C++ OOPs concept

OOPs :-

Object-oriented programming is a methodology or paradigm to design a program using class & object.

The main aim of OOP is to bind together the data & the functions that operate on them so that no other part of the code can access this data except that function.

Some basic concepts that act as the building blocks of OOPs :-

i) class
ii) object
iii) Encapsulation
iv) Abstraction
v) Polymorphism
vi) Inheritance
vii) Dynamic Binding
viii) Message passing

Advantages of OOPs :-

i) Faster & easier to execute
ii) Provides a clear structure of the programs.
iii) Helps to keep the c++ code DRY "Don't Repeat Yourself"
iv) Makes the code easier to maintain, modify & debug.
v) Makes it possible to create full reusable applications with less code & shorter ~~time~~ development time.

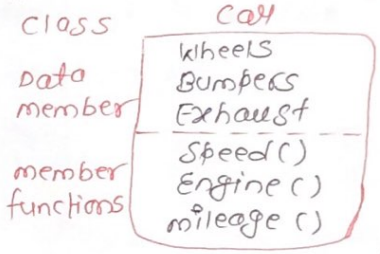
Class & objects.

* A class is a blueprint of the object. collection of objects. It is a logical entity A class contains data members (variables) & member functions. These member functions are used to manipulate the data members inside the class. Data members & member functions can be accessed & used by creating an instance of that class.

* An object can be defined as an entity that has a state & behaviour, or in other words, anything that exists physically in the world is called an object. An object is an instance of a class.

class	object
fruit	Apple Banana Mango

class	object
car	Audi Toyota Benz



When the individual objects are created, they inherit all the variables & functions from the class.

A class is a user-defined data-type that we can use in our program, & it works as an object constructor, or a "blueprint" for creating objects.

Create a class :-

To create a class, use the class keyword:

Eg:
```
class Student {
    int age;
    String name;        } Data members / Attributes
    void eat() {}       } Member function
};
```

Create an object :-

To create an object of 'Student' specify the class name, followed by the object name.

To access the class attributes (age, name) & member functions use the 'dot syntax' (.) on the object.

* Create an object called 'myobj' & access the attributes :

Eg:
```
class Student {          // The class
  public:                // Access specifier / modifier
    int age;             // Attribute (int variable)
    string name;         //     "        (string  "  )
};

int main() {
    Student myobj;  // create an object of Student class
    // Access attributes & set values
    myobj.age = 10;
    myobj.name = "Suraj";
    // Print attribute values.
    cout << myobj.age << endl;
    cout << myobj.name << endl;
    return 0;
}
```

output :- 10
Suraj

Syntax for creating object is :- class_name object_name;

Eg:
```
#include <iostream>
using namespace std;

class person {
    char name[20];
    int id;

  public:
    void getdetails() {

    }
};

int main() {
    person p1;  // p1 is a object
    return 0;
}
```

keyword user-defined class name
↑ ↑
class ClassName {

    Access modifier: // can be public, private or protecte

    Data member; //variable to be used

    Member functions() { // Method to access data memb

    }

}; class name ends with a semicolon

```
// create a car class with some attributes
class car {
    public:
        string brand;
        string model;
        int year;
};
int main() {
  // create an object of car
      Car carobj1;
      carobj1.brand = "BMW";
      carobj1.model = "X5";
      carobj1.year = 1999;
  // create another object of car
      Car carobj2;
      carobj2.brand = "Ford";
      carobj2.model = "Mustang";
      carobj2.year = 1969;
  // print attribute values
    cout <<carobj1.brand << " " << carobj1.model << " " << carobj1.year << endl;
    cout << carobj2.brand << " " << carobj2.model << " " << carobj2.year << endl;
     return 0;
}
```

output :- BMW X5 1999
Ford Mustang 1969

Eg:

```
class Name {
      public: Access modifier
        string name; Data member
        void printname() { member function
            cout << "Name is :" << name;
        }
};
int main () {
      Name obj; Declare an object
      obj.name = "Suraj"; Accessing data member
      //Accessing member function
      obj.printname();
      return 0;
}
```

Class Method / Function.

