Features Of OOP

Introduction

- Operator overloading
  - Enabling C++’s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care; when overloading is misused, program difficult to understand
  - Examples of already overloaded operators
    - Operator **<<** is both the stream-insertion operator and the bitwise left-shift operator
    - **+** and **–**, perform arithmetic on multiple types
  - Compiler generates the appropriate code based on the manner in which the operator is used

Overloading Unary Operators

- Overloading unary operators
  - Can be overloaded with no arguments or one argument
  - Should usually be implemented as member functions
    - Avoid **`friend`** functions and classes because they violate the encapsulation of a class
  - Example declaration as a member function:

```
class String {
public:
   bool operator!() const;
   ...
};
```

Example

```
#include<iostream>
using namespace std;
class example{
int a,b,c,d;
public:
void input(){
cout<<"enter values";
cin>>a>>b>>c>>d;
}
void operator ++(){
++a;
++b;
++c;
++d;
}
void display(){
cout<<a<<endl;
cout<<b<<endl;
cout<<c<<endl;
cout<<d<<endl;
}
};
int main(){
example e;
e.input();
e.display();
++e;
e.display();
return 0;
}
```

Overloading Binary Operators

– Non-member function, two arguments

– Example:

```
class String {
    friend const String &operator+=(
                String &, const String & );
    ...
 };
```

– **`y += z`** is equivalent to **`operator+=( y, z )`**

Example

```
#include<iostream>
using namespace std;
class complex{
int real,img;
public:
void input(){
cout<<"enter values";
cin>>real;
cin>>img;
}
complex operator + (complex b ){
complex m;
m.real = real + b.real;
m.img = img + b.img;
return m;
}
void display(){
cout<<real <<"+"<<img<<"i";
}
};
int main(){
complex x,y,z;
x.input();
y.input();
z = x+y;
z.display();
return 0;
}
```

Overloading ++ and --

- Pre/post incrementing/decrementing operators
  - Allowed to be overloaded
  - Distinguishing between pre and post operators
    - prefix versions are overloaded the same as other prefix unary operators

```
d1.operator++();        // for ++d1
```

    - convention adopted that when compiler sees postincrementing expression, it will generate the member-function call

```
d1.operator++( 0 );     // for d1++
```

    - **`0`** is a dummy value to make the argument list of **`operator++`** distinguishable from the argument list for **`++operator`**

Wap to add two complex number using binary operator overloading with friend function.

```
#include<iostream>
using namespace std;
class complex{
int real,img;
public:
void input(){
cout<<"enter values";
cin>>real;
cin>>img;
}
friend complex operator +(complex a,complex b){
complex t;
t.real = a.real + b.real;
t.img = a.img + b.img;
return t;
}
void display(){
cout<<real <<"+"<<img<<"i";
}
};
int main(){
complex x,y,z;
x.input();
y.input();
z = x+y;
z.display();
}
```

Data Conversion

- We can use variables of different data types in the same exprssions eg.float with integer of an integer with float.In these cases,c++ automatically converts one data type to another as the situation demands.
- The data type of the right-hand side operand or expression is converted to the data type of the left hand side.
- Thus conversion from one data type to another is known as data type conversion.

1) Implicit type conversion

The compiler performs the conversion of data types when an expression consists of data type of different types. This is called an implicit type conversion.

2). Explicit type conversion

- This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.

  Syntax

  (type)expression

Implicit type conversion

Example

```
#include<iostream>
using namespace std;
int main(){
int x =10;
float z;
z = x + 1.0;
cout<<z;
return 0;
}
```

Explicit type conversion

Example

```
#include<iostream>
using namespace std;
int main()
{
float x =1.2;
int sum = (int)x + 1;
cout<<sum;
}
```

Inheritance in C++

- The mechanism of deriving a class from another class is known as inheritance.
- The main advantage of inheritance is, it provides an opportunity to reuse the code functionality and fast implementation time.
- The member of the class can be public, private and protected.

- **Syntax**

class derivedclass: Accessspecifier BaseClass

The default access specifier is Private.

Inheritance help user to create a new class from a existing class.

Derive class inherits all the features from a base class including additional features of its own.

Types of inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

Single Inheritance

- In single inheritance, one class is derived from another class.
- It represent a form of inheritance where there is only one base and derived class

```
#include<iostream>
using namespace std;
class A
{
int x;
protected:
int y;
public:
int z;
void getxyz(){
cout<<"enter value of x,y,z"<<endl;
cin>>x>>y>>z;
}
void showxyz()
{
cout<<"x="<<x<<"y="<<y<<"z="<<z<<endl;
}
};
class b:public A
{
private:
int k,sum;
public:
void get_k()
{
cout<<"enter value of k"<<endl;
cin>>k;
}
void show_k(){
cout<<"k="<<k<<endl;
}
void addition(){
sum =y+z+k;
}
void display(){
cout<<"sum =" <<sum<<endl;
}
};
int main(){
b b1;
b1.getxyz();
b1.get_k();
b1.show_k();
b1.addition();
b1.display();
return 0;
}
```

Multiple Inheritance

It is a type of inheritance in which new class is created from more than one base class. In this type of inheritance a single derived class uses behaviour of two base classes.

