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

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 $2^4 = 16$.

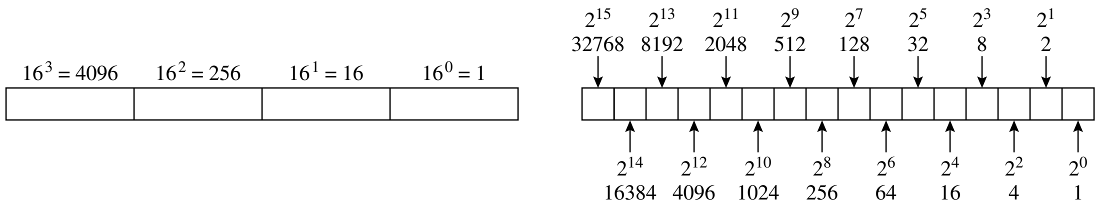


Figure 7.1. Place values.

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.

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 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

[1]The interested reader can refer to Appendix E for algorithms for converting numbers from one base to another.Figure E.2 shows the decimal values of the 16 hexadecimal digits; Figure E.4 shows some equivalent values in decimal and hexadecimal.

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.

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.

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.
- 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

[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.

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.

[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.

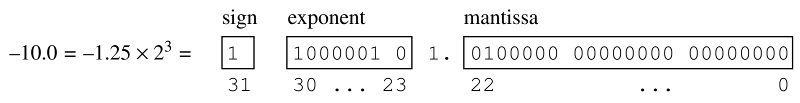
7.3.1 Representation of Real Numbers

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 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`.



- 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.

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

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.

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 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`,.

[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
`.021`	`long double`	8, 10, or 12 bytes
`171.L`	`long double`	8, 10, or 12 bytes

Figure 7.8. Floating-point literal examples

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 7.29.

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

[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.

```
#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.

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 7.29.

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.

```
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
```

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

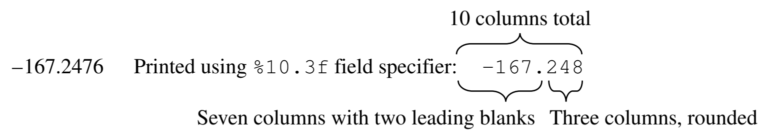

10 columns total
–167.2476
Printed using %10.3f field specifier:
  -167.248
Seven columns with two leading blanks
Three columns, rounded

Figure 7.11. The `%f` output conversion.

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 7.12 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]

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

[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 7.12. Basic floating-point conversion specifications.

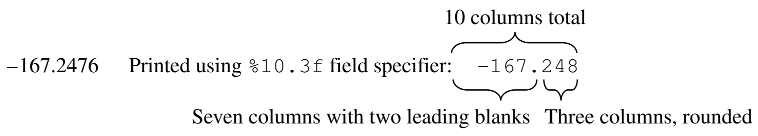

10 columns total
–167.2476
Printed using %10.3f field specifier:
  -167.248
Seven columns with two leading blanks
Three columns, rounded

Figure 7.13. The `%f` output conversion.

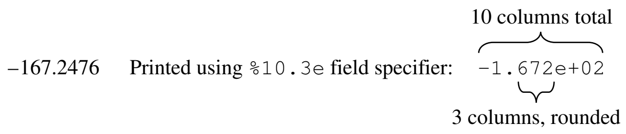


Figure 7.14. The `%e` output conversion.

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 7.13.

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 7.14). 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,

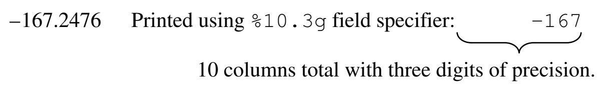

−167.2476 Printed using %10.3g field specifier: -167
10 columns total with three digits of precision.

Figure 7.15. Sometimes %g output looks like an integer.

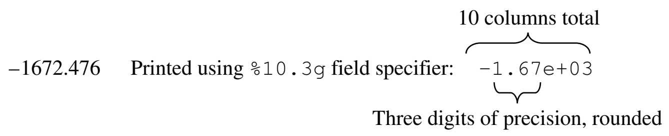

10 columns total
−1672.476 Printed using %10.3g field specifier: -1.67e+03
Three digits of precision, rounded

Figure 7.16. Sometimes %g looks like %e.

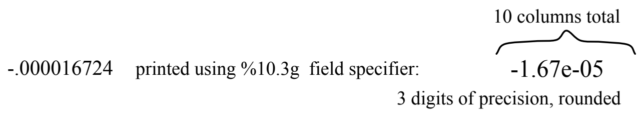

10 columns total
-.000016724 printed using %10.3g field specifier: -1.67e-05
3 digits of precision, rounded

Figure 7.17. For tiny numbers, %g looks like %e.

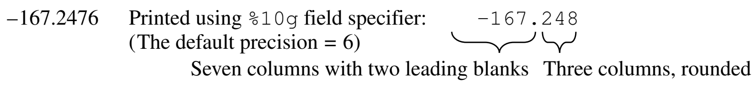

−167.2476 Printed using %10g field specifier: -167.248
(The default precision = 6)
Seven columns with two leading blanks
Three columns, rounded

Figure 7.18. Sometimes %g looks like %f.

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 7.15).
- 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 7.16).
- 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 7.17).
- 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 7.18).

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 7.19 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 7.19. 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 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

Input at keyboard	`%g`	32.1786594	2.3	12345678
Internal bit value		32.17865753173828125	2.2999999523162841796875	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 7.19. Output conversion specifiers.

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 7.22 and 7.29 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.

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 7.20. 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 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.