Methods are function that belongs to the class.
There're two ways to define functions that belongs to a class:

i) Inside class definition
ii) outside " "

Note:- You access method just like access attributes; by creating an object of the class & using the dot syntax (.):

Inside Example:

```
class myclass {                    The class
    public:                        Access Modifier
        void mymethod () {         function defined inside the class
            cout << "Hello";
        }
};
int main () {
    myclass myobj;                 create an object of myclass
    myobj.mymethod();              call the function
    return 0;
}
```

output: Hello

To define a function outside the class definition, you have to declare it inside the class and then define it outside the class. This is done by specifying the name of the class, followed the scope resolution ::

Outside Example:

```
class myclass {                    The class
    public:                        Access modifier
        void mymethod ();          function declaration
};

// method / function definition outside the class
void myclass :: mymethod () {
    cout << "Hello";
}

int main () {
    myclass myobj;                 create object of myclass
    myobj.mymethod();              call the method.
    return 0;
}
```

Output: Hello

Parameters :

We can also add parameters :

```
#include<iostream>
using namespace std;
Class Car {
   public:
     int speed (int maxSpeed);
};
int Car::speed (int maxspeed){
   return maxspeed;
}
int main (){
   Car myobj;      create an object of car
   cout << myobj.speed (200);  call the method with an argument.
   return 0;
}
```

output :- 200

Constructor

A constructor is a special type of member function that is called automatically when an object is created. constructor has the same name as that of the class & it doesn't have a return type. It is always public.

To create a constructor, use the same name as class, followed by parentheses ( ) :

```
Class Car {
   public: Access modifier
     Car () { constructor
       //
     }
};
```

Types of constructor :-

1) Default constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Eg:-

```
class Wall {
   public :
     double length;
     Wall () { default constructor
       length = 5.5;  Initialize variables
       cout<<" creating a Wall." << endl;
       cout<<" Length = " << length << endl;
     }
};
   int main (){
     Wall wallobj;  create an object of Wall (default constructor called
     return 0;      automatically when object is created)
   }
```

output: creating a Wall.
Length: 5.5

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

class Myclass {
    public
        int a, b;
      // default constructor
        Myclass () {
            a = 10;
            b = 20;
        }
};

int main() {
    // default constructor called automatically when the object is created
        Myclass myobj;

        cout << "a:" << myobj.a << endl << "b:" << myobj.b;
        return 0;
}
```

output: a: 10
b: 20

NOTE: Even if we don't define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. Parameterized Constructor

A constructor with parameter is known as a parameterized constructor. This is the preferred method to initialize member data.

To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Note:- When the parameterized constructor is defined & no default constructor is defined explicitly, the compiler will not implicitly call the default constructor

Eg:

```
class Car {                          The class
    public:                          Access modifier
        string brand;                Attribute
        string model;                   "
        int year;                       "

        Car (string x, string y, int z) {   constructor with parameters
            brand = x;
            model = y;
            year = z;
        }
};

int main () {
    // create car objects & call the constructor with different values
    Car carobj1 ("BMW", "X5", 1999);
    Car carobj2 ("Ford", "Mustang", 1969);
```

```
    cout << carobj1.brand << " " << carobj1.model << " " << carobj1.year << endl;
    cout << carobj2.brand << " " << carobj2.model << " " << carobj2.year << endl;
}   return 0;
```

Output:- BMW X5 1999
Ford Mustang 1969

Just like functions, constructors can also be defined outside the class. First declare the constructor inside the class, & then define it outside of the class by specifying the name of the class, followed by the scope resolution :: operator, followed by the name of the constructor.

Eg:
```
class Car {
    public :
    string brand;
    string model;
    int year;
    Car(string x, string y, int z);
};
// constructor definition outside the class
Car::Car(string x, string y, int z) {
    brand = x;
    model = y;
    year = z;
}

int main () {
    // create Car objects & call the constructor with different values
    Car mycar1 ("BMW", "X5", 1999);
    Car mycar2 ("Ford", "Mustang", 1969);

