Computer Programming

C++

Prof. Amit Agarwal

amitfce@iitr.ac.in *amit.agarwal@mfs.iitr.ac.in*

Arrays in C++

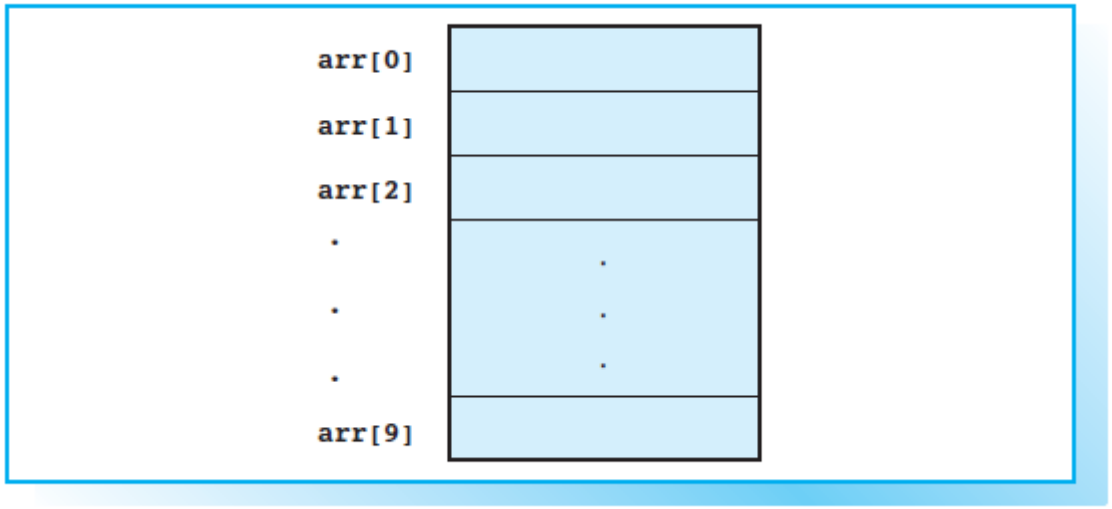
- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

```
string cars[4];
string cars[4] = {“Maruti Suzuki", “Honda", "Ford", “Kia"};
int myNum[3] = {10, 20, 30};
```

- a collection of a fixed number of components wherein all of the components have the same data type

Do we need to state the size of the array if initialized during declaration?

Arrays in C++




```
int arr[10];          // memory space is 10 * sizeof(int);
```

Arrays in C++

- Access the elements of an array by referring to index number.
- Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

```
cout << cars[2];// → Ford
```

- An element of an array can be changed by referring to the index of that element.

```
string cars[4] = {“Maruti Suzuki", “Honda", "Ford", “Kia"};

cars[0] = “Toyota";
cout << cars[0]; // → Toyota
```

Arrays in C++

Let us try to create an array, which has size of 10, and elements are sum of previous two elements. Initial two elements are 0 and 1.

```
int list[10];
list[0] = 0;
list[1] = 1;

for (int i = 2; i < 10; i++) {
  list[i] = list[i-2] + list[i-1];
  cout << list[i] << "\n";
}
```

TODO: write a program to determine the maximum element from an integer array.

Arrays in C++

- Array Index out of bound → if someone attempts to access the element at a position which does not exist or is higher than the *size of the array – 1*.

- Arrays can be initialised during declaration; size is not required.

```
string cars[] = {“Maruti Suzuki", “Honda", "Ford", “Kia"};
```

TODO: Try output of he following…

```
string cars[5] = {“Maruti Suzuki", “Honda", "Ford", “Kia"};
cout << cars[4];
```

Copying an Array

```
int firstArr[50];
int secondArr[50];
```

To copy, can we simply use the assignment operator?

```
firstArr = secondArr;
```

```
for (int i=0;i<50;i++){
        firstArr [i] = secondArr[i];
}
```

Similarly, there is no aggregate input/ output for arrays, aggregate arithmetic, etc.

Initializing Arrays in a loop

```
int a[10];

int length = sizeof(a)/sizeof(a[0]);

for(int i = 0; i < length; i++) {
   a[i] = 0;
}
```

```
int num;
cout << "Enter the number of students" << endl;
cin >> num;

int a[num];
for(int i = 0; i < num; i++) {
    cout << "Enter the " << i << "th number." << endl;
    cin >> a[i];
    cout << a[i];
}
```

Looping over array

```
#include <iostream>
#include<array>
using std::cout;
using std::endl;

int main() {
   int a[] = {1,3,6,2,4,7};

    for (int i : a) {
         cout << i << endl;
     }

   return 0;
}
```

Looping over array

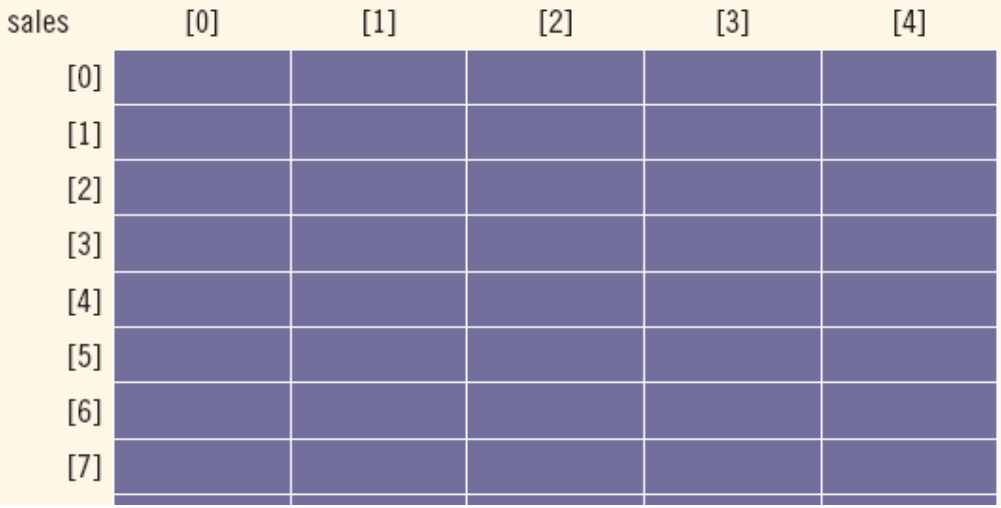
```
#include <iostream>
#include<array>
using std::cout;
using std::endl;

int main() {
    float a [] = {5.0, 3.2, 6.6, 2.4, 4.0, 7.9};

    for (int i : a) {
         cout << i << endl;
     }

    return 0;
}
```

What will be the output of the above program?

2D Arrays in C++

- Also known as matrix

```
string sales[10][5];
```



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

2D Arrays in C++

```
int rows = 2;
int cols = 2;
int mat[rows][cols];

for (int i = 0; i < rows; i++) {
     for (int j = 0; j < cols; j++) {
          mat[i][j] = i * j;
          cout << mat[i][j] ;
     }
     cout<< endl;
}
```

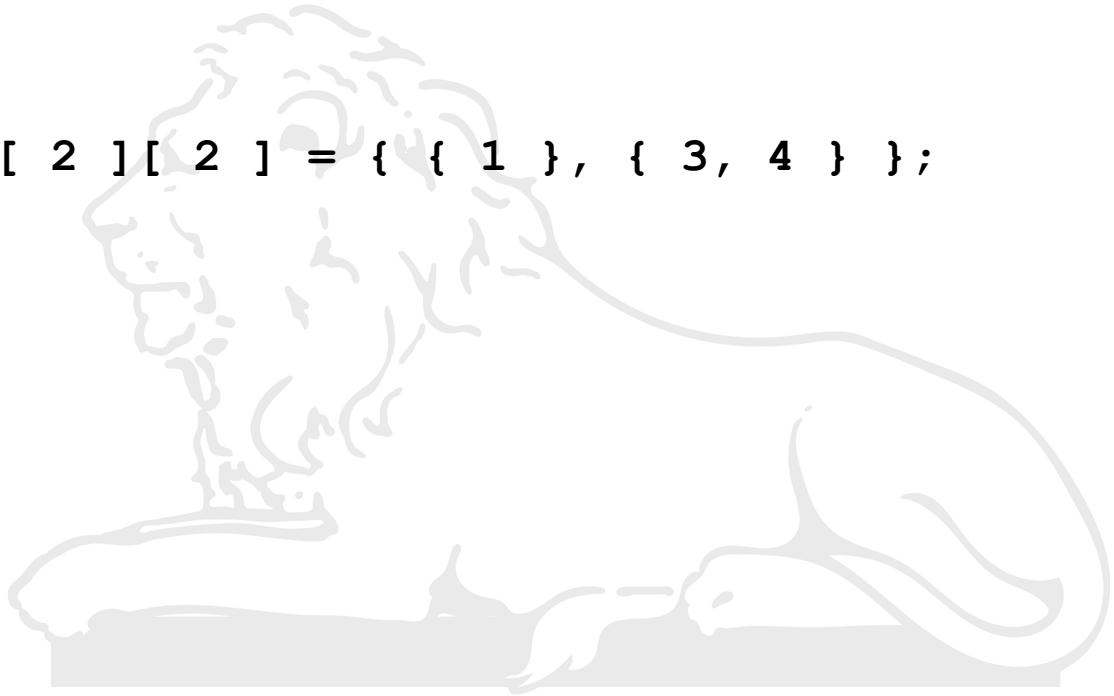
2D Arrays in C++

```
int b[ 2 ][ 2 ] = { {1, 2}, {3, 4}};
```

