INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEC-101
Programming with C++

Module-3:
Pointers

About Subject

- Aggregate Data-types:
  - Arrays
  - Pointers
  - Structures
  - Dynamic data and Pointers
  - Dynamic arrays

Memory

Memory used by a program is typically divided into four different areas:

- The code area: where the compiled program resides in memory.
- The global area: where global variables are stored.
- **The heap: where dynamically allocated variables are allocated from.**
- The stack: where parameters and local variables are allocated from.

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack.**

- **What are applications?**
- One use of dynamically allocated memory is to allocate memory of variable size, which is not possible with compiler allocated memory.
- The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore.

- **How is it different from memory allocated to normal variables?**
  - For normal variables like “int a”, “char str[10]”, etc., memory is automatically allocated and deallocated. For dynamically allocated memory like “int *p = new int[10]”, it is the programmer’s responsibility to deallocate memory when no longer needed. If the programmer doesn’t deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).

- **How is memory allocated/deallocated in C++?**
  - C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory.
  - C++ supports these functions and also has two operators **new** and **delete,** that perform the task of allocating and freeing the memory in a better and easier way.

Dynamic Memory Allocation

Allocating storage from the heap at runtime

- We often don't know how much storage we need until we need it
- We can allocate storage for a variable at run time
- Recall C++ arrays
  - We had to explicitly provide the size and it was fixed
- We can use pointers to access newly allocated heap storage

Using the keyword “new”

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax: pointer= new data_type;

using new to allocate storage

```
int *int_ptr {nullptr};

int_ptr = new int;          // allocate an integer on the heap

cout << int_ptr << endl;    // 0x2747f28

cout << *int_ptr << endl;   // 41188048 - garbage

*int_ptr = 100;

cout << *int_ptr << endl;   // 100
```

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;

            OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

- Here, the pointer variable is the pointer of type data-type.
- Data type could be any built-in data type, including an array, or any user-defined data type, including structure and class.

Initialize memory

- We can also initialize the memory for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

- pointer-variable = new data-type(value);

- What if enough memory is not available during runtime?
  - If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer.
  - Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

Using the Keyword “delete”

using `delete` to deallocate storage

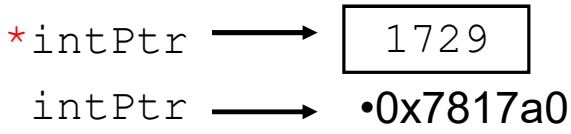
```
int *int_ptr {nullptr};

int_ptr = new int;   // allocate an integer on the heap

. . .

delete int_ptr;   // frees the allocated storage
```

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory.

```
int *intPtr{nullptr};
```
Create a pointer

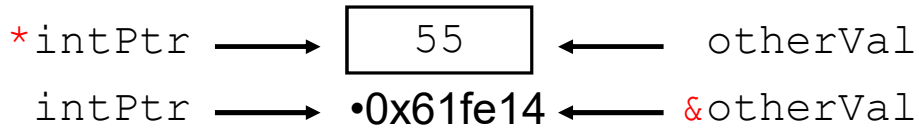
```
intPtr = new int;
```
Allocate memory on the heap

```
*intPtr = 3456;
```
Set value at given address




```
delete intPtr;
```
Deallocate memory

```
int otherVal = 5;
intPtr = &otherVal;
```
Change `intPtr` to point to a new location



Example

```
// using new and delete operator
#include<iostream>
using namespace std;
int main()
{
int *intPtr{nullptr};
intPtr=new int;
*intPtr= 1729;// have no name
cout <<intPtr<<endl;
cout<<*intPtr<<endl;
delete intPtr;
cout<<*intPtr<<endl;
int otherVal = 55;
cout<<&otherVal<<endl;
intPtr = &otherVal;
cout<<intPtr<<endl;
cout<<*intPtr<<endl;
system ("pause");
return 0;
}
```

```
0x7817a0
1729
7870976
0x61fe14
0x61fe14
55
Press any key to continue . . .
```

Using new[ ] to allocate storage for an array

pointer = new type [number_of_elements]

