-

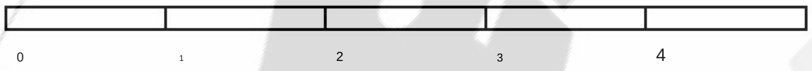
UNIT -2

Array - This is a collection of variables of the same data type.

• Whatever can be the data type of Array. • Array's

variables are stored at their nearest memory location. • Initialization of Array starts from

'0'.



• The first element of the array has the lowest memory address.

Why is Array used?

If the names of any students are to be stored, then separate variables have to be made for them.

for eg.
char stud1 = "Rakesh" stud2 = "Uday", stud3 = "Raj" ;

Only one array_variable has to be created for all these names in Array. for eg. char arr[] =

{ "Rakesh", "Uday", "Raj" };

There are two types of Array.

• Single/One Dimensional Array • Multi Dimensional Array

Single/One Dimensional Array

Syntax for Single Dimensional Array Declaration

data_type array_name[size_of_array];

for eg.

```
int arr[5];
```

Syntax for Single Dimensional Array Initialization

data_type array_name[size_of_array] = {value1, value2,...,value n};

for eg.

```
int arr[5] = {1, 2, 3, 4, 5};
```

Example for One/Single Dimensional Array

Source Code :

```
#include <iostream.h>

using namespace std;

int main(){

int i,num[5]={1, 2, 3, 4, 5};

for(i=0;i<5;i++){

cout<<"Array Number is "<<num[i]<<endl;

}
return 0;

}
```

Output :

```
Array Number is 1

Array Number is 2

Array Number is 3

Array Number is 4

Array Number is 5
```

Array Memory Representation

• Here there is a 4-4 buffer in the address because the size of the integer is 4bytes due to the system being 32-bit. That's why there is a buffer of 4-4 in the Memory Address. • If the data type of the array was character, then there would have been a 1-1 buffer in the address.

index of array	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Array Elements	1	2	3	4	5
Memory Address	0x69fee8	0x69feec	0x69fee0	0x69fef4	0x69fef8

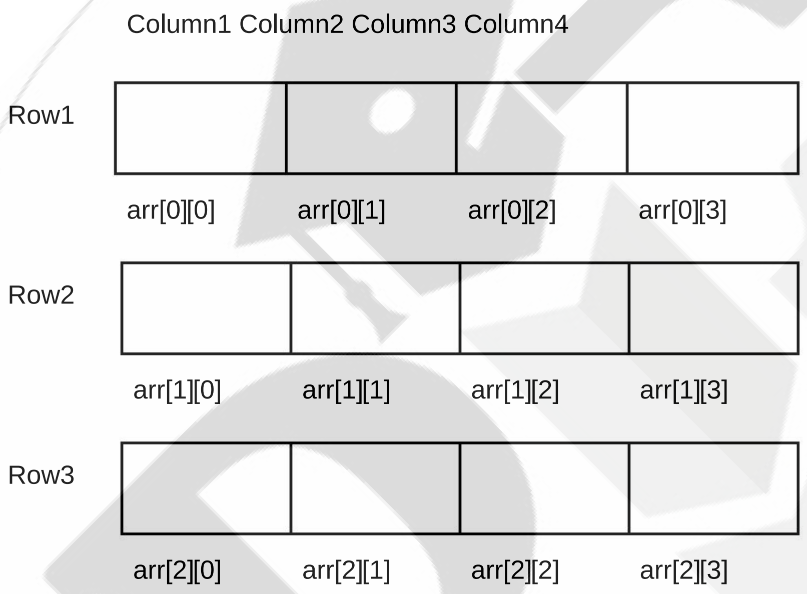
-

Multi Dimensional Array

Syntax for Multi Dimentional Array Declaration

data_type array_name[row_size][column_size]; for eg.

```
int arr[3][4];
```



Syntax for Multi Dimentional Array Initialization

for eg.

```
int arr[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12}; OR


int arr[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

Example for Multi Dimensional Array

Source Code :

```
#include <iostream.h>

using namespace std;

int main() {

int arr[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };

int i, j;

for ( i = 0; i < 3; i++ ){

for ( j = 0; j < 4; j++ ){

cout<<"["<<i<<"]["<<j<<"] = "<<arr[i][j]<<endl;

}

}

return 0;

}
```

Output :

```
[0][0]=1

[0][1]=2

[0][2]=3

[0][3]=4

[1][0]=5

[1][1]=6

[1][2]=7

[1][3]=8

[2][0]=9
```

```
[2][1]=10

[2][2]=11

[2][3]=12
```

What is pointer

introduction

Pointer is a main feature of C++ language. Pointer is a special variable which is used to store the memory address of another variable. The pointer is also declared in the same way as an ordinary variable is declared.

What is Pointer? (Definition)

A variable that stores the address of another variable is called a pointer.

Declaration

Like any other variable, pointer also needs to be declared before it is used. (*) sign is used to declare the variable of pointer, it is called pointer operation.