    // Print values
    cout << mycar1.brand << " " << mycar1.model << " " << mycar1.year << endl;
    cout << mycar2.brand << " " << mycar2.model << " " << mycar2.year << endl;
    return 0;
}
```

Output:- BMW X5 1999
Ford Mustang 1969

3) Copy constructors

Access Modifiers:

Access modifiers are used to implement an important aspect of object-oriented programming known as Data Hiding.

Access modifiers in a class decide the accessibility of the class members, like variables or functions in other class. That is, it will decide whether the class members or functions will get directly accessed by the block present outside the class or not, depending on the type of Access modifier

There are 3 types of access modifiers in c++

i) Public ii) Private iii) Protected

Syntax of Declaring Access modifiers in c++

We need to mention the types of specifier with a colon just before the class's variable or methods.

```
class ClassName {

    Private:
      // Declare private members/functions here.

    Public:
      // Declare public members/functions here.

    Protected:
      // Declare protected members/functions here.

};
```

...odifiers for the members inside the

Note:- If we don't specify any access modifiers for the members inside the class, then by default the access modifiers for the members will be Private.

Eg: There're 3 variables in an Employee class. One is employeeName, the second is employeeId & the third is employeeSalary.

```
Class Employee {
    private:
     int employeeSalary;

    public :
     int employeeId;
     String employeeName;

     String getEmployeeName() {
         return employeeName;
     }

    protected:
      void setEmployeeSalary (int n) {
          employeeSalary = n;
          return ;
      }
};
```

1) Public Access Modifier

All the class members declared under the public specifier will be available to everyone. The data members & member functions declared as public can be accessed by other classes & functions too. The public members of a class can be accessed from anywhere in the program using the direct access operator (.) with the object of that class.

2) Private Access Modifier

The private keyword is used to create private variables or private functions. The class members declared as 'private' can be accessed only by the member functions inside the class. They're not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

Note:- i) Protected & private data members or class methods can be accessed using a function only if that function is declared as the friend function.

ii) We can use the keyword friend to ensure the compiler understand & make the data accessible to that function.

3) Protected Access Modifier

The protected keyword is used to create protected variables or protected functions. The protected members can be ~~ass~~ accessed within and from the derived / child class

Note:- A class created or derived from another existing class (base class) is known as a derived class. The base class is also known as a superclass. It is created and derived through the process of inheritance.

It can't be accessed outside of its class except the derived class or super class of that class. However, it can be accessed by the friend function of that class, similar to a private ~~class~~ specifier.

Summary: Public, private & Protected

specifier	same class	derived class	outside class
Public ⟶	Yes	Yes	Yes
Private ⟶	Yes	NO	NO
Protected ⟶	Yes	Yes	NO

This Keyword :-

In c++ programming, this is a keyword that refers to the current instance of the class. 'this' pointer stores the address of the class instance, which is called from the member function that enables functions to access the correct object data members.

There can be 3 main usage of this keyword in c++.

1) It can be used to pass current object as a parameter to another function.
2) " " " refer " class instance variable.
3) It " " declare indexers

Syntax of this pointer

```
this.member_identifier;
```

In c++, this pointer is used when the data member & the local variables of the member function have the same name then the compiler will be in ambiguity until an unless we use 'this' pointer, because if we want to assign some values of a local variable to the data member then this can't be done without this pointer.

Eg.

