Data Types and Representations

Since we will be performing numerical calculations using the C compiler, it is useful to know how the different data types are stored. Because we have 10 fingers and 10 toes, we have been brought up expressing integers and real numbers in base 10. However, the computer doesn't have fingers or toes. The easiest method of arithmetic operations for computers is in binary, base 2. We will represent integers and real numbers in the following notation: the number in parenthesis with the base as a subscript: $(number)_{base}$ or $(yyy.xxxx)_{base}$.

The main data types that we will be using will be integers, long integers, float, and double precision.

char types

Computers do numerical operations and store data in base 2. The smallest data type is a byte, which is 8 bits. Each bit can be a 0 or 1. The simplest data type is an **unsigned char**. The unsigned char type is one byte long, and the number it represents are just the 8 bits in base 2. For example, $(00000000)_2$ is zero, and $(11111111)_2$ in base 2 is $(255)_{10}$ in base 10. So the unsigned char is an integer between 0 and $(255)_{10}$.

The next simplest type is the **char**. The first bit represents the sign of the integer, with 0 being +, and 1 being -. The next 7 bits are the integer in base 2. The smallest integer is 11111111 which is $(-127_{10}$ in base 10. The largest integer is 01111111, which is $(+127)_{10}$ in base 10. So the char type is an integer between $(-127)_{10}$ and $(+127)_{10}$.

int type

The **int** type is stored in four bytes, or 32 bits. The first bit represents the sign of the integer, with 0 being + and 1 being -. The next 31 bits are the integer in base 2. The largest integer is $011 \cdots 11$ which is $+2^{31} - 1 = +(2147483647)_{10}$. The most negative integer is $-(2147483648)_{10}$. So the int type between $-(2147483648)_{10}$ and $+(2147483647)_{10}$.

The **long int** type also stores integers in four bytes, or 32 bits. Storage is similar to the int type with the first bit representing the sign. The most negative integer is therefore $111\cdots 11$ which is $-(2^{31}-1) = -(2147483647)_{10}$. The most positive integer is $0111\cdots 111 = +(2147483647)_{10}$. So the long int type is an integer between $-(2147483648)_{10}$ and $+(2147483647)_{10}$.

float type

We are used to expessing real numbers in decimal notation: $(0.1)_{10} = 1/10 = 10^{-1}$, $(0.01)_{10} = 1/100 = 10^{-2}$, etc. We are not as familiar expressing real numbers in base 2. Using the same idea as in base 10, $(0.1)_2 = 1/2 = 2^{-1}$, $(0.01)_2 = 1/4 = 2^{-2}$. In general, real numbers can be expressed in base 2 notation as:

$$(x_2 x_1 x_0 . x_{-1} x_{-2} \cdots)_2 = x_2 2^2 + x_1 2^1 + x_0 + x_{-1} 2^{-1} + x_{-2} 2^{-2} \cdots \tag{1}$$

Now we are ready to understand how the computer will store real numbers.

The **float** data type uses 4 bytes, or 32 bits, to store a real number. The convention that has been chosen is to have the first bit represent the sign of the number, the next 8 bits represent the exponent in base 2, and the last 23 bits the mantisa of the number in base 2. More specifically:

$$s c_1 c_2 \cdots c_8 f_1 f_2 \cdots f_{23} \rightarrow (-1)^s (1.f)_2 \, 2^{c-127} \tag{2}$$

where s, the c_i, and the f_i correspond to the "zero" or "one" value of the different 32 bits that make up the four bytes.

Using this convention, we see that the largest value for the float type is when $c = (11111111)_2 = (255)_{10}$. This value for c yields an exponent of $2^{255-127} = 2^{128} \approx 10^{38}$. So the largest real number that can be expressed in the float type is around 10^{38}. Similarly, the most negative real number that can be expressed is around -10^{38}. Between these two limits, there are only $2^{31} - 1 \approx 2 \times 10^9$ possible float numbers.

The smallest non-zero number is also determined by the exponent c. If $c = 0$, then the exponent is $2^{-127} \approx 6 \times 10^{-39}$.

Although the smalest float type number is 10^{-39}, this doesn't mean that we have precision to 39 significant figures. The number of significant figures is determined by the 23 bits in the mantissa (f) region. The number of digits after the decimal point is $2^{-23} \approx 10^{-7}$. Thus, the float type can have a precision of 7 decimal digits.

Seven significant figures (base 10) suffices for most physics applications, but there are cases when more significant figures will be needed. The **double** type stores real numbers with more bits than the float type. The **double** type uses 8 bytes, or 64 bits to store the number. The convention for storing double types is to have the first bit represent the sign, the next 11 bits represent the exponent, and the last 52 bits represent the mantissa. Everything is in base 2 as follows:

$$sc_1c_2 \cdots c_{11}f_1f_2 \cdots f_{52} \rightarrow (-1)^s(1.f)_2\, 2^{c-1023} \tag{3}$$

where s, the c_i, and the f_i correspond to the "zero" or "one" value of the different 64 bits that make up the eight bytes as was the case in the float type.

In the case of the double type, the largest number is $2^{1024} \approx 3.6 \times 10^{308}$. The precision in the double type is $2^{-52} \approx 2.210^{-16}$, which gives around 16 significant figures in base 10. One rarely needs more accuracy than this.

Final Notes

We are so used to using base 10 for real numbers that we forget its special features. Using a binary representation, one can only express real numbers exactly if the fraction can be written as a sum of inverse powers of 2. For example one tenth in base 10 is exactly written as $(0, 1)_{10}$. However in base 2 one tenth cannot be expressed exactly using a finite number of bits:

$$\begin{aligned} (0.1)_{10} &= (0.00011001100110011\cdots)_2 \\ &= (-1)^0(1.1001100110011\cdots)_2\, 2^{-4} \end{aligned}$$

If we truncate the series on the right side then we only get an approximation of 1/10. In float type, the 32 bits which would be used to represent 1/10 would be as follows. $s = 0$ since the number is positive. $c - 127 = -4$, or $c = (123)_{10} = (01111011)_2$. Finally, $f = (10011001100110011001101)_2$. Note the last "1" is due to rounding up since "1" would be the next bit. The actual 32 bits which would represent the best approximation in float for 1/10 are therefore:

$$00111101110011001100110011001101 \tag{4}$$

Note that the number representated by these bits for float is not exactly equal to 1/10:

$$\begin{aligned}(0.000110011001100110011001101)_2 &= \frac{1}{2^4}+\frac{1}{2^5}+\frac{1}{2^8}+\frac{1}{2^9}+\cdots+\frac{1}{2^{24}}+\frac{1}{2^{25}}+\frac{1}{2^{27}} \\ &\approx (1.000000015)_{10}\end{aligned}$$

You shouldn't get upset that the computer can't store 1/10 exactly. It doesn't have 10 fingers and it only has a finite amount of storage space for its zero's and one's.

Base 2 is probably a more universal base to use. If we were to communicate with intelligent life on other planets, most likely they will not be using base 10. Although we have 10 fingers and toes, we have two eyes, two ears, two hands, two arms, two feet, two legs, etc. Actually, base 8 would have been better than base 10. If we had chosen base 8, then in High School 7/8 = 87.5% would have been an "A" grade, and we would have a special birthday every 8 years.