1	2
3	4

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

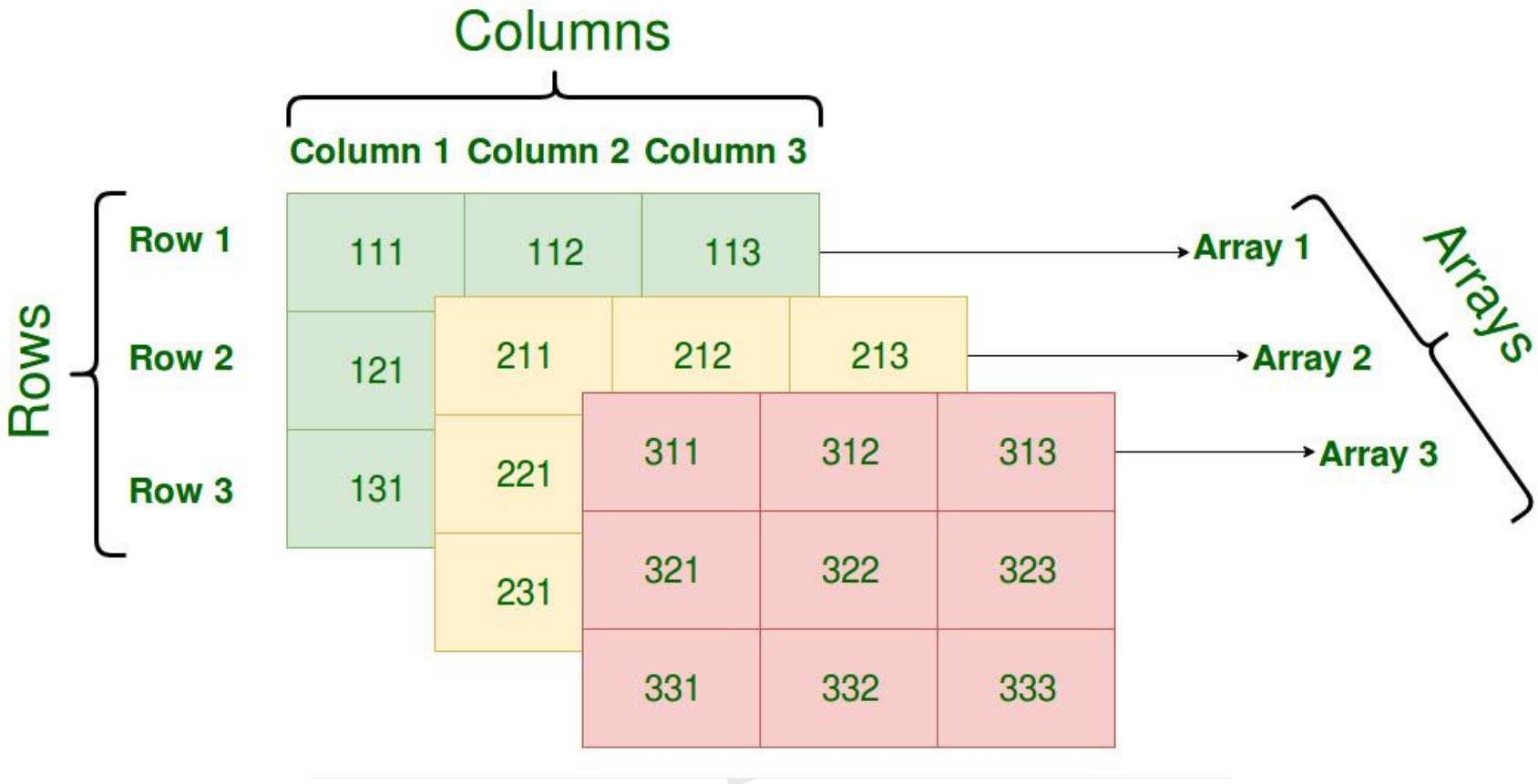
1	0
3	4



2D Arrays in C++

```
int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}

for (int i = 0; i < 3; i++) {
     for (int j = 0; j < 4; j++) {
          cout << x[i][j] << ‘ ’;
     }
     cout<< endl;
}

// a better way

int x[3][4] = {
               {0, 1 ,2 ,3},
               {4, 5 , 6 , 7},
               {8 , 9 , 10 , 11}
               }
```

3D Arrays in C++



3D Arrays in C++

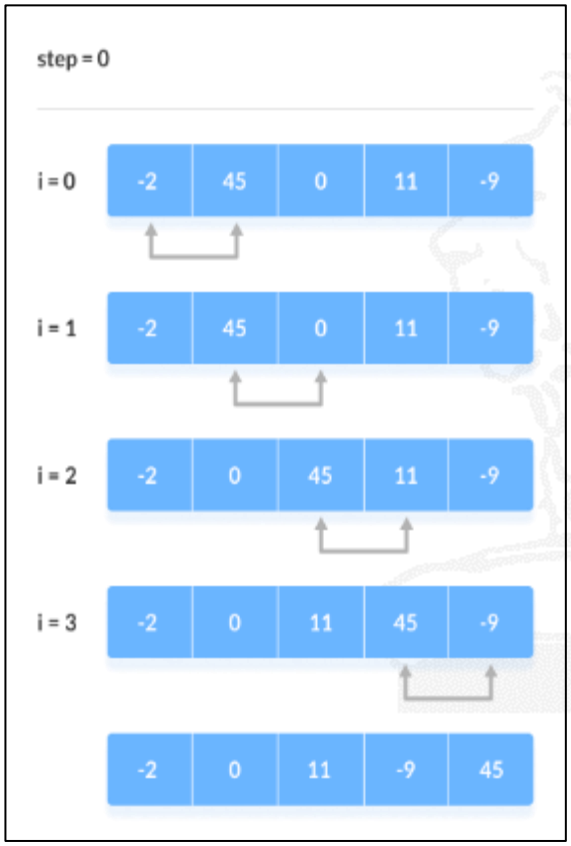
```
int x[2][3][4] =
 {
   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
   { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };

for (int i = 0; i < 2; i++) {
     for (int j = 0; j < 3; j++) {
         for (int k = 0; k < 4; k++) {
              cout << x[i][j][k] << ‘ ’;
         }
         cout<< endl;
     }
     cout << endl;
}
```

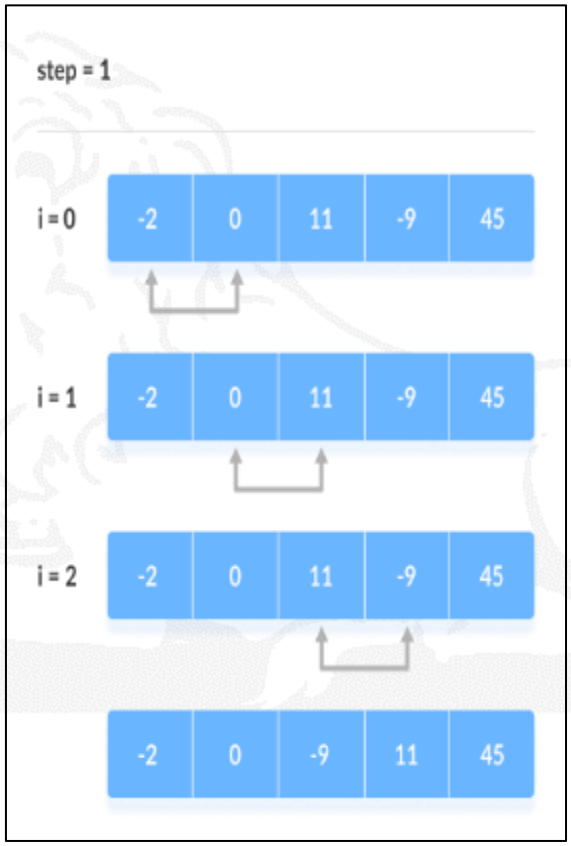
Arrays in C++: user input

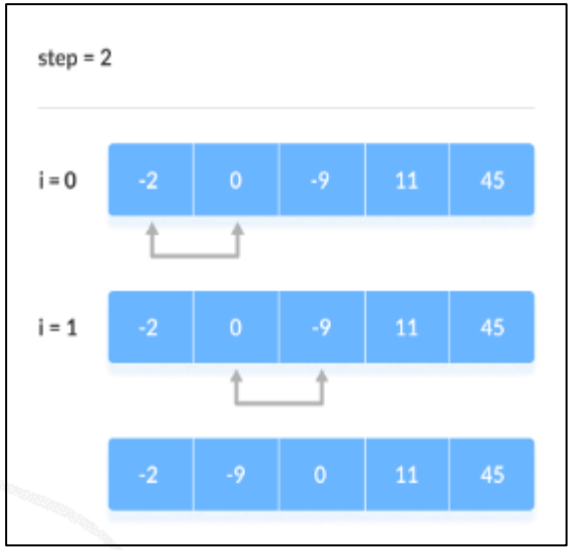
```
int rows = 2;
int cols = 2;
int mat[rows][cols];

for (int i = 0; i < rows; i++) {
     for (int j = 0; j < cols; j++) {
          cout <<"Enter the value for” << i <<"th row and "<< j <<"th column!";
          cin >> mat[i][j];
      }
}

for (int i = 0; i < rows; i++) {
     for (int j = 0; j < cols; j++) {
          cout << mat[i][j] ;
     }
     cout<< endl;
}
```

