CS225 : Data Structures and Software Engineering
Arrays and Strings

Jason Zych

Chapter 1

Arrays and Strings

We will now turn our attention to the usage of arrays and strings in C++. Unlike in Java, where arrays, string, and memory usage are only slightly-related topics, in C++ there is a *very* strong relationship between the ideas of pointers, arrays, and strings. The actual usage of these ideas tends to not be too different from Java, though, so first we will examine how to declare and use arrays in C++, and then we will turn our attention to the actual implementation of arrays in memory and how this implemention idea is related to the memory ideas we mentioned during the discussion of pointers.

1.1 Using Arrays in C++

As you should recall from Java, an *array* is a collection of cells of the same type which are numbered with consecutive integers and referred to by a single name. To access individual cells of the array, you use the array name and the particular number (called an *index*) of the particular cell you want within that array.

An array was declared in Java as follows:

```
int[] x;
x = new int[6];
```

The first line created a variable `x` which was a reference to an array. Just as all objects of user-defined types in Java were allocated with `new` and referred to by references, likewise arrays were also allocated with `new` and referred to by references – i.e. they were treated just like user-defined types, instead of treated as built-in types. So, the second line above, then, was the creation of the actual array object using `new` and the assigning of the array reference `x` to refer to that new array object.

The first array cell was always numbered 0, and the index values (the cell numbers) proceeded upward consecutively from there. So, in Java, an array of size 6, like the one we declared above, would have 6 cells, and those cells would have index values 0 through 5.

Finally, you accessed these cells by using the array name and the cell index together. The notation that was used was to first list the array name, followed by a pair of brackets (`[]`). Inside the brackets, you placed the index of the cell you wanted. So, you could then use statements in your code such as the following:

```
x[3] = 7;     // assigns the integer 7 to cell 3 of array x

int valueTwo = x[2];  // reads cell 2 of array x and stores
                      //   the value in the variable valueTwo

int total = x[0] + x[5];  // adds the values of the first and last
                          //   cells of the array and stores the
                          //   sum in the variable total.
```

Now, the reason we have chosen to review these Java ideas before proceeding to C++ is because arrays in C++ work almost exactly the same way! In fact, the actual usage in code statements is identical. It is the array declarations that are slightly different.

Recall from our pointer discussion that unlike in Java, where objects of user-defined types can only be created using `new` and referred to by references, in C++ you can create objects of user-defined types either as local variables *or* as dynamic objects. Likewise, in Java arrays must be created using `new` and referred to by references, whereas in C++ you can create arrays locally (in which case their existence ends when their scope ends – i.e. when you leave the function they were declared in or destroy the class they are a member of), or you can create them dynamically (in which case the memory for the array comes from the heap, not the stack, thus ensuring that the array will continue to exist until you explicitly delete it).

For now, we will focus mostly on local arrays, i.e. stack-based arrays. We will mention how to create dynamic arrays right now as well, but don't worry about that too much – just make a mental note of the idea and move on. We'll spend the next section discussing stack-based arrays in detail, and once we have completed that discussion, the things we need to say about dynamic arrays will make a lot more sense and we'll go back and explore dynamic arrays in more detail then.

To declare an array locally in C++, you don't need two lines or the use of the `new` function as with Java. To create an array called `x` of 6 cells indexed 0 through 5, you would use the following line of code:

```
int x[6];
```

That's it! – just the element type, followed by the array name and then the bracket pair with the array size inside. The array is used the same way it is used in Java – that is, the three examples of array usage that we had above, which we said came from Java code, could just as easily have come from C++ code. You use the array name, followed by the brackets with the index inside, and that gives you an array cell that you can either write to or read from, just as in those three examples above. Also, just as in Java, an array of size n is automatically indexed from 0 through $n-1$, and there is no way to choose to start at, say, 1 or 2 instead. So, the only real difference between C++ arrays and Java arrays in is the way those arrays are first declared.

It is also possible, as we stated above, to declare arrays dynamically in C++. Although we're going to postpone an involved discussion of that topic for just a bit, we will at least introduce how to declare such arrays right now. The declaration of a dynamic array is quite similar to the declaration of arrays in Java. First you need to create a (reference in Java, pointer in C++) to hold the address of the new array, and then you use `new` to create the array and send the address of that array to the (reference or pointer).

```
int* arrayPtr;
arrayPtr = new int[6];
```

Then, you would again use the array just as you did in Java, with statements such as `x[3]=7;` or whatever else you wanted to do.

