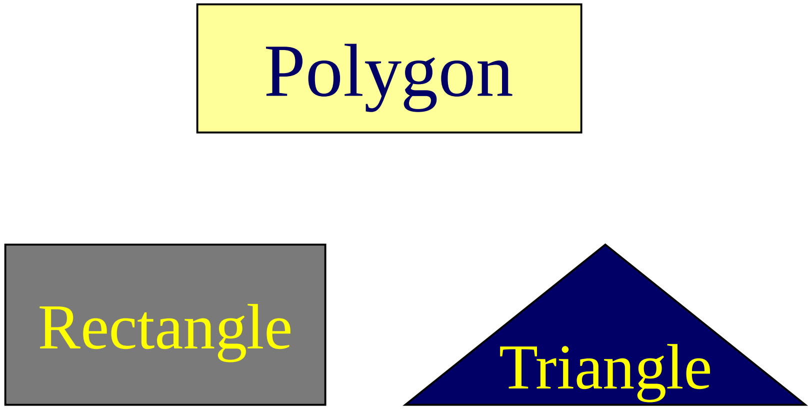
Inheritance Concept

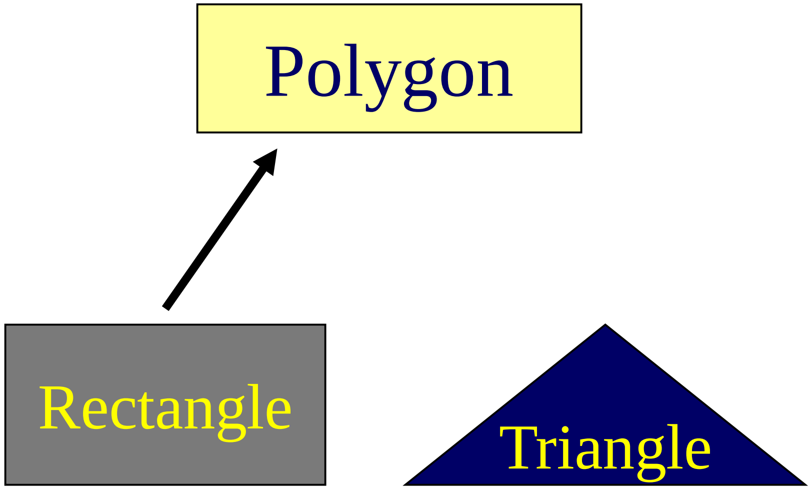


```
class Polygon{
  private:
   int numVertices;
   float *xCoord, *yCoord;
  public:
   void set(float *x, float *y, int nV);
};
```

```
class Rectangle{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
    float area();
};
```

```
class Triangle{
  private:
    int numVertices;
   float *xCoord, *yCoord;
  public:
   void set(float *x, float *y, int nV);
   float area();
};
```

Inheritance Concept



Polygon

Rectangle

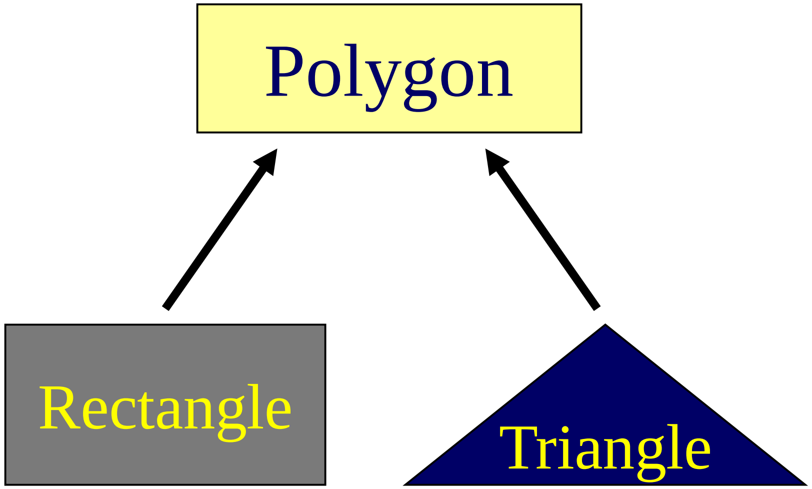
Triangle

```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
    public:
       float area();
};
```

```
class Rectangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
       float area();
};
```

