

Lecture 4

2D Arrays using And Dynamic Memory Allocation (DMA)

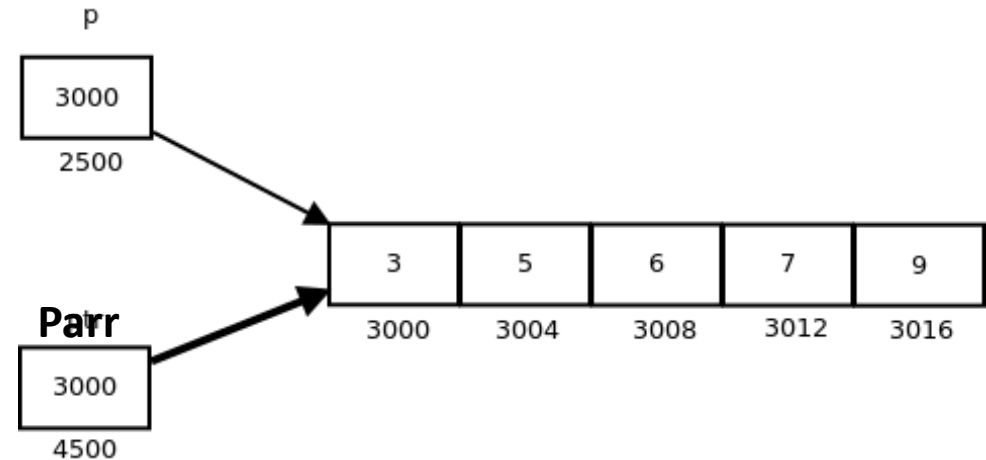
Original author of the sides is Jawad Hassan, FAST-NUCES, Islamabad but slides have been modified time by time as per the requirements of the class.

Program

A pointer that points to the 0th element of an array and a pointer that points to the whole array are totally different. Let's see an example:

```
#include<iostream>

int main() {
    int *p; // pointer to int
    int (*parr)[5]; // pointer to an array of 5 integers
    int my_arr[5]; // an array of 5 integers
    p = my_arr;
    parr = &my_arr;
    cout<<"Address of p \n"<< p);
    cout<<"Address of parr \n", parr );
    cout<<"Address of parr \n", *parr );
    p++;
    parr++;
    cout<<"\nAfter incrementing p and parr by 1 \n\n";
    cout<<"Address of p \n"<< p);
    cout<<"Address of parr \n", parr );
    return 0; }
```



```
Address of p = 2293296
```

```
Address of parr = 2293296
```

```
After incrementing p and parr by 1
```

```
Address of p = 2293300
```

```
Address of parr = 2293316
```

How it works?

- Here p is a pointer which points to the 0th element of the array my_arr.
- While parr is a pointer which points to the whole array my_arr.
- The **base type** of p is of type (int *) or pointer to int.
- The **base type** of parr is pointer to an array of 5 integers.
- Since the pointer arithmetic is performed relative to the base type of the pointer,
 - that's why parr is incremented by 20 bytes i.e (5 x 4 = 20 bytes).
 - On the other hand, p is incremented by 4 bytes only.
- Remember:
 - Whenever a pointer to an array is dereferenced we get the address (or base address) of the array to which it points.

Types of Pointers

- By declaring `int *ptr = my_arr; //where my_arr is 1-d array.`
- we have created a pointer which points to the 0th element of the array whose base type was `(int *)` or pointer to `int`.
- We can also create a pointer that can point to the whole array instead of only one element of the array.
- This is known as a pointer to an array

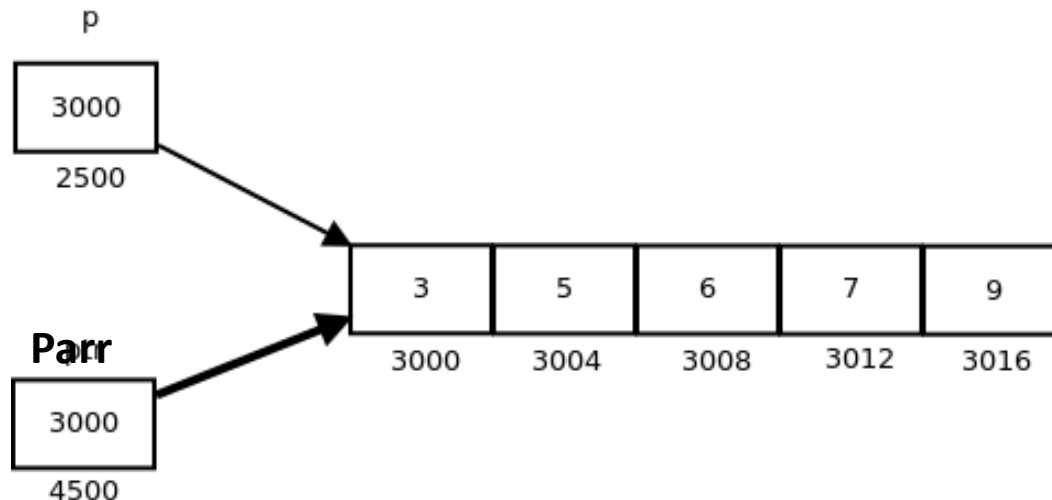
`int (*p)[10];`

`p` is a pointer that can point to an array of 10 integers.