- This syntax is used to assign a block (an array) of elements of type **type**, where the number_of_elements is an integer value representing the number of elements needed.
- It returns a pointer to the beginning of the new block of memory allocated

```
int *array_ptr {nullptr};
int size {};

cout << "How big do you want the array? ";
cin >> size;

array_ptr = new int[size]; // allocate array on the heap

// We can access the array here
// we'll see how in a few slides
```

Using delete [ ] to deallocate

```
int *array_ptr {nullptr};
int size {};

cout << "How big do you want the array?";
cin >> size;

array_ptr = new int[size]; // allocate array on the heap

. . .

delete [] array_ptr;   // free allocated storage
```

Example

```
#include<iostream>
using namespace std;
int main()
{
    int n;
cout <<" Enter the number of students : ";
cin>> n;
int *p{0};
p=new int[n];// dynamic array of size n
// store marks of i students
for (int i=0;i<n; i++)
{
    cout<<"Enter the marks of student number " <<i+1<<" : ";
    cin>> p[i];
}

cout<< "Entered marks are ";
for (int i=0;i<n; i++)
{
    cout<< p[i]<<", ";
}
delete [] p; //deallocate the memory
return 0;
}
```

```
er the number of students : 5
r the marks of student number 1 : 30
r the marks of student number 2 : 40
r the marks of student number 3 : 56
r the marks of student number 4 : 78
r the marks of student number 5 : 34
ered marks are 30, 40, 56, 78, 34,
cess returned 0 (0x0)   execution time : 13.248 s
s any key to continue.
```

Memory Leak

- Memory leak takes place when a memory allocated from the heap is no longer needed and is not released by delete operator.

Example

```
#include <iostream>
int main()
{
using namespace std;
short *pPointer;
pPointer= new short;
// allocate a memory with a new operator on the free store
// and assign a memory address to a pointer named pPointer pPointer = new short;
*pPointer = 51;
// print out the value of pPointer and its associated memory address
cout << "pPointer Value: "<< *pPointer << "\tMemory Address:" << pPointer<< endl;
// reassign a pointer to a new memory address from a free store
pPointer = new short (19);
// assign an integer value to a pointer
// print out a value of pPointer and its corresponding memory address
cout<<"*pPointer Value: "<< *pPointer << "\tMemory Address: "<< pPointer<< endl;
// de-allocating memory back to the free store
delete pPointer; // if not used it will cause memory leak
system("pause");
return 0;
}
```

```
pPointer Value: 51      Memory Address:0x7617a0
*pPointer Value: 19     Memory Address: 0x7617c0
```

Dynamic Arrays

Static arrays

```
int intArray[10];
intArray[0] = 3456;
```

Dynamic arrays

```
int *intArray;
intArray = new int[10];
intArray[0] = 3456;

...

delete[] intArray; // de-allocates the memory
```

- An array name is a constant pointer that is allocated at compile time.

```
float a [ 20 ];        // a is a const pointer to a block of 20 floats
```

- The declaration of a is called static binding because it is allocated at compile time.
- We can use a non-constant pointer to postpone the allocation of memory until the program is running.
- It is called run-time binding or dynamic binding:

```
float* p = new float [ 20 ];
```

- An array that is declared this way is called a dynamic array.

```
float a [ 20 ]; // static array
float* p = new float [ 20 ] ; // dynamic array
```

- The static array a is created at compile time ;its memory remains allocated throughout the run of the program
- The dynamic array p is created at run time ; its memory allocated only when its declaration executes.
- The memory allocated to the array p is deallocated as soon as the delete operator is invoked on it:

```
delete [ ] p;  // de-allocated the array p
```

(subscript operator [ ] must be included this way, because p is an array)

