Arrays in C++

An array is a consecutive group of memory locations that all have the same name and the same type. To refer to a particular location, we specify the name and then the positive index into the array. The index is specified by square brackets, []. The first index is 0.

For example, let's say we define an array of char of size six:

```
char a[6];
```

Java people: notice that the [ ] are in a different place! The above in Java would be declared as:

```
char[] a = new char[6];
```

The array allocates in the computer 6 bytes for the array, one for each character:

Memory Address	**Array Index**	**Contents**
X, e.g. 1000	a[0]	
X+1, e.g. 1001	a[1]	
X+2, e.g. 1002	a[2]	
X+3, e.g. 1003	a[3]	
X+4, e.g. 1004	a[4]	
X+5, e.g. 1005	a[5]	

Initially the contents of the array variables are unpredictable values, just like other un-initialized variables. For example, the way we defined the array in C++ will create the array on the stack. It will get whatever values happened to be sitting around on the stack. By knowing that arrays are created on the stack tells us we can't create arrays that are too big or we will overflow the stack.

How does the computer know how much memory to allocate to each array element? It allocates enough memory to hold the size of the data type the array is defined for. Let's say that our computer allocates 2 bytes for an integer, and we define our array as:

```
int a[6];
```

This results in memory allocation:

Memory Address	**Array Index**	**Contents**
X, e.g. 1000	a[0]	
X+2, e.g. 1002	a[1]	
X+4, e.g. 1004	a[2]	
X+6, e.g. 1006	a[3]	
X+8, e.g. 1008	a[4]	
X+10, e.g. 100A	a[5]	

Each array index is treated like a separate integer variable.

Here are some simple ways we can access the array just like it was a variable:

```
a[3] = 54;                  // Stores 54 into array element 3
a[0] = a[3];                // Copies 54 into array element 0
a[5] = a[2+1];              // Copies contents of a[3] to a[5]
i=5;
a[i]++;                     // Increment value in a[5]

for (i=0;i<6; i++) cout << a[i] << endl;        // Print each array element
```

The flexible thing here is we can programmatically access each variable, based on the index, instead of hard-coding the access by hand.

To initialize an array, we could use a loop as above:

```
int a[6];
int j;

for (j=0; j<6; j++) a[j]=0;
```

We can also use initializers for arrays. To define an array and initialize it at the same time, we can use curly braces { } :

```
int a[6] = {1, 2 , 3, 4, 5, 6};
```

This sets a[0] to 1, a[1] to 2, a[2] to 3, a[3] to 4, a[4] to 5, and a[5] to 6.

An equivalent way to initialize the same array is to use:

```
int a[] = {1, 2, 3, 4, 5, 6};
```

Since we are initializing the array with six elements, the compiler knows to make the size of the array six.

C++ also has a special initializer to set all elements of an array to zero:

```
int a[6] = {0};          // sets all array elements to 0
```

This sets a[0] to 0, a[1] to 0, etc. up to a[5]. However note that this ONLY works for the value zero, it does not work for anything else. The following code would only set a[0] to 5, a[1] through a[5] will all contain unpredictable, uninitialized data:

```
int a[6] = {5};          // ONLY sets a[0] to 5, not all elements
```

A common programming technique when using arrays is to define some constant for the array size. This has the benefit that if you ever need to change the size of the array, you can do it in one place. If you hardcoded a number like "6" into the program, then there may be many places you need to change:

```
const int ARRAYSIZE = 6;
int a[ARRAYSIZE], j;
for (j=0; j<ARRAYSIZE; j++) {
        a[j]=j;
}
```

C++ lets you use a variable for the size instead of a constant if you like, but then you can't use any initializers.

Here is a sample program that uses arrays. What is the output of this program?

```
int main ()
{
     const int MAX_ARRAY = 5;
     int  numbers[MAX_ARRAY];
     int  i,j;
     int  sum;

    // Store values in the array.
    for (i =0,j=1; i < 5; i++,j+=2)
      numbers[i] = j;

    for (i=0,sum=0; i <5; i++)
      sum+=numbers[i];

    cout << sum - (MAX_ARRAY * MAX_ARRAY) << endl;

    return 0;
}
```

C++11 has a handy construct called a **for-each** loop. The for each loop will iterate through every item in an array for you, so you don't need to figure out what the size is.

Here is an example:

```
string arr[3];
arr[0]="foo";
arr[1]="bar";
arr[2]="zot";

for (string s: arr)
    cout << s << endl;

for (auto s: arr)
    cout << s << endl;
```

The for each loop will output each item in the array. The second for each loop illustrates where we might use the "auto" keyword. It will automatically set s to the proper type for whatever is stored inside the array. There are some limitations if you pass an array to a function. We will more commonly use the for each loop with the STL collection classes, which we will cover later.

C-Strings

So far we've mainly been looking at arrays of numbers. Arrays of characters have some special functions all their own. Arrays of characters are called **strings**, or more specifically, C-Strings (named after their use in the C programming language). C-Strings are not to be confused with the string data type that we have used previously. C-Strings are much more primitive and came before the string data type.