```
#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 7.20. Converting a `float` to a `double`.

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.

[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 7.21. Kinds of casts.

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 7.21.

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 7.22 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.

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

    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 7.22. Rounding and truncation.

Notes on Figure 7.22. Rounding and truncation.

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.

This program demonstrates the effects of some type coercions. Unlike the program in Figure 7.22, 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 back to float. */
    w = sin( x );   /* C. Calling sin() coerces x to type double and */
                    /*    = 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 7.23. Type coercion.

The output.

```
Casting:    y=  17.70  k=  17
Assignment: y=  17.70  m=  17
Rounding:   y=  17.70  m=  18

Re-casting: 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 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 7.23, 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 7.23, 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 7.22 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 7.23, 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.

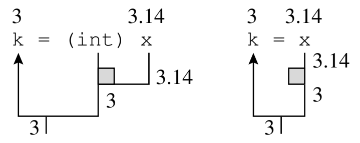
7.5.3 Diagramming Conversions

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 7.24 shows an example of each and Figure 7.25 diagrams casts and coercions within larger expressions.

Notes on Figure 7.24. Diagramming a cast and a coercion.

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 7.24. Diagramming a cast and a coercion.

The examples use these declarations:

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

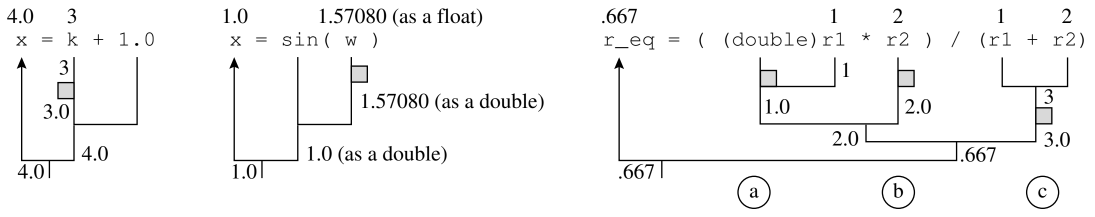


Figure 7.25. Expressions with casts and coercions.

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.

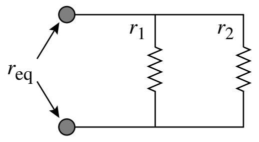
Diagram on right: a coercion. When a real number is stored in an integer variable, it must be coerced first. We represent the coercion by a black box on a branch of the parse tree. Even though no cast is written here, the real value 3.14 above the box is converted at the box to become the integer 3 below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

Notes on Figure 7.25. 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.

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 7.26. Problem specification: computing resistance.

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 7.27 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 7.26 and 7.27 illustrate an application in which the use of a cast operation on `int` values achieves the necessary precision in the answer.

A division application: Computing resistance. Figure 7.26 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 7.27, 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.

We show how to use a type cast or coercion to solve the problem specified in Figure 7.26.

