value		32.17865753173828125	2.299999952316284179687 5	12345678
Output using	%f	32.178658	2.300000	12345678.000000
	%e	3.217866e+01	2.300000e+00	1.234568e+07
	%g	32.1787	2.3	1.23457e+07
	%.3f	32.179	2.300	12345678.000
	%.3e	3.218e+01	2.300e+00	1.235e+07
	%.3g	32.2	2.3	1.23e+07
	%10.3f	32.179	2.300	12345678.000
	%-10.3f	32.179	2.300	12345678.000

Figure 7.18. Output conversion specifiers.

However, this is not true for `scanf()`. According to the ISO C standard, you *must* use `%g` for `float`, `%lg` for `double`, and `%Lg` for `long double` in input formats.

7.5 Mixing Types in Computations

Since we have introduced both the integer and floating-point data types that are typically used in calculations, it is time to discuss how to use them effectively and convert values from one data type to another.

7.5.1 Basic Type Conversions

Two basic types of data conversion can occur, a length conversion and a representation conversion. The **length conversion** occurs between two values of the same data category; that is, between two integers or between two reals. These are **safe conversions** if they lengthen the data representation and thereby do not introduce any representational error. For example, any number that can be represented as a `float` can be represented with exactly the same precision using the `double` type. **Unsafe conversions** may happen if the data representation is shortened.

A **representation conversion** involves switching between two categories, from integer to real or real to integer. Even in systems where a `float` value and an integer value have the same number of bits, their patterns are very different and incompatible. The computer hardware cannot add a `float` to a `long`—one of them must first be converted to the other representation. Depending on the direction of the conversion, it might be classified as safe or not. Therefore, let us examine the basic properties of type conversions more closely.

Safe conversions. Converting from a "short" version of a data type to a "long" version is considered to be a safe operation. All the bits stored in the shorter version still can be stored in the longer form, with extra padding in the appropriate positions. Converting from a longer form to a shorter one may or may not be safe. For integers, if the magnitude of the value in the longer form is within the representation range of the shorter one, everything is fine. For real numbers, not only must the magnitude be within the proper range, but the number of significant digits in the mantissa must be small enough as well.

Converting from a `float` to a `double` is safe but the effects can be misleading. The value is lengthened, but it does not increase in precision. A `float` has six or seven decimal places of precision, and the precision of a lengthened value will be the same; the extra bits in the `double` representation will be meaningless zeros. For example, one-tenth is an infinitely repeating fraction in binary. We can store only a finite portion of this value in a `double` and even less in a `float`. Consider the code in Figure 7.19. We initialize both `f` and `d` to 0.1. In both cases, the number actually stored in the variable is only an approximation to 0.1. However, the approximation stored in `d` is more precise. It is accurate up to the 17th place after the decimal point, while `f` is accurate only up to the 8th place. When the value of `f` is converted to type `double` and stored in `x`, it still is accurate only to eight places. The precision does not increase because there is no opportunity to recompute the value and restore the lost bits.

Converting from an integer type to a floating-point type usually is safe, in the sense that most integers can be represented exactly as `floats` and all can be represented exactly as `doubles`. The opposite is not true; most floating-point values cannot be represented exactly as integers.

Unsafe conversions. When a value of one type is converted to a shorter type, the number being converted can be too large to fit into the smaller type. We call this condition **representation error**. This is handled quite differently for integers and floating-point numbers.

When a large integer is converted to a smaller integer type, only the *least* significant bits are transferred, resulting in garbage. Converting a negative signed number to an unsigned type is logically invalid. Similarly, it is a logical error to convert an unsigned integer to a signed type not long enough to contain the value. The programmer must be careful to avoid any type conversions of this nature, because the C system gives little or no help with detecting the error. Some compilers will display a warning message when potentially

```
#include <stdio.h>
int main( void )
{
    float  f = 0.1;  // Precision limited to about 7 decimal places.
    double d = 0.1;  // Precision limited to about 15 decimal places.
    double x = f;    // Converted float; same precision as original value.

    printf( "0.1 as a float  = %.17f\n", f );
    printf( "0.1 as a double = %.17f\n", d );
    printf( "0.1 converted from float to double = %.17f\n", x );
}
```

Output:

```
0.1 as a float  = 0.10000000149011612
0.1 as a double = 0.10000000000000001
0.1 converted from float to double = 0.10000000149011612
```

Figure 7.19. Converting a `float` to a `double`.

unsafe conversions are discovered, others will not. At run time, if such an error happens, no C system will stop and give an error comment.

The shortening action that happens when a `double` value is converted to a `float` has two potential problems. First, it truncates the mantissa, discarding up to nine decimal digits of precision. Second, if the exponent of the value is too large for type `float`, the number cannot be converted at all. In this case, the C standard does not say what will happen; the result is "undefined" and you cannot expect the C system to warn you that this problem has occurred. If it happens in a program, you might observe that some of the output looks like garbage or that certain values are displayed as `Infinity` or `NaN` (not a number).[12]

When a floating-point number is converted to an integer type, the fractional part is lost. To be precise, it is *truncated*, not rounded; the fractional part is discarded, even if it is .999999. To maintain the maximum possible accuracy in calculations, C avoids converting floating-point values to integer types and does so only in four situations, which are listed and explained in the next section. Remember that, in these cases, rounding does not happen, so the floating-point value `1.999999999` will be converted to `1`, not to `2`.

A last source of unsafe conversions is the use of incorrect conversion specifiers in a `scanf()` statement. For example, using a `%g` (for `float`) in a `scanf()` format when you need `%lg` (for `double`) will not be detected by most compilers as an error. On these systems, the faulty program will compile with no warnings but will not run correctly. It will read the data from the keyboard and convert it into the representation indicated by the format. The corresponding bit pattern, whether the right length or not, will be stored into the waiting memory location without further modification. This will put the wrong information into the variable and inevitably produce garbage results.

7.5.2 Type Casts and Coercions

All the different conversions just mentioned can be invoked explicitly by the programmer by writing a type cast. They might also be produced by the compiler because of a type-mismatch in the program code; we call this **type coercion**.

Type casts. A **type cast** is an explicit operation that performs a type conversion. A cast is written by enclosing a type name in parentheses and writing this unit before either a variable name or an expression. Any type name can be made into a cast and used as an operator in an expression. Technically, a type cast is a unary operator with precedence lower than all other unary operators but higher than all the binary operators. When applied to an operand, it tells the system to convert the operand to the named type, if possible. Sometimes this adjusts the length of the operand, sometimes it alters the representation, and sometimes it just changes the type labeling. Examples of casts are shown in Figure 7.20.

[12] More is said about these error conditions in Section 7.6.

These examples of casts use the following declarations:

```
float x;    double t;       long k;      unsigned v;
```

The starred casts can cause run-time errors that will not be detected by the system and may cause the user's output to be meaningless.

	Cast	Nature of Change
*	(float) t;	Shortening (possible precision loss and magnitude may be too large)
	(double) x;	Lengthening (safe)
	(double) t;	No change (this is legal)
	(float) k;	Representation conversion (usually safe)
	(int) x;	Representation conversion (fractional part is lost)
*	(short) k;	Shortening (error if value of k is larger than 32,767)
*	(signed) v;	Type relabeling only (error if v > INT_MAX)
*	(unsigned long) k;	Type relabeling only (error if k is negative)

Figure 7.20. Kinds of casts.

A cast from a floating-point type to an integer type truncates the value (in the same way that assignment truncates). Casting does not round to the nearest integer; if rounding is needed, it must be done explicitly, by using `rint()` before the value is converted or assigned to an integer variable.

Figure 7.21 contains a simple example of how information can be lost unintentionally during type conversions. We start with a `float` value, convert it into an `int`, and then convert it back again. The values of `y` and `x` are different because information was lost when the value of `y` was cast to `int`. That information cannot be recovered by converting it back again.

Notes on Figure 7.21. Rounding and truncation.

This program uses type casts and compares the effects of rounding, casting, and assignment.

```
#include <stdio.h>
#include <math.h>

int main( void )
{
    double x, y = 17.7;
    int k, m, n;

    k = (int) y;    // Casting a float to type int truncates
    m = y;          // Assigning a float to an int variable causes truncation.

    printf( "Casting:\t    y= %6.2f k= %3i x= %6.2f \n", y, k, x );
    printf( "Assignment:\t y= %6.2f m= %3i x= %6.2f \n", y, m, x );

    n = rint(y);    // Rounding before assignment.

    printf( "Rounding:     y= %6.2f n= %3i \n\n", y, n );

    x = (double) k; // Casting back does not restore the fractional part.

    printf( "Re-casting:  y= %6.2f k= %3i x= %6.2f \n", y, k, x );

    return 0;
}
```

Figure 7.21. Rounding and truncation.

This program demonstrates the effects of some type coercions. Unlike the program in Figure 7.21, automatic type conversions (not explicit casts) cause the values to change here.

```
#include <stdio.h>
#include <math.h>

int main( void )
{
    float t, w;
    float x = 17.7;
    int k;

    k = x;          // A. The = coerces the float value to type int.
    t = k + 1.0;    // B. The + coerces value of k to type double.
                    //    The = coerces the sum to type float.
    w = sin( x );   // C. Calling sin() coerces x to type double.
                    //    The = coerces the double result of sin() to float.
    printf( "x= %.2f   k= %3i   t= %.2f   w= %.2f\n", x, k, t, w );
}
```

The output is

```
x= 17.70   k=  17   t= 18.00   w= -0.91
```

Figure 7.22. Type coercion.

First box: truncation. In the first line, a cast is used to convert a floating value to an integer value, and the result is stored in an integer variable. This truncates the fractional part of the number, which is lost permanently. The second line has the same effect. The compiler sees that the type of the variable on the left side of the assignment does not match the type of the value on the right, so it automatically generates the `(int)` type cast to make the assignment possible. We say the compiler *coerces* the `double` value to an `int` value. The first two lines of output, below, show the results.

Second box: rounding. This box contains a call on `rint()` which rounds `y` to the nearest integer, but returns a value of type `double`. That value is immediately coerced to type `int` and stored in an integer variable. The third line of output, below, shows the result.

Third box: casting back to type `double`***.*** The value of `y` was previously cast to `int` and stored in `k`. Now we take the value of `k` and cast it back to type `double`. Note that this does not (and can not) restore the fractional part; once it is gone, it is gone. The fourth line of output, below, demonstrates these results.

The output.

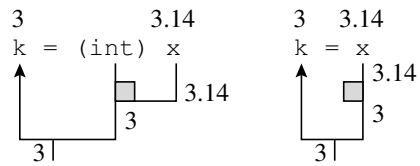
```
Casting:         y=  17.70  k=  17  x=   0.00
Assignment:      y=  17.70  m=  17  x=   0.00
Rounding:        y=  17.70  n=  18
Casting back:    y=  17.70  k=  17  x=  17.00
```

Automatic type coercion. All of the arithmetic operators defined in Figure 4.1, except `%`, can be used with floating-point types. Within the representational limits of the computer, these operators implement the mathematical operations of addition, subtraction, multiplication, and division. If both operands are `float`s, the result is a `float`. If both are `double`s, the result is a `double`. If the operands have different types, the compiler will recognize this and attempt to unify the types. A type coercion is a type conversion applied by the compiler to make sense of the types in an expression. C will insert the conversion code before it compiles the operation that requires it. Coercions happen in three basic cases:

1. When the type of the value in a `return` statement does not match the type declared in the function's prototype. If you are performing your calculations carefully, this case should not happen, and many

The examples use these declarations:

```
int k;
double x = 3.14;
```



The diagram on the left is a cast; on the right is a coercion. In both cases, a black conversion box marks the point at which a value is converted from one type to another.

Figure 7.23. Diagramming a cast and a coercion.

compilers give warnings when it does occur. It is better style to use an explicit cast in the return statement than to rely on coercion in this case.

2. When the type of an argument to a function does not match the type of the corresponding parameter in the function's prototype. Many of the functions in the math library have `double` parameters and frequently `int` or `float` arguments are passed to them. This is seen in Figure 7.22, line C, where the `float` value of `x` is coerced to type `double`. This kind of coercion may be safe or unsafe. It is normal style to use the safe (lengthening) coercions, and they are used very often. However, coercion should not be used for unsafe (shortening) conversions; use an explicit cast instead.
3. When an arithmetic or comparison operator is used with operands of mismatched types, such as
   (a) When the value of an expression is being saved into a variable using an assignment statement, as in Figure 7.22, line A. This conversion from the expression type (`float`) to the target type (`int`) is automatic and performed whether safe or not. Examples of coercing a `double` value to type `float` are given in lines B and C. Note that neither of the explicit casts used in Figure 7.21 was necessary. The compiler would have coerced the values into the new formats automatically because a value of one type was being stored in a variable of a different type.
   (b) When an operator has two real operands or two integer operands, but the operands have different lengths. The shorter value is converted to the type of the longer value, so that no information is lost. Therefore, `short` is converted to `int`, `int` to `long`, and `float` to `double`. The result of the operation will have the longer type.
   (c) When an operator has operands of mixed representations, as in Figure 7.22, line B. The compiler must convert one value to the type of the other. The rule here is that the conversion always must be done safely, if possible. Therefore, the less inclusive type is converted into the more inclusive type (say, integer to `float` or `double`), so that usually no information is lost. Because of this, most expressions that have a real operand will produce a real result, and many of these are in the `double` format.

Is coercion a good thing? Yes, because it allows functions to be easily used with arguments types that are compatible with the parameter types, but not exactly the same. Coercion frees the programmer to think about the calculations, not the representation of the data. Using coercion instead of explicit casts shortens the program and brings the basic calculation into clearer focus. However, it is not wise to depend on coercion unless you are sure that the resulting conversion will be safe, and type warning errors should be eliminated by using explicit casts wherever necessary.

7.5.3 Diagramming Conversions

We use parse trees to help us understand the structure of expressions as well as to manually evaluate them. Since coercions and casts affect the results of evaluation, we need a way to show them in a parse tree. Both will be noted on a parse tree as small black squares. Figure 7.23 shows an example of each and Figure 7.24 diagrams casts and coercions within larger expressions.

The examples use these declarations:

```
int k=3, r1=1, r2=2;
float w=1.57080;
double x, r_eq;
```

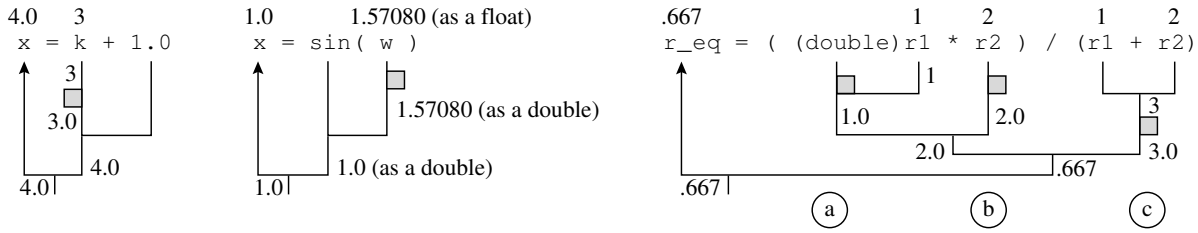


Figure 7.24. Expressions with casts and coercions.

Notes on Figure 7.23. Diagramming a cast and a coercion.

Diagram on left: a cast. A type cast is a unary prefix operator; we diagram it with the usual one-armed unary bracket. In addition, we write a black box on the bracket to denote a type conversion. In this example, the real value 3.14 is on the tree above the box; it is converted at the box and becomes the integer 3 below the box.

Diagram on right: a coercion. When a real number is stored in an integer variable, it must be coerced first. We represent the coercion by a black box on a branch of the parse tree. Even though no cast is written here, the real value 3.14 above the box is converted at the box to become the integer 3 below the box. The result is the same as if it had been cast.

Notes on Figure 7.24. Expressions with casts and coercions.

Leftmost diagram: coercion of left operand of `+`. The first operand of `+` is an `int`, the second is a `double` literal. The integer will be coerced to type `double` and real addition will be performed. The result is a `double` stored in `x` with no further conversion.

Middle diagram: coercion of an argument. The trigonometric functions in the standard mathematics library are `double`→`double` functions whose arguments must be given in radians. Here we call `sin()` with a `float` argument, which is coerced to `double` before calling `sin()`. The result is a `double` that is stored in `x` with no further conversion.

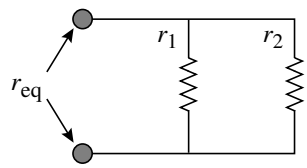
Right diagram: a larger expression. The formula from the third box in Figure 7.26 is diagrammed on the right. In this example, the operand `r1` is explicitly cast (a) from `int` to `double`. This forces the second operand `r2` to be coerced (b) so that real multiplication can happen, producing a `double` value for the numerator of the fraction. The result of the addition in the denominator is an `int`; it is coerced to type `double` (c) to match the numerator. Real division is done and the `double` result is stored in `r_eq`, a `double` variable, with no change.

7.5.4 Using Type Casts to Avoid Integer Division Problems

As discussed earlier, division is an operation whose meaning is quite different for integers and reals; a programmer needs to be aware of these differences. At times, integer division (keeping only the quotient) is a desirable outcome; but at many other times, it is not. Often, even when dividing one integer by another, the fractional part of the answer is needed for the application. Therefore, be careful when writing expressions; divide one integer operand by another only when an integer answer is needed. Otherwise, one of the integers must be cast to a floating-point type before the division. Figures 7.25 and 7.26 illustrate an application in which the use of a cast operation on `int` values achieves the necessary precision in the answer.

Problem scope: Find the electrical resistance equivalent, r_{eq}, for two resistors wired in parallel.
Input: Two integer resistance values, r_1 and r_2.
Limitations: The resistances will be between 1 and 1,000 ohms.
Formula:

$$r_{\text{eq}} = \frac{r_1 * r_2}{r_1 + r_2}$$



Output required: The two inputs and their equivalent resistance.
Computational requirements: The equivalent resistance must be accurate to two decimal places.

Figure 7.25. Problem specification: computing resistance.

A division application: Computing resistance. Figure 7.25 is a simplification of the problem presented in Figure 4.27; it computes the equivalent resistance of two parallel resistors (rather than three). In Figure 7.26, we show how precision can be lost due to careless use of integer divison to calculate `r_eq`. Then we compare this answer to a second value calculated using floating-point variables.

Notes on Figure 7.26. Computing resistance.

First box: the input.

- We use one prompt and one `scanf()` statement to read two input values; the format contains two `%` codes and we supply two addresses. As long as it is logical and causes no confusion, it is better human engineering to combine the inputs on one line, because this is faster and more convenient for the user.

We show how to use a type cast or coercion to solve the problem specified in Figure 7.25.

```
#include <stdio.h>

int main( void )
{
    int r1, r2;           // integer input variables for two resistances
    double r1d, r2d;      // double variables for two resistances
    double r_eq;          // equivalent resistance of r1 and r2 in parallel

    printf( "\n Enter integer resistances #1 and #2 (ohms): " );
    scanf( "%i%i", &r1, &r2 );
    printf( "    r1 = %i    r2 = %i \n", r1, r2 );

    r_eq = (r1 * r2) / (r1 + r2);           // Oops!  Integer division.
    printf( "    The truncated resistance value is %g\n", r_eq );

    r_eq = ((double)r1 * r2) / (r1 + r2);   // Better:  we cast first.
    printf( "    We cast to double first and get %g\n", r_eq );

    r1d = r1;                         // Coerce to type double...
    r2d = r2;                         // by copying into double variables
    r_eq = (r1d * r2d) / (r1d + r2d); // and compute using doubles.
    printf( "    The true value of equivalent resistance is %g\n", r_eq );
}
```

Figure 7.26. Computing resistance.

- We use integer input here because we want to illustrate a potential problem with integer arithmetic. However, the inputs could have been read directly into `double` variables, avoiding the need for the casts or coercions demonstrated next.

Second box: the integer calculation.

- Since `r1` and `r2` are integers, integer arithmetic will be used throughout the expression and the result will be an integer. The result will be coerced to type `double` after the calculation and before being stored in `r_eq`. Two serious problems arise with this computation, as it is written, that can cause the answer to be less accurate than desired.
- First, the programmer intended to have a real result. You might think that, since the answer is stored in a `double`, it would have a fractional part. But that is not how `C` works. It does not look at the context surrounding the division to find out what kind of division to perform; it looks only at the two operands, both of which are integer expressions in this case. For two integer operands, it performs integer division, so the fractional part of the result stored in `r_eq` will be 0.
- Second, on a machine with 2-byte `int`s, the result of the multiplication could be a number too large to be represented as an `int`, even when the inputs are relatively small. If this occurs, the overall result will be wrong due to the overflow, a condition we discuss in Section 7.6.

Third box: using a cast.

- If any one of the original four operands or the resulting numerator or denominator is cast to a floating-point type, real division will be performed, as demonstrated by the fractional portion of the output.
- Here we cast the first operand of the numerator to `double`, thereby causing real multiplication to be used. Integer addition still will be performed, however, because neither of the operands in the denominator was cast. The result of the addition will be coerced to type `double` before the division is done.

Fourth box: using `double` ***variables and coercion.***

- The output from two runs of this program is shown below. The fractional parts of the correct answers are lost when integer division is used. However, correct answers are obtained when floating-point operations are performed.

```
Enter integer resistances #1 and #2 (ohms): 20 24
   r1 = 20    r2 = 24

   The truncated resistance value is 10
   We cast to double first and get 10.9091
   The true value of equivalent resistance is 10.9091

Enter integer resistances #1 and #2 (ohms): 1 2
   r1 = 1    r2 = 2

   The truncated resistance value is 0
   We cast to double first and get 0.666667
   The true value of equivalent resistance is 0.666667
```

- When we assign a value to a variable of a different type, the compiler coerces the value to the type of the variable. The integer values entered into the program are transferred from `int` variables into `double` variables. This tells `C` to find the `double` representation of the numbers `r1` and `r2`. Since integers are a subset of the real numbers, this type conversion is usually safe.

7.6 The Trouble with Numbers

Now that we better understand the limitations of the various data types and how conversions between the types occur automatically or at our instruction, we need to consider how to use this knowledge to our advantage. In this section, we discuss how to deal with some computational problems, such as how to properly compare two numbers and what happens when a computed value is outside of the representable range of the data type.

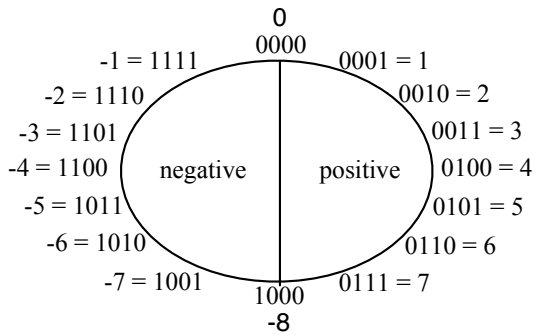


Figure 7.27. Overflow and wrap with a four bit signed integer.

The integer types provide a precise representation for numbers within a restricted range. The restriction is particularly severe for short integers, which are not large enough to store the results of many computations. While the overall range of numbers that can be represented by floating-point types is vastly greater, it still is finite and the representation used is an approximation of the real number with limited precision. We saw some of this precisional error in the last section. The various mathematical operations can produce inaccurate or completely incorrect results if the operands are either too large or too small or if the two operands differ greatly in size. These computational problems are demonstrated in more detail by the following short programs.

7.6.1 Overflow

Overflow is the error condition that occurs when the result of an operation becomes larger than the limits of the representation, as described in Figures 7.7 and 7.8. How this error condition is detected and handled differs for the integer and real data types. These overflow situations are a serious problem. They cannot be detected by the compiler. The compiler cannot predict that a result will overflow because it cannot know what data will be used later, at run time, to make calculations. Also, a C system will not detect the error at run time and will not give any warning that it has happened. It usually is possible to look at the results of calculations on the screen and notice when something has gone wrong, but this is far from a desirable solution.

Integer overflow and wrap. Integer overflow happens whenever there is a carry *into* the sign bit (leftmost bit) of a signed integer. The result is that the number "wraps around" from positive to negative or negative to positive. **Wrap** is illustrated for 4-bit integers in Figure 7.27. With four bits, we can represent the numbers -8...+7. The four bits represent $-8, 4, 2, and 1$ so that, for example, 1101 represents $- * +4 + 1 = -3$.

Suppose we start with the value +5 and repeatedly add 1. We get +6 and +7, then there is a carry into the leftmost bit (the sign bit), and wrap happens, giving us -8. If we continue adding 1, we progress, through all possible negative values, toward zero, and back into the positive range of values. Formally, we can say that when x is the largest positive signed integer that we can represent, $x + 1$ will be the smallest (farthest from 0) negative integer.

Overflow also happens when an operation produces a result too large to store in the variable that is supposed to receive it. Unfortunately, there is no systematic way to detect overflow or wrap after it happens. Avoidance is the best policy, and that requires a combination of programmer awareness and caution when working with integers. Expressions that cause **integer overflow** are fairly common on small computers because the range of type `int` is so restricted. For a 2-byte signed integer, the largest value is 32,767, which is represented by the bit sequence x =01111111 11111111. The value of $x+1$ is 10000000 00000000 in binary and -32768 in base 10.

Similarly, with unsigned integers, overflow and wrap happen whenever there is a carry *out of* the leftmost bit of the integer. In this case, if x is the largest unsigned integer, $x+1$ will be 0. To be specific, for a 2-byte model, the largest unsigned value is 65,535, which is represented by the bit sequence x =11111111 11111111. The value, in binary, of $x + 1$ is 00000000 00000000.

Integer calculations like addition, subtraction, and multiplication with large numbers are likely to exceed the maximum limit, perhaps by quite a lot.[13] On a 16-bit machine, if a computation causes overflow and the result is stored in a variable, only the rightmost (least significant) 16 bits of the overlarge answer will be stored; the rest will be truncated. If the result then is printed, it will appear much smaller than the mathematically correct result (or even negative). Noticing the faulty value is the only way for a user to detect an overflow.

For example, suppose that the 2-byte integer variable `k` contains the number 32300 and you enter a loop that adds 100 to `k` seven times. The value stored in `k` would be, in turn, 32400, 32500, 32600, 32700, −32736, −32636, and finally −32536. The value has **wrapped** around and become negative, but that does not stop the computer! The program will continue running with a faulty number that is not even approximately correct. For these reasons, 2-byte integers (type `int` on smaller machines and type `short int` on most machines) are not very useful for serious numeric work. We generally use type `long` if we want to use integers in these calculations. But even though type `long`, with a range up to 2.1 billion, can handle many more calculations properly, it still is limited to 10 digits.

Floating-point overflow and infinity. The phenomenon of wrap is unique to integers; floating-point overflow is handled differently. The IEEE **floating-point standard** defines a special bit pattern, called `Infinity`, that will result if overflow occurs during a computation. (The exponent field of this value is set to all 1 bits, the mantissa to 0 bits.) The constant `HUGE_VAL`, defined in `math.h`, is set to be the "infinity" value on each local system. One way that an overflow can be detected is by comparing a result to `HUGE_VAL` or `-HUGE_VAL`. Systems that implement the full IEEE standard provide the function `finite(x)` in the `math` library, which returns `true` if `x` is a legitimate number or `false` if it is an infinite value or a `NaN` error. Also, in such systems, `printf()` will output overflow values as `+Infinity` or `-Infinity`. Note that the `finite()` function is used to end the `for` loop in Figure 7.28.

Factorial: A demonstration of overflow. As an illustration of what all this means in practice, consider the mathematical factorial operation:

$$N! = 1 \times 2 \times \ldots \times (N-2) \times (N-1) \times N$$

Factorial, by its nature, is a function that grows large very rapidly. Figure 7.28 shows a version of the factorial program that computes $N!$ for values of N ranging from 1 to 40. The computation is made with variables of five different types so that we can compare the range and precision of these types.

Wrap error. The output on the first seven lines is fully correct. (Horizontal spacing has been reduced.)

```
N    N factorial
      short  unsigned
       int   short int  long int        float            double

 1       1       1            1            1                 1
 2       2       2            2            2                 2
 3       6       6            6            6                 6
 4      24      24           24           24                24
 5     120     120          120          120               120
 6     720     720          720          720               720
 7    5040    5040         5040         5040              5040
```

However, 7! is the largest factorial value that can be stored in a short signed (16-bit) integer. From line 8 on, the numbers in the first column are meaningless; overflow has happened and the answer wraps around to become a negative number. This will not always occur, but when it does, it is a good indication of trouble.

```
       short  unsigned
        int   short int  long int       float       double

 8  -25216   40320        40320        40320        40320
 9  -30336   35200       362880       362880       362880
10                      3628800      3628800      3628800
11                     39916800     39916800     39916800
```

We compute the factorial function using five different types so that we can compare their range and precision.

```
#include <stdio.h>

int main( void )
{
    int N;                      // Loop counter.
    short int       facts = 1;  // We compute factorial using 5 types.
    short unsigned  factu = 1;  // 0! is defined to be 1.
    long int        factl = 1;
    float           factf = 1.0;
    double          factd = 1.0;

    puts( "\n N    N factorial \n       short unsigned \n"
          "       int   short int  long int \t float \t\t\t double \n" );

    // Compute N! using each type, quit after 40 factorial.
    for (N = 1; finite( factd ); ++N) {
        facts *= N;  factu *= N;  factl *= N;
        factf *= N;  factd *= N;

        if (N <= 9)
            printf( "%3i %7hi %7u ", N, facts, factu );
        else
            printf( "%3i                 ", N );
        if (N <= 17)
            printf( "%12li", factl );
        else
            printf( "            " );
        printf( "  %18.12g  %23.22g\n", factf, factd );
    }
}
```

Figure 7.28. Computing *N*!.

One more value, 8!, can be stored in a short unsigned integer. But, beginning with 9!, the answer overflows into the 17th bit position and the number stored in the variable `factu` is garbage. When working with unsigned numbers, as in the second column, there is not even a negative sign to warn us about the wrap. Nonetheless, the numbers are wrong from line 9 on. This is what makes it so difficult to detect this error in practice. The program suppresses output in these two columns after line 9.

	long int	float	double
11	39916800	39916800	39916800
12	479001600	479001600	479001600
13	1932053504	6227020800	6227020800
14	1278945280	87178289152	87178291200
15	2004310016	1.30767427994e+12	1307674368000

Using long integers, as in the third column, we get correct answers all the way up to 12!, the largest N for which the calculation can be made using either signed or unsigned long integers. Starting at line 13, the answer for long integers is garbage; it should be the same as the value in the other columns. Here, we get no negative sign to warn us that wrap has occurred, because the value has wrapped past all of the negatives and into the positives again.

Representational error. Between $N = 14$ and $N = 34$, using type `float`, we encounter the limits of the IEEE `float`'s precision, rather than its range. Although we can compute the factorial function for $N > 13$, the answers are only approximations of the true answer. (The `float` value computed for $N = 14$ is 87,178,289,152; this is close to, but smaller than, the true answer, 87,178,291,200, shown in the last column

[13] Unlike the `*`, `+`, and `-` operators, integer division cannot cause overflow. The smallest integer value you can divide by is `1`, which will not increase the magnitude of the value being divided.

for the `double` calculation.) The `float` simply lacks enough bits to hold all the significant digits, even though the maximum `float` value has not been reached. We say that such an answer is **correct but not precise**. It may be a fully acceptable approximation to the true answer, but it differs in the last few digits. Whether the precision is adequate depends on the application.

```
            float                   double

15   1.30767427994e+12            1307674368000
16   2.0922788479e+13            20922789888000
17   3.55687414628e+14          355687428096000
18   6.40237353042e+15         6402373705728000
19   1.21645096004e+17       121645100408832000
20   2.43290202316e+18      2432902008176640000
21   5.10909408372e+19     51090942171709440000
22   1.12400072481e+21   1124000727777607680000
23   2.58520174446e+22  25852016738884978212 86e+22
```

Using type `double` (as in the last column) instead of `float` extends the range of accuracy. The same factorial program goes up to 22! with total precision; this number has 18 nonzero digits. For $N = 23$, the last few digits show evidence of the error, they should be 664000 not 821286.

```
            float                   double

33     8.68331850985e+36  8.68331761881188593871 6e+36
34     2.95232822997e+38  2.95232799039604119555 1e+38
35             +Infinity  1.03331479663861442222 1e+40
36             +Infinity  3.71993326789901177492 4e+41
```

At $N = 35$, floating-point overflow happens, for the `float` number format where `3.402e+38` is the maximum representable number. However, this does not stop the program, which continues to try to compute the numbers up to 170! before overflow happens using the `double` format. At 171! the test for `finite(facto)` ends the loop.

7.6.2 Underflow

The opposite problem of overflow is **underflow**, which occurs when the magnitude of the number falls below the smallest number in the representable range. This cannot occur for integers, only for real numbers, since the minimum magnitude of an `int` is 0. For real numbers, underflow happens when a value is generated that has a **0 exponent**[14] and a **nonzero mantissa**. Such a number is referred to as **denormalized**, which means that all significant bits have been shifted to the right and the number is less than the lowest number specified by the standard. This effect is shown in the left column of the last few lines of output from the division program in Figure 7.29:

```
N= 43   frac= 9.9492191e-44    1+frac=             1
N= 44   frac= 9.8090893e-45    1+frac=             1
N= 45   frac= 1.4012985e-45    1+frac=             1
N= 46   frac=             0    1+frac=             1
N= 47   frac=             0    1+frac=             1
```

The program continually divides a value by 10. The actual lower limit of the representation range is 1.175e−38, and some systems will generate the 0 value when this limit is reached. Others, like the one shown here, still use the denormalized values. But even these, at $N = 46$, have all the bits shifted so far to the right that the result becomes 0.

Underflow can result from several kinds of computations:

- Dividing a number by a very large number or repeated division, as just illustrated.
- Multiplying a small number by a near-zero number, which has the same effect as dividing by a very large number.
- Subtracting two values that are near the smallest representable `float` and ought to be equal but are not quite equal because of round-off error.

[14] That is, all zero bits in the exponent, which corresponds to a large negative exponent in scientific notation.

This program continually divides a number by 10 until the result is too small to store as a normalized `float`.

```
#include <stdio.h>

int main( void )
{
    int N;
    float frac = 1.0;

    puts( "\n Dividing by 10; frac=1/(10 to the Nth power)\n" );
    for (N = 0; N < 50; ++N) {
        printf( " N=%3i   frac= %13.8g    1+frac= %13.8g\n",
                N, frac, 1+frac );
        frac = frac / 10;
    }
}
```

Figure 7.29. Floating-point underflow.

7.6.3 Orders of Magnitude

The limits of `float` precision can be a problem with addition as well as with multiplication. For example, if you attempt to add a small `float` number to a large one, and their exponents differ by more than 10^7 (or 7 **orders of magnitude**), the addition likely will have no effect. The answer will be the same large number that you started with. This is because the floating-point hardware starts the operation by lining up the decimal points of the two operands. In the process, the mantissa bits of the smaller value get denormalized (shifted to the right). But the hardware register in which this happens has a finite width, so the least significant (rightmost) bits of the smaller operand "fall off" the right end of the register and are lost. If the difference in exponents between the operands is great enough, all of the mantissa bits of the smaller value will be lost and only a value of 0 will be left when the addition happens. You can add a millimeter to a kilometer in single precision, but the answer is still 1 kilometer.

This effect is illustrated in the right column of the first few lines of output from the program in Figure 7.29, shown below. This program starts with the value 1.0, divides it repeatedly by 10, and adds each fractional result to 1. After only nine divisions, the original fraction is so small that the addition has no effect. We say that the fraction is insignificant in comparison to 1.0.

```
Dividing by 10; frac=1/(10 to the Nth power)

N=  0   frac=             1    1+frac=             2
N=  1   frac=           0.1    1+frac=           1.1
N=  2   frac=  0.0099999998    1+frac=          1.01
N=  3   frac= 0.00099999993    1+frac=         1.001
N=  4   frac=  9.999999e-05    1+frac=        1.0001
N=  5   frac= 9.9999988e-06    1+frac=       1.00001
N=  6   frac= 9.9999988e-07    1+frac=      1.000001
N=  7   frac= 9.9999987e-08    1+frac=     1.0000001
N=  8   frac= 9.9999991e-09    1+frac=             1
N=  9   frac= 9.9999986e-10    1+frac=             1
```

The order of operations. When dealing with a set of numbers that have highly variable magnitudes, the accuracy of a final result can depend on the order in which operations are performed. For instance, suppose you want the total of a large number of values. If they are all nearly the same size, order does not matter. However, if a few are huge and most are very small, adding up the huge ones first will cause the small ones to be insignificant in proportion to the sum of the large ones. However, if the small ones are added first, their sum may be of an order of magnitude similar to the large values, and therefore, make an important contribution to the overall sum.

Some techniques in numerical analysis also require attention to the magnitude of the numbers that are involved. One example is the Gaussian elimination algorithm for solving a set of simultaneous linear equations. In this algorithm, coefficients of the equations are repeatedly subtracted from, multiplied by, and divided by other coefficients. The subtractions can produce results that are close to, but not quite, zero.

Calculate the number of 11-card Canasta hands that can be dealt from a deck of 104 cards. The two calculation methods below are mathematically equivalent, but the first one fails due to overflow. The second one works properly.

Method 1: Use the mathematical formula and call the factorial function.

```
float factorial( int n );       // Prototype of factorial function.
combinations = factorial( 104 ) / (factorial( 11 ) * factorial( 104-11 ));
```

Method 2: Alternate division and multiplication to keep answer within the range of type `float`.

```
float combinations = 1;  // The answer, so far.
float quotient;          // One term of the formula.
for (int k=11; k>0; --k){
        quotient = (double)(104-k+1) / k;
        combinations *= quotient;
}
```

Figure 7.30. Calculation order matters.

But dividing by such a number might cause floating-point overflow. Happily, there is considerable choice about the order in which the coefficients are used, and the solution to this problem is always to process the largest remaining coefficient next.

7.6.4 Not a Number

Last, a special value called `NaN`, which stands for "not a number," can be generated through operations such as `0 / 0`. This is another special bit pattern that does not correspond to a real value. The IEEE standard specifies that any further operation attempted using a `NaN` or `Infinity` as an operand will return the same value. This was seen for `+Infinity` in the factorial example. On our system, the hardware computes `Infinity` and `NaN` values correctly and C's `stdio` library prints them (as was shown) instead of printing meaningless digits.

Sometimes the order in which a set of calculations is performed can cause an error, while the same operations done in a different order can be correct. For example, suppose we wished to calculate the number of different hands a player might get in the card game Canasta. In this game, a deck has $n = 104$ cards and each player is dealt a hand of $k = 11$ cards. The formula for the number of different hands H can be given two ways:

$$H = \frac{n!}{k! \times (n-k)!} = \frac{n}{k} \times \frac{n-1}{k-1} \cdots \frac{n-k+1}{1}$$

The first formula is the one you are likely to see in a book on probability. However, the number of Canasta hands is calculated using type float and using the formula as written, overflow happens. This is shown by Method 1 in Figure 7.30, which gives this result:

```
Numerator= inf  Denom1=3.99168e+07  Denom2=inf  Combinations=nan
```

The second method works correctly and gives this answer, which is correct:

```
Combinations= 2.23045e+14
```

7.6.5 Representational Error

When two integers are compared, they are either equal or not; this is because we use an exact representation for integers, and they are discrete values (each one differs by exactly 1 from the next). In contrast, the real numbers are not discrete; they are continuous, that is, an infinite number of real values lie between any two we care to write. We can represent some of those numbers exactly but most can be represented only by

an approximation. The difference between the true value and its representation is called **representational error**. Types `float` and `double` are **approximate representations** for the real numbers, but with differing precision. As an example, consider this code fragment:

```
float w = 4.4;
double x = 4.4;
printf( " Is x == (double)w? %i \n", (x == (double)w) );
printf( " Is (float)x == w? %i \n", ((float)x == w) );
```

The output, shown below, is unexpected if you forget that the two numbers are represented with limited, and different, **precision** and that the `==` operator tests for exact bit-by-bit equality.

```
Is x == (double)w? 0
Is (float)x == w? 1
```

When the more-precise value is cast to the less-precise type, the extra bits are truncated and the numbers are exactly equal. When the shorter value is cast to the longer type, it is lengthened by adding zero bits at the end of the mantissa, not by recomputing the additional bit values. In general, these zeros are not equal to the meaningful bits in the `double` value.

Computation also can introduce representational error, as shown by the next code fragment. We start with `y`, divide it by a number, then multiply it by the same number. According to mathematics, the result should be the same real number we started with. According to our computer it is, but only sometimes, as with this first set of initial values:

```
float w;
double x,  y = 11.0,  z = 9.0;

x = z * (y / z) ;
w = y - x;
printf( "\n w=%g  x=%.10f \n", w, x );
```

The results from computing this on our system are

```
w=0  x=11.0000000000
```

But if we change the initial values to `y = 15.0` and `z = 11.0`, the results are different and the value of `w` is nonzero:

```
w=1.77635e-15  x=15.0000000000
```

Why does this happen? The answer to a floating-point division has a fractional part that is represented with as much precision as the hardware will allow. However, the precision is not infinite and there is a tiny amount of truncation error after most calculations. Therefore, the answer to `y / z` may have error in it, and that error is increased when we multiply by `z`. This is why the answer to `z * (y / z)` does not always equal the number `y` that we started with.

7.6.6 Making Meaningful Comparisons

The question then arises, when are two floating-point numbers really equal? The answer is that they should be called *equal* if both are approximations for the same real number, even if one approximation has more precision than the other. Therefore, an approximate test for equality is necessary to compare values that are approximations.

Practical problems often require comparing a calculated value to a specific constant or setpoint or comparing two calculated values that should be equal. Such a comparison is not as simple as it seems, because even simple computations with small floating-point values can have results that differ from the mathematically correct versions. If you read two identical floating-point values into variables of the same floating type and compare them, the values will be equal. However, as soon as you begin to compute, truncation and round-off error can happen. Any computed value could be affected by floating-point representational error. Further, two computed values could be affected by different amounts and in different directions.

With the epsilon shown (2 mph), we say that $speed_1 = 49.0$ equals 50.0 because it is within epsilon of 50.0, but $speed_2 = 54.0$ does not equal 50.0 with this value of epsilon.

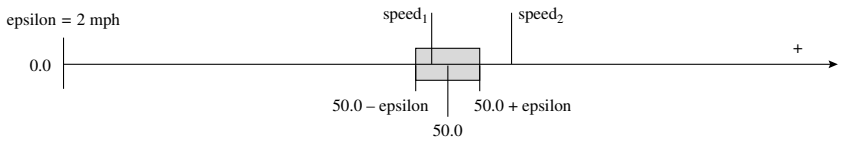


Figure 7.31. An approximate comparison.

Although truncation itself always results in a value smaller than it should be, using a truncated answer as a divisor gives a quotient that is too large. It takes considerable expertise to analyze how severely a number might be affected and in what way. In the example of representational error given previously, doing `y - (z * (y / z))` gave a nonzero answer for `y = 15.0` and `z = 11.0` because of round-off error due to the division, but the same computation on other values of `y` and `z` gave the answer 0.0. There was no obvious pattern to these zero and nonzero answers when the test was tried with other inputs.

Even though we know that the various results of `z * (y / z)` will be very close to the value of `y`, the `==` operator tests for exact, not approximate, equality. Since any floating-point value that results from a computation may be imprecise, we cannot use `==` and `!=` on `float`s and `double`s. We can get around this comparison problem by comparing the *difference* of the two numbers to a preset epsilon value, as in Figure 7.31. We call this an **approximate comparison** for equality with an **epsilon test**. For any given application, we can choose a value of epsilon that is slightly smaller than the smallest measurable difference in the data. We then ask if the absolute value of the difference between the values is less than epsilon—if so, we say the operands are equal. This can be done in one `if` statement by using the absolute value function, `fabs()`, as shown in Figure 7.32.