- In this case, the type or base type of `p` is a pointer to an array of 10 integers.
- Note that parentheses around `p` are necessary, so you can't do this:
- In **`int *p[10];`** , `p` is an array of 10 integer pointers. (array of pointers will be discussed later.)

Dereferencing parr

- So, on dereferencing parr, you will get *parr.
- Although parr and *parr points to the same address:
 - but parr's base type is a pointer to an array of 5 integers,
 - while *parr base type is a pointer to int.
- This is an important concept and will be used to access the elements of a 2-D array.

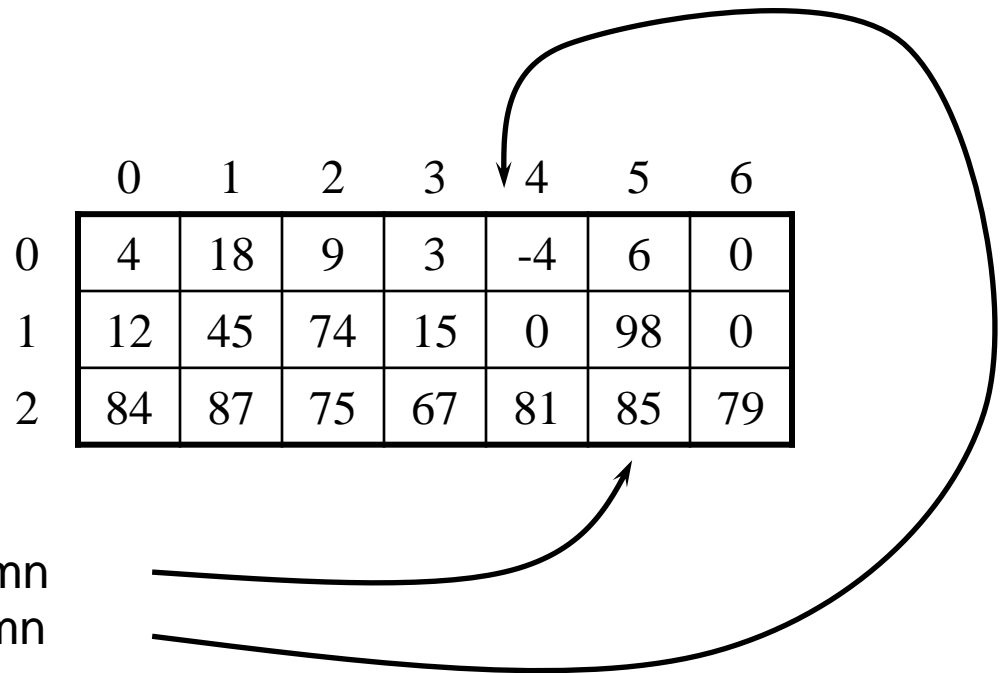


Multidimensional Arrays

C++ also allows an array to have more than one dimension.

For example, a two-dimensional array consists of a certain number of rows and columns:

```
const int NUMROWS = 3;  
const int NUMCOLS = 7;  
int Array[NUMROWS][NUMCOLS];
```



A 3x7 array is shown with row indices 0, 1, 2 on the left and column indices 0, 1, 2, 3, 4, 5, 6 on top. The array contains the following values:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|----|----|----|----|----|----|----|
| 0 | 4 | 18 | 9 | 3 | -4 | 6 | 0 |
| 1 | 12 | 45 | 74 | 15 | 0 | 98 | 0 |
| 2 | 84 | 87 | 75 | 67 | 81 | 85 | 79 |

Arrows indicate the following:

- An arrow from the text "3rd value in 6th column" points to the value 85 at row 2, column 5.
- An arrow from the text "1st value in 5th column" points to the value -4 at row 0, column 4.
- A curved arrow from the text "Array[2][5]" points to the value 85 at row 2, column 5.
- A curved arrow from the text "Array[0][4]" points to the value -4 at row 0, column 4.

Array[2][5]

3rd value in 6th column

Array[0][4]

1st value in 5th column

The declaration must specify the number of rows and the number of columns, and both must be constants.

Processing a 2-D Array

A one-dimensional array is usually processed via a for loop. Similarly, a two-dimensional array may be processed with a nested for loop:

```
for (int Row = 0; Row < NUMROWS; Row++) {  
    for (int Col = 0; Col < NUMCOLS; Col++) {  
        Array[Row][Col] = 0;  
    }  
}
```

Each pass through the inner for loop will initialize all the elements of the current row to 0.

The outer for loop drives the inner loop to process each of the array's rows.

Initializing in Declarations

```
int Array1[2][3] = { {1, 2, 3} , {4, 5, 6} };  
int Array2[2][3] = { 1, 2, 3, 4, 5 };  
int Array3[2][3] = { {1, 2} , {4} };
```

If we printed these arrays by rows, we would find the following initializations had taken place:

Rows of Array1:

```
1 2 3  
4 5 6
```

Rows of Array2:

```
1 2 3  
4 5 0
```

Rows of Array3:

```
1 2 0  
4 0 0
```

```
for (int row = 0; row < 2; row++) {  
    for (int col = 0; col < 3; col++) {  
        cout << setw(3)  
            << Array1[row][col];  
    }  
    cout << endl;  
}
```


