Some more on Data Storage in Computers

(Adapted from the C Programming Notes by Michael Ashley)

Some Basic Definitions

Bits, bytes, and words

Bit

A bit is the smallest unit of information. It can be thought of as the answer to a yes/no question. A bit has the value 1 or 0 (which can be equated with yes/no, true/false).

Byte

A byte consists of eight bits, ordered from the most significant bit (MSB) on the left, to the least significant bit (LSB) on the right. E.g., 011011000 is a byte. A byte can be interpreted as a base-2 number. There are 256 (i.e., 2 to the power 8) possible combinations of 8 bits, ranging from 00000000 to 11111111. These 256 combinations could, in principle, be used to represent any 256 things (e.g., we could define 00000000 to be "apple" and 00000001 to be 3.14159265359). In practice there are three commonly used interpretations of a byte:

Unsigned byte

An unsigned byte can represent integers from zero to 255.

Signed byte

A byte can also be used to represent integers from -128 to +127. Such a byte is called a signed byte, and the MSB is used to indicate the sign (0 for positive, 1 for negative), -128 is represented as 10000000, -1 is 11111111, 0 is 0, and +127 is 01111111. Non-examinable: Technical Stuff (To find the bit pattern for a negative number, you first write down the representation of the number without the minus sign, then invert all the bits, then add one. e.g., -1 is 00000001 inverted (which is 11111110), plus one (which gives 11111111). This representation is called "2's complement", and is used because (1) it makes binary arithmetic easier to implement in the computer hardware, and (2) it avoids the waste of a bit pattern in the case of 10000000, which you might think was -0.)

A byte as an ASCII character

The American Standard Code for Information Interchange (ASCII) associates the various possible bit patterns in a byte with characters (i.e., things like 'A', 'B', 'C'...'Z', 'a', 'b', 'c',...'z', '*', '&', '@', etc). For example, 'A' is stored as 65 (decimal). The ASCII table allows you to translate a simple text document into a list
of bytes (which you might then store in a file, see below). For a complete list of the ASCII table, type "man ascii".

**Words**

In order to represent numbers outside the range of an unsigned or signed byte, multiple bytes can be grouped together to form words of various lengths. Words usually contain a power of two bytes, e.g., 2, 4, 8, 16, and so on. They may either be signed or unsigned. For example, an unsigned 2-byte word can represent numbers from 0 to 65536 (2 to the power 16), a signed 2-byte word ranges from -32768 to +32767. Words are sometimes called "short words", "words", and "long words", for the 2, 4, and 8-byte variations, but this usage is not uniform.

**32-bit computer, 64-bit computers**

Most PCs are 32-bit computers, i.e., they process 32-bits (4 bytes) of data in one operation. In a few years, 64-bit computer will become very common.

**Binary, octal, decimal, hexadecimal**

Integers can be written in various bases. The common bases used with computers are binary (base-2), octal (base-8), decimal (base-10), and hexadecimal (base-16). Octal and hexadecimal are often useful when doing low-level programming, since they encode a binary number in a smaller amount of typing than using binary notation.

The following table compares the various numbering systems (note that you can’t directly specify a binary constant in the C programming language); octal constants are distinguished by a leading zero; hexadecimal constants by a leading "0x"):

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>0x0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>001</td>
<td>0x1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>002</td>
<td>0x2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>003</td>
<td>0x3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>004</td>
<td>0x4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>005</td>
<td>0x5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>006</td>
<td>0x6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>007</td>
<td>0x7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>010</td>
<td>0x8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>011</td>
<td>0x9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>012</td>
<td>0xa</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>013</td>
<td>0xb</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>014</td>
<td>0xc</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>015</td>
<td>0xd</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>016</td>
<td>0xe</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>017</td>
<td>0xf</td>
</tr>
</tbody>
</table>

**Floating-point numbers**

Scientific computation often requires the use of floating-point numbers, e.g., 1.23. There is an IEEE standard that describes how floating-point numbers can be represented as a group of bytes. The basic idea is to allocate a certain number of
bits to the exponent (base-2), and the rest to the mantissa (which has been normalised to be between 0.5 and 1). Both the exponent and mantissa have sign bits.