```
class Demo {
  private:
    int i;
    float f;
    char c;
  public:
    void name(int i, float f, char c) {
      this->i = i;
      this->f = f;
      this->c = c;
    }
    void display() {
      cout<<" The integer value is = "<<i<<endl;
      cout<<" The float value is = "<<f<<endl;
      cout<<" The character value is = "<<c<<endl;
    }
};

int main() {
    Demo obj;
    int i = 20;
    float f = 2.056;
    char c = 'A';
```

(data member: i, f, c; member function: name; member function: display)

create an object

```
    cout << "The display of values which have both local variable & data membe
            have same name and using this pointer" << endl;
    obj.name (f, f, c);   } call member function
    obj.display ();

    return 0;
}
```

Eg:
```
class Employee {
    public:
        int id;
        string name;        } Data member (also instance variable)
        float salary;

        Employee (int id, string name, float salary) {  // constructor with
            this -> id = id;                                parameter
            this -> name = name;        } using this pointer
            this -> salary = salary;    } inside the constructor to set values in data
        }                                                       member

        void display () {   //member function
            cout << id << "  " << name << "  " << salary << endl;
        }
};

int main () {
    Employee e1 = Employee (100, "Suraj", 123456);  } creating an object of
    Employee e2 = Employee (105, "Srj", 567899);    } Employee

    e1.display ();
    e2.display ();

    return 0;
}
```

output :- 100 Suraj 123456
105 Srj 567899

Another way:

```
int main () {
    Employee e1 (100, "Suraj", 123456);
    Employee e2 (105, "Srj", 567899);

    e1.display ();
    e2.display ();

    return 0;
}
```

output :- 100 Suraj 123456
105 Srj 567899

Deleting this pointer

Generally, the delete operator shouldn't be used with the 'this' pointer to de-allocate it from memory.

Trying to delete 'this' pointer inside the member function is wrong & must be avoided, but if we try deleting the 'this' pointer following things can happen.

i) If the object is created on stack memory, then deleting the 'this' pointer from the objects member function can either result in the program crashing or undefined behaviour.

ii) If the object is created in heap memory, the deleting objects from the 'this' pointer will destroy the object from the program's memory. It will not crash the program, but later, if any object member function tries to access the 'this' pointer, the program will crash.

Eg.

```
class Demo{
    private:
      int member;
    public:
      Demo(int member){   constructor
        this->member = member;
      }
      void bar(){
        cout<<"Member:"<<this->member;
      }
      void displayText(){
        cout<<"Not accessing any member variable";
      }
      void destroy(){
        delete this;   Deleting 'this' pointer using keyword delete.
      }
};
int main(){
    Demo *ptr = new Demo(5);   creating object of Demo
    ptr->destroy();   Delete 'this' pointer
    ptr->displayText();   Accessing member function after 'this' is destroyed.
    return 0;
}
```

Output :- Not accessing any member variable

const Keyword:

constant is something that doesn't change. In c++ we use the keyword const to make program element constant, which means unchangeable & read-only.

Rules for the declaration & initialization of the constant variables :-

i) The const variable can't be left un-initialized at the time of the assignment.
ii) It can't be assigned value anywhere in the program.
iii) Explict value needed to be provided to the constant variable at the time of declaration of the constant variable.

① const int var; ✗
② const int var;
var = 5; ✗
③ const int var = 5; ✓

Constant keywords with different parameters:

i) Const variable :-

It is a const variable used to define the variable values that never be changed during the execution of a program.

Syntax:- const data-type variable-name;

Eg:
```
# include <iostream>
using namespace std;

int main () {
    const int num = 25;    Declare the value of const
    num = num + 10;
    return 0;
}
```
It shows the error because, we update the assigned value of num 25 by 10.

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

int main () {
    const x = 20;  } declare
    const y = 25;  } const variable
    int z = x+y;

    cout << " The sum of two number is :" << z << endl;
    return 0;
}
```

Output :- The sum of two number is : 45

ii) Constant Pointer :-

To create a const pointer, we need to use the const keyword before the pointer's name. We can't change the address of the const pointer after its initialization, which means the pointer will always point to the same address once the pointer is initialized as the const pointer.

Note:- We can also have a const pointer pointing to a const variable.

eg:
```
#include <iostream>
using namespace std;
int main (){
  int x = 10, y = 20;
  // use const keyword to make constant pointer
  int * const ptr = &x; const integer ptr variable point address to the variable x
  // ptr = &y; // Now ptr can't changed their address
    *ptr = 15; // ptr can only change the value.
    cout << "The value of x :" << x << endl;
    cout << " The value of ptr:" << *ptr << end1;
    return 0;
}
```

output:- The value of x: 15
" " " ptr: 15

// Pointer to constant variable.