2d array representation in memory

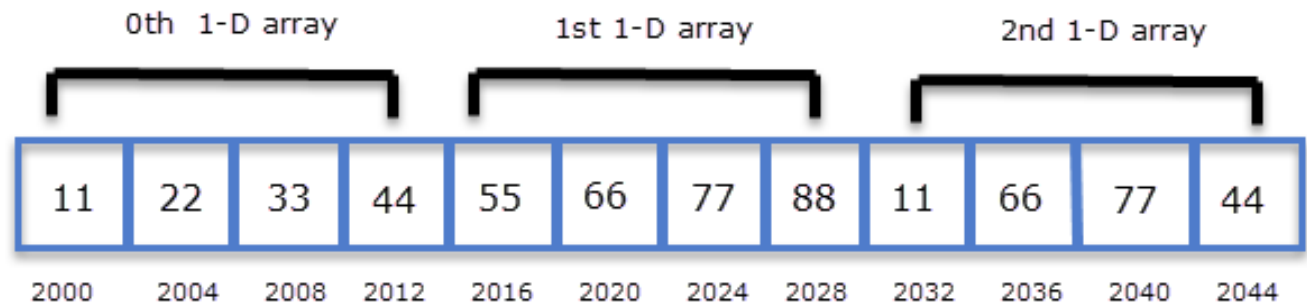
- 2d array declaration in c++:

```
int arr[3][4] = { {11,22,33,44}, {55,66,77,88}, {11,66,77,44} };
```

- Theoretical representation:

| | | | | |
|-------|----|----|----|----|
| Row 0 | 11 | 22 | 33 | 44 |
| Row 1 | 55 | 66 | 77 | 88 |
| Row 2 | 11 | 66 | 77 | 44 |

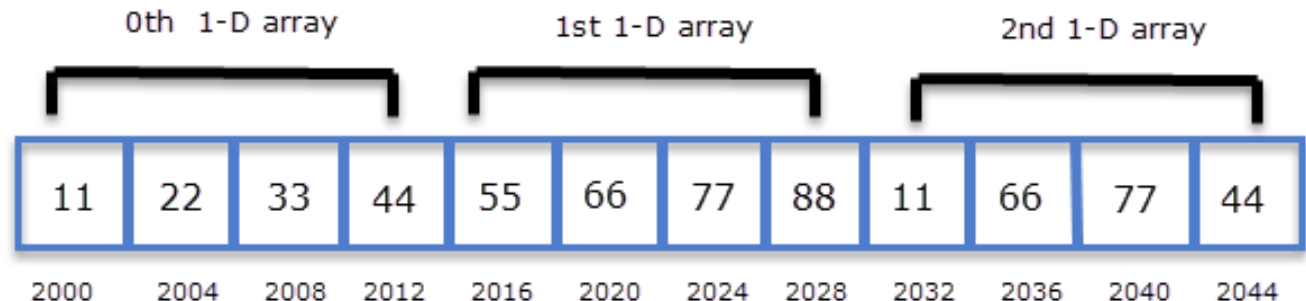
- Actual representation in memory:



A 2-D array is actually a 1-D array in which each element is itself a 1-D array. So arr is an array of 3 elements where each element is a 1-D array of 4 integers.

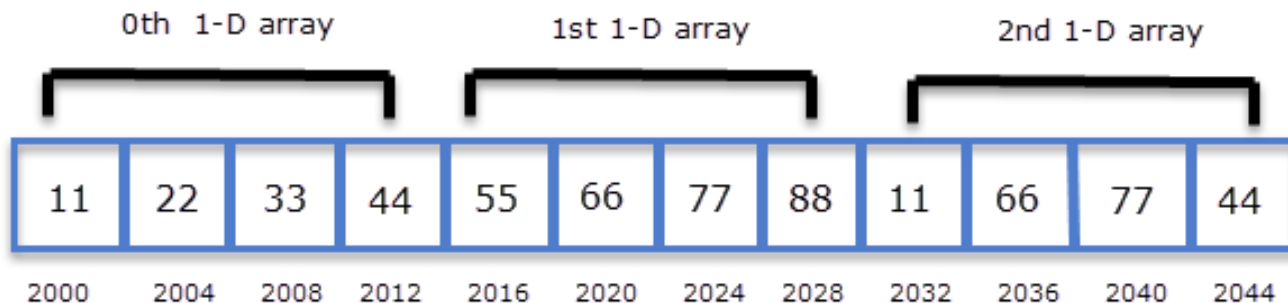
2d array representation in memory

- According to definition, the type or base type of **arr** is a pointer to an array of **4** integers.
- Since pointer arithmetic is performed relative to the base size of the pointer.
- if **arr** points to address **2000** then **arr + 1** points to address 2016
- Concluding:
 - arr** points to **0th** 1-D array.
 - (arr + 1)** points to **1st** 1-D array.
 - (arr + 2)** points to **2nd** 1-D array.
- More Generally,
 - (arr + i)** points to **ith** 1-D array.