```
data_type * pointer_name  program
```

code of Pointer

```
#include<iostream.h>
#include<conio.h>  void
main()
{
   int number[50];*ptr  int n,i;
   cout<<"\n Enter the count \n;
   cin>>n;
```

```
  cout<<"\n Enter the number one by one;  for(i=0; i<n; i++)

 {
  cin>>numbers[i];  }

ptr = numbers;  int
sum = 0;  for (i=0; i<n;
i++)  {

  if (*ptr %2 == 0)
sum = sum+ **ptr;  ptr+
+;  }  cout<<"\n\n sum of
even numbers ="<<sum;  }
```

Output

```
Enter the count
5
Enter the number one by one  10

16
25
45
34
sum of even numbers = 60
```

Uses of Pointer

1. Returning more than one value in all functions.
2. To access memory elements.
3. Pointers are more efficient in handling memory and data tables.
4. Pointers reduce the time duration and complexity of the program.
5. To access the memory address of a variable.
6. To pass array and string in dynamic memory.

7. In passing array and string in a function.

8. In doing low level programming.

Pointer to function

Function is similar to variable which has a memory location. The function pointer is another important feature of C++. Just as integer, character and floats have addresses in memory, functions also have physical addresses in memory. This address is the entry point of the function which is assigned to the pointer and the pointer can be used to invoke the function.

Program code

```
#include<iostream.h>
#include<conio.h>
class A

{  private:
  void swap(int* p*int*a)  {  int*r;  *r =
   p;  *p = *a;  *a = *r;  }  };  void
  main()







{  int x,y;
  cout<<"Enter two numbers"<<endl;  cin>>x>>y;
  A ob;  ob.swap(&x,&y);  cout<<"endl<<x<<y<<endl;
  cout<<y;  getch();




}
```

Output

```
Address
Enter two numbers
5
4
4
6
```

Pointer to Pointer

Pointer to pointer is used to bring flexibility in array and use pointer in function.

Syntax

```
data_type **            pte to ptr
```

example

```
#include<iostream.h>
#include <conio.h>
void main()  {

  int * i ptr  int **
               ptr i ptr;
  int data;
  int data;
  ipt = & data;  ptr i ptr
  = & iptr;  * i ptr = 100;

  printf("The variable data contains : %d\n", data);  **ptr i ptr = 200;  data = 300;


  printf("ptr i ptr is pointing to = %d\n" *ptr i ptr);  getch();  }
```

Pointer to Array

The way array can be created from a common data_type like integer, float, character etc. In the same way an array is declared with a pointer, in the same way an int array is a group of int values. Similarly, array of pointer is a group of many memory addresses. The memory addresses in the array of pointers can be the surfaces of different variables.

Syntax

```
<data_type> * pointer_name[size];
```

example

```
#include<iostram.h>
#include<conio.h>  void
main()

{  static int # a[4]={1,2,3,4};  int i, n; temp;  n=4;
  cout<<"contents of the array"<<endl;  for(i=0;
  i<=n-1; i++)



{  temp= *(&(a)[0]+(i);  cout<<"value
  =<<temp<<endl;  }  };
```

Pointer to Object

Just as there are pointers to other types of variables, in the same way we can also create pointers to Objects, Pointers specify the objects created by a class. Pointers to object are the special pointers specifying the object of a class.

Syntax

```
class_name * object pointer_name
```

example

```
#include<iostream.h>
```

```
#include <conio.h>
class product  {

  int code;  float
  price;  public:

void get_data (int a, int b)  {  code=a;
 price=b;  }



void main()
 {
  cout<<"code"<<code<<endl;;
  cout<<"price"<<price<<endl;  }

};
void main()

{  clrscr();
  product x;
  product *  ptr = & x;  ptr -
  get_data(100,60.50);  ptr - show();
  getch();  }
```

Call By Value ÿÿ Call By Reference -

There are two types of parameters in the function.

1. Formal Parameter
2. Actual Parameter

Formal Parameter

The parameter which is written in the declaration and definition of the function is called Formal Parameter.

for eg.

