

Lecture 4

DYNAMIC MEMORY ALLOCATION

Memory Division in C++ Program

- Memory in C++ program is divided into two parts
 - **The Stack**
 - All variables declared inside the function will take up memory from the stack
 - Variables cannot be resized
 - Normally stack memory is of limited size
 - **The Heap**
 - This is unused memory of the program and can be used to allocate large block of memory dynamically while your program executes

- Programmer deallocate memory when not needed anymore



- It is the dynamic memory that can grow or shrink as per our need. Also called the free storage. The size of all other segments is decided at compile time but heap can grow during runtime. We can control memory allocation and de-allocation in heap space.
- It is the space where the local variables gets space. The variables which are declared within functions live in stack.

Dynamic Memory Allocation

- Dynamically allocated memory is kept on the memory

heap (also known as the free store)

- Dynamically allocated memory does not have a "name" or it cannot be accessed using a regular variable, instead it must be referred to (or ***pointed to by the pointer***) ▪ the ***new*** operator is used to dynamically allocate memory

Dynamic Memory Allocation (DMA)

- You can allocate memory dynamically (as our programs are running) • and assign the address of this memory to a pointer variable.

```
int *ptr = new int;
```

?

new and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements –

```
double* pvalue = NULL; // Pointer initialized with null  
pvalue = new double; // Request memory for the variable
```

ptr dynamic variable

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the 'delete' operator as follows –

```
delete pvalue; // Release memory pointed to by pvalue
```

Dereferencing

- We have learned how to set up a pointer variable to point to another variable or to point to memory dynamically allocated.
- But, how do we access that memory to set or use its value?
- By dereferencing our pointer variable:

```
*ptr = 10;
```

Dereferencing

- Now a complete sequence:

```
int *ptr;  
  
ptr = new int;  
  
*ptr = 10;  
  
••• //some code  
  
cout <<*ptr; //displays 10
```

10

ptr dynamic variable

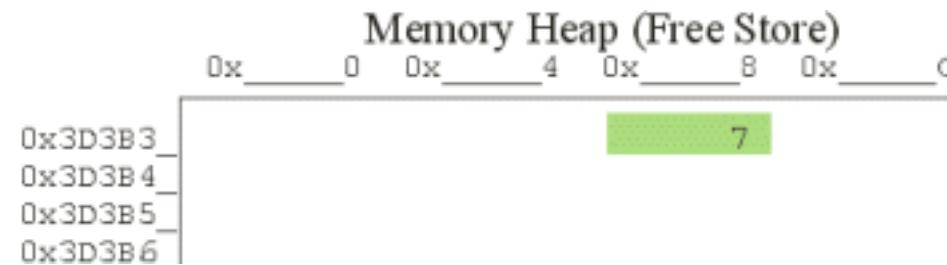
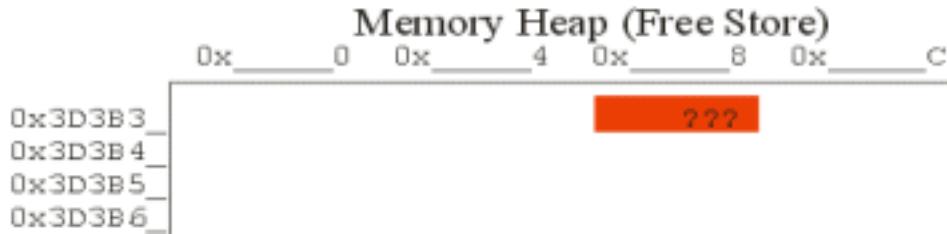
Example: Pointing to Memory Allocated

```
int *ptr;
```

```
ptr = new int;
```

```
*ptr = 7;
```





Deallocating Memory

- Once done with dynamic memory,
 - we must deallocate it
 - C++ does not require systems to do “garbage collection” at the end of a program’s execution!
- We can do this using the `delete` operator:

```
delete ptr;
```

- this does not delete the pointer variable!

- It only de-allocates the memory.