So, given that information, you now know how to declare and use arrays in C++. The details shouldn't be too hard to remember, because except for the declaration of the arrays, everything is exactly the same as it was in Java, and so if you remember how to handle arrays in Java, then you mostly remember how to handle them in C++ as well. Just remember the ways in which the array declaration syntax is different and you are all set.

1.2 Arrays in stack memory

(NOTE: During the course of this discussion, the phrase "array cell" or "array location" will be used to refer to the abstract idea of a cell in the array, and the phrase "memory cell" or "memory location" will be used to refer to a physical location in memory. If the type stored by the array only needs one memory cell (such as with an `int`), then we will say that the given array cell is stored in the given memory cell. However, if the type stored by the array needs more than one memory cell for each object of that type (such as with `Coord`, which needed four cells), then we will say that it is that *group* of memory cells that is needed to store the array cell. For example, we might say that an array of `Coord` objects needs four memory cells for each cell in the array.)

In addition to being indexed with consecutive integers, when arrays are implemented in memory they must use consecutive memory locations. That is, you cannot split an array inside memory, and use some memory cells from one portion of memory to store part of the array and some memory cells from some other portion of memory to store the other part of the array. All the memory cells used to implement an array must be in one consecutive sequence. This is due to the way that arrays are compiled and translated to machine code, not just in C++ but in programming languages in general. So, the discussion that follows is not just a discussion of arrays in C++, but how arrays in any language are implemented on the machine-code level.

Assume we have made the following array declaration:

```
int x[6];
```

This array consists of 6 cells, indexed with the integers 0 through 5. Since the type of the array elements is `int`, each array location will need one memory cell (since in our example machine, an integer takes up one cell, i.e. four bytes). So, our array x will need 6 consecutive memory cells.

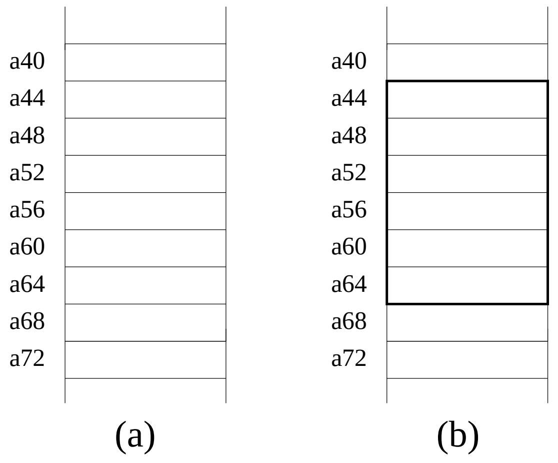

Figure 1.1: **(a)** Our memory diagram. **(b)** The allocated array needs six consecutive memory cells.

Now, conceptually, each of these cells has a name. The first cell, at a44, is named `x[0]`. The second cell, at a48, is named `x[1]`. And so on.

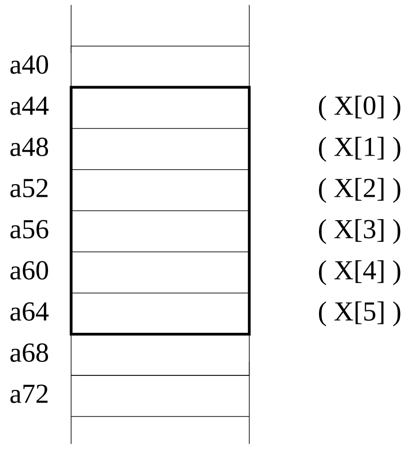

Figure 1.2: The six array cells have names – but only in the conceptual sense (which is why the names are in parenthesis).

However, imagine how this would be for larger arrays, such as an array of size 10000. If the compiler had to assign an individual name (`x[0]`, `x[1]`, etc.) to each of 10000 memory cells, that could be very time consuming and also would take up a lot of memory in the compiler. In addition, we may very well have *many* arrays of this size, making the problem even worse.