```
void swap(int x, int y); // Formal Parameters
int main(){
int a=2; b=10
swap (a, b)
}
void swap (int x, int y){                    //Formal Parameters
.........
.........
}
```

The parameter which is written in the function call is called Actual Parameter.

for eg.

```
void swap(int x, int y);
int main(){
int a=2; b=10
swap (a, b)                //Actual Parameter
}
void swap (int x, int y){
```

```
..........

..........

}
```

There are two types of Function Calling.

1. Call By Value
2. Call By Reference

1. Call By Value

• In Call By Value the value of the variable as a parameter to the function • In pass is done.

Call By Value the value of the Actual Parameter is copied to the Formal Parameter.

Here the variables 'a' and 'b' are passed to the function 'add' in the program.

Here the values of 'a' and 'b' are copied to the variables 'x' and 'y'.

for eg.

```
#include <iostream.h>

using namespace std;


void swap(int x, int y);
```

-

```
int main(){

int a = 2, b = 10;
            cout<<"Before Swapping a = "<<a<<" and b "<<b<<endl;
            swap(a, b);
}
void swap(int x, int y)
{
int temp;
            temp = x;
            x=y;
            y = temp;
            cout<<"After Swapping a = "<<x<<" and b "<<y;
}
```

Output :

```
Before Swapping a = 2 and b 10
After Swapping a = 10 and b 2
```

2. Call By Reference

• In Call By Reference, the address of the variable is given as a parameter to the function. pass is done.

• In Call By Value the value of Actual Parameter is not copied to Formal Parameter.

Here the variable 'a' and 'b' address have been passed to the function 'add' in the program.

Here the values of 'a' and 'b' are not copied to the variables 'x' and 'y'.

Yellowsurf holds the address of the variables.

for eg.

```
#include <iostream.h>

using namespace std;


void swap(int *x, int *y);

int main(){

int a = 2, b = 10

        cout<<"Before Swapping a = "<<a<<" and b "<<b<<endl;

        swap(&a, &b);

}

void swap(int *x, int *y)

{

int temp;

        temp = *x;

        *x = *y;

        *y = temp;

        cout<<"After Swapping a = "<<*x<<" and b "<<*y;
}
```

Output :

```
Before Swapping a = 2 and b 10

After Swapping a = 10 and b 2
```

Introduction for Memory Allocation

Dynamic Memory Allocation This is a very good feature for c++.

When variable is declared, then variable; Memory is allocated according to the data type.

If the user created a variable of integer data type, then memory of 2 bytes on 16-bit or 4 bytes on 32-bit

is allocated.

There are two types for memory allocation.

1. Compile-time/Static Memory Allocation 2.
Run-time/Dynamic Memory Allocation

1. Compile-time/ Static Memory Allocation

• When the variable is created, the compiler allocates memory at compile-time.

goes

-

• When the variable is created, then from which data type it is created, then the compiler allocates that memory only according to it.

• Declaration of variable is done at Compile-time.

For example,

4 bytes for 32-bit Integer

1 byte for character 4

bytes for float

array If you allocate memory at compile-time, then the space of this memory is constant. • When

For example,

int arr[10];

When such an array is created, only 10 integer values can be declared in it and each variable will allocate memory of 4 bytes.

2. Run-time/ Dynamic Memory Allocation

• Memory is allocated at Run-time in Dynamic Memory Allocation. • Memory is

allocated when the program runs in Dynamic Memory Allocation. • Declaration of variables are created according to need. • Here variables

is not done in Dynamic Memory Allocation. just at compile-time

The declaration of the variable is done.

In C programming, malloc, calloc, realloc and free are used for dynamic allocation, similarly there are two such operators in C++, which are used for dynamic memory.

Allocation is done for.

1. new operator
2. delete operator

1. new operator

The new operator is used to allocate dynamic memory.

This memory allocation; Occurs at run-time

Syntax for new

Here dynamic memory can be allocated with any data type and array with new operator. Here the class of memory allocation can also be taken.

```
new data_type

new data_type[array_size];
```

For Example.