An array of characters can be initialized as follows:

```
char str[] = "hello";
```

This actually creates an array with a special null character at the end. The null character (essentially the number 0) marks the end of the string. The actual contents of the array str are:

str[0]='h', str[1]='e', str[2]='l', str[3]='l', str[4]='o', **str[5]= '\0'**

The \0 character is the **null** character that ends the string. If we were to explicitly initialize the array with a size, then make sure that you add in an extra character to account for the null character:

```
char str[5];        // Would be too small to hold "hello" with a null
char str[6];        // Large enough to hold "hello" with the null
```

Once a C-string has been declared, you can use it much like you would use any other variable. For example, you can input and output data into a string with cin:

```
cin >> str;
cout << str << endl;
```

However, you must be very careful not to input anything more than the length of the string! The result will be that the input data will blindly be copied into memory as if the string was large enough. You might then end up overwriting parts of your program with the input data. C++ does NOT perform any bounds checking on your arrays. In other words, if you try to input or access anything outside the bounds of the array, C++ will let you , sometimes with fatal consequences to your program. Some other programming perform error checking and don't allow any access outside the bounds of the array.

For the above string, if we tried to input some data more than 5 characters long, we will end up overwriting some other data outside the end of the array. This is sometimes called the **buffer overrun problem** and it is a very common bug / exploit that can be used to crash or break into systems that are not properly coded.

While we can print and input into C-strings, we can't directly copy one to the other (for reasons we will describe shortly). For example, the following is invalid:

```
char str1[10]={'h','i','\0'};
char str2[10];

str2="yo";                  // INVALID!  No direct assignment

str[0]='y';
str[1]='o';
str[2]='\0';                // Valid

str2=str1;                  // INVALID!  No direct assignment

if (str1==str2) …           // Valid!  But probably not what you want
```

The last one only compares the memory addresses and is true if str1 has the same address as str2.

Instead, C/C++ provides a library of string functions that support copying.
To use them, add **#include <cstring>** as a header. Note the leading c must be added; this denotes the old C string library from the newer C++ STL String library included via #include <string>.

A few functions that are supported in the string library are:

strcpy(dest,source);

- Copies *source* string to *dest* string, up to and including the \0 in the source

int strlen(str);

- Returns the length of the string by counting the number of chars to the \0

strcat(dest, extra);

- Appends string *extra* onto the end of string *dest*

int strcmp(str1, str2);

- Compares str1 to str2 and returns a negative number if str1 is less than str2, zero if str1 is equal to str2, and a positive number if str1 is greater than str2

Note that to use strcpy and strcat, the dest string must be large enough so that there is no buffer overrun. For example, the following would result in problems:

```
char dest[10];

strcpy(dest,"hello");          // OK so far, dest[5]='\0'
strcat(dest,"world");          // Problems, tries to copy \0 to dest[10]
                                  and only have space allocated dest[0]…dest[9]
```

These examples illustrate that you must be very careful with C-strings, as there are many potential places for bugs to creep in, especially with exceeding the bounds of the array. For these reasons, the C++ string data type is much more flexible and much safer. You can also perform direct comparisons and assignments:

```
string str1="hi";
string sr2;

str2="yo";                     // Valid!
str2=str1;                     // Valid!
if (str1==str2) …              // Valid!
```

Be careful not to confuse C-strings with the C++ string data type. In general, the two are not compatible with one another. For example, you couldn't use the strcpy function on a variable of type string, only a variable that is an array of char.

If you ever need to convert back and forth between a C-string and a string, then going from a C-String to a string is easy. Just use assignment:

```
char str[] = "hello";
string myString = str;
```

To turn a string into a C-String, there is a function called c_str() :

```
myString.c_str();
```

This returns a C-String from myString.

Two-Dimensional Arrays

A two-dimensional array is a collection of data of the same type that is structured in two dimensions. Individual variables are accessed by their position within each dimension. You can think of a 2-D array as a table of a particular data type. The following example creates a 2-D array of type float:

```
const int MAX_ROWS = 10;
const int MAX_COLUMNS = 5;
float  twoDimAry[MAX_ROWS][MAX_COLUMNS];
```

twoDimAry is an array variable that has 10 rows and 5 columns. Each row and column entry is of type float. The following code fragment sets all the entries in twoDimAry to zero:

```
for (int column = 0; column < MAX_COLUMNS; column++)
   for(int row = 0; row < MAX_ROWS; row++)
      twoDimAry[row][column] = 0.0;
```

In processing two-dimensional arrays, the number of rows and columns in the two-dimensional array variable is fixed at compile time. The number of rows and columns that contain valid data items can vary as the program executes. Therefore, each dimension should have a parameter associated with it that contains the number of rows or columns actually used.

Processing a two-dimensional array variable requires two loops: one for the rows and one for the columns. If the outer loop is the index for the column, the array is processed by column. If the outer loop is the index for the row, the array is processed by row. The preceding loop processes twoDimAry by columns.