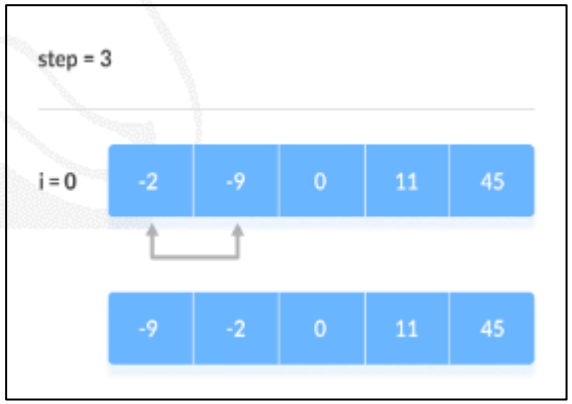
Arrays-Examples

Bubble Sorting



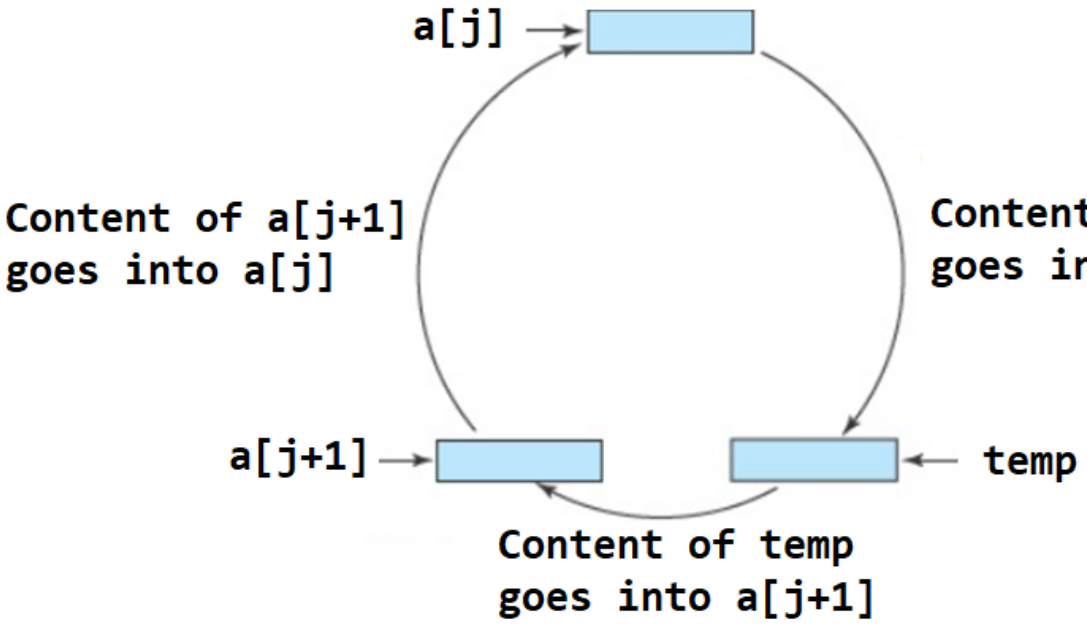






Arrays-Examples

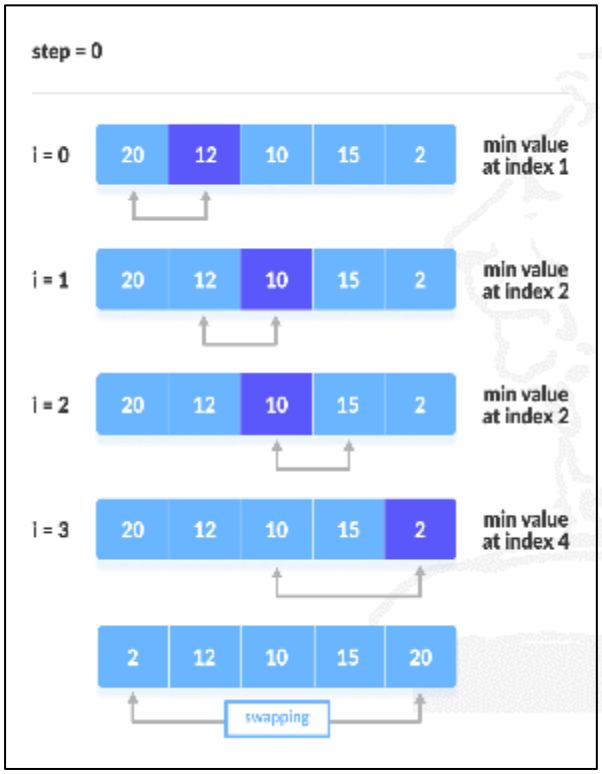
Bubble Sorting

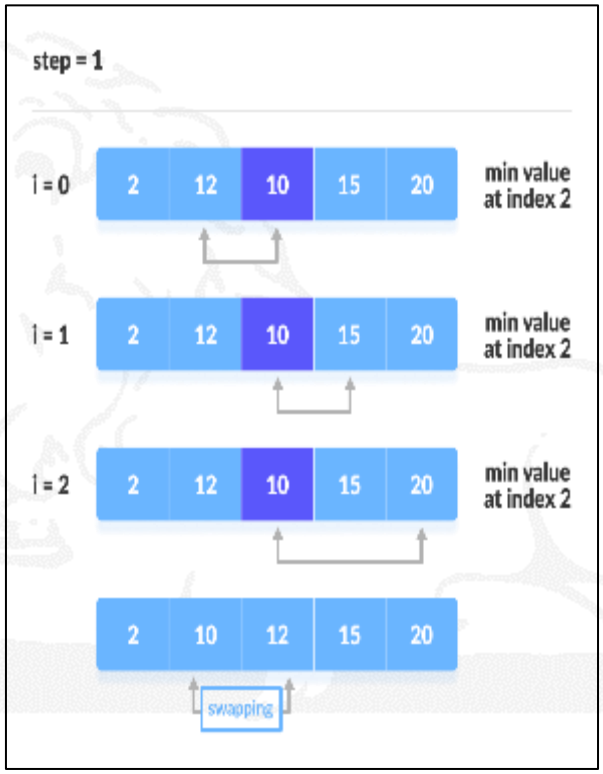


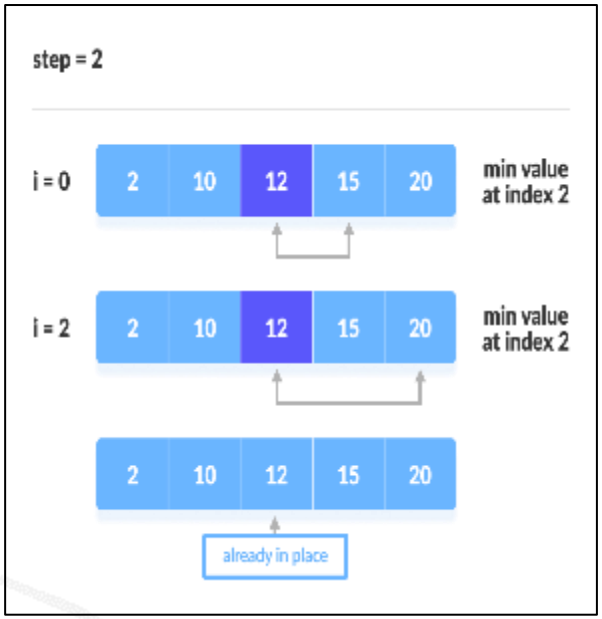

```
if (a[j]>a[j+1])
{
   int temp = a[j];
   a[j]     = a[j+1];
   a[j+1]   = temp;
}
```

