Programming Notes 10: Pointers

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

1. Memory and addresses

All data from your program is stored in your computer's memory, so it has to have some method of storing and retrieving data from memory. Hence, each location in memory has an address, which is basically a number (up to 2^64 – 1 for 64-bit machines) that corresponds directly to its place in memory. How your program uses this memory is managed by the operating system and is very complicated, but memory and addresses in your program is very simple. The fundamental concept in this topic is pointers. A pointer is simply a variable in your program, just like any other, except it holds a memory address. This means that it can "point" to another value in your program, meaning. They are very useful and hence used extensively in programming, so as with all these basic topics, they will be very important.

When using pointers, if you don't understand the relationship between some values, always try to draw a diagram—they are very helpful. A very basic example:

If we have:

Simplified memory:

Address	Type	Identifier	Value
0	-	-	-
1	Character	data	'x'
2	-	-	-
3	-	-	-
4	char pointer	dataPtr	1
5	-	-	-

2. Using Pointers

So, when creating a pointer, you must always (well, almost always) declare what data type it will point to. This is because when you use your pointer later, your program must know what kind of data it points to. To declare a pointer, first describe what type the pointer will point to, then add an asterisk ('*'). This marks the variable as a pointer to the type before the asterisk. Then, add your pointer's identifier and you've declared a pointer. Example:

```
char* characterPtr;
double* dblPtr;
```

To assign an address for your pointer to point to, you can't just set it to a literal value, as the memory "space," or range of values available will change with every execution of your program. In fact, I don't believe most compilers will even let you do this, unless you set the pointer to 0. A

pointer with the value 0 is called a null pointer, and will cause an error if it is used (a good thing, as you will see). Hence, to set your pointer to point to a certain other value in your program, you use the "address of" operator, which is a single ampersand. For example:

```
char value;
char* valuePtr = &value;

valuePtr now points to value.
```

At least in this use case, pointers wouldn't be very useful without a way to access the actual data they point to, rather than just the address it actually contains. To do this, you simply use the dereference operator (also an asterisk) followed by the pointer you want to dereference. This statement will act like the data type your pointer points to. The data you get from dereferencing the pointer will be the *actual* data contained in what it's pointing to. This means that if you change that data, you are actually be changing it within the value you're pointing to. This means that if you have a pointer to an integer, for example, and you assign a value to dereferencing that pointer, you have changed the value in the actual integer. For example:

```
char character = 'x';
char* valuePtr = &character;

*valuePtr = 'y';                    //This will actually change the value in "character;" it will hold 'y'
                                    after this line.

cout << valuePtr << endl;           //Will output the address of "value"
cout << *valuePtr << endl;          //Will output the actual data of "character," which is now 'y'
```

Finally, pointers don't ever *have* to be pointing to a valid piece of data—and because of this, if you try to dereference a pointer that doesn't point to a valid object, you will get one of those dreaded segmentation faults. This is why null pointers are so important. If you know a pointer does not currently point to any actual piece of data, always set it to null (or 0), so that you can test if it is valid later. Also, remember that pointers, like everything else, are also initialized with garbage values, so if you try to dereference a point you've just created, you will probably crash. For example:

```
char* valuePtr;
cout << *valuePtr << endl;          //Will probably crash - wrong


char* valuePtr = NULL;
if(valuePtr)
  cout << *valuePtr << endl;        //Will not crash because it makes sure that valuePtr is not
                                    NULL, and it is set to NULL if it is not pointing to a valid piece
                                    of data - correct
```

3. Pointers and Arrays

You've actually been using pointers ever since we learned about arrays—arrays are technically just pointers to data that cannot be changed. The actual identifier of your array (say "array") actually holds the address of the first element in the array, and to get a value from the array, you put the offset from that pointer in your brackets. This is why arrays are 0-based: to get to the first element, you move 0 pieces of data away from the first one. Note that offset notation does NOT move the pointer, it simply take a value relative to it. Example:

Address	Type	Identifier	Value
0	char pointer	array	2
1	-	-	-
2	char	-	'h'
3	char	-	'e'
4	char	-	'l'

So, "array" is the address of the start of the array. array[0] would go 0 places from the start of the array (so it would be at memory address 2, the first element), and then dereferences that address to get the actual value at that spot in the array ('h'). Likewise, array[2] would go to address 4 and return 'l.'

