Programming Notes 7: Reference Parameters & Basic Arrays

As usual, more examples of the notes are in the example program.

1. Reference Parameters

Before we get to arrays, there's a small but important concept that you need to be familiar with—reference parameters. In the past, I've emphasized how the formal parameters of your functions are simply copies of the actual parameters passed into the function when you call it. However, if you define a parameter as a reference parameter, how you change it in your function will actually change the value that was passed into it, for example in main. To define a reference parameter, the syntax is exactly the same as a normal parameter, except that you put an ampersand before the parameter name. As always, remember to also include it in the function prototype. Example:

```
void refFunction(int&);

main() {

  int variable = 5;

  cout << "Variable before function: " << variable << endl;

  refFunction(variable);

  refFunction << "Variable after function: " << variable << endl;

}

void refFunction(int &value) {

  value += 5;

}
```

This short program will first output that the variable is 5, as you would expect. However, after the function is called, because it has a reference parameter, the value will actually be changed within MAIN, and the program will output that the variable holds the value 10 after the function.

2. Concept of Arrays

Now, on to another vitally important concept of programming—arrays! Arrays are the basic way you can store more than one piece of data in a variable. They're quite simple: an array is essentially a collection of a type of variable. For example, you could have an array of 10 integers, meaning you can store 10 integer values in that one array. That's really all you need to know to get started using them, although we will get into more complexities when we do pointers and dynamic memory.

3. Declaring Arrays

Arrays are declared in the same way as variables, except with the addition of square brackets. As with normal variables, you first declare the type of your variable, for example an int, double, or char. Then, type your identifier, and finally the array size in square brackets at the end of the identifier. Finally, when you declare an array of a fundamental type (and only when you declare it), you can set all values to a certain initial value by adding "= {value}" to your declaration. Examples:

int integerArray[50];   Declares an array of 50 integers

char characterArray[10] = {'a'};   Declares an array of 10 characters, and initializes all of them to 'a'

4. Using Arrays

Now, arrays aren't very useful if you don't know how to get at the data contained within them. Well, it's also quite simple. To access an element in the array, simply write the array's identifier followed by brackets with the position of the element in the array. Always remember that the positions of the elements, or indices, always begin at 0. This means that the first element in the array is at index 0, the second at 1, and so on. Additionally, while it may be tempting to simply set one array identifier equal to another to set the arrays equal, that is not allowed (for reasons we will get to later). Hence, if you want to cope one array to another, you must loop through it and copy each element individually.

This is one of biggest sources of errors for new programmers, as it can produce many of those "off-by-one" errors I mentioned when discussing loops. However, it's not just off-by-one errors. If you try to access or assign a value that is past the end of an array, or before it begins, you may or may not generate an error, but it will always be a bug. You won't always generate an error because if you go off the end of an array, you will start assigning values to whatever memory was after the end of your array. If that memory was used by the operating system, your program will throw an error known as a segmentation fault, or seg fault for short. People tend to have a love/hate relationship with seg faults, as if you have one, it's good—you know that you have an error, and probably where the error is. However, they're also bad—they might mean that you have other seg faults waiting to happen, but haven't—remember that going into memory past or before your array might not cause your program to crash. To top it all off, the memory arrangement will be different on different computers, so your program may behave differently. If there's anything to take from all this, just remember to be very diligent in looking for errors when using array.

Now, let's get to some actual examples, shall we? Using the arrays declared in the last example:

integerArray[0] = 5;   This sets the first element in the array to 5

characterArray[5] = 'l';   This sets the sixth element in the array to 'l'

char testValue = characterArray[10];

The 3rd example will try to access the 11th element in the array, when it only contains 10 values. Hence, it will try to read the memory after your array, and might throw a seg fault.

5. Manipulating Arrays with Loops

While being able to assign values to individual elements in an array is useful, most the array processing you'll be doing will be in the form of loops. There isn't anything new to learn here, but realize that you can do things like loop from 0 to your array size, and assign a value to each element in the array using your counter variable. This is one of the ways that the "for" loop is most useful. Finally, remember to be careful of off-by-one errors—for example, if the first example looped until "i" was less than or equal to 10, in the last iteration it would try to access array sub 10, meaning the 11th element and possibly an error. Examples:

```
for(int i = 0; i < 50; i++)
  integerArray[i] = 0;
```

This example simply loops through each element in the array and sets it to 0;

```
for(int i = 0; i < 10; i++)
  characterArray[i] += i;
```

This example will loop through each value of the array, adding one more to each one consecutively (i.e. add zero to the first element, so it will remain 'a', add one to the second element, so it will be 'b', etc.).

6. Passing Arrays to Functions

Remember the reference parameters we discussed at the start of this topic? Well, here is where they come into play with arrays. When you pass an array to a function, it will automatically act as a reference parameter, even if you don't use the ampersand. Keep this in mind.

To actually pass an array to a function, you must first make it clear that your function will take an array: in the prototype, simply type a pair of square brackets with nothing in between. You don't actually have to specify the size of the array when you pass it to a function, it just has to know it's an array (although this will change slightly with multidimensional arrays). You do the same in the function implementation, and finally, when calling the function with actual parameters, simply type the array identifier without any brackets at all. Lastly, remember that when you pass arrays to functions, they automatically act as reference parameter, so your function will change the variable in main. Example:

```
void clearArray(int[],int);

main() {

  int integerArray[10];

  clearArray(integerArray,10);

}

void clearArray(int array[], int size) {
  for(int I = 0; I < size; I++)
    array[i] = 0;
  }
}
```

7. C-Strings

The final subtopic in this section are what are called C-Style strings, or just c-strings for short. C-strings are technically just arrays of characters, but there are many special ways of working with them. The basic principle is quite simple—each element in the array holds a corresponding character. C-strings are "null-terminated," meaning a null character ('\0') signifies the end of the string. Because of this null character, the size of the array must always be *at least* one more than the number of actual characters in your string. You can use character arrays like you would expect: for example, you can cin right to a c-string and your program will automatically copy the inputted word to your array, and add a null character to the end. When you use a c-string like that, it is called using it in the aggregate, and is what creates the special properties like automatically adding a null terminator. Finally, you can easily do character-by-character manipulation of c-strings, as you know that you have reached the end of the string when you find a null character. We won't be going very in-depth with c-strings (at least until we get to dynamic memory), but you should know how to declare and use them. Examples:

```
char cstring[50];
```

The first example declares a c-string that can hold 50 characters, but only 49 actual characters, as the null character must be the last one.

```
cin >> cstring;
```

This will input one word from the console and store it in the "cstring" array, automatically adding a null character at the end.

```
for(int i = 0; cstring[i]; i++)
  cout << cstring[i] << endl;
```

The final example will loop through the array until it comes to a null character (you can simply use the character as the conditional statement, as it will evaluate to true if it is anything but null) and output each character on a new line.