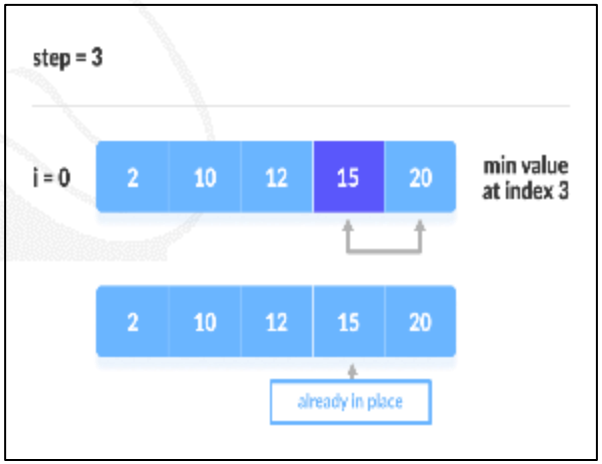
Arrays-Examples

Selection Sorting

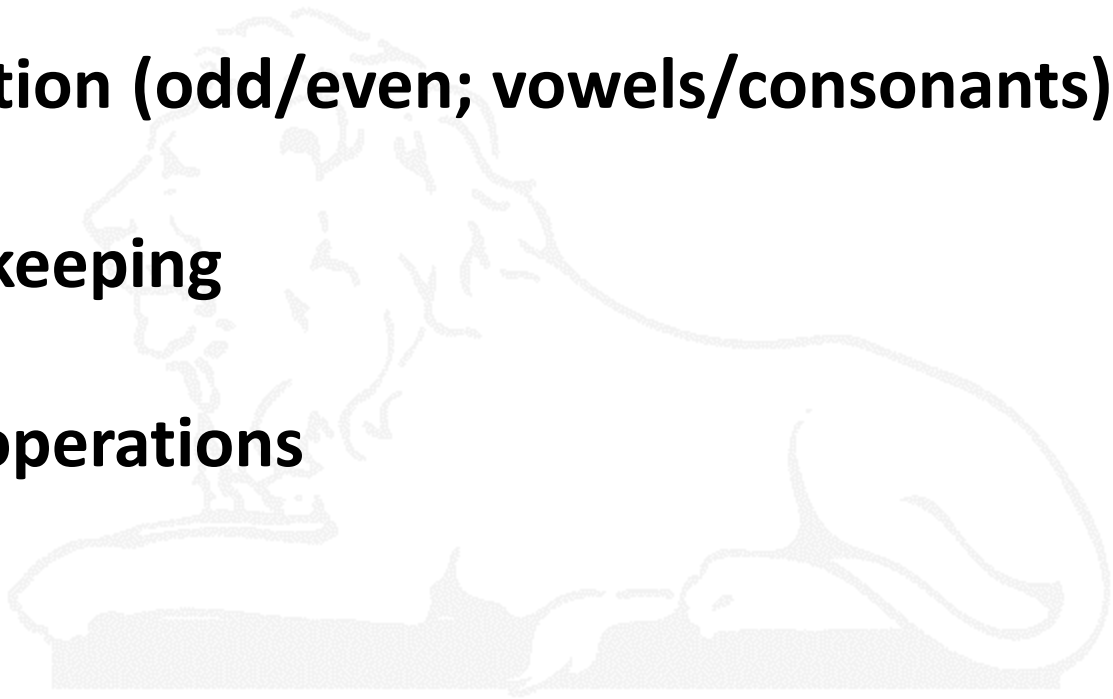








Arrays-Examples



Palindromes

Segregation (odd/even; vowels/consonants)

Record keeping

Matrix operations

Functions in C++

- A **function** is a block of code which **only** runs whenever it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform certain action(s).
- Functions are useful when a block of code is to be used again and again.
- **main()** is one of the pre-defined function

Functions in C++

If `f(x) = 2x + 5`,

then `f(1) = 7`,

`f(2) = 9`, and

`f(3) = 11`

`1`, `2`, and `3` are arguments

`7`, `9`, and `11` are the corresponding values (returned values)

predefined mathematical functions are:

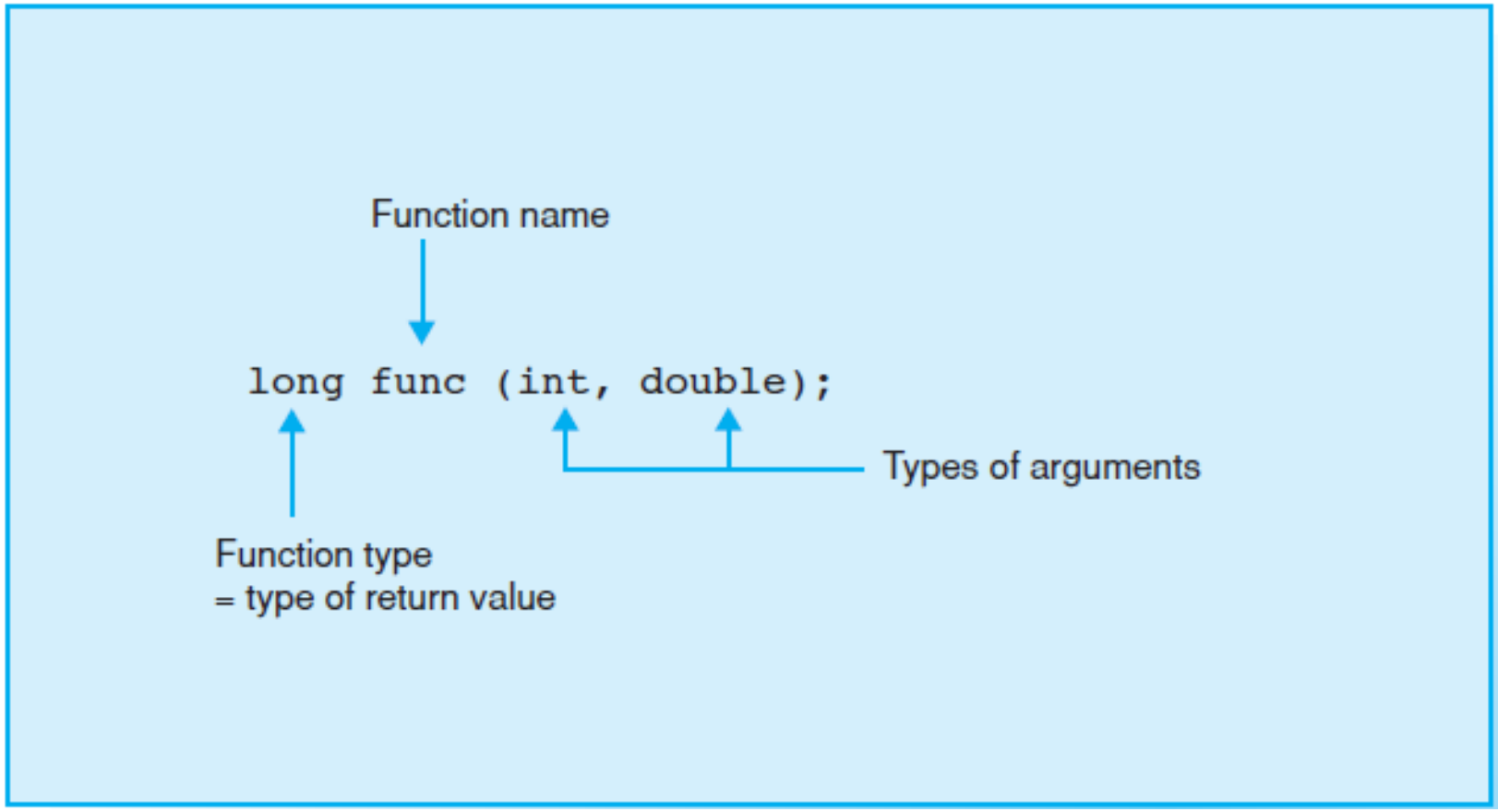
```
sqrt(x)
pow(x, y)
floor(x)
```

Functions in C++

```
void myFunction() {
  // code to be executed
}
```

- "myFunction" is name of the function
- "void" means that the function does not have a return value.
- "code to be executed" → body of the function (code that defines what the function should do)
- "()" → no data is passed

Functions in C++



Functions in C++

- Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.
- To call a function, write the function's name followed by two parentheses () and a semicolon ;

```
// Create a function
void fun() {
  cout << “My first function.\n”;
}

int main() {
   fun(); // calling the function
   fun(); // calling multiple times
   fun(); // calling multiple times
  return 0;
}
```

Try to declare the “fun” after “main”.