It means the pointer points to the value of a const variable that can't change.

Declaration of the pointer to the constant variable:

const int* x; → Here, x is a pointer that point to a const integer type variable, & we can also declare a pointer to the const variable as,

char const* y; → In this case, y is a pointer to point a char type const variable.

eg:
```
#include <iostream>
using namespace std;
int main (){
  int x = 7; int y = 10;
  const int *ptr = &x; // Here, x become constant variable
  cout << "The initial value of ptr :" << *ptr << end1;
  cout << " The value of x :" << x << end1;
  // *ptr = 15; It is invalid; we can't directly assign a value to the ptr variable
  ptr = &y; Here, ptr variable pointing to the non const address 'y'
  Cout << " The value of y :" << y << end1;
  cout << " The value of ptr :" << *ptr << end1;
  return 0;
}
```

output:- The initial value of ptr: 7
The value of x : 7
The " " y : 10
" " " ptr : 10

{ lvalue → value having memory location.
rvalue → " doesn't have memory location. }

iv) constant function Arguments

The value of function arguments is declared const, it doesn't allow changing its value.

Syntax: return-type function_name (const int x) {

}

Eg: # include <iostream>
using namespace std;

int Test (const int num) { function contains an argument num
// If we change the value of the const argument, it throws an error.
// num += 10;
cout << "The value of num :" << num << endl;
}

int main () {
Test (5); function call
return 0;
}

Output:- The value of num : 5

v) const Member function of class

1) A const is a constant member function of a class that never changes any class data members, & it also doesn't call any non-const function.

2) It is also known as the read-only function.

3) We can create a constant member function of a class by adding the const keyword after the name of the member function.

Syntax: return-type member_fun() const {

}

In the above syntax, member_fun() is a member function of a class, & the const keyword is used after the ~~fn~~ name of the " " to make it constant.

Eg: class ABC {
public:
int A;

void fun() const { Declare member function as constant using const keyword
A = 5; // It shows compiler error, because, we're trying to change class Data member.
}
};

int main () {
ABC obj;
obj.fun();
return 0;
}

The above code throws a compilation error because fun() function is a constant member function of class ABC & we're trying to assign a value to its data member 'A' that returns an error.

ii) Constant Data member of class :-

Data member are like the variable that is declared inside a class, but once the data " is initialized, it never changes, not even in the constructor or destructor. The const data member is initialized using the const keyword before the data type inside the class.

The const data members can't be assigned the values during its declaration; however, they can assign the constructor values.

Eg:
```
#include <iostream>
using namespace std;
class ABC {
  public:
    const int A;   const keyword to declare constant data member
    ABC (int y) : A (y) {  create class constructor
      cout << " The value of y :" << y << endl;
    }
};
int main () {
  ABC obj (10);   object of class ABC
  cout << " The value of const data member 'A' is :" << obj.A << endl;
  // obj.A = 5;
  // cout << obj.A ;   } shows an error
  return 0;
}
```

output :- The value of y : 10 ;
The " " constant data member 'A' is : 10

iii) Constant objects :-

When we create an object using the const keyword, the value of data members can never change till the life of the object in a program. The const objects are also known as the read-only objects.

Syntax : const class-name object-name;

Eg:
```
#include <iostream>
using namespace std;
class Demo {
  public:
    int A;
    Demo () {  constructor of class Demo
      A = 20;  Define a value to 'A'
    }
};
int main () {
  const Demo obj;  Declare a constant object
  cout << The value of A :" << obj.A << endl;
```

```
    // obj.A = 30; // it returns a compile time error
    return 0;
}
```

output :- The value of A : 20

Static Data Member :-

When we define the data member of a class using the static keyword, the data members are called static data member. A static data is similar to the static member function because the static data can only be accessed during the static data member or static member function. And all, the objects of the class share the same copy of the static member to access the static data.