In addition to testing for equality, occasionally we also need to test for a greater-than or less-than condition. In a one-sided test, we still need an epsilon value to compensate for representational error, but the `fabs()` can be omitted. This kind of test is used in Figure 7.34.

7.6.7 Application: Cruise Control

Sometimes different actions are required for values below, above, and equal to a target, so we need to use a series of `if` statements to test for these conditions. The program specified in Figure 7.33 and written in Figure 7.34 demonstrates this technique. The figures present an initial version of a cruise control program that would run on a computer embedded in the acceleration system of an automobile to regulate the setting of the automobile throttle. A real cruise control would need to be more complex to avoid drastically overshooting and undershooting the target speed.

Use an epsilon test with the absolute value function to compare floating-point values for equality. Note, the `fabs()` function is part of the `math` library and is used with real numbers, as opposed to `abs()`, which is used with integers.

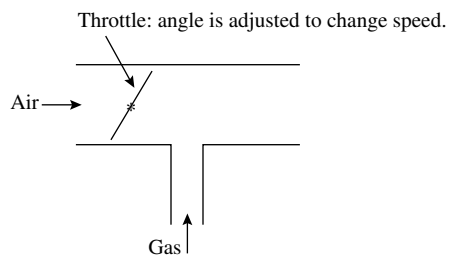
```
double epsilon = 1.0e-3;
double number, target;

if (fabs( number - target ) < epsilon)    // fabs is floating abs.
    // then we consider that number == target
else
    // we consider the values significantly different.
```

Figure 7.32. Comparing floats for equality.

Problem scope: A simple version of a cruise control program that would run on a computer embedded in the acceleration system of an automobile.

Inputs: These come from the functions `read_on_switch()`, `read_speed()`, `read_throttle()`, and `read_brake()`, which are attached to the car's sensors. Prototypes for these functions are in the file `throttle.h`. There is no direct interaction with a user.

Formulas: Increasing or decreasing the angle of the throttle affects the speed. An angle of 0° corresponds to a horizontal throttle and high speed, while the maximum angle of 90° corresponds to a vertical throttle and low speed. At 90°, we assume that some air still can enter the system because the throttle plate is designed to be smaller than the diameter of the tube. We need some air at all times for combustion of the gas-air mixture.



Constants required: `eps` = 2.5 mph, the "fuzz factor" for the speed comparison, and `delta` = 5°, the incremental correctional change in throttle setting.

Output required: No output report is displayed on a screen. Instead the output is in the form of appropriate calls on the `set_throttle()` function, which controls the position of the car's throttle.

Figure 7.33. Problem specification: Cruise control.

Notes on Figure 7.34. Cruise control.

First box: the constants.

- We define `eps` to be small so that the cruise control system can regulate the speed within a narrow range.
- The throttle setting ranges between 0.0° (horizontal) and 90.0° (vertical); we adjust it by 5° each time we need to raise or lower the car's speed.

Second box: waiting for the "set" signal.

- When the cruise control first is turned on, it waits for the driver to press the "set speed" switch. This is performed by the one-line loop that repeats until the "set" signal is received. Note that no actual statement is being executed each time through the loop. The loop simply repeats the test until the test is false. This typically is called a *busy wait loop* and was discussed in Chapter 6.
- As soon as the "set" switch is recognized, we leave the loop and get the input values by calling functions to read the current speed and throttle settings.

Outer box: responding to conditions.

- This loop will monitor the car's progress. The cruise control will remain active until the driver touches the brake.
- While active, it continually reads the current speed and compares it to the target speed.
- After computing the new throttle setting in the inner box, we generate the output signal of the program by calling the `set_throttle()` function.

Inner box: adjusting the throttle.

- Because the speed is a `float`, we use an epsilon test; that is, we declare the numbers to be equal if they differ by less than epsilon (2.5 mph).
- Here it is not just important to know whether the speeds are the same but, when they are different, which is greater. We use an `if...else` sequence to handle this.
- If the current speed is slower than the target speed minus epsilon, we subtract delta from the throttle setting. If the current speed is too fast, we add delta. (An actual cruise control algorithm would use more information than just the current speed to make this decision.) If both these tests fail, then the speeds would be "equal" according to our original approximate-equality test.

The car's throttle setting is increased or decreased in response to the measured speed being too low or too high.

```
#include <stdio.h>
#include "throttle.h"    // Prototypes for switch and throttle functions.

int main( void )
{
    const float eps = 2.5;      // Fuzz factor for comparison.
    const float delta = 5.;     // Change for throttle setting, in degrees.

    float throttle;             // Current throttle setting.
    float target;               // Desired speed setpoint.
    float speed;                // Current speed.
    float dif;                  // Target speed - current speed.

    while (! read_on_switch()); // Leave loop when driver sets speed.
    target = read_speed();      // Initial speed and throttle settings.
    throttle = read_throttle();

    while (! read_brake()){          // Leave loop when driver hits brake.
        speed = read_speed();

        dif = target - speed;        // Compare current speed to target
        if (dif < -eps){
            puts( "Speed is too low; open throttle." );
            throttle -= delta;
            if (throttle > 90.0) throttle = 90.0;
        }
        else if (dif > eps){
            puts( "Speed is too high; close throttle." );
            throttle += delta;
            if (throttle < 0.)  throttle = 0.;
        }
        else puts( "Speed is ok; do nothing." );
        set_throttle( throttle );
    }
}
```

Figure 7.34. Cruise control.

7.7 What You Should Remember

7.7.1 Major Concepts

Integers and their properties.

- *Integers come in many forms in* C: `signed` and `unsigned`, `short` and `long`. An integer type permits us to represent a limited range of values exactly. The `short signed` integers can store numbers only up to 32,767. The largest `long signed` integer is 2,147,483,647. The largest `long unsigned` integer is 4,294,967,295. If numbers larger than this are needed, a floating-point type must be used.
- *Literal integers.* A literal integer is a sign followed by a sequence of digits. The exact type of a literal depends on its sign, its magnitude, and the range of type `int` on the supporting computer hardware. The digits may be followed by a letter `L` or `U` to indicate that the number should be long or unsigned. `C` supports literals in bases 8 (octal), 10 (decimal), and 16 (hexadecimal). A literal integer is interpreted as octal if it starts with a 0 digit or hexadecimal if it starts with `0x`. Otherwise it is a decimal number.
- *Integer input formats.* Integers can be read using `%i` or `%d`. The `%i` is more general and can input base ten and hexadecimal numbers. The `%d` is older and is limited to base ten (decimal) inputs. Long and short integers require different format specifiers: `%li` or `%ld` for long and `%hi` or `%hd` for short. When reading, it is important to use the format specifier that matches the type of the variable that will receive the input. Many compilers to not check for correct specifiers, and using an incompatible specifier will cause strange and unpredictable results.
- *Integer output formats.* Integers can be printed using `%i` or `%d`, with an optional field width specification between the percent sign and the letter. As with input, long and short integers require different format specifiers: `%li` or `%ld` for long and `%hi` or `%hd` for short. When printing a value it is important to use the format specifier that matches its type.
- *When making calculations with large integers,* the programmer must be wary of integer overflow and wrap. If a variable contains the maximum integer value, adding 1 will cause the value to wrap. The answer will be the minimum representable negative value (farthest from zero).

Floating-point numbers.

- C *supports floating-point numbers in two or three lengths.* These types, named `float`, `double`, and `long double`, are used to represent real numbers. As with scientific notation, a floating-point number has an exponent that encodes the order of magnitude of the number and a mantissa that encodes the numeric value to a limited number of places of precision. The limits of floating-point representation were examined in Figure 7.7.
- *The type* `double` *is the most important* of the three floating-point types, because the C mathematics library is written to process `double` numbers (not `float` or `long double`) and does all its computations with `double`s. Type `float` exists to give the programmer a choice; `double` provides twice as much precision and a much larger range of exponents but takes twice as much storage space as `float` and may take twice as long to process in a computation. When memory space and processing time do not matter, many programmers use `double` because it provides more precision.
- *Type* `float` *vs.* `double`. Because a programmer can combine types `float` and `double` freely in expressions, most of the time, it does not matter which real type is used. Sometimes the degree of precision required for the data dictates the use of `double`. Since all the functions in the math library expect `double` arguments and return `double` results, some programmers just find it easier to declare all real variables as `double`.
- *Floating-point literals.* A floating-point literal may be written with a decimal point or in scientific notation. In the first case, the decimal point may be written before the first digit, after the last digit, or anywhere in between. A literal in scientific notation starts the same way, but then has an exponent part: the letter `E` or `e`, an optional `+` or `-` sign, and an integer exponent in the range 0...38 for type `float` or 0...308 for type `double`.

- *Floating-point input formats.* For input, we have been using the type specifiers `%g` for `float` and `%lg` for `double`. The specifiers `%f` and `%lf` are also appropriate and commonly used. It is essential to use the format specifier that matches type of the variable in which the input will be stored. Otherwise, the results will be wrong and appear to be garbage.
- *Floating-point output formats.* For output, there are three formatting strategies, with a different specifier for each. The basic type specifier is `%g` for both `float` and `double`. It will print the output in whatever format seems most appropriate for the size of the number. (No letter `l` is needed or even permitted by the standard, although many compilers will accept `%lg` and do the right thing.) Optional width and precision specifications may be written between the percent and the `g`.

  The specifiers `%e` and `%f` are used when the programmer needs more control over the appearance or position of the output number. When `%e` is used, the output will be printed using scientific notation and `%f` is used with width and precision specifications to print numbers in neat columns.
- *Floating-point computations* can also cause overflow. This happens when you divide by a near-zero value or multiply two very large numbers. The number will have an exponent of all 1-bits. On some systems this will be printed as `+Infinity`; on others, it will be a number with a very large exponent.
- *Underflow.* This error condition happens when a real number becomes very close to zero but does not exactly equal zero. In this situation, some systems store the number in a denormalized form, others simply set the result to 0.
- *Comparing reals.* Computations on real numbers commonly introduce small and somewhat unpredictable representational errors. For this reason, all comparisons between computed reals should be made using a tolerance.

Choosing the proper data type.

- *An integer type or a real type?* For most problems the details of the specification will make it rather obvious whether an integer or real data type should be used to represent a particular entity. Integers typically are used for such things as loop counters, simple quantities, menu choices, and answers to simple questions. Real variables more typically are used for measurements and mathematical calculations.
- *Two other important issues.* Memory limitations and speed of execution must sometimes be considered in choosing a data type. If you are processing large amounts of data and precision is not important, then `float` variables use only half as much space as `double`s and an `int` may use even less (depending on the compiler and computer system). If speed is of concern, integer arithmetic is performed more quickly than real computations on many machines. So if integers can be used, do so. Otherwise, in general, computations involving `float` values are faster than those using `double`s, due to the smaller amount of information (number of bits) being processed.

Computational issues.

- An integer in the computer is an exact representation of the corresponding mathematical integer. However, each size of computer integer has a limited range and cannot store a number outside that range. An attempt to do so causes overflow and wrap.
- A floating-point number is an approximate representation of the corresponding mathematical real number. Computations with type `float` and `double` are subject to possible overflow, underflow, and loss of precision. After overflow or underflow, further computation is meaningless and is trapped by some (but not all) contemporary C systems. Such numbers are labeled `NaN`.

Casts and mixed-type operations.

- C supports mixed-type arithmetic. Integer and floating-point types can be mixed freely in arithmetic expressions. When two values of differing types are used with an operator, the value with less precision automatically is coerced to the more precise representation.

Group	Operators	Complications
Casts	`(int)`	Conversion from `double` or `float` will discard the fractional part.
	`(short)`	Conversion from a `long` will produce a garbage result if the value of the `long` is too great to fit into a `short`.
	`(float)`	Conversion from `int` is safe; from `double`, precision may be lost.
Coercions	`=`	Loss of precision does occur during assignment of a more-precise value to a less-precise variable.
	parameters	Argument values are coerced to match the declared types of the parameters.
	return values	The value returned by a function is coerced to match the declared function return type.

Figure 7.35. Casts and conversions in C.

- If an integer is combined with a `float` or a `double` in an expression, the integer operand always is converted to the type of the floating-point operand before the operation is performed; the result of the operation is a floating-point value.
- A type conversion may be "safe," in that it will cause no loss of information, or it may be "unsafe," because it can cause a loss of precision or simply result in total garbage. Knowing when a type conversion can be used safely is important. However, sometimes an unsafe conversion is exactly what the programmer needs.
- An explicit type cast must be used to perform real division with integer operands.

7.7.2 Sticky Points and Common Errors

Operators. The table in Figure 7.35 gives a brief summary of the difficulties that might be encountered when using C casts and conversions.

Algorithms. Know the weak points in your algorithm as well as any assumptions on which the calculations might be based. If the algorithm can "blow up" at any point, guard against that possibility.

Formats. Using the wrong conversion specifier in a format can cause input or output to appear as garbage. Default length, `short`, and `long` integers have different conversion codes, as do `signed` and `unsigned` integers.

Debugging. Insert printouts into your program after every few calculations to spot potential calculation errors.

Precision. When using reals, there is no way to tell from the printed output whether a value came from a `double` or a `float` variable. If you specify a format such as `%.10f`, you might see 10 columns of nonzero digits printed, but that does not mean that all 10 are accurate. If the number came from a `float` variable, the eighth through tenth digits usually will be garbage. A similar problem happens when the precision specification of the output is made greater than the actual precision of the input. If an answer was calculated from input having two places of precision, all decimal positions in the output after the second will be meaningless. Remember that it is up to you to limit the columns of output to the precision of the number inside the machine or the known accuracy of the calculation, whichever is smaller.

Avoiding and handling runtime errors.

- Do not try to add or subtract values of widely differing magnitudes.
- If there is any possibility that a divisor could be zero, test for it!

- Define an epsilon value, related to the precision of the input, which is the smallest meaningful value in this context. Any number whose absolute value is smaller than epsilon should be considered zero and any two numbers whose difference is less than epsilon can be considered equal.
- An output with a huge and unreasonable exponent means it is probably the result of an overflow. Each programmer needs to be able to recognize the overflow and undefined values that will be printed by the local compiler and system. If these values appear in the output, the programmer should identify and correct the erroneous computation that caused them.
- Not a number. The C standard does not specifically cover how a compiler must handle the special values `NaN`, `+Infinity`, and `-Infinity`; it leaves these results officially "undefined". This means that a particular compiler may do anything that is convenient about the problem. Many do nothing; a garbage result is returned and the user is not notified that there is an error. However, most computer hardware will set an error indicator when the various floating-point problems occur. This permits a program to test for a particular result and thereby discover these illegal operations. The user can get control by defining a signal handler[15] to trap these types of signals and process them. However, most programmers have no idea how to do this, and most user programs don't attempt to use the interrupt system. Avoidance is the best policy for the ordinary programmer. The careful programmer takes these precautions:

7.7.3 Programming Style

- It is appropriate to use integers for loop counters and customary to give them short names such as `j`, `k`, `m`, and `n`.
- Although the letters `i` and `l` have traditionally been used to name integer counters, they are poor choices, because they are easily mistaken for each other and for the numeral 1.
- Floating-point numbers traditionally have been given names starting with `f`...`h` and `r`...`z`.
- When implementing standard scientific or engineering formulas, it makes sense to use whatever variable names are used traditionally to express that formula, even when those names are single letters. Otherwise, use variable names long enough to convey the meaning or purpose of the variable.
- Use a `%f` conversion specifier if your output needs to be in neat columns. Use `%g` if you have no good idea whether the value to be printed is large or small. Use `%e` if the range of values is extreme.
- To print a table in neatly aligned columns, use a `%f` conversion specifier and include a field width. The `%g` conversion is not appropriate for tables.

Portability.

- There is some variation among compilers in the way floating-point types are handled. Sometimes the underlying computer hardware does not support floating-point arithmetic, in which case floating-point representation and computation must be emulated by software. Emulation, of course, is much slower.
- Although all ISO C compilers must permit use of the type name `long double`, many simply make it a synonym for `double`.
- Some systems use 2 bytes to represent an `int`, others use 4 bytes. The two lengths of integers make portability of code a nightmare. Unless a programmer is aware of the different meanings of `int` and assiduously avoids relying on the size of his `int`s, it is very unlikely that his or her programs will run on each kind of machine without additional debugging. Furthermore, errors due to integer sizes are among the hardest to find because of the ever-present automatic size conversions all C translators perform.

  It would be nice to avoid type `int` altogether and use only `short` and `long`. However, this is impractical because the integer functions in the C library are written to use `int` arguments and return `int` results. So what should the responsible programmer do?

[15] This subject is beyond the scope of this text.

1. Be aware.
2. Use `short` or `long` when the length is important in your application.
3. Do not rely on assumptions about the size of things.
4. Check all the possible data coercions and conversions and think about what can be done for those labeled *unsafe*.

Algorithms. Know the weak points in your algorithm as well as any assumptions on which the calculations might be based. If the algorithm can "blow up" at any point, guard against that possibility. Do not try to add or subtract values of widely differing magnitudes.

Debugging. Insert printouts into your program after every few calculations to spot potential calculation errors.

Handling error conditions. The C standard does not specifically cover how a compiler must handle the special values `NaN`, `+Infinity`, and `-Infinity`; it leaves these results officially "undefined." This means that a particular compiler may do anything convenient about the problem. Many do nothing; a garbage result is returned and the user is not notified of an error. However, most computer hardware will set an error indicator when the various floating-point problems occur. This permits a program to test for a particular result and thereby discover the illegal operations. The user can get control by defining a signal handler[16] to trap these types of signals and process them. However, most programmers have no idea how to do this, and most user programs don't attempt to use the interrupt system. Avoidance is the best policy for the ordinary programmer. The careful programmer takes these precautions:

- An output with a huge, unreasonable exponent probably is the result of an overflow. Each programmer needs to be able to recognize the overflow and undefined values that will be printed by the local compiler and system. If these values appear in the output, the programmer should identify and correct the erroneous computation that caused them.
- If a divisor possibly could be 0, test for it.
- Define an epsilon value, related to the precision of the input, that is the smallest meaningful value in this context. Any number whose absolute value is smaller than epsilon should be considered 0 and any two numbers whose difference is less than epsilon can be considered equal.

7.7.4 New and Revisited Vocabulary

These are the most important terms and concepts from this chapter that deal with the representation of numbers:

representation range	IEEE floating-point standard	precision
integer overflow	exponent	normalized and denormalized
wrap	mantissa	order of magnitude
scientific notation	floating-point overflow	correct but not precise
approximate representation	underflow	order of performing operations
	representational error	

These are the most important terms and concepts from this chapter that deal with the C language:

integer types	floating-point types	type coercion
2- and 4-byte models	floating-point type specifier	I/O conversion specifier
integer type specifier	floating-point literal	field width specifier
integer literal	representation conversion	precision specifier
literal modifiers	type cast	default output precision

[16] This subject is beyond the scope of this text.

These are the most important terms and concepts from this chapter that deal with numeric algorithms:

integer division	indeterminate results	approximate comparison
division by 0	safe conversions	epsilon test
truncation	unsafe conversions	factorial
rounding	length conversion	Euclid's method for square root

The following conversion specifiers, functions, prototypes, and library files were discussed in this chapter:

`const`	`double`	`+Infinity` and `-Infinity`
`int`	`long double`	`finite()`
`i` and `d` conversions	`e` and `le` conversions	`HUGE_VAL`
`long int`	`f` and `lf` conversions	`NaN`
`li` and `ld` conversions	`g` and `lg` conversions	`limits.h`
`short int`	`INT_MIN`	`float.h`
`hi` and `hd` conversions	`INT_MAX`	`rint()`
`signed int`	`FLT_MIN`	`sin()`
`unsigned int`	`FLT_MAX`	`cos()`
`u`, `hu` and `lu` conversions	`DBL_MIN`	`abs()`
`float`	`DBL_MAX`	`fabs()`

7.7.5 Where to Find More Information

- Unsigned integer types and the bitwise operators that work on them are covered in Chapter 15.
- The last primitive data type, pointers, is introduced in Chapter 11 (pointer parameters), and discussed extensively in Chapter 16 (dynamic allocation), Chapter 17 (pointer algorithms), Chapter 22 (linked lists), and Chapter 20 (pointers to functions).
- The IEEE Standard for floating point computation can be found through this website `en.wikipedia.org/wiki/IEEE_Floating_Point_Standard`
- A list of disasters caused by numeric errors can be found on the web by searching for "space program disaster overflow". Among the incidents listed there are:
  - Failed Navy rocket launches, 1999: bad decimal point.
  - Ariane explosion, 1996: Large float converted to integer, causing overflow.
  - Patriot-Scud, 1991: rounding error.
  - Loss of Mars orbiters, 1999: mixture of pounds and kilograms.
  - USS Yorktown "dead in the water", 1998: input and division by 0.

7.8 Exercises

7.8.1 Self-Test Exercises

1. The following functions and constants are all defined in the standard ISO C libraries. Name the specific header file that must be `#include`d to use each one.

   (a) `HUGE_VAL`
   (b) `sin()` and `cos()`
   (c) `INT_MAX`
   (d) `finite()`
   (e) `scanf()`
   (f) `fabs()`
   (g) `rint()`
   (h) `FLOAT_MAX`

2. What is the type of each of the following integer literals in a C compiler, where type `int` is the same length as type `short`? If the item is not a legal literal, say so.

   (a) 33333
   (b) 10U
   (c) 32270
   (d) −20
   (e) 3000000000
   (f) 100L
   (g) 32,767
   (h) 65432

3. Will the result of each of the following expressions be true or false? All variables are type `int`. Use the integer data values `k = 3`, `m = 9`, and `n = 5`.

   (a) `m == k * 3`
   (b) `k * (9 / k) == 9`
   (c) `k * (n / k) == n`
   (d) `k = n`

4. What will be stored in `k` or `f` by the following sets of assignments? Use these variables: `int h, k, m; float f; double g;`.

   (a) `f=1.6;  k = f;`
   (b) `f=1.4;  k = (int) f;`
   (c) `g=5.1;  f = (float) g;`
   (d) `g=9.6;  k = (float) g;`
   (e) `g=9.7;  k = g + 1.8;`
   (f) `h=13;  m=4;  f = (float) h / m;`
   (g) `h=13;  m=4;  f = (float)(h / m);`
   (h) `g=1.02;  f = 10.2  f == g * 10;`

5. Draw a parse tree for each of the following computations (include conversion boxes). Then use the given data values to evaluate each expression and record the final values stored in the variables `f`, `g`, and `k`. Use these declarations and initial values: `int k, j=70;  float  f=32.08;  double g=10.0;` .

   (a) `f = g * (int) f + j;`
   (b) `k = g * (int) f + j;`
   (c) `g = pow( f, 10.0);`
   (d) `g = pow( f, f);`

6. Draw a parse tree for each of the following computation (include conversion boxes). Then use the given data values to evaluate each expression and record the final values stored in the variables `J`, `L`, and `F`. Indicate if overflow occurs during an evaluation.

```
short int J, K=100;
long int L, M=2000;
float F;
```

   (a) `J = L = K * K * K;`
   (b) `L = M * M * M;`
   (c) `F = M * M * M;`

7. We can represent all integer values using the `double` representation. List two situations in which we would still want to use the `int` data type.

8. Given the variable declaration `double x = 1234.5678;`, what is printed by the following statements?

   (a) `printf( "%e %f %g", x, x, x );`
   (b) `printf( "%10.3e %10.3f %10.3g %10.5g", x, x, x, x );`

9. Given the following variable declarations and input prompt, what is stored in `k`, `m`, `x`, or `d` by the following statements when the user enters the number shown on the left of each item? (If the result is garbage, say so.)

```
short int k;
long int m;
float x;
double d;
printf( " Please enter a number: " );
```

   (a) 33 `scanf( "%hi", &k );`
   (b) 33000 `scanf( "%hi", &k );`
   (c) −44000 `scanf( "%li", &k );`
   (d) 33 `scanf( "%li", &m );`
   (e) 33 `scanf( "%g", &d );`
   (f) 109e−02 `scanf( "%lg", &d );`
   (g) 123.456789 `scanf( "%lg", &x );`
   (h) −43.21098765 `scanf( "%f", &x );`

10. Answer the following questions about the computer you use. Write a short program to find the answers, if necessary.

    (a) What is the largest `int` that you can enter on your machine and print correctly?
    (b) What is the biggest `unsigned int` you can read and write?
    (c) When is $x + 1 < x$?

11. Write one or a few lines of code that will cause integer overflow and wrap to happen.

12. Say whether each of the following computations will give a meaningful answer or is likely to cause overflow, underflow, or a serious precision error.

```
float f;
float g = 0.1
float h = cos(0); // This should be 1.0
```

    (a) `f = 0.000001 - pow(g, 5);`
    (b) `f = 233344455.5 * .1;`
    (c) `f = 233344455.5 + .1;`
    (d) `f = pow(3.14159, 100);`

13. When a floating-point number is printed in `%e` format, it is printed in normalized form, with exactly one digit to the left of the decimal point. Rewrite the following numbers in normalized scientific notation:

    (a) `75.23`
    (b) `.00012`
    (c) `.9998`
    (d) `32,767`

14. Each item that follows compares two numbers. For each, answer whether the result is `true`, `false`, or indeterminate and explain why. To get the correct answers, you must know about the type conversions used in mixed-type expressions.

```
float w = 3.3;
int j = w, k = 3;
double x = 3.0, y = 3.3, z = 4.2;
```

(a) `x == k`
(b) `y == k`
(c) `x != y`
(d) `w == j`
(e) `w == y`
(f) `x == w`
(g) `(float)x == w`
(h) `y == z * (y / z)`
(i) `x + 1.0 == k + 1`
(j) `x == .3 * 10`

7.8.2 Pencil and Paper

1. Draw a parse tree for the following computation (include conversion boxes). Then use the tree to evaluate the expression. Use these variable declarations and initial values: `int k, j=10; double g=402.5; float f=32.08;`.

   ```
   k = g - (int) f * j;
   ```

2. Given the variable declarations, what is printed by the following statements?

   ```
   int k = 1234;
   float x = 1681.700612;
   float y = 23.28765;
   ```

   (a) `printf( "k =%i\n", k );`
   (b) `printf( "k =%10i\n", k );`
   (c) `printf( "k =%-10i\n", k );`
   (d) `printf( "x = %10.3f \n", x );`
   (e) `printf( "x = %10.4f \n", x );`
   (f) `printf( "x = %10.4e\n", x );`
   (g) `printf( "x = %.3g\n", x );`
   (h) `printf( "y = %.3g\n", y );`

3. Given the following variable declarations and input prompt, what is stored in `m`, `x`, or `d` by the statements when the user enters the number shown on the left of each item? (If the result is garbage, say so.)

   ```
   long int m;
   float x;
   double d;
   printf( " Please enter a number: " );
   ```

   (a) 33000 `scanf( "%li", &m );`
   (b) −44000 `scanf( "%hi", &m );`
   (c) 76.5 `scanf( "%g", &x );`
   (d) 5.12e20 `scanf( "%Lg", &d );`
   (e) −3000000033 `scanf( "%li", &m );`
   (f) 5,000,000,033 `scanf( "%li", &m );`
   (g) −3000000033 `scanf( "%li", &d );`
   (h) 333222111000.9 `scanf( "%lg", &d );`

4. Which operation (integer division or real division) will be used to evaluate each of the following divisions? Assume that `h`, `k`, and `m` are type `int` while `x` is type `double`.

   (a) `k = h / 3;`
   (b) `k = 3.14 / m;`
   (c) `x = h / m;`
   (d) `k = h / x;`
   (e) `h = x + k / m;`
   (f) `h = k + x / m;`

5. Define the following and give an example of code that might cause it:

   (a) Integer overflow error
   (b) Floating-point underflow error
   (c) `NaN` error
   (d) Precision error

6. Show the output produced by the following program. Be careful about spacing.

```
#include <stdio.h>
int main(void)
{
   int i = 0;
   float x = 1.2959;

   while (i < 4) {
       printf( "%6.2e %6.2f %6.2g \n", x, x, x );
       x *= 10;
       i++;
   }
   return 0;
}
```

7. For each computation that follows, say whether overflow will occur if integers are 2 bytes long? If they are 4 bytes long?

```
int J, K=100, M=2000;
```

   (a) `J = K * K * K;`
   (b) `J = (float)K * K * K;`
   (c) `J = 30 * M / K * M;`
   (d) `J = M * M * M;`

8. Say whether each computation that follows will give a meaningful answer or is likely to cause overflow, underflow, or serious precision error.

```
float c = 80000;
float d = 1.0e-5;
float f;
```

   (a) `f = pow( c, 5 ) / d;`
   (b) `f = ceil( d ) * 2e-90;`
   (c) `f = d + sqrt( 100 * c );`
   (d) `f = sqrt( 10 * d );`

9. Show how the following normalized numbers would look when printed in `%.3f` format:

(a) `3.245E+02`
(b) `1.267E-03`
(c) `3.14E+04`
(d) `1.02E-03`

10. Several problems are associated with doing calculations with real values. Which of these do you believe occurs most often? Which of these, even if it does not occur often, causes the most trouble and why?

11. Something is wrong with the following tests for equality. For each item, explain why the answer will be different from the intended answer.

```
short int s=1, t=32767;
long int k=65536;
float w=3.3;
double x=3.3, y=33.0;
```

(a) `x == w`
(b) `x*10.0 == y`
(c) `s == (short)k`
(d) `32769 == (int)t + 2`

7.8.3 Using the Computer

1. Summation.

   A simple mathematical function can be defined by the equation

$$f(N) = \sum_{x=1}^{N} x \sin(x)$$

   as x increases from 1 to N degrees in 1-degree increments. This equation will sum N terms, each of which multiplies x times a value of the `sin()` function. Write a function with a parameter N that will print a table of the N terms and return the value of $f(N)$. Write a main program that will input a value for N, then call the function $f(N)$, and print the result. Check to make sure that the value of N is positive. If not, give the user another chance to enter a valid value, until it is proper. Remember that the `sin()` function requires the angle to be in radians rather than degrees.

2. Bridge hands.

   A bridge deck has 52 cards and a hand consists of 13 cards. Calculate the following facts about possible bridge hands. Use the information in Section 7.6.4 and Figure 7.30 as guidance.

   (a) How many possible different bridge hands are there? (Call this number H.)
   (b) How many hands include all four of the Aces in the deck? The formula is:

$$A = \frac{48!}{9! \times 39!}$$

   (c) What is the probability of receiving a hand that has all four aces? The formula is: A/H.
   (d) Do any of the above computations require special care when done with type `float`? Explain why or why not.

3. See your money grow.
   Assume you are loaning money to a friend, who will pay it back as a lump sum at the end of the loan period, with interest compounded monthly. Write a program that will allow you to enter an amount of money (in dollars), a number of months, and an annual interest rate. From these data, first calculate a monthly interest rate (1/12 of the annual rate). Then print a table with one line per month, showing the month number, the amount of interest your money will earn that month, and the total amount of

your investment so far after the interest is added. Print one line per month, from the time the loan is made until the time it is repaid. Print column headings and print all values in neat columns under them. Break your output into readable blocks by printing a blank line after every twelfth month. Be sure to test your program with a loan period greater than 12 months.

4. Loan payments.
   Compute a table that shows a monthly payback schedule for a loan. The principle amount of the loan, the annual interest rate, and the monthly payment amount are to be read as inputs. Calculate the monthly interest rate as 1/12 of the annual rate. Each month, first calculate the current interest = the monthly rate × the loan balance. Then add the interest amount to the balance, subtract the payment, and print this new balance. Continue printing lines for each month until the normal payment would exceed the loan balance. On that month, the payment amount should be the remaining balance and the new balance becomes 0. Print a neat loan repayment table following this format:

```
Payment schedule for $1000 loan
at 0.125 annual interest rate
and monthly payment of $100.00

Month    Interest    Payment    Balance
----------------------------------------
  1       10.42       100.00     910.42
  2        9.48       100.00     819.90
 ...        ...         ...        ...
 10        1.66       100.00      60.99
 11        0.64        61.63       0.00
----------------------------------------
```

5. Bubbles.

   The internal pressure inside a soap bubble depends on the surface tension and the radius of the bubble. The surface tension is the force per unit length of the inner and outer surface. The equation for the pressure inside the bubble relative to the air pressure outside is

$$P = \frac{4\sigma}{r} \quad (\text{lb/ft}^2)$$

   where σ is the surface tension (lb/ft) and r is the bubble radius (ft).

   Define a function, `bubble()`, that will compute the pressure P given a value of r and assuming the constant σ to be 0.002473 lb/ft. Then write a main program that will input a value for r, call the `bubble()` function to compute the pressure, convert the units of pressure from lb/ft^2 to psi (pounds per square inch, lb/in^2), and print the answer. Make sure that the input radius is valid; that is, greater than 0. Allow the user to continue entering values until the radius is valid.

6. How functions grow.

   Write a program that will ask the user to enter an integer, $Nmax$, then print a table like the following one, with $Nmax$ lines. If $Nmax$ is less than 1 or more than 20, print an error comment and ask the user to reenter $Nmax$. Store the result of all calculations in variables of type `int`. Use the `pow()` function in the math library to calculate 2^N. The `C` system will coerce the `double` result of `pow()` to an integer for you. Are all the results correct when you use $N = 20$? If not, why not?

```
 N    sum(1..N)     N squared     2 to the power N
--------------------------------------------------
 1        1             1                2
 2        3             4                4
 3        6             9                8
...      ...           ...              ...
```

7. Fibonacci numbers.

   A Fibonacci sequence is a series of numbers such that each number is the sum of the two preceding numbers in the sequence. For example, the simplest Fibonacci sequence is: 1, 1, 2, 3, 5, 8, 13, 21, ... In this sequence, the first two terms are, by definition, 1. Write a program to print the terms of this sequence in five columns, as follows:

```
0.  1       1.  1       2.  2       3.  3       4.  5
5.  8       6.  13      7.  21 ...
```

Hint: Consider having three variables in your loop, called `current`, `old`, and `older`. After computing the new current value, shift the old values from one variable to the next to prepare for the next iteration. Run your program and determine experimentally how many terms of the Fibonacci series can be computed on your machine before an overflow if you use variables of type `short`, `long`, `float`, and `double` to hold the results.

8. Square root.

   Over 2000 years ago, Euclid invented a fast, iterative method for approximating the square root of a number. Let N be a positive number and *est* be the current estimate of its square root. (Initially, let $est = N/2$.) At each step of the iteration, let $quotient = N/est$. If *quotient* equals *est*, they are the square root of N and the iteration should end. Otherwise, let the new *est* be the average of *quotient* and the old *est* and repeat the calculation until *quotient* equals *est* within some epsilon value. Print a table showing the iteration number and the current values of *est* and *quotient*. Let the user enter the values of N and epsilon. Then print the value calculated using the standard `sqrt()` function. Run your program several times with epsilon equal to 0.01, 0.001, 0.0001, and so on. Summarize your results in a neat chart with columns for epsilon, the approximation for $\sqrt{x}$, and the number of iterations needed to converge with that value of epsilon.

9. A table.

   Write a program that will ask the user to enter a real number, N, then print a table showing how certain functions grow as N doubles. For $N = 3.14$, the output should start thus:

```
              N            1/N          N * log(N)

 1            3.14    3.184713e-01    3.592860e+00
 2            6.28    1.592357e-01    1.153868e+01
...           ...          ...             ...
```

   Let all your variables be type `float`. Continue computing and printing lines until an underflow occurs in the column for $1/N$ *and* an overflow occurs in the last column. Use the `log()` function, which computes the natural log of a number, and use the constant `HUGE_VAL` from the `math` library to test for an overflow. Remember that a value becomes 0.0 when an underflow occurs.

Chapter 8

Character Data

This chapter is concerned with the character data type. We show how characters are represented in a program and in the computer; how they can be read, written, and used in a program; and how they are related to integers.

8.1 Representation of Characters

A character is represented using a single byte (8 bits). We have shown how numeric values (integers and reals) are represented using the binary number system. Characters also are represented by bits. However, when we think of a text character, a binary number is not the first thing that comes to mind. There is no obvious way in which numbers or bit patterns correspond to the letters, digits, and special characters on a keyboard. So people have invented arbitrary codes to represent these characters. The most common of these is the ASCII (American Standard Code for Information Interchange) **code**, which is listed in a table in Appendix **??**. Each ASCII character is represented by 7 bits,[1] which are stored on the rightmost side of a byte. You can think of the value of this byte either as a character or an integer.

The ASCII characters are listed in the appendix in numeric order, according to the value of their bit representations. We can find a character in the table and see its code or use a numeric code as an index into the table to determine the associated character. For historical reasons, the indexing number often is listed in two forms, decimal and hexadecimal.[2]

The ASCII codes from 33 to 126 represent printable characters; most of these are letters of the alphabet in upper or lower case. Note that each lower-case letter is 32 greater than the corresponding upper-case letter, for example, 'A' + 32 == 'a'. A single bit in the representation, called the *case bit*, makes this difference. This is illustrated in Figure 8.1

[1] International ASCII uses 8 bits to represent each character and Unicode uses 16 bits.
[2] The hexadecimal number system is discussed in Chapter 15 and in Appendix E.

In the ASCII code, upper and lower case letters differ by only one bit, the sixth when counting from the right. This bit has the binary value of 32.

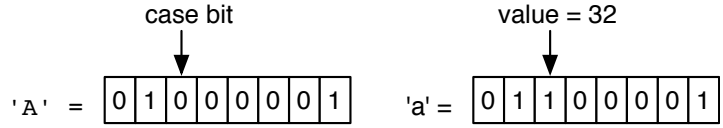


Figure 8.1. The case bit in ASCII.

Data type	Define Constant Limits	Range
`signed char`	`SCHAR_MIN..SCHAR_MAX`	$-128 \ldots 127$
`unsigned char`	`0..UCHAR_MAX`	$0 \ldots 255$
`char`	`CHAR_MIN..CHAR_MAX`	Same as `signed` or `unsigned char`

Figure 8.2. Character types.

8.1.1 Character Types in C

In reality, characters are just very short integers in C; the single byte of a character holds a number. Anything you can do with an integer, you can do with a character. Anything you can do with a character, you can do with 1 byte of an integer. *There is no difference between a character and an integer* except the number of bytes used and the format specifiers used to read and print the two types.

8.1.2 The Different Interpretations

Figure 8.2 lists the character types defined by the C standard. The type `signed char` is not used often, and when used, it is generally thought of as a very short `int`. The more common type, `unsigned char`, is useful primarily when a program must store large quantities of small positive integers in memory and conserve storage to avoid running out of it. In this case, the `unsigned char` actually is being used as a very short `unsigned int`. A program that does image processing is an example of such an application (see Chapter 18). The type `char` is used for most character-handling applications.

A character code such as ASCII uses a fixed number of bits to represent the letters of the alphabet, numerals, punctuation marks, special symbols, and control codes. ASCII uses 7 bits and therefore can represent 128 codes. The C standard permits type `char` to be defined either as signed or unsigned values; the compiler manufacturer makes that decision. Normally, it is of no concern to the programmer, since the index values for the ASCII table (0–127) are present in both forms, so there is not such a **portability** problem as there is with `int`. However, an international version of the ASCII code uses all 256 index values. The extra codes are used to represent additional letters and special symbols used in various European languages. In systems that use International ASCII, `char` is implemented as an unsigned type.

8.1.3 Character Literals

Most often **character literals** are written in C using single quotes, like this: `'A'`. However, nothing in C is simple. The character literal inside the quotes can be written two ways, as shown in Figure 8.3:

- If it is an ordinary printable character, we write it directly in quoted form. Thus, the first letter of the alphabet is written as `'a'` (lower case) or `'A'` (upper case).
- Some characters are written with an **escape code** or escape sequence. This consists of the `\` (escape) character followed by a code for the character itself. The predefined symbolic escape codes are listed in Figure 8.4, a few of which we already used in output formats.

Escape code characters are included in C for two different reasons: to resolve ambiguity and to provide visible symbols for invisible characters. Three of the escape codes, `\'`, `\\`, and `\"`, are used to resolve lexical and syntactic ambiguity. The backslash (also called *escape*), single quote, and double quote characters have special meaning in C, but we also need to be able to write and process them as ordinary characters. The escape character tells C that the following keystroke is to be treated as an ordinary character, not as an element of C syntax.

The other escape characters are invisible; their purpose is to cause a side effect. For example, the *attention* code, `\a`, is used to alert the user that something exceptional has happened that needs attention. (Note that `'\a'` and `'a'` are very different; the first means "attention" and should cause most computers to beep; the second is an ordinary letter.) A very important escape code is the *null character*, `\0`, which is used to mark the end of every character string and will be discussed further in Chapter 12.

One set of escape code characters that we use frequently are called **whitespace** *characters*; they affect the appearance of the text but leave no visible mark themselves. The list includes newline, `\n`; return, `\r`;

Character constants can be written symbolically or numerically. The symbolic form is preferable because it is portable; that is, it does not depend on the particular character code of the local computer.

Meaning	Symbol (Portable)	Decimal Index (ASCII Only)
The letter *A*	`'A'`	65
Blank space	`' '`	32
Newline	`'\n'`	10
Formfeed	`'\f'`	12
Null character	`'\0'`	0

Figure 8.3. Writing character constants.

horizontal tab, `\t`; vertical tab, `\v`; formfeed, `\f`; and the ordinary space character. The two tab characters insert horizontal spaces or vertical blank lines into the output (the precise number of horizontal or vertical spaces depends on the system). Whitespace characters often are treated as a group in C and handled specially in a variety of ways.[3] Note that whitespace characters are all invisible, but many invisible characters, including null and attention, are not classified as "whitespace".

8.2 Input and Output with Characters

Character input and output can be performed using the standard `scanf()` and `printf()` functions. In addition, other special functions exist just for characters. Some of these functions have subtle difficulties associated with them, which we discuss.

8.2.1 Character Input

The standard library functions for reading characters are

1. `getchar()`. This function has no parameters. It reads a single character of input and returns it as an `int`. The character is stored in the rightmost part of that `int`, and bytes to the left of the character are filled with **padding** bits. When the padded value is stored in a character variable, the padding is discarded. This process of adding and stripping off padding is automatic, and transparent, and can be ignored by beginning programmers. Normally, the value returned by `getchar()` is used in an assignment statement.

   Example: `ch = getchar();`

2. `scanf()` *with a* `"%c"` *conversion specifier*. In this format, there is *no space* between the opening quotation mark and the `%c` specifier. This will read the next input character, whether or not it is whitespace, and store it in the address provided. This version is equivalent to using the `getchar()` function.

   Example: `scanf( "%c", &ch );`

[3]These ways will be explained as they become relevant to the text.

Code	Meaning	Code	Meaning	Code	Meaning
`\0`	Null	`\a`	Attention	`\"`	Double quote
`\n`	Newline	`\b`	Backspace	`\'`	Single quote
`\r`	Return	`\f`	Formfeed	`\\`	Backslash (escape)
`\t`	Horizontal tab	`\v`	Vertical tab		

Figure 8.4. Useful predefined escape sequences.

3. `scanf()` *with a* `" %c"` *conversion specifier.* In this format, there is a space in the format string before the `%c` specifier. The space causes `scanf()` to skip leading whitespace (if any exists) before reading a single nonwhitespace character and storing it in the address provided. This is similar to the manner in which other data types are scanned.