Functions in C++

- If there are many functions, they are declared above “main” and then functions can be defined after “main”.
- This will make the code better organized.

```
// function declaration
void fun();

// The main method
int main() {
  fun();  // call the function
  return 0;
}

// Function definition
void fun() {
  cout << “My first function.\n”;
}
```

Function parameters

```
void myFun(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

```
void myFun(string fname) {
  cout << fname << endl;
}

int main() {
  myFun("Anjaneya Dixit");
  myFun("Amit Agarwal");
  return 0;
}
```

Function default parameter

```
void myFun(string fname, string country="India") {
  cout << fname << "is from " << country << endl;
}

int main() {
  myFun("Amit Agarwal");
  myFun("Anjaneya Dixit", "USA");
  return 0;
}
```

Function return value

```
int myFun(string fname, string country="India") {
  cout << fname << "is from " << country << endl;
 return fname.size();
}

int main() {
  int s = myFun("Amit Agarwal");
  cout << s << endl;
  int k = myFun("Anjaneya Dixit", "USA");
  cout << k << endl;
  return 0;
}
```

Function return value

```
int myFun(string fname, string country {
  cout << fname << “is from ” << country << endl;
 return fname.size();
}
```

Is the following correct?

```
int main() {
  int z = myFun(“Amit”);
  cout << z << endl;
  return 0;
}
```

Function return value

```
int multiply(int x, int y) {
  return x * y;
}

int main() {
  int z = multiply(5, 3);
  cout << z << endl;
  return 0;
}
```

TODO: write a function for factorial ‘n’

Function: overloading

multiple functions can have the same name with different parameters

```
int myFun(int x)
float myFun(float x)
double myFun(double x, double y)

int sumInteger(int x, int y) {
  return x + y;
}

double sumDouble(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = sumInteger(3, 4);
  double myNum2 = sumDouble(2.1, 2.9);
  return 0;
}
```

Overload the function

Function: overloading

multiple functions can have the same name with different parameters

```
int myFun(int x)
float myFun(float x)
double myFun(double x, double y)

int sum(int x, int y) {
  return x + y;
}

double sum(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = sum(3, 4);
  double myNum2 = sum(2.1, 2.9);
  return 0;
}
```

Function: overloading

multiple functions can have the same name with different parameters

Conditions:

- Parameters should have different types (e.g., int vs double)
- Different number of parameters (e.g., two in one function, three in another)
- Different sequence of parameters (e.g., int and double vs double and int)

Function: random numbers

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

int main() {

    unsigned int seed;
    int n1, n2, n3;

    cout << "Enter  an integer value " ;
    cout << "(i.e., seed for Randome number generator" << endl;
    cin >> seed;
    srand(seed);

    n1 = rand();
    n2 = rand();
    n3 = rand();

    cout << n1 << ' ' << n2 << ' ' << n3 << endl;

    return 0;
}
```

Function: *srand()*

function

srand

<cstdlib>

```
void srand (unsigned int seed);
```

Initialize random number generator

The pseudo-random number generator is initialized using the argument passed as *seed*.

For every different *seed* value used in a call to `srand`, the pseudo-random number generator can be expected to generate a different succession of results in the subsequent calls to `rand`.

Two different initializations with the same *seed* will generate the same succession of results in subsequent calls to `rand`.

If *seed* is set to 1, the generator is reinitialized to its initial value and produces the same values as before any call to `rand` or `srand`.

In order to generate random-like numbers, `srand` is usually initialized to some distinctive runtime value, like the value returned by function `time` (declared in header <ctime>). This is distinctive enough for most trivial randomization needs.

Source: https://cplusplus.com/reference/cstdlib/srand/

Function: *srand()*

function

rand

<cstdlib>

```
int rand (void);
```

Generate random number

Returns a pseudo-random integral number in the range between 0 and RAND_MAX.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand.

RAND_MAX is a constant defined in <cstdlib>.

Source: https://cplusplus.com/reference/cstdlib/rand/

Function: random numbers

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

int main() {

    unsigned int seed;
    int n1, n2, n3;

    cout << "Enter  an integer value " ;
    cout << "(i.e., seed for Randome number generator" << endl;
    cin >> seed;
    srand(seed);

    n1 = rand();
    n2 = rand();
    n3 = rand();

    cout << n1 << ' ' << n2 << ' ' << n3 << endl;

    return 0;
}
```

TODO: would you always get **seed** as input?

TODO: how can you get a random number between 0 and 100?

Function: swap numbers

```
void swap(int variable1, int variable2) {
        variable1 = variable2;
        variable2 = variable1;
}
```

```
void swap(int variable1, int variable2) {
        int temp = variable1;
        variable1 = variable2;
        variable2 = temp;
}
```

Memory Address in C++

- Variable declaration → memory address is assigned to the variable
- The address can be accessed using **&** operator

```
string food = "Pizza";
cout << &food; //Outputs 0x6ffe00 or similar hexadecimal form
```

- Variable assignment → value is stored in the memory

- **&** operator can be used to get the **memory address** of a variable which is the location of where the variable is stored on the computer.

References in C++

- A reference variable is a "reference" to an existing variable, and it is created with the & operator
- The reference itself isn't an object (it has no identity; taking the address of a reference gives you the address of the referent)

```
string food = "Pizza";  // food variable
string &meal = food;    // reference to food (another name
for food)


cout << food << "\n";   // food variable
cout << meal << "\n";   // reference to food
cout << &meal << "\n"; // address of meal

food = “Pasta";
cout << food << "\n";
cout << meal << "\n";
```

TODO: Check the memory address of variable and reference variable.

Call-by-value vs call-by-reference

- With the **call-by-value,** ➔ the function takes only the value of the variable and does not change the variable in any way

- With the **call-by-reference** ➔ formal parameters are called as reference parameters ➔ the corresponding argument in a function call must be a **variable**

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

int square(int num) {
    int sqrNr = num*num;
    return sqrNr;
}

int main() {

    int num = 5;
    cout << square(num);

    return 0;
}
```

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

void square(int &num) {
    num = num*num;
}

int main() {

    int num = 5;
    square(num);
    cout << num;


    return 0;
}
```

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

int square(int num) {
    int sqrNr = num*num;
    num = sqrNr;
    return sqrNr;
}

int main() {

    int num = 5;
    cout << square(num) << endl;
    cout << num ;
    return 0;
}
```

Call-by-value vs call-by-reference

- The **formal parameters** for a function are listed in the function declaration and are used in the body of the function definition. A **formal parameter** (of any sort) is a kind of **blank** or **placeholder** that is *filled in* with something when the function is called.

- An **argument** is something that is *used to fill in a formal parameter*. When you write down a function call, the arguments are listed in parentheses after the function name. When the function call is executed, the arguments are "**plugged in**" for the formal parameters.

- The terms ***call-by-value*** and ***call-by-reference*** refer to the mechanism that is used in the "**plugging in**" process.
  - In the **call-by-value** method, only the **value of the argument** is used. In this call-by-value mechanism, the formal parameter is a **local variable** that is **initialized to the value** of the corresponding argument.
  - In the **call-by-reference** mechanism, the argument is a variable, and the **entire variable is used**. In the call-by-reference mechanism, the argument variable is **substituted for the formal parameter** so that any change that is made to the formal parameter is actually made to the argument variable.

References in C++

- References and pointers in C++ give you the ability to manipulate the data in the computer's memory - which can reduce the code and improve the performance.

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int a=6, b =8;
    swap(a, b);
    cout << a << endl;
    cout << b << endl;
    return 0;
}
```

References in C++

```
void figure_me_out(int& x, int y, int& z) {
        using namespace std;
        cout << x << " " << y << " " << z << endl;
        x = 1;
        y = 2;
        z = 3;
        cout << x << " " << y << " " << z << endl;
}
int main( ) {
        int a, b, c;
        a = 10;
        b = 20;
        c = 30;
        figure_me_out(a, b, c);
        cout << a << " " << b << " " << c;
return 0;
}
```

Output of this program?

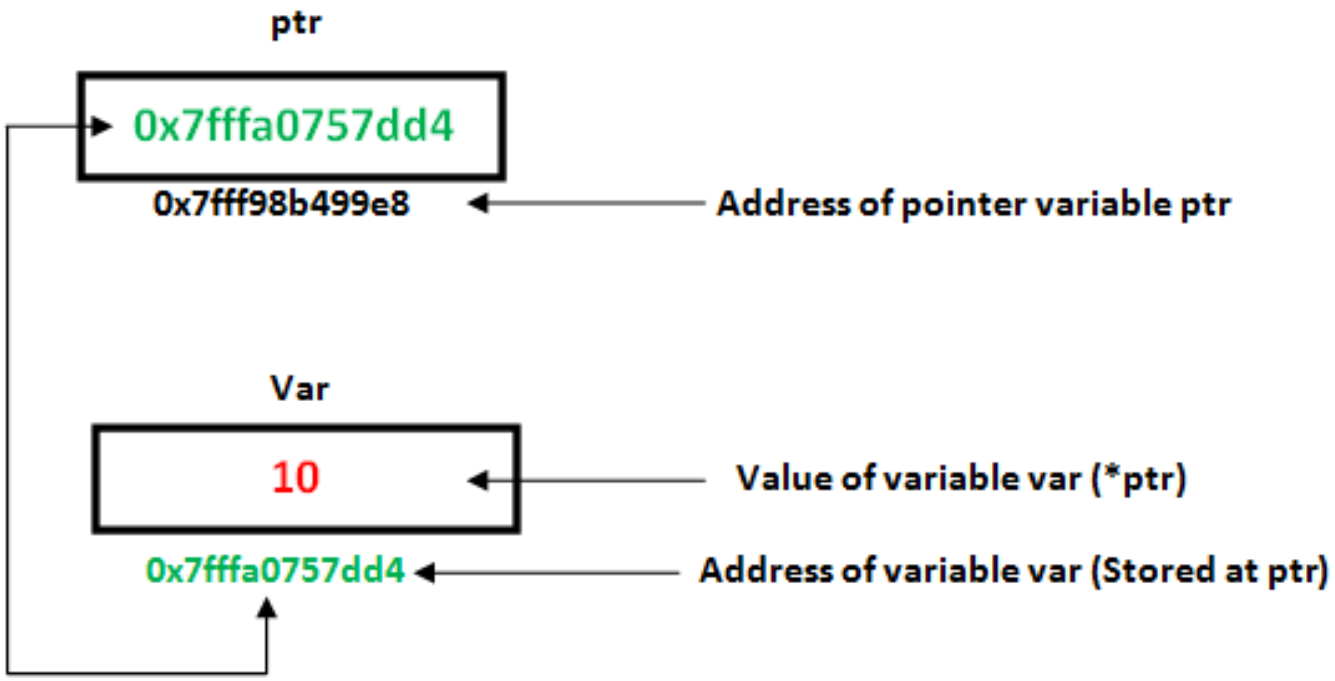
Pointers in C++

- A **pointer** is a variable that **stores the memory address as its value**.
- Pointers are a symbolic representation of addresses.
- A pointer variable points to a data type (like **int** or **string**) of the same type and is created with the * operator.
- Pointers enable programs to simulate **call-by-reference** as well as to create and manipulate dynamic data structures.