This problem is solved by the fact that the compiler can calculate the address of an array cell, provided it has, **(1)** the starting address of the array, **(2)** the index of the cell we want, and **(3)** the number of bytes needed to store an object of the type in question. It makes a decision on **(1)** when you declare your array, you tell it the index for **(2)**, and the type held in the array determines **(3)**, so the compiler should have all of that information.

So how is this calculation done? Well, let the starting address of an array be x. If we know how many memory cells are needed to store one object of the given type, we can simply multiply that by four to get the number of bytes needed to store an object of the given type (recall that each memory cell is composed of four bytes, which is why our memory cells are addressed with multiples of four).

So, let the number of bytes needed to store one object of the relevant type be L. If the index of the array cell we want is i, then we can find the memory address of the start of this array cell with the following formula:

```
starting address of x[i] = x + L * i
```

That is, you take the starting address of the array, and then move downward in memory $L * i$ bytes. (Remember, each memory cell in our memory pictures is composed of four bytes.) If you examine the diagrams above, you can see this equation in action. For the very first array cell, `i` will be 0. In this case, the memory cell we want is the very first cell used for the array (i.e. the cell addressed by **a44**), since the first cell of the array will naturally be located at the start of the array and thus the address of the first cell of the array will be the starting address of the entire array. If we look at the equation and fill in the appropriate values, we see that it works out.

```
starting address of x[0] = x   +   L   *   0
                         = a44 +   4   *   0
                         = a44 + 0
                         = a44
```

If, on the other hand, we want, say, the array cell with index 3, well, that will be the fourth array cell, since the array cells with indices 0, 1, and 2 come before it. Therefore, it is necessary to skip over the first three array cells so that we can reach the fourth array cell. Each of these array cells we want to skip over takes up L bytes, because that is the size of the type. So, since we need L bytes to implement each array cell, if we want to skip over three of them we will be skipping over $3 * L$ bytes. In the case of our example array `x` which holds six integers, each integer takes one memory cell, i.e. four bytes. So, L will be 4 in our example array. It follows that if we want the starting address of the fourth array cell (i.e. the cell with index 3), we should start at the beginning of the array and then skip down 12 bytes, since $3 * 4bytes = 12bytes$. This corresponds with the formula we have above.

```
starting address of x[3] = x + L * i              // here i = 3
                         = a44 +  4 bytes  *  3
                         = a44 +  12 bytes
                         = a56
```

And, if you look back at the memory diagrams, you can see that if we want to reach the array cell `x[3]`, the address of that location is indeed 12 greater than the address of the start of the array. We started at the beginning of the array, and then skipped over three memory cells, i.e. 12 bytes, to reach `x[3]`.

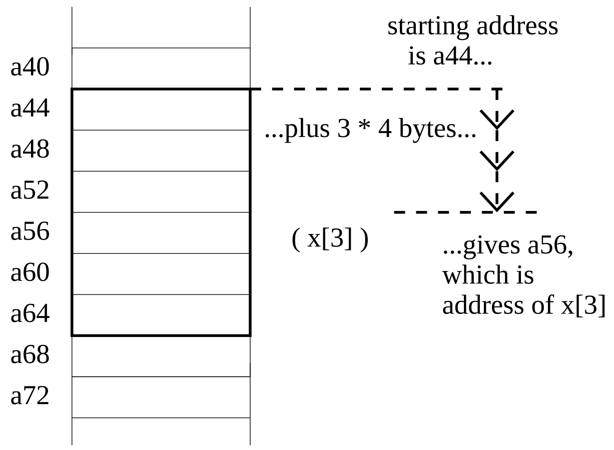
starting address is a44...
...plus 3 * 4 bytes...
...gives a56, which is address of x[3]
(x[3])
a40
a44
a48
a52
a56
a60
a64
a68
a72

Figure 1.3: Calculation of the address of the cell with index 3.

If our array x had been holding `Coord` objects instead of integers, the calculation would work the same way. A `Coord` object took up four memory cells, which is 16 bytes. So, for the type `Coord`, L is 16.

```
Coord c[6];
```

Now, if we want to access `c[2]`, the calculation works as follows:

```
starting address of c[2] = c + L * i            // here i = 2
                         = a44 +  16 bytes  *  2
                         = a44 +  32 bytes
                         = a76
```

And, in the diagram on the next page, you can see that `c[2]` does indeed start at **a76**.

