

1-2 Integers and floating numbers

Most computers make a clear distinction between integers and floating numbers. An integer is a number, such as 5, -213 , and 0, without a part that must be represented by a decimal point. A floating or *real* number is any other type of number, such as $3.1415926\dots$, 3.0×10^{23} , and -9.9 , that requires a decimal point to specify its value. The reason for differentiating between these two types of numbers comes from the structure of the computer memory.

The basic unit for storing a number in a computer is a *bit*, the state of an electronic component that is either on or off, as we saw earlier. The two possible states of a bit may be used to represent two numerical values 0 (off) and 1 (on). Since a single bit is too small for most interests, 8 bits are grouped together into a *byte*. The status of a byte may be represented by an eight-digit binary number $b \square \square \square \square \square \square \square \square$, where we have added a prefix b in front of the number to indicate that it is in the binary representation. For example, the integer 5 is shown as $b00000101$ or simply as $b101$. The largest integer that can be represented in one byte of storage is then $b11111111 = 2^8 - 1 = 255$.

Before we leave the subject of internal representation of integers in the computer memory, we shall define two other representations. The binary representation used above is inconvenient in many cases because of the large number of digits required to express most of the integers of interest to us. The hexadecimal representation is based on powers of 16 (in an analogous way as the decimal system is based on powers of 10). Each hexadecimal digit can take on values 0 through 15, and they are usually written as $z0, z1, z2, z3, z4, z5, z6, z7, z8, z9, zA, zB, zC, zD, zE$, and zF , where we have added a prefix z to indicate that they are given in hexadecimal representation. In terms of computer memory, each hexadecimal digit represents one of the possible values stored in four binary digits ($2^4 = 16$). The value that can be stored in a byte is then represented by two hexadecimal digits. Examples of numbers in the hexadecimal representation and their corresponding values in decimal and binary representations are given in Table 1-1.

Another way of displaying binary-based numbers is the octal representation. To distinguish it from others, we shall prefix a number in the octal representation with the letter o . (The lowercase o is used here instead of the uppercase so as to make it easy to differentiate it from the number zero.) Each octal digit represents the value given by 3 bits. This is convenient for some models of computer whose memories are made of multiples of 3 bits, such as 36 and 60. In addition, octal representation is also a convenient way for carrying out many types of manipulations.

In many calculations, a byte, which can only store integers up to 255, is still too small. For this reason, each integer is often assigned either two or four bytes of memory. Such a grouping of bytes is sometimes called a computer word, or just *word* for short. Before we go into the question of the range of integer values a computer word can store, we must recall that most numbers we are interested in have a \pm sign associated with them. It is common practice to designate the first bit of an integer

Table 1-1: Decimal, hexadecimal, octal, and binary representations of numbers.

Decimal	Hexa- decimal	Octal	Binary	Decimal	Hexa- decimal	Octal	Binary
0	z0	o0	b0	10	zA	o12	b1010
1	z1	o1	b1	11	zB	o13	b1011
2	z2	o2	b10	12	zC	o14	b1100
3	z3	o3	b11	13	zD	o15	b1101
4	z4	o4	b100	14	zE	o16	b1110
5	z5	o5	b101	15	zF	o17	b1111
6	z6	o6	b110	16	z10	o20	b1 0000
7	z7	o7	b111	17	z11	o21	b1 0001
8	z8	o10	b1000	18	z12	o22	b1 0010
9	z9	o11	b1001	19	z13	o23	b1 0011
255	zFF	o377	b1111 1111				
256	z100	o400	b1 0000 0000				
257	z101	o401	b1 0000 0001				

word as the sign bit. Thus, for a two-byte integer I_2 , only 15 bits are available to store the magnitude of the number. The possible values that can be represented by such a word is then

$$-32,768 \leq I_2 \leq +32,767.$$

That is, -2^{15} through $(2^{15} - 1)$. The reason that the maximum positive integer value is one less than 2^{15} comes from the fact that +0 must also be considered as one of the integers. (Why, then, is the maximum absolute value of a negative integer one larger?) For a four-byte word, the possible value of an integer is then

$$-2,147,483,648 \leq I_4 \leq +2,147,483,647$$

corresponding to -2^{31} to $(2^{31} - 1)$. Although the allowed range of integer values for I_4 may be large, it is still not adequate for many purposes. For example, the upper limit of I_4 is less than $13! = 6,227,020,800$. As a result, it may not be possible to carry out certain types of calculations that involve factorials. We shall see later ways to circumvent this difficulty.

For most calculations in science and engineering, the use of integers alone is too restrictive. Floating numbers broaden the range of values that can be kept in the computer memory by allocating a part of a word to store the exponent of each number. That is, each number now has a sign, a fraction part or mantissa, and an exponent. Since the computer memory is made of bits, some arrangements must be made to represent a number in this way. The usual case is to use four bytes for a single-precision number. Among the 32 bits in such a word, 1 bit is devoted to the sign, 8 bits are assigned to the exponent, and the remaining 23 bits are left for the mantissa. In this way, numbers with absolute values in the range from approximately

1.2×10^{-38} to 3.4×10^{38} can be represented. In double precision, eight bytes are usually used for each floating number. Among the 64 bits here, 1 bit is for the sign, 11 bits for the exponent, and 52 bits for the mantissa. Floating numbers with absolute values approximately in the range from 2.2×10^{-308} to 1.8×10^{308} may be represented by such an arrangement.

If we get a floating number whose absolute value is too small to be represented in a computer, an *underflow* condition is created. This happens, for example, when we get a single precision floating number with absolute value less than 10^{-38} . The usual course of action on such occasions is to replace the number by zero. If a number is produced with an absolute value much larger than 10^{38} in single precision or 10^{308} in double precision, an *overflow* condition is created. In this case, the usual response of the computer is to suspend the calculation, as an error will be introduced if we replace the number by anything else. Generally, one should check for possible underflows and, in particular, overflows in a program where such conditions are likely to occur and take the appropriate response.

The maximum number of significant figures that can be achieved in a floating number calculation is limited, in the first place, by the number of bits assigned to the mantissa. In single precision, the maximum is roughly seven significant figures and in double precision it is possible to reach 15 significant figures. Since the maximum number of significant figures is limited, *truncation* errors become a part of any floating number calculations. One important consideration in numerical work is to choose an algorithm that minimizes the truncation errors. If this is not done, the numerical errors may accumulate and become so large that no significant figures are left in the final results. At the same time, there is no use in trying to design an algorithm with a precision beyond the limitations imposed by truncation errors. We shall see different aspects of these considerations in many of the examples following.

1-3 Programming language and program library

In addition to hardware, such as central processor, memory and disk, software is also an integral part of any computer. As users, we do not wish to be overly involved in all the aspects that make the machine work for us; however, some background knowledge is useful. In getting a computer to solve our problem, the pieces of the software of most direct concern to us are the programming language and the subroutines to carry out some of the tasks to solve our problem.

High-level programming language For a computer to carry out a task, it is necessary to give it a set of specific instructions. This is generally referred as programming the computer. From a user point of view, it is desirable to give these instructions in terms of a language that is close to how we think of the problem, for example, in terms of equations. On the other hand, since the computer hardware is designed to carry out a very limited number of basic operations, we are still a long way from the possibility of such a direct communication. Furthermore, equations, by themselves,