

UNIT-2

Classes

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class.

A class declaration is similar syntactically to a structure. Here is the entire general form of a **class** declaration that does not inherit any other class.

```
class class-name {  
    private data and functions  
    access-specifier:  
    data and functions  
    access-specifier:  
    data and functions  
    // ...  
    access-specifier:  
    data and functions  
} object-list;
```

The *object-list* is optional. If present, it declares objects of the class. Here, *access-specifier* is one of these three C++ keywords:

```
public  
private  
protected
```

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The **public** access specifier allows functions or data to be accessible to other parts of your program. The **protected** access specifier is needed only when inheritance is involved. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

You may change access specifications as often as you like within a **class** declaration. For example, you may switch to **public** for some declarations and then switch back to **private** again. The class declaration in the following example illustrates this feature:

C++ Class

In C++, object is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

```
class Student  
{
```

```

    public:
    int id; //field or data member
    float salary; //field or data member
    String name;//field or data member
}

```

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1; //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```

#include <iostream>
using namespace std;
class Student {
    public:
        int id;//data member (also instance variable)
        string name;//data member(also instance variable)
};
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}

```

Output:

```

201
Sonoo Jaiswal

```

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
#include <iostream>
using namespace std;
class Student {
public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    void insert(int i, string n)
    {
        id = i;
        name = n;
    }
    void display()
    {
        cout<<id<<" "<<name<<endl;
    }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

Output:

```
201 Sonoo
202 Nakul
```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
#include <iostream>
using namespace std;
class Employee {
```

```

public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    float salary;
    void insert(int i, string n, float s)
    {
        id = i;
        name = n;
        salary = s;
    }
    void display()
    {
        cout<<id<<" "<<name<<" "<<salary<<endl;
    }
};

int main(void) {
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    e1.insert(201, "Sonoo",990000);
    e2.insert(202, "Nakul", 29000);
    e1.display();
    e2.display();
    return 0;
}

```

Output:

```

201  Sonoo  990000
202  Nakul  29000

```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```

#include <iostream>
using namespace std;
class Employee {
    public:

```

```

int id; //data member (also instance variable)
string name; //data member(also instance variable)
float salary;
Employee(int id, string name, float salary)
{
    this->id = id;
    this->name = name;
    this->salary = salary;
}
void display()
{
    cout<<id<<" "<<name<<" "<<salary<<endl;
}
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}

```

Output:

```

101  Sonoo  890000
102  Nakul  59000

```

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```

class class_name
{
    friend data_type function_name(argument/s);    // syntax of friend function.
};

```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or **scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
#include <iostream>
using namespace std;
class Box
{
    private:
        int length;
    public:
        Box(): length(0) { }
        friend int printLength(Box); //friend function
};
int printLength(Box b)
{
    b.length += 10;
    return b.length;
}
int main()
{
    Box b;
    cout<<"Length of box: "<< printLength(b)<<endl;
    return 0;
}
```

Output:

```
Length of box: 10
```

Let's see a simple example when the function is friendly to two classes.

```
#include <iostream>
using namespace std;
class B;      // forward declaration.
class A
{
    int x;
    public:
```

```

    void setdata(int i)
    {
        x=i;
    }
    friend void min(A,B);    // friend function.
};
class B
{
    int y;
    public:
    void setdata(int i)
    {
        y=i;
    }
    friend void min(A,B);    // friend function
};
void min(A a,B b)
{
    if(a.x<=b.y)
        std::cout << a.x << std::endl;
    else
        std::cout << b.y << std::endl;
}
int main()
{
    A a;
    B b;
    a.setdata(10);
    b.setdata(20);
    min(a,b);
    return 0;
}

```

Output:

```
10
```

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

Let's see a simple example of a friend class.

```

#include <iostream>

using namespace std;

class A
{
    int x =5;
    friend class B;    // friend class.
};

class B
{
public:
    void display(A &a)
    {
        cout<<"value of x is : "<<a.x;
    }
};

int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}

```

Output:

```
value of x is : 5
```

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

C++ static field example

Let's see the simple example of static field in C++.