```
#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 7.27. Computing resistance.

Notes on Figure 7.27. 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 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.
- 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.

Output. Here is some sample output:

```
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
```

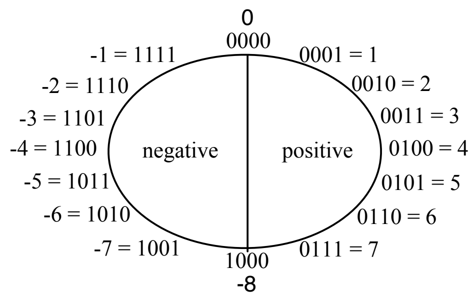


Figure 7.28. Overflow and wrap with a four bit signed integer.

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.

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 7.28. 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

[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.

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 7.29.

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 7.29 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
```

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.

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 7.29. Computing *N*!.

```
         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 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 2.585201673888497821286e+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  8.683317618811885938716e+36
     34     2.95232822997e+38  2.952327990396041195551e+38
     35             +Infinity  1.033314796638614422221e+40
     36             +Infinity  3.719933267899011774924e+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

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 7.30. Floating-point underflow.

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 7.30:

```
 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.

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 7.30, 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. 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:

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

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 7.31, 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
```

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.  */
int k;                   /* Loop counter. */
for (k=11; k>0; --k){
        quotient = (double)(104-k+1) / k;
        combinations *= quotient;
}
```

Figure 7.31. Calculation order matters.

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.

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 7.32. 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 7.33.

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 7.35.

With the epsilon shown (2 mph), we say that $\text{speed}_1 = 49.0$ equals 50.0 because it is within epsilon of 50.0, but $\text{speed}_2 = 54.0$ does not equal 50.0 with this value of epsilon.

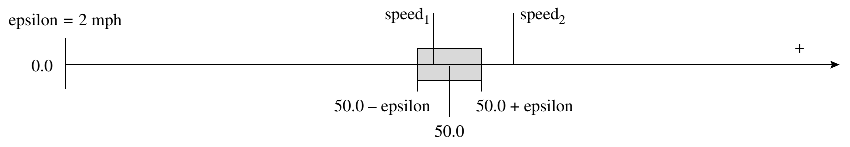


Figure 7.32. An approximate comparison.

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 7.33. Comparing floats for equality.

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 7.34 and written in Figure 7.35 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.

Notes on Figure 7.35. 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.

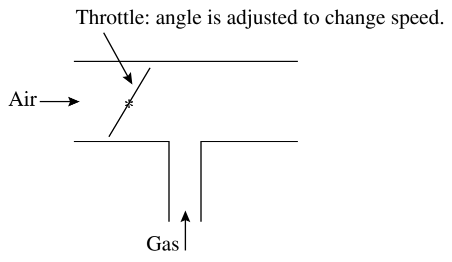
Second box: waiting for the "set" signal.

- 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

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 7.34. Problem specification: Cruise control.

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.

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 7.35. Cruise control.

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.

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.

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`. However, two other important issues must be considered in choosing a data type: memory limitations and speed of execution. 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.
- *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 $0 \ldots 38$ for type `float` or $0 \ldots 308$ 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.

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.

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.
- 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 7.36 gives a brief summary of the difficulties that might be encountered when using `C` casts and conversions.

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.

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 7.36. Casts and conversions in C.

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.

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?

  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[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:

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

- 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 discussed in this chapter:

representation range	truncation	integer overflow
integer type specifier	rounding	wrap
integer literal	safe conversions	IEEE floating-point standard
literal modifiers	unsafe conversions	precision
integer division	length conversion	floating-point overflow
indeterminate results	2- and 4-byte models	underflow
division by 0	type cast	representational error
floating-point types	type coercion	normalized and denormalized
mantissa	I/O conversion specifier	order of magnitude
exponent	field width specifier	order of performing operations
scientific notation	precision specifier	approximate comparison
floating-point literal	default output precision	epsilon test
floating-point type specifier	representational error	correct but not precise
representation conversion		factorial

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

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

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 21 (linked lists), and Chapter 22 (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 precisional 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 precisional 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 7.31 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.