Internally, data is stored starting at a row and going into the columns. This means if you want an array of ten C-strings, you would declare it so that the length of each C-string is in the columns:

```
const int MAX_STR_LEN = 50;
const int MAX_STRINGS = 10;
char strArray[MAX_STRINGS][MAX_STR_LEN];
int i;

for (i=0; i<10; i++) strcpy(strArray[i], “string up to 49 chars”);
```

Once again, for a C++ program, it is recommended to instead declare an array of string instead of using C-strings:

```
string strArray[MAX_STRINGS];
```

Multidimensional Arrays

You have seen one-dimensional and two-dimensional arrays. In C++, arrays may have any number of dimensions. To process every item in a one-dimensional array, you need one loop. To process every item in a two-dimensional array, you need two loops. The pattern continues to any number of dimensions. To process every item in an n-dimensional array, you need n loops.

For example, if we wanted to declare an array of 3 dimensions, each with 10 elements, we could do so via;

```
int three_d_array[10][10][10];
```

Passing Arrays To Functions

When arrays are passed as a parameter to a function, specify the name of the array without any brackets in the function call. For a one-dimensional array, the function definition should include the brackets, but it is not necessary to specify the exact length of the array. For example, the following illustrates passing a single one-dimensional array to a function:

```
void  SomeFunction(int arr[])          // Use empty brackets for 1-d array
{
        arr[0]=10;
}

int main()
{
        int intArray[100];

        SomeFuntion(intArray);         // Leave brackets off
        cout << arr[0] << endl;        // Will output 10 since array is reference
}
```

The way that arrays are passed is always by reference. That is, if we change the contents of arr[] inside SomeFunction, the changes will be reflected back in intArray inside main. This is because arrays are always passed by reference. One reason for this is efficiency – if arrays were passed by value, it would mean copying the entire contents of the array on the stack. If the array was very large, this would take a long time. In the above example, the function changes array element 0 to 10, and this change will be reflected in main.

What actually happens is that **using the array variable without any brackets is really a pointer to the place in memory when the array is stored**. This has implications later, but for passing the array as a parameter, it means we're really passing a pointer to the place where the data is stored. We follow the pointer inside the function to change the contents of the source array.
It is important to note that since C++ doesn't specify the length of the array in the function, the function either has to already know the length of the array (perhaps through some global variable, or by definition, or by some special character to signal the end of the array, like '\0' for strings), or we must pass in the length of the array. If we were to pass in the length of the array, the function prototype would look like:

```
void  SomeFunction(int arr[], int length);
```

and would be invoked via:

```
int intArray[100];
SomeFunction(intArray, 100);
```

The function then knows how many elements are in the array so that it wouldn't inadvertently exceed the bounds of the array and access memory that it shouldn't get its fingers on.

Passing Multidimensional Arrays to Functions

For arrays of more than one dimension, the function must know the sizes of all of the dimensions except the first. For example, if a function is defined to set the first num values of each row in twoDimAry to a specific value, the prototype might look like this:

```
const int MAX_ROWS = 10;                    // Global Variables
const int MAX_COLUMNS = 5;

// Function Prototype
void ProcessArray(float twoDimAry[][MAX_COLUMNS], int rowsUsed);

In main:
        float  twoDimAry[MAX_ROWS][MAX_COLUMNS];
        ProcessArray(twoDimAry, MAX_ROWS);
```

The reason the compiler needs to know the array bounds is so it knows how far to "jump" in memory to access a particular cell. For a 2D array with 3 columns and 4 rows, with one byte of storage per element (e.g. char a[4][3]), if the array was stored starting at memory location 100 then each cell has the following memory addresses:

Address=100 a[0][0]	Address=101 a[0][1]	Address=102 a[0][2]
Address=103 a[1][0]	Address=104 a[1][1]	Address=105 a[1][2]
Address=106 a[2][0]	Address=107 a[2][1]	Address=108 a[2][2]
Address=109 a[3][0]	Address=110 a[3][1]	Address=111 a[3][2]

If we wanted to access a[2][2] for example, the computer needs to access memory location 108. This is actually computed by taking the starting address of 100, and then figuring out the memory address of the row we want to access. In this case, row 3 begins at starting-memory-address + (element-size * num-columns * desired-row), or 100 + (1 * 3 * 2) or 106. From that, we add on (element-size * desired-column) or 106 + (1 * 2) to give us memory location 108. If we didn't know the number of columns, we couldn't properly compute where in memory to go.

Why don't we need the number of elements in the function declaration for a one dimensional array?

Consider (char a[4]) also starting at memory location 100:

Address=100 a[0]
Address=101 a[1]
Address=102 a[2]
Address=103 a[3]

A function operating on this array doesn't need to know the length of the array to access a particular element. It only needs to know the size of each element, which it already knows from the data type (e.g., char is 1 byte).

To access a[2] for example, the computer needs to access memory location 102. It computes this by taking the starting address of the array and adding (element-size * desired-index). In this case, we get 100 + (1 * 2) or memory location 102.

Array/Function Exercise:

Write a program that inputs 10 integer, numeric grades into an array and finds the max. Write main and the following functions:

```
int findMax(int grades[]);
void inputNumbers(int grades[], int numGrades);
```