11

Returning Memory to the Heap

- The Opposite of **new**:
 - The **delete** operator.
- How to use it:

```
int *ptr;  
ptr = new int;  
*ptr = 7;  
delete ptr;
```

DMA: Another Example

```
#include <iostream>
using namespace std;

int main () {
    double* pvalue = NULL; // Pointer initialized with null
    pvalue = new double;   // Request memory for the variable

    *pvalue = 29494.99;    // Store value at allocated address
    cout << "Value of pvalue : " << *pvalue << endl;

    delete pvalue;         // free up the memory.

    return 0;
}
```

DYNAMICALLY ALLOCATING 1D ARRAYS

Dynamic Arrays

- Two approaches are used if you cannot even guess the array size.
 - to declare an array that is *large enough* to process a variety of data sets.
 - during program execution, you could prompt the user to enter the size of the array and then create an array of the appropriate size.
- Pointers help in creating arrays during program execution and process.
- An array created during the execution of a program is called a **dynamic Array**.

Dynamically allocating 1D Array

- Allocate block of memory: new operator is also used to allocate a block(an array) of memory of type *data-type*.

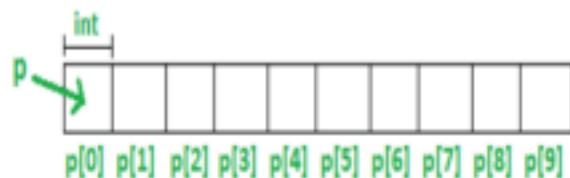
```
pointer-variable = new data-type[size];
```

where size(a variable) specifies the number of elements in an array.

Example:

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

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

Dynamic Memory Allocation to 1D Arrays

- To allocate an array dynamically, we use the array form of new and delete (often called new[] and delete[]):

```
cout<<"Enter a positive integer: ";\n\nint length;\n\ncin>>length;\n\nint *array = new int[length];
```

To remove the array we just created we will use the delete[] operator

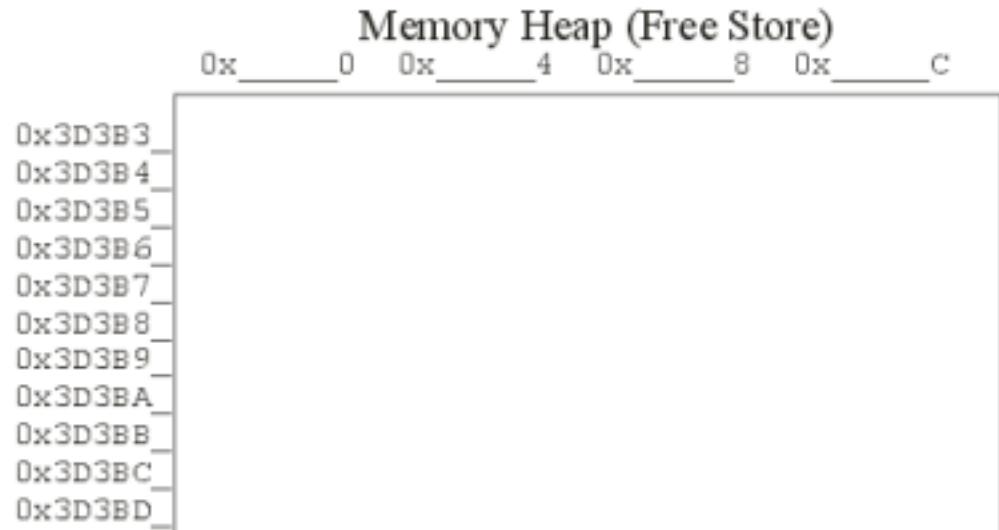
```
delete[] array;
```

Example: Pointing to

Memory Allocated at Run Time

Time

```
int *a;  
a = new int[3];  
*a = 300;  
*(a+1) = 301;  
*(a+2) = 302;
```



Using array notations to access elements

- After performing a few arithmetic operations, it is possible to lose track of the first array component.
- C++ allows us to use array notation to access these memory locations. For example, the statements:

```
a [0] = 300;  
a [1] = 301;  
a [2] = 302;
```

- a[0] refers to the first array component, a[1] refers to the second array component and so on.