```
string* mystring;
string *mystring;
string * mystring;
```

- Frequent operations:
  - Define a pointer variable
  - Assign the address of a variable to a pointer
  - Access the value at the address in the pointer variable

Pointers in C++



Pointers

- when the variable is used as a **call-by-reference** argument, it is this **address**, not the identifier name of the variable, that is passed to the calling function.

- An address that is used to name a variable in this way (by giving the **address in memory where the variable starts**) is called a **pointer** because the address can be thought of as "pointing" to the variable.

- when a variable is a call-by-reference argument in a function call, the function is given this argument variable in the form of a pointer to the variable.

Pointers

- A variable to hold a pointer must be declared to have a **pointer type.**

*Type_Name *Variable_Name1, *Variable_Name2, . . .;*

- you cannot perform the normal arithmetic operations on pointers

- You can assign the value of one pointer variable to another pointer variable. This copies an address from one pointer variable to another pointer variable.

```
int num = 8;
int *ptr = &num;

cout << &ptr << endl;
cout << ptr << endl;
```

```
int num = 8;
int *ptr = num;

cout << &ptr << endl;
cout << ptr << endl;
```

Pointers in C++

```
string food = "Pizza";  // A food variable of type string
string* ptr = &food;    // A pointer variable, same type as that of
food variable with the name ptr, that stores the memory address of
food

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";
```

Dereference in C++

- It is also possible to use the pointer to get the value of the variable, by using the * operator (**dereference** * operator).

```
string food = "Pizza";  // Variable declaration
string* ptr = &food;    // Pointer declaration

// Reference: Output the memory address of food with the pointer (0x6dfed4)
cout << ptr << "\n";

// Dereference: Output the value of food with the pointer (Pizza)
cout << *ptr << "\n";
```

- When used in declaration (string* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

Modify pointers in C++

```
string food = "Pizza";
string* ptr = &food;

// Output the value of food (Pizza)
cout << food << "\n";

// Output the memory address of food (0x6dfed4)
cout << &food << "\n";

// Access the memory address of food and output its value (Pizza)
cout << *ptr << "\n";

// Change the value of the pointer
*ptr = "Hamburger";

// Output the new value of the pointer (Hamburger)
cout << *ptr << "\n";

// Output the new value of the food variable (Hamburger)
cout << food << "\n";
```

Modify pointers in C++

- Since a pointer can be used to refer to a variable, your program can **manipulate variables** even if the variables have **no identifiers** to name them.

- The operator new can be used to create variables that have no identifiers to serve as their names.

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

dynamic variables

```
int *ptr = new int;

cout << &ptr << endl;
cout << ptr << endl;
```

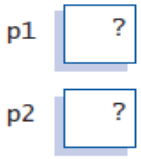
```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

```
int *ptr = new int(15);

cout << &ptr << endl;
cout << ptr << endl;
cout << *ptr << endl;
```

Modify pointers in C++

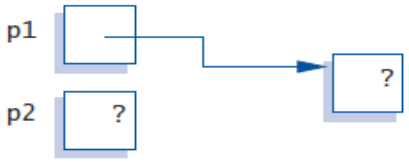
(a)

```
int *p1, *p2;
```



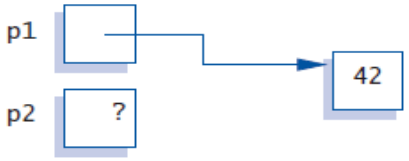
(b)

```
p1 = new int;
```



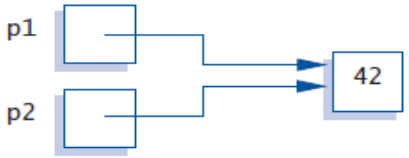
(c)

```
*p1 = 42;
```



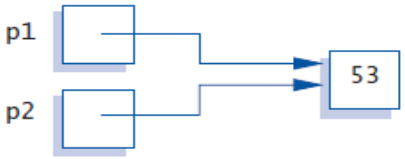
(d)

```
p2 = p1;
```



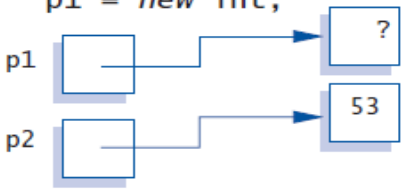
(e)

```
*p2 = 53;
```



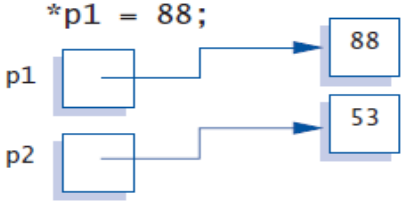
(f)

```
p1 = new int;
```



(g)

```
*p1 = 88;
```



Modify pointers in C++

```
p1 = p2;
```


```
*p1 = *p2;
```

Modify pointers in C++

```
struct Patient {
    int id;
    int height;
    int weight;
};

int main() {
    Patient patient;
    Patient* ptr = &patient;

    cout << ptr << endl;
    return 0;
}
```

```
cout << *ptr.weight << endl;
cout << (*ptr).weight << endl;
```

Which one is correct?

Order of precedence …

```
cout << (*ptr).weight << endl;
cout << ptr->weight << endl;
```

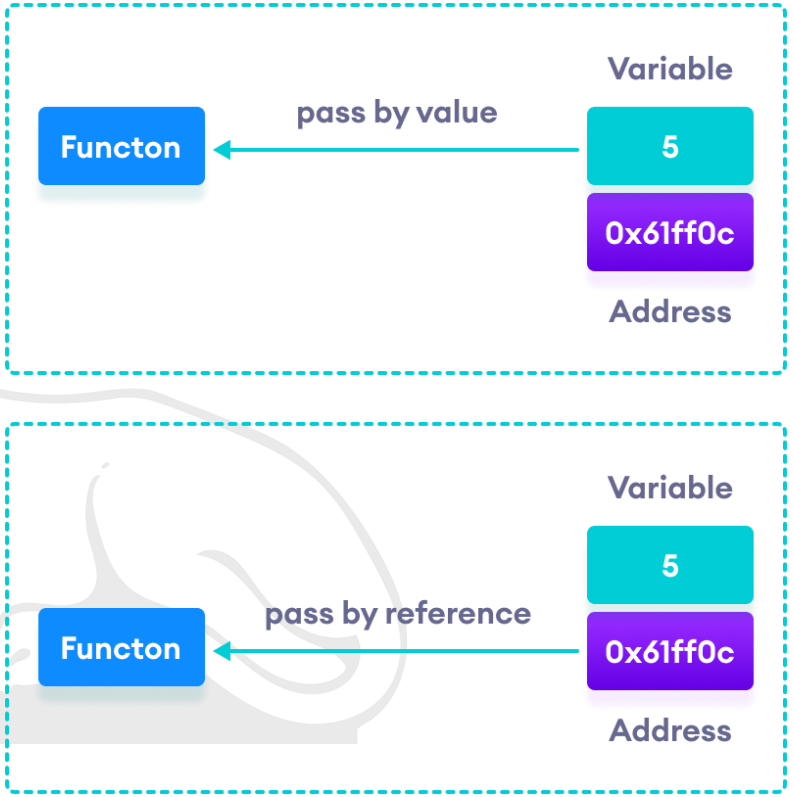
Double pointers in C++

```
int var = 20; // variable declaration and assignment
int *ptr; // pointer declaration
ptr = &var; // pointer assignment

cout << "Value at ptr = " << ptr << "\n";
cout << "Value at var = " << var << "\n";
cout << "Value at *ptr = " << *ptr << "\n";

*ptr = 30; // dereference ptr and assigning new value
cout << "Value at var = " << var << "\n";

int **ptr2 = &ptr; // pointer declaration and assignment

cout << "Value at ptr2 = " << ptr2 << "\n";
cout << "Value at *ptr2 = " << *ptr2 << "\n";
cout << "Value at **ptr2 = " << **ptr2 << "\n";

**ptr2 = 40; // dereference ptr and assigning new value
cout << "Value at var = " << var << "\n";
```

