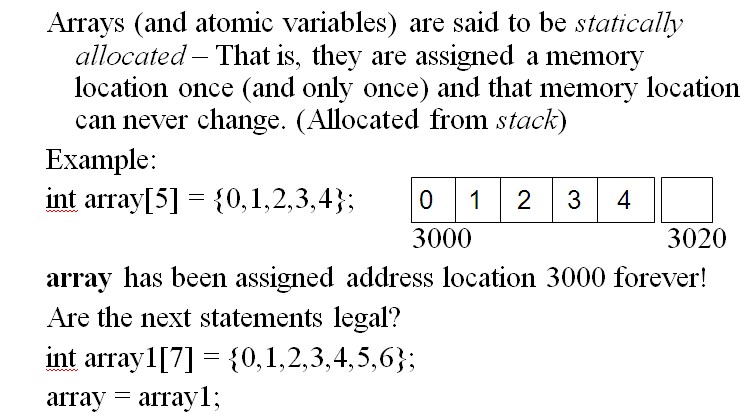
**Chapter 9**

**Pointers - Dynamic Memory Allocation**

**What is Static Memory Allocation**

All variables declared in our programs up to this point have been statically allocated by the compiler (before the program runs)

****

Do we always know how much memory is needed by our program? Do we always know the appropriate size for our array?

Can I do something like this instead?

int no\_of\_test\_scores = 0;

cout << “Enter no of test scores: “;

cin >> no\_of\_test\_scores;

int test\_scores[ no\_of\_test\_scores];

What is the problem here?

**Introduction to Dynamic Memory Allocation**

* Dynamic memory allocation is used for a allocating **exactly as much memory** as you need (when you need it)
* Used for creating variables/data structures at **run-time**.
* Only pointer variables can be assigned to a dynamically allocated memory location
* Use the *new* operator for dynamic memory allocation
* **Once an address has been allocated dynamically by *new*, it can never be used again until it is freed by using *delete*!**
* Dynamic memory allocation more complicated because *programmer* *must manage the memory*.

**What is the *new* operator?**

new is used for dynamically allocating memory (i.e., variables) at run time. Its most basic syntax is:

new dataType;

new always returns an address of where the variable is allocated. To allocate a single integer

use:

new int;

Not useful unless you store the address. Must declare pointer variable to store the address:

int\* ptr;

ptr = new int;

Generally don’t use new to allocate single variables. Usually use new to allocate arrays. The syntax for dynamically allocating 1-d array is:

int \*test\_scores;

test\_scores = new int[no\_of\_test\_scores];

Placed in context, here is how we can dynamically allocate array:

cout << “Enter no of test scores: “;

cin >> no\_of\_test\_scores;

int \*test\_scores = new int[no\_of\_test\_scores];

for (int ii = 0; ii < no\_of\_test\_scores; ii++)

{

cout << “Enter test score # ii: “ ;

cin >> test\_scores[ii]

Dynamic memory allocation requires a pointer variable. One of the most important uses is to dynamically allocate a 1-d array.

**Steps For Dynamically Allocating 1-d array of integers**

1. Declare Pointer Variable (int\* ptr; int\* ptr1)

2. Use new to allocate the array (new int[no\_ele))

Store the address returned by new in the pointer variable:

ptr = new int[no\_ele]; //array of ints with no\_ele elements

ptr1 = new int; //allocates one integer off the heap

3. Initialize the data at the dynamic heap address (Just like you

would with a statically declared array)

ptr[0] = 9;

ptr[1] = 22;

......

\*ptr1 = 8;

ptr1[0] = 23; //Can only use index 0!

