25+26+38

Bùi Lê Phi Long	16619

Tuesday, March 23, 2021   3:09 PM

Chapter 25 : Classes - Inheritance and Polymorphism

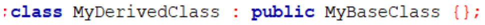
1. Inheritance

What is it?

- We can build a class from an existing class. It is said that a class could be *derived* from an existing class, known as inheritance

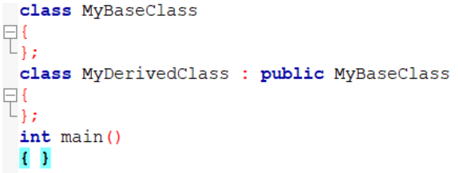
Syntax:

- To derive a class from an existing class:



```
class MyDerivedClass : public MyBaseClass {};
```

Example:

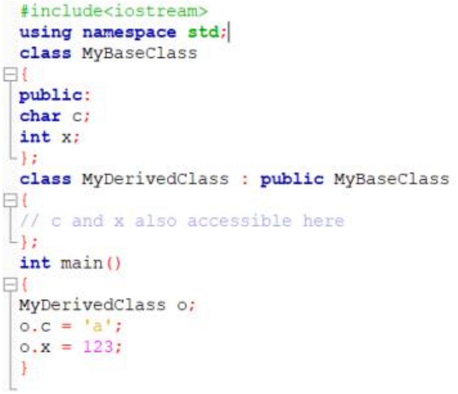


```
class MyBaseClass
{
};
class MyDerivedClass : public MyBaseClass
{
};
int main()
{ }
```

In this example, **MydDerivedClass** inherits the **MyBaseClass**
- Or it is said that **MyDerivedCLass** is derived from MyBaseClass, or MyBaseClass is a base class for MyDerivedClass

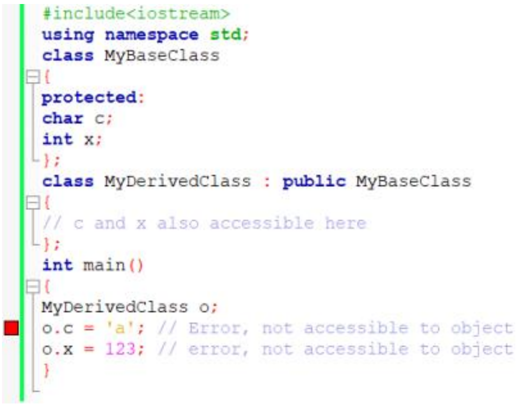
- These two classes have some sort of *relationship*. This relationship can be expressed through different naming conventions, but the most important one is INHERITANCE.
- Derived class and objects of a derived class can access public members of a base class

Example:

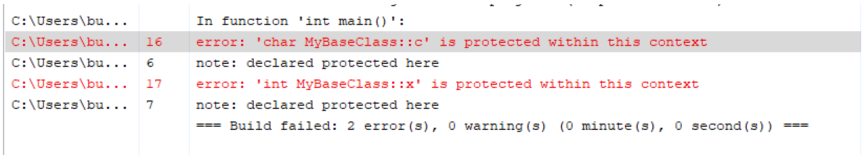


```
#include<iostream>
using namespace std;
class MyBaseClass
{
public:
char c;
int x;
};
class MyDerivedClass : public MyBaseClass
{
// c and x also accessible here
};
int main()
{
MyDerivedClass o;
o.c = 'a';
o.x = 123;
}
```

- New access specifier called **'protected'**. The derivied class itself can access protected members of a base class
- The protected access specifier allows access to the base class and derived class, but not to objects

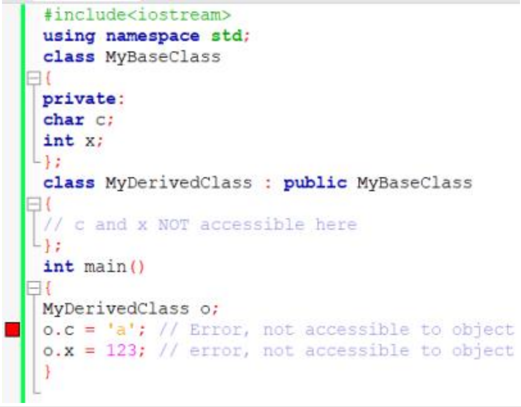


```
#include<iostream>
using namespace std;
class MyBaseClass
{
protected:
char c;
int x;
};
class MyDerivedClass : public MyBaseClass
{
// c and x also accessible here
};
int main()
{
MyDerivedClass o;
o.c = 'a'; // Error, not accessible to object
o.x = 123; // error, not accessible to object
}
```