Returning Memory to the Heap for dynamically allocated 1d arrays

- For Arrays
 - `delete[] a; // delete, or deallocate memory`

- Using **delete []**, the memory pointed by pointer **a** is deallocated and is returned to the OS. OS is now free to assign that memory block to another program/process.

Dynamically allocating int array:

this is operator **new []** and it is different from operator **new**.

Allocate dynamically an integer

1452

array :

```
int *q;           int[4];      q[i] = i*2;      0
q = new          for (int       1452 q            2 4 6
                   i=0;i<4;i++)
```

De-allocate the dynamically allocated integer array :

```
145
2
delete [] q;
q
```

this is operator delete[] and it is
different from operator delete.

1452

NOTE : it is a good practice to set
the pointer to NULL after the
de-allocation

Dangling Pointers

1. The delete operator does not delete the pointer, it takes the memory being pointed to and returns it to the heap 2. It does not even change the contents of the pointer 3. Since the memory being pointed to is no longer available (and may even be given to another application), such a **pointer is said to be dangling.**
4. In other words, a ***dangling pointer*** is a pointer to memory that your program does not own or a pointer that the program should not use.

Scenario 1:

```
int *ptr = new int; *ptr = 10;  
...//code
```

Dangling Pointers

Scenario 2:

```
int *ptr1 = new int;  
*ptr1 = 10;  
  
int *ptr2;  
  
delete ptr; //ptr is dangling  
pointer  
...//code  
  
//dereferencing ptr may result  
in unpredictable behavior
```

Remedy: undangling the pointer

```
int *ptr = new int;  
*ptr = 10;  
...//code
```

```
delete ptr;  
ptr = NULL;  
Ptr2 = ptr1;  
...//code  
  
delete ptr2; //ptr1 and ptr2 are
```

dangling pointers

...//code

//dereferencing ptr1 may result in
unpredictable behavior

Memory Leak

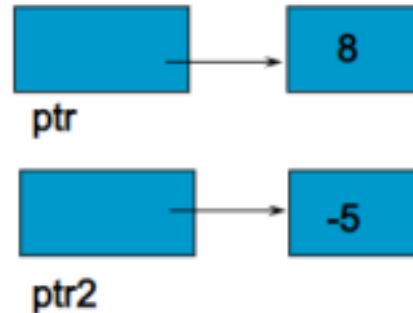
- Memory leak happens when the dynamically allocated memory is not returned back to the operating system after using it.
- In other words, when there is a memory area in a heap but no variable in the stack pointing to that memory.
- A memory leak reduces the performance of the computer by reducing the amount of available memory.
- Memory leak usually happens when programmer forgets to call **delete**.

Memory Leak Scenario

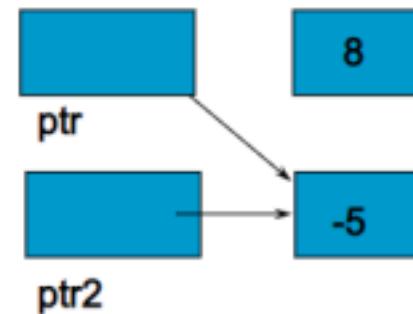
- Pointer ptr is set to ptr2 without releasing the memory pointed by the ptr.
- Therefore, memory that was initially pointed by the ptr is now leaked and will not be used by the program any more

Causing a Memory Leak

```
int* ptr = new int;  
*ptr = 8;  
  
int* ptr2 = new int;  
*ptr2 = -5;
```



```
ptr = ptr2; // here the 8 becomes inaccessible
```



34

Dangling Pointer and Memory Leak

Scenario • There are two problems in the below scenarios.

- There is a memory leak (memory having contents 8) as well as the dangling pointer (ptr).