Pointers in C++

There are 3 ways to pass C++ arguments to a function:

- call-by-value
- call-by-reference with pointer argument
- call-by-reference with reference argument



Pointers in C++: call-by-value

```
int square(int n){
    cout << "address of n1 in square(): " << &n << "\n";
    n *= n;
    return n;
}

int main()
{
    int n1=8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    cout << "Square of n1: " << square(n1) << "\n";
    cout << "No change in n1: " << n1 << "\n";
    return 0;
}
```

```
address of n1 in main(): 0x6ffe0c
address of n1 in square(): 0x6ffde0
Square of n1: 64
No change in n1: 8
```

Pointers in C++:call-by-reference with pointer argument

```
void square(int *n){
    cout << "address of n1 in square(): " << n << "\n";
    *n *= *n;
}

int main()
{
    int n1=8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    square(&n1);
    cout << "Square of n1: " << n1 << "\n";
    cout << "Change reflected in n1: " << n1 << "\n";
    return 0;
}
```

```
address of n1 in main(): 0x6ffe0c
address of n1 in square(): 0x6ffe0c
Square of n1: 64
Change reflected in n1: 64
```

Pointers in C++:call-by-reference with reference argument

```
void square(int &n){
    cout << "address of n1 in square(): " << &n << "\n";
    n *= n;
}

int main()
{
    int n1=8;
    cout << "address of n1 in main(): " << &n1 << "\n";
    square(n1);
    cout << "Square of n1: " << n1 << "\n";
    cout << "Change reflected in n1: " << n1 << "\n";
    return 0;
}
```

```
int  x = 5;
int &y = x;

cout << &x << " : " << &y << "\n";
```

```
address of n1 in main(): 0x6ffe0c
address of n1 in square(): 0x6ffe0c
Square of n1: 64
Change reflected in n1: 64
```

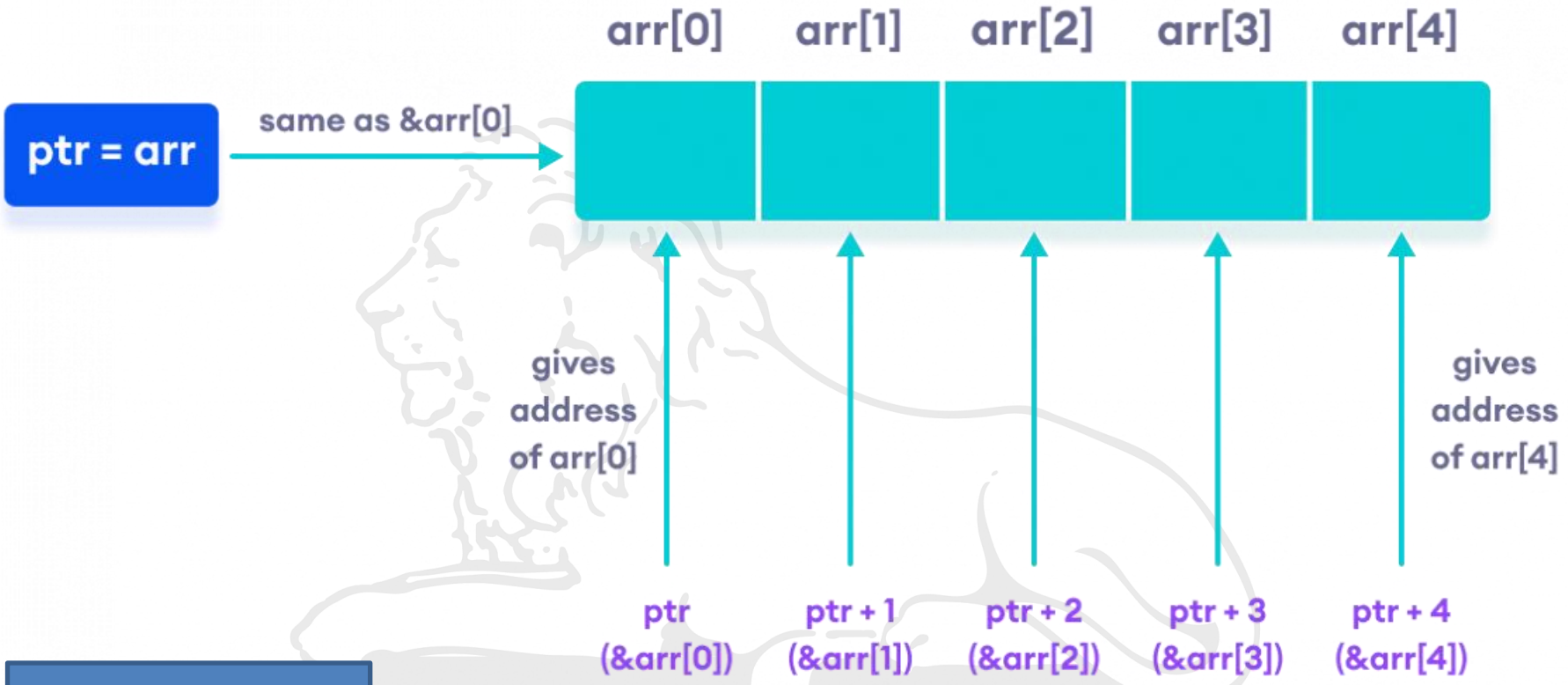
Pointers vs References in C++

```
int i=10; //simple or ordinary variable.
int *p=&i; //single pointer
int **pt=&p; //double pointer
int ***ptr=&pt; //triple pointer
// All the above pointers differ in the value they store or point to.
cout << "i=" << i << "\t" << "p=" << p << "\t" << "pt=" << pt << "\t" << "ptr=" << ptr << "\n";
int a=5; //simple or ordinary variable
int &S=a;
int &S0=S;
int &S1=S0;
cout << "a=" << a << "\t" << "S=" << S << "\t" << "S0=" << S0 << "\t" << "S1=" << S1 << "\n";
// All the above references do not differ in their values
// as they all refer to the same variable.
```

Pointers to Arrays in C++

```
//Declare an array
int val[3] = { 5, 10, 20 };

//declare pointer variable
int *ptr;

//Assign the address of val[0] to ptr
// We can use ptr=&val[0];(both are same)
ptr = val;
cout << "Elements of the array are: ";
cout << ptr[0] << " " << ptr[1] << " " << ptr[2] <<endl;

cout << ptr << endl;
cout << ptr+1 << endl;
cout << ptr+2 << endl;
```

Pointers to Arrays in C++



This is also called as arithmetic (incremental or decremental) pointers

Null Pointer in C++

- It is considered a good practice to assign the pointer NULL to a pointer variable in case exact address to be assigned to the pointer is unavailable.
- This is done at the time of variable declaration.
- A pointer that is assigned NULL is called a null pointer.

```
int *ptr = NULL;
```

- One can perform error handling in pointer related code e.g., dereference pointer variable only if it's not NULL

```
if (ptr) {
   cout << "Pointer is not null.";
}

if (ptr!=NULL) {
   cout << "Pointer is not null.";
}
```

Will pointer to a null pointer is also NULL?

```
char* np = NULL;
char** pnp = &np;
```

Pointer Arithmetic in C++

- pointer is an address which is a numeric value, thus, arithmetic operations can be performed.

```
//Declare an a
int val[3] = {

//declare poin
int *ptr;

//Assign the address of val[0] to ptr
// We can use ptr=&val[0];(both are same)
ptr = val;
cout << "Elements of the array are: ";
cout << ptr[0] << " " << ptr[1] << " " << ptr[2] <<endl;

cout << ptr << endl;
cout << ptr+1 << endl;
cout << ptr+2 << endl;
```

Try to understand the output of the following:

```
cout << ptr <<" "<<    ptr+1 <<" " <<  ptr+2 <<endl;
cout << ptr[0] <<" "<<ptr[1] <<" " <<  ptr[2] <<endl;
cout <<*ptr <<" "<<*(ptr+1) <<" " <<*(ptr+2) <<endl;
```