So, whenever you use an array, the compiler always translates your

```
arrayName[index]
```

code to the internal calculation

```
arrayName + index * type_size_in_bytes
```

and then once that multiplication is done, you have the starting address of the cell you want. That is how arrays are implemented!

1.3 The relationship between pointers and arrays

In C++, due to the pointer construct, there is a very close relationship between pointers and arrays. This is because pointers are able to hold memory addresses. Since, as we saw in the previous section, an array cell location is simply described as an offset from the starting location of the array, we can use a pointer to store the starting address of an array and add an offset to the pointer's address value to get a different array location. In this manner we can actually

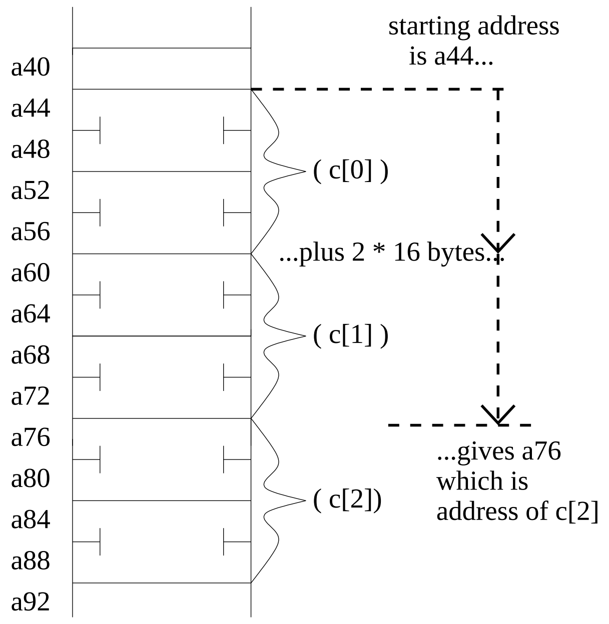

Figure 1.4: Calculation of the address of the cell with index 2.

calculate array cell locations naturally in our code just as they are calculated by the system after being translated from the more common `x[i]` type of format. This general idea of the calculation of address offsets in code – an idea known as *pointer arithmetic* – isn't used too often in C++ anymore, but it is still helpful to know, firstly for the sake of gaining a better understanding of memory issues, and secondly so that our usage of dynamic arrays will make more sense.

So, first consider our example array `x` from the last section, the array which held six integers. One question that might come to mind is, "what is `x`?". Meaning, we know that `x[0]`, `x[1]`, etc. refer to specific memory locations, but what about just plain `x`? Is `x` the name of a memory location?

The answer is slightly confusing. `x` is indeed a name, but there is no variable `x`. That is, while variables have names, values, and memory locations, our `x` is a name, and it refers to a value (namely **a44**, the starting address of the array), but it does not have a location in memory, or at least, not one that can be written to. The name `x` holds a constant value, cannot be changed, and for our purposes can be thought of as nothing more than a name in the compiler symbol table (see the pointer packet, section 2) and nothing more.

In other words, `x` is not an *lvalue*. The term "lvalue" more or less means "left-hand side value", or "something you can write a value *to* in an assignment." The idea of "something that is in memory" is a more accurate description, because if you can write to it, then it must be stored in memory. For example, if we have the code

```
int a, b;
```

then you can write statements such as

```
b = 2;
```

```
a = b;
a = 5;
2 = b;      // yes, this is an error
```

In the first statement, you are writing 2 into the location named by `b`. So, since `b` is being written into, it is an *lvalue*. This term came about because assignment statements are always organized so that the location being written into is placed on the left ("lvalue" is shorter than "left-hand-side value"). In the second statement, we are writing the value of `b` into `a`. Here, `b` is not being written into; instead, we are reading it rather than writing into it. This statement shows that `a` is an lvalue, and the third statement shows this as well. The idea is that variables are sometimes written into, and sometimes they are on the right-hand side instead, and are read from instead of written into. This is only true for variables, however. Values, such as 2, 3.4, or 'e', cannot be written into and thus are not lvalues. This can be seen in the fourth statement above, which makes no sense and is not allowed.