Example

```
#include<iostream>
using namespace std;
int main()
{
    int n;
cout <<" Enter the number of students : ";
cin>> n;
int *p{0};
p=new int[n];// dynamic array of size n
// store marks of i students
for (int i=0;i<n; i++)
{
    cout<<"Enter the marks of student number " <<i
    cin>> p[i];
}

cout<< "Entered marks are ";
for (int i=0;i<n; i++)
{
    cout<< p[i]<<", ";
}
delete [] p; //deallocate the memory
return 0;
```

```
 Enter the number of students : 5
Enter the marks of student number 1 : 44
Enter the marks of student number 2 : 32
Enter the marks of student number 3 : 45
Enter the marks of student number 4 : 76
Enter the marks of student number 5 : 89
Entered marks are 44, 32, 45, 76, 89,
Process returned 0 (0x0)   execution time : 14.186 s
Press any key to continue.
```

```cpp
//Create Dynamic array using new operator.
#include<iostream>
using namespace std;
int main(){
    // 1.Create a Dynamic array having name   marksArray
 int N = 4;
 int *marksArray = new int [N];//dynamic array marksArray with size 4 is declared
 //int maksArray[N];  same as above but static array declaration
 cout<<"Enter marks:"<<endl<<endl;
 //2.Enter marks in the array
 for (int k = 0; k < N; k++)// enter marks in position 0 to 3 in marksArray.
  { cout<<"Enter marks for marksArray["<<k<<"]: ";
   cin>>marksArray[k];}
 // 3.Display the values of the elements of tha array
 cout<<"Display of elements of marksArray: "<<endl<<endl;
 for (int k = 0; k < N; k++)
 cout<<"marksArray["<<k<<"] = "<<marksArray[k]<<endl;cout<<endl;
 //4.Create another array of size twice than marksArray having name marksArrayA
 int *marksArrayA = new int[2*N];//dynamic array marksArrayA with size 2*N=8 is declared
 //5.Transfer marks array elements  to marksarrayA
 for (int j = 0; j < N; j++)
 marksArrayA[j] = marksArray[j];// transfer all elements of marksArray to
 //marksArrayA in position 0 to 3.
 //5.Delete the dynamic array marksArray
 delete [] marksArray;//dynamic array marksArray is deleted
 //6.Now enter marks in vacant elelments of marksArrayA
 //Note only elements 4 to 7 are vacant
 for (int k = N; k < 2*N; k++)//marks in marksArrayA in position 4 to 7 are entered
  {cout<<"Enter marks for marksArrayA["<<k<<"]: ";
   cin>>marksArrayA[k];}
  //7.Dispaly elements of marksArrayA
  cout<<"Display of elements of marksArrayA: "<<endl<<endl;
 for (int k = 0; k < 2*N; k++)
```

```
C:\Documents and Settings\Administrator\De...
Enter marks:

Enter marks for marksArray[0]: 60
Enter marks for marksArray[1]: 70
Enter marks for marksArray[2]: 88
Enter marks for marksArray[3]: 79
Display of elements of marksArray:

marksArray[0] = 60
marksArray[1] = 70
marksArray[2] = 88
marksArray[3] = 79

Enter marks for marksArrayA[4]:
```

```
//Create Dynamic array using new operator.
#include<iostream>
using namespace std;
int main(){

// 1.Create a Dynamic array having name marks
int N = 4;
int *marks = new int [N];//dynamic array marks with size 4 is declared
//int maks Array[N]; same as above but static array declaration
cout<<"Enter marks: "<<endl<<endl;
//2.Enter marks in the array
 for (int k = 0; k < N; k++) // enter marks in position 0 to 3 in marks Array.
    {
    cout<<"Enter marks for marks["<<k<<"]: ";
    cin>>marks[k];}
// 3.Display the values of the elements of the array
cout<<"Display of elements of marksArray: "<<endl<<endl;
for (int k = 0; k < N; k++)
cout<<"marks["<<k<<"] = "<<marks[k] <<endl;
cout<<endl;
//4.Create another array of size twice than marks  having name marks_new
int *marks_new = new int[2*N]; //dynamic array marks ArrayA with size 2*N=8 is declared
//Transfer marks array elements to marks_new
for (int j=0; j < N; j++)
marks_new[j] = marks[j]; // transfer all elements of marks to
//marks_new in position 0 to 3.
//5.Delete the dynamic array marks
delete [] marks;//dynamic array marks Array is deleted
//6. Now enter marks in vacant elements of marks_new
//Note only elements 4 to 7 are vacant

for (int k = N; k < 2*N; k++)//marks in marks_new in position 4 to 7 are entered
    {
        cout<<"Enter marks for marksArrayA["<<k<<"]: ";
        cin>>marks_new[k];}
//7.Dispaly elements of marksArrayA
cout<<"Display of elements of marksArrayA: "<<endl<<endl;
 for (int k = 0; k < 2*N; k++)
cout<<"marks_new["<<k<<"] = "<<marks_new[k] <<endl;
cout<<endl;
delete [] marks_new;
return 1;
}
```

```
Enter marks for marks[0]: 10
Enter marks for marks[1]: 20
Enter marks for marks[2]: 30
Enter marks for marks[3]: 40
Display of elements of marksArray:

marks[0] = 10
marks[1] = 20
marks[2] = 30
marks[3] = 40

Enter marks for marksArrayA[4]: 50
Enter marks for marksArrayA[5]: 60
Enter marks for marksArrayA[6]: 70
Enter marks for marksArrayA[7]: 80
Display of elements of marksArrayA:

marks_new[0] = 10
marks_new[1] = 20
marks_new[2] = 30
marks_new[3] = 40
marks_new[4] = 50
marks_new[5] = 60
marks_new[6] = 70
marks_new[7] = 80


Process returned 1 (0x1)   execution time : 16.298 s
Press any key to continue.
```

