Lecture 8

Arrays And Dynamic Allocation Using Pointers

Instructor: Ashley Gannon

ISC3313 Fall 2021

Warm Up Activity

Activity

Using slide 24 from the last lecture *NamespacesVariableScopesControlStructuresFunctions*, write a function `division` that

- takes in two arguments,
- divides them,
- and returns a number.

Post your code in the discussion board **Division function** for participation credit. Your code doesn't have to be working, or even correct- **just make an honest effort**!

Arrays

Arrays

Up until now we have used variables to represent a single quantity. What if there are many quantities? An array is simply a means to store multiple values, to be accessed by a single variable name. The syntax is

```
type arrayname[dimension]
```

An example is:

```
int myarr[4];
myarr[0] = 4;
myarr[1] = 1;
myarr[2] = 8;
myarr[3] = 13;
```

Arrays

Arrays can also be initialized as

```
int myarr[4] = {4, 1, 8, 13}
```

The size of the array does not need to be specified if it is initialized this way. The compiler will figure it out. The alternative declaration is:

```
int myarr[] = {4, 1, 8, 13}
```

Indexing

We access the elements of an array using the index, which is the value that goes in the square brackets. The index must be an integer. We could also use a `for` loop to fill in the values:

```
int myarr[4];
for (int i = 0; i < 4; i++)
{
    cin » myarr[i];
}
```

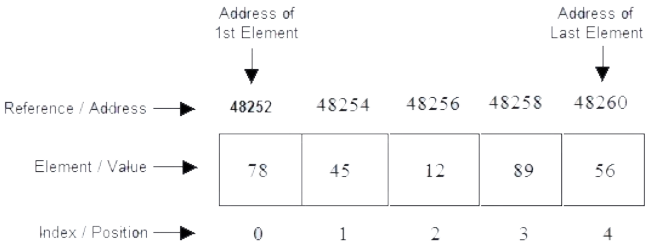
Here the variable `i` is the index.

Addresses in memory

Consider the array:

```
{78, 45, 12, 89, 56}
```

The elements of this array are stored in contiguous memory addresses. In this figure, the address increases by 2 due to the size of each integer element (2 bytes, 1 address space per byte).



Character arrays

One simple application of arrays is a string. A string is an array of type `char`. We could create a string as follows:

```
char mystring[20] = {'H','E','L','L','O','\0' };
```

Notice that at the end we make use of the null character `'\0'`, which specifies the end of a character array. We could also use `char mystring[20] = "Hello";`

This is a string literal and always has a null character (done automatically)

Out of bounds indexing

- `C++` does not perform array bounds checking for you

Out of bounds indexing

- `C++` does not perform array bounds checking for you
- This means that you may try to access an element which is outside of the array's bounds

Out of bounds indexing

- `C++` does not perform array bounds checking for you
- This means that you may try to access an element which is outside of the array's bounds
- This is an example of a runtime error, and it will likely cause the program to crash

Out of bounds indexing

- `C++` does not perform array bounds checking for you
- This means that you may try to access an element which is outside of the array's bounds
- This is an example of a runtime error, and it will likely cause the program to crash
- Even worse, it may *not* crash but instead overwrite other data in your program

Class activity

Write a program that asks the user to enter an 8-digit password. Your program should:

- Allow the user to type the password
- Store it in a `char` array
- Print the password back to the user

You should manually put the null character at the end of the array after the password is entered so that you can print the array with `cout`.

Post your code in the discussion board **Password Code** for participation credit.

Dynamic allocation with pointers

Static versus dynamic allocation

So far we have seen static arrays.

- Their size must be known (specified by you) at compile-time
- Their size cannot be determined once the program begins executing

The size of a dynamic array can be determined at run-time. These are far more useful. However, they require knowledge of *pointers*!

Pointers

Pointers are a type of variable whose value is actually a memory address (of another variable). We saw how to get the address of a variable using referencing. Recall the example:

```
int b = 5;
int&c = b;
b = 3;
cout « c « endl;
```

There is a difference between using `&` in the declaration (left of equal sign) and with the variable itself (right of equal sign).

Address-of operator

If we use the `&` symbol next to the variable, it acts as an 'address-of' operator:

```
int a = 5;
cout « &a « endl;
```

This would just print the address given for `a` in memory. Not exactly useful, yet.

Pointers and address-of

The pointer is used to store the address of a variable. We can declare a pointer using `*` as follows:

```
int* p;
```

Then we can set it to the *address* of any variable:

```
int a = 5;
p = &a;
```

Dereferencing

Dereferencing is how you find out what *value* is at the *address* pointed to by a pointer. We again make use of the ⋆ symbol, but again, it has a different meaning then in a declaration. Consider the following code:

```
int a = 5;
int* p = &a;
cout « *p « endl;
```

Dereferencing

We can also use dereferencing to *change* the value at the address pointed to by the pointer. consider the same example as before:

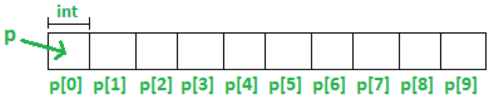
```
int a = 5;
int* p = &a;
*p = 3;
cout << a << endl;
```

Dynamic arrays with pointers

Pointers are very useful when working with arrays. To create an array with a variable size we would do the following:

```
int n = 10;
int* p = new int[n];
```

The `new` operator is another special keyword that allocates *memory* for an array of integers. Not the same thing as a static array. YOU are responsible for this memory!



Dynamic arrays with pointers

Once you create this memory you can access it using the same syntax as a static array:

```
int n = 10;
int* p = new int[n];
for (int i = 0; i < 10; i++)
{ p[i] = i;
} cout « p[5] « endl;
```

Note that if you do `*p` now, it is the value of the first entry (`p[0]`). The pointer always points to the first element in memory.