Leaving a Dangling Pointer

```
int* ptr = new int;  
*ptr = 8;
```



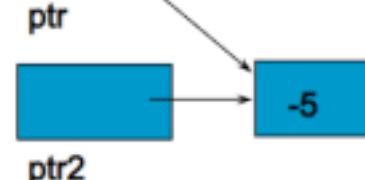
```
int* ptr2 = new int;
```

```
*ptr2 = -5;
```

```
ptr = ptr2;
```

```
delete ptr2;      // ptr is left dangling
```

```
ptr2 = NULL;
```



Class activity

- Suppose that we have a dynamically allocated array with 4 entries and we now need an array containing 8 entries where we want to keep the first 4 entries.

```
int main() {  
    int array_size = 4;  
    int * array = new int[array_size];  
    for ( int i = 0; i < array_size; ++i ) {  
        array[i] = i*i; }  
/* Start Resizing the Array */  
//write your code here  
/* Finish Resizing the Array */  
    // continue using the new array...  
    for ( int i = array_size/2; i < array_size; ++i ) {  
        array[i] = i*i; }  
    cout << "The squares are:";  
    for ( int i = 0; i < array_size; ++i ) {  
        cout << " " << array[i]; }  
    cout << endl;  
    // clean up...
```

```
    delete [] array; }

int main() {
    int array_size = 4;
    int * array = new int[array_size];
    for ( int i = 0; i < array_size; ++i ) {
        array[i] = i*i;
    }
    /* Start Resizing the Array */
    // 1. we require a temporary pointer
    // 2. allocate twice as much memory
    // 3. copy over the old values
    // 4. delete the old array
    // 5. assign the temporary pointer to the
    // original // 6. set the temporary pointer to NULL
    array_size *= 2;
    /*Finish Resizing the Array */
    // continue using the new array...
    for ( int i = array_size/2; i < array_size; ++i ) {
        array[i] = i*i;
    }
    cout << "The squares are:";
    for ( int i = 0; i < array_size; ++i ) {
```

```
        cout << " " << array[i]; }

cout << endl;
// clean up...
delete [] array; }

int main() {
    int array_size = 4;
    int * array = new int[array_size];
    for ( int i = 0; i < array_size; ++i ) {
        array[i] = i*i; }

/* Start Resizing the Array */
// 1. we require a temporary pointer
int * array_tmp = NULL;
// 2. allocate twice as much memory
array_tmp = new int[2 * array_size];
// 3. copy over the old values
for ( int i = 0; i < array_size; ++i ) {
    array_tmp[i] = array[i]; }

// 4. delete the old array
delete [] array;
array = NULL;
```

```
// 5. assign the temporary pointer to the original  
array = array_tmp;  
// 6. set the temporary pointer to NULL  
array_tmp = NULL;  
// 7. update the size of the array  
array_size *= 2;  
/*Finish Resizing the Array */  
// continue using the new array...
```

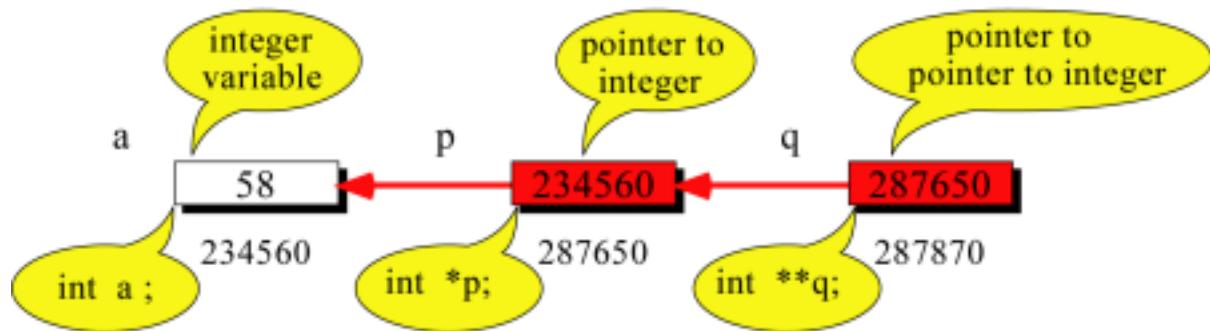
DYNAMICALLY ALLOCATING 2D ARRAYS

- There are two important concepts to understand before we proceed to dynamically allocate 2D arrays:
 - **First:** Pointer to Pointer
 - **Second:** Dynamically allocating array of pointers

First thing: Pointer to Pointer

- A pointer to a pointer
“Holds the address of another pointer.”
- It is declared using double asterisk, int **ptr

```
// Local Declarations
int      a ;
int      *p ;
int      **q ;
```



```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<      a << " ";
cout <<      *p << " ";
cout <<      **q << " ";
```

What is the output?

