CSCI 240 Lecture Notes - Part 8

Memory Organization

All variables are stored in memory (often called RAM - random access memory).

Memory is measured using different units. The most common are:

- **bits** - can hold a 0 or a 1 - nothing more. A 2 is too big.
- **bytes** - can hold 8 bits, or a decimal value from 0 to 255
- **words** - depends on the system. Usually 2 or 4 bytes. 2 bytes can hold a value form 0 up to about 65,000. 4 bytes can hold a value up to about 4 billion.

Note: in the C++ compiler that Quincy uses, the data type *int* occupies 4 bytes. But since its values can be positive or negative, its values range from about -2 billion to about +2 billion because the leading bit is not part of the number, but rather the sign.

An "unsigned int" can also be declared. It has a range from 0 to about 4 billion (no sign bit - always positive).

There are also a "long int" or an "unsigned long int" which may have still different sizes depending on systems. These are official C++ data types.

sizeof (see below) can be used to get the sizes of the different data types, which means that it is then possible to calculate the ranges of values that can be held.

The uncertainty of the size of various data types in C and C++ is a problem. It was never defined as part of the language and it's too late to change now. More modern languages such as Java define the size and therefore the range of numeric data types so there is no uncertainty.

Usually memory is "addressed" in terms of *bytes*. Memory addresses range from 0 to the maximum amount on the computer.

We rarely have to know the actual address of a variable, but we do need to understand the idea of addresses in memory and the fact that variables take up a certain amount of space in memory.

Size of different data types (Quincy C++):

- char = 1 byte
- int = 4 bytes
- float = 4 bytes
- double = 8 bytes
- string = depends on number of chars

To find the *size* of a data type or a variable if you don't know it, use the *sizeof* operator:

```
sizeof (int);       //an expression that evaluates to 4

sizeof x;             //evaluates to amount of memory x takes up
```

Note: *sizeof* looks a bit like a function, but it's not, really.

So to summarize: each variable has an ***address*** (which we usually don't know or care) and a ***size***.

Arrays and Functions

First, understand that for any array, the array starts at some given address (say 1000). Each element in the array comes after the preceding one in memory, one after another.

A Surprise!

When an array is passed to a function, the function ***can change values in the array***.

This is ***unlike*** anything we have done before when passing arguments to a function. Before, values were passed as a ***copy***.

An array is passed to a function by passing its name.

```
//prototype: note the notation for an array of int: int[]

void incrAr(int[]);


//****************************************

int main()
{
int i;

int ar[4] = {1, 2, 3, 4};

incrAr(ar);

for (i = 0; i < 4; i++)
  {
  cout << setw(3) << ar[i];      // 2, 3, 4, 5
  }

return 0;
}

//****************************************

void incrAr(int A[])
{
int i;

for (i = 0; i < 4; i++)
  {
  A[i]++;   // this alters values in ar in main()
  }
}
```

So - why does passing an array to a function allow the function to alter the array? What is different about passing an array and passing a simple variable such as an int or double or float?

> The unsubscripted name of an array is the **address** of the array.

So passing the name of the array to the function is passing the location in memory where the array was created and therefore the function "knows" where to get and place values.

In the example above, when we called *incrAr(ar);* we were giving the **address** of the array *ar* to the function. So any change the function makes to the array is to the (original) array in the calling function, not to a copy of the array. Since the function has the address of the array, it can operate directly on it - it "knows where it is".

So, to summarize, the rules for passing *arrays* to a function are *different* than the rules for passing *simple* data types.

Rules for passing arrays:

1. The **function prototype**. Declare an array argument as "data-type []"

   ```
   int dodomething( double [] );
   ```

   Note: a number can be included in the [], but usually the effective length of the array will be passed as a 2nd argument. (Discussed later under #4.)

2. In the **function header** itself, same as above, but also include a name for the array argument

   ```
   int dodomething( double anArray[] )
   ```

3. Inside the function, use the subscripted local name as given in the argument list:

   ```
   int dodomething( double anArray[] )
     {
     double j;

     j = anArray[3];

     anArray[5] = 22.5;

     return j*2;
     }
   ```

   Note: as explained above, any assignment to an array element will change the actual value in the array passed by the calling program.

4. As mentioned above, it is typical (there are exceptions) to pass a second argument to a function that processes an array, to tell it how many elements there are to be processed. This is sometimes referred to as the "effective size" of the array.

   For example, suppose you want to call a function to load values into an array. When you call this function, you don't know how many values there will be. So you design the function to **pass** the array and **return** the number of values loaded.

   ```
   int loadAr( int[] );              // note the return type
   ```

   Then later, let's say you want to print the array. You design a function to pass the array and the number of array elements you want to print (i.e. the value returned from *loadAr()*).

   ```
   void printAr( int[], int )
   ```

   returns *void* - since there's no "answer" to report back.

   1st arg is the array

   2nd arg is the number of elements to process. This was returned by *loadAr()* and (presumably) stored into a variable so it could be passed to *printAr()*.

   ```
   int numArray[100];
   int arSize;

   arSize = loadAr( numArray );

   printAr( numArray, arSize );

   ....

   void printAr( int ar[], int size )
   {
   int i;

   for( i = 0; i < size; i++ )
     {
     cout << ar[i] << endl;
     }
   }
   ```

It's important to recognize that if a single array element is passed to a function, it is passed by value (i.e. a copy is passed) since an element of an array is a **simple** data item.