Floating-point numbers typically come in 4, 8, or 16 byte sizes. In C these are usually called "float", "double", and "long double" respectively.

For any given number of bytes allocated to store a floating-point number, there is clearly a tradeoff between the range of the exponent, and the precision of the mantissa: the more bits you use for the exponent, the fewer are left over for the mantissa. We will see later how this compromise has been chosen by the IEEE for the various byte sizes.

Note that there are some redundant bit patterns in the representation of floating-point numbers: e.g., 0.0 with any exponent is still zero. One of these bit patterns is used as a flag to indicate that a calculation produced an illegal result. This bit pattern is called "NaN" which stands for "not a number". E.g., if you take the square root of a negative number, you will receive the result NaN. Any operation on a NaN will continue to produce a NaN (e.g., if you subtract two NaNs you will still have a NaN), this is useful to propagate such error conditions in your programs so that you are aware when they affect the final result of a calculation.

Character constants, escape sequences, and string constants

A character constant in C is a single character (or an escape sequence such as \n) enclosed in single quotes, e.g., 'A'. It occupies one byte of storage.

The value of a character constant is the numeric value of the character in the computer's character set (e.g., 'A' has the value 65). In 99.99% of cases this is the ASCII character set, but this is not defined by the C standard!

But how do you represent a character such as a single quote itself, or one that doesn't print (such as the ASCII BEL character)? The answer is to use an escape sequence.

For reference, here is a program to print out all the special escape sequences. Have a look at it now.

In addition, you can specify any 8-bit ASCII character using either \ooo or \xhh where `ooo` is an octal number (with from 1 to 3 digits), and 'xhh' is a hexadecimal number (with 1 or 2 digits). For example, \x20 is the ASCII character for SPACE, so ' ' and '\x20' are identical.

String constants are a sequence of zero or more characters, enclosed in double quotes. For example, "test", "", "this is an invalid string" are all valid strings (you can't always believe what a string tells you!). String constants are stored in the computer's memory as a sequence of numbers (usually from the ASCII character set), and are terminated by a null byte (\0). So, "test" would appear in memory as the numbers 116, 110, 115, 116, 0.

A consequence of the null byte terminator is that a C character string can never contain a null byte. In practice, this isn't often a limitation.

A program to give information on C data types
Like other programming languages, C has a variety of different data types. The ANSI standard defines the data types that must be supported by a compiler, but it doesn't tell you details such as the range of numbers that each type can represent, or the number of bytes of storage occupied by each type. These details are implementation dependent, and defined in the two system include-files "limits.h" and "float.h". To find out what the limits are, try running the following program with your favourite C compiler.

```c
/* A program to print out various machine-dependent constants */
/* Michael Ashley / UNSW / 03-Mar-2003                        */

#include <stdio.h>      /* for printf definition       */
#include <limits.h>     /* for CHAR_MIN, CHAR_MAX, etc */
#include <float.h>      /* for FLT_DIG, DBL_DIG, etc   */

int main(void) {
    printf("char           %d bytes %d to %d \n",   sizeof(char          ), CHAR_MIN, CHAR_MAX  );
    printf("unsigned char  %d bytes %d to %d \n",   sizeof(unsigned char ), 0       , UCHAR_MAX );
    printf("short          %d bytes %hi to %hi \n", sizeof(short         ), SHRT_MIN, SHRT_MAX  );
    printf("unsigned short %d bytes %hu to %hu \n", sizeof(unsigned short), 0       , USHRT_MAX );
    printf("int            %d bytes %i to %i \n",   sizeof(int           ), INT_MIN , INT_MAX   );
    printf("unsigned int   %d bytes %u to %u \n",   sizeof(unsigned int  ), 0       , UINT_MAX  );
    printf("long           %d bytes %li to %li \n", sizeof(long          ), LONG_MIN, LONG_MAX  );
    printf("unsigned long  %d bytes %lu to %lu \n", sizeof(unsigned long ), 0       , ULONG_MAX );
    printf("float          %d bytes %e to %e \n",   sizeof(float         ), FLT_MIN , FLT_MAX   );
    printf("double        %d bytes %e to %e \n",   sizeof(double       ), DBL_MIN , DBL_MAX   );
    printf("precision of float %d digits\n", FLT_DIG);
    printf("precision of double %d digits\n", DBL_DIG);
    return 0;
}
```