   Example: `scanf( " %c", &ch );`

Keyboard input is buffered. Whether you are entering numeric or character data into a program, your input is not sent immediately to the program. Until you hit the Enter key, it is displayed on the screen but remains in a holding tank called the **keyboard buffer** so that you can inspect and change it, if necessary. It is not the case that as soon as you type a character the program will read it and begin processing. Some languages provide this feature, but C is not one of them.[4] After you hit Enter, your input moves to another area called the **input buffer** and becomes available to the program. The program will read as much or as little as called for by the `scanf()` or `getchar()` statement. Unread data remain in the input buffer and will be read by future calls on `scanf()` or `getchar()`.

Whitespace characters complicate input. When reading integers or floating-point numbers, `scanf()` skips over leading whitespace characters and starts reading with the first data character. However, with the `"%c"` specifier, leading whitespace is not ignored. If the first unread character is whitespace, that is what the system reads and returns. The function `getchar()` does the same thing. It reads a single character, which might be whitespace. Reading data in this manner leads to surprising behavior if the input contains unexpected whitespace characters such as `\n` and `\t`. Since these are not visible, it is easy to forget that they may be present.

In any text-processing program, it is frequently necessary to skip over indefinite amounts of whitespace. As mentioned previously, the behavior of `scanf()` can be changed by adding a single blank to the format: `" %c"`. The space inside the quotes and before the `%c` tells `scanf()` to skip over leading whitespace, if any exists. However, there is no way to force `getchar()` to skip over these invisible characters. For this reason, we usually use `scanf()` rather than `getchar()` to input single characters interactively.

8.2.2 Character Output

Character output is relatively straightforward. The `stdio` library provides two ways to display or print a single character:

1. `putchar()`. When only one character of output is needed, `putchar()` is the easiest way to do the job; we simply pass it the character we want to display. For example, to move the screen cursor to the beginning of the next line, we might say `putchar( '\n' );`. Note that the `putchar()` function does not automatically move the cursor to the next line as `puts()` does.
2. `printf()` *with a* `%c` *conversion specifier.* When printing a character mixed in with other kinds of data, we use `printf()` with the `%c` format specifier. Example:

   ```
   printf( "Child is %c, %i years old.", gender, age );
   ```

   Of course, a specific (nonvariable) character can be included in the format string itself. As with the other data types, it is possible to specify a field width between the `%` and the `c`, and the printed character will be right or left justified in the field area, depending on the sign of the width specifier.

Since characters *are* integers, it is legal to read or print them as integers. When you read an integer that is the ASCII code of a letter and print that number using a `"%c"` format or `putchar()`, you see the letter. Conversely, when you read a letter and print it using a `"%i"` format, you see a number.[5] This technique is demonstrated in Figure 8.5.

Notes on Figure 8.5. Printing the ASCII codes.

[4]Some old PC-based single-user C systems support the unbuffered character input functions `getch()` and `getche()`, in a library named `conio`. When using `getch()`, any character that the user typed would go directly to the program, rather than being held in the keyboard buffer until a newline was typed. This seems like a convenient function and it was popular with students. It is not supported by the standard because modern systems are multi-processing systems and cannot make a direct connection between any input device and any one process among the set that is running concurrently. We recommend against using any nonstandard feature because it is not portable.

[5]It also is possible to print the hexadecimal form of the character's index by using a `%x` conversion specifier. See Chapter 15.

```
#include <stdio.h>

int main( void )
{
    char ch;

    puts( "\n Demo: Printing the ASCII codes." );
    printf( "\n Please type a character then hit ENTER: " );
    ch = getchar();

    printf( " The ASCII code of %c is %i \n\n", ch, ch );
}
```

Figure 8.5. Printing the ASCII codes.

First box: declaration. We declare a `char` variable. In the remaining code, we use it to perform both character and integer output.

Second box: character input. We read a character (one keystroke); its ASCII code is stored as a binary integer in the `char` variable. The character is not read until the Enter key is pressed.

Third box: output. We print the input character twice: first as a character, using `%c`; then as an integer, using `%i`. Sample program output looks like this:

```
Demo: Printing the ASCII codes.

Please type a character then hit ENTER: A
The ASCII code of A is 65
```

8.2.3 Using the I/O Functions

The program in Figure 8.6 illustrates the use of the four character input and output functions and demonstrates how a simple program can produce very confusing output if the problem of whitespace in the input is not addressed.

Notes on Figure 8.6. Character input and output.

First box: character output.

- The function `putchar()` prints a single character. Its argument can be a character variable, a literal character, or an integer. If the argument is an `int`, the rightmost byte of its value is interpreted as an index for the ASCII code table, and the character in that position is printed. The command `putchar( 42 )` prints an asterisk because 42 is the ASCII code for `*`.
- We also can print a single character using `printf()` with `%c`; in this case, we print a dollar sign. If we try to print an integer with a `%c` conversion specifier, we see the character that corresponds to that integer's index in the code table. For example, the output from `printf( "%c", 42 )` would be an `*`.

Second box: reading the first response.

- The line `scanf( "%c", ... )` reads a single character of input.
- When a program begins executing, the input buffer is empty. Normally, the user will not enter any input until prompted, so the user's response to the first prompt will be the only thing in the input buffer. The first character of that response will be read, while the newline character generated by the Enter key will remain in the buffer. We presume that the user will type `y`, causing control to enter the loop.

Third box: the loop.

- In this loop, we "give" $5.00 to the user and prompt for another response:

  ```
  Here is $5.00

  Do you need more (y/n)?
  ```

We demonstrate various ways to read and write single characters and how a whitespace character in the input can cause unexpected results.

```
#include <stdio.h>

int main( void )
{
    char input;
    char money = '$';
    char star  = '*';

    putchar( '\n' );  putchar( star );  putchar( 42 );   putchar( '\n' );
    printf( " Do you need %c (y/n)? ", money );

    scanf( "%c", &input );

    while (input == 'y') {
        printf( " Here is %c5.00\n", money );
        printf( "\n Do you need more (y/n)? " );
        input = getchar();   /* This code is wrong!  Use scanf( " %c", &input ) */
    }

    printf( " OK --- Bye now.  %c \n", '\a' );
}
```

Figure 8.6. Character input and output.

- This time we use `getchar()` to read the next input character. If it is `y`, we will stay in the loop; for any other input (including whitespace), we leave the loop.
- This logic seems simple enough, but *it does not work*. As shown in the following output, the user sees the second prompt but the program quits and says goodbye without giving that user a chance to enter anything. The reason for this problem is explained in the following section.
- The complete output for this run is

```
**
Do you need $ (y/n)? y
Here is $5.00

Do you need more (y/n)?  OK --- Bye now.
```

Fourth box: the closing message. The `%c` "prints" the escape character `\a`. On some systems, if the computer has a sound generator and the volume is turned up, you should hear a ding when you print the attention character, but you see nothing. On other systems, the output may be visible (for instance, a small box) but not audible.

Problems with `getchar()`. A common programming error was illustrated in Figure 8.6. After `scanf()`, the program initially performs as expected; if the input is `y`, it enters the loop and "gives" the user $5.00. Then the loop prompts the user to enter another `(y/n)` response. However, when the second prompt is displayed, the system does not even wait for the user to respond; it simply quits. Why?

When entering the answer to the first question (above the loop), the user types `y` and hits the Enter key. This puts the character `y` and a newline character into the input buffer. The newline character is necessary because, in most operating systems, the system does not send the keyboard input to the program until the user types a newline. The `scanf()` above the loop reads the `y` but leaves the `\n` in the buffer. At the end of the first time around the loop, that `\n` still is sitting in the input buffer, unread. The loop prompts the user for a

choice, but the program does not wait for a key to be hit because input already is waiting. The `getchar()` then reads the `\n`, emptying the buffer. Since `\n` is not equal to `y`, the loop ends and the program says goodbye.

Now, if the user had typed `yyy` followed by a newline instead of a single `y` at the first prompt, the characters waiting in the input buffer would have been `yy\n`. The input to the second prompt then would have been `y`, and the program would have given the user another $5.00 bill. Altogether, the user would get three bills before the program could read the newline and leave the loop, all with no further typing by the user. The resulting output would be

```
**
Do you need $ (y/n)? yyy
Here is $5.00

Do you need more (y/n)?  Here is $5.00

Do you need more (y/n)?  Here is $5.00

Do you need more (y/n)?  OK --- Bye now.
```

Using `scanf( "%c", &input )` in place of `input = getchar()` does not solve the problem, because `scanf()` with `"%c"` works the same way as `getchar()`. However, we can solve this whitespace problem by using a single space in the format for `scanf()`. Replace the call on `getchar()` in Figure 8.6 by this call on `scanf()`:

$$\underbrace{\texttt{scanf(" \%c",}}_{\text{Note space in format}}\ \texttt{\&input);}$$

With this change, everything will work as intended: The program will query the user, wait for a response every time, and do the appropriate thing. A typical output would look like this:

```
**
Do you need $ (y/n)? y
Here is $5.00

Do you need more (y/n)? y
Here is $5.00

Do you need more (y/n)? n
OK --- Bye now.
```

If the user initially were to type `yyy`, the output would begin as shown earlier, but after three times through the loop, the user would have a chance to enter responses again.

8.2.4 Other ways to skip whitespace.

Skipping whitespace by inserting a space into a format specifier is a little-known technique, even though it is easy to do and easy to understand. A common mistake among programmers is to use the C function `fflush()` to do this task. The C standard sates that *the function* `fflush()` *is defined only for output streams*, not for input. Sometimes it seems to work for input, but it does so for indirect and subtle reasons, and only in some implementations. A correct alternative technique for skipping whitespace is to use a simple loop, as shown in Figure 8.7.

Notes on Figure 8.7. A function for skipping whitespace during keyboard input.

Line 1: The variable declaration. We declare an `int` variable to store the character being read. Although this seems wrong, remember that the function `getchar()` returns an `int`, not a `char`.

```
void skip_ws( void ) {
    int ch;                          // Most recently read character.
    while (isspace( ch=getchar() ); // Tight loop; exit when non-space is read.
    ungetc( ch, stdin );             // Put non-space character back into stream.
}
```

Figure 8.7. A function for skipping whitespace.

Line 2: Reading and testing the data.

- The function `isspace()` is explained in Section 8.3.5. Briefly, it returns `true` if its argument is a white-space character, `false` otherwise.
- This line is a "tight loop"; it has no body at all and does nothing except read and test, read and test, until the input is non-whitespace.
- This loop will work correctly for any combination and any number (zero or more) of whitespace characters. It will just keep reading until the user types something else. (Remember that many invisible characters are not classified as "whitespace".)
- We store the input character in a variable because we will use the final character read after the end of the loop.

Line 3: Get and give back. We leave the `getchar()` loop when a non-whitespace character is found. However, that is too late! That character is real input and cannot be processed in this function. The easiest remedy is to put that character back into the input stream so that it can later be read by the proper part of the program. Happily, `C` supplies a function for this task: `ungetc()`. The arguments in this call are the character that must be put back into the input stream, and the name of the input stream. This function will be explained more fully in the chapter about streams and files, Chapter 14.

A useful tool. This is a function that will often be useful in programs that analyze character input. Copy the definition into your personal file of C-tools.

8.3 Operations on Characters

8.3.1 Characters Are Very Short Integers

The basic operations defined for characters are the same operations that are defined for integers because, technically, characters *are* integers in the range $0 \ldots 255$ or $-128 \ldots 127$. Some kinds of data (such as digital images) are composed of a very large number of very small integers. In such cases, it is useful to minimize the amount of storage occupied by the data, so the data are stored as type `char` or `unsigned char` rather than `short int` or `int`. In such applications, it is important that all the integer operations can be applied to variables of type `char`.

The common use of type `char`, however, is to represent characters. Even then, many integer operators are useful; These are summarized in Figure 8.8. A little caution is warranted here; some integer operations are legal but not useful with characters. For example, it makes no sense to multiply or divide one character by another. Unfortunately, useless or not, the compiler will not identify such expressions as errors. In addition to the basic integer operators, there is also a set of functions in the `ctype` library that can manipulate character values. We now examine some of the library functions and take a closer look at the operators in Figure 8.8.

8.3.2 Assignment

We have seen that the values of both character and integer variables can be assigned to a `char` variable. As long as the integer value is not too big, everything will be fine (large values lead to overflow). Automatic coercion will shorten the integer and a single byte will be assigned. Literal characters also can be assigned to `char` variables. The columns of Figure 8.3 show two different values that will assign the same character code to a variable.

Operation	Meaning and Use
`char c1, c2;`	Declare two character variables
`c1 = c2;`	Copy the value of `c2` into `c1`.
`c1 == c2`	Do `c1` and `c2` contain the same letter?
`c1 != c2`	Do `c1` and `c2` contain different letters?
`c1 < c2`	Does `c1` come before `c2` in alphabetical order? The `<=`, `>`, and `>=` operators also are defined for characters.
`c1 + 1`	The letter that follows the value of `c1` in the alphabet.
`c1 - 1`	The letter that precedes the value of `c1` in the alphabet.
`++c2;`	Change `c2` from its current value to the next letter in the alphabet. All four increment and decrement operators are defined for characters.
`c2 - c1`	Assuming that `c2 > c1`, this is the number of letters in the alphabet between `c1` and `c2`. If `c2` is the ASCII code for a base-10 digit, then `c2 - '0'` is the numeric value of that digit.

Commonly used character operations are listed here. The phrase *alphabetical order* used here means "the order defined by the ASCII code or whatever code is in use on the local hardware." This normally is an extension of ordinary alphabetical order to include all of the characters in the local character set.

Figure 8.8. Character operations.

8.3.3 Comparing Characters

The operators `==` and `!=` are used to test whether two characters are equal. These operators are straightforward and portable; that is, they work identically on all systems. The other four comparison operators, `<`, `>`, `<=`, and `>=`, also are useful for characters, but their results can vary because they depend on the local computer system.

ASCII and International ASCII are the two codes used most commonly in personal computers today, but some systems use different underlying character codes. The particular code in use on the local system determines the "alphabetical order" on that system. (The technical terms for alphabetical order are **collating sequence** and **lexical order**.) Numerals and letters of the English alphabet are arranged in the usual order in most codes, but the special symbols may be arranged in arbitrary and incompatible ways. Therefore, two dissimilar machines might produce different results for some character comparisons.

8.3.4 Character Arithmetic

The operators `+`, `-`, `++`, and `--` are used in **character arithmetic** to compute the next (or prior) letters of the alphabet. The operator `-` can be used to determine how far apart two letters are in the alphabet. All these operations are useful for text processing programs and all depend on the collating sequence of the machine.

8.3.5 Other Character Functions

One of the standard C libraries is the character processing library, whose header file is `ctype.h`. This library contains a group of functions essential to a system programmer, including ones to test whether a character is in a particular set, such as the alphabet, as well as certain transformation routines. Several of these functions are frequently useful, even in simple programs:

1. `isalpha()`. This function takes one argument, a character. If the character is an alphabetic character (A ... Z or a ... z), the value `true` (1) is returned; otherwise, `false` (0) is returned.
2. `islower()`. This function takes one argument, a character. If the character is a lower-case alphabetic character (a ... z), the value `true` (1) is returned; otherwise, `false` (0) is returned.
3. `isupper()`. This function takes one argument, a character. If the character is an upper-case alphabetic character (A ... Z), the value `true` (1) is returned; otherwise, `false` (0) is returned.

4. `isdigit()`. This function takes one argument, a character. If it is a digit (0 . . . 9), `true` (1) is returned; otherwise, `false` (0) is returned.
5. `isspace()`. This function takes one argument, a character. If the character is a whitespace character (space, newline, return, formfeed, horizontal tab, or vertical tab), `true` is returned; otherwise, `false` is returned. We use `isspace()` to help analyze input data so that the results will be the same whether the user types spaces, newlines, or tab characters.
6. `tolower()`. This function takes one argument, a character. If the character is an upper-case alphabetic character (between A and Z), the return value is the corresponding lower-case character (between a and z). If the argument is anything else, it is returned unchanged. This function sets the case bit to 1.
7. `toupper()`. This function is the opposite of `tolower()`. If the argument is a lower-case character, the return value is the corresponding upper-case character. If the argument is anything else, it is returned unchanged. This function sets the case bit to 0.

When an input buffer might contain an unpredictable sequence of input items, a program can use the functions `isalpha()`, `islower()`, `isupper()`, `isdigit()`, and `isspace()` to analyze each input character and handle it appropriately. This makes it easier to build a well-designed user interface.

The functions `toupper()` and `tolower()` often are used in conjunction with testing character input, so that it does not matter whether the user types an upper-case or lower-case character. The programmer chooses one case to use for processing (often, it does not matter which) and converts all input characters to that case. By doing so, the logic of the program can be simplified. For example, suppose that a program needs to read a character into a variable named `ch` and use it to select one action from a set of actions. We could write the code like this:

```
scanf( " %c", &ch );

if (ch == 'x' || ch == 'X') {       /* X actions here */ }
else if (ch == 'y' || ch == 'Y') { /* Y actions here */ }
else if (ch == 'z' || ch == 'Z') { /* Z actions here */ }
else {                                /* default actions */ }
```

By using a case-shift function, we can eliminate one comparison in each test:

```
scanf( " %c", &ch );
ch = tolower( ch );

if (ch == 'x') {       /* X actions go here */  }
else if (ch == 'y') {  /* Y actions go here */  }
else if (ch == 'z') {  /* Z actions go here */  }
else {                 /* default actions go here */  }
```

8.4 Character Application: An Improved Processing Loop

This example combines the use of `scanf( " %c", ... )` with `toupper()`, `tolower()`, and a `switch` with character case-labels.

In Chapter 6, Figure 6.11, we demonstrate how a process can be repeated using a query loop until the user chooses to quit. Using the codes 1 to continue and 0 to quit is adequate, but it is not good human engineering and not customary. Now that we have shown how to read a single input character and how to force that character into a particular case, it is possible to improve the human interface by giving the more usual prompt: `Do you want to continue (y/n)?` and accepting either an upper-case or lower-case answer. The next program implements this improved interface.

The `work()` function, in Figure 8.10, is a simple application that computes the area of a regular polygon or circle. It prompts the user to select the kind of polygon from a menu and uses `tolower()` with a `switch` and character case labels to process that choice.

Notes on Figure 8.9. Improving the workmaster.

First box: the `#include` ***statements.*** In addition to `<stdio.h>`, we must include `<ctype.h>` because we are using the character-function library.

We improve the user interface of the main program from Figure 6.11 by permitting a `y/n` or `Y/N` response to the question, "Do you want to continue?" This program calls the `work()` function in Figure 8.10.

```
#include <stdio.h>
#include <ctype.h>    /* For toupper() and tolower(). */

void work( void );
double circle_area( double diam ) { return 3.1416 * diam * diam / 4; }
double square_area( double side ){ return side * side; }
double triangle_area( double side ) { return side * side / 4.0 * sqrt( 3 ); }

int main( void )
{
    char more;             /* repeat-or-stop switch */

    puts( "\n Calculate the area of a regular figure." );

    do { work();
         puts( "\n Do you want to continue (Y/N)? " );
         scanf( " %c", &more );
         more = toupper( more ) ;
    } while (more != 'N');
    return 0;
}
```

Figure 8.9. Improving the workmaster.

Second box: three functions. These functions perform the area computations for three different shaped figures. They are called from the `work()` function in Figure 8.10.

Third box: the `char` ***variable.*** We use a character variable rather than an integer to store the user's quit-or-continue response.

Fourth box: the repetition loop. We change the `do...while()` termination condition to test for the letter `'N'` instead of the number 0. This is more natural for the user. However, to make this test work reliably, we need to change two things in the inner box. Either response, `'N'` or `or 'n'` will cause the loop to end; any other response will permit it to continue.

Inner box: reading input. To read the input, we use a `scanf()` format that will skip whitespace in the input buffer. This is important in a loop that controls a `work()` function, because the input operations performed by that function usually leave whitespace (at least one newline character) in the input buffer.

Innermost box: case conversion. Even when instructions call for a `Y` or `N` response, many users will often type `y` or `n` instead. Good human engineering dictates that the program should accept upper-case and lower-case letters interchangeably. We achieve this by using `toupper()` to force the response into upper case. This permits us to make a simple check for an upper-case response instead of the more complex test for either a lower-case or upper-case letter. To achieve the same result without `toupper()`, we would have to write the loop test as

```
while (more != 'n' && more != 'N').
```

Notes on Figure 8.10. Using characters in a switch.

First box: the menu. We display a simple menu and prompt the user for a choice. When we read the input character, we use a space in the format to skip over the carriage return character left in the input stream by `main()`.

Second box: using `tolower()` ***in a*** `switch`**.** We use `tolower()` here so that the user can enter either upper-case or lower-case choices. This line could be written thus: `switch (ch)` . However, if it were written without the call on `tolower()`, the following case labels would need to be more complex.

This function is called from `main()` in Figure 8.9. Code from both Figures should be in the same file.

```
void work( void )
{
    char ch;      /* Length of one side or of the diameter */
    double x;     /* Length of one side or of the diameter */
    double area;  /* Area of the figure */

    printf (" Enter the code for the shape you wish to calculate:  \n" );
    printf (" C Circle\n S Square\n T Equilateral triangle\n > " );
    scanf( " %c", &ch );

    switch (tolower( ch ))
    {
      case 'c':
        printf( " Enter the diameter of the circle:  " );
        scanf( "%lg", &x );
        area = circle_area( x );
        printf( " The area of this circle = %.2f\n", area );
        break;
      case 's':
        printf( " Enter the length of one side of the square:  " );
        scanf( "%lg", &x );
        area = square_area( x );
        printf( " The area of this square = %.2f\n", area );
        break;
      case 't':
        printf( " Enter the length of one side of the triangle:  " );
        scanf( "%lg", &x );
        area = triangle_area( x );
        printf( " The area of this triangle = %.2f\n", area );
        break;
      default:  printf( "%c is not a meaningful choice.  Try again.", ch );
    }
    return 0;
}
```

Figure 8.10. Using characters in a switch.

Third box: the* `case` *label. If the call on `tolower()` were omitted in the second box, we would need to write two case labels for each case, like this: `case 'c': case 'C':`

Fourth box: the* `case` *actions. We perform all of the actions needed for a circle: input, calculation using the appropriate function, and output. Of course, a `break` statement must end the sequence. It is good style to use functions to do much or most of the work for each case, so that the entire `switch` statement fits on one computer screen.

Fourth box: the default. This case traps illegal menu choices. You can see the result in the last block of output, below.

‘

Output. A sample of the output follows. Note that whitespace and case differences are ignored, and that the invalid response to the second query causes the process to continue, not quit.

```
Calculate the area of a regular figure.
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > c
   Enter the diameter of the circle: 10
   The area of this circle = 78.54

Do you want to continue (Y/N)? :      y
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > t
   Enter the length of one side of the triangle: 3.5
   The area of this triangle = 5.30

Do you want to continue (Y/N)? : t
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > s
   Enter the length of one side of the square: 10
   The area of this square = 100.00

Do you want to continue (Y/N)? : y
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > w
   w is not a meaningful choice.  Try again.

Do you want to continue (Y/N)? : n
```

8.5 What You Should Remember

8.5.1 Major Concepts

- ASCII is a character code. It uses 7 bits to represent the set of 128 characters that are part of the code. International ASCII is an 8-bit code that represents 256 characters. These codes are used with the `char` data type.
- Character literals are written between single quotemarks.
- Escape codes are used to write literals for invisible characters.
- There are several whitespace characters, including space, horizontal and vertical tabs, and newline.

8.5.2 Programming Style

Escape codes. Most ASCII characters are printable characters; that is, they leave a visible mark when displayed on a video screen or a printer. These characters correspond to keys on a typical computer keyboard. Some keys, such as the space bar, the Tab key, and the Enter key, do not represent printable characters but are used for their effect on the printed text. These, called *whitespace* characters, are represented in a C program by symbolic escape codes. There also are nonprintable ASCII characters that have no symbolic escape codes; they are used infrequently but may be referenced, if necessary, by using the underlying value. To be sure that your program is portable, use only the literal form of a character or a symbolic code.

Avoiding errors. Use character processing for a better human interface, like that in the revised `work()` function. Use functions like `toupper()` and `tolower()` to handle both upper-case and lower-case responses.

8.5.3 Sticky Points and Common Errors

`char` ***vs.*** `int`**.** Technically, characters are very short integers in C. Conceptually, though, they are a separate type with separate operations and different methods for input and output. Be sure not to do meaningless operations like multiplying two characters and avoid potentially nonportable operations like `<`. Every C implementation uses the character code built into the underlying hardware. For most modern machines, that code is either International ASCII or ASCII, which is given in Appendix A.

Character input. Whitespace can be a confusing factor when doing character input. The `scanf()` input conversion process for numeric types automatically skips leading whitespace and starts storing data only when a nonwhitespace character is read. However, `getchar()` returns the first character, no matter what it is; and `scanf()` with a `"%c"` does the same thing. To skip leading whitespace, you must use `scanf()` with a `" %c"` specifier (a space inside the format and before the percent sign). If this space is omitted, the program is likely to read whitespace and try to interpret it as data, which usually leads to trouble. Therefore, a programmer must have a clear idea of what he or she wishes to do (read whitespace or skip it) and choose the appropriate input mechanism for the task.

8.5.4 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

character literal	lexical order	padding
escape code	collating sequence	character arithmetic
portability	input buffer	improved `work()` function
ASCII code	whitespace	`switch` with `char` cases

The following C keywords, functions, and symbols are discussed in this chapter:

`\n` (newline)	`getchar()`	`isalpha()`
`\r` (return)	`putchar()`	`isupper()`
`\b` (backspace)	`printf()`	`islower()`
`\t` (horizontal tab)	`scanf()`	`isspace()`
`\v` (vertical tab)	`ungetc()`	`isdigit()`
`\f` (formfeed)	`"%c"` conversion	`tolower()`
`\a` (attention)	`" %c"` conversion	`toupper()`
`\0` (null character)	`ctype.h`	

8.6 Exercises

8.6.1 Self-Test Exercises

1. Explain the difference between `'6'` and `6`.
2. What is a whitespace character? List three of them. What is an escape code character? List three of them.

3. Show the output produced by the following program. Be careful about spacing.

```
#include <stdio.h>
int main( void )
{
    for (int k = 1; k <= 5; k += 2) {
        printf( "    %i:", k );
        putchar( '0'+k );
    }
    putchar( '\n' );
}
```

4. What will be stored in `k`, `c`, or `b` by the following sets of assignments? Use these variables:

```
int k;    char c, d;   int b;
```

   (a) `d = 'b';  c = d+1;`
   (b) `d = 'b';  c = d--;`
   (c) `d = 'E';  c = toupper( d );`
   (d) `d = '7';  k = d - '0';`
   (e) `b = isalpha( '@' );`
   (f) `b = 'A' == 'a';`

5. What will be stored in each of the variables by the input statements on the right, given the line of input on the left. If the combination is an error, say so. Use these variables:

```
int k;   char d;
```

   (a) `a` `scanf( "%c", &d);`
   (b) `66` `scanf( "%c", &d);`
   (c) `70 C` `scanf( "%i%c", &k, &d);`
   (d) `F` `scanf( "%i", &k);`
   (e) `go!` `d = getchar();`
   (f) `\n` `scanf( "%c", &d);`

6. What is the output from the following program if the user enters `Z` after the input prompt?

```
#include <stdio.h>
int main( void )
{
    char ch;
    printf( "\n Type a character and hit ENTER: " );
    ch = getchar();
    printf( "%3i %c \n ", ch, ch );
    putchar( ch ); putchar( '\n' );
}
```

8.6.2 Using Pencil and Paper

1. What will be stored in `k`, `c`, or `b` by the following sets of assignments? Use these variables:

```
int b, k;    char c, d;
```

(a) `d ='A';  k = d;`
(b) `d ='c';  c = toupper( d );`
(c) `d ='@';  c = tolower( d );`
(d) `k = 66;  c = k-1;`
(e) `b = isupper( 'A' );`
(f) `b = 'A' < 'a';`

2. What will be stored in each of the variables by the input statements on the right, given the line of input on the left. If the combination is an error, say so. Use these variables:

```
int k, m;  char c, d;
```

(a) `a` `scanf( "%c", &k);`
(b) `126` `scanf( "%c", &d);`
(c) `70 D` `scanf( "%i %c", &k, &d);`
(d) `70 71` `scanf( "%i%i", &k, &m);`
(e) `U2` `scanf( "%c%i", &d, &k);`
(f) `I O` `c = getchar();  d = getchar();`

3. Without running the following program, show what the output will be. Use your ASCII table.

```
#include <stdio.h>

int main( void )
{
    int upper = 65;
    int lower = upper + 32;
    int limit = 26;
    int step = 0;

    puts( " Do you read me?" );
    while ( step < limit ) {
        printf( "%2i.   %c  %c\n", step, upper, lower );
        ++upper;
        ++lower;
        ++step;
    }
    printf( "=========\n" );
}
```

4. What is the output from the following program if the user enters 80 after the input prompt?

```
#include <stdio.h>
int main( void )
{
   int k;
   printf( "\n Enter a number 65 ... 126: " );
   scanf ( "%i", &k );
   printf( " %3i %c \n", k, k );
   putchar( k ); putchar( '\n' );
}
```

5. Write a code fragment to compare two character variables and print `true` if both are alphabetic and they are the same letter, except for possible case differences. Print `false` otherwise.

8.6.3 Using the Computer

1. Character manipulation practice.

   Write a program to prompt the user once for a series of characters. Read the characters one at a time using `scanf( " %c", ...)` in a loop. Generate the following output, based on the input, one response per line:

   (a) Echo the input character.
   (b) Call `isalpha()` to find out whether it is alphabetic.
   (c) If so, call `toupper()` to convert it to an upper-case letter and print the result. If not, print an error comment.
   (d) If the character is a period, print a statement to that effect and quit.

2. Volumes.

   Write a program to calculate volumes. Start by writing three double→double functions for these three geometric shapes:

   (a) `cylinder()`. The single argument, d, is the height of a cylinder and also the diameter of its circular base. Calculate and return its volume. The formula is: $volume = \pi \times d^3/4$
   (b) `cube()`. The argument is the length, s, of one side of a cubical box. Calculate and return its volume. The formula is: $volume = s^3$
   (c) `sphere()`. The argument is the diameter, d, of a sphere. Calculate and return its volume. The formula is: $volume = (4/3) \times \pi \times d^3/8$

   Write a program that will permit the user to compute the area of several shapes. Use a menu and a switch to process the menu selections. Include a menu item "Q Quit", and use this instead of a uery loop to end the program.

   Read the letter in such a way that whitespace does not matter. Test the input in such a way that upper-case and lower-case differences do not matter. If the letter is `q`, terminate the program. Otherwise, read and validate a real number that represents the size of the figure. If this length is 0 or negative, print an error comment. If it is positive, call the appropriate area function and print the answer it returns. If the letter entered is not `c`, `s`, `t`, or `q`, print an appropriate error message.

3. Palindromes.

   This program will test whether a sentence is a palindrome; that is, whether it has the same letters when read forward and backward. First, prompt the user to enter a sentence. Read the characters one at a time using `getchar()` until a period appears. As they are read,

   (a) Echo the input character.
   (b) Call `tolower()` to convert each character to lower case.
   (c) Count the number of characters read (excluding the period).
   (d) Store the converted character in the next available slot in an array.

   When a period appears, start from both ends of the array and compare the letters. Compare the first to the last, the second to the second-last, and so forth. If any pair fails to match, leave the loop and announce that the sentence is not a palindrome. If you get to the middle, stop, and announce that the input is a palindrome. Assume that the input will be no more than 80 characters long.

4. Ascending or descending.

   Your program should read three numbers and a letter. If the letter is , 'A' or 'a', output the numbers in ascending order. If it is 'D or 'd', output the numbers in descending order. For any other letter, give an error message and output them in the opposite order that they were read in..

5. Vowels.

   Prompt the user to enter a sentence, then hit newline. Read the sentence one character at a time, until you come to the newline character. Count the total number of keystrokes entered, the number of alphabetic characters, and the number of vowels (`'a', 'e', 'i', 'o',` and `'u'`). Output these three counts.

6. Tooth fairy time.

   This program will "pronounce" an ordinary sentence with a lisp. Prompt the user to enter a sentence. Read the characters, one at a time, until a period appears. As they are read, convert everything to lower case and test for occurrences of the character `'s'` and the pair `"ss"`. Replace each `'s'` or `"ss"` by the letters `'t'` and `'h'`. Print the converted message using `putchar()`. For example, given the sentence `I see his house`, the output would be `I thee hith houthe`.

7. Building numbers.

   Write a program that will use a work function to input and process several data sets. Each data set will consist of three input characters if they are all valid base-10 digits, you will convert them to an integer and print it. Otherwise, print an error comment.

   - Use `isdigit()` to test for valid inputs.
   - Convert the ASCII code for a digit to its corresponding numeric value by subtracting the character `'0'`. For example, if `ch` contains the character `'7'`, then `ch-'0'` is the integer value 7.
   - If the numeric values of your inputs are stored in the variables a, b, and c, then the answer is $a * 100 + b * 10 + c$

8. A tall story.

   Develop a program for a baker who makes wedding cakes. These cakes have multiple layers, and the layers have different shapes. The top layer always is circular, with a diameter of 6 inches. The next layer down is square, each side 7.5 inches long. The third layer would be circular again, with a diameter of 8 inches; and the fourth layer is a 9.5-inch square. The shapes continue to alternate in this pattern and get bigger until the bottom layer is reached. In addition, each layer is 2 inches thick, so the area of its side is 2" $\times$ the perimeter of the layer. The baker wants to know how much frosting to make for the cake to frost the entire top and side of every layer. Write a program that will read in the number of layers desired for a cake, and then print the total square inches of cake to be covered with frosting. Break up your program as follows:

   (a) Write a function called `surface_area()`. This function has two parameters. One is a character with the value `C` for circle or `S` for square. The other is an integer that represents either the diameter for a circle or the length of a side of a square. This function will compute the sum of the top area and the side area of either a circular or a square layer, depending on the character value.
   (b) Write a main program that first will read in the number of layers of the cake. Then it will call the `surface_area()` function for each layer of the cake and total the areas. Finally, print the total.

9. Temperatures.

   Write a program and three functions that will convert temperatures from Fahrenheit to Celsius or vice versa. The main program should implement a work loop and use the improved interface of Figure 8.9. The `work()` function should prompt the user to enter a temperature, which consists of a number followed by optional whitespace and a letter (`F` or `f` for Fahrenheit, `C` or `c` for Celsius). Appropriate inputs might be `125.2F` and `-72 c`. Read the number and the letter, test the letter, and call the appropriate conversion function, described here. Test the converted answer that is returned and print an error comment if the return value is -500. Otherwise, echo the input temperature and print the answer with the correct unit, `F` or `C`. (Note that there is no difficulty reading an input in the form `125F`; `scanf()` stops reading the digits of the number when it gets to the letter and the letter then can be read by a `%c` specifier.)

   Write two functions: `Fahr_2_Cels()` converts a Fahrenheit argument to Celsius and returns the converted temperature; `Cels_2_Fahr()` converts a Celsius argument to Fahrenheit and returns it. Both functions must test the input to detect temperatures below absolute 0 ($-273.15°$C and $-459.67°$F). If an input is out of range, each function should return the special value -500.

10. Try it, I dare you!

    Write a program that will display the ASCII code in a table that looks like the one in Appendix **??**. Be sure not to try to print the unprintable characters directly. Omit the column that lists the hexadecimal codes.

Chapter 9

Program Design

In Chapter 5, we introduced the concepts of functions and function calls and defined some basic terminology. In this chapter, we review that terminology, extend the rules for defining and using functions, and formalize many aspects of functions presented in preceding chapters: prototypes, function definitions, function calls, how these elements must correspond, and how the necessary communication actually happens. The concepts of local, global, and external names are presented.

We discusses modular organization and the ways that parts of a modular program must relate to each other. The process of designing a modular program is described and illustrated with a programming example.

9.1 Modular Programs

When a program has only 20 to 50 lines, a programmer can keep the entire program structure in mind at once. Many programs, though, have thousands of lines of code. To deal with this complexity, it is necessary to divide the code into relatively independent modules and consider each module in isolation from the others, usually with a main program in one module and groups of closely related functions in others. Each module is composed of functions and declarations that relate to one identifiable phase of the overall project. This is the way professionals have been able to develop the large, sophisticated systems that we use today.

A **module** is a pair of files (a `.c` file and its matching `.h` file) containing programmer-defined types, data object declarations, and function definitions. The order of these parts within the module is quite flexible; the only constraint is that *everything must be declared before it is used*. The modules themselves and the functions within them serve several purposes: they make it easy to use code written by someone else or reuse your own code in a new context. Far more important, though, is that they permit us to break a large program into manageable pieces in such a way that the interface among pieces is fixed and controllable. Functions, their prototypes, and header files make this possible in C; class definitions make it easier in C++.

Each **function** is a block of code that can be invoked, or called, from another function and will perform a specific, defined task. All its actions should be related and work at the same level of detail. For example, some functions "specialize" in input or output. Others calculate mathematical formulas or do error handling. No function should be very long or very complex. Complexity is avoided by calling other functions to do subtasks.

One guideline for good style is to keep each function short enough that its parameters, local variable declarations, and code will fit on the video screen at the same time. This limits functions to about 20 lines of code on many computers. Following this guideline, any moderate-sized program will have many functions that are organized into several code modules or classes, with each module or class in a separate source file. The question then arises of where various objects should be declared: in which module and where within the module. This level of design is touched on in Chapter 21; we consider only the organization of a single module here.

9.2 Communication Between Functions

Before beginning this section, you should review the overview of functions and module organization that is given in Chapter 5.

The arrows show the ways that information flows to and from functions in Figure 9.3.

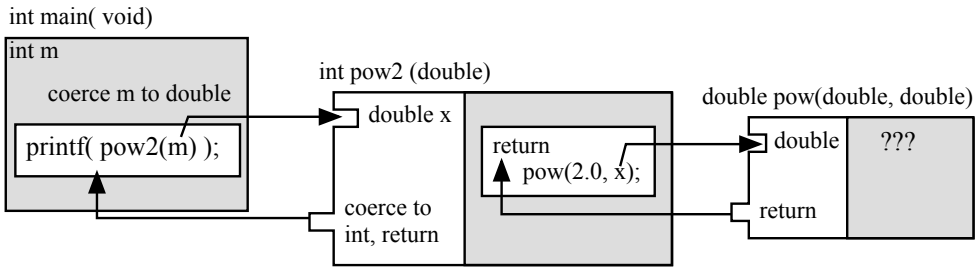


Figure 9.1. Information flow in `pow2()`.

One function in every program must be named `main()`; it is often called the *main program*. The **main program** can call other functions and those functions can call each other, but none can call `main()`. We use the term **caller** to refer to the function that makes the call and **subprogram** to refer to the called function. For example, in Figures 5.24 and 5.25, `main()` is the caller and `cyl_vol()` is a subprogram that is called from the second box in `main()`.

9.2.1 The Function Interface

Each function has a defined **interface**, which is declared by writing a prototype for the function. The prototype lists the return type and the type and name of one or more parameters. Either the parameter list or the return type may be replaced by `void`.

Each parameter listed in the prototype becomes a separate communication path by which the caller can send information into the function. The function's result or return value (if not `void`) is a communication path from the subprogram back to the caller, as is each address parameter. Information also can be passed between a caller and a subprogram through global variables, which will be discussed in Section 9.4. However, this practice is discouraged and should be avoided wherever possible. Extensive use of global variables can make a program difficult to debug because it vastly increases the number of possible interactions among program modules and introduces the possibility for unintended and undocumented interactions. Thus, modern programming style dictates that all information passed between caller and subprogram should go through the declared interface.

In Figure 9.1, the body of each function is shaded, indicating that it may appear as a "black box" to the programmer. The inner workings of a library function frequently are hidden from the programmer, like those of `exp()` in this case. You need not know the details of what is inside a function to be able to use it.

In contrast, interfaces are white. This symbolizes that the programmer can (and must) know the details of the interface. This information is supplied in a header file and by the documentation that normally accompanies a software library. **Header files** are used to keep the declarations consistent from module to module and to permit functions in one module to call functions in another.

The passage of information into the subprogram is represented by right-facing arrows. Function execution begins at the entry point and, when complete, control returns to the caller along a left-facing arrow, which ends at the return address in the caller. The caller continues processing from that point. The function also may return a result along the left-facing arrow.

A **function call** consists of the name of the function followed by a list of arguments in parentheses. Some functions have no parameters, in which case the parentheses in the function call still must be written but with nothing between them. During the calling process, two kinds of information are sent from the caller into the subprogram:

- One argument value for each declared parameter.
- The **return address**, which is the address of the first instruction in the *caller* after the function call. This address is passed by the caller to the subprogram on every call so that the function knows where to go when its execution is finished.

During a function call, the C run-time system allocates a stack frame, stores the argument values there, and stores the return address in the adjacent locations. Control is then transferred to the function, which allocates (an possibly initializes) storage for local variables. Control then jumps to the **entry point** of the function,

which is the first line of code in its body. Execution of the subprogram begins and continues until the last line of code is completed, the program aborts by calling `exit()`, or control reaches a `return` statement, which returns control to the caller at the return address, taking along any return value produced.

The **return type** declared in the prototype is the type of value that will be returned. If the `return` statement returns an answer of some other type, C will convert it to the declared type and return the converted value. If such a conversion is not possible, the compiler will issue an error comment. This is discussed more fully in Section 9.3.

Missing prototypes. The general rule in C is that everything must be declared before it is used. There are two ways to "declare" a function: either supply a prototype or give the complete function definition. To guarantee correctness, one or the other must occur in your program before any call on that function. The C compiler[1] must know the prototype of a function to check whether a call is legal. Sometimes, however, a programmer forgets to either `#include` a necessary header file or write a prototype for a locally defined function. Sometimes the prototype is in the file but in the wrong place, coming after the first call on the function.

In any of these cases, the compiler does *not* just give an error comment about a missing prototype and quit. The first time it encounters a call on a nonprototyped function it simply *makes up* a prototype and continues compiling. The compiler will use the types of the actual arguments in the call to construct a prototype that matches, but all such created prototypes have the return type `int`. Sometimes this prototype is exactly what the programmer intended; other times it is wrong because the call depends on an automatic type conversion or contains an error. In any case, the constructed prototype becomes the official prototype for the function and is used throughout the rest of the program. If it has an incorrect parameter or return type, the compiler will compile too many or too few type coercions for each function call.

If a misplaced prototype is found later in the file and it is the same as the prototype constructed by the compiler, there is no problem. However, if it is different, the compiler will give an error comment such as *type conflict in function declaration* or *illegal redefinition of function type*. This can be an astonishing error comment if the programmer does not realize that the problem is *where* the prototype was written, not *what* was in it. If the prototype really is missing, not just misplaced, a similar error comment may be produced when the compiler reaches the actual function definition. If you see such an error comment, check that all functions have correct prototypes and that they are at the top of the program.

9.2.2 Parameter Passing

When each function is called, a new and separate memory area, called a *stack frame*, is allocated for it. The function's parameters are allocated in its stack frame and argument values are copied into the parameter during the function call. A function's local variables are also in the frame, as is the information necessary to return from the function. Figure 9.2 is a diagram of the run time memory allocated for the program in Figure 5.24. As each function is called, a new frame for it is placed on the stack. First `main()` was called, then it called `cyl_vol()`, which called `pow()`. Parts of the `pow()` frame are gray because we have no idea how this library function is implemented or what its parameters are named.

When a function returns, its stack frame is deleted. Later a frame for another function might be stored in the same addresses. In the cylinder program, storage for the `cyl_surf()` function will end up in the memory locations that `cyl_vol()` had previously occupied, and the stack frame for `pow()` will be at a slightly different address because the frame for `cyl_surf()` is bigger than the frame for `cyl_vol()`.

