Basic C data types and format specifiers

C/C++ requires each variable to have an explicit data type. When variables are passed to and by functions, the types must match or compile errors pop up. Don't pass a float to a function which requires an int, the compiler will not understand what you mean. C's smallest data type is the char, normally used for the storage of characters. Bits can be addressed, but only as portions of larger containers, such as an int or a long. Signed chars are of the same length using one bit for the sign and seven bits for the number. Thus, a signed char can range from {-127 to 127}. The next larger data type uses 16 bits and is normally signed. It is called int for integer. You can have an unsigned int, but you must specify that with unsigned int. While an int can range from {-32,767 to 32,767} an unsigned int ranges from {0 to 65536} or 2^{16}. For many problems, ints are the only data types you will use for your variables. But, your problem space may lie in the real world, where scientific notation is necessary. Those problems require floating point numbers. I don't normally use the data type float, preferring instead to use double. A float uses 32 bit floating point notation while double uses 64 bit notation. For especially large, or finely detailed problems, you can use long double which uses 80 bit floating point IEEE notation. I use double as a compromise between using more memory, and having to type extra characters to manipulate the larger numbers.

```
{Type,   memory size,   number range,   format specifier}

unsigned char 8 bits   [0 to 255]          %c
signed char   8 bits   [-127 to 127]       %c
int           16 bits  [-32767 to 32767]   %d  (check your system, they may
                                                be 32 bits in length)
long          32 bits  [-2,147,483,647 to 2,147,483,647]
                                           %li, %ld
float         single precision floating point    1 bit sign, 8 bit exponent,
                                                 23 bit fraction in 32 bits
                                           %f   %E for scientific notation
double        double precision floating point    1 bit sign, 11 bit exponent,
                                                 52 bit fraction  in 64 bits
                                           %fl   %El for scientific notation
long double   extended precision floating point  1 bit sign, 15 bit exponent,
                                                 63 bit fraction  in 80 bits
                                           %Lf  %LE  for scientific notation
   OR         quadruple precision floating point 1 bit sign, 15 bit exponent,
                                                 112 bit fraction  in 128 bits
```

Strings, what computer folks use to say "word" or "sentence", are made up of chars. This snippet of code reads the file women.txt into the char pointer array woman. The file is a list of women's names in upper case characters. The fscan() function reads a line of text from the file, then puts it into the preallocated space in str[80]. Use the library function strlen() to get the length of each name. We can allocate memory from the heap and return a pointer to that memory location. The line s = (char *)

malloc(strlen(str)); allocates the correct amount of memory and returns the address in s. Now we can point to the beginning of the string, and to the beginning of the newly allocated memory, to start copying characters one by one. The line s[0]=str[0]; copies the first character from the string into the first memory location. The rest of the characters are copied, modified, then written to the new memory slot. Some are modified because the list I found on the Internet was in all capital letters, but I changed the format to upper case for the first character of the name, then lower case letters for all the rest. ASCII notation makes this case change fairly trivial, simply add 32 to each character. In C, the first memory location of an array is also where any pointer to that array would point. So we can use s[0]=str[0] to describe exactly what we mean instead of saying s = str[0], or even s = str and getting confused. They each mean the same thing and will compile to the same code. One notation is convenient for humans to read, the other not so much. I find not getting confused is preferable, so I add extra characters.

```
char *woman[100];
char str[80];
char *s;

fp = fopen("women.txt", "r");
if (fp != NULL)
   {
   for (int i=0; i<100; i++)
      {
      fscanf(fp, "%s", str);
      fscanf(fp, "%d");

      s = (char *) malloc(strlen(str));
      woman[i] = s;       // set pointer to first char of name string
      s[0]=str[0];        // s and s[0] are exactly the same
      for (int j=1; j<strlen(str); j++)
         s[j]=str[j]+32;  // change to lower case
      s[strlen(str)]='\0';

      //printf("%s\n", (char *) woman[i]);
      }
   fclose(fp);
   }
```

These are the data types which I use most often. Integers, floating point numbers, and strings comprise most of the types you will use on any normal project. When you want to print them you will need to remember the format specifiers for each. Integers require %d, floating point uses %f, strings require %s, characters require %c, and pointers %p. These specifiers cover 95% of what you will use. If you see %12.9f or %12d you'll know you are formatting the output. In the first case a floating point number with a sign (if necessary), one digit before the decimal point, and nine after it. In the second case an integer needs to fit within 12 spaces. You can write nicely formatted reports, with correctly aligned columns, by using the right numbers in your format strings.