2D Dynamic Array

- To allocate a 2D array, we use a double pointer, or a pointer to a pointer, as the base.
- To create a 3x2 2D array of ints.
- For that array, allocated an array of 3 int pointers for the first dimension and 3 arrays of 2 ints for the second dimension

3 dynamic Arrays

stack

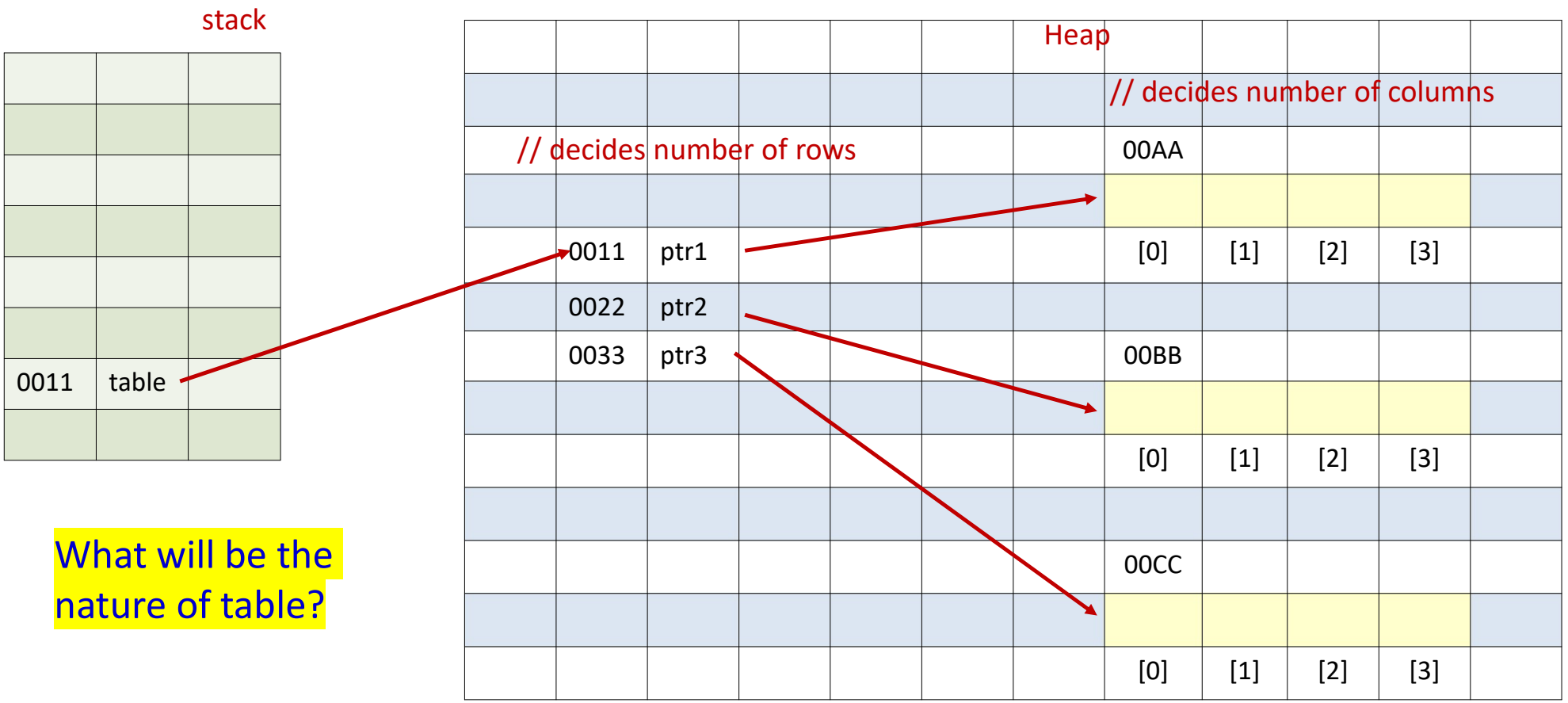
		00AA	ptr1	
		00BB	ptr2	
		00CC	ptr3	

Heap

			00AA						