Call by value. Most arguments in C are passed from the caller to the subprogram **by value**. This means that a copy of the value of the argument is sent into the subprogram and becomes the value of the corresponding parameter. The function does not know the address of the argument, which could be a variable or the result of an expression. If the argument is a variable, the subprogram cannot change the value stored there. For example, in Figure 5.25, the subprogram `cyl_vol()` receives the value of `main()`'s variable `diam` but not the address of `diam`. The code in the body of `cyl_vol()` can change the value of its own parameter, `d`, but doing so will not change the value stored in `main()`'s `diam`. Information cannot be passed back to the caller through an ordinary parameter.

[1] ISO C and older C implementations differ extensively on the rules for function prototypes, definitions, and type checking. In this text, we speak only of standard ISO C.

We see the memory allocated for the program in Figure 5.24 at two moments during run time. The diagram on the left shows memory just after the `pow()` function is called by `cyl_vol()`. The diagram on the right is a later moment, during execution of `cyl_surf()`.

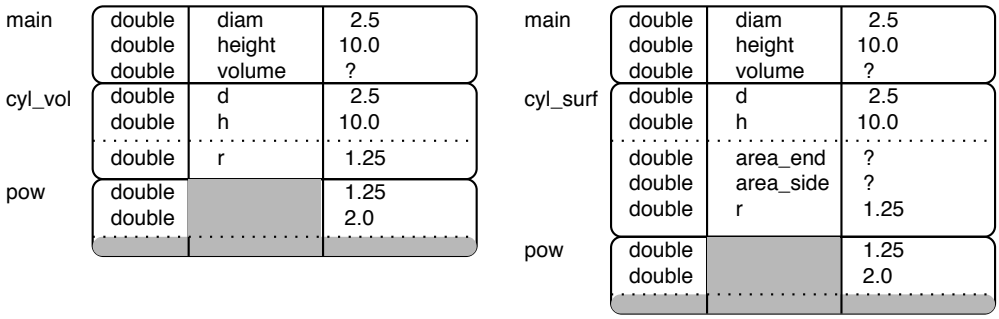


Figure 9.2. The run-time stack.

Address parameters. In contrast, some arguments are passed by passing the address of a variable (rather than its value) into the subprogram. The subprogram can both use and change the information at the argument address. An example of a function that sometimes must return more than one piece of information is `scanf()`. It uses the return value to return an error code, which we have ignored so far,[2] and it returns one or more data items through **address arguments**. When we call `scanf()`, we send it the address of each variable to be read. It reads the data, stores the input(s) in the given address(es), and returns a success or failure code. A programmer also can define such functions with address parameters; we explain how in Chapter 11.

9.3 Parameter Type Checking

The compiler uses a prototype for two purposes: checking whether the call is legal and compiling any type conversions necessary to make the argument types match the parameter types.

Number of arguments. If the number of arguments in a function call is appropriate, the compiler considers each parameter-argument pair, one by one, comparing the parameter type declared in the prototype to the type of the argument expression. If they match exactly, code is compiled to copy the argument values into the subprogram's parameters and transfer control to its entry point. If the number of arguments supplied by a function call does not match the number declared in the prototype, the compiler prints an error comment[3].

Type coercion of mismatched arguments and parameters. If the number of arguments is the same as the number of parameters but their **types do not match** exactly, the compiler will attempt to convert each argument to the declared parameter type according to the standard type-conversion rules. We already discussed a large number of variations of the basic integer, floating-point, and character types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `char`, `signed char`, `unsigned char`, `float`, `double`, and `long double`. Taken as a set, these are called the **arithmetic types**. An argument of any arithmetic type can be coerced to any other arithmetic type, if needed, to make the argument's type match the type declared in the function's prototype. An instance of such **type coercion**, is shown in the Figure 9.3; the argument in the call on `pow2()` is an `int`, while the parameter is a `double`. The C compiler will include code to convert the `int` argument value to type `double` as part of the function call, and the function will receive the `double` value that it expects.

Meaningful conversions. Type conversion or coercion is possible if there is a meaningful relationship between the two types, such as both being numbers. If conversion is not possible, the compiler will issue a

[2]This will be explained in Chapter 14.

[3]Some functions, such as `printf()` and `scanf()` permit the number of arguments to vary. Similar functions can be written using special argument-handling mechanisms supported by the `stdard` library.

```
#include <stdio.h>
#include <math.h>
int pow2 ( double x ) { return pow( 2.0, x ); }      // 2 to the power x

int main( void )
{
    for (int m=0; m<10; ++m)  printf( "%10i\n", pow2(m) );
    return 0;
}
```

This program prints a table of the powers of 2 using a `double`→`int` function named `pow2()`. When `pow2()` is called from `main()` with an integer argument, the argument will be coerced to type `double` to match `pow2()`'s prototype. Within the function, the result of calling `pow()` will be coerced to type `int`, to match `pow2()`'s prototype, before being returned to `main()`.

Figure 9.3. The declared prototype controls all.

fatal error comment.[4] The rule in C is that any numeric type can be converted to any other numeric type. Therefore, a `short int` can be converted to a `long int` or an `unsigned int` or a `float` and vice versa. Some kinds of argument coercions are very common and compilers simply include code for the conversion and do not notify the programmer that it was necessary. For example, normally no warning would be given when a `float` value is coerced to type `double`. At other times, compilers warn the programmer that a conversion is occuring. This happens when an unusual kind of argument conversion would be required or the conversion might result in a loss of information due to a shortening of the representation, as when a `double` value is converted to type `float`. The warning you get depends on the nature of the type mismatch, the severity of the possible consequences, and your particular compiler.

Type coercion of returned values. The declared return type also is compared to the actual type of the value in the `return` statement. If they are different, the value will be coerced to the declared type before it is returned. The compiler generates the conversion code automatically. For example, in Figure 9.3, the value calculated by the expression in the `return` statement is of type `double` (the math-library functions always return `double`s). However, the prototype for `pow2()` says that it returns an `int`. What happens? The C compiler will notice the type mismatch and compile code to coerce the `double` value to an `int` during the return process. The compiler may also give a warning message.

Parameter order. The order of the arguments in a function call is important. If a function's parameters are defined in one order and arguments are given in a different order in a call, the results generally will be nonsense. There is no "right" order for the parameters in a function definition; this is a design issue. However, once the definition has been written, the order of the arguments in the call must be the same. If the program has several functions that work with the same parameters, the designer should choose some order that makes sense and consistently stick to that order when defining the functions to avoid absentmindedly writing function calls with the arguments in the wrong order.

Sometimes a compiler, by performing its normal parameter type checking, can detect an error when a programmer scrambles the arguments in a function call. More often, this is not possible. For example, the function `cyl_vol()` in Figure 5.24 has two parameters, the diameter and height of a cylinder. Since the parameters are the same type, the compiler cannot tell when the programmer writes them in the wrong order. The result will be a program that compiles, runs, and produces the wrong answer. To further demonstrate the kind of nonsense that can result from mixed-up arguments, we will use a silly two-parameter function named `n_marks()` that inputs a number N and a character C and prints N copies of C. (Figure 9.4)

The output is

```
Enter a number and a character: 45.7 !
You entered 45.70 and !.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Compare this output to the output from the erroneous call in the next paragraph.

[4] If the function has an ANSI prototype, the coercions allowed for arguments and return values are the same as those allowed for assignment statements.

This program is used to show the effects of calling a function with parameters in the wrong order.

```
#include <stdio.h>
#include <math.h>
void n_marks( double n, char ch );   // function prototype

int main( void )
{
    char ch;     // The character to print
    double x;    // How many characters to print

    printf( "\nEnter a number and a character: " );
    scanf( "%lg %c", &x, &ch );

    printf( "\nYou entered %.2f and %c.", x, ch );
    n_marks( x, ch );            // Call function to print x many ch's
    return 0;
}
// ----------------------------------------------------------------------
void n_marks( double n, char ch )    // Print n copies of character ch.
{
    putchar( '\n' );
    for (int k = 0; k<n; ++k) putchar( ch ) ;  // print n characters
    printf( "\n\n" );                          // flush output to screen
}
```

Figure 9.4. The importance of parameter order.

Parameter coercion errors. When the compiler translates a function call, it either accepts the arguments as given, coerces them to the declared type of the parameter, or issues a fatal error comment. This automatic conversion can result in some weird and invalid output. For example, suppose a programmer called the `n_marks()` function but wrote the arguments in the wrong order: `n_marks( ch, x );` The output would be:

```
Enter a number and a character: 45.7 !
You entered 45.70 and !.
---------------------------------
```

This call certainly is not what a programmer would intend to write. However, according to the ISO C standard, this is a legal call. The standard dictates that the character `ch` will be converted first to an `int` and then to a `double`, while the `double x` will be converted first to an `int` and then to a `char` to match the parameter types in the prototype `void n_marks( double, char )`. Using this input, the programmer would expect to see a line of 46 `!` signs, but instead a line of 33 `-` signs is printed because of the conversions: The ASCII code for `!` is 33, which will be converted to 33.00, and the 45.7 will be converted to 45, which is the code for `-`. Some compilers at least may give a warning comment about these two "suspicious" type conversions, but all standard ISO C compilers will compile the conversion code and produce an executable program. When you try to run such a program, the results will be nonsense, as shown.

9.4 Data Modularity

A large program may contain hundreds or thousands of objects (functions, variables, constants, types, etc.). If a programmer had to remember the name, purpose, and status of this many objects, large programs would be very hard to write and harder to debug. Happily, C supports modular programming. Each module has its own independent set of objects and interaction between modules can be strictly controlled. The same name can be

used twice, in different modules, to refer to different objects, so a programmer need not keep a mental catalog of the hundreds of names that might have been used. C's accessibility and visibility rules determine *where* a variable or constant may be used and *which* object a name denotes in each context. We use the program in Figures 9.5 and 9.7 to illustrate these concepts and the related concept of scope.

9.4.1 Scope and Visibility

The **scope** of a name is the portion of the program in which it exists and can be used. The three levels of scope are local, global, and external; respectively meaning that access to an object can be restricted to a single program block or function, shared by all functions in the same file or program module, or shared by parts of the program in different files or program modules. More specifically,

- Parameters and variables defined within a function are completely **local**; no other functions have access

This program calculates the pressure of a tank of CO gas using two gas models. The functions in Figure 9.7 are part of this program and should be in the same source file. A call graph is given in Figure 9.6

```
#include <stdio.h>
#define R   8314              // universal gas constant
float ideal( float v );       // prototypes
float vander( float v );
float temp;                   // GLOBAL variable: not inside any function
                              // temperature of CO gas

int main( void )
{
    // Local Variables -------------------------------------
    float m;                   // mass of CO gas, in kilograms
    float vol;                 // tank volume, in cubic meters
    float vmol;                // molar specific volume
    float pres;                // pressure (to be calculated)

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );
    vmol = 28.011 * vol / m;        // molar volume of CO gas
    pres = ideal( vmol );           // pressure; ideal gas model
    printf( "\n The ideal gas at %.3g K has pressure "
            "%.3f kPa \n", temp, pres );
    pres = vander( vmol );       // pressure; Van der Waal's model
    printf( "\n Van der Waal's gas has pressure "
            "%.3f kPa \n\n", pres );

    return 0;
}
```

Figure 9.5. Gas models before global elimination.

This is a function call graph for the gas pressure program in Figures 9.5 and 9.7.

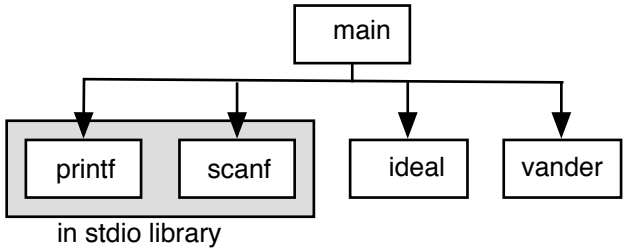


Figure 9.6. A call graph for the CO gas program.

to them.

- We are permitted to declare variables outside of the function definitions, either at the top of a file or between functions. These are called **global** variables. They can be used by other modules and all the functions in the code module that occur after their definition in the file. It is possible (but not the default) to restrict the use of these symbols to the module in which they are defined.
- All global names and all the functions defined in a module default to `extern`; that is, they are **external** symbols unless declared otherwise. This means that their names are given to the system's linker and all the modules linked together in a complete program can use these variables.[5]

[5]External linkage will be discussed in Chapters 19 and 23.

These functions are part of the gas models program in Figure 9.5 and are found at the bottom of the same source code file.

```
// ---------------------------------------------------------------------
//  Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
float ideal( float v )
{
    float p;                  // LOCAL variable DECLARATION

    p = R * temp / v;         // pressure in Pascals
    return p / 1000.0;        // pressure in kilo Pascals (kPa)
}
```

```
// ---------------------------------------------------------------------
//  Pressure of CO gas in a tank, using Van der Waal's equation,
//  P = RT/(v-b) - a/(v*v).
float vander( float v )
{
    float p;                 // LOCAL declaration, not same p as above
    const float a = 1.474E+05;                 // constants for CO gas
    const float b = .0395;

    p = R * temp / (v - b) - a / (v * v);      // pressure in Pascals
    return p / 1000.0;                         // kPa pressure
}
```

Figure 9.7. Functions for gas models before global elimination.

The diagram shows the scope of the functions, variables, and constants defined in the gas models program from Figures 9.5 and 9.7. Each box represents one scope; the gray box represents the global scope. Local scopes are white boxes; within each, parameters are listed above the dotted line and local variables and constants below it. Symbols defined in the gray area are visible everywhere in the program where they are not masked. Symbols in the white boxes are visible only within the scope that defines them.

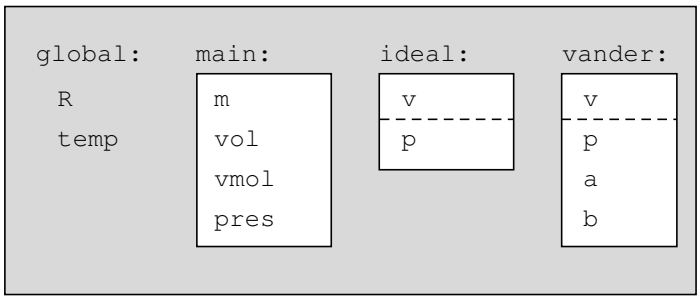


Figure 9.8. Name scoping in gas models program, before global elimination.

We say that a locally declared variable or parameter is **visible**, or **accessible**, from the point of its declaration until the block's closing `}`. This means that the statements within the block can use the declared name, but no other statements can (i.e., the scope and visibility of a local variable are the same). A global or external variable is visible everywhere in the program after its definition, *except* where another, locally declared variable bears the same name. If a function has a parameter or local variable named `x` and a statement `x = x+1`, the local `x` will be used no matter how many other objects named `x` are in the program's "universe." Therefore, the visibility of a global variable is that portion of its scope in which it is not masked or hidden by a local variable. This is a very important feature; it is what frees us from concern about conflicts with the names of all the hundreds of other objects in the program.

9.4.2 Global and Local Names

Insofar as possible, all variables should be declared locally in the function that uses them. Information used by two or more functions should be passed via parameters. The use of global variables usually is a bad idea; any variable that can be seen and used by all the functions in a program might be changed erroneously by any part of the program. When global variables are in use, no one part of the program can be fully understood or debugged without considering the entire thing.

While global variables are not encouraged, constants and types usually are declared at the top of a file because they are necessary to coordinate the actions of different parts of a program. Since the values of `const` variables and `#define`d symbols cannot be changed, their global visibility does not foster the same kind of problems as a global variable. Further, declaring constants and types at the top of a file makes them easier to locate and revise, if necessary.

A function prototype can be declared globally or locally in every function that calls it. Each style of organization has its advantages. In this text, we declare most prototypes globally because it is simpler. We illustrate some of these issues using the program in Figure 9.5, which has two subprograms (Figure 9.7) and a variety of local and global declarations (Figure 9.8).

Notes on Figures 9.5, 9.7, and 9.8. Gas models before global elimination. We use a main program, two functions, a global constant, and a global variable to examine the scope and visibility of names in C. Figure 9.8 illustrates the scope of the symbols defined in this program.

First box, Figure 9.5: global declarations and included files .

- The `#include <stdio.h>` means that everything in the file `stdio.h` will be copied into this program at this point. All the objects declared in `stdio.h`, therefore, will have a global scope in this program.
- The constant `R` and the variable `temp` are declared globally. In Figure 9.8, these names are written in the gray box, which represents the global scope. Here, they are visible and can be used by any function in this

file; that is, by `main()`, `ideal()`, and `vander()`. The functions `ideal()` and `vander()` also can be called from any part of this file.

- It is considered very bad style to use a global variable such as `temp`. We do so only to demonstrate the meaning of global declarations and show how to eliminate them.
- A global constant such as `R` creates fewer problems than a global variable. It is common for a program to use global constants and is not considered bad style.

Second box, Figure 9.5: declarations for `main()`***.***

- The four variables `m`, `vol`, `vmol`, and `pres` are local to `main()` and therefore visible only within `main()`. In Figure 9.8, the leftmost white rectangle represents the scope created by `main()`. Inside it is a list of `main()`'s local variables.
- The functions `ideal()` and `vander()` cannot access the values in `m`, `vol`, `vmol`, and `pres` because the values are local within `main()`.

Third box, Figure 9.5: code for `main()`***.***

- This code can use the definitions and the global variable and constant defined in the first box as well as the variables defined in the second box.
- This code cannot use the parameters, variables, or constants defined by the two functions in Figure 9.7. If we tried to use `v`, `p`, `a`, or `b` here, it would cause an undefined-symbol error at compile time.
- The two inner boxes contain the calls on the functions `ideal()` and `vander()`. These functions will need to be modified to eliminate the use of the global variable.

Sample output from this program is:

```
Input the temperature of CO gas (degrees K): 28.5

The mass of the gas (kg) is: 1.2

The tank volume (cubic m) is: 3

The ideal gas at 28.5 K has pressure 3.385 kPa
Van der Waal's gas has pressure 3.357 kPa
```

First box, Figure 9.7: code for the function `ideal()`***.***

- Because this function is compiled at the same time as the code in Figure 9.5, it can use any object, such as `R` or `temp`, that is defined (or included) in the first box in Figure 9.5.
- In Figure 9.8, the center rectangle represents the scope created by `ideal()`. Inside it are `ideal()`'s parameter `v` and the local variable `p`, which are visible only within `ideal()` and cannot be used outside this function.

Second box, Figure 9.7: code for the function `vander()`***.***

- This function also can use objects such as `R` and `temp` that are defined (or included) in the first box in Figure 9.5.
- In Figure 9.8, the rightmost rectangle represents the scope created by `vander()`. Inside it are `vander()`'s parameter `v` and the local variables `a`, `b`, and `p` (`a` and `b` actually are constants). These are visible only within `vander()` and cannot be used outside this function.
- Each parameter or local variable declaration creates a new object. Therefore, the `p` defined here is a different variable than the `p` defined in `ideal()`, with its own memory location, even though they have the same name.
- If the local variable `p` had been named `temp`, this function would compile but not work properly because the local variable would mask the global variable. Every reference to `temp` in `vander()` would access the local variable, and the information in the global `temp` would not be accessible within the function. The next section shows how to improve the lines of communication so that these difficulties do not occur.

9.4.3 Eliminating Global Variables

We introduced a programming style in which interaction with the user is done by one function (ofen `main()`) and calculations by another. This separation of work makes a program maximally flexible and easier to modify at a later date. However, since one function reads the input data and another uses it for calculations, the data value must be communicated from the first to the second.

A beginning programmer often will be tempted to use a global variable because it provides one way to solve this communication problem. A global variable is visible to both the data input function and the calculation function, so nothing special has to be done to communicate the value from the first to the second.

In a small program, using global variables might seem easy and communicating through parameters might seem to be a nuisance. However, global variables almost always are a mistake,[6] because they allow unintended interactions between distant parts of the program. They make it hard to follow the flow of data through the process, and they make it harder to modify and extend the program. In a large program, global variables

[6]The program in Figure ?? shows an instance where the use of global variables is acceptable.

This program is a revised and improved version of the gas models program in Figures 9.5 and 9.7; it solves the communication problem by using a parameter instead of a global variable. The functions in Figure 9.10 are part of the revised program and should be in the same source file.

```
#include <stdio.h>
#define R  8314           // universal gas constant

float ideal( float v, float temp );
float vander( float v, float temp );

int main( void )
{
    // Local Variables -------------------------------------
    float temp;           // Temperature of CO gas
    float m;              // mass of CO gas, in kilograms
    float vol;            // tank volume, in cubic meters
    float vmol;           // molar specific volume
    float pres;           // pressure (to be calculated)

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );

    vmol = 28.011 * vol / m;                      // molar volume of CO gas
    pres = ideal( vmol, temp );    // pressure; ideal gas model
    printf( "\n The ideal gas at %.3g K has pressure "
            "%.3f kPa \n", temp, pres );
    pres = vander( vmol, temp );   // pressure; Van der Waal's model
    printf( " Van der Waal's gas has pressure "
            "%.3f kPa \n\n", pres );

    return 0;
}
```

Figure 9.9. Eliminating the global variable.

These functions illustrate how to eliminate a global variable from Figure 9.7. The boxes highlight the changes necessary to replace the global variable by adding a parameter to each function.

```
// ----------------------------------------------------------------------
//  Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
*/
float ideal( float v, float temp )
{
    float p;                  // LOCAL variable DECLARATION

    p = R * temp / v;         // pressure in Pascals
    return p / 1000.0;        // pressure in kilo Pascals (kPa)
}
// ----------------------------------------------------------------------
//  Pressure of CO gas in a tank, using Van der Waal's equation,
//  P = RT/(v-b) - a/(v*v).
*/
float vander( float v, float temp )
{
    float p;                  // LOCAL declaration, not same p as above
    const float a = 1.474E+05;
    const float b = .0395;                  // constants for CO gas

    p = R * temp / (v - b) - a / (v * v);   // pressure in Pascals
    return p / 1000.0;                      // kPa pressure
}
```

Figure 9.10. Functions for gas models after global elimination.

become a debugging nightmare. It is hard to know what parts of the program change them and under what conditions. Therefore, it is important, from the beginning, to avoid global variables and learn to use parameters effectively.

We will use the program in Figures 9.5 and 9.7 to demonstrate the technique for eliminating global variables. The result is shown in Figures 9.9 and 9.10. Parameters are the right way to solve the communication problem. They make the sharing of data explicit and they prevent unintended sharing with unrelated functions. In general, global variables should be replaced by parameters. The transformation works for globals used to send information into a function; a variant of this technique[7] is needed if the function also uses the global variables to send information back out.

Notes on Figures 9.9 and 9.10. Eliminating the global variable. We start with the main program in Figure 9.5 and the two functions in Figure 9.7, which communicate through a global variable. To eliminate this variable and replace it by a parameter, we need to change five lines in Figure 9.5 and two lines in Figure 9.7.

First box of Figure 9.9. The prototypes for the functions `ideal()` and `vander()` found in the first box of Figure 9.5 need to be changed to include an additional parameter of the same type as the global variable. In both cases, we add the new parameter second. It is important to be consistent about parameter order when adding parameters, to avoid the problems mentioned earlier.

Second box of Figure 9.9. The declaration of `temp` must be moved out of the global area where it was defined in Figure 9.5 and into `main()`'s local area.

Third and fourth boxes of Figure 9.9. The function calls in these boxes must be modified to include an additional argument, the value of the former global variable, which is now a local variable in `main()`.

[7] This variation uses pointer parameters, which will be covered in Chapter 11.

First box of Figure 9.10. A new parameter was added to the parameter list in the prototype for `ideal()`. We also must add the parameter to the function header.

Second box of Figure 9.10. We must add the new parameter to the list for `vander()` as well.

9.5 Program Design and Construction

In the gas pressure example, we started with a program and its two functions and analyzed it section by section. This is a good way to understand how a given program works, but it gives little perspective on how a program with functions is developed. In this section, we reverse the process and show how to develop a program and its subprograms from the specification. Section 9.5.1 lists the steps and describes them briefly. Section 9.5.2 explains these steps more fully and applies them to a real problem.

9.5.1 The Process

The DNA: a complete specification. The first step in designing a program is to decide what you want the program to do and specify it as precisely as possible. If there is any doubt about the specifications or any missing information, this must be cleared up before proceeding.

Start with the "skin" of `main()`***.*** Write the routine portions of `main()`. If you wish to test or run the program on several data sets, write a work loop in the body of main.

Define the skeleton of the work to be done. Start by listing the major phases in processing a single data set. Generally these phases are input, calculation, and output; but one of these phases might not be needed and the calculation phase may have multiple steps. Write the code to perform each phase if it is only one or two lines long. Otherwise, invent a name for a function that will do each task. These statements will be in a `work()` function if you have one, otherwise in `main()`.

The circulatory system: declarations and prototypes. Go back to the top of `main()` and write whatever declarations and prototypes you will need to support your skeleton code. These do not have to be perfect. Write a comment for each one.

Health check: compile. It is much easier to find compiler errors when you compile and check the code a little bit at a time. However, a program will not compile if it calls functions that have not yet been written.

One solution to this is to write a *stub* for each function that has been named but not yet defined. A stub is a function that does nothing except print its own greeting message and return some value of the correct type (or void). The return value does not need to be meaningful; any definition is OK, as long as its parameter types and return value type match the prototype. Compile the program initially as soon as the stubs are written. Then later, after every function is fleshed-out, compile the program again.

An alternative to writing function stubs is to temporarily **amputate** calls on functions that have not yet been written by enclosing them in comment marks. Sometimes entire lines are amputated. At other times, a function call is commented out and replaced by a literal constant. The comments can easily be removed when the function is written, later.

The brains: developing the functions. Tackle the functions one at a time, in any convenient order. For each, write a comment block that describes the purpose of the function and any preconditions. Then go through the same steps as for `main()`: routine parts, skeleton, declarations, and possibly more prototypes and stubs. As you learn more about your functions, you may need to add parameters to the prototypes you have previously written.

Integration and tesing. Combine and test all of the program parts according to your plan.

9.5.2 Applying the Process: Stepwise Development of a Program

We now apply the design steps from Section 9.5.1 to develop a program for a real problem: calculating the temperature of a cooling fin. This problem arose from a real application: designing the cooling apparatus for a piece of machinery. Figure 9.11 is a diagram of the cooling fin and specifications for a program to analyze its temperature gradient. The problem specification comes directly from the set of engineering principles and formulas in use by the designer.[8]

[8] F. Incropera and D. DeWitt, *Fundamentals of Heat and Mass Transfer*, 4th ed. (New York: Wiley, 1996).

A fin is slender if its length is an order of magnitude greater than the height or width of its rectangular cross section. The cooling properties of a fin depend on temperatures of the wall and the air and a fin constant, *FinC*, which is a function of the film coefficient of the air and the thermal conductivity and cross-sectional area of the fin.

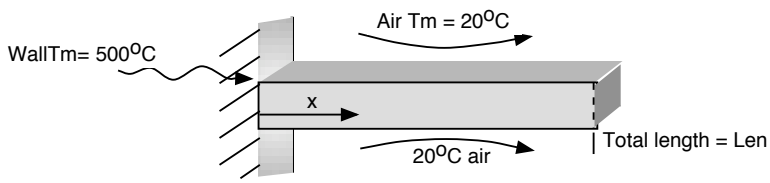
Steps in writing `main()`.

- *The skin.* We start by writing the `#include` commands (`stdio`, as usual, and `math` because this is a numeric application.) and the first and last few lines of `main()`, the greeting message and return statement

```
#include <stdio.h>
#include <math.h>
int main( void )
{
    puts( " Temperature Along a Cooling Fin" );
    // Program code will go here.
    return 0;
}
```

- *Multiple data sets.* We will write this program for only one data set, so we do not need a processing loop or a `work()` function. Therefore, the skeleton of the program becomes the body of `main()`.

Problem scope: A long, slender cooling fin of rectangular cross section extends out from a wall, as shown. Print a table of temperatures every 0.005 m along the fin.



Formulas: The temperature of the fin at a distance x from the wall is:

$$\text{Temperature}(x) = AirTm + (WallTm - AirTm) \times \frac{\cosh[FinC \times (Len - x)]}{\cosh(FinC \times Length)}$$

Constants:
Temperature of the air, $AirTm = 20°\text{C}$
Temperature of the wall at the base of the fin, $WallTm = 500°\text{C}$
Fin constant, $FinC = 26.5$

Input: Length of the fin, Len, in meters, which must be greater than zero.

Output required: Print column headings `Distance from base (m)` and `Temperature (C)`. Beneath the headings print a table of temperatures starting with the temperature at the wall ($x = 0$). Print one line for each increment of 0.005 m up to the length of the fin.

Computational requirements: Print the distance from the wall to three decimal places and the temperature to one.

Test plan: Using the numbers from the diagram, for a wall that is .6 meters long, the temperature at the wall should be $500°C$ and $.01m$from the wall, the temperature should be $398.535°C$.

Figure 9.11. Problem specification and test plan: fin temperature.

- *The skeleton.* Next, prepare the **function skeleton**; We start by listing the two major phases in creating a table for a single fin situation. We need two basic steps:

  a. Input and validate a fin length.
  b. Print a temperature table for that length.

  We write the code for step (a) directly because it requires only a few lines of code.

```
do {
     printf( " Please enter length of the fin (> 0.0 m): " );
     scanf( "%g", &Length );
} while (Length <= 0);
```

- *The circulatory system.* We finish step (a) by writing the declaration for `Length`.

```
float Length;       // Length of the cooling fin.
```

- Step (b) is more complex, however, so we invent a function to perform the task and name it `print_table()`. We write a first draft of the prototype for `print_table()`, giving it the parameters and return type we think it will need. This information comes from the formula given in the specification: all values are constant except the fin length, so the length is the only parameter needed. We set the return type to `void` because most printing functions are `void`.

```
void print_table( float Length );
```

  Only the prototype is written at this stage; the code itself will be written later. This is just a first guess at the proper prototype: if there are too many parameters or too few, or the types are wrong, that will be corrected later. Now we return our attention to `main()` and write a call on the new function.

```
print_table( Length );
```

  If (unlike this example) the new function returns a value, we must store the result in a variable and we may need to write a declaration for that variable.

- *Health check.* The first draft of the main program is now complete; it is shown in Figure 9.12 with the corresponding call graph. We would like to use the C compiler to check the work so far but we cannot compile the program as it stands because the definition of `print_table()` is missing. So we create a stub for `print_table()`. This consists of an identifying comment, a function header that matches the prototype, and a single output statement to let the programmer track the program's progress. Since this function has a parameter, we use the stub to print its value, giving us confidence that the data is being correctly communicated to the function. We put this stub at the end of the program.

  The stub served its purpose when we compiled this file. There were a few typographical errors, but the program is so short that they were easy to find and fix. The corrected code is shown in Figure 9.12. It ran successfully, producing this output:

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06
>>> Entering print_table with parameter 0.06
```

  From this, we can see that the program is starting and ending properly and receiving its input correctly.

- *The brains.* We now can start work on the function. If a program has two or more functions, we code them one at a time, in any convenient order. For each, we work on the skin, the skeleton, the circulation, and the brains.

  Sometimes this leads to inventing another function; sometimes it means writing statements that compute the formulas given in the specification.

Steps in writing `print_table()`.

- *The DNA and skin.* We have written a prototype and a function stub for `print_table()`. Now we need to think carefully about the function itself. One part of that process is to write a set of *preconditions* for the function, that is, things that must be true when the function is called. This function has one parameter, the length of the fin, which must (obviously) be a positive number, so we add a line to the comment block to document this precondition. This precondition announces that it is the job of the caller `main()` to validate the input – it will not be validated here. We now have this skeleton:

```
// Stub:  ----------------------------------------------------------------
// Print a table of temperatures for a fin of given length, in meters.
// The length must be greater than 0.

void print_table( float Length )
{
    printf( " >>> Entering print_table with parameter %g", Length );
}
```

- *The skeleton.*
  To print a table, we must
  1. print table headings,
  2. print several lines of detail, and
  3. print table footings.

This code was written piece by piece on the preceding few pages. The corresponding call graph follows.

```
#include <stdio.h>
#include <math.h>

void print_table( float Length );

int main( void )
{
    float Length;     // Input:  the length of the cooling fin.

    puts( " Temperature Along a Cooling Fin" );
    do {
        printf( " Please enter length of the fin (> 0.0 m): " );
        scanf( "%g", &Length );
    } while (Length <= 0);

    print_table( Length );
    return 0;
}

// Stub:  ----------------------------------------------------------------
// Print a table of temperatures for a fin of given length, in meters.

void print_table( float Length )
{
    printf( " >>> Entering print_table with parameter %g", Length );
}
```

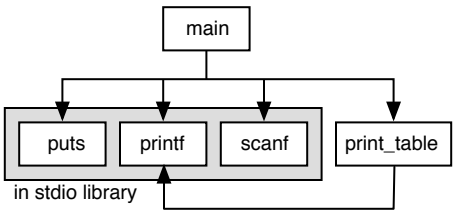


Figure 9.12. First draft of `main()` for the fin program, with a function stub.

Steps (1) and (3) are simple enough to write directly. We do that and leave space to insert step (2) later.

```
printf( "\n Distance from base (m)    Temperature (C) \n" );
printf( " ----------------------    --------------- \n" );
        // STEP 2 WILL GO HERE
printf( " ----------------------    --------------- \n" );
```

- *The circulatory system.*
  We need a constant in the `print_table()` function: the step size for the loop. The specification says that the temperature should be calculated every 0.005 meters, but it always is unwise to bury constants like that in the code. We define the step size as a local constant so that it will be easy to modify in the future. It is obvious that we also need at least one variable, `distance`, to hold the current distance from the wall and another, `temp`, to hold the result of the temperature calculation:

  ```
  float dist;               // Distance from wall.
  float temp;               // Temperature at that distance.
  const float step = .005;  // Step size for loop.
  ```

  For step 2, we need a loop that will produce one line of output on each iteration. For each line, we must compute and print the temperature at the current distance, x, from the wall. We invent a function named `compute_temp()` to compute the temperature. It needs a parameter, x, which we declare as type `float`, like all the other variables in this program. The function result is a temperature, so that will also be type `float`. The prototype becomes:

  ```
  float compute_temp( float x );
  ```

  The revised function call graph is shown in Figure 9.13.

- *The brains.*
  Printing the detailed lines of the table requires a loop that starts at distance 0.0 from the wall and increases to the length of the fin in increments of the defined step size. Since the loop variable, `dist`, is not an integer, we need an epsilon value for the floating comparison that ends the loop. This epsilon must be smaller than the step size; we arbitrarily set it to half the step size:

  ```
  float epsilon = step/2.0;       // Tolerance
  ```

  We are ready to code the loop. We start with the loop skeleton and defer the computation and printout:

  ```
  for (dist = 0.0; dist < Length+epsilon; dist += step) {//Defer}
  ```

  Now we approach the loop body. The loop prints the lines of the table one at a time. For each one, we must compute the current distance, x, from the wall and the temperature at distance x. The distance computation is handled by the loop control; the remaining tasks are done by `compute_temp()` and `printf()`:

  ```
  temp = compute_temp( dist );
  printf( "%12.3f %24.1f \n", dist, temp );
  ```

  In the format, we use `%f` conversion specifiers because we want a neat table with the same number of decimal places printed for every line (3 for `dist` and 1 for `temp`, written with `%12.3f` and `%24.1f`, respectively). We supply a field width so that the output will appear in neat columns. To find the correct field width, we either count the letters in the headings or guess; a guess that is too big or too small can be adjusted after we see the output. We now have completed the first draft of the code for `print_table()`; it is shown in Figure 9.13.

- *Health check.*
  While coding the brains of a function, a programmer sometimes discovers that the function needs more information than its parameters supply. If that information is not supplied by global constants, the function header, prototype, and call(s) must be edited to supply the added data.

  In the process of writing `print_table()`, we did not need to change the parameter list, so the call on `print_table()` inside `main()` still is correct and we do not need to modify `main()` at this time. (However, we might need to do this later.)

  We have now completed a main program and one function with a tentative call on a second function that has not been written. It is time to compile again. We cannot compile the program as it stands because

```
// ------------------------------------------------------------
// Print a table of temperatures for a fin "Length" meters long.
// Length must be greater than 0.

void print_table( float Length )
{
    float dist;                   // Distance from wall.
    float temp;                   // Temperature at that distance.
    const float step = .005;      // Step size for loop.
    float epsilon = step/2.0;     // Tolerance

    printf( "\n Distance from base (m)    Temperature (C) \n"
    printf( " ----------------------    --------------- \n" );
    for (dist = 0.0; dist < Length + epsilon; dist += step) {
        temp = 1.1;      // compute_temp( dist );
        printf( "%12.3f %24.1f \n", dist, temp );
    }
    printf( " ----------------------    --------------- \n" );
}
```

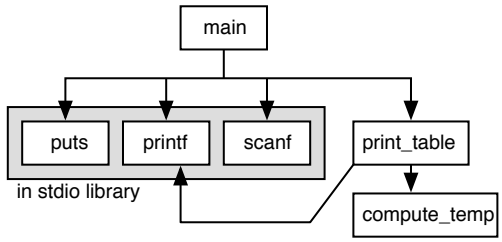


Figure 9.13. Fin program: First draft of `print_table()` function.

of the call on `compute_temp()`. Although we could write another function stub, in this case we choose to temporarily "amputate" the call on `compute_temp()` by enclosing it in a comment and setting the variable `temp` to an arbitrary (but recognizable) constant value:

```
temp = 1.1;  // compute_temp( dist );
```

Now we compile the program, fix any compilation errors, and run it. Our program ran successfully, producing the output that follows (only the first and last few lines are shown).

Looking at this table, we see that the number of rows printed is correct and the numbers are adequately centered under the headings. The numbers in the first column are correct and the numbers in the second column are the constant value we used when we amputated the call on `compute_temp()`.

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06

Distance from base (m)    Temperature (C)
----------------------    ---------------
      0.000                      1.1
      0.005                      1.1
      0.010                      1.1
       ...                       ...
      0.050                      1.1
      0.055                      1.1
      0.060                      1.1
```

Steps in writing `compute_temp()`.

- *The DNA and the skin.*
  The `compute_temp()` function must calculate the temperature of the fin at a given distance from the wall. Tentatively, we have given it one parameter, a `float` named x, which will range between 0 meters and the length of the fin. The function must return a `float` value to `print_table()`. We write a comment block and the shell of the function:

```
// ------------------------------------------------
// Compute and return the temperature at distance x from the wall.
// x must be between 0.0 meters and the length of the fin.

float compute_temp( float x ){// Code goes here.}
```

- *The skeleton and the brains.*
  We compute the temperature using the formula in the specifications (see Figure 9.11) and return the answer. This is a simple task that need not be broken down further. Here is the resulting statement:

```
return AirTm + (WallTm-AirTm) * cosh( FinCnow*(Length-x) ) / cosh( FinC*Length );
```

  Looking at this formula, we see that it involves several variables (wall temperature, air temperature, fin constant, and length of the fin), not just the distance x from the wall. However, the first three values are given as constant numbers in the problem specification. We could write constants for these quantities in our formula, but the program would be much more useful if these values could be varied. Therefore, we choose to define all three constants in `main()` and add them to the parameter lists of both `print_table()` and `compute_temp()`. By placing all these constants in `main()`, we make them easy to find and modify. It also would be easy to change them into input variables if that were desired. Last, the fin length is a missing parameter; it is read in `main()` and we must pass it from `main()` through `print_table()` to this function. Similarly, we must pass the constants from `main()` to `compute_temp()`. To do so, we need to change the prototypes and headers of both functions and correct two function calls.

- *Circulation.*
  We now have three constants and one input variable that must be passed from `main()` to `compute_temp()` as parameters. Although the order of the new parameters is not crucial, we want an order that makes sense and can be remembered, so we put all the properties of the fin together. Our new prototypes are

```
void print_table( float Length, float FinC, float WallTm, float AirTm );
float compute_temp( float x, float Length, float FinC, float WallTm, float AirTm )
```

  Next, we add three constant definitions to `main()` and change the call on `print_table()` to use them:

```
const float FinC = 26.5;    // Fin constant.
const float WallTm = 500.0; // Wall temperature, C
const float AirTm = 20.0;   // Air temperature, C
print_table( Length, FinC, WallTm, AirTm );
```

  Last, the function call in `print_table()` must also be changed:

```
temp = compute_temp( dist, Length, FinC, WallTm, AirTm );
```

- *Health check.* The completed program is shown in Figure 9.14. The first and last few lines of the output are shown below. The final development step is to compare these results to the ones in the test plan, to verify that we are fully finished.

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06

Distance from base (m)    Temperature (C)
----------------------    ---------------
      0.000                    500.0
      0.005                    445.5
      0.010                    398.5
       ....                    .....
      0.055                    209.6
      0.060                    208.0
----------------------    ---------------
```

The problem specifications are given in Figure 9.11.

```
#include <stdio.h>
#include <math.h>

void print_table ( float Length, float FinC, float WallTm, float AirTm );
float compute_temp( float x, float Length, float FinC, float WallTm, float AirTm );

int main( void )
{
    float Length;                   // Length of the cooling fin, in meters.
    const float FinC = 26.5;        // Fin constant.
    const float WallTm = 500.0;     // Wall temperature, C.
    const float AirTm = 20.0;       // Air temperature, C.

    puts( " Temperature Along a Cooling Fin" );
    do {
        printf( " Please enter length of the fin (> 0.0 m):  " );
        scanf( "%g", &Length );
    } while (Length <= 0);
    print_table( Length, FinC, WallTm, AirTm );
    return 0;
}
// ------------------------------------------------------------------
// Print a table of temperatures for a fin that is "Length" meters long.
// Length must be greater than 0.

void
print_table( float Length, float FinC, float WallTm, float AirTm )
{
    float dist;                     // Distance from wall.
    float temp;                     // Temperature at x.
    const float step = .005;        // Step size for loop.
    const float epsilon = step/2.0; // Tolerance.

    printf( "\n Distance from base (m)    Temperature (C) \n" );
    printf( " -----------------------    --------------- \n" );
    for (dist = 0.0; dist < Length + epsilon; dist += step) {
        temp = compute_temp( dist, Length, FinC, WallTm, AirTm );
        printf( "%12.3f %24.1f \n", dist, temp);
    }
    printf( " -----------------------    --------------- \n" );
}

// ------------------------------------------------------------------
// Compute the temperature at distance x from wall; 0<=x<=Length

float
compute_temp( float x, float Length, float FinC, float WallTm, float AirTm )
{
    return AirTm + (WallTm - AirTm) * cosh( FinC*(Length-x) ) / cosh( FinC*N );
}
```

Figure 9.14. Temperature along a cooling fin.

9.6 What You Should Remember

9.6.1 Major Concepts

Modular design. Large programs are organized into modules, which contain related sets of functions and constants. It is the compiler's job to properly compile and link together the files containing the modules. The organization of a module follows a stylistic pattern. Modular design is a skill worth learning if you intend to write programs of any substantial size.

Parameter passing. Most parameters are passed by value; that is, the value of the argument is copied into the parameter variable. This isolates the subprogram from the caller, making it impossible for the subprogram to change the values of the caller's variables. In Chapter 11, another parameter-passing mechanism (call by value/address) will be introduced that can support two-way communication between caller and subprogram.

Parameter and return value type conversion. The type of a function argument in a call should match the type of the corresponding formal parameter in the prototype. If they are not identical, the argument will be coerced (automatically converted) to the parameter's type, if that is possible. Any numeric type (including `char`) can be converted to any other numeric type. Similarly, the value returned by a function will be converted to the declared return type. Normally, this type coercion causes no trouble. However, converting from a longer representation to a shorter one can cause overflow or loss of precision and converting from `char` to anything except `int` usually is wrong.