```
#include <iostream>
using namespace std;
class Account {
public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
    }
    void display()
    {
        cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
    }
};
float Account::rateOfInterest=6.5;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Employee
    Account a2=Account(202, "Nakul"); //creating an object of Employee
    a1.display();
    a2.display();
    return 0;
}
```

Output:

```
201 Sanjay 6.5
202 Nakul 6.5
```

C++ static field example: Counting Objects

Let's see another example of static keyword in C++ which counts the objects.

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

```

class Account {
public:
    int accno; //data member (also instance variable)
    string name;
    static int count;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
        count++;
    }
    void display()
    {
        cout<<accno<<" "<<name<<endl;
    }
};

int Account::count=0;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Account
    Account a2=Account(202, "Nakul");
    Account a3=Account(203, "Ranjana");
    a1.display();
    a2.display();
    a3.display();
    cout<<"Total Objects are: "<<Account::count;
    return 0;
}

```

Output:

```

201 Sanjay
202 Nakul
203 Ranjana
Total Objects are: 3

```

C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

C++ static field example

Let's see the simple example of static field in C++.

```
#include <iostream>
using namespace std;
class Account {
public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
    }
    void display()
    {
        cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
    }
};
float Account::rateOfInterest=6.5;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Employee
    Account a2=Account(202, "Nakul"); //creating an object of Employee
    a1.display();
    a2.display();
    return 0;
}
```

Output:

```
201 Sanjay 6.5
202 Nakul 6.5
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

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

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
{
    public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
    public:
        int id;//data member (also instance variable)
        string name;//data member(also instance variable)
        float salary;
        Employee(int i, string n, float s)
        {
```

```

        id = i;
        name = n;
        salary = s;
    }
    void display()
    {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
    }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}

```

Output:

```

101  Sonoo  890000
102  Nakul  59000

```

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```

#include <iostream>
using namespace std;
class Employee
{
    public:
        Employee()
        {
            cout<<"Constructor Invoked"<<endl;
        }
        ~Employee()
        {
            cout<<"Destructor Invoked"<<endl;
        }
};
int main(void)
{

```

```

Employee e1; //creating an object of Employee
Employee e2; //creating an object of Employee
return 0;
}

```

Output:

```

Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked

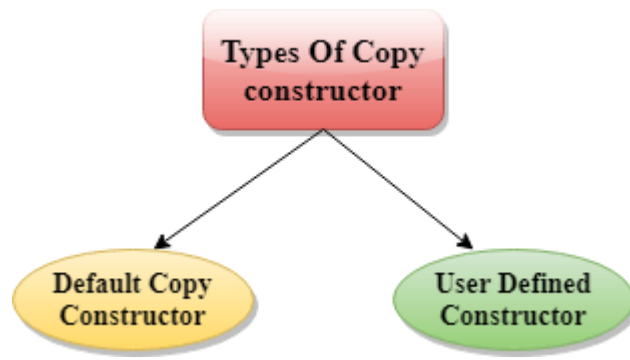
```

C++ Copy Constructor

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
- **User Defined constructor:** The programmer defines the user-defined constructor.



Syntax Of User-defined Copy Constructor:

1. Class_name(**const** class_name &old_object);


Consider the following situation:

```

class A
{
    A(A &x) // copy constructor.
    {
        // copyconstructor.
    }
}

```

In the above case, **copy constructor can be called in the following ways:**

- `A a2(a1);`
 - `A a2 = a1;`
- 
- a1 initialises the a2 object.**

Let's see a simple example of the copy constructor.