			[0]	[1]	[2]	[3]			
			00BB						
			[0]	[1]	[2]	[3]			
			00CC						
			[0]	[1]	[2]	[3]			

- To create 3 dynamic arrays, we need 3 pointers.
- What if you need to create 100s of them?
- The pointers are stored in the stack. So they are not dynamic.

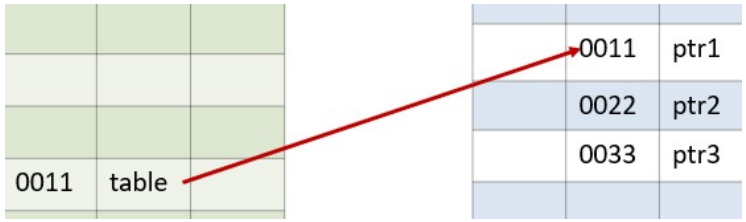
3 dynamic Arrays

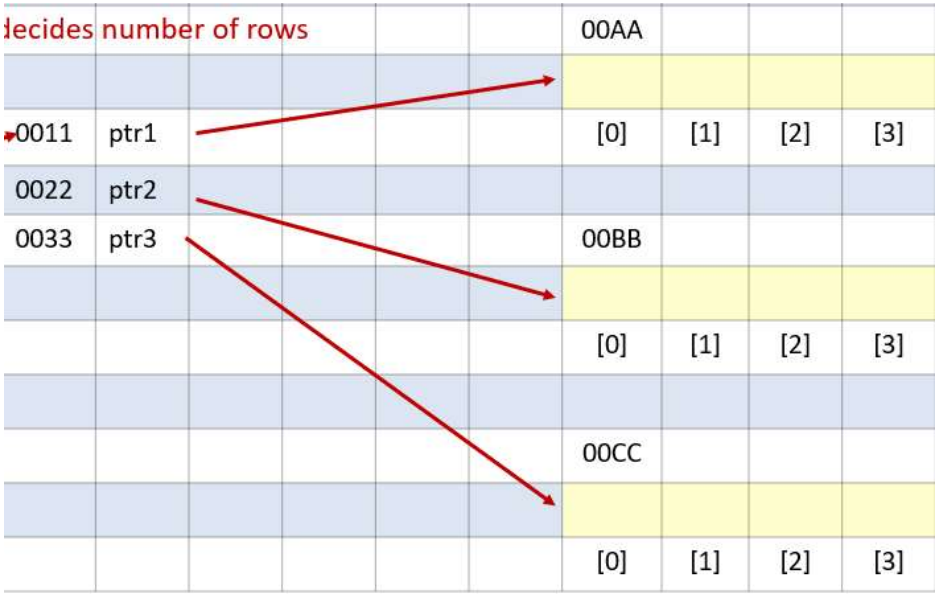


What will be the nature of table?

