ECE 222 System Programming Concepts
Lecture 5 – Memory map

A good method to understand all variable types, especially those involving pointers, is to construct a map of memory. Memory can be thought of as a 1D array. In most modern chips and computer systems, memory is byte addressed, so it is convenient to break up the memory array into bytes. In addition, when we program in C, we do not reference variables using explicit addresses; instead, we use variable names that provide labels for addresses. For example, consider the following code:

```
char    cv;
int     iv;
float   fv;
double  dv;

cv=2;
iv=7;
fv=3.2;
dv=-4.6;
```

The corresponding memory map can be drawn as follows (the starting address value of 1000 is chosen arbitrarily):

label	address (byte)	value
cv	1000	2
iv	1001-1004	7
fv	1005-1008	3.2
dv	1009-1016	-4.6

A char takes up one byte, an int and float take 4 bytes each, and a double 8 bytes.

What about an array?

```
char    ca[3];
int     ia[3];
```

Each cell of an array has a unique label, and address:

ca[0]	1017	
ca[1]	1018	
ca[2]	1019	
ia[0]	1020-1023	
ia[1]	1024-1027	
ia[2]	1028-1031	

Now, how does a pointer fit into this? For example:

```
char    *cp;    /* pointer to char */
int     *ip;    /* pointer to int */
float   *fp;    /* pointer to float */
double *dp;     /* pointer to double */
```

A pointer variable holds an address. How big is an address? (How many bytes are needed to store an address?) Suppose an address were only 1 byte. How many different addresses could be stored? Only 2^8, or 256 unique addresses. That's not very many, certainly not enough for large numbers of variables, or even a single large array.

Try working backwards. A modern computer can have as a maximum 4GB of memory. Why? It's because the modern chips use 32 bit addressing, and 2^{32} = 4GB. How many bytes are in 32 bits? The answer is 4 bytes, which is how many bytes an address takes to store. Therefore, the memory map for the pointers looks like:

cp	1032-1035	
ip	1036-1039	
fp	1040-1043	
dp	1044-1047	

No matter what the pointer "points to", or addresses, it takes 4 bytes to store the value (which is an address).

In **variable declaration**, the symbol * indicates a pointer variable. It's sort-of like indicating a fifth variable type, called "pointer".

In **variable usage**, there are two symbols of interest: & indicates "address of" a variable, and * indicates "at the address in" a variable. For example:

```
cp=&cv;         /* fill in the appropriate entry in the memory map */
ip=&(ia[0]);    /* fill in the appropriate entry in the memory map */
*ip=42;         /* fill in the appropriate entry in the memory map */
```

Note the two different uses for the * symbol, one during variable declaration, and one during variable usage. Do not be confused; one is only giving the variable type (pointer), the other is using the pointer to place a value at another address.

A natural question to ask is why use the complicated notation

```
char    *cp;
```

to declare a variable, instead of something simple like

```
pointer cp;
```

After all, if we have keywords to define the other four data types, why not a keyword to define this fifth data type (that stores an address)? The answer is **pointer arithmetic**. When adding/subtracting amounts from an address, pointer arithmetic does it in quantities of bytes equal to the size of the thing referenced. For example:

```
cp=&(ca[0]);   /* fill in the appropriate entry in the memory map */
*(cp+1)=8;     /* fill in the appropriate entry in the memory map */
     ^^
     +1 what?  +1 char, = +1 byte
*(ip+2)=33;    /* fill in the appropriate entry in the memory map */
     ^^
     +2 what?  +2 int, = +8 byte
```

Notice that using pointer arithmetic, the offsets match the indices of the arrays. This is the whole point. Pointers and arrays can be used interchangeably, in fact they are often the same thing. We study this more next time.