So, in usage, we say that `a` and `b` are lvalues, because they *can* be used on the left-hand side of an assignment. They aren't always used that way; the usage of `b` in second statement above is one example where an lvalue would not be used on the left-hand side of an assignment. But, since you *could* use it that way, `b` (as in, just `b`, just considering `b` alone without placing it in any specific statement) is considered an lvalue. So, the expressions that are *not* lvalues are those that do not refer to any specific memory cells, and thus could never be written to (you can't write to 2.3 or 'e', for example).

The point of this discussion is that `x`, our array name, is not an lvalue – it cannot be written to in the same way that a variable can (and thus we are not going to include it in our memory diagrams). This is good, because we don't *want* to assign to it. The last two lines in the following segment of code are somewhat non-sensical:

```
int x[6];
int n;
x[0] = 5;
x[1] = 2;
x = a48;     // not allowed!
x = &n;      // not allowed!
```

We know that `x` holds an address – the starting address of the array. And yet, to assign to `x` makes no sense – we don't *want* to suddenly change the starting address of the array (which would require copying all the array elements, i.e. moving the contiguous chunk of memory being used by the array to the new locationm), even if we *could*. And even if we *wanted* to do that, we *really* don't want to change that starting location to that of an already existing variable, as we are doing in the very last assignment above – because that would mean writing over the values that are already at that memory location.

So, you should think of the array name (the name on its own, not with the brackets and an index attached) as simply a marker, a stand-in for the starting address of the array, just as the symbol "2" is a stand-in for the bit-string representing "2" in the machine arithmetic. You can read from `x` and get the starting address of the array, but you can't write to `x`.

However, since you *can* read from `x`, you can write the following code:

```
int* startPtr;
startPtr = x;
```

This code will first create the integer pointer `startPtr`, and then it will assign to this pointer the address represented by `x`, that is, the starting address of the array. In other words, the above code accomplishes *exactly* the same thing as writing:

```
int* startPtr;
startPtr = &(x[0]);
// the above line could also have been written as
//    startPtr = &x[0];
// i.e. the parenthesis are not needed here
```

In either case, we end up with an integer pointer variable `startPtr` which holds **a44**, the starting address of our array.

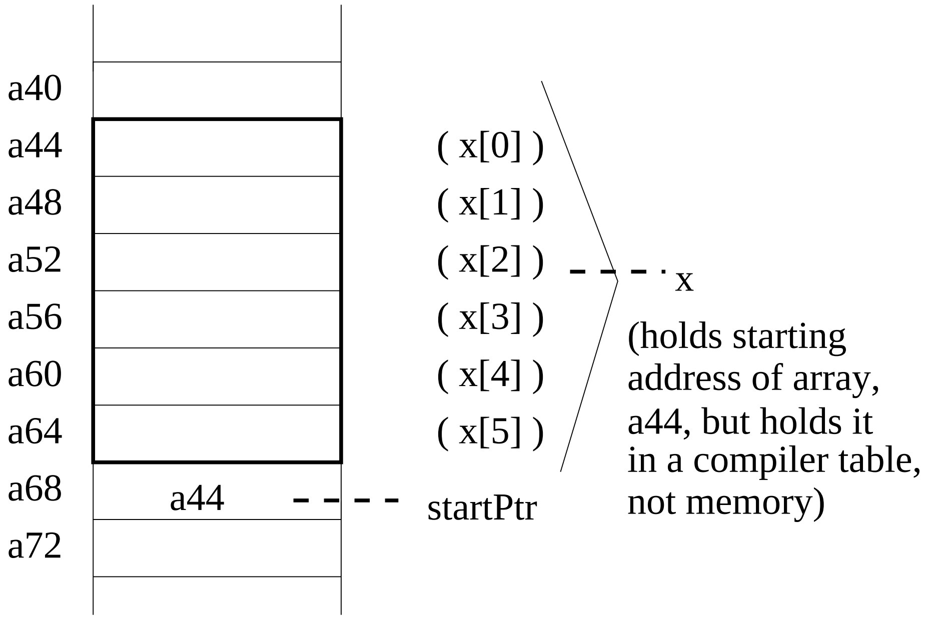

Figure 1.5: Pointer variable stored in memory.