// program of the copy constructor.

```
#include <iostream>
using namespace std;
class A
{
    public:
    int x;
    A(int a)           // parameterized constructor.
    {
        x=a;
    }
    A(A &i)           // copy constructor
    {
        x = i.x;
    }
};
int main()
{
    A a1(20);         // Calling the parameterized constructor.
    A a2(a1);         // Calling the copy constructor.
    cout<<a2.x;
    return 0;
}
```

Output:

20

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

- When we initialize the object with another existing object of the same class type. For example, `Student s1 = s2`, where `Student` is the class.
- When the object of the same class type is passed by value as an argument.
- When the function returns the object of the same class type by value.

Two types of copies are produced by the constructor:

- Shallow copy
- Deep copy

Shallow Copy

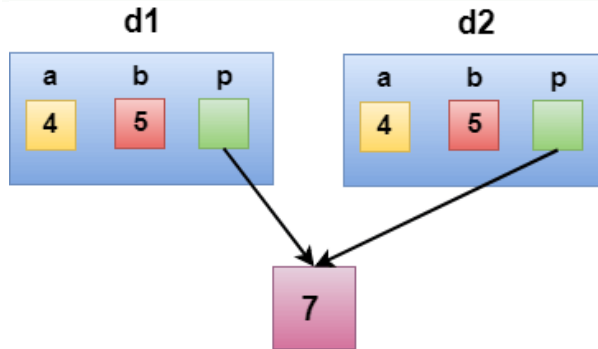
- The default copy constructor can only produce the shallow copy.
- A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
class Demo
{
    int a;
    int b;
    int *p;
public:
    Demo()
    {
        p=new int;
    }
    void setdata(int x,int y,int z)
    {
        a=x;
        b=y;
        *p=z;
    }
    void showdata()
    {
        std::cout << "value of a is : " <<a<< std::endl;
        std::cout << "value of b is : " <<b<< std::endl;
        std::cout << "value of *p is : " <<*p<< std::endl;
    }
};
int main()
{
    Demo d1;
    d1.setdata(4,5,7);
    Demo d2 = d1;
    d2.showdata();
    return 0;
}
```


Output:

```
value of a is : 4  
value of b is : 5  
value of *p is : 7
```



In the above case, a programmer has not defined any constructor, therefore, the statement **Demo d2 = d1;** calls the default constructor defined by the compiler. The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer *p* of both the objects point to the same memory location. Therefore, when the memory of a field is freed, the memory of another field is also automatically freed as both the fields point to the same memory location. This problem is solved by the **user-defined constructor** that creates the **Deep copy**.

Deep copy

Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations. In this way, both the source and copy are distinct and will not share the same memory location. Deep copy requires us to write the user-defined constructor.

Let's understand this through a simple example.

```
#include <iostream>  
using namespace std;  
class Demo  
{  
    public:  
    int a;  
    int b;  
    int *p;  
  
    Demo()  
    {  
        p=new int;  
    }  
    Demo(Demo &d)  
    {  
        a = d.a;  
        b = d.b;  
        p = new int;  
        *p = *(d.p);  
    }  
};
```

```

}
void setdata(int x,int y,int z)
{
    a=x;
    b=y;
    *p=z;
}
void showdata()
{
    std::cout << "value of a is : " <<a<< std::endl;
    std::cout << "value of b is : " <<b<< std::endl;
    std::cout << "value of *p is : " <<*p<< std::endl;
}
};
int main()
{
    Demo d1;
    d1.setdata(4,5,7);
    Demo d2 = d1;
    d2.showdata();
    return 0;
}

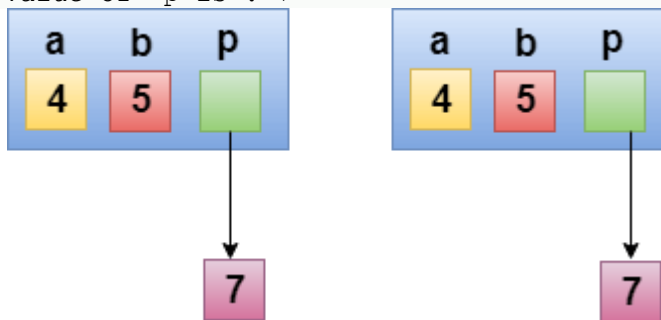
```

Output:

```

value of a is : 4
value of b is : 5
value of *p is : 7

```



In the above case, a programmer has defined its own constructor, therefore the statement **Demo d2 = d1;** calls the copy constructor defined by the user. It creates the exact copy of the value types data and the object pointed by the pointer p. Deep copy does not create the copy of a reference type variable.

Differences b/w Copy constructor and Assignment operator(=)

Copy Constructor

Assignment Operator

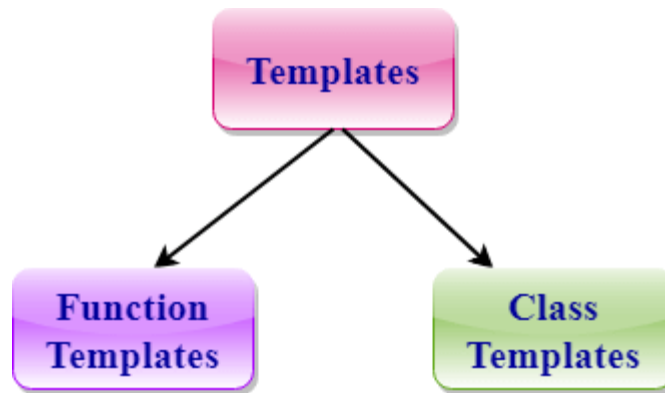
It is an overloaded constructor.	It is a bitwise operator.
It initializes the new object with the existing object.	It assigns the value of one object to another object.
Syntax of copy constructor: <pre>Class_name(const class_name &object_name) { // body of the constructor. }</pre>	Syntax of Assignment operator: <pre>Class_name a,b; b = a;</pre>
<ul style="list-style-type: none"> ○ The copy constructor is invoked when the new object is initialized with the existing object. ○ The object is passed as an argument to the function. ○ It returns the object. 	The assignment operator is invoked when we assign the existing object to a new object.
Both the existing object and new object shares the different memory locations.	Both the existing object and new object shares the same memory location.
If a programmer does not define the copy constructor, the compiler will automatically generate the implicit default copy constructor.	If we do not overload the "=" operator, the bitwise copy will occur.

C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

```
template < class Ttype> ret_type func_name(parameter_list)
{
    // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Let's see a simple example of a function template:

```
#include <iostream>
using namespace std;
template<class T> T add(T &a,T &b)
```

```

{
    T result = a+b;
    return result;
}

int main()
{
    int i =2;
    int j =3;
    float m = 2.3;
    float n = 1.2;
    cout<<"Addition of i and j is :"<<add(i,j);
    cout<<'\n';
    cout<<"Addition of m and n is :"<<add(m,n);
    return 0;
}

```

Output:

```

Addition of i and j is :5
Addition of m and n is :3.5

```

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```

template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
    // body of function.
}

```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Let's see a simple example:

```

#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}

```

```

        std::cout << "Value of b is : " <<b<< std::endl;
    }

    int main()
    {
        fun(15,12.3);

        return 0;
    }

```

Output:

```

Value of a is : 15
Value of b is : 12.3

```

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```

#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
    std::cout << "Value of b is : " <<b<< std::endl;
    std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
    fun(10);
    fun(20,30.5);
    return 0;
}

```

Output:

```

Value of a is : 10
Value of b is : 20
Value of c is : 30.5

```

In the above example, template of fun() function is overloaded.

