5/9/2019 CS 16 Lecture

Const keyword

Const is a modifier for types. Const is not a type on its own. (e.g. use const like : int const **or** const char **or** double const **or** const float (notice order doesn’t matter)).

Can always convert a non-const to a const.

Pointer arithmetic

Recall from the last lecture that pointer arithmetic is used with contiguous memory (arrays are contiguous memory).

So if we make a pointer point to the zeroth element of an array, we can use pointer arithmetic to move around the array.

Sample file:

#include <iostream>

#include <string>

using namespace std;

int **main**()

{

 int \*point;

 int arr[] = {123,234,345,456,567,678,789};

 point = arr; *// Recall that the name of an array is a pointer to the zeroth element*

 cout << "Here is the 0th element of the array using \*(point+0) : " << \*(point + 0) << "\n\n";

 cout << "Here is the 3rd element of the array using \*(point+3) : " << \*(point + 3) << "\n\n";

 cout << "We can also use pointer arithmetic to loop over an array!\n\n";

 for(int \*myP = arr;(myP - arr) < 7; ++myP){ *// (myP - arr) < 7 becasue there are only 7 elements in arr*

 *// myP - arr gives you the index you are at*

 cout << "Element " << myP - arr << " is " << \*myP << endl;

 }

 return 0;

}

Output:

Here is the 0th element of the array using \*(point+0) : 123

Here is the 3rd element of the array using \*(point+3) : 456

We can also use pointer arithmetic to loop over an array!

Element 0 is 123

Element 1 is 234

Element 2 is 345

Element 3 is 456

Element 4 is 567

Element 5 is 678

Element 6 is 789

New keyword

We can dynamically allocate memory using the new keyword.

The new keyword puts things on the “heap”. Before today, we have been putting variables on the stack.

Basically, we create variables on the heap to be like global variables .

Delete keyword

We use this when we are done using a variable or array that we originally allocated in the heap.

As a general rule you should have a delete keyword for each new keyword used.

Memory Leak

A memory leak is when you allocate memory on the heap and then lose access to that piece of memory. This happens when you allocate memory and then no longer have a pointer to the piece of memory before you have deleted it.

Dangling Pointer

When you return a pointer that was allocated on the stack from a function.

Sample of dangling pointer: (when we return from makeArrayOfSize5() the array declared in the function is recalled, and not used how you want it to)

#include <iostream>

#include <string>

using namespace std;

int\* **makeArrayOfSize5**(){

 int myArr[5] = {1,2,3,4,5};

 int \*p = myArr; *// Recall that the name of an array is a pointer to myArr[0]*

 return p;

}

int **main**()

{

 int \*point = **makeArrayOfSize5**();

 for( int i = 0; i < 5; ++i){

 cout << "Element "<< i << " = " << \*(point + i) << "\n";

 }

 cout << endl << endl;

 return 0;

}

Output: (we can see from the output that the values of myArr were erased once we were back in main)

Element 0 = 20053757

Element 1 = 1

Element 2 = -1511921808

Element 3 = 32767

Element 4 = 20053757

// End output

^^^^^^^^^^^^^^^^^^^

The above is not what we wanted (1,2,3,4,5)

New keyword example

Sample file: (the above file done the right way)

#include <iostream>

#include <string>

using namespace std;

int\* **makeArrayOfSize5**(){

 int\* myArr = new int[5];

 *// \*myArr = {1,2,3,4,5}; won't work*

 for(int i = 0; i < 5; i++){

 myArr[i] = i + 1;

 }

 int \*p = myArr; *// Recall that the name of an array is a pointer to myArr[0]*

 return p;

}

int **main**()

{

 int \*point = **makeArrayOfSize5**();

 for( int i = 0; i < 5; ++i){

 cout << "Element "<< i << " = " << \*(point + i) << "\n";

 }

 cout << endl << endl;

 delete [] point; *// delete [] (which means the whole array) point (which is the array name)*

 return 0;

}

Output:

Element 0 = 1

Element 1 = 2

Element 2 = 3

Element 3 = 4

Element 4 = 5

// This is what we wanted!