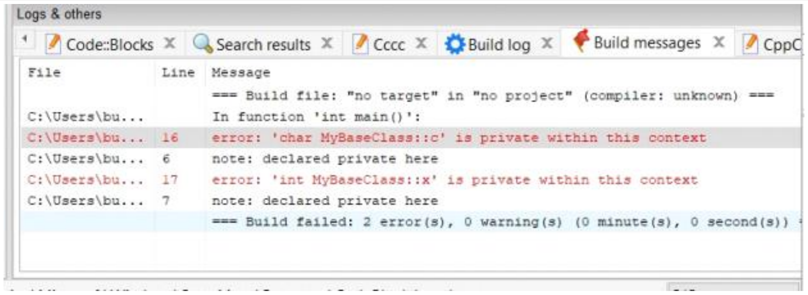


```
C:\Users\bu...        In function 'int main()':
C:\Users\bu...  16    error: 'char MyBaseClass::c' is protected within this context
C:\Users\bu...  6     note: declared protected here
C:\Users\bu...  17    error: 'int MyBaseClass::x' is protected within this context
C:\Users\bu...  7     note: declared protected here
                      === Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

- Seems like the data from protected access specifier could be accessed by a derived class, and the base class but not out side both of them, for instance in the main function

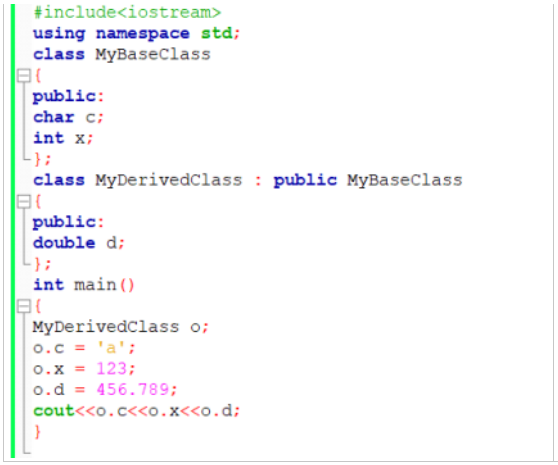


```
#include<iostream>
using namespace std;
class MyBaseClass
{
private:
char c;
int x;
};
class MyDerivedClass : public MyBaseClass
{
// c and x NOT accessible here
};
int main()
{
MyDerivedClass o;
o.c = 'a'; // Error, not accessible to object
o.x = 123; // error, not accessible to object
}
```



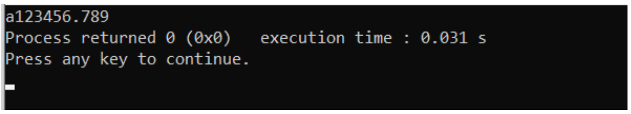
```
File            Line  Message
                      === Build file: "no target" in "no project" (compiler: unknown) ===
C:\Users\bu...        In function 'int main()':
C:\Users\bu...  16    error: 'char MyBaseClass::c' is private within this context
C:\Users\bu...  6     note: declared private here
C:\Users\bu...  17    error: 'int MyBaseClass::x' is private within this context
C:\Users\bu...  7     note: declared private here
                      === Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s))
```

- Data from private can not be accessed everywhere else except the class itself