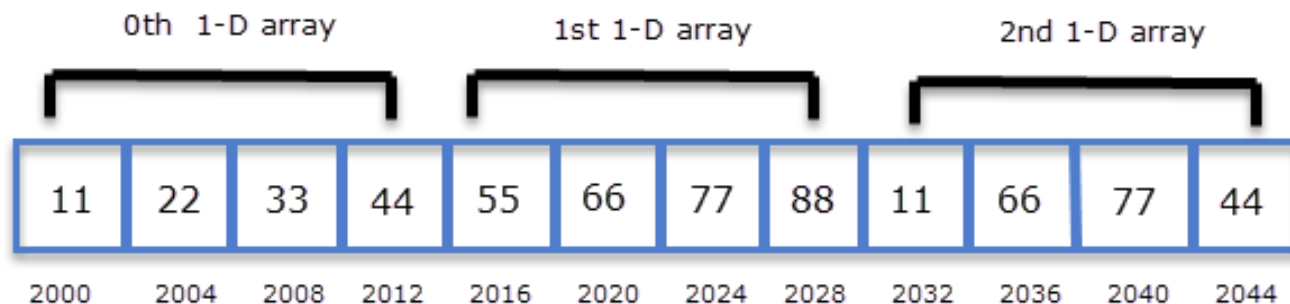
Dereferencing a pointer to array

- Dereferencing a pointer to an array gives the **base address** of the array.
- dereferencing **arr** we will get ***arr** → base type of *arr is (int*)
- Similarly, dereferencing **arr+1** we will get ***(arr+1)**
- Generally, ***(arr+i)** points to the base address of the **ith** 1-D array.
- **Note:** type **(arr + i)** and ***(arr+i)** points to same address but their base types are completely different.
 - base type of **(arr + i)** is a pointer to an **array of 4 integers (in our example)**
 - the base type of ***(arr + i)** is a pointer to **int** or (int*).



How to use **arr** to access individual elements of 2d array?

- From previous slides: **$*(arr + i)$** points to the base address of every *i*th 1-D array and it is of **base type pointer to int**
- Now let's use using pointer arithmetic
- $*(arr + i)$ points to the **address** of the 0th element of the 1-D array.
- $*(arr + i) + 1$ points to the **address** of the 1st element of the 1-D array
- $*(arr + i) + 2$ points to the **address** of the 2nd element of the 1-D array
- Generally, **$*(arr + i) + j$** points to the **base address** of ***j*th element of *i*th** 1-D array.
- After finding the base address, we can find the value by dereferencing the pointer to the base address, i.e., **$*(*(arr + i) + j)$**



Program

```
#include<iostream>
using namespace std;
int main() {
    int arr[3][4] = { {11,22,33,44}, {55,66,77,88}, {11,66,77,44} };
    int i, j;

    for(i = 0; i < 3; i++)
    {
        cout<<"address of "<<i<<"th array\t\t"<<*(arr + i)<<endl;
        for(j = 0; j < 4; j++)
        {
            cout<<"arr["<<i<<"]["<<j<<"] = "<< *( *(arr + i) + j)<<endl ;
        }
        cout<<"\n\n";
    }
    return 0; }
```

Address of 0 th array 2686736

arr[0][0]=11

arr[0][1]=22

arr[0][2]=33

arr[0][3]=44

Address of 1 th array 2686752

arr[1][0]=55

arr[1][1]=66

arr[1][2]=77

arr[1][3]=88

Address of 2 th array 2686768

arr[2][0]=11

arr[2][1]=66

arr[2][2]=77

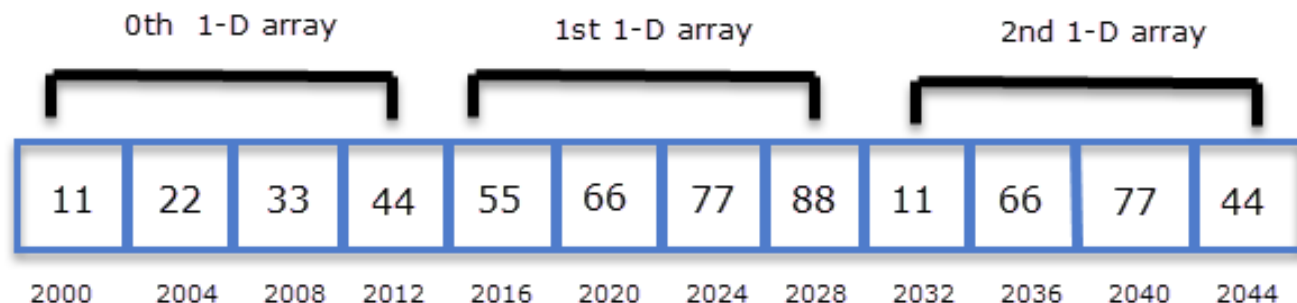
arr[2][3]=44

ASSIGNING 2-D ARRAY TO A POINTER VARIABLE

Assigning 2-D Array to a Pointer Variable

- Name of the array can be assigned to the pointer variable
- And then that pointer variable can be used to index through the array
- For 1d array, **pointer to int**, i.e., `int * ptr` is needed, but
- For 2d array, **pointer to array** is needed, i.e., `int (*ptr)[n]`.
- Recall the array declaration from the previous slides:

```
int arr[3][4] = { {11,22,33,44}, {55,66,77,88}, {11,66,77,44} };
```
- Remember a 2-D array is actually a 1-D array where each element is a 1-D array.
- So arr is an array of 3 elements where each element is a 1-D arr of 4 integers.
- To store the base address of **arr**, a pointer to an array of 4 integers is needed, `int (*p)[4]`;



Assigning 2-D Array to a Pointer Variable

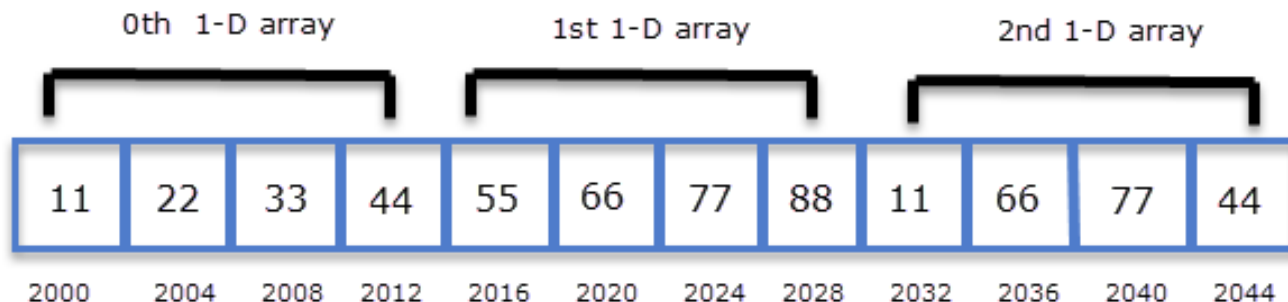
- Similarly, If a 2-D array has 4 rows and 5 cols i.e `int arr[4][5]`, then a pointer to an array of 5 integers is needed, `int (*p)[5]`;
- Let's continue from the previous example, **`int (*p)[4]`**;
`p = arr;` //We are considering the first element of the 1-d

Here `p` is a pointer to an array of 4 integers.

- According to pointer arithmetic, in other words,
`p+0` points to the 0th 1-D array,
`p+1` points to the 1st 1-D array and so on.

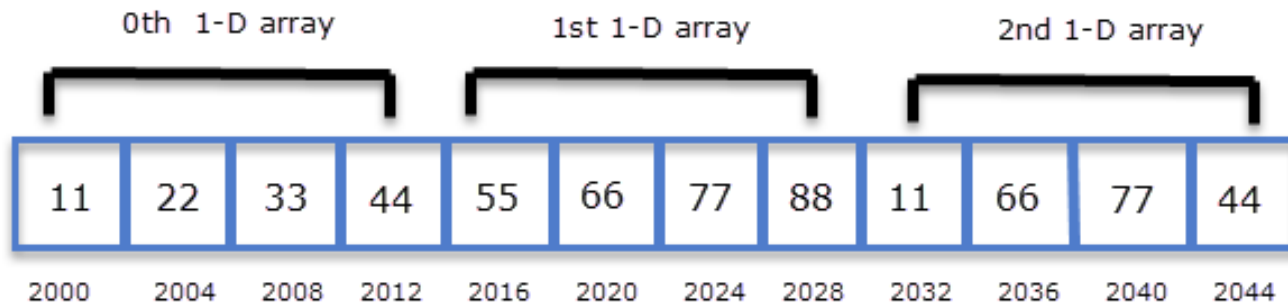
Generally, `p+i` points to the `i`th 1-D array

The base type of `(p+i)` is a pointer to an array of 3 integers.



Dereferencing $p+i$

- Dereferencing $(p+i)$, i.e., $*(p+i)$ gives the **base address** of i th 1-D array
- More precisely, base type of $*(p + i)$ is a pointer to int. or $(int *)$.
- To access the j th element of the i th 1-D array, we use $*(p+i)+j$
- So $*(p + i) + j$ points to the address of j th element of i th 1-D array.
- Now further, dereferencing the expression $*(p + i) + j$, i.e., $*(*(p + i) + j)$ gives the value of the j th element of the i th 1-D array.



Program: **Assigning 2-D Array to a Pointer Variable**

```
#include<iostream>

int main()
{
    int arr[3][4] = { {11,22,33,44},{55,66,77,88}, {11,66,77,44} };
    int i, j;
    int (*p)[4];    p = arr;
    for(i = 0; i < 3; i++)
    {
        cout<<"address of "<<i<<"th array\t\t"<<*(arr + i)<<endl;
        for(j = 0; j < 4; j++)
        {
            cout<<"arr["<<i<<"]["<<j<<"] = "<< *( *(arr + i) + j)<<endl ;

        }
        cout<<"\n\n";
    }
    // signal to operating system program ran fine
    return 0;
}
```

```
Address of 0 th array 2686736
arr[0][0]=11
arr[0][1]=22
arr[0][2]=33
arr[0][3]=44
```

```
Address of 1 th array 2686752
arr[1][0]=55
arr[1][1]=66
arr[1][2]=77
arr[1][3]=88
```

```
Address of 2 th array 2686768
arr[2][0]=11
arr[2][1]=66
arr[2][2]=77
arr[2][3]=44
```

DYNAMIC MEMORY ALLOCATION

Dynamic Memory Allocation

- *Used when space requirements are unknown at compile time.*
- *Most of the time the amount of space required is unknown at compile time.*
 - *For example consider the library database example, what about 201th book information.*
 - *Using static memory allocation it is impossible.*
- *Dynamic Memory Allocation (DMA):-*
 - *With Dynamic memory allocation we can allocate/delete memory (elements of an array) at runtime or execution time.*

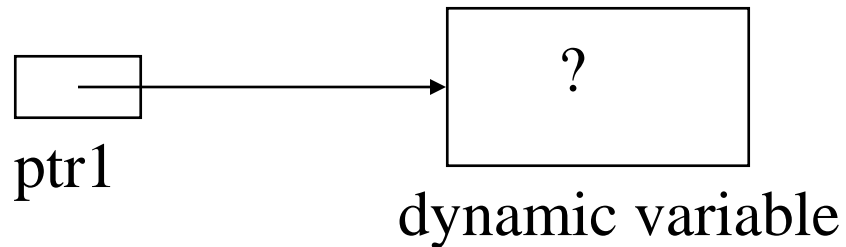
Differences between Static and Dynamic Memory Allocation

- Dynamically allocated memory is kept on the memory heap (also known as the free store)
- Dynamically allocated memory does not have a "name", it must be referred to
- ***Declarations*** are used to statically allocate memory, the ***new*** operator is used to dynamically allocate memory

Allocating Memory

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

```
int *ptr1 = new int;
```



```
int *ptr1 = new int;
```

- The diagram used is called a
 - pointer diagram
 - it helps to visualize what memory we have allocated and what our pointers are referencing
 - notice that the dynamic memory allocated is of size int in this case
 - and, its contents is uninitialized
 - new is an operator and supplies back an address of the memory set allocated

Dereferencing

- Ok, so 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:

```
*ptr1 = 10;
```


Dereferencing

- Now a complete sequence:

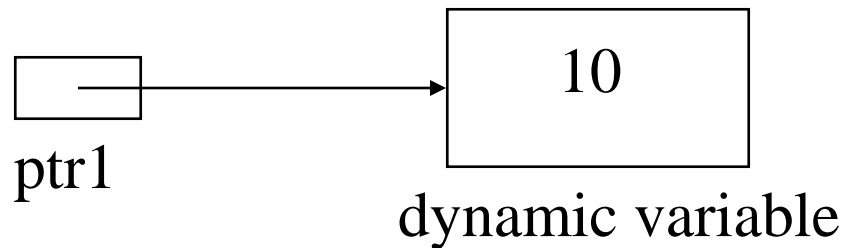
```
int *ptr1;
```

```
ptr1 = new int;
```

```
*ptr1 = 10;
```

```
...
```

```
cout <<*ptr1; //displays 10
```



Example: Pointing to Memory Allocated at Run Time

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

| Name Table | | | Address |
|------------|-------------|----------|----------|
| Name | Type | Contents | |
| ptr | int pointer | ??? | 0x22FF68 |

| Name Table | | | Address |
|------------|-------------|----------|----------|
| Name | Type | Contents | |
| ptr | int pointer | 0x3D3B38 | 0x22FF68 |

Memory Heap (Free Store)

0x____0 0x____4 0x____8 0x____C

0x3D3B3
0x3D3B4
0x3D3B5
0x3D3B6
0x3D3B7

B8
B9
BA
BB
BC
BD

Memory Heap (Free Store)

0x____0 0x____4 0x____8 0x____C

0x3D3B3
0x3D3B4
0x3D3B5
0x3D3B6
0x3D3B7
0x3D3B8
0x3D3B9
0x3D3BA
0x3D3BB
0x3D3BC
0x3D3BD

7

Memory Heap (Free Store)

0x____0 0x____4 0x____8 0x____C

0x3D3B3
0x3D3B4
0x3D3B5
0x3D3B6
0x3D3B7
0x3D3B8
0x3D3B9

???

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 ptr1;
```

- this does not delete the pointer variable!
- It only de-allocates the memory.

Deallocating Memory

- Again:

this does not delete the pointer variable!

- Instead, it deallocates the memory referenced by this pointer variable
 - It is a no-op if the pointer variable is NULL
 - It does not reset the pointer variable
 - It does not change the contents of memory
 - *Let's talk about the ramifications of this...*

Returning Memory to the Heap

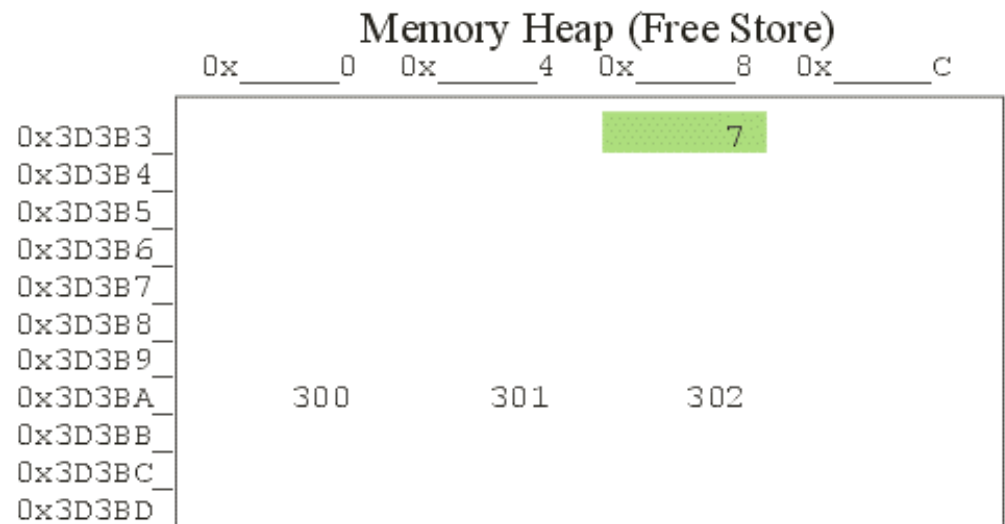
- Most applications request memory from the heap when they are running
- It is possible to run out of memory (you may even have gotten a message like "Running Low On Virtual Memory")
- So, it is important to return memory to the heap when you no longer need it

Returning Memory to the Heap

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

`delete ptr;`

| Name Table | | | |
|------------|-------------------|----------|----------|
| Name | Type | Contents | Address |
| ptr | int pointer | 0x3D3B38 | 0x22FF68 |
| a | int array pointer | 0x3D3BA0 | 0x22FF64 |



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.

Undangling the pointer

- How to undangle the pointer?
- `ptr = NULL;`

| Name Table | | | |
|------------|-------------------|----------|----------|
| Name | Type | Contents | Address |
| ptr | int pointer | 0x000000 | 0x22FF68 |
| a | int array pointer | 0x3D3BA0 | 0x22FF64 |

Returning Memory to the Heap

- Remember:
 - Return memory to the heap before undangling the pointer
- What's Wrong with the Following:
 - `ptr = NULL;`
 - `delete ptr;`

Allocating memory to Arrays Dynamically

- But, you may be wondering:
 - Why allocate an integer at run time (dynamically) rather than at compile time (statically)?
- The answer is that we have now learned the mechanics of how to allocate memory for a single integer.
- Now, let's apply this to arrays!

Allocating Arrays

- By allocating arrays dynamically,
 - we can wait until run time to determine what size the array should be
 - the array is still “fixed size”...but at least we can wait until run time to fix that size
 - this means the size of a dynamically allocated array can be a variable!!

Allocating Arrays

- First, let's remember what an array is:
 - the name of an array is **a constant address to the first element in the array**
 - So, saying

```
char name[21];
```

means that name is a constant pointer whose value is the address of the first character in a sequence of 21 characters

Allocating Arrays

- To dynamically allocate an array
 - we must define a pointer variable to contain an address of the element type
- For an array of characters we need a pointer to a char:

```
char *char_ptr;
```

- For an array of integers we need a pointer to an int:

```
int *int_ptr;
```

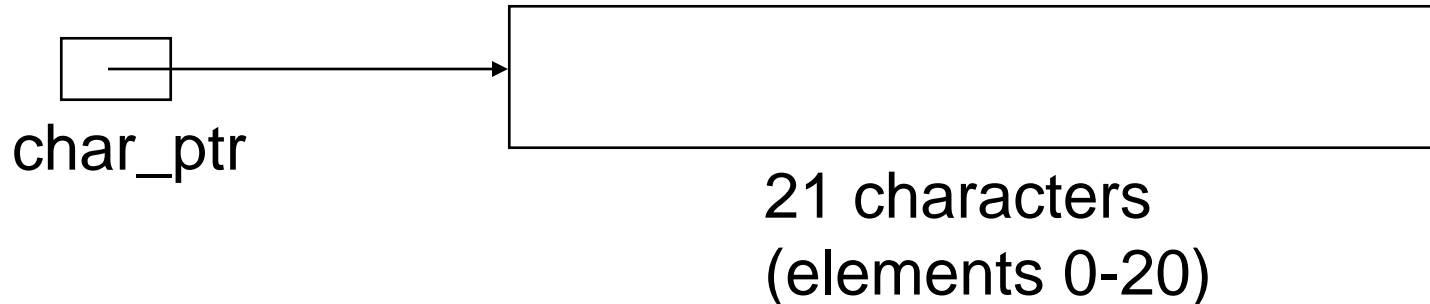
Allocating Arrays

- Next, we can allocate memory and examine the pointer diagram:

```
int size = 21; //for example
```

```
char *char_ptr;
```

```
char_ptr = new char [size];
```



Allocating Arrays

- Some interesting thoughts:
 - the pointer diagram is identical to the pointer diagram for the statically allocated array discussed earlier!
 - therefore, we can access the elements in the exact same way we do for any array:

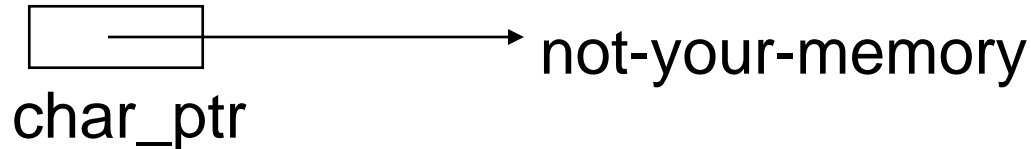
```
char_ptr[index] = 'a';    //or
```

```
cin.get(char_ptr,21,'\n');
```

Deallocating Arrays

- The only difference is when we are finally done with the array,
 - we must deallocate the memory:

```
delete [] char_ptr;
```

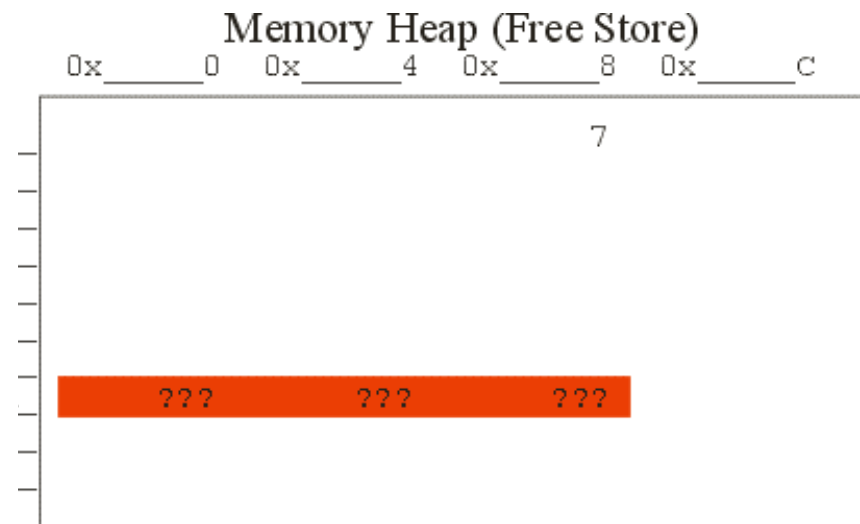
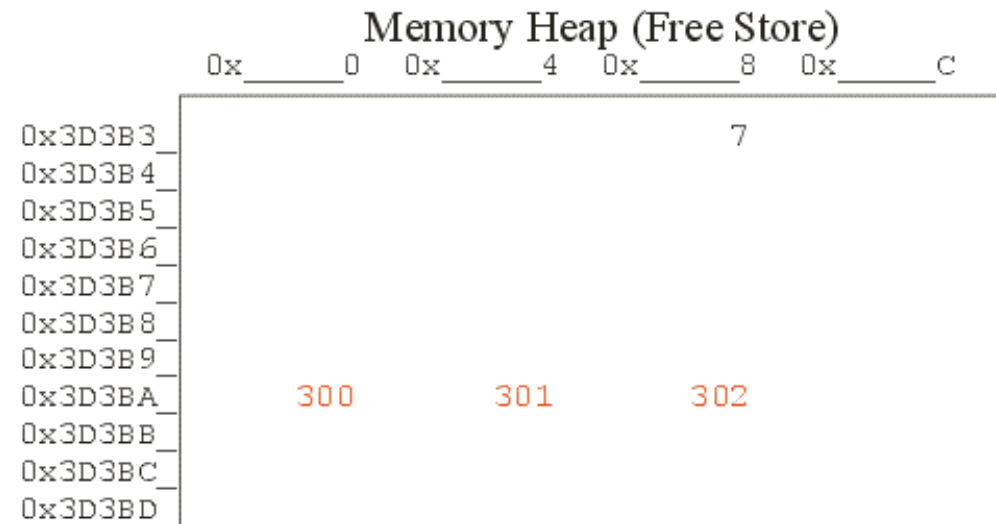
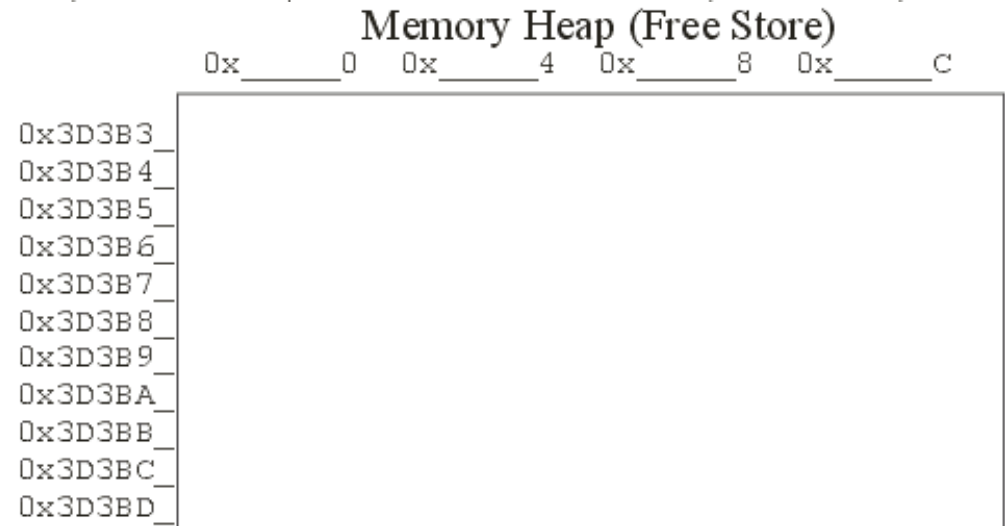


It is best, after doing this to say: `char_ptr = NULL;`

Example: Pointing to Memory Allocated at Run Time

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

| Name Table | | | |
|------------|-------------------|----------|----------|
| Name | Type | Contents | Address |
| ptr a | int pointer | 0x3D3B38 | 0x22FF68 |
| | int array pointer | 0x3D3BA0 | 0x22FF64 |



Returning Memory to the Heap for dynamically allocated arrays

- For Arrays
 - `delete[] a;` //(1) First delete, or deallocate memory
 - `a = NULL;` //(2) Then set the pointer to NULL

Memory Leaks

- **Memory *leaks*** when it is allocated from the heap using the new operator
but not returned to the heap using the delete operator

Memory Leak

- *int *otherptr;*
- *otherptr = new int;*
- **otherptr = 4;*
- *otherptr = new int;*
- *Now the previously allocated memory to otherptr is leaked.*

| Name Table | | | |
|------------|-------------------|----------|----------|
| Name | Type | Contents | Address |
| ptr | int pointer | 0x000000 | 0x22FF68 |
| a | int array pointer | 0x000000 | 0x22FF64 |
| otherptr | int ptr | 0x3D3BB8 | 0x22FF60 |