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
void fun(double a)
{
    cout<<"value of a is : "<<a<<"\n";
}

void fun(int b)
{
    if(b%2==0)
    {
        cout<<"Number is even";
    }
    else
    {
        cout<<"Number is odd";
    }
}

int main()
{
    fun(4.6);
    fun(6);
    return 0;
}
```

Output:

```
value of a is : 4.6
Number is even
```

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
    .
    .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

1. class_name<type> ob;

where class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

Let's see a simple example:

```
#include <iostream>
using namespace std;
template<class T>
class A
{
    public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
    }

};

int main()
{
    A<int> d;
    d.add();
    return 0;
}
```

Output:


```
Addition of num1 and num2 : 11
```

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, .....>
class class_name
{
    // Body of the class.
}
```

Let's see a simple example when class template contains two generic data types.

```
#include <iostream>
using namespace std;
template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
    public:
    A(T1 x,T2 y)
    {
        a = x;
        b = y;
    }
    void display()
    {
        std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
    }
};

int main()
{
    A<int,float> d(5,6.5);
    d.display();
    return 0;
}
```

Output:

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments. In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. **Let's see the following example:**

```
template<class T, int size>
class array
{
    T arr[size];    // automatic array initialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;           // array of 15 integers.
array<float, 10> t2;        // array of 10 floats.
array<char, 4> t3;          // array of 4 chars.
```

Let's see a simple example of nontype template arguments.

```
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
    public:
    T arr[size];
    void insert()
    {
        int i =1;
        for (int j=0;j<size;j++)
        {
            arr[j] = i;
            i++;
        }
    }

    void display()
    {
        for(int i=0;i<size;i++)
        {
            std::cout << arr[i] << " ";
        }
    }
};
```

```

    }
};
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}

```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.

Abstract Classes

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by

1. Abstract class

Abstract class and interface both can have abstract methods which are necessary for abstraction.

C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as `<>strong>pure virtual` function. A pure virtual function is specified by placing `"= 0"` in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method `draw()`. Its implementation is provided by derived classes: `Rectangle` and `Circle`. Both classes have different implementation.

```

#include <iostream>
using namespace std;
class Shape
{
    public:
    virtual void draw()=0;
};
class Rectangle : Shape
{
    public:
    void draw()
    {
        cout << "drawing rectangle..." << endl;
    }
};
class Circle : Shape
{
    public:
    void draw()
    {
        cout << "drawing circle..." << endl;
    }
};
int main( ) {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
    return 0;
}

```

Output:

```

drawing rectangle...
drawing circle...

```

Data Abstraction in C++

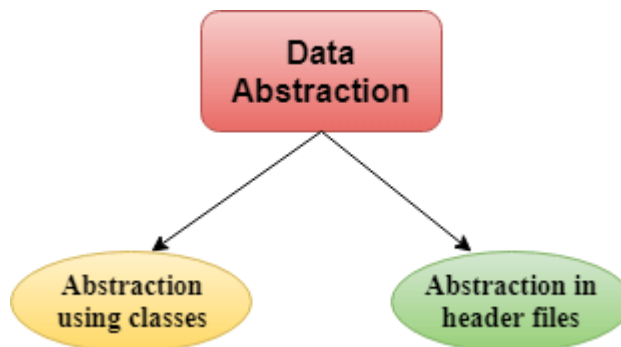
- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.

- C++ provides a great level of abstraction. For example, `pow()` function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.



Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: Another type of abstraction is header file. For example, `pow()` function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```

#include <iostream>
#include<math.h>
using namespace std;
int main()
{
    int n = 4;
    int power = 3;
    int result = pow(n,power);    // pow(n,power) is the power function
    std::cout << "Cube of n is : " << result << std::endl;
  
```

```
    return 0;
}
```

Output:

```
Cube of n is : 64
```

In the above example, `pow()` function is used to calculate 4 raised to the power 3. The `pow()` function is present in the `math.h` header file in which all the implementation details of the `pow()` function is hidden.

Let's see a simple example of data abstraction using classes.

```
#include <iostream>
using namespace std;
class Sum
{
private: int x, y, z; // private variables
public:
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()
{
Sum sm;
sm.add();
return 0;
}
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members `x`, `y` and `z` are only accessible by the member functions of the class.

Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.

- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.