```
#include<iostream>
using namespace std;
class MyBaseClass
{
public:
char c;
int x;
};
class MyDerivedClass : public MyBaseClass
{
public:
double d;
};
int main()
{
MyDerivedClass o;
o.c = 'a';
o.x = 123;
o.d = 456.789;
cout<<o.c<<o.x<<o.d;
}
```

Output:

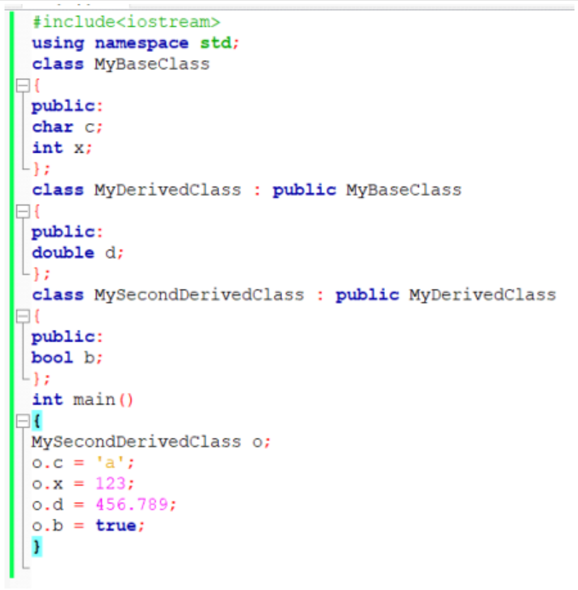


```
a123456.789
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Comments:

Here we inherited everything from the MyBaseClass class and introduced a new member field in MyDerivedClass called d. So, with MyDerivedClass, we are extending the capability of MyBaseClass. The field d only exists in MyDerivedClass and is accessible to derived class and its objects. It is not accessible to MyBaseClass class as it does not exist there.

- There are other ways of inheriting a class such as through protected and private inheritance, but the public inheritance suchas class *MyDerivedClass: public MybaseClass* is the most widely used:
    - **Example:**



```
#include<iostream>
using namespace std;
class MyBaseClass
{
public:
char c;
int x;
};
class MyDerivedClass : public MyBaseClass
{
public:
double d;
};
class MySecondDerivedClass : public MyDerivedClass
{
public:
bool b;
};
int main()
{
MySecondDerivedClass o;
o.c = 'a';
o.x = 123;
o.d = 456.789;
o.b = true;
}
```

Now our class has everything MyDerivedClass has, which includes everything MyBaseClass has, plus an additional bool field. It is said the inheritance produces a particular *hierarchy* of classes.

- This approach is widely used when we want to extend the functionality of out classes
- The derivde class is compatible with a base class

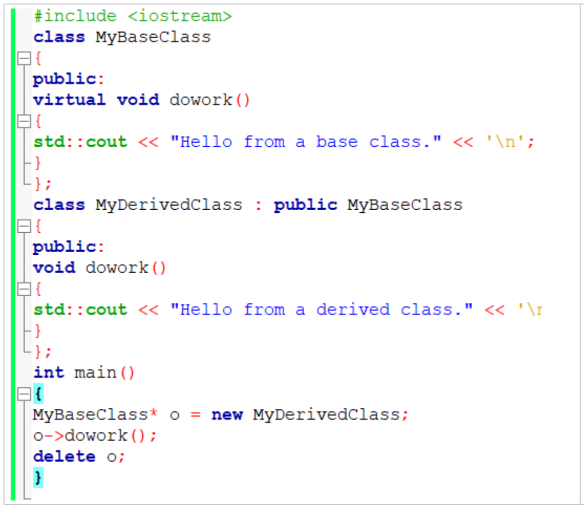
2. Polymorphism

WHAT IS IT?:

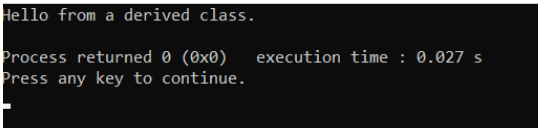
- *Polymorphism means the object can morph into different types*
- *Polymorphism* in C++ is achieved through an interface known as virtual functions
- It is said that the derived class is a base class. Its type is compatible with the base class type
- A pointer to a derived class is compatible with a pointer to the base class.