```
new int;

new int[10];
```

When dynamic memory allocation is to be done then 'pointer' is used. When new operator is a dynamic allocation then its address; The new operator set is stored in the given pointer.

Syntax for new operator with pointer

```
pointer = new data_type;

pointer = new data_type[array_size]; //for Array
```

If seen in the example of Dynamic Memory Allocation, then there the address of int with new; Int *ptr has been taken to store and the address of int will be returned from new in ptr and will be stored inside ptr.

As in Pointer, if you want to hold the address of integer data type variable, then the pointer is also of the same data type, similarly if you want to store a new int address, then the pointer should also be of nteger.

For Example

```
int *ptr;

ptr = new int;

ptr = new int[50]; //for array
```

or

```
int *ptr = new int;

int *ptr = new int[50]; //for array
```

The size of the array is also defined by any user.

For Example

```
int a;

eat>>a;

int *ptr;

ptr = new int[a]; or int *ptr = new int[a];
```

-

delete Operator

The delete operator de-allocates the allocated dynamic memory.

When new operator; Allocates memory, then de-allocates or frees the memory allocated with to the delete operator.

new operator is used in program, then delete operator is used when program is

Syntax for delete

```
delete pointer_name;

delete []pointer_name; //for array
```

Example for delete

```
delete ptr;

delete []ptr; // for array
```

Example for new and delete operator

Source Code :

```
#include <iostream.h>

using namespace std;


int main(){
```

```
int *ptr;

ptr = new int;



cout<<"Enter value of ptr : ";

cin>>*ptr;

cout << "Value is ptr :                    " <<*ptr<< endl;



delete ptr;



return 0;

}
```

Output :

```
Enter value of ptr : 2

Value is ptr : 2
```

Example for new and delete operator with array

Source Code :

```
#include <iostream.h>

using namespace std;



int main(){
```

```
int a, i;

int *ptr;

        cout<<"Enter number of elements : ";

        eat>>a;

ptr = new int[a];

        cout<<"Enter "<<a<<" elements "<<endl;

for(i=1; i<=a; i++){

        cout<<"Element "<<i<<" : ";

        cin>>ptr[i];

}

cout<<"Entered Elements :"<<endl;

for(i=1; i<=a; i++){



        cout<<ptr[i]<<endl;

}



delete []ptr;

return 0;

}
```

Output :

```
Enter number of elements : 4

Enter 4 elements
```

```
Element 1 : 6

Element 2 : 4

Element 3 : 8

Element 4 : 9

Entered Elements :

6

4

8

9
```

CLASS WITH CONSTRUCTOR, DESTRUCTOR AND ARRAY

Source Code :

```
#include <iostream.h>

using namespace std;


class A{

public:

        A(){

        cout << "Constructor"<<endl;

}

        ~A(){

        cout << "Destructor"<<endl;
}

};

int main()

{

        A *a = new A[3];

        delete[] a;

return 0;

}
```

Output :

```
Constructor

Constructor

Constructor

Destructor

Destructor

Destructor
```

Introduction to Constructors

Constructor is a type of member function ~~that initializes~~ the object of the class. In this function , you assign the initial ~~value to~~ the data members (variables) of the class . Whenever a new object of a class is created, then the constructor of that class is ~~called.~~ happens and all the statements given in it are executed constructor provides you the ability to perform the necessary tasks before using any object like you can initialize class members variables, establish connection to database/server, etc. For example suppose you have created a class whose name is product in this class you have declared 2 variable price and batchld as given below

```
class product
{
  private:
            int price;
             int batchld;
};
```

When you create an object of this class, the default constructor is called by c++ and 0 values are assigned to the price and batchld variables, and zero is the initial value for the integer.

If you want, using the constructor itself, you can assign these variables with your desired value or with the values that the user will pass as arguments while creating the object. Below is an example of assigning different values to these variables through the constructor.