Parameter names. The name of a parameter is arbitrary. It identifies the parameter value within the function and therefore should be meaningful, but it has no connection to anything outside the function, even if other objects have the same name. The order in which arguments are given in the call, not their names, determines which parameter receives each argument value.

Parameter order. The order of parameters for a function is arbitrary. However, related parameters should be grouped together, and when several functions are defined with similar parameters, the order should be consistent. The number and order of arguments are not arbitrary. Parameters and arguments are paired according to the order in which they are written (not by name or type).

Call graphs. A function call graph is a diagram that shows the caller-subprogram relationships of all the functions in a program. More sophisticated forms may include descriptions of the information being sent into and out of the subprogram.

Scope and visibility. The scope of an object is the portion of a program in which it exists. The visibility of an object is the portion of its scope in which it can be accessed. An object can have external, global, or local scope, meaning that it is available to the entire program, restricted to a single module, or restricted to a single function, respectively.

Stub testing. When a program is long and has many functions, stub testing often is the best way to construct and debug it. In this technique, the main program is written first, along with a stub for each function it calls. The stub is a function header, some comments, and only enough code to print the parameter values. If not a `void` function, it also must return some fixed and arbitrary answer. After the main function compiles and runs correctly by calling the stubs, the stubs are filled in, one at a time, with real code. This code, in turn, may require the construction of new stubs. After one or a few stubs have been fleshed out, the code is compiled and tested again. This is repeated until all stubs have been replaced by complete functions. This technique is important because it forces the programmer to work in a structured way; it ensures that all parts of the program are kept consistent with each other as they are developed. Also, compiler errors are easier to find and fix because there is never a large amount of new code being compiled for the first time.

9.6.2 Programming Style

Monolithic vs. modular. If you cannot look at a function on your computer screen and understand what it is doing, it is too long, too complex, or both. Keep function definitions short enough to see the beginning and the end at the same time. Generally, it is good to keep code for user interaction and code for mathematical computation in separate functions. Modular development also aids the compiling and debugging process.

Placement of prototypes in the file. The easiest way to be sure that the compiler uses the correct prototype to translate every function call is to write all the `#include` commands and all your prototypes at the top of each code module. Although other arrangements may be legal, according to C's rules, putting the prototypes at the top is the easiest way to avoid errors caused by misplaced prototypes, missing arguments, and incorrect argument order in function calls.

Global vs. local. Define variables locally, not globally, wherever possible. This tactic makes logical errors easier to locate and fix because it limits unintentional interaction between parts of a program. In general, parameters should be used for interfunction communication. Global definitions should be used only for new type definitions and constants shared by several parts of a program.

Function documentation. Every function should start with a highly visible comment line, such as a row of dashes. This line is very useful during debugging because it helps you find the functions quickly. You should be able to give a succinct description of the purpose of each function. This description should start on the second line of the comment at the top of the function definition. This comment also must make clear the purpose of each parameter and include a discussion of any preconditions.

Parameter and argument consistency. If more than one function uses the same set of parameters, reduce confusion by being consistent in their order and naming.

Function layout. The parts of a function definition can be arranged in many ways. The layout recommended here is the easiest to read and extends best to advanced programming in C++.

- Every function definition should start with a comment block, as described previously. The first line of code should be the return type and a comment, if needed, that describes the meaning of the return value (nothing else).
- The second line of code should start with the name of the function, followed by a left parenthesis. If all parameters will fit on one line, they should come next, ending with a right parenthesis. Otherwise, put each parameter on a separate line, with a comment. Put the closing right parenthesis on a line by itself, aligned directly under the matching left parenthesis.
- On the next line, write the left curly bracket in the first column.
- Next come the declarations with their comments; indent them two to four columns. Then leave a blank line and write the function body, indented similarly. Increase the indentation for each conditional or loop statement.
- Write the right curly bracket that ends the function in the leftmost column on a separate line.

9.6.3 Sticky Points and Common Errors

Parameter order. The arguments in a function call must be written in the same order as the matching parameters in the definition. If the order is scrambled, the code often will compile and run but produce incorrect results. Using an incorrect number of arguments can lead to confusing errors at compile time.

The declaration must precede the call. Either a function prototype or the complete function definition must precede all calls on the function. If this is not done, the compiler will construct a prototype automatically, which often will be wrong. This normally results in an error comment about an *illegal function redefinition* when the function definition is translated.

Call vs. prototype vs. definition. Beginning programmers often confuse the syntax of a function call with the syntax of the corresponding prototype and header. These have parallel but different forms. The function call supplies a list of argument values; although each value has a type, the type names are not written in the call. In contrast, the function prototype and header do not know what values eventually will be supplied by future calls, so they list the types of the parameters. In addition, the function header must give parameter names, so that the code can refer to the parameters and use them. Last, a semicolon is found at the end of a prototype but not at the end of a function header.

A global vs. local mix-up. Having global variables leads to trouble for a variety of reasons. One scenario is the following: A local variable declaration is omitted, and it happens to have the same name as a global variable. The compiler cannot detect the omission and will use the global variable. Any assignments to this variable will change the global value, causing unexpected side effects in other parts of the program. Such errors are hard to track down because they are not in the part of the program that seems to be wrong and produces incorrect results.

9.6.4 New and Revisited Vocabulary

A large number of terms relating to functions and function calls have been introduced in this chapter. This list is provided as a review of the new concepts covered.

main program	function call	return type
subprogram	calling sequence	return value
function	caller	scope
module	call graph	visibility
library	argument	accessibility
header file	call by value	local
prototype	type coercion	global
function definition	arithmetic types	stack frame
interface	call by address	external
formal parameter	address argument	preconditions
parameter list	entry point	function skeleton
parameter order	return address	function stub
type matching	`return` statement	stub testing
type conflict	missing prototype	testing by amputation

9.7 Exercises

9.7.1 Self-Test Exercises

1. Call graph. The main program, which follows, calls two of the functions declared at the top. Which standard libraries would need to be included to compile this program? Draw a function call graph for this program.

```
#include <stdio.h>
#define MAX 7

void pattern( int p );
void stars( int n ) {  for (int k = 0; k < n; ++k) printf( "*" ); }
void space( int m, int n ) {  for (int k = 0; k <= n; k += 2) printf( " %i", m ); }
int odd( int n ) { return n % 2; }

int main( void ) //--------------------------------------------------
{
    for (int k = 0; k < MAX; ++k) {
        if (k < 2 || k >= MAX-2) stars( MAX );
        else pattern( k );
        printf( "\n" );
    }
}

void pattern( int p ) //---------------------------------------------
{
    stars( 1 );
    space( p, MAX-4 );
    if (odd(MAX)==1) printf( " " );
    stars( 1 );
}
```

2. Tracing calls. Trace the execution of the code in problem 2. Make a list of the function calls, one per line, listing the name, arguments, and return value for each. Show the program's output as well.
3. Local and Global. Look on the web site at the program for Newton's method. Fill in the following chart listing the symbols *defined* (not used) in each program scope. List the global symbols on the first line and add one line below that for each function in the program; remember that every function definition creates a new scope. The line for `main()` has been started for you.

Scope	Parameters	Variables	Constants
`global`	—		
`main()`	—		

4. Local names. List all the local symbols defined in the main program in Figure 9.4. List the parameters and local variables in the function named `n_marks()`.
5. Visibility. List all the variable names used in the function named `cyl_vol()` in Figure 5.24. For each, say whether it is a parameter or a local variable.
6. Control flow. Draw a flow diagram for the fin temperature program in Figure 9.14.
7. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and meaningful. If the call would cause a type conversion of either an argument or the return value, say so. If the call has an error, fix it.

```
void   squawk( int );
int    triple( int );
double power( double, int );

int j, k;
float f;
double x, y;
```

(a) `squawk( 3 );`
(b) `squawk( f );`
(c) `triple( 3 );`
(d) `f = triple( k );`
(e) `j = squawk( k );`
(f) `y = power( 3, x );`
(g) `y = power( x, 3 );`
(h) `x = power( double y, int k );`
(i) `y = power( triple( k ), x );`
(j) `printf( "%i %i", k, triple( k ) );`

9.7.2 Using Pencil and Paper

1. Control flow. Draw a flow diagram for the Newton's method program on the text web site.
2. Call graph. Draw a call graph for the `n_marks` program in Figure 9.4.
3. Tracing calls. Trace the execution of the program in exercise 4, above. Make a list of the function calls, one per line, listing the name, arguments, and return value for each. Show the program's output as well.
4. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and meaningful. If the call would cause a type conversion of either an argument or a return value, say so. If the call has an error, fix it.

```
double rand_dub( void );
int half( double );
int series( int, int, double );

int j, k;
float f;
double x, y;
```

(a) `half( 5 );`
(b) `rand_dub( y );`
(c) `x = rand_dub();`
(d) `j = half();`
(e) `f = half( x );`
(f) `j = series( x, 5 );`
(g) `j = series( 5, (int)x, y );`
(h) `y = series( j, k, rand_dub() );`
(i) `printf( "%g %g", x, half( x ) );`
(j) `printf( "%i %g", k, rand_dub( k ) );`

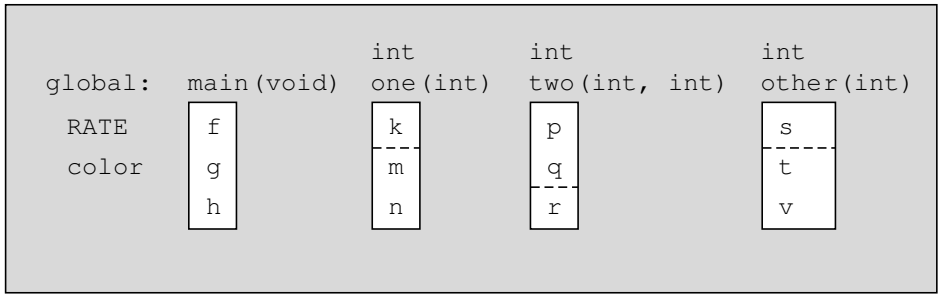
5. Call graph. The main program that follows calls the three functions defined at the top. Which standard libraries would need to be included to compile this program? Draw a function call graph for this program.

```
#include <stdio.h>
#include <math.h>

double f( double x ) { return x / 2.0; }
double g( double x ) { return 1.0 + x; }
double h( double x ) { return x * 3.0; }

int main( void )
{
    double x = 1;
    double sum = 0.0;

    while (sum < 100) {
        x = h( x );
        printf( " %6.2f \t", x );
        sum += x;
        if (fmod( x, 2.0 ) == 0) x = f( x );
        else x = g( x );
        printf( " %6.2f \n", x );
    }
    printf( " ------\n %6.2f \n", sum );
}
```

6. Local and Global. Look at the program for fin temperatures in Figure 9.14. Fill in the following table, listing the symbols that are *defined* (not used) in each program scope. List the global symbols on the first line and add one line below that for each function in the program (the line for `main()` has been started for you).

Scope	**Parameters**	**Variables**	**Constants**
`global`	—		
`main()`	—		

7. Visibility. The diagram that follows depicts a main program with three functions: `one()`, `two()`, and `other()`. Within the box for each function, parameters are shown above the dashed line, local variables below it. All functions return an `int` result. All variables and parameters are type `int`. The global constant, `RATE`, and global variable, `color`, also are type `int`.



For each function call shown that follows, say whether it is legal or illegal. Fix any illegal statements.

(a) In `main()`: `f = one( RATE );`.
(b) In `main()`, after calling `one()`: `f = two( g, k );`.
(c) In `one()`: `n = two( m );`.
(d) In `one()`: `n = two( color, m );`.
(e) In `one()`: `n = other( k );`.
(f) In `two()`, after being called from `one()`: `r = other( k );`.

9.7.3 Using the Computer

1. A function.

   Write a function to compute the formula

   $$f(x) = (3x + 1)^{\frac{1}{2}}$$

   Write a main program that will sum $f(x)$ for the values $x = 0$ to $x = 1000$ in steps of 50. Print out the value of $f(x)$ and the current sum at every step in a nice neat table.

2. Tables.

   Write a program that contains two function definitions:

   $$f1(n, x) = e^{\sqrt{nx}} \times sin(nx)$$
   $$f2(n, x) = e^{\sqrt{nx}} \times cos(nx)$$

   where n is an integer and x is a `double`. In the main program, input a value for x and restrict it to the range $0.1 \leq x \leq 2.5$. Print a neat table showing the values of $f1(n, x)$ and $f2(n, x)$ as n goes from 0 to 30. Print column headings above the first line. Show all numbers to three decimal places.

3. Bubbles.

   Modify your program from computer exercise 3 in Chapter 7; change the `bubble()` function so that it takes both σ and r as parameters. Then change the `main()` function to ask the user to enter a value for σ as well. Define an error function that prints a message "Input out of range." Use it to screen out values of σ less than 0.001 or greater than 0.003 lb/ft and values of r less than 0.0002 or greater than 0.015 ft. Call this function from `main()`.

4. An arithmetic series.

   Each term of an arithmetic series is a constant amount greater than the term before it. Suppose the first term in a series is a and the difference between two adjacent terms is d. Then the kth term is

$t_k = a + (k-1)d$. Write a function `term()` with three parameters, a, d, and k, that will return the kth term of the series defined by a and d. Use type `long int` for all variables. Write a program that prompts the user for a and d, then prints the first 100 terms of the series, with 5 terms on each line of output, arranged neatly in columns. Quit early if integer overflow occurs.

5. A geometric series.

   A *geometric progression* is a series of numbers in which each term is a constant times the preceding term. The constant, R, is called the *common ratio*. If the first term in the series is a, then succeeding terms will be aR, aR^2, $aR^3, \ldots$ The kth term of the series will be $t_k = aR^{k-1}$. Write a function `term()` with three parameters (a, R, and k) that will return the kth term of the series defined by a and R. Use type `double` for all variables, since the terms grow large rapidly when R is greater than 1. Write a main program that prompts the user for a and R, then prints the terms of the series they define until either 50 terms have been printed, 5 terms per line, or floating-point overflow or underflow occurs.

6. Torque.

   Given a solid body, the net torque T (Nm) about its axis of rotation is related to the moment of inertia I and the angular acceleration acc (rad/s^2) according to the formula for the conservation of angular momentum:

   $$T = I \cdot acc$$

   The moment of inertia depends on the shape of the body; for a disk with radius r, it is

   $$I = 0.5mr^2$$

   (a) Write a function named `moment()` to calculate and return the moment of inertia of a solid disk. It should take two parameters: the radius and mass of the disk.

   (b) Write a `work()` function that will prompt the user to enter the radius, mass, and angular acceleration of a disk. Limit the inputs to be within these ranges:

   $$0.093 \le r \le 0.207$$
   $$0.088 < m \le 11$$

   Compute and print the torque of the disk, calling `moment()` to compute I first.

   (c) Write a main program that will allow the user to compute several torques.

7. An AC circuit.

   An AC circuit that you have designed is operating at a voltage V (rms volts) alternating at a frequency f (cyc/s). It is constructed of a capacitor C (farads), inductor L (henrys), and a resistor R (ohms) in series. Some important properties of this circuit are

Impedance:	$Z = \left[R^2 + \left(2\pi fL - \frac{1}{2\pi fC}\right)^2\right]^{0.5}$	(ohms)
Current:	$I = \frac{V}{Z}$	(rms amps)
Power used:	Power $= \frac{V \times I \times R}{Z}$	(watts)

   Write a function to compute an answer for each formula. Assume that f is a constant 120 cyc/s and $C = 0.00000001$ farads. Then write a main program that will input values for V, L, and R and output the impedance, current, and power used. Validate the inputs and ensure that they are within the following ranges:

   $$60 \le V \le 200$$
   $$0.1 \le L \le 10$$
   $$100 \le R \le 1000$$

8. Ice cream cones.
Suppose you are a professional party planner. Given the number of guests expected, you must plan a menu and deliver enough food to serve the crowd. In this problem, you will write a program to calculate how many packages of ice cream must be bought to fill one ice cream cone for each guest. The guest will select the size of the cone (diameter, height). Your suppliers sell ice cream in containers of various sizes and shapes.

   Write a function named `cone()` that will prompt for and read the diameter, d, and the height, h, of the cone to be used, then calculate and return the volume of ice cream needed to fill the cone. Assume that the cone part will be filled entirely and there will be a hemisphere of ice cream on top. The formulas are:

$$\text{Volume of cone} = \frac{\pi \times d^2 \times h}{12}$$

$$\text{Volume of hemisphere} = \frac{\pi \times d^3}{12}$$

   Ice cream comes in cartons that are either the shape of a barrel or a box. Write a function named `carton()` that will prompt for an alphabetic code, R for "barrel" or X for "box". Use a switch statement to execute the appropriate input and computation instructions, then return the volume of the selected carton. For a barrel, input the diameter and height; for a box, input the length, width, and height. Use one of these formulas:

$$\text{Volume of barrel} = \frac{\pi \times diameter^2 \times height}{4}$$

$$\text{Volume of box} = length \times width \times height$$

   In your main program, prompt for and input the number of guests, then call your `cone()` function and your `carton()` function. Finally, compute and display the number of cartons of ice cream you must buy to fill all those cones. Use the `ceil()` function to round up to a whole number of cartons.

9. Wedding cake.
Another common party food is wedding cake. Given the number of guests expected at the wedding, write a program to calculate how many layers your cake must have to serve everyone, and how much that cake will cost.

   The cake will be square and have two or more tiers. The top tier will be 6" wide and will not be eaten at the wedding reception. Each other tier will be 2" thick and 4" wider than the tier above it. Guests will be served from layers 2, 3, 4. . . ; each serving will be 2" square.

   Write the following one-line functions:

   - `width()`: calculate the width of a tier given the layer number (layer 1=6", layer 2 = (6+4)", layer 3 = (6+4+4)", etc.).
   - `servings()`: calculate the number of servings a tier will provide, given the layer number. Call `width()`.
   - `price()`: the top tier and decorations cost $40.00; the other tiers cost $1.00 per serving. (This will be more than $1.00 per guest because part of the bottom tier will be left over.)

   Write a function named `layers()` that will calculate how many tiers are needed. Hint: start with 1 tier, which serves 0 people. Use a loop to call `servings()` and add tiers until you have accumulated enough portions to serve the party. In general, you will end up with more than enough servings, since part of the last tier will be left over.

   In your main program, prompt for and read the number of guests, then call the `layers()` and `price()` functions to calculate the size and cost of the cake. Print out these answers.

10. Scheduled time of arrival.
Airline travelers often want to know what time a flight will arrive at its destination in the local time of the destination. This can be calculated given the following data:

   - The scheduled takeoff time, in hours and minutes on a 24-hour clock. (Valid hours are 0. . . 23, valid minutes are 0. . . 59)

- The scheduled duration of the flight, in hours and minutes.
- The number of time zone boundaries the flight will cross. This number should be negative if traveling from East to West, positive if going West to East.
- Whether the international date line will be crossed. This number should be +1 if it is crossed traveling from East to West, −1 if crossed while going West to East, and 0 if it is not crossed. Legal values are in the range -23. . . 23.

Using top-down design, write a program with functions that will make this calculation for a series of flights. For each flight, your program must input these data values from the user and print the scheduled time at which the flight should arrive at its destination. This time is calculated as follows:

- Starting with the takeoff time, add or subtract an hour for each time-zone change.
- Then add the duration of the flight to this time.
- Finally, adjust the time by adding or subtracting a day if the flight crossed the international date line.
- Use integer division and the modulus operator to convert minutes to hours + minutes, and hours to days + hours.

Print the local time of arrival using a 24-hour clock. Also print `-1 day` if the flight will land the day before it took off or `+1 day` if it will land the day after it took off (both are possible).

Suggestion: Write a function that will input, validate, and return one integer. It should have two integer parameters: the minimum and maximum acceptable values for the input.

Chapter 10

An Introduction to Arrays

The data types we have studied so far have been simple, predefined types and variables of these types are used to represent single data items. The real power of a computer language, however, comes from its ability to define complex, multipart, structured data types that model the complex properties of real-world objects. For example, to model the periodic table of the elements, we would need a collection of 110 objects that represent elements; each object would have several parts (name, symbol, atomic number, atomic weight, etc.). We call such types *compound types* or **aggregate types**. An array is an aggregate whose parts are all the same type. In this chapter, we study how to define, access, and manipulate the elements of an array. The last half of the chapter presents simple, important array algorithms.

10.1 Arrays

In many applications, each data item is processed once, just after it is read, and never needs to be used again. In such programs, an array can be used to store the data, but it is not necessary. In contrast, some programs must read all the data, perform a calculation, then go back and process the data again. In these programs, we must store all the data between reading it and reprocessing it.

Consider the problem of assigning grades to a class based on the average score of an exam. If we were computing only the exam average, this could be done without storing the data in an array; all that is needed is a summation loop. However, to assign a grade, we must have both the exam score and the average. The scores must be processed once to compute the average, then we must go back and reprocess the scores to assign the grades. In between we need to store the values.

We could do this using individual variables. For instance, in the example of computing the average of three numbers in Figure 2.7, we used three separate variables to hold the numbers. While this works well for only three numbers, it does not work on larger data sets. Imagine how tedious it would be to write a `scanf()` statement and a formula with many variable names. We can solve this problem by using an array. For example, a program to compute the average temperature over a 24-hour period might store 24 temperature readings in an array named `temperature`. An array used to determine the average high temperature for one year would have 365 (or 366) entries, each containing a daily high temperature.

An **array** is a consecutive series of variables that share one variable name. We will call these variables the **array slots**; the data values stored there are called the **array elements**. The number of slots in a *one-dimensional array* is called the **array length**. Arrays with two or more dimensions also can be defined; these will be studied in Chapter 18. The variables, or slots, that form an array have a uniform type called the **base type**.

An array object, as a whole, is given a name. We can refer to the entire array by this name or to an individual slot by appending a number in square brackets to the name. This number is called the *subscript*. A **subscript** is an integer expression enclosed in square brackets that, when written after an array name, designates a particular slot in the array.

In C, all arrays start with slot 0 (rather than 1) because it is easier and more efficient for the system to implement. This means that the first element in an array named `ary` would be called `ary[0]`; the next one would be `ary[1]`. If this array had six elements, the last one would be `ary[5]`.

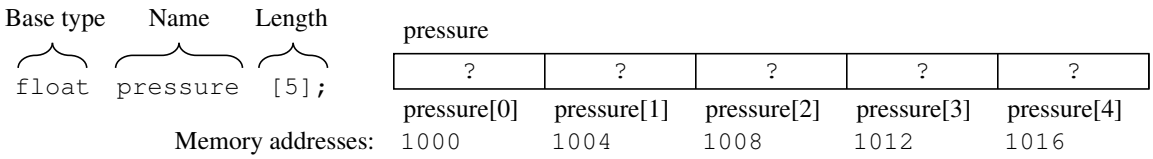


Figure 10.1. Declaring an array of five floats.

10.1.1 Array Declarations and Initializers

Declarations. An array variable is declared and initialized very much like a simple variable. The **array declaration** starts with the base type, which can be any type—simple or compound. Thus, we can have an array of `int`s, an array of `char`s (also known as a *string*), or even an array of arrays. Following the base type in the declaration is the array name and a pair of square brackets enclosing the length, which must be an integer constant or an integer **constant expression** (an expression with only constant operands). The length determines the number of slots in the array. Figure 10.1 shows the declaration for an array named `pressure` containing five `float`s. This creates a series of five `float` variables that we can refer to as `pressure[0]`, `pressure[1]`, `pressure[2]`, `pressure[3]`, and `pressure[4]`. These five `float`s will be stored in a contiguous set of memory locations with `pressure[0]` at the location with the lowest memory address, 1000, in this case. In later chapters, the address of each slot may also be of interest; if so, we write the address above or below the slot, as shown in Figure 10.1.

To diagram an array, we draw a row of connected boxes that are the right size for the base type. We write the name of the array above and the subscripts below this row. If there is an initializer, as in Figure 10.2, we copy the initial values into the boxes. Otherwise, as in Figure 10.1, we leave them blank or write a question mark.

Initial values. An array can be declared with no initial values, like the array named `pressure` in Figure 10.1. The contents of an uninitialized array are as unpredictable as ordinary variables.[1]

Alternatively, it may have an **array initializer**, which is a list of values enclosed in curly brackets. The values will be stored in the array slots when the array is created, as illustrated in Figure 10.2. The values in the initializer list must be constants or constant expressions. The types of values in the initializer must be appropriate for the base type of the array.[2]

If an initializer *is* given, the array length may be omitted from the square brackets. The C translator will count the number of initial values given and use that number as the length; exactly enough space will be allocated to store the given values. Note the absence of the length value in the declaration for `temperature` in Figure 10.3.

What if both a length and an initializer are given and the sizes do not match? This is an error if the initializer contains too many values; the compiler will detect this error and comment on it. However, if an initializer list is too short, it is not an error. In this case, the values provided are used for the first few array slots and the value 0 is used to initialize all remaining slots. The declaration for `inventory` on the third line

[1]Global arrays and static local arrays are initialized to 0 values.
[2]The initializer type must match or be coercible to the array base type.

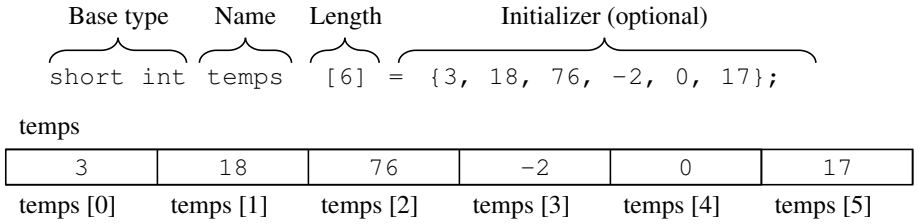


Figure 10.2. An initialized array of short `int`s.

Base type | Name | Length | Initializer (optional)

```
short int          age   [6] ;
short int   temperature   [] = {3, 18, 76, -2, 0, 17};
short int     inventory   [6] = {1};
int             instock   [6] = {};
```

age

[0]	[1]	[2]	[3]	[4]	[5]

temperature

[0]	[1]	[2]	[3]	[4]	[5]
3	18	76	-2	0	17

inventory

[0]	[1]	[2]	[3]	[4]	[5]
1	0	0	0	0	0

instock

[0]	[1]	[2]	[3]	[4]	[5]
0	0	0	0	0	0

Figure 10.3. Length and initializer options.

in Figure 10.3 illustrates an initializer that is shorter than the array. It also is possible to initialize an entire array to zeros by supplying an empty set of brackets, as in the initializer for `instock` in Figure 10.3.

Arrays of characters. There are two ways to initialize an array of characters: with a comma-separated series of char literals, or with a string literal, as shown in Figure 10.4. It is certainly much easier to use a one string literal than to write a lot of pairs of single quote marks. Note, however, that the string initializer puts one extra character into the array that is not visible in the code: a null character that marks the end of the string. This will be explained in more detail in Chapter 12.

10.1.2 The Size of an Array

Two different aspects of an array's size are important: its length and the total number of bytes of memory required to store it. When we use `sizeof` with an array variable, we get the number of bytes, which is the product of the array's length and the size of one element of its base type.

While knowing the number of slots is necessary to write an array declaration, a programmer will not always know the **size of the array**, because that depends on the size of the base type, which may vary from one machine to another. The program in Figure 10.5 shows the sizes of the arrays declared in Figures 10.1 and 10.2, and two arrays of chars. An output from this program is

```
pressure:  sizeof(float)  is 4 * length 5 = sizeof array 20
```

```
char  vowels[]    = {'a','e','i','o','u'};
char  operators[] = { "+-*/%" };
```

vowels

'a'	'e'	'i'	'o'	'u'

operators

'+'	'-'	'*'	'/'	'%'	'\0'
[0]	[1]	[2]	[3]	[4]	[5]

Figure 10.4. Initializing character arrays.

```
#include <stdio.h>

#define DIMP 5    // Lengths of the arrays.
#define DIMT 6
#define DIMV 3

int main( void )
{
    float pressure[DIMP] = { .174, 23.72, 1.111, 721.2, 36.3 };
    short int temps[DIMT] = { 3, 18, 76, -2 };
    double vec2[DIMV] = { 2.0, 0.0, 1.0 };
    char vowels[] = 'a','e','i','o','u';
    char operators[] = "+-*/%";

    printf( " pressure:   sizeof(float)  is %i * length %i ="
            " sizeof array %i \n", sizeof(float), DIMP, sizeof(pressure) );
    printf( " temps:      sizeof(short)  is %i * length %i ="
            " sizeof array %i \n", sizeof(short), DIMT, sizeof(temps) );
    printf( " vec2:       sizeof(double) is %i * length %i ="
            " sizeof array %i \n", sizeof(double), DIMV, sizeof(vec2) );
    printf( " vowels:     sizeof(char)   is %li sizeof vowels is %li\n",
            sizeof(char), sizeof(vowels) );
    printf( " operators:  sizeof(char)   is %li sizeof operators is %li \n",
            sizeof(char), sizeof(operators) );
    return 0;
}
```

Figure 10.5. The size of an array.

```
temps:     sizeof(short)  is 2 * length 6 = sizeof array 12
vec2:      sizeof(double) is 8 * length 3 = sizeof array 24
vowels:    sizeof(char)   is 1   sizeof vowels is 5
operators: sizeof(char)   is 1   sizeof operators is 6
```

Often, the first portion of an array will hold data, while the last portion is not in use. This happens when the array is intended to hold a variable amount of information and its length is set to the maximum length that might ever be needed. Even in this case, the size returned by `sizeof` is the total amount of memory allocated including both the portion in use and the unused portion. This is illustrated by the array `temps` in Figure 10.5.

When an array is passed as an argument to a function, (see Section 10.4) the size information does not travel along with it. No operation in C will give us the actual length of an array argument inside a function. If you apply `sizeof` to an array parameter, the result always will be the number of bytes needed to store the starting address of the array. Unfortunately, an array-processing function frequently needs the information to work properly. For this reason, the programmer, who knows the array's length when it is declared, must make the information available for use by every part of the program that operates on the array. One way to do this is to declare the array length using a `#define` at the top of the program, as in our first several examples.

10.1.3 Accessing Arrays

The elements of an array can be accessed in two ways: by using pointers or by using subscripts. Both ways are important in C and need to be mastered. Pointers often are used when the slots of an array will be used in sequential numeric order, because this technique can lead to greater efficiency. While subscripts can be used for this purpose, too, they are better at accessing the elements in a random order. Since pointer processing techniques are harder to master than those based on subscripts, using pointers will be deferred until Chapter 17, while subscripting is explained in this chapter.

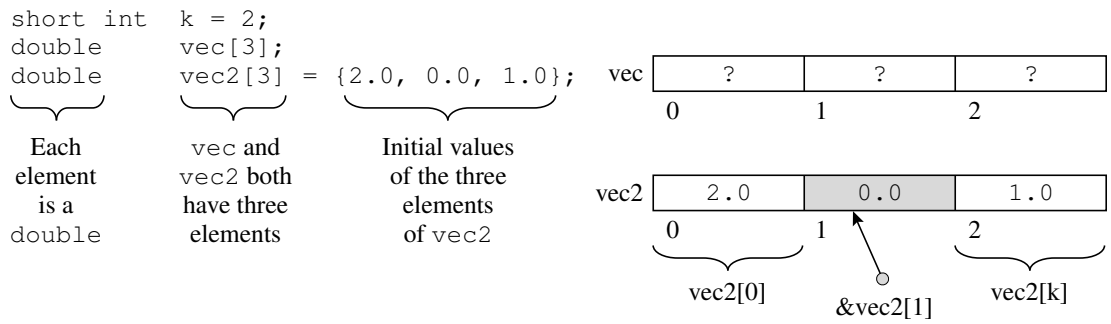


Figure 10.6. Simple subscripts.

Subscripts. Figure 10.6 shows how constant subscripts are interpreted. It shows two arrays named `vec` and `vec2`, which represent vectors in a three-dimensional space. The first slot of `vec2` is `vec2[0]` and represents the x-component of the vector, containing the value 2.0. The y-component has the value 0.0 and is stored in `vec2[1]`, the shaded area in the diagram. In an object diagram, we use an arrow to represent an address. The arrow in Figure 10.6 represents `&vec2[1]`, the address of the shaded area. When both an ampersand and a subscript are used, the subscripting operation is done first and the "address of" operation is applied to the single variable selected by the subscript.[3] These addresses are important when using arrays as function arguments, as demonstrated in Section 10.4.

We also can use an expression involving a variable to compute a subscript (Figure 10.7). A subscript expression can be as simple as a constant or as complex as a function call, as long as the final value is a nonnegative integer less than the length of the array. The most frequently used subscript expression is a simple variable name, which also is illustrated in Figure 10.6. The phrase `vec2[k]` means that the current value of `k` should be used as a subscript for the array. Since `k` contains a 2 here, `vec2[k]` means `vec2[2]`, which contains the value 1.0. The flexibility and versatility of these subscript rules enables a variety of powerful array-processing techniques.

Notes on Figure 10.7. Computed subscripts. Figure 10.6 demonstrates how we can use computed subscripts. It depicts an array containing the ages of N members of a family: father, mother, and children, in order of age. The array length is `#define`d. Often this is done so that it can be used for calculations throughout the code yet modified easily if the program's needs change.

The variable `pos_mother` is used here to store the position in the array of the mother. In the last line on the left, the expression `pos_mother+1` is used to find the age of the oldest child. Also, we often need to use the

[3]Appendix **??** contains a precedence table that includes both subscript (`[]`) and address of (`&`) operators. Note that the precedence of subscript is higher, and therefore, the subscript will be applied to the array name before the address operator.

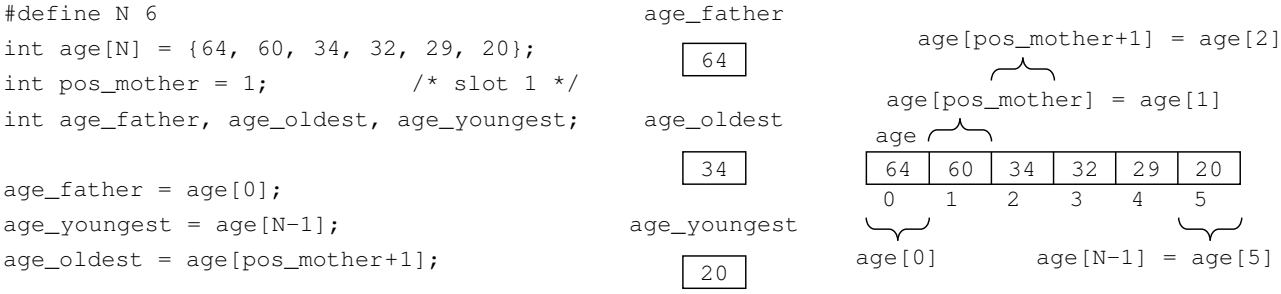


Figure 10.7. Computed subscripts.

This brief demonstration program shows how to do input, output, and computation with the elements of an array.

```
#include <stdio.h>
#include <math.h>

int main( void )
{
    double vec[3];      // A vector in 3-space.
    double magnitude;

    puts( "\n Subscript Demo: The Magnitude of a Vector" );

    printf( " Please enter the 3 components of a vector: " );
    scanf( "%lg%lg%lg", &vec[0], &vec[1], &vec[2] );

    magnitude = sqrt( vec[0] * vec[0] + vec[1] * vec[1] + vec[2] * vec[2] );

    printf( "    The magnitude of vector ( %g, %g, %g ) is %g \n",
            vec[0], vec[1], vec[2], magnitude );
    return 0;
}
```

Figure 10.8. Subscript demo, the magnitude of a vector in 3-space.

subscript of the last element in an array. To compute this, we subtract 1 from the array length. Therefore, `age[N-1]` is the age of the last (youngest) child in the family.

The example in Figure 10.8 declares an array variable named `vec` and shows simple input, computation, and output statements that operate on the array elements.

Notes on Figure 10.8. Subscript demo, the magnitude of a vector.

First box: input into an array. When we call `scanf()` to read the value of a variable, we write `&` before the variable's name to refer to its address. Similarly, we can use `&` to refer to the address of a single slot of an array. To read just one value into the first component, we would write `scanf( "%lg", &vec[0] );`.

Second box: computation on array elements.

- We can use a subscripted array name like a simple variable name in any expression.
- Here, to compute the magnitude of a vector, we add the squares of the three components, take the square root of the sum, and store the result in `magnitude`. The easiest and most efficient way to square a number is to multiply it by itself.

Third box: output from an array.

- To print an array element, give the array name and the subscript of the element.
- You cannot print the entire contents of an array with just one format field specifier. Here, we use three separate `%g` specifiers to print the three array elements. A loop would be used to print all the elements of a large array,
- The output from two runs of this program is

```
Subscript Demo: The Magnitude of a Vector
Please enter the 3 components of a vector: 1 0 1
   The magnitude of vector ( 1, 0, 1 ) is 1.41421
--------------------------------------------------

Subscript Demo: The Magnitude of a Vector
Please enter the 3 components of a vector: 1 2 0
   The magnitude of vector ( 1, 2, 0 ) is 2.23607
```

```
#include <stdio.h>
#define N 3

int main( void )
{
    float dimension[N];     // Dimensions of a box.
    float volume;           // The volume of the box.

    printf( "\n Array Input Demo: the Volume of a Box \n"
            " Please enter dimensions of box in cm when prompted.\n" );

    for (int k = 0; k < N; ++k) {  // End loop when k reaches length of array.
        printf( " > " );
        scanf( "%g", &dimension[k] );
    }

    volume = dimension[0] * dimension[1] * dimension[2] / 1e6;
    printf( " Volume of the box ( %g * %g * %g ) is %g cubic m.\n\n",
            dimension[0], dimension[1], dimension[2], volume );
    return 0;
}
```

Figure 10.9. Filling an array with data.

10.1.4 Subscript Out-of-Range Errors

One important reminder and caution: It is *up to the programmer* to ensure that all subscripts used are legal. C does not help you confine your processing to the array slots that you defined. C uses the subscript value to compute a memory address called the **effective address** according to this formula:

$$\texttt{effective_address} = \begin{array}{l}\text{address of beginning of the array} + \\ \text{subscript} \times \texttt{sizeof}\ (\text{base type of array})\end{array}$$

If you use a subscript that is negative or too large, C will compute the theoretical "address" of the nonexistent slot and use that address even though it will not be between the beginning and the end of the array. C does absolutely no subscript range checking. The compiler will give no error comment and there will be no error comment at run-time either. Your program will run and either access a memory location that belongs to some other variable or attempt to access a location that does not exist.

10.2 Using Arrays

A common array processing pattern involves a counting loop, where the loop variable starts at 0 and stops before `N`, the length of the array. The loop variable is used as a subscript to access each array element, in turn. This kind of loop is seen again and again in programs that use arrays to process large amounts of data. Often, one loop is used to read data into the array, another to process the data items, and a third loop to print them all.

10.2.1 Array Input

The best way to read data into an array is with a `for` loop, where the loop counter starts at 0, increments through the subscripts of the array, and ends at `N`, the declared length of the array. The loop does not try to process slot `N`, which is past the end of an `N`-element array. This simple idiom is illustrated in Figure 10.9.

Notes on Figure 10.9. Filling an array with data.

This program is a modification of the vector magnitude program in Figure 10.8. It demonstrates how an incorrect loop can destroy the value of a variable and result in unpredictable behavior.

```
#include <stdio.h>

int main( void )
{
    int k;              // Loop counter.
    float v[3];         // A vector in 3-space.
    float sum;          // The sum of the squares of the components.
    float magnitude;

    puts( "\n Subscript Demo: Walking on Memory" );
    printf( " Please enter one float vector component at each prompt.\n" );

    for (sum = 0.0, k = 0; k <= 3; ++k) {       // Loop goes too far.
        printf( "\tv[%i]: ", k );
        scanf( "%g", &v[k] );
        sum += v[k] * v[k];
    }
    magnitude = sqrt( sum );
    printf( "    The magnitude of vector ( %g, %g, %g ) is %g \n",
            v[0], v[1], v[2], magnitude );
    return 0;
}
```

Figure 10.10. Walking on memory.

Outer box: the* `for` *loop. Since the loop counter takes on the values from 0 to `N` and the loop ends when `k == N`, all `N` array slots (with subscripts `0...N-1`) will be filled with data.

Inner box: reading data into one slot. Within the loop, a `scanf()` statement reads one data value on each iteration and stores it in the next array slot. Note that both an ampersand and a subscript are used in the `scanf()` statement to get the address of the current slot.

Sample output.

```
Array Input Demo: the Volume of a Box
Please enter dimensions of box in cm when prompted.
> 100
> 100
> 100
Volume of the box ( 100 * 100 * 100 ) is 1 cubic m.
```

10.2.2 Walking on Memory

A loop that runs amok can cause diverse kinds of trouble. One common outcome is that variables that occupy nearby memory locations are overwritten with information that was supposed to go into one of the array's slots. Afterward, any computation or output that uses these variables will be erroneous.

If you write a loop to print the values in an array and it loops too many times, you will start printing the values stored adjacent to the array in memory. When storing data, you will start erasing other information when the loop exceeds the array bounds. This is demonstrated by the program in Figure 10.10 (a modification of the program in Figure 10.8), which reads input into an array named `v`. Figure 10.11 is a diagram of memory for this program. It shows the variables and the memory addresses that a typical compiler might assign. The array is colored gray.

The upper diagram in Figure 10.11 shows the values of the variables just after the third trip around the loop. All the array slots have been filled, and `k` has been incremented to 3 and is ready for the loop exit test.

These diagrams illustrate the contents of memory while processing the input sequence described in the text.

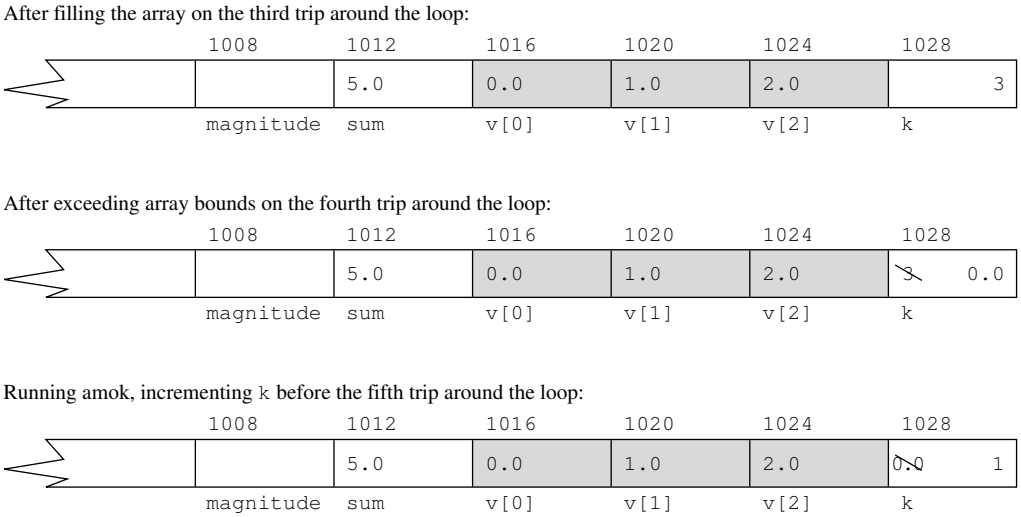


Figure 10.11. Before and after walking on memory.

However, since the test was written incorrectly (with a `<=` operator instead of a `<` operator), the loop does not end.