★ A pointer to derived class is compatible with a pointer to a base class, togertherwith inheritance, this is used to achieve the functionality known as polymorphism

Example:

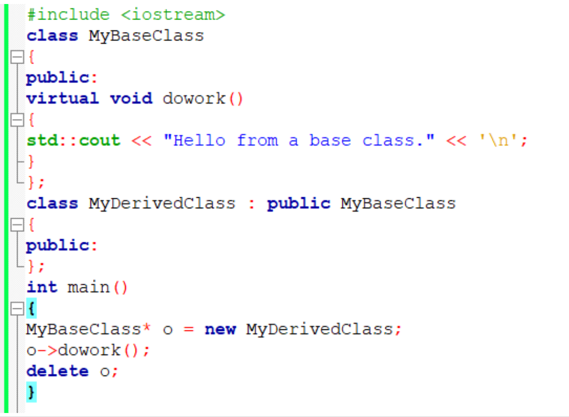


```
#include <iostream>
class MyBaseClass
{
public:
virtual void dowork()
{
std::cout << "Hello from a base class." << '\n';
}
};
class MyDerivedClass : public MyBaseClass
{
public:
void dowork()
{
std::cout << "Hello from a derived class." << '\r
}
};
int main()
{
MyBaseClass* o = new MyDerivedClass;
o->dowork();
delete o;
}
```

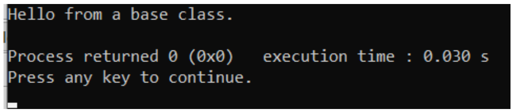


```
Hello from a derived class.

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

- In this example, we have a simple inheritnace where MyDerivedClass is derived from MyBaseClass
- The BaseClasshas a function called dowork() with a virtual specifier. Virtual means this function can be overriden in subsequent derived classes, and the appropriate version will be invoked through a polymorphic object
- The derived class has a function with the same name and same type of arguments in the derived class
- In our main program, we create an instance of a MyDerivedClass class **through** a base class pointer. Using the arrow operator -> we invoke the appropriate version of the function. Here the o object *morphs* into different types to invoke the appropriate function. Here it invokes the derived version. That is why the concept is called *polymorphism*

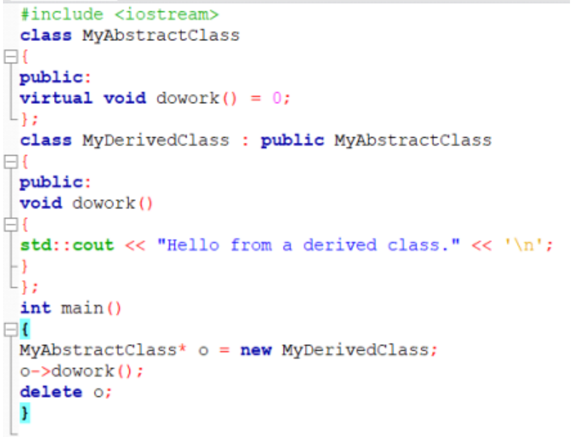


```
#include <iostream>
class MyBaseClass
{
public:
virtual void dowork()
{
std::cout << "Hello from a base class." << '\n';
}
};
class MyDerivedClass : public MyBaseClass
{
public:
};
int main()
{
MyBaseClass* o = new MyDerivedClass;
o->dowork();
delete o;
}
```

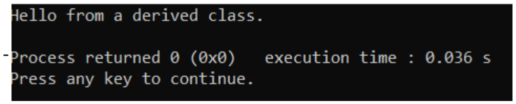


```
Hello from a base class.

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

If there were no dowork() function in the derived class, it would invoke the base class version



```
#include <iostream>
class MyAbstractClass
{
public:
virtual void dowork() = 0;
};
class MyDerivedClass : public MyAbstractClass
{
public:
void dowork()
{
std::cout << "Hello from a derived class." << '\n';
}
};
int main()
{
MyAbstractClass* o = new MyDerivedClass;
o->dowork();
delete o;
}
```



```
Hello from a derived class.

Process returned 0 (0x0)   execution time : 0.036 s
Press any key to continue.
```

- Functions can be pure virtual by specifying the = 0 at the end of the function declaration .Pure virtual functions do noot have definitions and are also called interfaces.
- Pure virtual functions must be re-defined in the derived class. Classes having at least one pure virtual funciton are called *abstract classes* and cannot be instantiated. They can only be used as base class