Inheritance Concept



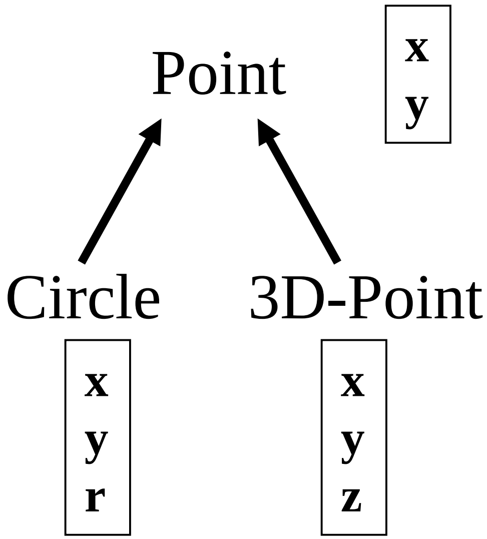
```
class Polygon{
      protected:
        int numVertices;
        float *xCoord, float *yCoord;
      public:
         void set(float *x, float *y, int nV);
};
```

```
class Triangle : public Polygon{
      public:
       float area();
};
```

⇔

```
class Triangle{
      protected:
        int numVertices;
        float *xCoord, float *yCoord;
      public:
         void set(float *x, float *y, int nV);
         float area();
};
```

Inheritance Concept



Point

x
y

Circle

x
y
r

3D-Point

x
y
z

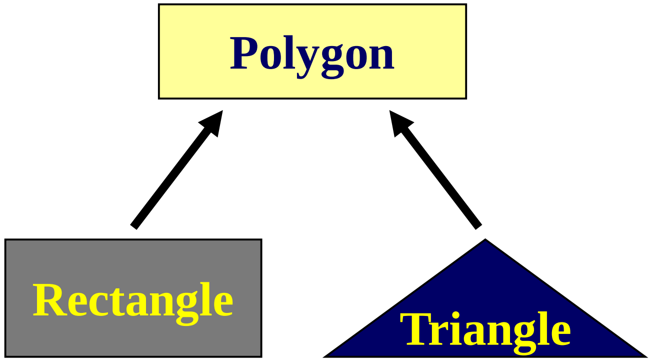
```
class Point{
    protected:
        int x, y;
    public:
        void set (int a, int b);
};
```

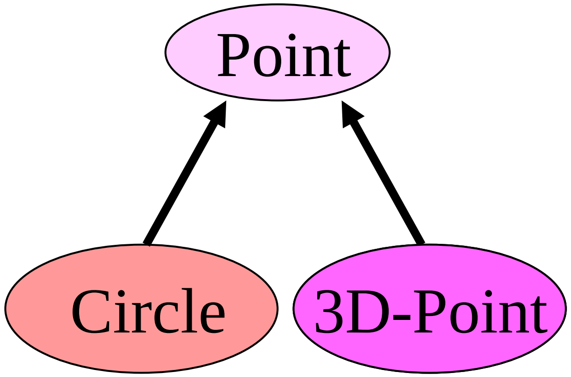
```
class Circle : public Point{
    private:
            double r;
};
```

```
class 3D-Point: public Point{
    private:
            int z;
};
```

