

CS 213 -- Lecture #5

“Late Night Guide to C++”
Chapter 4 pages 86 - 104
ARRAYS

Administrative...

- Assignment #2 due today
- Office Hours finalized -- 351 Upson
 - Wed 4PM - 5:30PM
 - More office hours if necessary
 - You can always make an appointment with me via email (rd29@cornell.edu)

Arrays

- What is an Array?
 - An array is a “chunk” of memory which contains consecutive instances of a given data type.
 - Think of it as a “list” of data, each data item of the same data type.
 - An array is declared by declaring a variable of a given type and then suffixing the declaration with a [n] where n is the number of elements you want in your array.

```
int iArray[8]; // declares an 8 element array
```

- At this point, iArray might look like this (in memory):

iArray

??	??	??	??	??	??	??	??
----	----	----	----	----	----	----	----

- Memory is allocated, but not initialized (hence the ??'s)

Accessing Array Elements

- To get at the contents of one of the array elements...
 - take the variable used to declare the array and suffix it with a left square bracket, the element number you wish to retrieve, followed by a right square bracket. This expression will evaluate to the value in the array at the specified index. C++ starts numbering array elements at 0, not 1.
- WARNING! C++ does no bounds checking on array accesses.
- Let's take a look...

```
int iArray[8]; // declares an 8 element array
for (int k=0; k<8; k++) // arbitrary initialization
    iArray[k] = k;
```

iArray

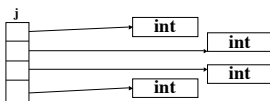
00	01	02	03	04	05	06	07
----	----	----	----	----	----	----	----

Pointers in Array Declarations

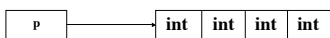
- What's the difference between...

```
int *j[4];
int (*p)[4];
```

- The first declaration is an array of 4 pointers to int



- The second is a pointer to an array of 4 integers



Static Initialization

- You can assign values to an array immediately right where they are declared...

```
int smallPrimes[7] = {2,3,5,7,11,13,17};
```

- The same can be done for a multiple dimension array:

```
int d[2][3] = {{0,1}, {1,0}, {1,1}};
```

- Let's see it in action...

Demonstration #1

Basic Arrays

Pointers and Arrays

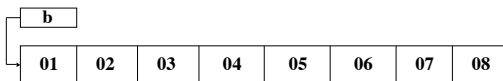
- Pointers and Arrays seem very closely related
 - Both seem to deal with accessing “chunks” of memory
 - Yet they both seem to be geared towards different tasks
 - Arrays are used for creating a list of elements of fixed length
 - Pointers are used for dynamically allocating data structures at runtime
- Well, in C++ an array is really just a pointer.
- Consider the following...

```
int main()
{
    int *a, b[8] = {1,2,3,4,5,6,7,8};
    a = b;
    cout << "a is " << *a << endl;
}
```

- What will be printed out as the value of *a?

Pointers and Arrays (cont)

- To better understand, consider a graphical representation of b:



- Now, since an array *is* a pointer, b actually *points* at its first element.
- That means that for any array, the following is true:

```
int main()
{
    int b[8] = {1,2,3,4,5,6,7,8};
    if (b == &b[0])
        cout << "This will always be true."
    return 0;
}
```

Demonstration #2

Sanity Check (Arrays as Pointers)

Pointer Arithmetic

- You might be wondering...
 - If **b* is the same as *b[0]* can I access other elements of *b* without using *[n]* notation?
- Yes.
- Actually, for any array *p* :
 - $p[n] == *(p+n)$
 - This is called *pointer arithmetic*
 - To add to the confusion, $p[n] == n[p]$ (because $*(p+n) == *(n+p)$)

```
int main()
{
    int b[8] = {1,2,3,4,5,6,7,8};
    cout << "b[1] = " << b[1] << endl;    // Prints out b[1]
    cout << "b[2] = " << 2[b] << endl;    // Prints out b[2]
    cout << "b[3] = " << *(b+3) << endl;  // Prints out b[3]
    return 0;
}
```

Back to Arrays...

- Since every array is a pointer, what do you suppose this does...

```
void swap(int A[8], int j, int k)
{
    int temp = A[j];
    A[j] = A[k];
    A[k] = temp;
}

int main()
{
    int b[8] = {1,2,3,4,5,6,7,8};
    swap(b,2,3);
    cout << "b[2]= " << b[2] << " and b[3]= " << b[3] << endl;
    return 0;
}
```

- Let's check it out...

Demonstration #3

Pointers as Parameters

Pointers As Parameters

- Since every array is passed by pointer it has the same effect as being “passed by reference”.
- Remember, C++ does no bounds checking.

```
int main()
{
    int a1[8];
    int a2[20];
    int a3[5];
    swap(a1,2,7); // a1 is the right size
    swap(a2,2,7); // a2 is too big
    swap(a3,2,7); // a3 is too small, there's no a3[7]
}
```

- These are all “legal”... Why?
- Remember, an array is just a pointer. That's why.

Dynamic Allocation of Arrays

- Yes, an array can be dynamically allocated. But you won't use:

```
int a1[8] = new int; // WRONG!
```

- Remember, when you use the [n] notation in a declaration you are actually allocating memory at that point.
- Remember also that an array is just a pointer.
- When dynamically allocating space for an array, you will be receiving a pointer back...

```
int *a = new int[8]; // RIGHT!
```

- The [8] tells the new operator to allocate an array of 8 ints.
- How do you delete such a dynamic allocation?

```
delete [] a; // Must use this, delete a is undefined
```

Dynamic Allocation of Arrays (cont)

- What's nice about this method of dynamic allocation is that the size of the array does not need to be known at compile time.
- Consider the following:

```
int main()
{
    Course *courses;
    int numCourses;
    cout << "How many courses to enter? ";
    cin >> numCourses;
    courses = new Course[numCourses];
    // Rest of program
    delete [] courses;
}
```

- Let's see this actually work...

Demonstration #4

Dynamic Allocation of Arrays

That Nasty Scope Thing Again

- As always, there are dangers...

```
int *MakeArray() // Will compile, but I wouldn't advise it
{
    int iArray[50];
    return iArray;
}

int *MakeArray(int size) // Will compile, and is safe
{
    int *anArray = new int[size];
    if (anArray == NULL)
        cout << "Couldn't allocate array of size " << size << endl;
    return anArray;
}
```

Final Thoughts

- Assignment #3 is posted!
- Due 9/16