Static member Definition & Declaration

> Declaration :-

Static members are declared once inside the class as follow :-

```
static data-type variable_name;
```

Eg :- `static int count;`

> Definition (Initialization) :-

Static members must be defined (initialized) outside the class using scope resolution (::) as follows :-

```
data-type class_name :: variable-name = value;
```

Eg :- `int student :: count = 1;`

Syntax :-

```
class className {
        public:
        static data-type variable_name;
};
data type className :: variable_name = some_initial-value;
```

Basic Properties of static data member :-

> Static data members are initialized with zero automatically

```
class Test {
    public:
    static int count; // Static Data member declaration
};

int Test :: count; // not defining (initializing) but making accessible

int main (){
    Test t1;
    cout << " value of static member :" << t1.count << endl;

    return 0;
}
```

output :- value of static member : 0

constant data member of class:-

2) Static data members ~~initialization~~ can be called directly using the class name.

```
Class Test {
    Public:
      Static int count; // static data member declaration.
};

int Test :: count = 50; // Initializing & making accessible.

int main (){
    // object Not created. static members can be accessed without created object.
    // Just use className :: variable_name. If not declared static members data
    // value will be '0' by default.
    cout << "value of static member :" << Test :: count << endl;
    return 0;
}
```

Output :- Value of static member : 50.

Note :- Static data members are class members, not object member.

Static Member Functions :

A static member functions are special functions used to access the data members or other static member functions. A static member function shares the single copy of the member function to any no. of the class objects. If the static member function accesses any non-static data member or non-static member function, it throws an error.

i) A static member function can be called even if no object of the class exist.

ii) Static member function have a class scope & do not have access to 'this' pointer of the class.

iii) The static function can only access static members of the class.

Syntax :

```
class className{
        static return-type function_name (arguments) {
            // ---
            // - - -
        }
};
```

Eg:

```
Class Sample{
    public:
      static int a, b; // static data member
      int c; // Non-static data member

      // Static fun definition, will only work with static data members
      static void add (){
          a = 50;
          b = 40;     } static function can only access static data member
        cout << " a+b =" << a+b << endl;
      // following will give error:
      // c = 30; error: can't access non static data 'c' in static function
```

```
        // c=30; error: can't access non static data c in static function
    }
};

int Sample :: a;
int Sample :: b;

int main () {
    // calling static function without object
    Sample:: add();
    return 0;
}
```

output :- a+b = 90;

) Static member functions have no '*this' pointer.

Initialization (Initializer) List :-

Initializer list is used to initialize data members. The syntax begin with a colon (:) & then each variable ~~before~~ along with its value separated by a comma (,). The initializer list does not end in a semicolon (;).

Syntax :-

```
constructor_name (data_type value1, data_type value2) : data_member (value1),
                                                         data_member (value2) {
    // ---
    // ---
}
```

eg

```
class Base {
    private :
        int value;

    public :
        Base (int value) : value (value) {    // Default constructor & initializer list
            cout << " value is " << value ;
        }
};

int main () {
    Base obj (10);
    return 0;
}
```

output :- value is 10

Uses of initializer List :-

- When no Base class default constructor is present.
- When reference type is used.
  If you have a data members as reference type, you must initialize it in the Initialization List. Reference are immutable hence they can be initialized only once.
- For initializing const data member.
- When data member & parameter have same name (as above example).
- For improving performance.

Eg:
```
#include <iostream>
using namespace std;
Class Rectangle {
    Int length;
    int breadth;          } Data member
    Public:
        Rectangle (int l, int b) : length (l), breadth (b) { // constructor
        }
        Int printArea (){
            return length * breadth ;   } member function
        }
};
Int main (){
    Rectangle obj (7,5);  // create object of class.
    Cout << "Area of Rectange =" << obj. printArea << endl;
    return 0;
}
```
Output:- Area of Rectangle = 35

Macros in c++

A macro is a label defined in the source code that is replaced by its value by the preproccessor befor compilation. Macros are initialized with the #define preprocessor command & can be undefined with the #undef command.
The termination of the c++ macro does not need semicolon (;).