Now, here is where it gets interesting. If you add an integer and a pointer, the result is another address. Specifically, since the pointer has a type it points to (for example, integer pointers point to integers – i.e. hold the addresses of integers), the integer is automatically multiplied by the size of that pointed-to type, and then the resultant product is read as a total number of bytes and is added to the address.

Huh? :-)

An example will explain it better. Consider the following line of code, which comes after the declaration and initialization of `startPtr` that we had above:

```
int n;
n = *(startPtr + 3);
```

What happens in that second line? Well, `startPtr` is a pointer to an integer. So, when you try and add the 3 to this pointer, the pointer reads the 3 as "$3 * sizeOfIntegerInBytes$". And since an integer needs 4 bytes in our example machine, the expression

```
   startPtr + 3
```

is effectly read as

```
   startPtr + (3 * 4 bytes)
```

This leads to

```
   n = *(startPtr + 3);

   // really means...

   n = *(startPtr + 3 * 4 bytes);

   // really means

   n = *(startPtr + 12 bytes);

   // since startPtr holds a44, this really means

   n = *(the address a56)

   // dereferencing...

   n = the integer held in the memory cell a56
```

That is, the line

```
   n = *(startPtr + 3);
```

is *exactly* equivalent to

```
   n = x[3];
```

Likewise,

```
   (startPtr + 3)  is equivalent to  &x[3]

and

   *(startPtr + 3) = 5;

is equivalent to

   x[3] = 5;
```

Basically, adding an integer to a pointer is the equivalent of putting the brackets after the pointer and placing the integer inside. This also means that you *can*, in fact, put the brackets after `startPtr` as well. That is, once `startPtr` has been assigned the value of `x`, you can use `startPtr` *in place of* `x`.

```
startPtr[3] = 5;
cout << startPtr[2];
total = startPtr[1] + startPtr[5];
```

So, the name of an array simply is a "stand-in" for the address of that array, and so once you have another pointer storing that array's address, you can use the name of that pointer variable in place of the name of the array.

1.4 Strings in stack memory

In C++, a built-in string is simply an array of type `char`. The only odd thing you need to know is that strings end with a NULL character, to signify that they have indeed ended. (Many string functions traverse down the `char` array until a NULL character is read, indicating you've reached the end of the string. This allows string functions to operate without needing to know the size of the string in advance.) This NULL character is represented with a backslash and 0 (`\0`). And of course, just like any other `char` value, such as `'e'` or `'M'` or `'7'`, if you want to specifically refer to the NULL character in your code, you need to enclose it in quotes (`'\0'`).

So, if we wanted to store the string "CS225" in C++, here is one way we could do it:

```
char stringA[6];   // creates an array indexed from 0 to 5
stringA[0] = 'C';
stringA[1] = 'S';
stringA[2] = '2';
stringA[3] = '2';
stringA[4] = '5';
stringA[5] = '\0';  // do not forget this character!
```

If you forget to add the NULL character to the end, later processing may not work, since a string function might read right off the end of the array and into other memory. This will eventually result in memory corruption, or memory access errors (segmentation faults or bus errors), or both. The NULL character acts as a "stop sign", letting functions know they have reached the end of the string.

You can imagine that trying to assign large strings to arrays of type `char` in this manner would be tedious and error-prone. So, there is a short-hand for initializing the `char` array. (There are actually other array initialization shorthands as well, that work in general. You can read about these things in a C++ manual, but it's a minor detail and so we won't worry about covering it here.) The following code will accomplish for `stringB` what the above code accomplishes for `stringA`:

```
char stringB[] = "CS225";
```

When you use the line above, the system encounters it and thinks,

1. Oops, the user has not filled in the size of this array. Oh, wait! It's not a problem because the array is being initialized on this exact same line, and so I can determine the size of the array from that.

2. Okay, there is an expression in double-quotes here; by definition, that is a string of characters. There are five of them, and so since I also need room for the NULL character, the total space the array `stringB` needs is space for six `char` values.

3. So, now that the memory has been set aside for those six `char` cells, I will copy the C, S, 2, 2, and 5 into the first five of those cells, and then the NULL character into the sixth cell. Done!

Or, in other words, that one line is a second way of initializing a string. You can specifically allocate all the `char` cells you need and fill them in one-by-one, or else just use the single line above, which indicates only that you want a `char` array and what string you want to store in it. Given those two pieces of information, the system can figure out how much space is needed and can do the assignment of characters to character cells, so you might as well let the machine do it and save yourself some typing.