```
class product
{
    private:
```

```
    int price;

    int batchld;

public:

product() //constructor

  {

price=300;

batchld=1005;

   }

};
```

In the above example, the price and batchld variables have been assigned 300 and 1005 values respectively inside the constructor. Whenever the object of this class is created, these values will be automatically assigned to these variables.

As I told you, if you want, you can also initialize the class variable with the values passed by the user, such a constructor is called a parameterized constructor, you will be told about it further, but before that let's see what are the features of the constructor.

Characteristics of constructor-

-

Some features of the following constructors are being given, knowing about which you can understand the constructor in a better way.

1. The constructor should be declared in the public access modifier section 2. You can design the constructor inside the class as well as outside the class can do
3. Constructor name is same as class name 4. Constructors are automatically called when object is created 5. No return type is defined with constructors and they do not return any value

6. No class can inherit the constructor, however, a derived class can call the constructor of the base class.
7. Constructor cannot be declared virtual

Let us now try to learn about the different constructor provided by c++

Type of constructor:

Below you are being told about the different types of constructors available in c++, so let's try to learn about them.

default constructor

You do not need to design the default constructor, when you design another constructor, the default constructor is automatically created and called by c++.

normal constructor

Normal constructors are those to which you cannot pass arguments, in such constructors you manually initialize the variables. The general syntax of normal constructors is being given below.

```
class-name()
{
//statement to be executed
}
```

Let us now try to understand it with the help of an example, in the introduction section you were told to create a normal constructor inside the class, so let us now see how to design a normal constructor outside the class.

```
#include<iostream>
Using namespace std;
Class student
{
Private:
Int rollno;
Int standard;
Public:
Student(); //constructor declaration
Void displayStu();
```

-

```
};

Student::student(void)

{

Rollno=1;

Standard=10;

}

Void student::displayStu(void)

{

Cout<<"roll no is :"<<rollno<<endl;

Cout<<"standard is :"<<standard<<"th"<<endl;

}

Int main()

{

Student obj1;

Obj1.displayStu();

Return 0;

}
```

The above program generates the following output

```
Roll no is :1
```

standard is :10th

parameterized constructor In

parameterized constructor you can design parameters while creating object you can pass argument to parameterized constructor which can be assigned to class variable which can be assigned to class variable general syntax of parameterized constructor is being given to you below

```
class-name(parameters list)

{

//standard to be executed

}
```

parameterized constructor is being explained with the help of below example

```
class square

{

        Private:

                Int whether;

        Public:

                Square(int n);

                 Int calsquare();

};

Square::square(int n)
```

```
{
      Num=n;
}
Int square::calsquare()
{
      Return num * num;
}
Int main()
    {
     Int result;
     Square obj1(5);
     Result=obj1.calsquare();
     Cout<<"square of 5 is :"result;
     Return 0;
}
```

The above program generates the following output

```
square of 5 is :25
```

copy constructor

If you want, you can also initialize another object from the values of one object, for this you use the copy constructor, the object is passed as an argument in the copy constructor.

When you create a new object, all the values of the object that you give as an argument are copied to the new object, that's why this type of constructor is called copy. The general syntax of the copy constructor is being given below.

```
class-name(class-name & obj)
{
// statement to be executed
}
```

Come, now let's try to understand it through an example

```
Class furniture
{
        Private:
                Int price;
        Public:
                 Furniture(int n) //parameterized constructor
                  {
                        Price=n;
                  }
```

```
            Furniture(furniture & obj) //copy constructor

            {

                Price=obj.price;

            }

            Void showprice(void);

};

Void furniture::showprice(void)

{

Cout<<"price of this object is :"<<price;

}

Int main()

{

    Furniture obj(5);

    Obj1.showprice();

    Furniture obj2(obj1); //passing obj1 as argument for copy constructor

    Obj2.showprice();

}
```

The above program generates the following output

```
price of this object is :5
```

destroyers

The members of the destructors class have functions that destroy the object only when executed, as soon as an object goes out of scope, the destructor is called and the object is destroyed, the destructor is automatically called.

Like constructor, destructor is also designed with class name but in destructor you put tilde(~) symbol before class name, parameters are never designed in destructor, general syntax of destructor is being given below ~class-name()

```
{

//statement to be executed

}
```

Let us now try to understand the use of destructor through an example

```
#include<iostream>

Using namespace std;

Class MyClass

{

   Public:

        MyClass()

        {

           Cout<<"object is created"<<endl;
```

-

```
              }

            ~MyClass()

              {

                        Cout<<"object is destroyed";

              }

};

Int main()

{

        MyClass obj1;

       If(3>5)

        {

      MyClass ob2;

     }

}
```

In the above example, as soon as the compiler comes out of the if statement, obj2 gets destroyed, this program generates the following output

```
object is created

object is destroyed
```