Pointer Arithmetic in C++

- pointer is an address which is a numeric value, thus, arithmetic operations can be performed.

```
//Declare an array
int val[3] = { 5, 10, 20 };

//declare pointer variable
int *ptr;

//Assign the address of val[0] to ptr
// We can use ptr=&val[0];(both are same)
ptr = &val[2];
cout << "Elements of the array are: ";
cout << ptr[0] << " " << ptr[-1] << " " << ptr[-2] <<endl;

cout << ptr << endl;          cout << *ptr << endl;
cout << ptr-1 << endl;        cout << *(ptr-1) << endl;
cout << ptr-2 << endl;        cout << *(ptr-2) << endl;
```

Pointer Comparison in C++

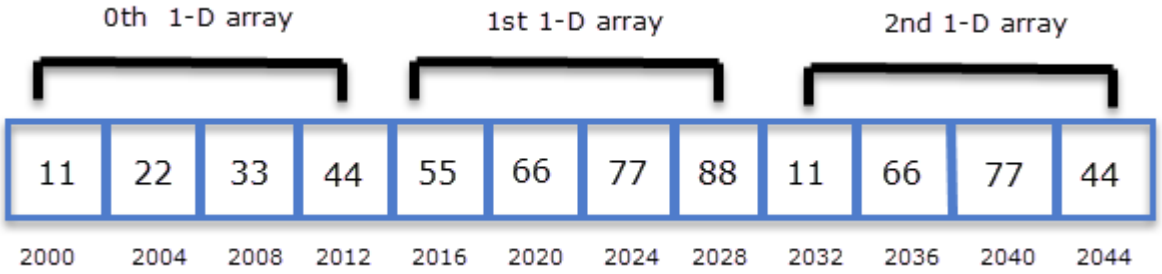
- pointer is an address which is a numeric value, thus, relational operations (==, >, <) can be performed.

```
//Declare an array
int val[3] = { 5, 10, 20 };

//declare pointer variable
int *ptr;

//Assign the address of val[0] to ptr
// We can use ptr=&val[0];(both are same)
ptr = val;
cout << "Elements of the array are: ";
cout << ptr[0] << " " << ptr[1] << " " << ptr[2] <<endl;

cout << ptr << endl;
cout << ptr+1 << endl;
cout << ptr+2 << endl;

bool boolVal = ptr <= ptr+1;
cout << std::boolalpha << boolVal << endl;
```

Pointers in C++

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

	Col 0	Col 1	Col 2	Col 3
Row 0	11	22	33	44
Row 1	55	66	77	88
Row 2	11	66	77	44



Pointer to 2D array in C++

```
int main() {
    int rows = 2;
    int cols = 2;

    int arr [rows][cols] = {{3,2},{5,9}};

    for (int i = 0; i < rows ; i ++) {
        for (int j =0; j < cols ; j ++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    for(int* ptr = &arr[0][0]; ptr <= &arr[rows-1][cols-1]; ptr++) {
        cout << "address: " << ptr << " and value at the address is " << *ptr << endl;
    }
    return 0;
}
```

Watch: https://www.youtube.com/watch?v=sHcnvZA2u88

Dynamic Memory Collection

- In C++, memory is typically divided into four parts –
  - **Code** (text): instructions
  - **Static/ global**
  - **Stack** - All the variables that are declared inside any function take memory from the stack.
  - **Heap** - It is unused memory in the program that is generally used for dynamic memory allocation.
- For storing the data, memory allocation can be done in two ways:
  - **Static allocation or compile-time allocation** - it means providing space for the variable. The **size and data type of the variable is known**, and it remains **constant** throughout the program.
  - **Dynamic allocation or run-time allocation** - The allocation in which memory is allocated dynamically. In this type of allocation, the exact size of the variable **is not known** in advance.

Dynamic Memory Collection

- Dynamically we can **allocate storage (**within the heap) while the program is in a running state, but **variables cannot be created "on the fly".**
- If dynamically allocated memory is not required anymore, it can be removed using **delete** operator, which de-allocates memory that was previously allocated by **new** operator.

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

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

- Dynamic memory allocation can make memory management more efficient. Especially for **arrays**, where a lot of the times we don't know the size of the array until the run time.

Dynamic Memory Collection

```
#include <iostream>
using namespace std;
int main() {
        double* val = NULL;
        val = new double;
        *val = 38184.26;
        cout << "Value is : " << *val << endl;
        delete val;
}
```

Note: ***reuse*** of memory is possible.

Dynamic Memory Collection

Why should we delete?

```
int* ptr;

// request memory in heap
ptr = new int;
//assign a value to the address
*ptr = 25;

// reassign
ptr = new int (100);
```

```
int* ptr;

// request memory in heap
ptr = new int;
//assign a value to the address
*ptr = 25;

//delete first
delete ptr;

// reassign
ptr = new int (100);
```

This will create garbage. Delete before reassigning to avoid it.

Dynamic Memory Collection

```
int main() {
    int* ptr;

    cout << ptr << endl;
    cout << &ptr << endl;

    ptr = new int;
    *ptr = 25;

    cout << ptr << endl;
    cout << &ptr << endl;
    cout << *ptr << endl;

    ptr = new int(100);

    cout << ptr << endl;
    cout << &ptr << endl;
    cout << *ptr << endl;

    return 0;
}
```

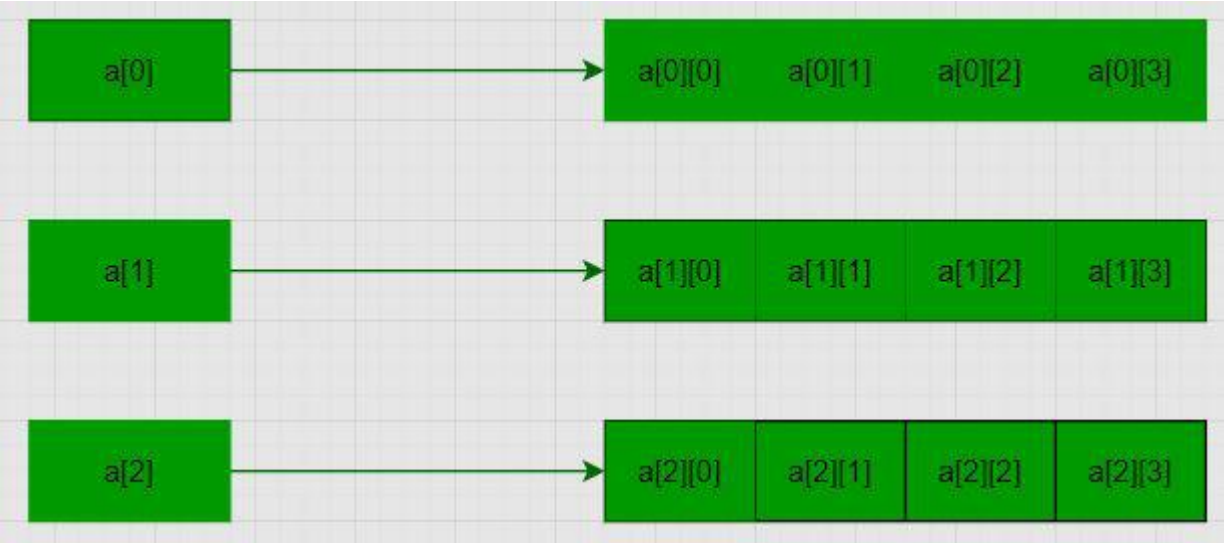
Try to understand the output of this program.

Dynamic Memory Collection

```
int main() {
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;
    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "Enter GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }

    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << " :" << *(ptr + i) << endl;
    }

    // ptr memory is released
    delete[] ptr;
    return 0;
}
```

Dynamic Memory Collection

```
int** a = new int*[10] ; // allocate an array of 10 int pointers → rows
```




```
for (int count = 0; count < 10; count++) {
        a[count] = new int[4]; // these are columns
}
```

Dynamic Memory Collection

```
// Declare memory block of size M
    int** a = new int*[m];


    for (int i = 0; i < m; i++) {
        // Declare a memory block of size n
        a[i] = new int[n];
    }
```

Is it possible to have different number of columns in each row?

```
 for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            // Assign values to the memory blocks created
            a[i][j] = i+j;
        }
    }
```

```
//Delete the array created
for(int i=0; i<m; i++)  {  //To delete the inner arrays
    delete [] a[i];
}
delete [] a;  //To delete the outer array which contained the pointers of all the inner arrays
```

Dynamic Memory Collection

```
int** a = new int*[3];
for (int i = 0; i<3; i++){
    a[i] = new int[2];
}

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
        a[i][j] = i+j;
    }
}

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
        cout << a[i][j] << " ";
    }
    cout <<endl;
}

cout << endl;

for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
        cout << *(*(a+i)+j) << " ";
    }
    cout <<endl;
}
```

Thanks …

amitfce@iitr.ac.in