58 58 58

Pointer to Pointer

- A pointer to a pointer works just like a normal pointer — you can perform indirection (dereferencing) through it to retrieve the value pointed to.
- And because that value is itself a pointer, you can perform indirection through it again to get to the underlying value.
- These indirections can be done consecutively, For example:

```
int value = 5;
```

```
int *ptr = &value;  
cout << *ptr; // Indirection through pointer to int to get int
```

value

pointer to int, second indirection to get int value

```
int **ptrptr = &ptr;
```

```
cout << **ptrptr; // first indirection to get
```

The program prints: 5 5

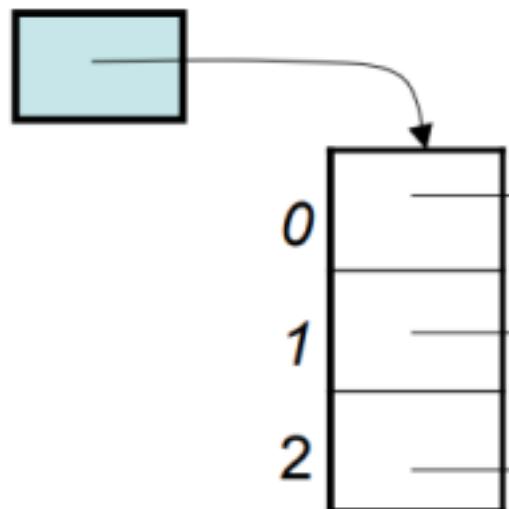
2nd thing: Array of Pointers

- Pointers to pointers have a few uses.
- The most common use is to dynamically allocate an array of pointers and make a **pointer to pointer** to point to it:

```
int **array;
```

```
array = new int*[3]; // allocate an array of 3 int pointers
```

- This works just like a standard dynamically allocated array, except the array elements are of type “***pointer to integer***” instead of integer
- All the three cells will be used to store the addresses of the 1D array.



Array

How to dynamically allocate 2D Arrays

- **Basic Idea:**

1. Allocate an array of pointers (first dimension), → just like we did in previous slide

```
int **array = new int*[row];
```

2. Make each pointer point to a 1D array of the appropriate size (2nd Dimension)

```
array[0] = new int[columns]; // these are four columns of our dynamically allocated 2d-array.  
array[1] = new int[columns];  
array[2] = new int[columns];
```

- Now each row has 4 columns.

Let's put it together

- Our dynamic two-dimensional array is a dynamic one-dimensional array of dynamic one-dimensional arrays!

```
int rows = 5;
```

```
int cols = 4;
```

```
int **ptr_2d = new int*[rows]; // allocate an array of 5 int pointers - these are our rows
```

```
for (int i = 0; i < rows; ++i)
```

```
ptr_2d[count] = new int[cols]; // these are our columns
```

0 1 2 3

- We can assign values to 2d-array using array[3][2] = 3;

0
1
A
2
3
4

Assigning values to Dynamically Allocated 2D Arrays

- Values can be assigned to the resultant 2D array using regular

syntax. ▪ We can then access our array like usual:

```
Ptr_2d[3][2] = 3; // This is the same as (array[3])[2] = 3
```

0 1 2 3

		0
		1
A	4	
	2 3	

Memory Deallocation

- Deallocating a dynamically allocated 2d array using this method requires a loop as well, i.e,
 - Each row must be deleted individually

- Be careful to delete each row before deleting the array pointer. **Step 1:**

```
for(int i=0; i<6; i++)  
    delete [ ] ptr_2d[i];
```

Step 2:

```
delete [ ] ptr_2d;
```

- **Note** that we delete the array in the opposite order that we created it (elements first, then the array itself).