girmiti software™

Introduction to C++

- Abstract Class
- Virtual Destructor

Abstract class

Abstract class:

Abstract Class is a class which contains atleast one Pure Virtual function in it. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract class:

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Abstract Class

Example using Abstract class:

```
#include<iostream>
using namespace std;
class Base                      //Abstract base class
{ public:
virtual void show() = 0;    //Pure Virtual Function
};
class Derived:public Base
{ public:
void show()
{
        cout << "Implementation of Virtual Function in Derived class";
}
};
int main()
{       Base *b;
        Derived d;
        b = &d;
        b->show();
}
```

Output:

```
Implementation of Virtual Function in Derived class
```

Abstract class

Example using abstract class:

```
#include <iostream>
using namespace std;
class Shape
{
protected:
int width;
int height;
public:
virtual int getArea() = 0;
void setWidth(int w)
{
width = w;
}
void setHeight(int h)
{
 height = h;
}
};
```

Abstract class

```
class Rectangle: public Shape
{ public:
int getArea()
{
return (width * height);
}
};
class Triangle: public Shape
{ public:
int getArea()
{
return (width * height)/2;
}
};
 int main(void)
{
Shape s;      //compile time error
Shape *s;
Rectangle Rect;
Triangle Tri;
```

Abstract class

```
s=&Rect;
s->setWidth(5);
s->setHeight(7);
cout << "Total Rectangle area: " << s->getArea() << endl;
s-> setWidth(5);
s-> setHeight(7);
cout << "Total Triangle area: " << s-> getArea() << endl;
}
```

Output:

Total Rectangle area: 35

Total Triangle area: 17

Pure virtual function

Pure virtual functions implementation:
Pure Virtual functions can be given a small definition in the Abstract class, which we want all the derived classes to have. Still we cannot create object of Abstract class. Also, the Pure Virtual function must be defined outside the class definition. If we will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

Example:

```
class Base         //Abstract base class
{
 public:
virtual void show() = 0;    //Pure Virtual Function
};
void Base :: show()         //Pure Virtual definition
{
cout << "Pure Virtual definition\n";
}
class Derived:public Base
{
 public:
void show()
{
Base::show();
cout << "Implementation of Virtual Function in Derived class";
}
};
```

Pure virtual function

```
int main()
{
Base *b;
Derived d;
b = &d;
b->show();
}
```

Output:
Pure Virtual definition
Implementation of Virtual Function in Derived class

Virtual destructor

Virtual Destructors:
Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destruction of the object when the program exits.
NOTE : Constructors are never Virtual, only Destructors can be Virtual.

Upcasting without Virtual Destructor:

Example:

```
class Base
{ public:
~Base()
{
cout << "Base Destructor\t"; }
};
class Derived:public Base
{
public:
~Derived()
{
 cout<< "Derived Destructor"; }
};
```

Virtual destructor

```
int main()
{
Base* b = new Derived;     //Upcasting
delete b;
}
```

Output:

Base destructor

In the above example, delete b will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called which results in memory leak.

Virtual Destructor

Upcasting with virtual Destructor:

Example:

```
#include<iostream>
using namespace std;
class Base
{
public:
        virtual ~Base()
        {
                cout << "Base Destructor\t";
        }
};
class Derived:public Base
{
public:
        ~Derived()
        {
                cout<< "Derived Destructor"<<endl;
        }
};
```

Virtual Destructor

```
int main()
{
        Base* b = new Derived;      //Upcasting
        delete b;
}
```

Output:

Derived Destructor

Base Destructor

When we have virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behavior.

Discussions