The array has three slots, but the loop was written to execute four times, with subscripts 0 through 3. The last time, the effective address calculated for `v[k]` actually is the address of `k`, and the input value, wrongly, is stored on top of the loop counter. The output from this run had more lines than expected:

```
Subscript Demo: Walking on Memory
Please enter one float vector component at each prompt.
       v[0]: 0.0
       v[1]: 1.0
       v[2]: 2.0
       v[3]: 0.0
       v[1]: 5.0
       v[2]: -1.0
       v[3]: 4.5
Segmentation fault
```

The middle diagram shows what happens during the next loop iteration, as the subscript goes beyond the end of the array. The input value of 0.0 is stored in the variable that follows the array, which is the memory location used for the loop counter. This destroys the value of the loop counter and leaves the input value in its place. In this example, the input is 0, which then gets incremented to 1 by the `for` loop (bottom diagram). The loop still does not terminate, because now `k` is 1. It continues taking input until some input value is stored in `k` that satisfies the loop exit test or until an abnormal termination happens, as occurs here.

In this example, storing input on top of the loop counter causes unpredictable behavior that depends on the data entered by the user. Usually, as in the output shown, the program crashes; other times it continues and terminates normally but produces erroneous results. Be sure to check the limits of array processing loops to minimize these problems.

Random locations and memory faults. Sometimes, a faulty subscript causes the program to try to use a memory location that is not legal for that program to access; the result is an immediate hardware error (a memory fault, bus error, or segmentation fault) that terminates the program. Usually, the system displays a message to this effect, as was seen in the last program.

Problem scope: Given the ID number and two exam scores (midterm and final) for each student in a class, compute the weighted average of the two scores. Also, compute the overall class average and the difference between that and each student's average.

Input: ID numbers will be long integers and exam scores will be integers.

Limitations: The class cannot have more than 16 students.

Formula: Weighted average = 0.45 × midterm score + 0.55 × final score

Output and computational requirements: All inputs should be echoed. In addition, for each student, print the exam average and the difference between that average and the overall average for the class, both to one decimal place. The overall exam average of the class should be printed using two decimal places.

Figure 10.12. Problem specifications: Exam averages.

If a program continues to run after a subscript error, it generally produces erroneous output. The cause of the errors may be difficult to detect because the value of some variable can be changed by a part of the program that does not refer to that variable at all, and output based on the mistake may not occur until long after the destructive deed. Prevention is the best strategy for developing working code. It is up to the programmer to use subscripts carefully and ensure that every subscript is legal for its array. With this in mind, remember that

- Arrays start with subscript 0, so the largest legal subscript is one less than the number of elements in the array.
- An input value must be checked before it can be used as a subscript. Negative values and values equal to or larger than the array length must be eliminated.
- A counting loop that processes an array should terminate when the loop counter reaches the number of items in the array.

10.3 Parallel Arrays

The next program (specified in Figure 10.12 and given in Figure 10.13) uses a set of **parallel arrays**, all of the same length, to implement a table of data. Each array in the set represents one column of the table and each array subscript represents one row of data. In this example, the first column is a list of student ID numbers. Parallel to it are three other arrays containing data about the students. The data at subscript `k` in each of these arrays corresponds to the student with subscript `k` in the ID array. This is illustrated by the declarations and diagram in Figure 10.14.

When a table is implemented as a set of parallel arrays, the same variable is used to subscript all of them. We can apply this principle here. A loop is used to select the array slots. For each slot, first, input is read into three of the arrays at the selected position, then an average is calculated and stored in the fourth array. After the input loop, we scan this last array to compute several more values.

Notes on Figure 10.13. Using parallel arrays.

First box: limiting the subscripts.

- Serious errors result from using a subscript beyond the end of the array. To avoid this, we check that the class size is within the limits we are prepared to handle. If it is too large, we abort. We also abort if the size is negative or 0, because these values are meaningless.
- If a class really had more than 16 students, this program would need to be edited to make `MAX` larger and then recompiled, so we print an error comment and end execution gracefully.

Second box: the input phase.

- Our input loop counts from 0 up to the class size the user has entered. Since this count has been validated, we can be sure that all array subscripts are legal.
- On each iteration we enter all the data for one student. Three numbers are read and stored in the corresponding slots of the first three arrays.

```
#include <stdio.h>
#define MAX 16
int main( void )
{
    int n;              // Number of students in class.
    long id[MAX];       // Students' ID numbers.
    short midterm[MAX], final[MAX];     // Exam scores.
    float average[MAX]; // Average of exam scores.
    float avg_average;  // Average of averages.
    float diff;         // Student's average minus class average.
    printf( " Exam average = .45*midterm + .55*final.\n"
            " How many students are in the class? " );
    scanf( "%i", &n );
    if (n > MAX || n < 1) {
            printf( "Size must be between 1 and %i.", MAX );
            exit(1);
    }

    printf( " At each prompt, enter an ID# and two exam scores.\n" );
    for (int k = 0; k < n; ++k) {
        printf( "\t > " );
        scanf( "%li%hi%hi", &id[k], &midterm[k], &final[k] );
        average[k] = .45 * midterm[k] + .55 * final[k];
    }

    for (avg_average = 0, k = 0; k < n; ++k) avg_average += average[k];
    avg_average /= n;
    printf( "\nAverage of the averages = %.2f\n", avg_average );

    puts( "\nID num   mid   fin   average   +/- " );
    puts( "---------------------------------- " );

     for (int k = 0; k < n; ++k) {
         diff = average[k] - avg_average;
         printf( "%li %5hi %5hi %8.1f %6.1f \n",
                 id[k], midterm[k], final[k], average[k], diff );
     }

    puts( "----------------------------------" );

    return 0;
}
```

Figure 10.13. Using parallel arrays.

This is a diagram of the memory for the program in Figure 10.13. A set of parallel arrays is used to represent the exam scores and exam average for a class. A common subscript, `k`, is used to subscript all four arrays. The maximum number of students this table can hold is 16, but this class has only 13 students, so the last three array slots are empty.

```
#define MAX 16

int k;
int n;
long id[MAX];
short midterm[MAX];
short final[MAX];
float average[MAX];
```

MAX:16

k: 3

n: 13

	id	midterm	final	average
0	825176	80	85	82.8
1	825301	72	68	69.8
2	824769	97	90	93.2
3	826162	57	66	62.0
4	824388	88	92	90.2
5	825564	42	61	52.5
6	825923	75	62	67.8
7	823976	82	81	81.4
8	824662	91	94	92.7
9	824478	68	80	74.6
10	826056	82	71	75.9
11	826178	95	97	96.1
12	825743	51	57	54.3
13				
14				
15				

Figure 10.14. Parallel arrays can represent a table.

- Sometimes the input loop does only input and other loops are used to process the data. In this example, the loop both reads the input and calculates the weighted average, which is part of a student's record and based directly on the input. This average is stored in the fourth array (the fourth column of the table). Merging these actions leads to a slightly more efficient program. However, if the additional calculations are lengthy, efficiency can be sacrificed for the added clarity of splitting the tasks into separate loops.
- The prompts and input process look like this:

```
Exam average = .45*midterm + .55*final.
How many students are in the class? 13
At each prompt, enter an ID# and two exam scores.
        > 825176 80 85
        > 825301 72 68
        > 824769 97 90
        > 826162 57 66
        > 824388 88 92
        > 825564 42 61
        > 825923 75 62
        > 823976 82 81
        > 824662 91 94
        > 824478 68 80
        > 826056 82 71
        > 826178 95 97
        > 825743 51 57
```

- We will echo the input data later, along with calculated values.

Third box: the average calculation.

- We sum the weighted averages as the first step of computing the overall class average. This task also could have been done as part of the input loop but was written as a separate loop because it has no direct connection to the input process or a single student's record.
- We use a one-line `for` loop, because summing the values in an array is a simple job that corresponds to a single conceptual action.
- Note how convenient the `+=` operator is for summing the elements of an array. The `/=` operator provides a concise way to say "now divide the sum by the number of students."
- The output from this box is

```
Average of the averages = 76.40
```

Fourth and fifth boxes: the output phase.

- In the outer box, we print table headings before the output loop and print a line to terminate the table after the loop.
- In the inner box, we print each student's record by including one value from each of the parallel arrays and a final value computed from the average array. Note that we use `%f` in the `printf()` format to make nicely aligned columns.
- The input data is printed side by side with the final output to make it easier to check whether the computations are correct.
- The final output of the program is:

```
ID num   mid   fin   average   +/-
-----------------------------------
825176    80    85     82.8    6.3
825301    72    68     69.8   -6.6
824769    97    90     93.2   16.8
826162    57    66     62.0  -14.5
824388    88    92     90.2   13.8
825564    42    61     52.5  -24.0
825923    75    62     67.8   -8.6
823976    82    81     81.4    5.0
824662    91    94     92.7   16.2
824478    68    80     74.6   -1.8
826056    82    71     75.9   -0.5
826178    95    97     96.1   19.7
825743    51    57     54.3  -22.1
-----------------------------------
```

10.4 Array Arguments and Parameters

No data type is very useful in a programming language unless it can be used in a function call to pass information into and out of a function. Therefore, we need to know how to write a function with an array parameter and how to call such a function with an array argument. C does not permit a function to *return* an array value.[4]

Array arguments in C are handled differently from other types of arguments. When an `int` , a `double`, or a single element from an array is passed to a function, its value is copied into the parameter variable that has been created for the function. Technically, we say that arguments of simple types are passed *by value*. However, when an array is passed to a function, it is passed *by reference*, that is, only the *address* of the first slot of the array, not its entire list of values, is copied into the function's memory area. This is similar to the way that `scanf()` works. The address of a variable is passed to `scanf()`, which fills it with information from the keyboard, and this information remains in the variable even after `scanf()` finishes.

Passing array arguments by reference permits a large amount of data to be made available to a function efficiently (since the actual data are not copied) and also allows the function to store information into the array.

[4]However, it is possible to return a pointer to an array. This topic is deferred until a later chapter.

Therefore, a program can pass an empty array into a function, which then will fill it with information. When the function returns, that information still is in the original array's memory and can be used by the caller.

To call a function with an **array argument**, the caller simply writes the name of the array and does *not* write a subscript or the square subscript brackets. Also, no `&` operator is used in front of the array name, because the array name automatically is translated into its starting address. To declare the corresponding **array parameter**, however, we use empty square brackets (with no length value). The length may be written between the brackets but it will be ignored by the compiler. This is done in C so the function can be used with arrays of many different lengths.

For example, if the actual argument were an array of `double`s, a formal parameter named `ara` would be declared as `double ara[]`. Within the function, the parameter name is used with subscripts to address the corresponding argument values.

The next program illustrates the basic array operations described so far: input, output, access, calculation, and the use of an array parameter. In it, the term `FName` means function name and `AName` means array name. We introduce and demonstrate the use of three new forms of function prototypes that manipulate arrays:

- `void FName( double AName[], int n );`

  A prototype of this form is used when the purpose of the function is to read data into the array or print the array data. The `get_data()` function in the next example has this form; its prototype is

  ```
  void get_data( double x[], int n );
  ```

  This function takes two parameters, an array of `double`s called `x` and an integer that gives the length of the array. Since the declaration of the parameter `x` contains no length, we need a limiting value. This can be the globally defined constant that was used to declare the array object. Often, though, we do not use the entire array, and a parameter is used to communicate the amount of the array currently in use. The `get_data()` function fills the array with input values that remain in it after the function returns. This is one way of returning a large number of values from a function to the calling program. Since there is no other return value, the return type is declared as `void`.

- `double FName( double AName[], int n );`

  A prototype of this form is used when an array contains data and we wish to access those data to calculate some result, which then is returned. The function named `average()` in the next example has this form; its prototype is

  ```
  double average( double x[], int n );
  ```

  It again takes two parameters, the array of `double`s and the current length of the array. The function calculates the average (mean) of those values and returns it to the caller via the `return` statement. Therefore, the function return type is declared as `double`.

- `double FName( double AName[], int n, double Arg );`

  We use a prototype of this form when we need both an array of data and another data value to perform a calculation. The function named `divisible()` in the next example has this form; its prototype is

  ```
  int divisible( int primes[], int n, int candidate );
  ```

  It takes three parameters, an array of prime numbers, its length `n`, and the `candidate` number we wish to test for primality. The numbers in the array are used to test the candidate; the answer will be *true* (1) or *false* (0).

10.5 An Array Application: Prime Numbers

The next application is a prime number generator that illustrates the use of arrays and array parameters. The task specifications are given in Figure 10.15, the main program in Figure 10.16, and a function in Figure 10.17.

Notes on Figure 10.16. Calculating prime numbers.

First box: prototype.

- This prototype follows the third pattern discussed above: the parameters are an array, its length, and another value that must be used with the array.
- The `divisible()` function compares the `candidate` number to the numbers in the array. If the `candidate` is divisible by anything in the array, *true* (1) is returned. Otherwise it is non divisible (prime), so *false* (0) is returned.

Problem scope: Print a list of all prime numbers, starting with 2, and continuing until `MANY` primes have been printed. A prime number is an integer that has no proper divisors except itself and 1.

Constant: `MANY`, the number of primes to be found and printed.

Restrictions: `MANY` must be small enough that an array of `MANY` integers can fit into memory and the last prime calculated is less than the maximum integer that can be represented.

Input: None.

Output required: A neat list of primes, one per line.

Figure 10.15. Problem specifications: A table of prime numbers.

This main program calls the functions in Figure 10.17. It calculates and prints a table of the first MANY prime numbers. Strategy: identify prime numbers in ascending order and print them. Save each prime in a table and use them all to test the next number in sequence.

```
#include <stdio.h>
#define MANY 3000

void print_table( int primes[], int num_primes );
int divisible( int primes[], int n, int candidate );   // Is it nonprime?

int main( void )
{
    int k;                  // Integer being tested.
    int primes[MANY]={2};  // To begin, put the only even prime in table.
    int n = 1;             // Number of primes currently in table.

    printf( "\nA Table of the First %i Prime Numbers\n", MANY );
    for (k = 3; n <= MANY; k += 2) {       // Test the next odd integer.
                                           // Quit when table is full.
        if (!divisible( k, primes, n )) {  // If it is a prime...
            primes[n] = k;                 // ... put it in the table...
            ++n;                           // ... and count it.
        }
    }
    print_table( primes, MANY );           // Print table of primes.
    return 0;
}
```

Figure 10.16. Calculating prime numbers.

Second box: the table of primes.

- The table will be filled in with prime numbers. The list will be generated in order by testing every possible odd number, starting with 3. As primes are discovered, we store them in the table. To test each integer, we use the previously computed portion of the table.
- We choose an arbitrary constant for the length of this table. Computing more primes requires more time and storage space. This method is limited by the space available and the largest integer that can be represented in the ordinary way. The latter limit usually occurs first.
- The first prime, and the only even prime, is 2. We initialize the first slot in the table to 2 so that the computation loop can be limited to testing odd numbers. The rest of the prime table will be initialized to 0.
- We already have stored one prime in the table, so we initialize `n` to 1. It will be incremented each time a new prime is found.

Third box: filling the table.

- We use a `for` loop that starts at 3 and counts by twos to test all the odd numbers.
- This is a very unusual loop. While we initialize `k` and increment it each time around the loop, we use `n`, the number of primes, to end the loop. We want to continue searching for primes until the table is filled; that is, `n==MANY`. Since we do not know how big `k` will be at that time, we do not use `k` to terminate the loop.

Inner box: calling the function to test for primality.

- By definition, N is prime if it has no divisors except itself and 1. If N did have a divisor, it would have to have two, and one of them would have to be less than or equal to $\sqrt{N}$. Also, if N did have a divisor, D, either D would be a prime number or D itself would have at least two other divisors smaller than itself. Thus, we can show that N is a prime by showing that it is not divisible by any prime less than or equal to $\sqrt{N}$.
- If the `divisible()` function returns 1 (true), `k` is not a prime. If it returns 0 (false), we put `k` into the table and increment `n`.

Last box: printing the table. The output is printed by calling `print_table()`. The first and last portions of it are

```
A Table of the First 3000 Prime Numbers
         2
         3
         5
         7
         9
        11
        13
        15
        17
        19
    ......
      5993
      5995
      5997
      5999
 ---------------------
```

Notes on Figure 10.17. Functions for the prime number program. The `divisible()` function can identify primes up to the square of the largest prime currently stored in the table. Its parameters are `candidate` (a number to test), `primes` (a table of primes), and `n` (the current length of the table).

These functions are called from Figure 10.16.

```
// ---------------------------------------------------------------------
// Test candidate number for divisibility by primes in the table.
// Return true if a proper divisor is found, false otherwise.
int
divisible( int primes[], int n, int candidate ) {
    int last = (int) sqrt( candidate );
    int found = 0;        // Initially false, no divisor has been found.

    // Divide by every prime < square root of candidate.
    for (int m = 0; m < n && primes[m] <= last; ++m) {
        if (candidate % primes[m] == 0) {
            found = 1;    // Set to true; divisor has been found.
            break;
        }
    }

    return found;
}
// ---------------------------------------------------------------------
// Print the list of prime numbers, one per line.
void
print_table( int primes[], int num_primes )
{
    for (int m = 0; m < num_primes; m++) printf( "%10i\n", primes[m] );
    printf( " ---------------------\n" );
}
```

Figure 10.17. Functions for the prime number program.

First and third boxes: the termination condition.

- We define a variable of type `int`, whose value will be returned later as the result of the function. We initialize the variable to 0 (false) and later set it to 1 (true) if we find what we are searching for; that is, a number that evenly divides the candidate.
- We return the value of `found` after the search either succeeds or exhausts the data in the table.
- This is a common control pattern and especially useful when several tests must be made and any one of them could terminate processing.

Second box: the search loop.

- We use the modulus operator to test whether one number is divisible by another; a is divisible by b if the remainder of `a/b` is 0; that is, if `a%b == 0`.
- To test a candidate number, we divide it by all the numbers in the table up to the square root of the candidate and leave the search loop with a `break` statement the first time we find a proper divisor.
- If no divisor is found, one of two conditions will terminate the loop: Either we have tested every prime in the table or the next prime in the table is greater than the square root of the candidate.

10.6 Searching an Array

A common application of arrays is to store a table of data that will be searched, and possibly updated repeatedly, in response to user inputs. In the noncomputer world, a table has at least two columns: a column of index values and one or more columns of data. For example, in a periodic table of the elements, the atomic numbers (1...109) are used as the **index column**, then the element names, atomic weights and chemical symbols are **data columns**. If we use the same data for other purposes, a different column, such as the name, might be chosen as the index column.

A table can either be sorted or unsorted. The data in a **sorted table** are arranged in ascending or descending order, according to some comparison function defined on the values in the index column. For example, a periodic table is sorted in ascending order by the atomic number. A dictionary is sorted in ascending alphabetic order. The typical university course catalog is sorted in ascending order by department code, and within a department, by course number.

To implement a table in the computer, we can use either a set of parallel arrays or an **array of structures**[5]. When we implement a table as a set of **parallel arrays**, we use one array to represent the index column and one more for each data column in the table.

As discussed in Chapter 6, a typical **search loop** examines a set of possibilities, looking for one that matches a given key value. A sequential search of a table examines the index column for an entry that matches the key, one item after another. In every table-searching application, we find the following elements:

- A table, consisting of an index column and one or more data columns.
- A **search key**, the input value that must be compared to the entries in the index column of the table.
- A **comparison function** that is defined for the base type of the array. With simple types, such as numbers or characters, the `==` operator is appropriate. However, a programmer must define some other comparison function to search an array whose base type is an aggregate type such as those defined in Chapters 12 and 13.
- The **position variable**, the output from the search process, set to the subscript that identifies the value in the index column matching the key value.
- A **success or failure code**, sometimes a separate output value, other times failure may be indicated by setting the position variable to a value either too large or too small to be a legal subscript.

The next program example shows a search loop used for a simple application: recording bill payments in an array of account information. Figure 10.18, gives the specification, Figure 10.19 is the main program, and Figure 10.21 is the **sequential search** function implemented by a search loop.[6]

Notes on Figure 10.19. Main program for sequential search.

First box: prototypes for this application. The main program will call these three functions to do all the work. The first two are more or less the same in every array application that works on parallel arrays: an input function that fills the arrays and an output function that prints them.

Second box: modeling a table. We use a parallel-array data structure to implement a table. It has an index column (`ID`) and one data column (`owes`). The maximum capacity of the table is defined at the top of the program, (20 in this case) and the actual number of rows in the table will be determined at run time and stored in `n`.

Third box: variables for the search. A sequential search function tries to locate a key value in an array. If it is located, `where` will be used to store its position.

Fourth box and last box: input, echo, and final output. We call functions to read the input. In a realistic program, the data would be input from a file rather than from the keyboard. With only minor changes, the code in `get_data` can be changed to read the data from a file.

The output function is called twice: once to echo the input and again to print the results. When the output code is written in a function, it is easy to use it more than once. This can be especially useful during debugging, when one might wish to see the data in the array within the loop after every change.

[5]Structures are explained in Chapter 13. If a table is modeled as an array of structures, the structure has one member for each column in the table. The array of structures usually is considered a better style because it is more coherent; that is, it groups together all the values for a table entry. The relative merits of these two approaches are discussed in Chapter 13.

[6]Discussion of the binary search algorithm, which is more complex but much faster for sorted arrays, is deferred until recursion is introduced in Chapter 19.

Problem scope: Starting with a list of payments that are due, record payment amounts and print a list of account balances after recording the payments.

Inputs: Input will happen in two phases.
Phase 1. For each account, enter the account ID number and the initial amount due.
Phase 2. The payments will be entered. For each one, the ID number is entered first. If it is found in the list of accounts, the user will be prompted for a payment amount.

Constants: `ACCOUNTS` must be defined as the maximum number of accounts that the company has simultaneously. The actual number of accounts can be smaller than this limit.

Output: For each account number entered in input phase 2, the position of that account in the list will be displayed. At the end of Phase 2, a list of final balances will be displayed. If the bill was overpaid, this balance will be negative.

Formulas: Each payment amount should be subtracted from the initial balance in the account.

Limitations: No attempt is made to validate payment amounts. ID numbers that are not in the list of accounts will cause an error comment but otherwise have no effect.

Figure 10.18. Problem specifications: Recording bill payments.

Sample output up to this stage. The first block of output came from `get_data()`, the second block from `print_data()`.

```
Enter pairs of ID # and unpaid bill (two zeros to end):
31         2.35
7          3.19
6          2.28
13         1.09
22         8.83
38         13.25
19         5.44
32         6.90
25         1.70
3          41.
0          0

Initial List of Unpaid Bills:
     ID  Amount
[ 0] 31    2.35
[ 1]  7    3.19
[ 2]  6    2.28
[ 3] 13    1.09
[ 4] 22    8.83
[ 5] 38   13.25
[ 6] 19    5.44
[ 7] 32    6.90
[ 8] 25    1.70
[ 9]  3   41.00

10 items were read; ready for payments.
```

Fifth box: payments and account balances. We have read an ID number and are ready to process a payment from that person. First, we must find the position of the person in the table; the call on `sequential_search()` does this, and stores the answer in `where`.

If the ID is not found in the table, `where` will have a negative value. Otherwise, its value will be between 0 and `n-1`. We check this condition, and go on to input and process a payment if the search was successful. Because this is a parallel-array data structure, the payment amount is subtracted from the bill at position `where` in the `owes` array which corresponds to the person at position `where` in the `ID` array.

Sample output from this phase.

```
Enter ID number (zero to end): 31
ID 31   found in slot 0.  Amount paid: 2.35
Enter ID number (zero to end): 25
ID 25   found in slot 9.  Amount paid: 2
```

```
#include <stdio.h>
#define ACCOUNTS 20

int  get_data( int ID[], float owes[], int nmax );
void print_data( int ID[], float owes[], int n );
int  sequential_search ( int ID[], int n, int key );

int main( void )
{
    int n;                  // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];     // ID's of members with overdue bills.
    float owes[ ACCOUNTS ]; // Amount due for each member.

    int key;                // ID number to search for.
    int where;              // Position in which key was found.

    float payment;           // Input: amount of payment for one ID#.

    n = get_data( ID, owes, ACCOUNTS );   // Input all unpaid bills.
    printf( "\nInitial List of Unpaid Bills:\n     ID  Amount\n" );
    print_data( ID, owes, n );            // Echo the input

    printf( "\n%i items were read; ready for payments.\n\n", n );
    for (;;) {
        printf( "Enter ID number (zero to end): " );
        scanf( "%i", &key );
        if (key==0) break;

        where = sequential_search( ID, n, key );
        if (where<0) printf( " Item %i \t not found.", key);
        else {
            printf( "\tID %2i \t found in slot %i. ", key, where );
            printf( " Amount paid: " );
            scanf( "%g", &payment );
            owes[where] -= payment;
        }
    }
    printf( "\n\nFinal Amounts Due:\n     ID  Amount\n" );
    print_data( ID, owes, n );      // Print final amounts owed.
    return 0;
}
```

Figure 10.19. Main program for sequential search.

```
// ---------------------------------------------------------------------
int                                     // Actual number of data sets read.
get_data( int ID[], float owes[], int nmax )
{
    int k;                              // Loop counter and array index

    printf( "Enter pairs of ID # and unpaid bill (two zeros to end):\n" );
    for (k=0; k<nmax; ++k) {            // Don't go beyond end of arrays.
        scanf( "%i%g", &ID[k], &owes[k] );
        if( ID[k]==0 ) break;           // No more data is available.
    }
    return k;
}
// ---------------------------------------------------------------------
void print_data( int ID[], float owes[], int n )
{
    for (int k=0; k<n; ++k) {       // Don't read beyond end of ID array.
        printf( "[%2i] %2i %7.2f{\bk}n", k, ID[k], owes[k] );
    }
}
```

Figure 10.20. Input and output functions for parallel arrays.

```
Enter ID number (zero to end): 13
ID 13   found in slot 3.  Amount paid: 1
Enter ID number (zero to end): 38
ID 38   found in slot 5.  Amount paid: 10
Enter ID number (zero to end): 0

Final Amounts Due:
     ID  Amount
[ 0] 31    0.00
[ 1]  7    3.19
[ 2]  6    2.28
[ 3] 13    0.09
[ 4] 22    8.83
[ 5] 38    3.25
[ 6] 19    5.44
[ 7] 32    6.90
[ 8]  3   41.00
[ 9] 25   -0.30
```

Termination. Both the `get_data()` function and the payment-processing loop terminate when a sentinel value (an ID number of 0) is entered.

Notes on Figure 10.20. Input and output functions for parallel arrays.

Sequential array processing. These two functions and the one in Figure 10.21 follow a common pattern used for processing an array sequentially. Each function uses a `for` loop to perform an operation (I/O or calculation) on every array element, starting with the first and ending with the last. A programmer can use arrays for a wide variety of applications by following this pattern and varying the operation.

The `get_data()` ***function.***

- We use variable `k` with a `for` loop to process the array. We must declare `k` before the loop (not inside the for's parentheses) because the value of `k` is used after the end of the loop.

A simple sequential search function for an unsorted table.

```
int sequential_search( int ID[], int n, int key )
{
    int cursor;      // Loop counter and array index

    for (cursor = 0; cursor < n; ++cursor) {
        if ( ID[cursor] == key ) return cursor;
    }

    return -1;
}
```

Figure 10.21. Sequential search of a table.

- Before entering the loop, we prompt the user to enter a series of data values. Within the loop, we use a very short prompt to ask the user individually for each value. This is a clear and convenient interactive user interface. During execution of `get_data()`, the user will see something like this:

```
Please enter data values when prompted.
x[0] = 77.89
x[1] = 76.55
x[2] = 76.32
x[3] = 79.43
x[4] = 75.12
x[5] = 64.78
x[6] = 79.06
x[7] = 78.58
x[8] = 75.49
x[9] = 74.78
```

- If we were reading data from a file, an interactive prompt would not be necessary.
- The variable `k` is used as the counter for the loop and also to subscript the array (the usual paradigm for processing arrays). We initialize `k` to 0 and leave the loop when `k` exceeds the last valid subscript, based on the current number of array slots that are in use.
- We also leave the loop if the user enters the sentinel signal: a zero ID number. Because we are reading the ID and the amount owed with the same `scanf()`, a second number must be entered after the sentinel value to "satisfy" the format. Our instructions say to enter two zeros, but the code does not test the second number.
- On each repetition of the loop, we read one data value directly into `&x[k]`, the `k`th slot of the array `x`. At the end of each repetition, we increment `k` to prepare for processing the next array element. Each time around the loop the variable `k` contains a different value between 0 and `n-1`; after `n` iterations, data fill the first `n` array slots and the remaining slots still contain garbage.
- After the sentinel value has been read, we exit from the loop. At this time, the value of `k` is the number of real data items that have been read and stored, excluding the sentinel. We return this essential information to the caller, which will store it and use it to control all future processing on these arrays.
- When control returns to the caller, it can use the values stored in the array by the function.

Notes on Figures 10.21 and 10.22. Sequential search. This function assumes that the data are unsorted and that all items must be checked before we can conclude that the search has failed. Two versions are given, a simpler one with two return statements and a longer one with a status flag.

The function header. The parameters include elements required for a search algorithm: a table, the number of items to be searched, and a search key. The table is a simple integer array containing `n` values. The return value will be a failure code or the subscript of the key value in the array if it exists.

This is an alternative way to code a search function that uses only one return statement. To accomplish this goal, we use a status flag.

```
int sequential_search( int ID[], int n, int key )
{
    int cursor;      // Loop counter and array index
    int found = 0;   // False now, becomes true if key is found.

    for (cursor = 0; !found && cursor < n; ++cursor) {
        if ( ID[cursor] == key ) found = 1;   // true ;
    }

    return (found ? cursor-1 : -1);
}
```

Figure 10.22. Searching without a second return statement.

In the bill-payment application we use a pair of parallel arrays containing a list of account numbers and the amount owed on each account. Most actions taken by the program (input, output) involve both arrays. However, the arrays are treated differently when we search. Only the index column, in this case the ID array, is passed to the search function.

Style. Some experts believe that programmers should never use two return statements in a function. This slightly longer version of the search loop does the same job by using a status flag in place of the second return statement.

First box: the status flag. To avoid using either a `break` statement or a second `return` statement, we introduce a status flag named `found`. This is initialized to false (0) and will be set to true (1) within the search loop if the key item is found.

Second box: the search loop. A counted loop is used to examine the data items. If a match is found for the key, the loop terminates; otherwise, all `n` values are checked.

First inner box: the comparison. Since the base type of the array is `int`, we use the `==` operator to compare each item to the key value. The search succeeds if the result is true (1).

Second inner box: Dealing with Success. If a match is found, we need to leave the loop. This is done in different ways by our two versions of this function.

- Sometimes we abort a loop with a `break` statement. Here, we use `return` for the same purpose. It causes control to leave both the loop and the function. The current item's subscript is returned to the caller.
- We avoid using a second return statement tests by setting a status flag to true (1), indicating that the key value has been found. Control does not leave the loop. It returns to the top of the loop and increments `cursor`, making it one too large. The function return statement will have to compensate for this extra increment.

Last box, Figure 10.21: Failure. If the loop goes past the last data item without finding a match, the search has failed. Failure is indicated by returning a subscript of -1, a subscript that is invalid for any array, and if often used to indicate error or failure.

Last box, Figure 10.22: Returning. This single return statement must handle both success and failure. If the item was not found, we want to return -1 to indicate failure. If it was found, we must return the value of `cursor`-1. The value of `cursor` is too largeby 1 because the `for` loop incremented it after `found` was set to true and before `found` was tested to terminate the loop.

The little-used conditional operator provides a way to use a single return statement to return one thing or another. It works like an `if...else` except that it is an expression, not a statement. Read the statement like this: "If `found` is true, then return `cursor-1` else return -1".

```
#include <stdio.h>
#define ACCOUNTS 20

int  get_data( int ID[], float owes[], int nmax );
int  find_max( int ID[], int n );

int main( void )
{
    int n;                  // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];     // ID's of members with overdue bills.
    float owes[ ACCOUNTS ]; // Amount due for each member.
    int where;              // Position of maximum value.

    n = get_data( ID, owes, ACCOUNTS );   // Input all unpaid bills.
    printf( "\nLargest Unpaid Account:\n" );

    where = find_max( owes, n );
    printf( "\tID# %i owes $ %.2f\n", ID[where], owes[where] );

    return 0;
}
```

Figure 10.23. Who owes the most? Main program for finding the maximum.

10.7 The Maximum Value in an Array

Finding the maximum value in an array is an easy but nontrivial task. The `find_max()` function presented here scans the data array sequentially, like a search function, but it does not use a search key. To illustrate this algorithm, we use the same parallel-array data structure that was used in Figure 10.19, and search the data for the largest unpaid bill.

Notes on Figure 10.23. Who owes the most?

First box: Prototypes. The `get_data()` function is in Figure 10.21 and `find_max` is in Figure 10.24.

Second box: data declarations. We are using the same data structure as in the sequential search application: a set of parallel arrays containing the ID numbers and unpaid balances of a set of up to `ACCOUNTS` customers. We need `n` to store the actual number of customers that were input and `where` to store the position of the customer with the largest bill.

```
int                                 // Find the maximum value in an array.
find_max( float data[], int n )
{
    int finger = 0;                 // Put your finger on the first value.
    int cursor;

    for (cursor = 1; cursor < n; ++cursor) {
        if (data[finger] < data[cursor])   // If you find a bigger value...
            finger = cursor;               // ...move your finger to it.
    }

    return finger;                  // Your finger is on the biggest value.
}
```

Figure 10.24. Finding the maximum value.

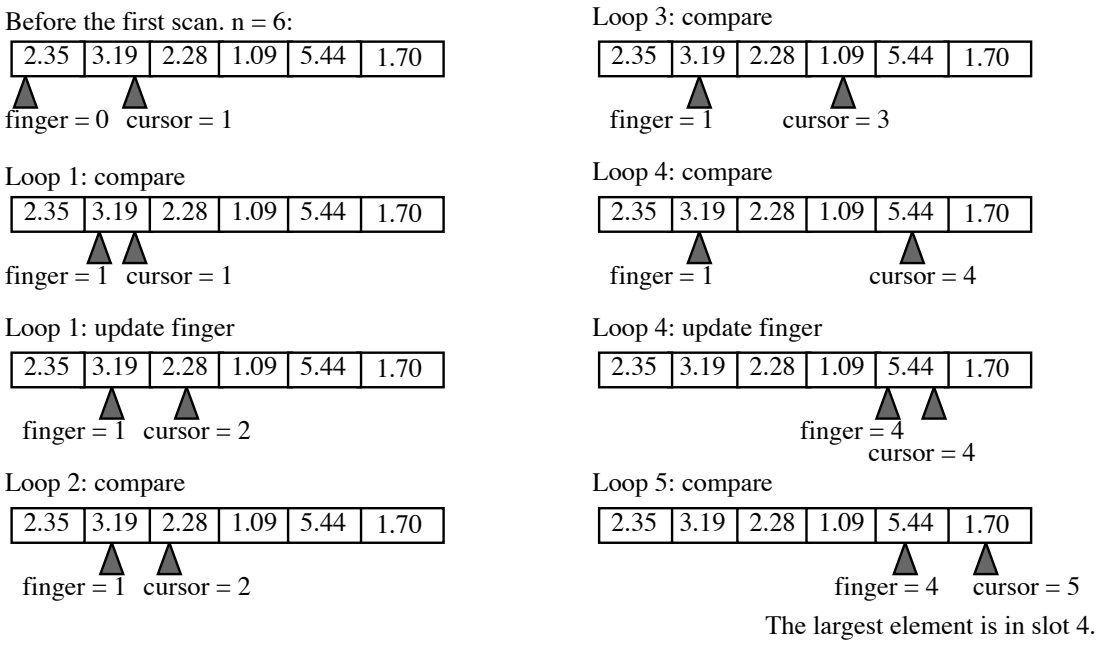


Figure 10.25. The maximum algorithm, step by step.

Third box: doing the work. We call the `find_max()` function in Figure 10.24 to search for the largest value in the array `owes` which contains `n` data items. The result is stored in `where`, and is then used to print out both the biggest bill and the ID number of the customer who owes the most.

Notes on Figure 10.24. Finding the maximum value.

The function header. The `find_max()` function is called from `main()` in Figure 10.23. We only need two parameters: the array to search and the length of that array.

First box: initialization. The idea is to scan the array sequentially, but at all times, to keep one "finger" on the biggest value we have seen so far. To get started, we set `finger` to slot 0.

Second box: the search loop. To find the largest value in an array, we must examine every array element. We start by designating the first value as the biggest-so-far, then scan start a sequential scan from array slot 1, looking for something bigger. Whenever we find the value under the `cursor` is bigger than the one under the `finger`, we update `finger`. When the loop ends, `finger` will be the position of the largest value. This process is illustrated in Figure 10.25.

10.8 Sorting by Selection

A common operation performed on arrays is sorting the data they contain. Many sorting methods have been invented: Some are simple, some complex, some efficient, some miserably inefficient. In general, the more complex sorting algorithms are the most efficient, especially if the array is very long.

In this section, we look at **selection sort**, which can be used on a small number of items. It is one of the simplest sorting methods, but also one of the slowest. Nonetheless, selection sort has the advantage that, if you stop in the middle of the process, one part of the array is fully sorted, so it is a reasonable way to find the either the largest or smallest few items in a long array.[7]

[7] A simple algorithm, **insertion sort**, has been shown to be the fastest sort of all when used on short arrays (fewer than

The basic selection strategy has several variations: the data can be sorted in ascending or descending order, the work can be done by using either a maximum or a minimum function, and the sorted elements can be collected at either the beginning or the end of the array. In this section, we develop a version that sorts the array elements in ascending order, by using a maximum function and collecting the sorted values at the end of the array. At any time, the array consists of an unsorted portion on the left (initially the whole array) and a sorted portion on the right (initially empty). To sort the array, we make repeated trips through smaller and smaller portions of it. On each trip, we locate the largest remaining value in the unsorted part of the array,

10 items). We present this algorithm in Chapter 17. The **quicksort** (discussed in Chapter 19) is a much better way to sort moderate length and long arrays. Two other simple sorts, bubble sort and exchange sort, have truly bad performance for all applications. They are not presented here and should not be used.

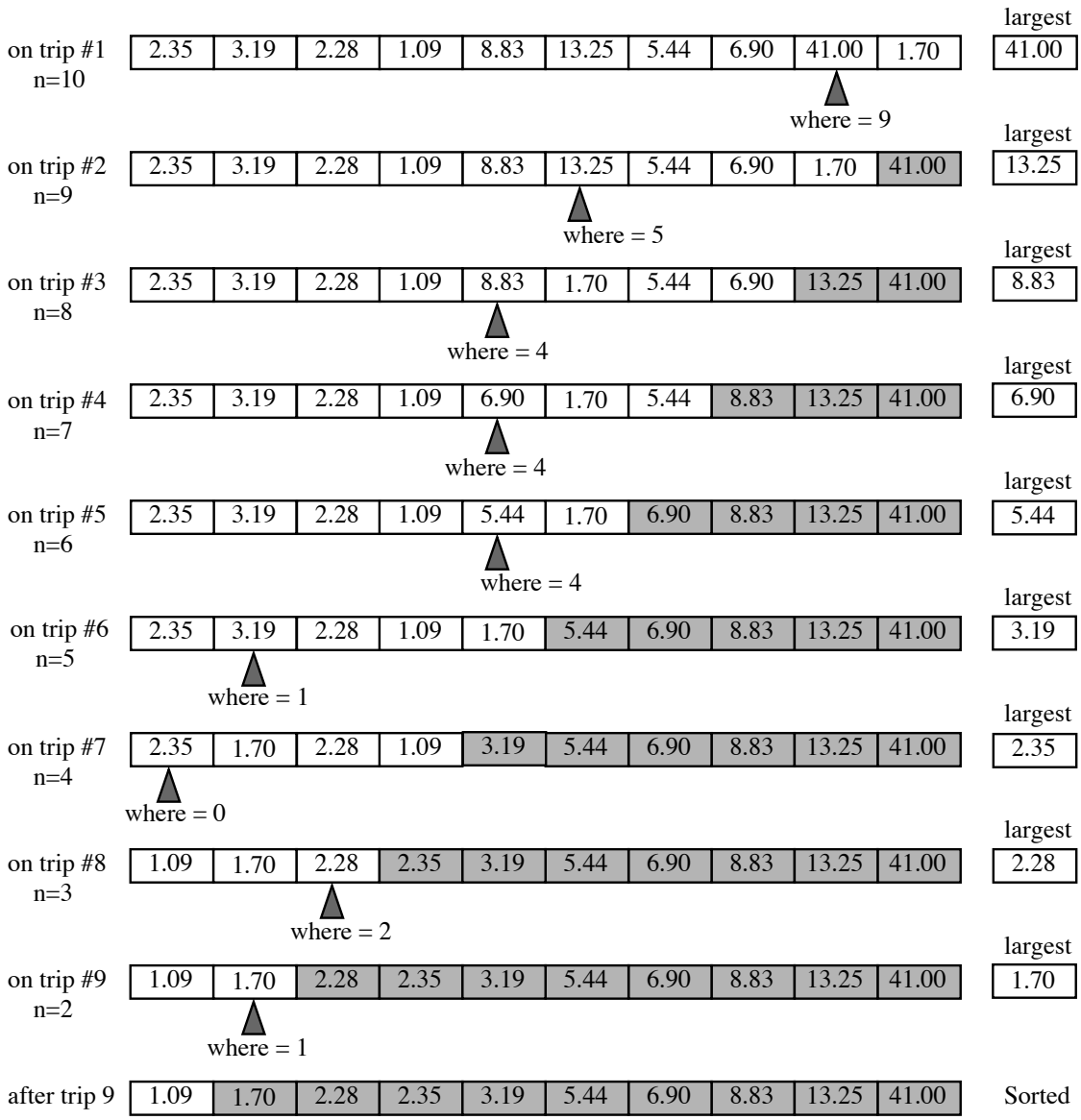


Figure 10.26. The selection sort algorithm, step by step.

Goal: Sort customer billing records in ascending order according to the amount owed. These records are stored in a parallel-array data structure with two columns: ID number and amount owed. There are `n` records altogether.

Input: ID numbers and amounts due for a set of customers.

Output: A list of customer records, in order, by the amount owed.

Algorithm: Use a selection sort, as follows:
Repeat the following actions $n - 1$ times:

1. Let `where` be the subscript of the maximum-valued element in the array between subscripts 0 and `n-1`.
2. Swap the element at position `where` with the element at position `n-1`. The swapped value will now be in its proper sorted position.
3. Decrement `n` to indicate that there are now fewer unsorted items.

Figure 10.27. Problem specifications: Sorting the Billing Records

then move it to the beginning of the sorted area by swapping it with whatever value happens to be there. After k trips, the k largest items have been selected and placed, in order, at the end of the array. After $n - 1$ trips, the array is sorted. This process is illustrated in Figure 10.26.

A program to implement this algorithm is developed easily by a top-down analysis. A problem specification is given in Figure 10.27.

10.8.1 The Main Program

A well-designed main program is like an outline of the process; it calls on a series of functions to do each phase of the actual job. This kind of design is easy to plan, easy to read, and easy to debug. The sorting task has three major phases: input (read the numbers), processing (sort them), and output (print the sorted list). For each phase, the main program should call a function to do the job and display a comment that reports the progress. Programs that contain arrays and loops often take a while to debug; during that time, generous feedback helps the programmer identify the location and nature of the errors.

We start by writing the obvious parts of `main()`, borrowing elements from the previous programs (sequential search and finding the maximum), where possible. Much can be borrowed, including

- Prototypes for the I/O functions that work with the parallel arrays (`get_data()` and `print_data()`,
- The skeleton of `main()`,
- Declarations within `main()` for the parallel-array data structure.
- Since the selection sort algorithm must find the maximum value in an array, we also include the prototype for `find_max()`.