**Notes:**

- `sizeof` looks like a function, but it is actually a built-in C *operator* (i.e., just like `+,-,*,`). The compiler replaces `sizeof(data-type-name)` (or, in fact, `sizeof(variable)` with the number of bytes of storage allocated to the data-type or variable.
- The 'unsigned' data types are useful when you are referring to things which are naturally positive, such as the number of bytes in a file. They also give you a factor of two increase in the largest number that you can represent in a given amount of space.
- An "unsigned char" is a single-byte C data type that can represent positive integers from 0 to 255. A "signed char" uses the most-significant bit of the byte as a sign-bit (0 indicates a positive number, 1 indicates a negative number), leaving the remaining 7 bits to encode the absolute value of the number; the range of signed chars is therefore -128 to +127.

Most of the time you don't need to worry about how many bytes are in each data type, since the limits are usually OK for normal programs. However, a common problem is that the "int" type can be either 2-bytes or 4-bytes in length, depending on the compiler.

**Here is the output of the preceding program when run on a GNU/Linux PC, using gcc:**

```
char                    1 bytes -128 to 127
```
unsigned char   1 bytes 0 to 255  
short            2 bytes -32768 to 32767  
unsigned short  2 bytes 0 to 65535  
int              4 bytes -2147483648 to 2147483647  
unsigned int    4 bytes 0 to 4294967295  
long             4 bytes -2147483648 to 2147483647  
unsigned long   4 bytes 0 to 4294967295  
float           4 bytes 1.175494e-38 to 3.402823e+38  
double          8 bytes 2.225074e-308 to 1.797693e+308  
precision of float   6 digits  
precision of double 15 digits  

Note:

- Integers are automatically assumed to be of the smallest type that can represent them (but at least an "int"). For example, 2147483648 was assumed by the compiler to be an "unsigned int" since this number is too big for an "int". Numbers too big to be an "unsigned int" are promoted to "long" or "unsigned long" as appropriate.
- An integer starting with a zero is assumed to be octal, unless the character after the zero is an "x" or "X", in which case the number in hexadecimal.

It pays to be very careful when specifying numbers, to make sure that you do it correctly, particularly when dealing with issues of precision. For example, have a look at this program (precision1.c, precision2.c):

```c
#include <stdio.h>

int main(void) {
    double r;
    r = 1.0 + 0.2F;
    r = r - 1.2F;
    printf("%22.16e\n", r);
    return 0;
}
```

The program gives at output of "-4.4703483581542969e-08", not zero as you might expect.

---

**Conversion between integers and floating point numbers**

In C, as in all computer languages, there are rules that the compiler uses when a program mixes integers and floating point numbers in the same expression.

Let's look at what happens if you assign a floating point number to an integer variable (convert1.c):

```c
#include <stdio.h>

int main(void) {
    int i, j;
    i =  1.99;
    j = -1.99;
    printf("i = %d; j = %d\n", i, j);
    return 0;
}
```
This program produces the result "i = 1; j = -1". Note that the floating point numbers have been truncated and not rounded.

When converting integers to floating-point, be aware that a "float" has fewer digits of precision than an "int", even though they both use 4 bytes of storage (on normal PCs). This can result in some strange behaviour, e.g. (convert2.c),

```c
#include <stdio.h>
int main(void) {
  unsigned int i;
  float f;
  i = 4294967295; /* the largest unsigned int */
  f = i; /* convert it to a float */
  printf("%u %20.13e %20.13e\n", i, f, f - i);
  return 0;
}
```

This program produces the following output when compiled with "gcc":

```
4294967295 4.2949672960000e+09 1.0000000000000e+00
```

Rather than rely on automatic type-conversion, you can be explicit about it by using a type-cast operator, e.g.,

```
f = (float)i;
```

This converts the number or variable or parethesised expression immediately to its right, to the indicated type. It is a good idea to use type-casting to ensure that you leave nothing to chance.