So if a function is called:

```
myFn( ar[i] );
```

Then *myFn() cannot* alter ar[i] no matter what it does, since ar[i] is a simple integer (or whatever type *ar* was declared as).

And, since ar[i] is an integer, the argument in the function **prototype** for *myFn()* would need to be *int* (not int[]):

```
void myFn(int);
```

To reiterate: It is good to recall that unlike arrays, when a simple argument is passed, a *copy* of the *value* of the argument is passed. Code in a function can't directly alter the original value of its argument (in the calling function) since it has no idea where it is. It has a *copy*. It does not have the *address*.

Examples:

Function cannot alter original value of the argument passed:

```
void fn( int );

int main()
  {
  int num = 5;

  cout << num;   // 5

  fn(num);

  cout << num;   // still 5

  return 0;
  }


void fn(int i)
  {
  cout << i;  // 5

  i = i + 1;

  cout << i;  // copy is now 6
  }
```

Notice that a function can be used to *indirectly* alter a value. If a function is written so it returns a value:

```
int fn(int i)
  {
  cout << i;  // 5

  i = i + 1;

  cout << i;  // now 6

  return (i);
  }
```

and (for example) *main()* calls it like this and assigns the return value to a variable:

```
cout << num;   //5

num = fn(num);    //num altered by assignment

cout << num;   //now 6
```

The variable num *is* altered. But understand clearly that **the function didn't alter it** - ***main()*** **altered it** by assigning a value to it.

Sample Exercises:

1. Suppose you have declared two arrays, A and B in a program. Array A already has values in it, and the number of values is stored in N. Write a function to copy the N values in A to B. Write the calling statement also.

2. Do the same, but store the values from A into B in *reverse* order.

3. Write a function to add up and return the sum of all the numbers in an array of int up to but not including the first 0 value. (Assume there is a 0 value in the array.)

Sorting an Array of Values: Selection Sort

Given an array of variables and a way of comparing them (which we can do with char, int, double, and float, and later with strings), it is often useful to be able to sort them in the array.

That is, no matter what order they were entered, we can sort them so they end up in numeric (or alphabetical) order.

There are many ways (algorithms) to do this. Some are simple and some are complex. Some are slow and some are fast. We will look at one way that is fairly simple and fairly fast, called **selection sort**.

Imagine you have an list of numbers:

```
8   4   2   5   3   7   6   5   9
```

Look through the whole list and find the smallest. (the 2)

Then exchange the smallest with the first item on the list and put a mark after the first number:

```
2   |  4   8   5   3   7   6   5   9
```

Note the | which denotes that everything to the left is (a) already sorted and (b) is smaller than any value on the right.

Now look through the right-hand part of the list (to the right of the |) and find the smallest in that part. (the 3)

Exchange it with the first number to the right of the | and move the mark to after the 2nd number

```
2   3   |  8   5   4   7   6   5   9
```

Repeat this:

```
2   3   4   |  5   8   7   6   5   9
2   3   4   5   |  8   7   6   5   9                  // either 5; doesn't matter
2   3   4   5   5   |  7   6   8   9
2   3   4   5   5   6   |  7   8   9
```

Now *you* can see you're done, but if you blindly continue, the results will be the same.

We will now turn this idea into a flowchart to represent computer code, with certain sub-functions to help control the complexity. (In lecture)

Two-Dimensional Arrays

So far we have used one-dimensional arrays. In C++, you can create multi-dimensional arrays. A 2-dimensional array can be imagined as a grid of cells or elements. To access or specify one of the elements, you need two subscripts - a row number and a column number, if you like.

In memory, the array elements are laid out in a linear manner (since memory is addressed linearly). So everything in the first "row" will come before everything in the second "row" and so on.

Declaration: data_type array_name[# of rows][# of columns];

```
int twoDArray[10][5];
```

this reserves memory for 50 elements.

Let's initialize it to contain some multiplication facts.

```
int row, col;

for (row = 0; row < 10; row++)
  {
  for (col = 0; col < 5; col++)
    {
    twoDArray[row][col] = row*col;
    }
  }
```

We would then have:

```
0   0   0   0   0   <- row 0
0   1   2   3   4   ...
0   2   4   6   8   ...
0   3   6   9  12   ...

etc.
```

Understand that these are arranged in memory as just a linear string of values from a beginning address to an ending address. All of each row is stored before the next. So if you were to look in memory, you would see:

```
0 0 0 0 0   0 1 2 3 4   0 2 4 6 8   0 3 6 9 12
```

2D arrays are passed to functions much like single-dimensional arrays are passed. The main difference is that you have to declare the number of elements in a row explicitly. That is, you have to tell C++ how many columns are in a row. Otherwise C++ would not know where one row ends and the next begins - remember all you pass is the address of the beginning of the array. So where would array[2][2] be? It would depend on how many elements (columns) are in a row.

So you'd declare the above array - passed as an argument - as follows:

```
void aFn(int ar[][5])
  {
  code to process the array goes here
  }
```

Note that the first dimension can be left empty. Alternately, you can supply it as well: (int ar[10][5]) in the function header above.

We will study more about multidimensional arrays in later courses.