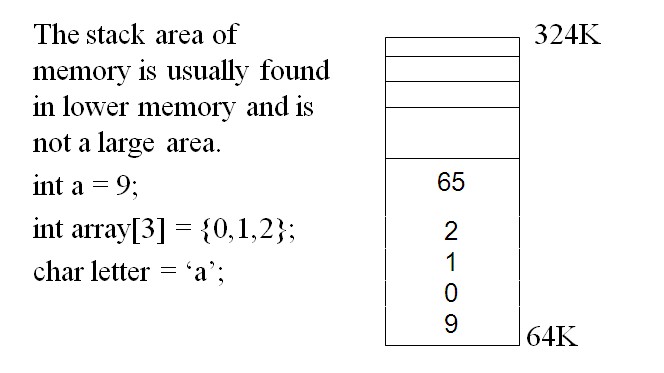
4. cleanup memory when done

delete[] ptr; //Use [] when cleaning up an allocated array

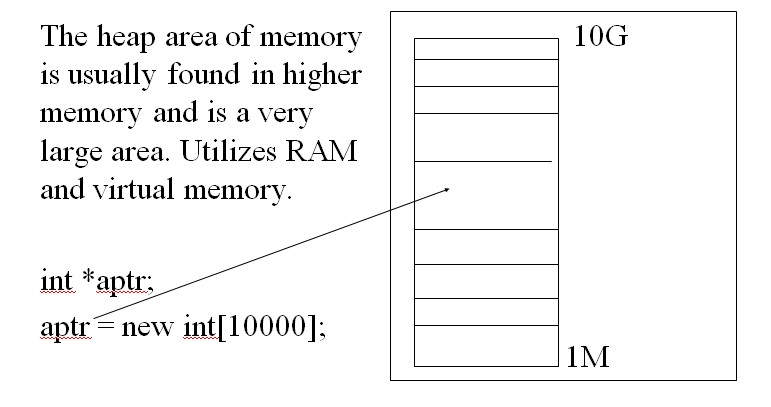
delete ptr1; //When deleting a non-array\*/

**Difference between Stack and Heap**

Statically allocated memory (i.e., memory allocated by the compiler by declaring arrays, variables, structures, classes) is allocated from a section of memory called the *stack*. The size of the stack is approximately 1 MB on most computers.

****

All dynamically allocated pointers are allocated from the *heap*. The *heap* is a massive amount of memory - much larger than the stack.



**Returning Pointers from Functions**

Consider the following program:

#include <iostream>

using namespace std;

int\* getData(int no\_ele);

int main()

{

//Variable Declaration/Initialization

int i = 0;

int no\_elements = 0;

int\* data = NULL;

//Prompt the user for # of elements to process

cout << "Enter number of elements: ";

cin >> no\_elements;

//Get the data as an array

data = getData(no\_elements);

//Display all data

for (i = 0; i < no\_elements; i++)

{

//Display each element

cout << data[i] << " ";

}

cout << endl;

//Cleanup memory

delete[] data;

return 0;

}

int\* getData(int no\_ele)

{

//Variable Declaration/Initialization

int i = 0;

const int MAX\_ELE = 500;

int array[MAX\_ELE] = {0};

//Iterate thru each element

for (i = 0; i < no\_ele; i++)

{

cout << "Enter element # " << (i+1) << ": ";

cin >> array[i];

}

return(array);

}

This statement generates the following warning message:

warning C4172: returning address of local variable or temporary

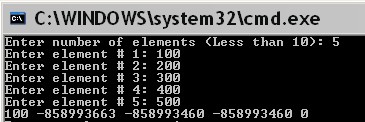
Why this warning?

* Automatic variables are allocated in the stack area of memory
* The stack area is very temporary. Each time a function call is performed, many addresses of the stack are overwritten by the automatic variables of the newly called function!
* The stack is an area of memory which contains:
  + All locally declared variables pertaining to the function (object) currently being performed
* The stack is constantly changing! Each time a function is called, the stack will reflect the variables needed by the newly called function.

Where are local variables allocated?

* Local variables (automatic variables) are created on the stack.
* As soon as the variable goes out of scope (a function call), the variable can be overwritten by the local variables of the function being called.
* Local variables **cannot** maintain their values between function calls!

What happens when running this program:



Why is this displayed?

The reason this occurs is because the stack address pertaining to *array* is re-used in main. The data is overwritten/destroyed.

**Only returns pointers that are dynamically allocated**

Correct the program as follows (within getData function):

int\* getData(int no\_ele)

{

//Variable Declaration/Initialization

int i = 0;

//Dynamically allocate array

int\* array = new int[no\_ele];

//Iterate thru each element

for (i = 0; i < no\_ele; i++)

{

cout << "Enter element # " << (i+1) << ": ";

cin >> array[i];

}

return(array);

}

Re-running this program results in the following:

