Lecture 3

ters and Arrays

Pointers and Arrays Spring 2024

Relationship Between Pointers and Arrays

- Arrays and pointers are closely related
- Array elements are laid out sequentially in memory
- By using the *address-of* operator (&), we can determine addresses of each

element of array. For example,

```
int array[5 ] = { 9, 7, 5, 3, 1 };

std::cout << "Element 0 is at address: " << &array[0] << '\n'; //0x7fffffffee20
std::cout << "Element 1 is at address: " << &array[1] << '\n'; //0x7fffffffee24
std::cout << "Element 2 is at address: " << &array[2] << '\n'; //0x7fffffffee28
std::cout << "Element 3 is at address: " << &array[3] << '\n'; //0x7fffffffee2c
std::cout << "Element 4 is at address: " << &array[4] << '\n'; //0x7fffffffee30
```

- Each of these memory addresses is 4 bytes apart, which is the size of an integer

Relationship Between Pointers and Arrays (Cont.)

- Accessing array elements with pointers
  - Assume declarations:

    int List[5]; //declare array

```
int *LPtr; //declare a pointer
LPtr = List; // Address of first Element of array List
```

Effect:-

- Name of the array is also the address of its first element

- List is an address, no need for ***&*** operator

- The LPtr pointer will contain the address of the first element of the **array**

List. • Element **List[2]** can also be accessed using **LPtr[2]**.

Accessing 1-Demensional Array Using Pointers

▪ We know, Array name denotes the memory address of its first slot.

▪ Example:

▪ **int List [50];**

▪ **int *LPtr;**

▪ **LPtr = List; //stores address of the first element in**

Pointer ▪ Other slots of the List [50] can be accessed by performing Arithmetic operations on Pointer.

▪ For example the address of 4th element can be accessed using:

▪ LPtr + 3;

▪ The value of 4th element can be accessed using:- ▪ *(LPtr + 3);

Address Data **980 Element 0 982 Element 1 984 Element 2 986 Element 3 988 Element 4 990 Element 5 992 Element 6 994 Element 7 996 Element 8**

…

…

998 Element 50

Accessing 1-Demensional

Array We can access all element of List [50] using

Pointers and for loop combinations.

```
…
…
int List [ 50 ];
int *LPtr;
LPtr = List;
for ( int i = 0; i < 50; i++ )
{
cout << *LPtr;
LPtr++; // Address of next element
}
```

```
for ( int i = 0; i < 50; i++ )
cout << List [ loop ] ;
```

Address Data 980 Element 0 982 Element 1 984 Element 2 986 Element 3 988 Element 4 990 Element 5 992 Element 6 994 Element 7 996 Element 8

…

…

998 Element 50

This is Equivalent to

Accessing 1-Demensional Array using

pointer arithmetic We can access all element of List

[50] using

Pointers and for loop combinations.

```
…
…
int List [ 50 ];
int *LPtr;
LPtr = List;
for ( int i = 0; i < 50; i++ )
{
cout << *(LPtr+i);

 }
```

This is Equivalent to

```
for ( int i = 0; i < 50; i++ )
cout << List [ loop ] ;
```

Address Data 980 Element 0 982 Element 1 984 Element 2 986 Element 3 988 Element 4 990 Element 5 992 Element 6 994 Element 7 996 Element 8

…

…

998 Element 50

Pointer to an Array

- ***Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array.***
- *ptr* is pointer that can point to an array of 10 integers.

- Here the type of ptr is ‘pointer to an array of 10 integers’.
- **Note :** The pointer that points to the 0^{th} element of array and the pointer that points to the whole array are totally different.

Syntax:

```
data_type (*var_name)[size_of_array];
```

Example:

```
int (*ptr)[10];
```


Pointer to an Array

Initially the value held by p and ptr is same, however, arithmetic on both pointers give a different result as both of these pointers are of different types.

Output:

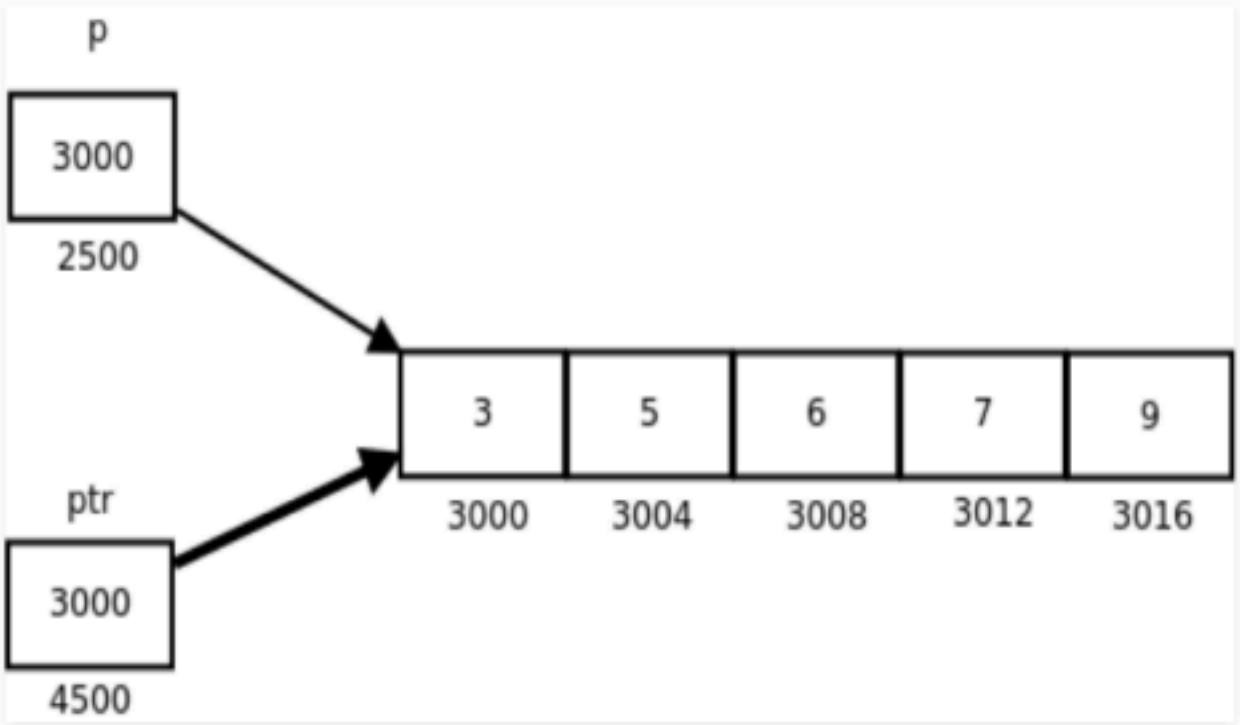
```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```

p: is pointer to 0^{th} element of the array *arr*, while ***ptr*** is a pointer that points to the whole array *arr*.

- The base type of *p* is int while base type of *ptr* is 'an array of 5 integers'.
- We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer *ptr* will be shifted forward by 20 bytes.

Pointer to an Array

The following figure shows the pointer p and ptr. Darker arrow denotes pointer to an array.



On dereferencing a pointer expression we get a value pointed to by that pointer expression. Pointer to an array points to an array, so on dereferencing it, we should get the array, and the name of array denotes the base address. So whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

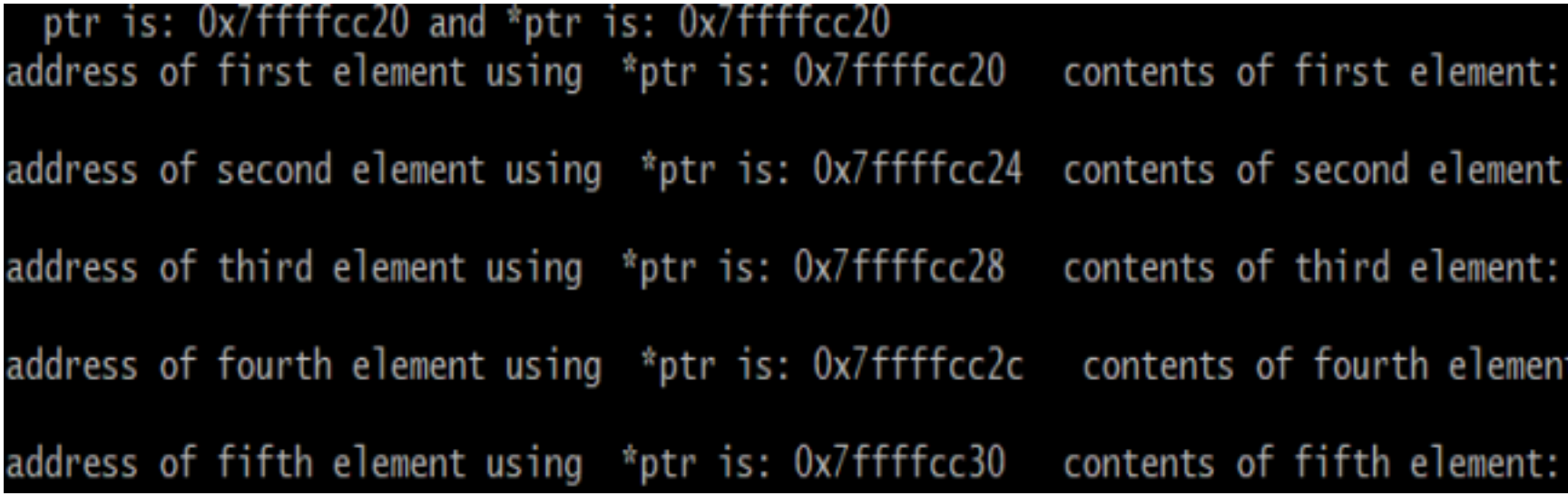
Class activity

- Say Array[5] is an int array:
  int Array[5] = {5,7,4,8,9};
- And (*ptr)[5] points to Array
  int (*ptr)[5] = &Array
- Now using ptr (and the dereferencing operator) find out the address of the:
  - Whole Array
  - First element of the array
  - Contents of each element of the array

Processing 1D array using “Pointer to Array”

```
int main() {
int array[5 ] = { 9, 7, 5, 3, 1 };
int (*ptr)[5] = &array;
cout<<" ptr is: "<<ptr<<" and *ptr is: "<<*ptr<<'\n';
cout<<"address of first element using *ptr is: "<<*ptr<<" contents of first element: "<<*(*ptr+0)<<'\n';
```

```
cout<<"address of second element using *ptr is: "<<*ptr+1<<" contents of second element:
"<<*(*ptr+1)<<'\n';
cout<<"address of third element using *ptr is: "<<*ptr+2<<" contents of third element:
"<<*(*ptr+2)<<'\n';
cout<<"address of fourth element using *ptr is: "<<*ptr+3<<" contents of fourth element:
"<<*(*ptr+3)<<'\n';
cout<<"address of fifth element using *ptr is: "<<*ptr+4<<" contents of fifth element:
"<<*(*ptr+4)<<'\n';
}
```



Memory layout of 2D array

- In C/C++, a multidimensional array is just an array of arrays.
- C/C++ stores arrays declared as two-dimensional using a one dimensional array
  - The first elements stored are those in the first row (in order). Then, the second row is stored, etc.
  - This memory allocation policy is called "***row-major ordering***".
- For example, when we declare the following 2d

  array int D[4][3]

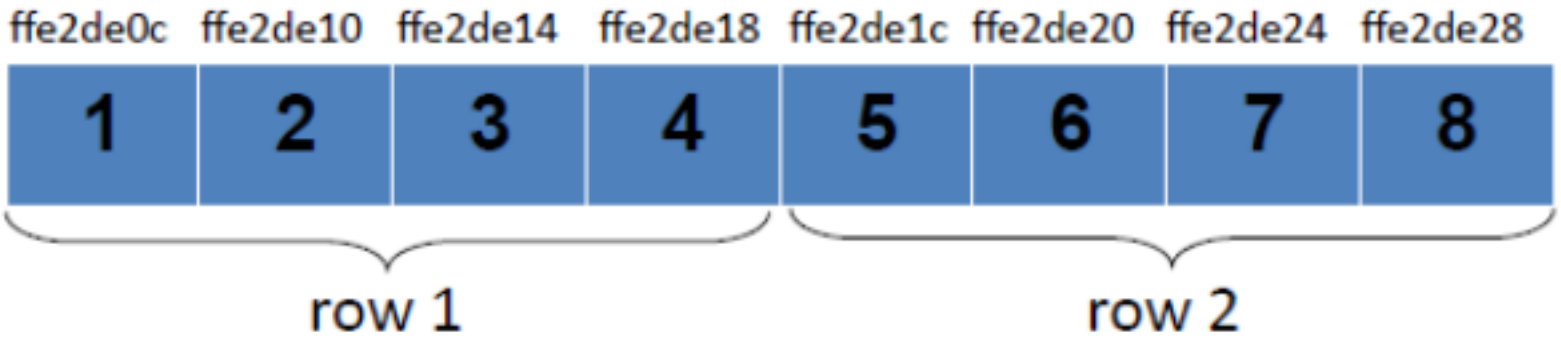
  - It is represented as a *one-dimensional array* with 4 elements, each of which is an array with 3 elements

Another example: 2d array in memory

- When we declare 2d array as follows:

int myArray[2][4]= {{1,2,3,4},{5,6,7,8}};

- It is stored in memory in the following form. Row 1 and row 2 are 1D arrays of the 2D array.



Array elements are stored in *row major* order
Row 1 first, followed by row2, row3, and so on

2d array representation in memory

• 2d array declaration in c++:

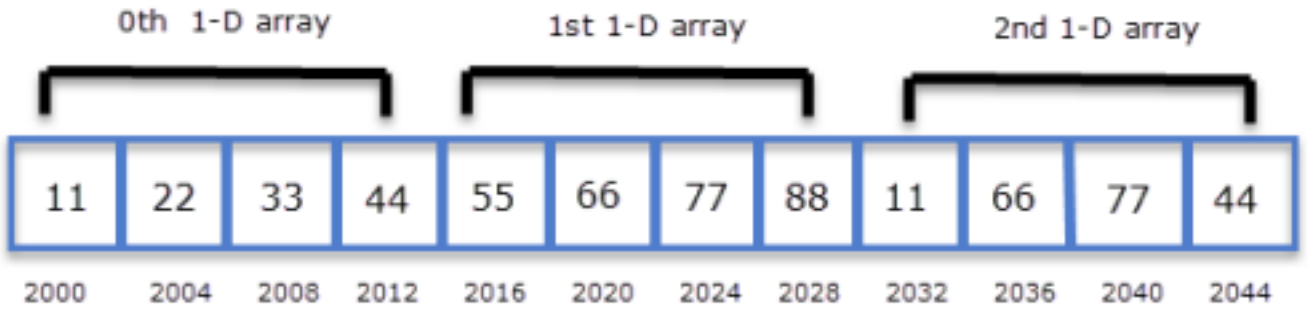
int arr[3][4] = { {11,22,33,44},

{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:

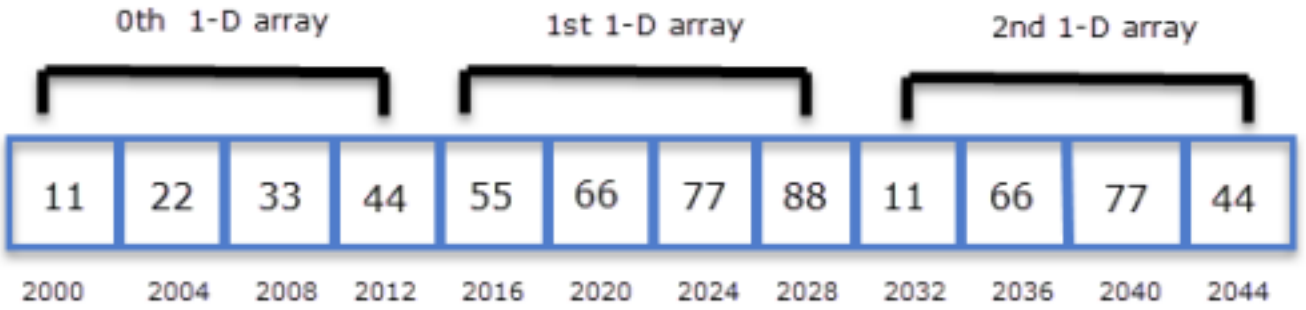
arr points to 0th 1-D array.

(arr + 1) points to 1st 1-D array.

(arr + 2) points to 2nd 1-D array.

• Generalizing,

(arr + i) points to ith 1-D array.



Dereferencing a pointer to array

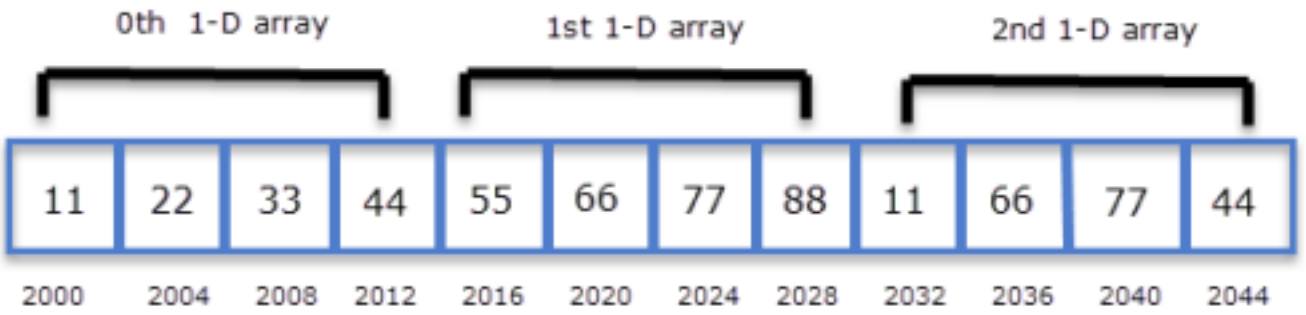
- Dereferencing a pointer to an array, i.e., *arr becomes a ***new type of pointer*** which gives the base address of the first 1D array.
- dereferencing arr we will get *arr → ***base type of *arr is*** ***(int*)***
- Similarly, dereferencing arr+1 we will get *(arr+1)
- Generalizing, *(arr+i) points 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*).

Now how to use arr to access individual elements of 2d array?

- From previous slides: *(arr + i) points to the *base address* of every ith 1-D array and it is of base type pointer to int
- Now let's use pointer arithmetic
- *(arr + i)+0 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 jth element of ith 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: Indexing through 2D array

```
#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
```

Another Example: Array names and

pointers. • Let's first declare the 2D array.

int aiData [3][3] = { { 9, 6, 1 }, { 144, 70, 50 }, {10, 12, 78} };

9	6	1
144	70	50
10	12	78

aiData[3][3]

- If aiData is the 2D array, then aiData is the address of the ***first element*** of the 2d array i.e., first row.

aiData[0] aiData[1] aiData[2]

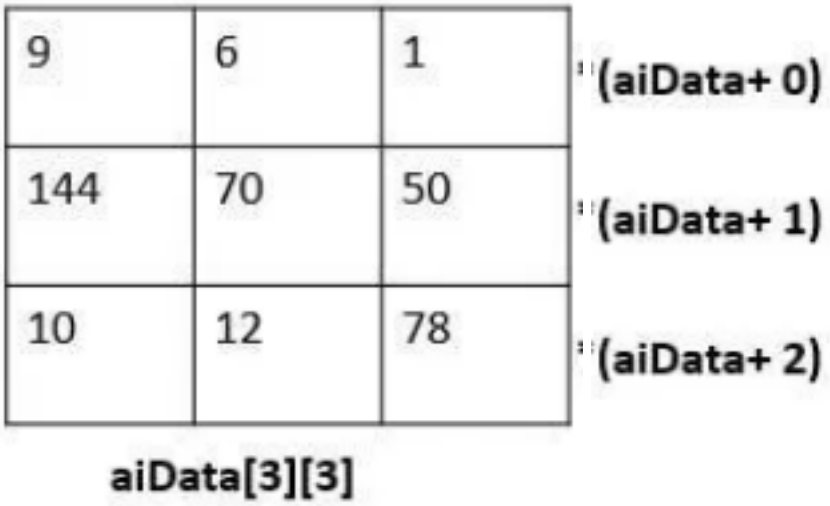
1st Row	**2nd Row**	**3rd Row**

- And dereferencing aiData, i.e., *aiData or *(aiData+0), gives the ***address*** of the

first element of the first row. [See next slide for visual description]

Example: Array names and pointers.

- aiData+0 is the address of the ***first element*** of the 2d array i.e., first row.
- aiData+1 is the address of the ***second element*** of the 2d array i.e., second row.
- aiData+2 is the address of the ***third element*** of the 2d array i.e., third row.

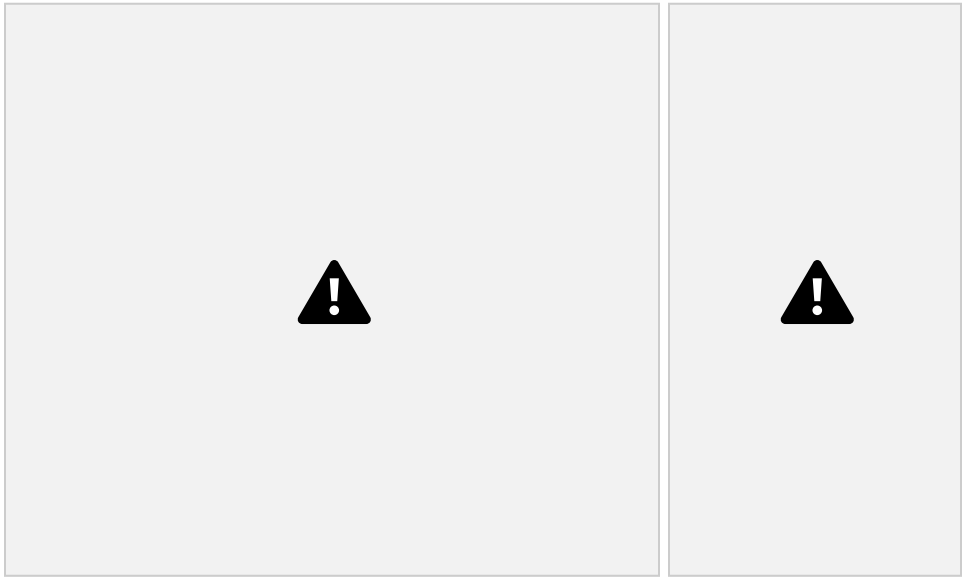


Example: Dereferencing the pointers.

- Dereferencing aiData+1 gives the ***address*** of the first element of the 1^{th} row,

- *(aiData+1) → is the ***address*** of first element of the 1^{st} row. • And

*(aiData+2) → is the ***address*** of first element of the 2nd row.



Indexing 2D array using pointers

- *(aiData+0) + 0 → address of first element of the 0th row

*(* (aiData+0) + 0) → value of 1st element → aiData[0][1]

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

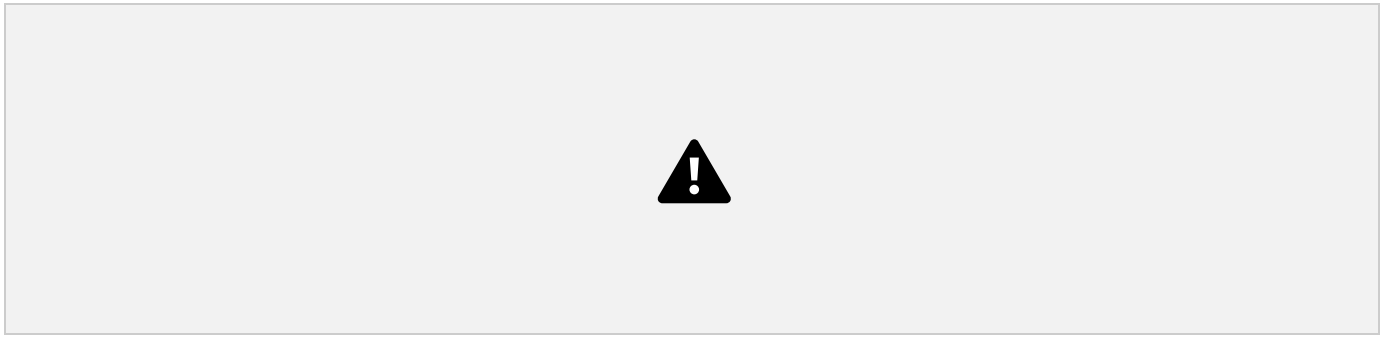
- Similarly, If a 2-D array has 4 rows and 5 cols i.e int arr[4][5], then a pointer to an array of 5 integers is needed, int (*p)[5];
- Let's continue from the previous example, **int (*p)[4];** p = arr; //note p points to the first 1-D array of the 2D array. Here p is a pointer to an array of 4 integers.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

The base type of (p+i) is a pointer to an array of 4 integers.



Dereferencing p+i

- Dereferencing (p+i), i.e., *(p+i) gives the base address of ith 1-D array
- More precisely, base type of *(p + i) is a pointer to int. or (int *). • To access the jth element of the ith 1-D array, we use *(p+i)+j • So *(p + i)

+ j points to the address of jth element of ith 1-D array.

- Now further, dereferencing the expression *(p + i) + j , i.e., *(*(p + i) + j) gives the value of the jth element of the ith 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} };
```

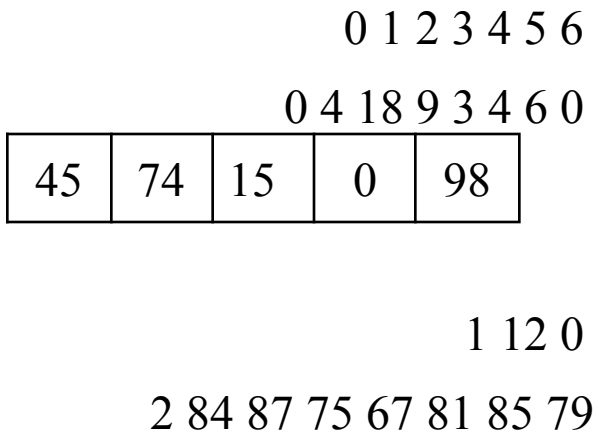
```
int i, j;
int (*p)[4]; //Pointer to array of 4 integers
p = arr; // p points to first element of the 2d array
for(i = 0; i < 3; i++)
{
cout<<"address of "<<i<<"th array\t\t"<<*(p + i)<<endl;
for(j = 0; j < 4; j++)
{
cout<<"arr["<<i<<"]["<<j<<"] = "<< *( *(p + i) + j)<<endl ;
            }
cout<<“\n\n”;
}

return 0;
}
```

Class activity

- You have the following 2D array, Array[3][7].

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



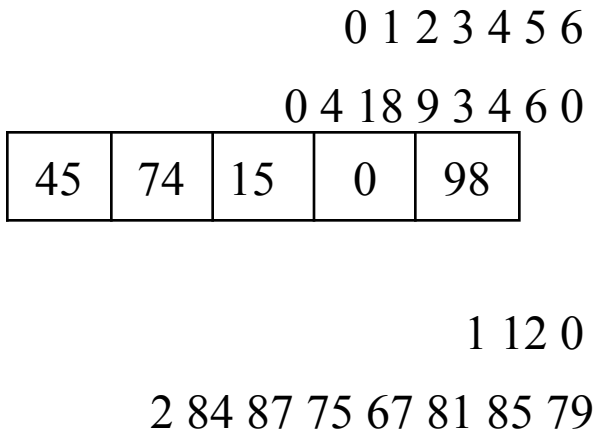
- Declare a pointer, ptr, to the above 2D array.
- Execute a nested loop to retrieve the contents of the 2D array.

Class activity

- You have the following 2D array, Array[3][7].

```
int NUMROWS = 3;
```

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



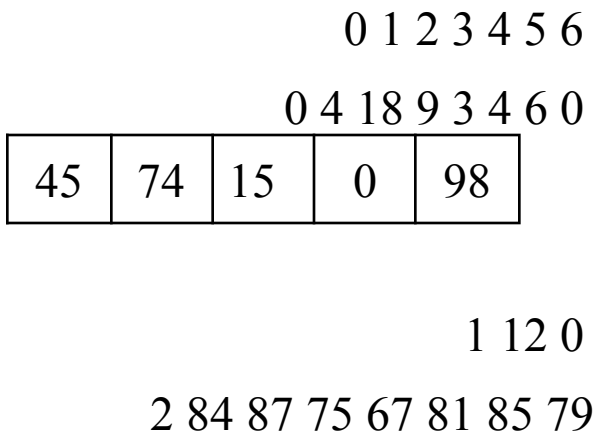
- Declare a pointer, ptr, to the above 2D array.
- Execute a nested loop to retrieve the contents of the 2D array.

- Hint: dimensions of the array are 3x7. So we need a pointer to an array of 7 integers.

Class activity

- You have the following 2D array, Array[3][7].

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



- Declare a pointer, ptr, to the above 2D array.
- Execute a nested loop to retrieve the contents of the 2D array.

- Hint: dimensions of the array are 3x7. So we need a pointer to an array of 7 integers.

```
int (*ptr) [7];
```

```
ptr = Array;
```