Syntax

```
Class A{
………………..
}
Class B
{
…………..
};
Class C:visibility _mode A,visibility_mode B{
……………..
}
```

Ambiguity in multiple inheritance

```
#include<iostream>
using namespace std;
class m
{
public:
void display()
{
cout<<" class m"<<endl;
}
};
class n{
public:
void display(){
cout<<"class n"<<endl;
}
};
class p:public m,public n
{public:
void display(){
m::display();
n::display();
}
};
int main(){
p p;
p.display();
return 0;
}
```

Multilevel Inheritance

- In multilevel inheritance a class is derived from another derived class.Thus base class of a derived class is also derived of another base class.An example

```
class A { ... .. ... };
class B: public A
{ ... .. ... };
class C: public B
{ … … … };
```

Multipath/ Hierarchical Inheritance

- It is the process of deriving two or more classes form single base class.And in turn each of the derived classes can further be inherited in same way.Thus it forms hierarchy of classes of a tree of classes which is rooted at single class.

GENERAL FORM

```
Class A{
};
Class B:public A{
};
Class C:publc A{
};
Class D:Public A{
};
```

Hybrid Inheritance

- The use of more than one type of inheritance is called hybrid inheritance .It is used in such situations where we need to apply two or more types of inheritance to design a program.

Need for Inheritance

Inheritance is needed in OOP because of the following reasons -

- To enable reusability, extendability, and modifiability easier in code.
- Makes an object easy to port and maintain.
- To make exceptional properties economical and practical.

```
#include <iostream>
using namespace std;
class Animals // indicates class A
{
public:
Animals()
   {
cout<< "This is an animal\n";
   }
};
class Mammals: public Animals // indicates class
B derived from class A
{
public:
Mammals()
   {
cout<< "This is a mammal\n";
   }
};
class Herbivores  // indicates class C
{
public:
Herbivores()
   {
cout<< "This is a herbivore\n";
   }
};
class Cow: public Mammals, public Herbivores
 // indicates class D derived from class B and class
C
{
public:
Cow()
   {
cout<< "A cow is a herbivore mammal\n";
   }
};
int main() {
   Cow c;
   return 0;
}
```

Constructor in C++

Constructor in C++ is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. It constructs the values i.e. provides data for the object which is why it is known as constructor.

• Constructor is a member function of a class, whose name is same as the class name.
• Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically, when an object of the same class is created.
• Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.
• Constructor do not return value, hence they do not have a return type.

```
#include<iostream>
using namespace std;
class a
{
public:
a (int nvalue){
cout<<"a:"<<nvalue<<endl;
}
};
class b: public a
{
public:
b(int nvalue,float dvalue):a(nvalue)
{
cout<<"b:"<<dvalue<<endl;
}};
class c: public b{
public:
c(int nvalue,float dvalue,char chvalue):b(nvalue,dvalue)
{
cout<<"c:"<<chvalue<<endl;
}};
int main(){
c cclass(5,4.3,'r');
return 0;
}
```

Constructor and destructor in destructor in inheritance

- Compiler automatically calls constructor of base class and derived class automatically when derived class object is created.

- If we declare derived class object in inheritance constructor of base class is executed first and then constructor of derived class.

- If derived class object goes out of scope or deleted by programmer the derived clas destructor is executed first and then base class destructor.

Constructor and Destructor

```
#include<iostream>
using namespace std;
class a
{
public:
a(){
cout<<"constructor in base class"<<endl;
}
~a()
{
cout<<"destructor in base class"<<endl;}
};
class b:public a
{
public:b(){
cout<<"constructor in derived class"<<endl;
}
~b(){
cout<<"destructor in derived class"<<endl;
}};
int main(){
b obj;
return 0;
}
```

Output

```
Constructor in base class
Constructor in derived class
Destructor in derived class
Destructor in base class
```

- **Constructors in multilevel inheritance**

```
class A
{
public:
    A ( )
    {
cout<<"Class A constructor is
called"<<endl;
    }
};
class B : public A
{
public:
    B ( )
    {
 cout<<"Class B constructor is
called"<<endl;
    }
};
class C : public B
{
public:
    C ( )
    {
cout<<"Class C constructor is
called"<<endl;
    }
};
void main ( )
{
    C obj;
}
```

Thank You !