1.5 Arrays and strings in dynamic memory

Finally, it is time to discuss dynamic arrays. Once you understand section 3.3, it is really not too difficult, simply because you are already used to the idea of some pointer variable serving in place of the array name. That is, you have already seen that the following code is perfectly legal.

```
int x[6];
int* startPtr = x;
startPtr[2] = 1;  // equivalent of ``x[2] = 1;''
```

So, you have seen that you can access a local array by using not its name, but rather a pointer that holds the array's starting address. Now, imagine a dynamically-allocated array. It doesn't have a name! So, there is no "real name" to refer to it by. The only way you can use a dynamic array is to assign a pointer to hold its starting address, and then that pointer serves as its name, just like `startPtr` is serving as an alternate name on the third line of the code snippet just above.

That is, when we perform our dynamic array declaration (as seen in section 3.1):

```
int* arrayPtr;
arrayPtr = new int[6];
```

all that happens is that `arrayPtr` is a local variable holding the address of the dynamic array, and the set of contiguous memory cells are still allocated for the array but are allocated from the heap instead of from the stack. Otherwise, there is *no* difference. So, essentially it is similar to the use of `startPtr` in section 3.3, except in this case the pointer and the address it holds are nowhere near each other, rather than being right next to each other as in the local memory example. The pointer variable `arrayPtr`, like the pointer variable `startPtr`, is in local memory. But the chunk of contiguous memory cells whose starting address is stored in `arrayPtr` is located in the heap, whereas for `startPtr` it is located in the stack. But, that is the *only* difference between the two. The usage is exactly the same, and works because we can substitute `arrayPtr` for the (in this case non-existant) *real* name of the array just as we could substitute `startPtr` for `x` in local memory.

```
arrayPtr[2] = 4;
arrayPtr[0] = 0;
cout << arrayPtr[2];
```

However, there is one thing you need to need to be aware of, which is that when memory is allocated in this manner, using `new` and the brackets together, then when you delete that memory, you need brackets there as well.

```
arrayPtr[1] = 6;
delete[] arrayPtr; // deletes dynamic array; arrayPtr still
                 //   holds address of former dynamic array
arrayPtr = NULL; // arrayPtr no longer holds its old value
```

These basic rules apply to strings as well – strings are just a specific case. When dealing with strings, since they are arrays of characters, the type that can hold the address of a dynamic string is a character pointer, or `char*`.

```
char* finalExample;
finalExample = new char[5];
finalExample[0] = 'B';
finalExample[1] = 'y';
delete[] finalExample;
```

Note the brackets at the end of `delete`, which are necessary because the allocation used brackets as well, to create many objects in an array arrangement. When `new` uses `[]`, `delete` must use `[]`. When `new` does not, `delete` should not.

There is one final thing to mention. Traditionally in C (the predecessor language of C++), the user dealt with strings of characters by passing character pointers (`char*` variables) around in memory. Likewise, the array syntax we have described above was frequently used as well. However, unlike in Java, there is no bounds-checking in C or C++. So, if your string or array has five cells and you try and access cell number 10, the best that you can hope for is that the program simply crashes, without the helpful "ArrayOutOfBoundsException" notice that you had in Java. (You can build code into your program to give you that notice, but it does not come automatically.) The worst that can happen is that the system actually *does* allow you to read the cell 10 cells away from the starting location of the array – even though that memory location isn't actually part of the array. By reading that cell, you are reading garbabe memory and corrupting your data!

So, when writing code that used arrays or strings, one had to be very careful not to write code that accessed a non-existent cell. As a result, in C++, arrays and `char*`-based strings are generally safely tucked away inside an `Array` or `String` class which *does* do things like range checking. The "real" array or string gets placed among the member data (i.e. instance variables) of the class, and the interface is designed to give you "array-like" access to that data. The resultant `Array` or `String` class – called a *wrapper* class because it wraps around the "real" array or string – can then be used in a manner far more similar to your use of other objects in the system. In the second discussion section, the TAs will go over the `Array` and `String` classes that we use in this course, and in addition to other important details about the code, they will explain how the "real" array or string gets used as member data by the wrapper class.