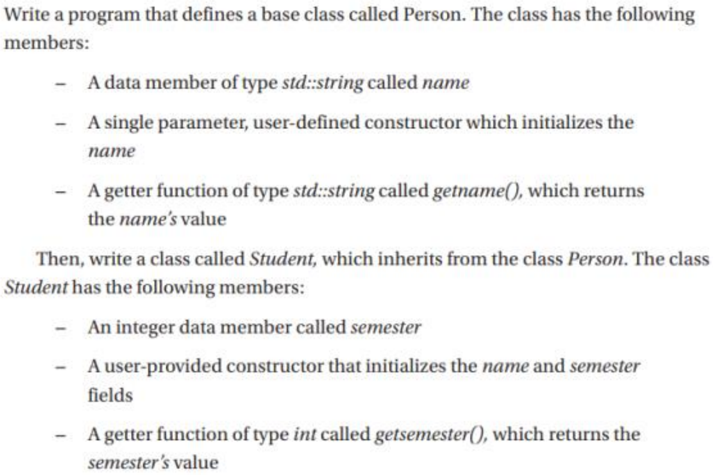
★ One more important things to add is that a base class must have virtual destructor if it is to be used in a polymorphic scenario. This ensures the proper deallocation of objects accessed through a base class poiner via the inheritance chain:

```
class MyBaseClass
{
public:
virtual void dowork() = 0;
virtual ~MyBaseClass() {};
};
```

So, three pillars of object-oriented programming are:
– Encapsulation: grouping the fileds into different visisbility zones, hiding implementation from the user
– Inheritance: creating classes by inheriting from a base class. Creating a certain class hierarchy and relationship types during runtime
– Polymorphism: an ability of an object to morph into different types during runtime, ensuring the proper function is invoked. Achieved through inheritance, virtual and overridden functions, and base and derived class pointers

26. Exercise

1. Inheritance

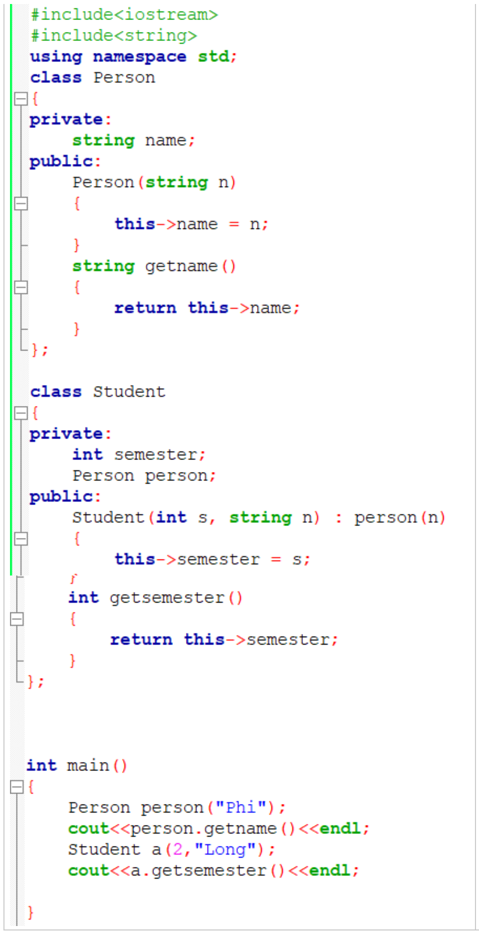


Write a program that defines a base class called Person. The class has the following members:

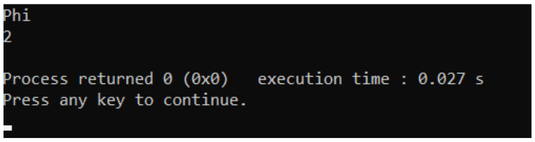
- A data member of type *std::string* called *name*
- A single parameter, user-defined constructor which initializes the *name*
- A getter function of type *std::string* called *getname()*, which returns the *name's* value

Then, write a class called *Student*, which inherits from the class *Person*. The class *Student* has the following members:

- An integer data member called *semester*
- A user-provided constructor that initializes the *name* and *semester* fields
- A getter function of type *int* called *getsemester()*, which returns the *semester's* value



```
#include<iostream>
#include<string>
using namespace std;
class Person
{
private:
    string name;
public:
    Person(string n)
    {
        this->name = n;
    }
    string getname()
    {
        return this->name;
    }
};

class Student
{
private:
    int semester;
    Person person;
public:
    Student(int s, string n) : person(n)
    {
        this->semester = s;
    }
    int getsemester()
    {
        return this->semester;
    }
};


int main()
{
    Person person("Phi");
    cout<<person.getname()<<endl;
    Student a(2,"Long");
    cout<<a.getsemester()<<endl;

}
```



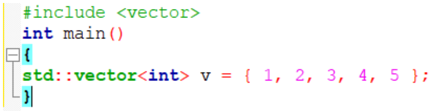
```
Phi
2

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

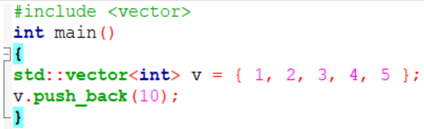
38.1.VECTOR

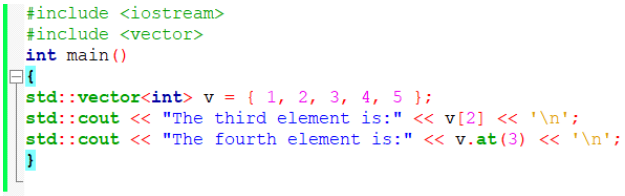
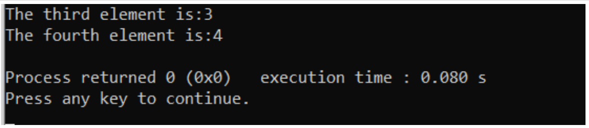
WHAT IS IT?

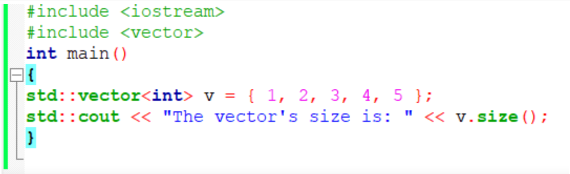
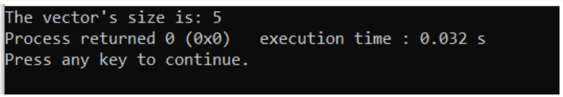
- Vector is a container defined in <vector> header. A vector is a sequnce of contiguous elements of any type.
- A vector and all other containers are implemented as class templates allowing for storage of any type

SYNTAX:


#include <vector>
int main()
{
std::vector<int> v = { 1, 2, 3, 4, 5 };
}
	-Here we defind a vector called v, of 5 integers, and we initialized a vector using brace initialization
	- Vecteor can grow and shrink on its own as we insert and delete elements into and from a vector.


#include <vector>
int main()
{
std::vector<int> v = { 1, 2, 3, 4, 5 };
v.push_back(10);
}
	- To insert an element at the end of the vector, we use the vector's.pusch_back() member function
	This example inserts a value of 10 at the end of our vector. Now we have a container of 6 elements: *1 2 3 4 5 10*


#include <iostream>
#include <vector>
int main()
{
std::vector<int> v = { 1, 2, 3, 4, 5 };
std::cout << "The third element is:" << v[2] << '\n';
std::cout << "The fourth element is:" << v.at(3) << '\n';
}
	
	The third element is:3
	The fourth element is:4
	
	Process returned 0 (0x0) execution time : 0.080 s
	Press any key to continue.
	Vector elements are indexed, the first element has an index of 0. Individual elements can be accessed via the subscript operator [element_index] or a member function at(element_index)


#include <iostream>
#include <vector>
int main()
{
std::vector<int> v = { 1, 2, 3, 4, 5 };
std::cout << "The vector's size is: " << v.size();
}
	
	The vector's size is: 5
	Process returned 0 (0x0) execution time : 0.032 s
	Press any key to continue.
	Vector's size as a number of elements, can be obtained through a .size() member function

- A vector is a sequential container, it stores elements in a sequence, other sequential containers are :
  - std ::list - a doubly linked list

- std::forward_list - A singly linked list
- std::deque - A double ended queu