Step 1: The obvious necessities.

```
#include <stdio.h>
#define ACCOUNTS 20

int  get_data( int ID[], float owes[], int nmax );
print_data( ID, owes, n );      // Print final amounts owed.
find_max( int ID[], int n );    // Located largest key value in array.

int main( void )
{
```

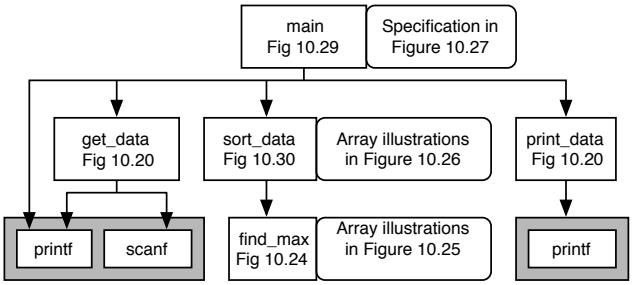


Figure 10.28. Call chart for selection sort.

```
    int n;                    // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];       // ID's of members with overdue bills.
    float owes[ ACCOUNTS ];   // Amount due for each member.
    ...
    return 0;
}
```

Step 2: Input. At the position of the dots, we add calls on the input and output functions, following the example of Figure 10.19.

```
n = get_data( ID, owes, ACCOUNTS );   // Input all unpaid bills.
printf( "\nInitial List of Unpaid Bills:\n     ID  Amount\n" );
print_data( ID, owes, n );            // Echo the input
```

Step 3: Processing. We invent the name `sort_data()` for the selection sort function. In general, the function needs to know what array to sort and the length of that array. It rearranges the data within the array and returns the sorted values in the same array, with no need for any additional return value. In this case, we are sorting a pair of parallel arrays, so both arrays must be rearranged in parallel. Thus, the `sort_data()` function must take both arrays as parameters. We write a prototype (at the top) and a call for this function inside `main()`:

```
void sort_data( int data[], float key[], int n );
...
sort_data( ID, owes, n );
```

Output. When sorting is finished, we need to output the sorted data. So we add another call on `print_data()` at the end of `main()`:

```
print_data( ID, owes, n );            // The sorted data
```

A call chart for the overall program is given in Figure 10.28. The completed `main()` function is shown in Figure 10.29.

10.8.2 Developing the `sort_data()` Function

We must start with a thorough understanding of the algorithm. This is illustrated in Figures 10.25 and 10.26 and defined carefully in Figure 10.27. Once the method is understood, we are ready to implement the `sort_data()` function. (The code fragments that follow are assembled in Figure 10.30.) We begin with the function skeleton and declare the variables mentioned in the problem specifications.

```
void sort_data( int data[], float key[], int n );
{
    int where; ...
}
```

This program calls the sort function in Figure 10.30, and the input and output functions from Figure 10.21.

```
#include <stdio.h>
#define ACCOUNTS 20

int get_data( int ID[], float owes[], int nmax );
void print_data( int ID[], float owes[], int n );
void sort_data( int data[], float key[], int n );

int main( void )
{
    int n;                  // #of ID items; will be <=ACCOUNTS.
    int ID[ ACCOUNTS ];     // ID's of members with overdue bills.
    float owes[ ACCOUNTS ]; // Amount due for each member.

    n = get_data( ID, owes, ACCOUNTS );   // Input all unpaid bills.
    printf( "%i items were read; beginning to sort.\n", n );
    sort_data( ID, owes, n );

    puts( "\nData sorted, ready for output" );
    print_data( ID, owes, n );
    return 0;
}
```

Figure 10.29. Main program for selection sort.

The loop skeleton and body. Step 2 of the specification calls for a process to be repeated $n-1$ times to sort n things. We write a `for` loop that implements this control pattern:

```
for (start=n-1; start>0; --start) {
    ...
}
```

Now we need to write the body of the loop. We have a function that finds the maximum value in an array, so we call it and save the result.

```
where = find_max( key, n );
```

Next, we must swap the large value at position `where` with the last unsorted value in the array, which is at position `start`. Swapping takes three assignments, but since we are working on a parallel-array data structure, identical swaps must be made on both arrays, for a total of six assignments. In the code below, the instructions on the left swap the key array, those on the right swap the data array.

Also, on each repetition, we start at the high-subscript end of the unsorted array, that is, at slot $n-1$. On each repetition, one item is placed in its final position, leaving one fewer item to be sorted. So we decrement n just before the end of the loop.

```
bigKey = key[where];        bigData = data[where];
key[where] = key[start];    data[where] = data[start];
key[start] = bigKey;        data[start] = bigData;
--n;
```

This completes the algorithm and the program. We gather all the parts of `sort_data()` together in Figure 10.30.

Testing. We combined all of the pieces of the sort program, compiled it, and ran it on the file `sele.in`, which contains the data listed in Figure10.31. The input is on the left and the corresponding output on the right.

This function is called from `main()` in Figure 10.29; it calls `find_max()` in Figure 10.24.

```
int find_max( float data[], int n );   // Prototype not included by main().

void sort_data( int data[], float key[], int n )
{
    int start;        // End of unsorted data in the array.
    int where;        // Position of largest value in the key array.
    int bigData;      // For swapping the data column.
    float bigValue;   // For swapping the key column.

    for (start=n-1; start>0; --start) {
        where = find_max( key, n );
        // Swap the two columns of the table in parallel.
        bigValue = key[where];      bigData = data[where];
        key[where] = key[start];    data[where] = data[start];
        key[start] = bigValue;      data[start] = bigData;
        --n;
    }
}
```

Figure 10.30. Sorting by selecting the maximum.

10.9 What You Should Remember

10.9.1 Major Concepts

Arrays and their use. An array in C is a collection of variables stored in order, in consecutive locations in the computer's memory. The array element with the smallest subscript (0) is stored in the location with the lowest address. We use arrays to store large collections of data of the same type. This is essential in three situations:

- When the individual data items must be used in a random order, as with the items in a list or data in a table.
- When each data value represents one part of a compound data object, such as a vector, that will be used repeatedly in calculations.

Input Phase		Output Phase	
Enter pairs of ID and unpaid bill (two zeros to end):		Data sorted, ready for output	
31	2.35	[0] 13	1.09
7	3.19	[1] 25	1.70
6	2.28	[2] 6	2.28
13	1.09	[3] 31	2.35
22	8.83	[4] 7	3.19
38	13.25	[5] 19	5.44
19	5.44	[6] 32	6.90
32	6.90	[7] 22	8.83
3	41.	[8] 38	13.25
25	1.70	[9] 3	41.00
0	0		

Figure 10.31. Input and output for selection sort.

- When the data must be processed in separate phases, as in the problem from Figure 10.18. In this program, all the account balances must first be read and stored. Then the stored data must be searched and updated, using bill-payment amounts.

Parallel arrays. A multicolumn table can be represented as a set of parallel arrays, one array per column, all having the same length and accessed using the same subscript variable. Multidimensional arrays also exist and are discussed in Chapter 18.

Array arguments and parameters. An array name followed by a subscript in square brackets denotes one array element whose type is the base type of the array. This element can be used as an argument to a function that has a parameter of the base type. To pass an entire array as an argument, write just the array name with no subscript brackets. The corresponding formal parameter is declared with empty square brackets. When the function call is executed, the address of the beginning of the array will be passed to the function. This gives the function full access to the array; it can use the data in it or store new data there.

Array initializers. C allows great flexibility in writing array initializers; we summarize the rules here:

- An array initializer is a series of constant expressions enclosed in curly brackets. These expressions can involve operators, but they must not depend on input or run-time values of variables. The compiler must be able to evaluate the expressions at compile time.
- If there are too many initial values for the declared length, C will give a compile-time error comment.
- If there are too few initial values, the uninitialized areas will be filled with 0 values of the proper type: an integer 0, floating value 0.0, or pointer `NULL`.
- The length of an array may be omitted from the declaration if an initializer is given. In this case, the items in the initializer will be counted and the count will be used as the length of the array.

Searching. Many methods can be used to search an array for a particular item or one with certain characteristics. A sequential search starts at the beginning of a table and compares a key value, in turn, to every element in the key column. The search ends when the key item is found or after each item has been examined.

The typical control structure for implementing a sequential search is a `for` loop that moves a subscript from the beginning of the array to the end. In the body of this loop is an `if...break` statement that compares the key value to the current table element.

The search can either succeed (find the key value) and break out of the loop or fail (because the key value does not match any item in the table). A sequential search for a specific item is slow and appropriate for only short tables. It is slightly more efficient when the table is in sorted order, because failure can be detected prior to reaching the end of the table. However, binary search (see Chapter 19) is an even faster algorithm for use with sorted data.

If the data are sorted according to the search criterion, shortcuts may be possible. However, a sequential search is necessary when the order of the data in the array is unrelated to the criterion because all the data items must be examined. For example, finding the longest word in a dictionary would require looking at every word (a sequential search) because a dictionary is sorted alphabetically, not by word length.

Sorting. Locating a particular item in a table can be done much more efficiently if the information is sorted. Many sort algorithms have been devised and studied; among the simplest (and slowest) is the selection sort. It sorts n items by selecting the minimum remaining element $n-1$ times and moving it to a part of the array that will not be searched again. Other more efficient techniques such as the insertion sort (Chapter 17) and the quicksort (Chapter 19) are examined later.

10.9.2 Programming Style

Usage.

- Use a defined constant to declare the length of an array. This way the use and the array declarations will be consistent and easily changed if the need arises.
- A `for` loop typically is used to process the elements of an array. The values of the loop counter go from 0 to the length of the array (which is given by a defined constant or a function parameter); for example, `for (k=0; k<N; ++k) printf( "%g ", volume[k] );`. Note that this loop paradigm stops before attempting to process the nonexistent element `volume[N]`.

Names. Variable names such as `j` and `k` typically are used as array subscripts since they are commonly found in mathematical formulas. However, when writing a program that sorts, it is very helpful to use meaningful names for the subscript variables. You are much more likely to write the code correctly in the first place, and then get it debugged, if you use names like `cursor` and `finger` rather than single-letter variable names such as `i` and `j`.

Local vs. global. Constants should be declared globally if they are used by more than one function or if they are purely arbitrary and likely to be changed. If a constant is used by only one function, it may be better to declare it locally. However, a large set or table of such constants will incur large setup times each time the function is called. These constants should be declared as `static const` values, which are only initialized once.

Don't talk to strangers. Each object name used in a function should represent an object that fits into one of the following categories:

- A global constant, `#define`d at the top of the program, or like `NULL`, in a header file.
- A parameter, declared and named in the function header.
- A local variable or constant, declared and named within the function (an object should be local if it is used only within a function and does not carry information from one function to another).
- A global function whose prototype is at the top of the program or in a header file.
- A local function, declared within the function and defined after it (a function should be declared locally if it is used only within that function. This does not cause any run-time inefficiency).

Modularity. We wrote a `sort_data()` function as a loop that calls the `find_min()` function. Within that function is another loop. When written like this, the logic of the program is completely transparent and easily understood. In many texts, this algorithm is written as a loop within a loop. This second form takes fewer lines of code and executes more efficiently, because no time is spent calling functions. However, it is not so easy to understand. Which form is better? The modular form. Why? Because it can be debugged more easily and is less likely to have persistent bugs. Doesn't efficiency matter? It often does, but if so, a better algorithm (such as quicksort) should be used instead. It is a false economy to use bad programming style to optimize a slow algorithm.

Sorted vs. unsorted. If the data we wish to search already are sorted, by all means we should take advantage of this. If not, we need to decide whether to sort the data before searching. This issue will be addressed to some extent in later chapters. However, it is a complex issue involving the data set size, how fast the data set changes, which data structures and algorithms are used, and how many times a search will be performed. The general topic of data organization and retrieval is the subject of dozens of books on data structures and databases.

Software reuse. Do not waste time trying to reinvent the wheel. If a library routine meets your need, use it. If you have previously written a function that does almost what you need, modify it as necessary. If someone else has developed a solution for a certain task, such as sorting, go ahead and use it, after you have verified that any assumptions it makes are satisfied by your data and structures.

10.9.3 Sticky Points and Common Errors

Array length vs. highest subscript. The number given in an array declaration is the actual number of slots in the array. Since array subscripts start at 0, the highest valid subscript is one less than the declared length. Often, though, there are more slots than valid data. This happens during an input operation and whenever the total amount of data entered falls short of the maximum allowed. In such situations, another variable is used to store the number of actual data elements in the array.

Subscript errors. Programmers accustomed to other languages often are surprised to learn that C does absolutely no subscript range checking. If a subscript outside the defined range is used, there will be no error comment from the compiler or at run-time. The program will run and simply access a memory location that belongs to some other variable. For example, if we write a loop to print the values in an array and it loops too many times, the program starts printing the values adjacent to the array in memory. At best, this results in minor errors in the results; at worst, the program can crash.

Caution: do not fall off the end of an array. Remember that C does not help you confine your processing to the array slots that you defined. When you use arrays, avoid any possibility of using an invalid subscript. Input values must be checked before using them as subscripts. Loops that process arrays must terminate when the loop counter reaches the number of items in the array.

Ampersand errors. Arrays and nonarrays are treated differently in C. An array argument always is passed to a function by address. We do not need to use an ampersand with an unsubscripted array name.

Array parameters. To pass an entire array as an argument, write just the name of the array, with no ampersands or subscript brackets. The ampersand operator is not necessary for an array argument because the array name automatically is translated into an address. The corresponding array parameter is declared with the same base type and empty square brackets. A number can be placed between the brackets, but it will be ignored.

Array elements as parameters. A single array element also can be passed as a parameter. To do this, write the array name with square brackets and a subscript. If the function is expected to store information in the array slot, as `scanf()` might, you must also use an ampersand in front of the name.

Sorting. Writing a sorting algorithm can be a little tricky. It is quite common to write loops that execute one too many or one too few times. When debugging a sort, be sure to examine the output closely. Check that the items at the beginning and end of the original data file are in the sorted file and that the output has the correct number of items. Examine the items carefully and make sure all are there and in order. It is common to make an error involving the first or last item. Test all programs on small data sets that can be thoroughly checked by hand.

Parallel arrays. A table can be implemented as a set of parallel arrays. When sorting such a table, it is important to keep the arrays all synchronized. If the items in one column are swapped, be sure to swap the corresponding items in all other columns. Using an array of structures may solve this problem, but this solution may have its own drawbacks, which we have discussed previously.

10.9.4 Where to Find More Information

- Arrays of strings are presented in Chapter 12, arrays of structures in Chapter 13 and arrays of functions in Chapter 17.
- Dynamic allocation of arrays and arrays of pointers are found in Chapter 16.
- Pointers are introduced in Chapter 11 and the use of pointers to process arrays is explained in Chapter 17.
- Two dimensional arrays and their applications are covered in Chapter 18. Multidimensional arrays and arrays of pointers to arrays are in the same chapter.
- Other sorting algorithms are presented in later chapters. Insertion sort is in Chapter 17; quicksort in Chapter 19.
- Other array algorithms presented are: Binary search: Chapter 19, Shuffling a deck: Chapter 14, Gaussian elimination: Chapter 18, Simulation: Chapter 16.

10.9.5 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

aggregate type	constant expression	status flag
array	parallel arrays	sequential search
base type	effective address	search loop
array slot	walking on memory	key column
array element	memory error	data column
array length	array argument	search key
size of an array	array parameter	sorted table
subscript	sequential array processing	position variable
array declaration	prime number	finding the minimum
array initializer	divisibility	selection sort

10.10 Exercises

10.10.1 Self-Test Exercises

1. Which occupies more memory space, an array of 15 `short int`s or an array of 3 `double`s? Explain your answer.
2. An array will be used to store temperature readings at four-hour intervals for one day. It is declared thus: `float temps[6];`

   (a) Draw an object diagram of this array.
   (b) What is its base type? Its length? Its size?
   (c) Write a loop that will read data from the keyboard into this array.
   (d) Write an `if` statement that will print `freezing` if the temperature in the last slot is less than or equal to 32°F and `above freezing` otherwise.

3. Array and function declarations.

   (a) Write a declaration with an initializer for the array of `float`s pictured here.

   ff

1.9	2.5	-3.1	17.2	0
[0]	[1]	[2]	[3]	[4]

   (b) Write a complete function that takes this array as a parameter, looks at each array element, and returns the number of elements greater than 0.
   (c) Write a prototype for this function.
   (d) Write a `scanf()` statement to enter a value into the last slot of the array.

4. In the indicated spots below, write a prototype, function header, and call for a function named `CHKBAL` that computes and returns the balance in a checking account. Its parameters are an initial account balance and an array of check amounts. You need not actually write a whole function to compute the new account balance; just fill in the indicated information.

```
#define  X  5
// insert prototype here

int main( void )
{
    float check_amounts[X];   // $ amounts of checks
    float start_balance;      // balance before checks
    float end_balance;        // balance after checks
    // put call here
}
// put function header here
```

5. The following are two declarations and a `while` loop.

```
int ara[13] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096};
int k = 12;
while (k > 0) {
    printf( "%2i: %i \n", k, ara[k] );
    k -= 2;
}
```

   (a) What is the output?

(b) Rewrite the code using a `for` loop instead of the `while` loop.

6. Given the following declarations, the prototypes on the left, and the calls on the right, answer *good* if a function call is legal and meaningful. If there is an error in the call, show how you might change it to match the prototype.

```
int j, a[5];
float f, flo ;
double x, dub[4];
```

(a)	`double fun( double d[] );`	`x = fun( dub[] );`
(b)	`void fill( double d );`	`x = fill( dub );`
(c)	`int fix( float f ) ;`	`fix( flo[5] );`
(d)	`int hack( int a[] );`	`j = hack( a );`
(e)	`double q( double d[], int n );`	`x = q( dub[0], a[0] );`

7. Use the following definitions in this problem:

```
#define  LIMIT 5
int j;
double load[LIMIT];
```

(a) Draw an object diagram for the array named `load`; identify the slot numbers in your drawing. Also show the values stored in it by executing the following loop:

```
for ( j=0; j<LIMIT; j++ ) {
    if ( j % 2 == 0 ) load[j] = 10.2 * (j + 1);
    else load[j] = (j + 1) * 2.5;
}
```

(b) Trace the following loop and show the output exactly as it would appear on your screen. Show your work.

```
for (j = 0; j < LIMIT; ++j) {
    if (j <= LIMIT / 2)  printf( "\t%6.3g", load[j] );
    else printf( "\t%5.2f", load[j] -.05 );
}
```

8. The following diagram shows an array of odd integers. Declare and initialize a parallel array of type `int` that contains 1 (true) in the slot corresponding to every prime number, and 0 (false) for the nonprime numbers.

3	5	7	9	11	13	15	17	19	21	23	25

9. Consider an array containing six data items that you must sort using the selection sort algorithm. How many data comparisons does the algorithm perform in the `find_min()` function? How many times is `find_min()` called, and how many comparisons are made (total) during all these calls? How many (total) data swaps does `sort_data()` perform?

10. Suppose you are searching for an item in a sorted array of N items. Using a sequential search algorithm, how many items are you likely to check before you find the right one? Is this number the same whether or not the item is present in the array? Express the answer as a function of N.

11. The selection sort in the text generated a list of values in ascending order. How would you change the algorithm to generate numbers in descending order?

12. Consider the following list of numbers: 77, 32, 86, 12, 14, 64, 99, 3, 43, 21. Following the example in Figure 10.24, show how the numbers in this list would be sorted after each pass of the selection sort algorithm.

10.10.2 Using Pencil and Paper

1. Draw a flow diagram for the main program in Figure 10.16 and for the `divisible()` function in Figure 10.17 (you may omit the details of the other function). In your diagram, show how control flows from one function to the other and back again.
2. An array will be used to store the serial numbers of the printers in the lab. It is declared thus:
   `long printer_ID[10];`
   (a) Draw an object diagram of this array.
   (b) What is its base type? Its length? Its size?
   (c) Write a loop that will read data from the keyboard into this array; the loop should end when the user enters a negative number. Store the actual number of data items in the variable named `count`.
   (d) Write a loop that will print out all `count` data items from the array.
3. Array and function declarations.
   (a) Write a declaration with an initializer for the array of small integers pictured here:

   scores

[0]	[1]	[2]	[3]
96	68	79	93

   (b) Write a complete function that takes this array and an integer `n` as parameters, tests each array element, and prints all array elements greater than `n`.
   (c) Write a prototype for this function.
4. Use the following definitions in this problem:

```
#define  LIMIT 7
int j;
short puzzle[LIMIT], crazy[LIMIT];
```

   (a) Draw an object diagram for the array named `puzzle`; identify the slot numbers in your drawing. Also show the values stored in it by executing the following loop. This loop performs an illegal operation. Your job is to figure out what will happen and why.

```
for (j = LIMIT; j > 0; --j) {
    if (j+4 > j*2) {
        puzzle[j] = j*2;
        crazy[LIMIT-j] = j + 2;
    }
    else {
        puzzle[j] = j/2;
        crazy[LIMIT-j] = j - 2;
    }
}
```

   (b) Trace the following loop and show the output exactly as it would appear on your screen. Show your work.

```
for (j = 0; j < LIMIT; ++j) {
    if (j % 2 == 1)
        printf( "\t%5i %3i", puzzle[j], crazy[j] );
    else  printf( "\t%3i %5i", crazy[j], puzzle[j] );
}
```

5. The following are two declarations and a `for` loop.
   (a) What is the output?
   (b) Rewrite the code using a `while` loop instead of the `for` loop.

```
int a[12] = {2, 4, 8, 3, 9, 27, 4, 16, 64, 5, 25, 125};
for (int k = 1; k < 10; k += 2) printf("%i: %i \n", k, a[k]);
```

6. Given the following declarations, the prototypes on the left, and the calls on the right, answer *good* if a function call is legal and meaningful. If there is an error in the call, show how you might change it to match the prototype.

```
int j, ary[5];
float f, flo ;
double x, dub[4];
```

(a)	`double list( double d[] );`	`x = list( ary );`
(b)	`void handle( int k );`	`handle( ary[5] );`
(c)	`void bank( float f, int n );`	`bank( flo[5], j );`
(d)	`int days( int a[] );`	`j = days( &flo );`
(e)	`int area( double d );`	`x = area( dub[0] );`

7. In the spots indicated below, write a prototype, function header, and call for a function named `MISSING`. Its parameters are an array of student assignment scores and the number of assignments that have been graded and recorded. The function will look at the data and return the number of nonzero scores in the array. You need not actually write the whole function, just fill in the indicated information.

```
#define  MAX  10
// insert prototype here

int main( void )
{
    int assignments[MAX];   // grades
    int actual;             // # of assignments so far.
    int done;               // # of nonzero grades.
    // put call here
}

// put function header here
```

8. An unsuccessful search for an item in sorted and unsorted data arrays will require different numbers of comparisons. Compare a sequential search on these two types of data and explain why they are different (in terms of the number of comparisons performed).
9. Modify the code in the sequential search function in Figure 10.21 so that it assumes the data in the array are sorted in descending order. Do not search any more positions than necessary. Still return a value of `-1` if the key value cannot be found.
10. Consider an array containing N data items that you must sort using the selection sort algorithm. How many data comparisons does the algorithm perform? How many data swaps does it perform? Express the answer as a function of N.
11. Consider the following list of numbers: 77, 32, 86, 12, 14, 64, 99, 3, 43, 21. Following the example in Figure 10.26, show how the numbers in this list would be sorted after each pass of a selection sort algorithm that sorts numbers in *descending* order.

10.10.3 Using the Computer

1. Seeing bits.
   The program in Figure 4.23 shows how to convert an integer to any selected base. Obviously, this program works for base 2, binary notation. Modify this program so that it inputs a number and prints the equivalent value in binary notation. Do not print it in the expanded form of the previous program, but as a series of ones and zeros. You will need to store the binary digits in an array and print them later in the opposite order, so that the first digit generated is the last printed. For example, if the input were 22, the output should be 10110.

2. Global warming.
   As part of a global warming analysis, a research facility tracks outdoor temperatures at the North Pole once a day, at noon, for a year. At the end of each month, these temperatures are entered into the computer and processed. The operator will enter 28, 29, 30, or 31 data items, depending on the month. You may use -500 as a sentinel value after the last temperature, since that is lower than absolute 0. Your main program should call the `read_temps()`, `hot_days()`, and `print_temps()` functions described here:

   (a) Write a complete specification for this program.
   (b) Write a function, `read_temps()`, that has one parameter, an array called `temps`, in which to store the temperatures. Read the real data values for one month and store them into the slots of an array. Return the actual number of temperatures read as the result of the function.
   (c) Write a function, `hot_days()`, that has two parameters: the number of temperatures for the current month and an array in which the temperatures are stored. Search through the temperature array and count all the days on which the noon temperature exceeds 32°F. Return this count.
   (d) Write a function, `print_temps()`, with the same two parameters plus the count of hot days. Print a neat table of temperatures. At the same time, calculate the average temperature for the month and print it at the end of the table, followed by the number of hot days.

3. The tab.
   An office with six workers maintains a snack bar managed on the honor system. A worker who takes a snack records his or her ID number and the price on a list. Once a month, the snack bar manager enters the data into a computer program that calculates the monthly bill for each worker. No item at the snack bar costs more than $2, and monthly totals are usually less than $100.

   (a) Write a complete specification for this program.
   (b) Using a top-down development technique, write a main program that will call functions to generate a monthly report. These functions are described here. Declare an array of `float`s named `tabs` to store total purchase amounts for each member and the guests.
   (c) The `purchases()` function should have one parameter, the `tabs` array. This function should allow the manager to enter two data items for each purchase: the price and the ID number of the worker who made the purchase. The ID numbers must be integers between 1 and 6. In addition, the code 0 is used for guests, whose bills are paid by the company. As each purchase is read, the amount (in dollars and cents) should be added to the array slot for the appropriate worker. When the manager enters an ID code that is not between 0 and 6, it should be considered a sentinel value and a signal to end the loop and return from the function. At that time, the array should contain the total purchases for each worker and for the guests.
   (d) The `bills()` function should have one parameter, the `tabs` array. Print a bill for each worker, giving the ID number and the amount due.

4. Payroll.
   The Acme Company has some unusual payroll practices and keeps the information in its personnel database in a strange way. The firm never has more than 200 employees and pays all its employees twice a month, according to the following rules:

   (a) If the person is salaried, the pay rate will be greater than $1,000. There are 24 pay periods per year, so for one period, `earnings = payrate / 24`.
   (b) If the person is paid hourly, the pay rate will be between $5 and $100 per hour and the earnings are calculated by this formula:
   `earnings = payrate * hours`.
   (c) A pay rate less than $5 per hour or between $100 and $1,000 per hour is invalid and should be rejected.

   (a) Write a complete specification for this program.

(b) Using a top-down development process and following the example of Figure 10.13, write a main program that prints the bimonthly payroll report. Since the number of employees changes frequently, `main()` should prompt for this information. Then call functions to perform the calculations and produce a report. Use the functions suggested here, and add more if that seems appropriate. You will need a set of parallel arrays to hold the ID number, pay rate, hours worked, and earnings for each employee.

(c) Following the example of Figure 10.21, write a function, `get_earnings()`. Read the data for all the employees from the keyboard into the parallel arrays.

(d) Write a function, named `earn()`, that uses the pay rate and hours worked arrays to calculate the earnings and fill in the earnings array.

(e) Write a function, named `pay()`, to calculate and return the earnings for one employee. If the pay rate is invalid, print an error comment and return 0.0.

(f) Write a function, named `payroll()`, that prints a neat table of earnings, showing all the data for each person. Also calculate the total earnings and print that value at the end of the list of employees.

5. Guess my weight.
At the county fair a man stands around trying to guess people's weight. You've decided to see how accurate he is, so you collect some data. These data are a set of number pairs, where the first number in the pair is the actual weight of a person and the second number is the weight guessed by the man at the fair. You decide to use two different error measures in your analysis: absolute error and relative error. Absolute error is defined as $E_{abs} = W_{guess} - W_{real}$, where W_{guess} and W_{real} are the guessed and real weights, respectively. The units of this error are pounds. The relative error is defined by $E_{rel} = 100 \times E_{abs}/W_{real}$, where the result of this equation is a percentage. Write a program that will input the set of weight pairs you accumulated, using a function with a sentinel loop to read the data. The number of weight pairs should be between 1 and 100. Write another function that will calculate both the absolute and relative errors of the guesses and display them in a table. Finally, compute and print the average of the absolute values of the absolute errors and the average of the absolute values of the relative errors.

6. Having fen.
Ms. Honeywell, an American businessperson, is preparing for a trip to Beijing, China, and is worried about keeping track of her money. She will take a portable computer with her, and wants a program that will sum the values of the Chinese fen (coins and bills) she has and convert the total to American dollars. Fen come in denominations of 1, 5, 10, 20, 50, 100, 200, 500, 1,000, and 2,000. The exchange rate changes daily and is published (in English) in the newspaper and on television. Write a program for Ms. Honeywell that will prompt her for the exchange rate and then for the number of fen she has of each denomination. Total the values of her fen and print the total as well as the equivalent in American dollars. Implement the table of fen values as a global constant array of integers.

7. A function defined by a table.
Write a function that computes a tax rate based on earned salary according to the following table. In this function, if the salary value is not given exactly, use the rate for the next lower salary in the table. For example, use the rate 20% for $38,596.

Salary ($)	**Tax Rate (%)**
0	0
10,000	5
20,000	12
30,000	20
40,000	33
50,000	38
60,000	45
70,000	50

Write a small program that will input a salary from the user, call your function to compute the tax rate, and then print the tax rate and the amount of tax to be paid. Validate the input salary so that it does not fall outside of the salary range in the table.

8. Moving average.
   Some quantities, such as the value of a stock, the size of a population, or the outdoor temperature, have frequent small fluctuations but tend to follow longer-term trends. It is helpful to evaluate such quantities in terms of a moving average; that is, the average of the most recent N measurements. (N normally is in the range 3...10.) This technique "smoothes out" the most recent fluctuations, exposing the overall trend. Write a program that will compute a moving average of order N for the price of a given stock on M consecutive days, where $M > N + 4$. To do this, first read the values of N and M from the user. Next read the first N prices and store them in an array. Then repeat the process below for the remaining $M - N$ prices:

   (a) Compute and print the average of the N prices in the array.
   (b) If all M values have been processed, quit and print a termination message.
   (c) Otherwise, eliminate the value in array slot 0 and shift the other $N - 1$ values one slot leftward.
   (d) Read a new value into the empty slot at the end of the array.

9. A bidirectional sort.
   Another sorting algorithm is similar to the selection sort, the "cocktail shaker" sort. This algorithm differs from the selection sort in the way it selects the next item from the array. Our selection sort always picks the maximum value from the remaining values and swaps it into the beginning of the sorted portion. For the cocktail shaker sort, the first pass finds the maximum data value and moves it to one end of the array. The second pass finds the minimum remaining value and moves it to the other end of the array. Subsequent passes alternate choosing the maximum and minimum values from the remaining data and moving that value to the appropriate end of the array. Eventually the two ends meet in the middle and the data are sorted. Write a program that implements the cocktail shaker sort just described and uses it to sort data sets containing up to 100 values.

Chapter 11

An Introduction to Pointers

In this chapter, we introduce the final remaining primitive data type, the pointer, which is the address of a data object. We show how to use pointer literals and variables, explain how they are represented in the computer, and present the three pointer operators: `&`, `*`, and `=`. We explain how, using pointer parameters, more than one result can be returned from a function. Pointers are covered here only at an introductory level and will be considered in greater depth in later chapters.

11.1 A First Look at Pointers

A **pointer** is like a pronoun in English; it can refer to one object now and a different object later. Pointers are used in C programs for a variety of purposes:

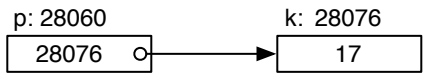
- To return more than one value from a function (using call by value/address).
- To create and process strings.
- To manipulate the contents of arrays and structures.
- To construct data structures whose size can grow or shrink dynamically.

In this chapter we study only the first of these uses of pointers; the others will be explored in Chapters 12, 13, 16, and 17.

11.1.1 Pointer Values Are Addresses

A pointer value (also called a *reference*) is the address (i.e., a specific memory location) of an object. A pointer variable can store different references at different times. If `p` is a pointer variable and the address of `k` is stored in `p`, then we say `p` *points at* `k` or `p` contains *is a reference to* `k` or `p` *refers indirectly to* `k`. In the other direction, we say that `k` *is pointed at by* `p` or `k` *is the* **referent** *of* `p`. In object diagrams, we represent pointer values as arrows and pointer variables as boxes from which arrows originate. The tail of each pointer arrow is a small circle that can be "stored" in a pointer variable; the head of the arrow points at its referent. Figure 11.1 shows a pointer variable named `p` that points at the integer variable `k`, which itself has a value of 17. Figure 11.1 shows a simplified way to diagram pointer variables, without the explicit memory addresses.

Pointer variables. To declare a **pointer variable**, we start with the **base type of the pointer**; that is, the type of object it can reference. After the type comes a list of pointer variable names, each preceded by an asterisk, as shown in Figure 11.3. A common mistake is to omit the asterisk in front of `p2`. This makes it a simple integer rather than a pointer. The asterisk must be written before each name, not just appended to the base type. A pointer can refer only to objects of its base type. Although a given pointer can refer to different objects at different times, we cannot use it to refer to an `int` at one moment and a `double` later. Therefore, both `p1` and `p2` in the diagram can be used to refer to an integer variable `k`, but neither could refer to a `char` or a `double`. When we use pointers in expressions, the base type of the pointer lets C know the actual type of the values that can be referenced, so that it can compile appropriate operations and conversions.



`p` is a pointer variable stored at address 28060.
`k` is an integer variable stored at address 28076.
The contents of `k` is the integer 17.
The contents of `p` is the address of `k`, which is 28076.
The arrow is a reference to `k`
We say that `p` points at `k` or `p` refers to `k`
We say that `k` is the referent of `p`.

Figure 11.1. A pointer and its referent.

Pointer initialization. When a pointer is declared without an initializer (as an **unitialized pointer**), memory is allocated for it but no address is stored there, so any value previously stored in that memory location remains. Therefore, a pointer always points at *something*, even when that thing is not meaningful to the current program. It could be an actual object in the program, a random memory address in the middle of the code, or an illegal address that does not even correspond to a memory location the program is allowed to access. Most C compilers will not detect or give error comments about uninitialized pointers. If a program unintentionally uses one, the consequences will not be discovered until run time and can be quite unpredictable, depending on the random contents of memory when the program begins execution. Anything can happen, from apparently correct operation to strange output results to an immediate program crash. To emphasize the unknown consequences of using such pointers, we diagram an uninitialized pointer as a wavy arrow that ends at a question mark, as in the middle diagram in Figure 11.2.

The `NULL` pointer. Many data types include a literal value that means "nothing": for type `double` this value is 0.0, for `int` it is 0, and for `char` it is `\0`. There also is a "zero" value for pointer types, the `NULL` **pointer**, and it is defined in `stdio.h`. We store the value `NULL` in a pointer variable as a sign that it points to nothing. `NULL` is represented in the computer as a series of 0 bits and, technically, is a pointer to memory location 0 (which contains part of the operating system). In diagrams, we represent `NULL` using the electric "ground" symbol, as shown in the rightmost part of Figure 11.2, or an arrow that loops around, crosses itself, and ends in midair. One of the basic uses of `NULL` is to initialize a pointer, to avoid pointing at random memory locations. We often initialize pointers to `NULL` (see Figure 11.4). This indicates that the pointer refers to nothing, as opposed to something undefined.

We diagram a pointer variable as a box containing an arrow (a pointer). Here, we diagram three pointer variables. The first points at an integer, the second is uninitialized, and the third contains the `NULL` pointer.

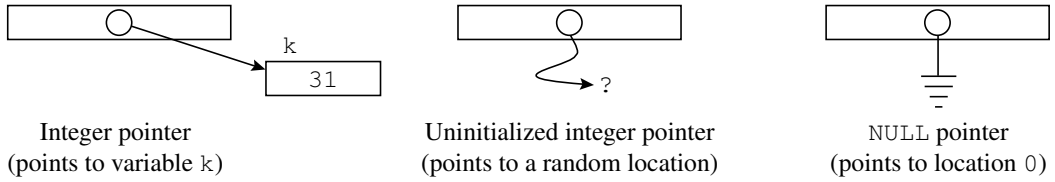


Figure 11.2. Pointer diagrams.

Two uninitialized pointer variables are declared.

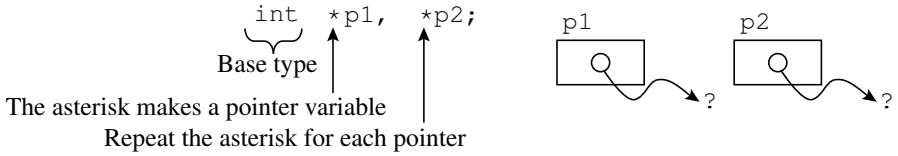


Figure 11.3. Declaring a pointer variable.

An integer pointer variable is declared and initialized to NULL.

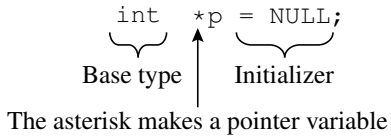


Figure 11.4. Initializing a pointer variable.

Which nothing is correct? A pertinent question is, What is the difference between 0.0, 0, \0, and NULL? First, even though all are composed entirely of 0 bits, they are of different lengths. A double 0.0 often is 8 bytes long, but the character \0 is only 1 byte long. The NULL pointer is the length of pointers on the local system, which, in turn, is determined by the number of bytes required to store a memory address on that system. Second, these zero values have different types and a C compiler treats them like other values of their type when it produces compile-time error comments. For example, if the value 3.1 would be legal in some context, then the value 0.0 also would be legal. The value 0 would be acceptable and require conversion, while the value NULL would be inappropriate and cause a compile-time error.

The implementation of pointers. A pointer variable is a storage location in which a pointer can be stored. The pointer itself is the address of another variable. Thus, a *pointer variable* has an address and also contains an address; a *pointer* is the address of its referent. The dual nature of pointers can be confusing, even to experienced programmers. Sometimes it helps to understand how pointers actually are implemented in the computer at run time. The basics are illustrated in Figure 11.5. There, we declare two integer variables and two integer pointers, then diagram the variables created by the declarations. (We assume that an int fills 2 bytes and a pointer 4.) Hypothetical memory addresses are shown above the boxes to help explain the actions of certain pointer operations in the next section.

11.1.2 Pointer Operations

C has three basic operators[1] that deal with pointers: &, *, and =. In addition, scanf() supports a format specifier for printing pointer values. While each operation is straightforward, sometimes the use of pointers can be confusing.

[1]Pointers to aggregate types follow different syntactic rules than pointers to simple variables and use an additional operator. The differences are explained in Chapters 13 and 17.

```
int *pt = NULL;      // An int pointer variable, initialized to NULL.
int *p;              // A pointer variable that can point at any int.
int k = 17;          // An integer variable initialized to 17.
int m;               // An uninitialized integer variable.
```

Storage for these variables might be laid out in the computer's memory as follows:

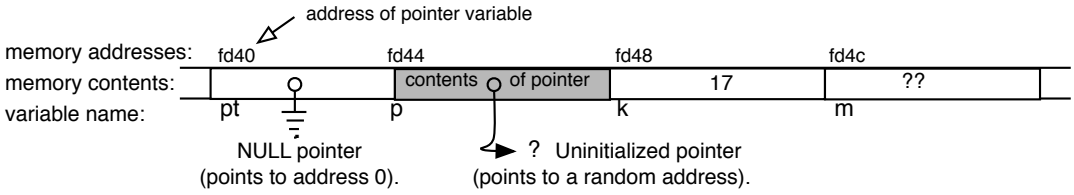


Figure 11.5. Pointers in memory.

References and indirection. The expression `&k` (the `&` **of a variable**) gives us the address of the variable `k`, also called a *reference to* `k`. The **dereference operator**, `*`, also called **indirection**, is the inverse of `&`; that is, `*p` means the referent of `p` (the object at which `p` points). Since, by definition, `&` and `*` are inverse operations, `*&k == k` and `&*p == p`.[2]

The expression `*p` stands for the referent the same way a pronoun stands for a noun. If `k` is the referent of the pointer `p`, then `m = *p` means the same thing as `m = k`. We say that we *dereference* `p` *to get* `k`. Similarly, `*p = n` means the same thing as `k = n`. Longer expressions also can be written; anywhere that you might write a variable name, you can write an asterisk and the name of a pointer that refers to the variable. For example, `m = *p + 2` adds 2 to the value of `k` (the referent of `p`) and stores the result in `m`. Since the dereference operator and the multiply operator use the same symbol, `*`, the C compiler distinguishes between them by context. If the operator has operands on both the left and right, it means "multiply." If it has only one operand, on the right, it means "dereference."

Pointer assignment. As just seen, a pointer can be involved in an assignment operation in three ways:

1. We can make an assignment directly *to* a pointer, as in `p = &k`.
2. We can access a value *through* a pointer and assign it to a variable, as in `m = *p`.
3. We can use a pointer to make an *indirect assignment* to its referent, as in `*p = m+2`.

Direct assignments are useful for string manipulation (Section 12.1). Indirect assignment and access through a pointer are used with call by value/address parameters (Section 11.2). Figure 11.6 illustrates the three kinds of pointer assignment using the variables declared in Figure 11.5.

Notes on Figure 11.6. Pointer operations. To show the actual relationship between a pointer variable, its contents, and its referent, the addresses and contents of each pointer and variable in this program have been printed and diagrammed.

First box: pointer declarations. Figure 11.5 contains a diagram of the initial contents of the two pointers and the variables declared here.

Second box: direct pointer assignments. There are two ways to assign a value to a pointer: assign either the address of a variable of the correct base type or the contents of another pointer of a matching type.

- The base type of `p` is the same as the type of `k`, so we are permitted to make `p` refer to `k` with the assignment `p = &k`. In the diagram below the output, note that the address of `k` is written in the variable `p` and that the arrow coming from `p` ends at `k`. We say that `p` *refers to* (or *points at*) `k`.
- Pointers `p` and `pt` are the same type, so we can copy the contents of `p` into `pt` with the assignment `pt = p`. This causes the contents of `p` (which is the address of `k`) to be copied into `pt`. In the diagram, you can see that both `p` and `pt` contain arrows with heads pointing at `k`.

Third box: indirect pointer assignments.

- The first line dereferences `p` to get `k`, then fetches the value of `k` and adds 2 to it. The result (19) is stored in `m`; the value stored in `k` is not changed at this time. It is not necessary to use parentheses in this expression because `*` has higher precedence than `+`.
- The second line copies the value of `m` into the referent of `p`, which still is `k`. This changes the value of `k` from 17 to 19, as diagrammed.

Fourth box: Output.

The output shows how memory is laid out in our computer, which is running Gnu C[3] and uses 4-byte integers. The memory addresses are printed using the `%p` format specifier, which prints numbers in unsigned hexadecimal notation, with a leading `0x`. Compare this diagram and the output to the memory diagram in Figure 11.5.

[2] The combinations `*&k` and `&*p`, therefore, are silly and not written in a program.

[3] This is the open-code C compiler from the Free Software Foundation.

11.2 Call by Value/Address

Most parameter values are passed from the caller to a function using **call by value**; that is, by copying the argument value from the caller's memory area into the parameter variable within the function's memory area. However, there are three situations in C in which copying the argument value is not done, is inefficient, or cannot do the job that is required:

1. When an array of any size is being passed (discussed in Chapter 10).
2. When a function needs to return more than one result (discussed in this section).
3. When a large structure is being passed (discussed in Chapter 13).

All these situations are handled in C by passing an address, not a value, to the function.