How to create

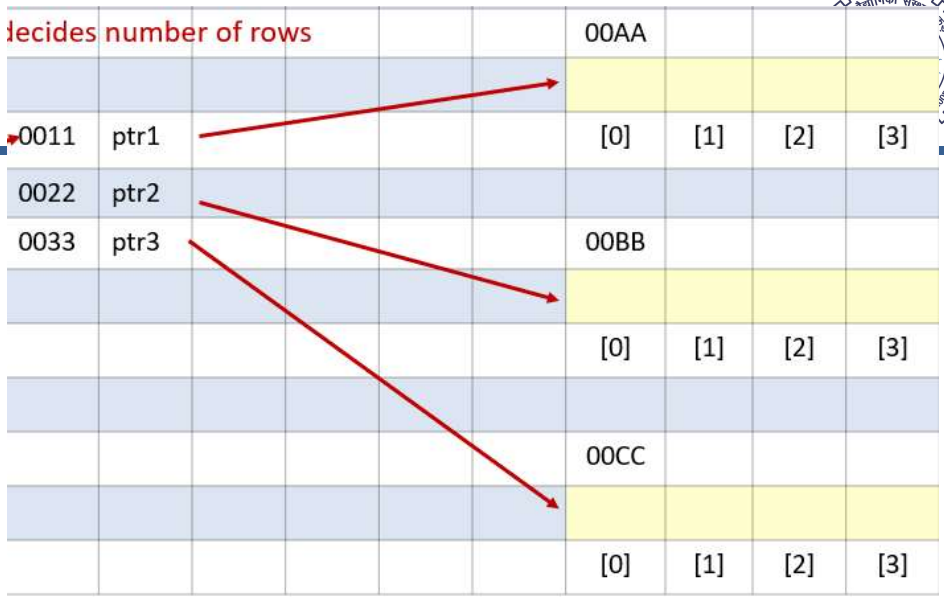
```
#include<iostream>
using namespace std;
int main(){
cout<< "enter number of rows and columns";
int rows, cols;
cin>>rows>>cols;
int ** table=new int*[rows];
for (int i=0;i<rows,i++){
    table[i]=new int[cols];
}

table[1][2]=88;// points to 2nd row 3rd colum
}
```

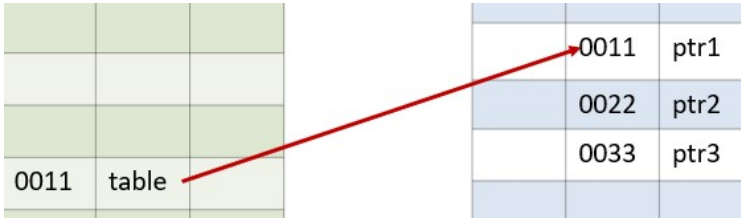




How to delete

```
#include<iostream>
using namespace std;
int main(){
cout<< "enter number of rows and columns";
int rows, cols;
cin>>rows>>cols;
int ** table=new int*[rows];
for (int i=0;i<rows,i++){
    table[i]=new int[cols];
}
table[1][2]=88;// points to 2nd row 3rd colum
// Deallocation sequence is opposite
// step-1
for (int i=0;i<rows,i++){
    delete[] table[i];
}
//step-2
delete [] table;

//step-3
table=nullptr;
return 0;
}
```





Example

```
#include<iostream>
using namespace std; int main()
{
// Allocate a two-dimensional 3*4 array of ints
int** table = new int*[3];
for(int i = 0; i < 3 ; i++ )
    table[i] = new int[4];
// Fill the array
for (int i = 0; i < 3; ++i){
    for (int j=0; j < 4; ++j){
     table[i][j] = i + j;}}
// Output the array
for (int i = 0; i < 3; ++i) {
        for (int j=0; j < 4; ++j)
        { cout<<table[i][j] << " ";}

cout << endl;}
cout << endl;
```

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

0 1 2 3
1 2 3 4
2 3 4 100


Process returned 0 (0x0)   exe
Press any key to continue.
```

```
// change data
table [2][3]=100;
// Output the array
for (int i = 0; i < 3; ++i) {
        for (int j=0; j < 4; ++j)
        { cout<<table[i][j] << " ";}

cout << endl;}
cout << endl;
// Deal Locate
for(int i = 0; i < 3 ; i++ ){
    delete [] table[i];}
    delete [] table;
    table=nullptr;
return 0;
}
```

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

0 1 2 3
1 2 3 4
2 3 4 100


Process returned 0 (0x0)   exe
Press any key to continue.
```