So, because arrays are technically just pointers, you can assign pointers to point to arrays, and basically use them exactly as you would use the same array. You can use brackets on pointers, which is called offset notation. Remember that offset notation will go that many spots after the first element in the array, and then dereference that address. For example:

```
char charArray[10];
char* arrayPtr = charArray;

for(int i = 0; i < 10; i++) {
  arrayPtr[i] = 'a';
}

cout << charArray[1] << endl;
```

This will output the letter 'a,' because when every value in "arrayPtr" was set to 'a,' it really set every value in "charArray" to 'a.'

4. Moving Pointers

One of the advantages to using pointers is that you can do math with them, just like you'd do math with integers. For example, if you have a pointer that points to the first value in an array, if you increment that pointer, it will now be pointing to the second value in an array. It's that simple! However, you must be very careful about doing pointer math, as you can easily send pointers into invalid memory, and when you try to access them, you will get a segmentation fault. For example, if you have a c-style string (which you know will be null terminated), you can loop through it, without knowing its length with a pointer as such:

```
char cstring[LENGTH];              //Declare string of unknown length
cin >> cstring;                    //Input unknown string

char* strPtr = cstring;
while(*strPtr) {
  cout << *strPtr << endl;
  strPtr++;
}
```

This will output the string with each character on a new line regardless of the length, as it will iterate through each value in the array until it comes to a value that is 0, i.e. the null character.

5. Passing pointers to functions

As with all other variables, you can pass pointer parameters to functions. They work just as you'd expect: the parameter will still hold the same address as it does in main, and you can access that data from within your function. Because of that, if you change data pointed to by a pointer parameter, it will be changed in main, or wherever else you call the function from. Because arrays are technically pointers, you can actually pass an array as a pointer parameter. Obviously, you're going to want to design your function so that it will deal with an array if it is going to be passed an array.

There are a couple other ways to pass pointers that you should know: first, do you remember leaning about how to pass normal parameters by reference? You can do the same thing with pointers. However, if you do, that means that the function can not only change the data pointed to by the pointer in main, it can actually change *where* the pointer points in main. This is pretty much useless right now, but it will come into play next week when we go over dynamic memory. Second, there is passing the pointer as a constant. If you do this, it does not allow your function to move the pointer (i.e. you can't set it to somewhere else, or increment it, etc.), but you can still change the data it points to. Additionally, this allows you to pass literal strings to your functions without a warning from the compiler. Note that this does not make the pointer in main constant. Examples (more in example file):

```
void func1(char* value);
```

This will take a pointer to a character value (or array), and can change the data in main

```
void func2(const char* value);
```

This will also take a char pointer, but WITHIN THE FUCNTION it cannot change where the pointer points, meaning you can't do stuff like value++. It can still change the data in main.

```
void func3(char*& value);
```

This function is just like the first, but not only can it change the data in main, it can actuallly change where the pointer in main is pointing to.

6. Void Pointers

Finally, void pointers. Void pointers are just like any other pointer, except they do not have to know what type they are pointing to; they do one thing and one thing only: store an address. These are useful when you don't know what exactly your type will be, but we won't be running into any of those situations for a while yet. To use a void pointer, you can use it just as any other pointer in terms of math and addresses, but to access any actual data that it holds, you must first cast it to the correct type. This basically tells the compiler to treat the void pointer as a pointer to a certain data type, and it can then get the data from that type. For example:

```
void* voidPtr;                          //Declares a void pointer

int value;
voidPtr = &value;                       //Points the void pointer to "value"

cout << *voidPtr << endl;               //This does NOT work, because the program doesn't know
                                        what type voidPtr points to.

cout << *((int*)voidPtr) << endl;       //This does work, as you tell the program to treat voidPtr as
                                        an integer pointer before dereferencing it.
```