This short program connects the program fragments from this section and adds output statements that show the contents and addresses of the variables.

```
#include <stdio.h>

int main( void )
{
    int * pt = NULL;    // An int pointer variable, initialized to NULL.
    int * p;            // A pointer variable that can point at any int.
    int   k = 17;       // An integer variable initialized to 17.
    int   m;            // An uninitialized integer variable.

    p = &k;             // Use & to set a pointer to k's memory location.
    pt = p;             // Copy a pointer.

    m = *p + 2;         // Add 2 to the value of p's referent; store result in m.
    p = m;              // Copy the value of m into p's referent.

    printf( "address of  p: %p  contents of  p: %p\n", &p, p );
    printf( "address of pt: %p  contents of pt: %p\n", &pt, pt );
    printf( "address of  k: %p  contents of  k: %i\n", &k, k );
    printf( "address of  m: %p  contents of  m: %i\n", &m, m );
    return 0;
}
```

When compiled and run on a system with 4-byte integers, the following output was produced:

```
address of  p: 0xbffffd44  contents of  p: 0xbffffd48
address of pt: 0xbffffd40  contents of pt: 0xbffffd48
address of  k: 0xbffffd48  contents of  k: 19
address of  m: 0xbffffd4c  contents of  m: 19
```

This corresponds to the memory layout that follows. Here, the memory addresses are shown in hexadecimal notation and only the last four digitsare shown.

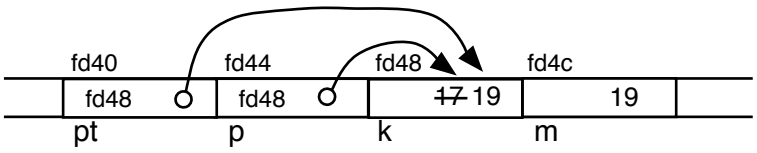


Figure 11.6. Pointer operations.

Chapter 10 discusses call by reference, which is used to pass array arguments[4]. In this method, we pass only the address of the first array slot, not the array's list of values, into the function. Within the function, the reference is transparent to the programmer. That is, references to the array, with or without subscripts, are written exactly the same way in the function as they are in the caller. This makes a large amount of data available to the function efficiently and allows the function to store information into the array. As demonstrated in Section 10.4, a program can pass an empty array into a function, which then fills it with information. When the function returns, that information is in the array and can be used by the caller.

This chapter discusses **call by value/address**. This is a special case of call by value in which the argument is the address of a variable or a pointer to a variable in the caller's memory area. This argument must be stored in a pointer parameter in the function's area. This gives the function full access to the variable that belongs to the caller. This much is exactly like call by reference. However, the similarity ends there because call by value/address is *not* transparent to the programmer; an *indirection* or *dereference* operator must be used in the function's code to access the underlying argument in the storage area of the caller. The remainder of this section explains, in detail, how this works and how to use it.

11.2.1 Address Arguments

When call by value is used to pass an address argument, the function receives a reference to the caller's variable and can read from and write to that variable. By reading the caller's variable, the function obtains the value placed there by the calling program. By writing (storing) into the caller's variable, the function can pass a value back to the calling program. When the function ends, the value that it wrote is still in the caller's variable.

You already are familiar with one function that requires an address argument: `scanf()`. When calling `scanf()`, we use an `&` with arguments of simple types such as `long` and `double`. This technique is not limited to `scanf()`. In any function, we can pass the address of a simple variable by writing an `&` followed by the name of the variable. The function must have a corresponding parameter of a pointer type.

Another way to pass the address of a simple variable is to write the name (with no ampersand) of a pointer variable that refers to it. In this case, the value of the pointer, which is the address of its referent, is passed.

11.2.2 Pointer Parameters

A function declares that it is expecting an address argument from the caller by declaring the corresponding parameter with a pointer type.

When the argument is an array, as in the statistics program of Figures 11.12 through **??**, the corresponding parameter can be declared either as a pointer or as an array with an unspecified dimension. For example, if the actual argument were an array of integers, a formal parameter named `ara` could be declared as either `int* ara` or `int ara[]`. The two declarations are identical in most respects, but it is cleaner style to use the latter for arrays. In either case, the parameter name can be used with subscripts within the function to refer to the array elements.

For nonarrays, a **pointer parameter** is declared with an asterisk. When the function is called, the address argument is copied into the corresponding pointer parameter. The base type of the pointer parameter must match the type of the address argument. For example, if a function expects to receive the address of an integer variable, it should declare the corresponding parameter to be of type `int*` (as in the function `f1()` in Figure 11.7). The result is an in-out parameter.

Sometimes an address argument is used to pass a large data structure efficiently[5]. In this case, we may want to have an input parameter that does not permit outward flow of information. To achieve this, the parameter is declared as a constant pointer. For example, `const int * xp` (as in the function `f3()` in Figure 11.7).

Notes on Figure 11.7. Call by value/address. Here we have two simple functions that illustrate two techniques for altering the value of a variable in the caller's memory area.

First box: prototypes. Functions `f1()` and `f2()` have the same prototype, having one parameter that is an integer pointer. The pointer permits the function to export information. C does not provide a way to restrict a parameter to output-only. In function `f2()`, the usage is output-only, but the syntax would permit two-way flow of information.

[4]Call-by-reference is much more widely used in `Java` and in C++.
[5]This will be discussed in Chapter 13.

This program illustrates syntax for call by value/address and gives examples of in, in-out and output parameters.

```
#include <stdio.h>
#include <math.h>

void f1( int * xp );          // uses an in/out parameter
void f2( int * xp );          // uses an output parameter
double f3( const int * xp );  // uses an in parameter

int main( void )
{
    int k = 1;
    double answer;

    puts( "\n Call by value/address Demo." );

    printf( "Original value of k:  %i\n", k );
    f1( &k );                 // This function changes the value of k.
    printf( "After f1(), changed value of k:  %i\n", k );

    f2( &k );                 // This function changes the value of k.
    printf( "After f2(), input is stored in k:  %i\n", k );

    answer = f3( &k );        // This function cannot change k.
    printf( "After f3(), the square root of %i = %.3f\n", k, answer );

    return 0;
} // ---------------------------------------------------------------------
void f1( int * xp )           // xp is an in/out parameter
{
    *xp = *xp + 2;            // add 2 to the old value of xp's referent.
}

// -------------------------------------------------------------------
void f2( int * xp )           // xp is an output parameter.
{
    printf( "Enter an integer:  " );
    scanf( "%i", xp );
}

// -------------------------------------------------------------------
double f3( const int * xp )   // xp is an in parameter.
{
    return sqrt( *xp );       // use xp's referent (no change).
}
```

Figure 11.7. Call by value/address.

This version of swap does not work because call by value is used to pass the parameters.

```
#include <stdio.h>
void badswap( double f1, double f2 );
int main( void )  {
    double x = 10.2, y = 7;
    printf( "Before badswap: x=%5.1g  y=%5.1g\n", x, y );
    badswap( x, y );
    printf( "After badswap:  x=%5.1g  y=%5.1g\n", x, y );
}
```

```
void badswap( double f1, double f2 )  {
    double swapper = f1;
    f1 = f2;
    f2 = swapper;
}
```

Figure 11.8. A swap function with an error.

Second box and function `f1`***: indirect reference through an in-out parameter.***

- We pass `&k`, the address of `k`, as the argument to the pointer parameter in `f1()`. This address will be stored in the memory location for `xp` when the call occurs, so `xp` will refer to `k`.
- We use the initial value of `k` in this function by writing `*xp`. After adding 2, we store the result back into `k` by writing `*xp =`
- We call `xp` an *input* parameter because we use the information that the caller stored in it. We call it an *output* parameter because we change that information. Thus, it is an in-out parameter.
- The value of `k` is displayed before and after the function call to show that the call both used and changed the value of `main`'s variable. (See the output, below.)

Third box and function `f2`***: using an output parameter.***

- We use `xp`, an output parameter here, to return a value from `scanf()` back to the caller. The new value is printed in `main()` after the function call.
- We want to pass the address of `k` to `scanf()` so that input can be stored in `k`. We could do this by writing `&*xp` as the argument, but this simplifies to just `xp`, which contains the original address of `k` as it was passed into the function. The `scanf()` call stores a value directly into `main()`'s variable `k`.

Fourth box and function `f3`***: using a*** `const *` ***input parameter.***

- Sometimes our data is stored in large structures (presented in Chapter 13) that occupy many bytes of memory. It is undesirable to copy the whole structure because of the time and the space that would consume. In such a situation we use call by value/address. However, we also want to protect the caller's variable against the possibility of being changed by the function or by any other function it might call. To do this, we can use a `const` *pointer* parameter, also known as a *read-only parameter*.
- In this function, we use a `const *` parameter. This lets us use but not change the value of `main`'s variable. To access that value, we write `*xp`.

Sample output.

```
Original value of k: 1
After f1(), changed value of k: 3
Enter an integer: 7
After f2(), input is stored in k: 7
After f3(), the square root of 7 = 2.646
```

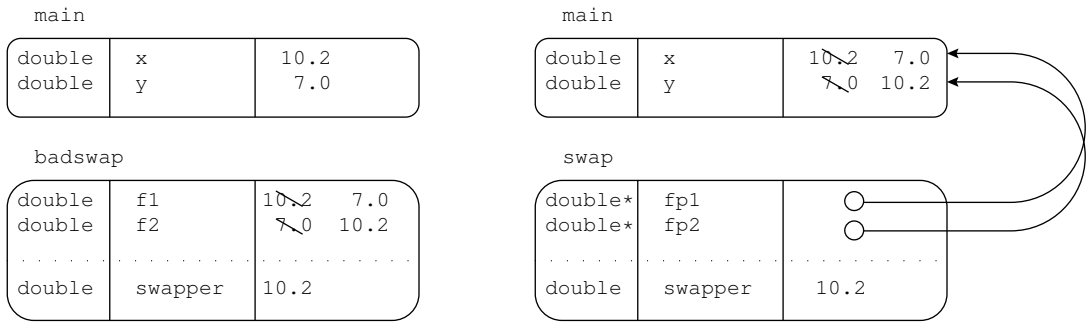


Figure 11.9. Seeing the difference.

11.2.3 A More Complex Example

In some situations, call by value does not provide enough information to enable a function to do its task. The simplest example consists of a function that wants to swap the values of its two parameters. In Figures 11.8 through **??**, we examine two possible versions of this simple **swap function**. The first fails to swap the values; the second works properly.

The memory use for Figure 11.8 is diagrammed on the left. Each function has its own memory area, and values of the arguments are copied from variables of `main()` into the parameters of `badswap()`. Assignments change only the values in the parameters.

The memory use for Figure 11.10 is diagrammed on the right. The arguments here are pointers to variables of `main()`. Indirect assignments made through these pointers change the underlying variables.

Notes on Figures 11.8, 11.10, and 11.9. Seeing the difference in the swap functions.

First and second boxes, Figure 11.8: the first version of the swap. The first box declares two parameters as type `double`, not `double*`, so the arguments will be passed by value. When `main()` calls `badswap(x,y)` (second box), the current values of `x` and `y` are copied into `badswap()`'s parameters `f1` and `f2`. The function receives these values, not the addresses of the variables. This is shown in the diagram for `badswap()` on the left side of Figure 11.9.

Third box, Figure 11.8: the bad swap. When `badswap()` swaps the values, it swaps the copies stored in the parameters, not the originals. In the diagram, note that the assignments in `badswap()` cause no changes to the variables of `main()`. The program output is

```
Before badswap: x= 10.2  y=  7.0
After badswap:  x= 10.2  y=  7.0
```

This version of swap works because call by value/address is used to pass the parameters.

```
#include <stdio.h>
void swap( double * fp1, double * fp2 );
int main( void )  {
    double x = 10.2, y = 7;
    printf( "Before swap: x=%5.1g  y=%5.1g\n", x, y );
    swap( &x, &y );
    printf( "After swap:  x=%5.1g  y=%5.1g\n", x, y );
}
void swap( double * fp1, double * fp2 )  {
    double swapper = *fp1;
    *fp1 = *fp2;
    *fp2 = swapper;
}
```

Figure 11.10. A swap function that works.

Problem scope: Calculate the arithmetic mean, variance, and standard deviation of N experimentally determined data values.

Input: The user will specify the number of data values, N, then N data values will be typed in. These will be real numbers.

Restrictions: No more that 50 data values will be processed.

Formulas:

$$\begin{aligned} Mean &= \frac{\sum_{k=1}^{N} x_k}{N} \\ Variance &= \frac{\sum_{k=1}^{N} (x_k - mean)^2}{N-1} \qquad \text{for } N < 20 \\ Variance &= \frac{\sum_{k=1}^{N} (x_k - mean)^2}{N} \qquad \text{for } N \geq 20 \\ Standard\ deviation &= \sqrt{Variance} \end{aligned}$$

Output required: The mean, variance, and standard deviation of the N points, accurate to at least two decimal places.

Figure 11.11. Problem specifications: Statistics.

First and second boxes, Figure 11.10: the good version of swap. In contrast, this version uses call by value/address (first box). The arguments are two addresses (second box), which are stored in the `swap()` parameters `fp1` and `fp2`. In the diagram, you can see that the parameters of `swap()` are pointers to the variables of `main()`.

Third box, Figure 11.10: the good swap. The function receives pointers to the variables, not copies of their values. When `swap()` says `swapper = *fp1`, it copies the value of the referent of `fp1` into `swapper`. When it executes `*fp1 = *fp2;`, it copies the value of the referent of `fp2` into the referent of `fp1`. This changes the value of the corresponding variable in `main()`, not the pointer in the `swap()` parameter. We say that `*fp1 = *fp2;` fetches the value *indirectly through* `fp2` and stores it *indirectly through* `fp1`. The final output from this program is

```
Before swap: x= 10.2  y=  7.0
After swap:  x=  7.0  y= 10.2
```

11.3 Application: Statistical Measures

In many experiments, the measured values of the experimental variable are distributed about the mean value in a bell-shaped curve centered on the **mean value of the array**; that is, the arithmetic average of the data values. Such a distribution is called a *normal*, or *Gaussian*, *distribution*.

The **variance** and **standard deviation** of a set of data are measures of how significantly the measured values differ from the true mean of the distribution. To compute these measures accurately, we need at least 20 data values. For fewer than 20 data points, we use the slightly different formulas, shown in Figure 11.11, to estimate the variance and standard deviation. In these equations, $x_1, x_2, x_3, \ldots, x_N$ are the data values and $N - 1$ is called the *degree of freedom* of the data.

In the next program example, we introduce a method for computing these statistical measures for N data values: $x_1, x_2, x_3, \ldots, x_N$.[6] The program specification is given in Figure 11.11 and the main program is shown in Figure 11.12. To keep the flow of logic in all parts of the program simple and uniform, major phases of the computation have been written as separate functions that work with a data array. We call `get_data()` to read the data into an array, then pass that array to the `stats()` function, and finally, print the answers. A call graph is given in Figure 11.13 and the two array-processing functions are found in Figure 11.14.

[6] J. P. Holman, *Experimental Methods for Engineers*, 7th ed. (New York: McGraw-Hill, 2001).

The specification for this program is in Figure 11.11 and the call graph is in Figure 11.13. The three programmer-defined functions are in Figures **??** and **??**.

```
#include <stdio.h>
#include <math.h>                // For sqrt()

#define N   50              // Maximum number of data values.

void get_data( double x[], int n );
void stats( const double x[], int n, double * meanp, double * variancep );

int main( void )
{
    double x[N];                 // An array for the N data values.
    int num;                     // Actual number of data values.

    double mean;                 // The mean of the values in array x.
    double var;                  // Variance of the data in array x.

    double stdev;                // Standard deviation of the data in array x.
    puts( "\n Computing statistics on a set of numbers." );
    printf( "\n Enter number of values in data set (2..%i):  ", N );
    for (;;) {
        scanf( "%i", &num );
        if (num > 1 && num <= N) break;
        printf( "Error:  %i is out of legal range, try again:  \n", num );
    }
    printf( " Computing statistics on %i data values.\n", num );

    get_data ( x, num );
    stats( x, num, &mean, &var );

    stdev = sqrt(var);

    printf( "\n\n The mean of the %i data values is = %.2f \n", num, mean );
    printf( " The variance is = %.2f\n", var );
    printf( " The standard deviation is = %.2f \n", stdev );
    return 0;
}
```

Figure 11.12. Mean and standard deviation.

This graph is for the program found in Figures 11.12 and 11.14.

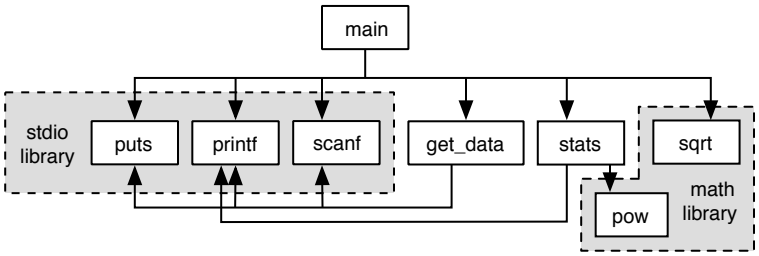


Figure 11.13. Call graph for the mean and standard deviation program.

Notes on Figure 11.12. Mean and standard deviation.

First and third boxes: the data array and the definition of N.

Every part of this program uses the value of N to define the number of data values that are to be read, processed, or output. We easily can increase or decrease the length by changing only the `#define` at the top of the program. This is one of the most important ideas in computing: By using loops and arrays, we can process a virtually unlimited number of data items. Files that store large amounts of data are the final element needed in this application. We will revisit this example in Chapter 14 to show how such data can be read from a file.

It is rare that the maximum number of data values is used, since the value of N usually is set to a comfortably large value. Therefore, the user needs to specify the size of the current data set. This value, rather than N, will serve as the processing limit in each of the array functions.

Second box: the function prototypes.

These functions are called from the main program in Figure 11.12. After reading a set of `n` experimentally determined data values into an array, `x`, we use summing loops to calculate the mean and variance of those values.

```
// ----------------------------------------------------------------
// Given an empty array of length n, input data values to fill it.  */
void get_data( double x[], int n )
{
    int k;                          // Loop counter and subscript.
    puts( "Please enter data values when prompted." );
    for (k = 0; k < n; ++k) {
        printf( "x[%i] = ", k );    // Prompt for kth value.
        scanf( "%lg", &x[k] );      // Read into array slot k.
    }
}

// ----------------------------------------------------------------
void stats( const double x[], int n, double * meanp, double * variancep )
{
    int k;                // Counter and array subscript for both loops.
    double sum;           // Accumulator for both loops.
    double divisor;       // From the definition of variance.
    double local_mean;    // Local copy of first answer.

    for (sum = k = 0; k < n; ++k) {
        printf( "\n    x[%d] = %.2f", k, x[k] );
        sum += x[k];
    }
    *mean = local_mean = sum / n;   // Store average locally, also return it.

    for (sum = k = 0; k < n; ++k) {
        sum += pow( (x[k] - local_mean), 2 );
    }
    if (n < 20) divisor = n-1;
    else divisor = n;
    *variance = sum / divisor;    // Return the variance.
}
```

Figure 11.14. The `get_data()` and `stats()` functions.

These are the prototypes for the functions in Figure 11.14. Both of these functions have an array parameter and an integer parameter that gives the size of the data set. In `stats()`, we restrict the array to input-only by writing `const` as part of the parameter type. We do this because `stats()` needs to use, but not to modify, the array values. In addition, `stats()` has two output parameters. Note that the square brackets must be used for an array in the parameter declaration, but they are omitted in the function call.

Fifth box: the function calls.

- The function calls must correspond to the prototypes. Each call must have the same number of arguments (of the same type and in the same order) as the parameters declared by the prototype. The pairs are
  - In `get_data()`, `x` is an array parameter which is passed by reference into a `double[]` parameter. Because `x` is an array, this is a a reference parameter, and therefore is in/out. However, it will be used in an output-only style, to carry the data back to `main()`.
  - In `stats()`, `x` is an array parameter which is passed by reference into a `const double[]` parameter. This is an input parameter. Because it is a reference parameter, the function will be able to read input from the array. Because of the `const`, the function will not be able to change the data in the array.
  - `num`, in both functions, the length of the array, which is passed by value into an `int` parameter. This is an input parameter; call by value with an argument of a simple type does not support outward flow of information.
  - `meanp` and `variancep` in `stats()` are used as output parameters. In the call, we write `& mean` and `& variance` to pass references into function, where they are stored in the parameters. The parameter `meanp` is type `double*`, which is appropriate for storing the address of a `double` variable.

Sixth box: Using the returned value.

After returning from `stats`, the variables `mean` and `var` contain meaningful values. We now call `sqrt(var)` to compute the standard deviation, then print the statistics.

Output.

The `main()` function prints headings, reads the number of data items, calls the four functions, and then prints the answers. Following is sample output:

```
Computing statistics on a set of numbers.

Enter number of values in data set (2..50): 5
Computing statistics on 5 data values.
Please enter data values when prompted.
x[0] = 77.89
x[1] = 76.55
x[2] = 76.32
x[3] = 79.43
x[4] = 75.12

   x[0] = 77.89
   x[1] = 76.55
   x[2] = 76.32
   x[3] = 79.43
   x[4] = 75.12

The mean of the 5 data values is = 77.06
The variance is = 2.72
The standard deviation is = 1.65
```

Notes on Figure 11.14. The `get_data()` and `stats()` functions.

The `get_data()` ***function.***

The array parameter, `x`, is an output parameter that is passed by reference, When control enters this function, the parameter array is empty. After the last data value has been read, control returns to the caller and the caller can use the values stored in the array by the function.

First box: variables for the `stats()` ***function.***

The third parameter will be used to send one answer (the average) back to the caller. But we also define a local variable for the average to make it easier and more efficient to use in later calculations.

Second box: the average.

This is the usual loop for computing the average of an array. Once all `n` values have been summed, we can calculate the average and return it to `main()`. The division that computes the mean is after the loop, rather than within it, because there is no need to perform a division on every repetition. We need not worry about division by 0 because `n` has been validated in `main()`.

The average is first stored in a local variable, then returned to the caller in the same statement by writing `*meanp = local_mean`. We keep a local copy of the average because we will use it again, repeatedly, in the next loop.

Third box: the `pow()` function.

The `pow()` function is in the `math` library. It raises a `double` value to a `double` power and returns a `double` result. In this call, the integer 2 will be coerced to type `double` before it is passed to the function. The result will be the square of the first argument.

Fourth box: returning the second answer.

Once all `n` squares have been summed, we set the divisor to `n` or `n-1`, according to the specifications, then perform the division and return the result to `main()` by assigning it to `*variancep`.

Unlike the average, we do not store the variance in a local variable after we compute it because we do not need it again in this function.

11.3.1 Summary: Returning Results from a Function

We have now demonstrated three ways to return a result from a function:

1. By using the `return` statement.
2. By using a pointer parameter.
3. By storing the results in an array parameter.

The first method is simple and it supports a total separation between the "territory" of the calling program and the actions of the function. The function need not receive an address from the caller to use `return`. This kind of separation is a highly important debugging tool and should be used wherever possible. Unfortunately, the `return` statement is limited to one result.[7]

The second method can be used for multiple results, as in the previous program, but it involves the use of the address-of operator (`&`) in the function call, the dereference operator (`*`) in the function body, and a pointer declaration in the function's prototype and header. Using these operators is somewhat awkward and can become confusing. Sometimes the required stars and ampersands are omitted accidentally. More significant, though, is that each pointer parameter supplies the function with an address in the memory area that belongs to its caller. Each such address can be a source of unintended damaging interaction between the two code units. Therefore, we prefer to pass parameters by value wherever possible. Even so, call by value/address is quite important in C and frequently must be used. [8]

Finally, returning large numbers of results of the same type is possible by using an array. It generally is convenient, efficient, and not confusing. However, the array parameter still is an address of storage that belongs to the caller. Therefore, an array parameter (like a pointer parameter) reduces the isolation of one part of the program from the other, potentially making debugging harder.[9]

11.4 What You Should Remember

11.4.1 Major Concepts

- A pointer is an address. If `p` is a pointer, then the base type of `p` determines how the compiler will interpret the data stored in the referent of `p`. For example, if a `float` pointer is dereferenced, the result is treated like a `float`.

[7] This one result however can be a complex object, such as a structure that contains multiple components. Structures will be discussed in Chapter 13.

[8] This is a defect of C. It is corrected in C++, which supports a third form of parameter passing, termed *call by reference*.

[9] Clearly, none of these three mechanisms is an ideal solution to the problem. For this reason, the reference parameter, a fourth method of returning results, was implemented in C++.

- There are two major pointer operators, `&` and `*`, which are the inverses of one another. The address operator, `&`, is used to refer to the memory location of its operand. The dereferencing operator, `*`, uses the address in the pointer operand to either retrieve or store a value at that location.
- Pointer variables, like all others, can have garbage in them if they are not initialized. Sometimes this garbage could be an accessible memory location and other times not. The value `NULL` often is used to initialize pointers. Any attempt to reference this location on a properly protected system will cause an immediate and consistent run-time error that can be tracked down more easily than the intermittent errors caused by using a random address.
- When a parameter is a pointer variable, the corresponding argument must be an array, a pointer, or an address. The called function then can store data at that address and, by doing so, change the value in a variable that belongs to the caller. We call this method of parameter passing *call by value/address*.
- When a parameter is a `const` pointer variable, the corresponding argument must be an array, a pointer, or an address. The called function then can use the data at that address but cannot change it.
- Using pointer parameters, one can return multiple results from a single function. An array parameter is translated as a pointer and lets us pass a large amount of data to or from a function efficiently.

11.4.2 Programming Style

- In this text, pointer variable declarations place the `*` next to the base type (as in `float* f`) or let it float (as in `float * f`) between the type and the name. However, it is quite common, for various historical reasons, for programmers to use the style `float *f`, in which the asterisk is attached to the variable name. We prefer the style `float* f` because it clarifies that the data type of the variable is a pointer type.
- To avoid confusion about which variables are pointers and which are not, it is best to declare only one pointer per line. This lets you maintain our preceding style convention and provides space for a comment. If you declare more than one pointer on the same line, be sure to use an `*` for each one.
- Use the correct zero literal. For pointers, use `NULL`. Reserve `0` for use as an integer.
- It is good practice to initialize pointers to `NULL`, which makes pointer usage errors easier to find.
- Do not use the operator combinations `&*` and `*&`. Since the operators cancel each other out, there is no need for either.
- In a function definition, put the parameters that bring information into the function first and the call-by-value/address parameters that carry information out of the function last. Any in-out parameters can be placed in between.
- Minimizing the use of call by value/address increases the separation between caller and subprogram, which is helpful when debugging. However, call by value/address is an important mechanism. Learn to use it wisely.
- Do not use a global variable to pass information into or out of a function. In all other cases, pass the information as a parameter. The one exception to this rule is when the global variable is used with a piece of pre-existing code that you cannot change,
- When using both the return value and pointer parameters to return results from a function, use the return value for a result that *always* is meaningful, such as a status code. Use parameters for results that may or may not be meaningful, depending on circumstances.
- If a function, `f()`, is not called by `main()` and is called by only a single other function, then the prototype for `f()` may be written inside the function that calls it. This properly limits its accessibility to the scope the programmer intended.

11.4.3 Sticky Points and Common Errors

- The most common pointer error is an attempt to use a pointer that has not been set to refer to anything. Sometimes such pointers have a `NULL` value; sometimes they contain garbage. In both cases, the attempt to use such a pointer is an error. On some systems, this causes an immediate crash. On others, execution may continue indefinitely before anything unexpected happens. Pointers are like pronouns; until they are initialized to point at specific variables, they must not be used. Be careful with your pointers and check them first when a program that uses pointers malfunctions.

- There can be confusion between the multiply and dereference operators. It should be clear from the context which is being used. If, by accident, an operand is omitted in an expression, the compiler might interpret the `*` as multiplication (when dereference was intended) without generating an error message. Using redundant parentheses can help the compiler interpret your code as you intended, but excess parentheses can clutter the expression, potentially causing other kinds of errors. Some compromise in style is needed.
- Do not reverse the use of `&` and `*`. Using the wrong operator always leads to trouble.
- When using call by value/address, a common oversight is to omit some of the required asterisks in the function or the ampersand in the call. This can produce a variety of compile-time error comments that may warn you about a type mismatch between argument and parameter but not tell you exactly what is wrong. For example, if the omission is in the parameter declaration, the error comment actually will be on the first line in the function where that parameter is used. Sometimes a beginner "corrects" the thing that was not wrong, which leads to different errors, and so on, until the code is a mess. When you have a type mismatch error, think carefully about why the error happened and fix the line that actually is wrong. Sometimes drawing a memory diagram can help clear up the confusion.
- Even if you have no type mismatches between arguments and parameters, you may not have the kind of communication you want. If you want call by value/address, you must declare things properly; otherwise, you get functions like `badswap()` in Figure 11.8. Still other times you might omit the `*` where it is needed inside the function, and the address in a pointer parameter will be changed rather than the contents of the other memory location. The compiler may not complain about this, but it certainly will affect the logic of your program. Also, forgetting the `&` in front of an argument for a pointer parameter may not generate a compiler error, but the value of the parameter during execution will be nonsense and usually cause the program to crash. Some compilers give warnings about errors like this.

11.4.4 Where to Find More Information

- Strings pointers and ragged arrays are covered in Chapter 12.
- Array processing using pointers is explained in Chapter 17.
- Pointers to functions are covered in Chapter 17.
- Pointers to dynamic storage allocation areas are found in Chapter 16.

11.4.5 New and Revisited Vocabulary

These are the most important terms, concepts, and keywords presented in this chapter.

pointer	memory address	call by reference
pointer variable	`&` of a variable	call by value/address
base type of pointer	`*` operator	address argument
`NULL` pointer	indirection	pointer parameter
uninitialized pointer	output parameter	swap function
referent	input parameter	mean value of an array
reference	in-out parameter	variance
dereference	call by value	standard deviation

11.5 Exercises

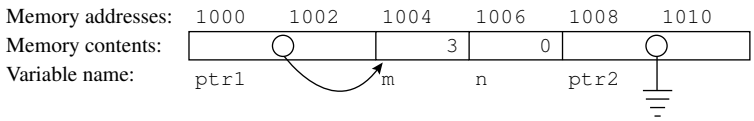
11.5.1 Self-Test Exercises

1. Complete each of the following C statements by adding an asterisk, ampersand, or subscript wherever needed to make the statement do the job described by the comment. Use these declarations:

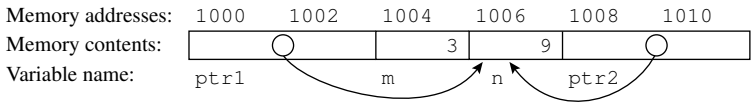
```
float x, y;
float s[4] = {62.3, 65.5, 41.2, 73.0};
float * fp;
```

(a) `fp = y;  // Make fp refer to y.`
(b) `fp = s;  // Point fp at slot containing 65.5.`
(c) `x = fp;  // Copy fp's referent into x.`
(d) `fp = y;  // Copy y's data into fp's referent.`
(e) `fp = s[];  // Copy 73.0 into fp's referent.`
(f) `scanf( "%g", x );  // Read data into x.`
(g) `scanf( "%g", fp );  // Read data into fp's referent.`
(h) `printf( "%g", s );  // Print value of second slot.`

2. (a) Given the following diagram of four variables, write code that declares and initializes them, as pictured. On this system, an `int` occupies two bytes.



(b) Now write two or three direct or indirect pointer assignment statements to change the memory values to the configuration shown here:



3. Consider the function prototype that follows. Write a short function definition that matches the prototype and the description above it. Then write a main program that declares any necessary variables, makes a meaningful call on the function, and prints the answer.

```
// Return true via out1 if in1 == in2, false otherwise.
void same( int in1, int in2, int* out1 );
```

4. All the questions that follow refer to the given partially finished program.

```
#include <stdio.h>
void freeze( int temperatures[], int n );
void show( int temperatures[], int n );

int main( void )
{
    int max = 6;
    int degrees[6] = { 34, 29, 31, 36, 37, 33 };
    freeze( degrees, max );
    printf( "\n After freeze: max = %i\n", max );
    ..........................
}
// ---------------------------------------------
void freeze( int temperatures[], int n )
{   int k;
    for(k = n-1; k >= 0; --k)
        if (temperatures[k] >= 32) --n;
}
```

(a) The second parameter of the `freeze()` function is supposed to be a call-by-value/address parameter. However, the programmer forgot to write the necessary ampersands and asterisks in the prototype, call, function header, and function code. Add these characters where needed so that changes made to the parameter `n` actually change the underlying variable `max` in the main program.

(b) Write a function named `show()`, according to the prototype given, that will display all the numbers in the array on the computer screen. Write a call on this function on the dotted line in `main()`.

(c) Draw a storage diagram similar to the one in Figure 11.9 and use it to trace execution of the call on `freeze()` and the following `printf()` statement. On your diagram, show every value that is changed.

(d) What is the output from this program after the additions?

5. (Advanced question) Trace the execution of the program that follows. Make a memory diagram following the example of Figure 11.9 and showing each program scope in a separate box. On your diagram, show the value given to each parameter when a function is called, how the values of its variables change as execution proceeds, and the value(s) returned by the function. Show the output produced on a separate part of your page.

```
#include <stdio.h>
int y = 2, z = 3;          // Global variables!

int func1( int* x, int* y );
void func2( int* x ){  *x = y;  y = z;  z = *x;  }

int main( void )
{
    int x = func1( &y, &x );
    printf( "X = %i  Y = %i  Z = %i\n", x, y, z );
}

int func1( int* x, int* y )
{
    *y = z+1;
    *x = *y;
    func2( y );
    z = *x+2;
    return *y;
}
```

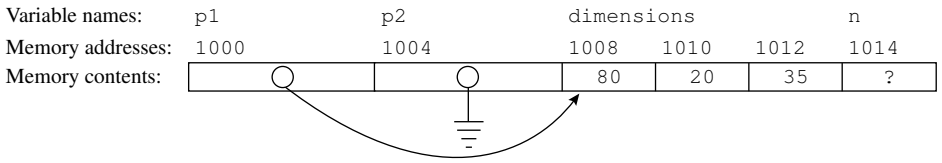
11.5.2 Using Pencil and Paper

1. Complete each of the following C statements by adding an asterisk, ampersand, or subscript wherever needed to make the statement do the job described by the comment. Use these declarations:

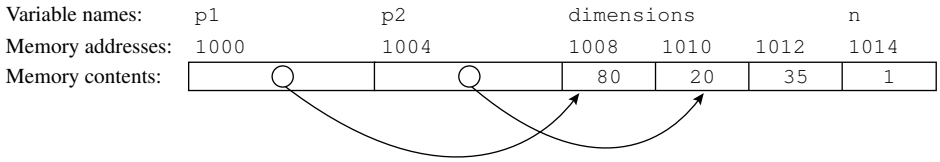
```
short s, t;
short age[] = { 30, 65, 41, 23 };
short * agep, * maxp;
```

(a) `agep = age; // Make agep refer to first age in array`
(b) `s = agep; // Copy value of agep's referent into s.`
(c) `agep = age[]; // Copy 65 into agep's referent.`
(d) `maxp = agep; // Make maxp refer to agep's referent.`
(e) `agep = (age[]+age[])/2;  // Store mean of 2nd and last ages in agep's referent.`
(f) `scanf( "%hi", age[] );  // Read into third array slot.`
(g) `scanf( "%hi", agep );  // Read into agep's referent.`
(h) `printf( "%hi", agep );  // Print agep's referent.`

2. (a) Given the diagram of three variables and an array, write code that declares the variables and initializes the array and the pointers, as pictured.



(b) Write a function that takes the `dimensions` array as its parameter. Use a nested `if...else` statement to identify the smallest dimension and return the subscript of its slot.
(c) Now call the function and store the result in `n`, then use `n` to set `p2` to point at the smallest dimension.



3. Draw a call graph for the bisection program on the text website. Include all programmer-defined and library functions.

4. Consider the function prototype that follows. Write a short function definition that matches the given prototype and description. Write a main program that declares necessary variables, makes a meaningful call on the function, and prints the answer.

```
// Set the referent of dp to its own absolute value.
// Return +1 if it was positive, 0 if it was zero,
// and -1 otherwise.
int signum( double* dp );
```

5. The questions that follow refer to the given partially finished program.

```
#include <stdio.h>
#define MAX 10
void count( int ages[], int adults, int teens );

int main( void )
{
    int family[MAX] = {44, 43, 21, 18, 15, 13, 11, 9, 7};
    int Nadults=0, Nteens=0, Nkids=0;

    puts( "\nFamily Structure" );
    count( family, Nadults, Nteens );
    Nkids = MAX - (Nadults + Nteens);
    printf( "Family has %i adults, %i teens, %i kids\n",
                Nadults, Nteens, Nkids );
    puts( "----------------------------------------\n" );
}
void count( int ages[], int adults, int teens )
{
    int k;
    for (k = 0; k < MAX; ++k) {
        if ( ages[k] >= 18 ) ++adults;
        else if ( ages[k] >= 13 ) ++teens;
    }
}
```

   (a) The second and third parameters of the `count()` function need to be pointer parameters. However, the programmer forgot to write the necessary ampersands and asterisks in the prototype, call, function header, and function code. Add these characters where needed so that changes made to the parameters `adults` and `teens` actually change the underlying variables in the main program.

   (b) Draw a storage diagram similar to the one in Figure 11.9 and use it to trace execution of the corrected program. On the diagram, show every value changed and show the output on a separate part of the page.

6. Look at the main program and functions for the bisection program on the text website. Fill in the following table, listing the symbols *defined* (not used) in each program scope. List global symbols on the first line. Allow one line below that for each function in the program (`main()` has been started for you).

Scope	Input Parameters	Output Parameters	Variables	Constants
`global`	—	—		
`main()`	—	—		
...				

11.5.3 Using the Computer

1. More stats.

   Start with the statistics program in Figures 11.12 through 11.14. Add a parallel array for the student ID numbers, which should be read as input. Add functions to do these three tasks:

   (a) Find the maximum score and return the score and its array index through pointer parameters.
   (b) Find the minimum score and return the score and its array index through pointer parameters.
   (c) Find the score that is closest to the average and return the score and its array index through pointer parameters.

   In `main()`, call your three functions and print the student ID number and score for the best, closest to average, and weakest student.

2. Class average and more.

   An instructor has a set of exam scores stored in a data file. Not only does he want a report containing the average and standard deviation for the exam, he wants lots of other statistics. These include the high score, the low score, the median score, and the coefficient of variation, *cv*. The *median score* is defined to be the middle one in the array of scores, if that array is sorted. This *coefficient of variation* relates the "error" measured by the standard deviation to the "actual" value measured by the arithmetic mean as $cv = \text{stdev}/\text{mean}$. Write a program that will read, at most, 100 exam scores from a user-specified file and print out the indicated statistics. Use portions of the statistics programs in this chapter, as appropriate.

3. Pointer and referent.

   Write a program that creates the integer array `int ara[] = {11, 13, 17, 19, 23, 29, 31}`. Also create an integer pointer `pt` and make it point at the beginning of the array. Write `printf()` statements that will print the address and contents of both `ara[0]` and `pt`. Use this format:

   ```
   printf( "address of pt: \t %p   contents:\t %p\n", &pt, pt );
   ```

   Then write 10 similar `printf()` statements following the same format to print the address and contents of the slots designated by the following expressions: `(*pt+3)`, `*pt`, `(pt[3])`, `*&pt`, `*pt[3]`, `&*pt`, `*(pt+3)`, `(*pt++)`, `*(pt++)`, `(*pt)++`.

   Some of these will cause compile-time errors when printing the address field, the contents field, or both; in such cases, delete the illegal expression and print dashes instead of its value. When the program finally compiles and runs, use the output to complete the following table, grouping together items that have the same memory address:

	Address	Contents
`pt`		
`ara`		
...		

   Finally, make four lists: (a) illegal pointer expressions, (b) expressions that have identical meanings, (c) expressions that change pointer values, and (d) expressions that change integer values. It will require careful reasoning to get the last two lists correct.

4. Exam grades.
   Start with the program in Figures 11.12 through 11.14; modify it as follows:

   (a) In the main program, declare an array to store exam scores for a class of 15 students. Print out appropriate headings and instructions for the user. Call the appropriate functions to read in the exam scores and calculate their mean and standard deviation. Print the mean and standard deviation. Then call the `grades()` function described here to assign grades to the students' scores and print them.

   (b) Modify the `average()` function so that it does not print the individual exam scores during its processing.

   (c) Write a new function, named `grades()`, with three parameters: the array of student scores, the mean, and the standard deviation. This function will go through the array of exam scores again and assign a letter grade to each student according to the following criteria. Using one line of output per student, print the array subscript, the score, and the grade in columns. The grading criteria are

      i. A, if the score is greater than or equal to the mean plus the standard deviation.
      ii. B, if the score is between the mean and the mean plus the standard deviation.
      iii. C, if the score is between the mean and the mean minus the standard deviation.
      iv. D, if the score is between the mean minus the standard deviation and the mean minus twice the standard deviation.
      v. F, if the score is less than the mean minus twice the standard deviation.

      If a score is exactly equal to one of these boundary limits, give the student the higher grade.

5. Positive and negative.

   Write a function, named `sums()`, that has two input parameters; an array, `a`, of `float`s; and an integer, `n`, which is the number of values stored in the array. Compute the sum of the positive values in the array and the sum of the negative values. Also count the number of values in each category. Return these four answers through output parameters. Write a main program that reads no more than 10 real numbers and stores them in an array. Stop reading numbers when a 0 is entered. Call the `sums()` function and print the answers it returns. Also compute and print the average values of the positive and negative sets.

6. Sorting.

   Write a `void` function, named `order()`, that has three integer parameters: a, b, and c. Compare the parameter values and arrange them in numerical order so that $a < b < c$. Use call by value/address so the calling program receives the values back in order. In addition, the function `order()` should start by printing the addresses and contents of its parameters, as well as the contents of the locations to which they point. Write a main program that enters three integers, prints their values and addresses, orders them by calling the function `order()`, and prints their values again after the call. Add a query loop to allow testing several sets of integers.

7. Compound interest.

   (a) Write a function to compute and return the amount of money, A, that you will have in n years if you invest P dollars now at annual interest rate i. Take n, i, and P as parameters. The formula is

   $$A = P(1+i)^n$$

   (b) Write a function to compute and return the amount of money, P, that you would need to invest now at annual interest rate i in order to have A dollars in n years. Take n, i, and A as parameters. The formula is:

   $$P = \frac{A}{(1+i)^n}$$

   (c) Write a function that will read and validate the inputs for this program. Using call by value/address, return an enumerated constant for the choice of formulas, a number of years, an interest rate, and an amount of money, in dollars. All three numbers must be greater than 0.0.

   (d) Write a `main` program that will call the input routine to gather the data. Then, depending on the user's choice, it should call the appropriate calculation function and print the results of the calculation.

8. Sorting boxes.
   A set of boxes are on the floor. We want to put them in two piles, those larger than the average box and those smaller than or equal to the average box. Write a program to label the boxes in the following manner:

   (a) Rewrite the `get_data()` function in Figure 11.14 so that you can enter data into three arrays rather than one. These arrays should hold the length, width, and height of the boxes, respectively.

   (b) Write a function, named `volume()`, to compute and store the volume of each box in a fourth parallel array. The volume of a box is the product of its length, width, and height.

   (c) Simplify the `stats()` function in Figure 11.14 so that it computes only the average, not the variance. Rename it `average()`. Then write a function, `print_boxes()`, that will print the data for each box in a nice table, including a column containing the appropriate label, `big` or `small`, depending on whether the volume of the box is larger or smaller than the average volume.