Eg:
```
# include <iostream>
using namespace std;
# define AREA (l,b) (l*b)  // Definition of macro.
int main (){
    int length = 20;
    int breadth = 2;
    area = AREA (length, breadth);  // Finding Area using macro
    Cout <<" Area of the rectangle =" << area ;
    return 0;
}
```
Output :- Area of the rectangle = 40

Types of Macro in c++

1) Chain Macros

Chain macros are defined as macros inside the macros. In the chain macro, the parent macro is defined first, & then its child macros.

Eg:
```
# include <iostream>
using namespace std;

// Macro definition
#define FACEBOOK FOLLOWERS
#define FOLLOWERS 150

int main () {
    cout << " This user have " << FACEBOOK << " K followers on facebook ! " ;
    return 0;
}
```

Output:- This user have 150K followers on facebook!

Eg:
```
# include <iostream>
using namespace std;

// Define a macro to square a number
#define SQUARE (x) ((x)*(x))

// Define a macro to double a number
#define DOUBLE (x) ((x)*2)

int main () {
    int num = 5;

    // chain the SQUARE & DOUBLE macros to square & then double the number
    int ans = DOUBLE (SQUARE (num));

    cout << " Result : " << ans;
    return 0;
}
```

output:- Result : 50

2) Object-Like Macros

In c++, an object-like macro is a macro that acts like a constant or a simple value. This macro is given the name object-like macro because the code segments looks like an object of code. This type of macro is generally used to replace a symbolic name with the variable being represented as a constant.

Eg:
```
# include <iostream>
using namespace std;

# define PI 3.1416

Int main () {
    float radius = 3;
    float area = PI * radius * radius;
    cout << " Area is : " << area ;
    return 0;
```

Eg:
```
#include <iostream>
using namespace std;
// Define an object-like macro for a max value
#define MAX_VALUE 100
int main () {
    int num[] = {15, 40, 60, 85, 105, 70};
    int size = sizeof(num) / sizeof(num[0]);
    for (int i = 0; i < size; i++) {
        if (num[i] > MAX_VALUE) {
            cout << "Number" << num[i] << " exceeds the max value" << endl;
        }
    }
    return 0;
}
```

output:- Number 105 exceeds the max value

3) function - like Macros

The function-like macro is almost similar to the function call in a program. In it the whole code is replaced by a function name. Since it is similar to function, there must be a pair of parentheses after the macro's name. We should never put a space betn the macro name & the parentheses in the macro definition.

Eg:
```
#include <iostream>
using namespace std;
// Definition of the macro
#define min(x,y)((x) < (y)) ? (x) : (y))
int main () {
    int x = 20;
    int y = 50;
    cout << "The minimum value betn " << x << "&" << y << "is :" << min(x,y);
    return 0;
}
```

output:- The minimum value betn 20 & 50 is : 20

Eg:
```
#include <iostream>
using namespace std;
#define PI 3.1416
#define AREA(r) r*r*PI
int main () {
    float radius = 5;
    float area = AREA(radius);
    cout << "Area is :" << area;
    return 0;
}
```

Output :- Area is 78.54

> Multi-Line Macros

A multi-line macro contains more than one line of definition in it. We create a multi-line macro by the use of backslash '\'.

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

// Definition of multi-line macro
# define ABA 1,\
             2,\
             3

int main () {
    // Array arr[] with elements defined in macro 'ABA'.

    int arr[] = { ABA };
     cout << " Elements of Array are : " << endl;

    for (int i = 0; i < 3; i++ ) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output :- Elements of Array are :
1 2 3

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

# define FACTORIAL(n) \   // define a multi-line macro to calculate the factorial of a number.
    ({ \
      int result = 1; \
      for (int i=0; i<=(n); i++ ) { \
          result = result * i; \
      } \
      result; \
    })

int main () {
    int num = 5;

    int fact = FACTORIAL (num);
    cout << " factorial of " << num << " is :" << fact << endl;

    return 0;
}
```

output :- factorial of 5 is : 120