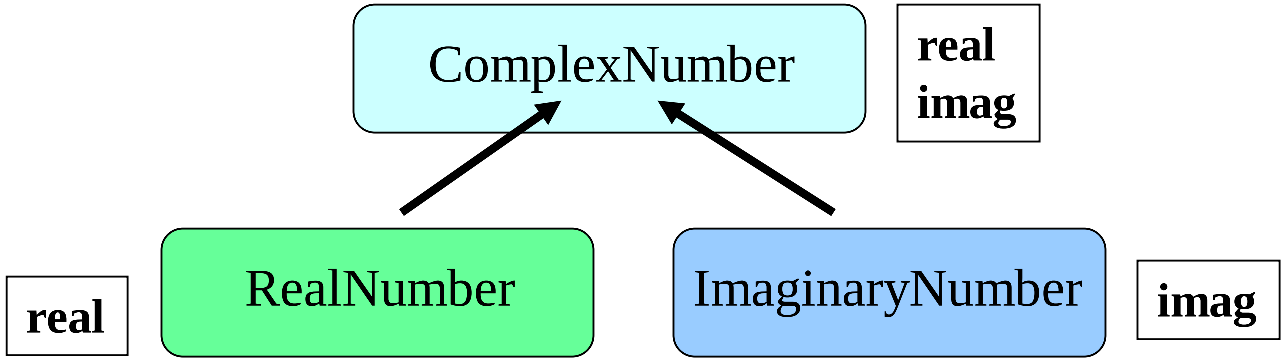
Inheritance Concept

- Augmenting the original class





- Specializing the original class



Why Inheritance ?

Inheritance is a mechanism for

- building class types from existing class types
- defining new class types to be a
    - specialization
    - augmentation

  of existing types

Define a Class Hierarchy

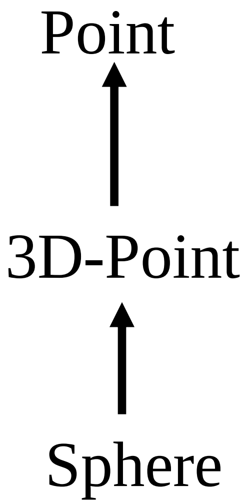
- **Syntax:**

  class *DerivedClassName* : access-level *BaseClassName*

  where

  - access-level specifies the type of derivation
    - private by default, or
    - public
- **Any class can serve as a base class**
  - Thus a derived class can also be a base class

Class Derivation



```
class Point{
    protected:
      int x, y;
    public:
      void set (int a, int b);
};
```

```
class 3D-Point : public Point{
    private:
      double z;
    … …
};
```

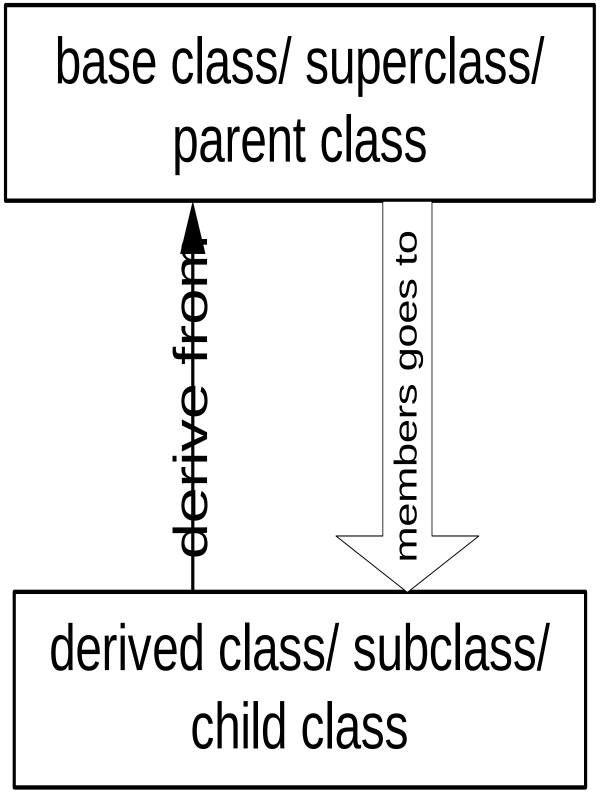
```
class Sphere : public 3D-Point{
    private:
      double r;
    … …
};
```

Point is the base class of 3D-Point, while 3D-Point is the base class of Sphere

What to inherit?

- **In principle, every member of a base class is inherited by a derived class**
  - just with different access permission

Access Control Over the Members



base class/ superclass/ parent class

derive from

members goes to

derived class/ subclass/ child class

- **Two levels of access control over class members**
  - class definition
  - inheritance type

```
class Point{
      protected: int x, y;
      public: void set(int a, int b);
};
```

```
class Circle : public Point{
      … …
};
```

Access Rights of Derived Classes

Type of Inheritance

Access Control for Members

	private	protected	public
private	-	-	-
protected	private	protected	protected
public	private	protected	public

- **The type of inheritance defines the access level for the members of derived class that are inherited from the base class**

Class Derivation

```
class mother{
      protected: int mProc;
      public: int mPubl;
      private: int  mPriv;
};
```

private/protected/public

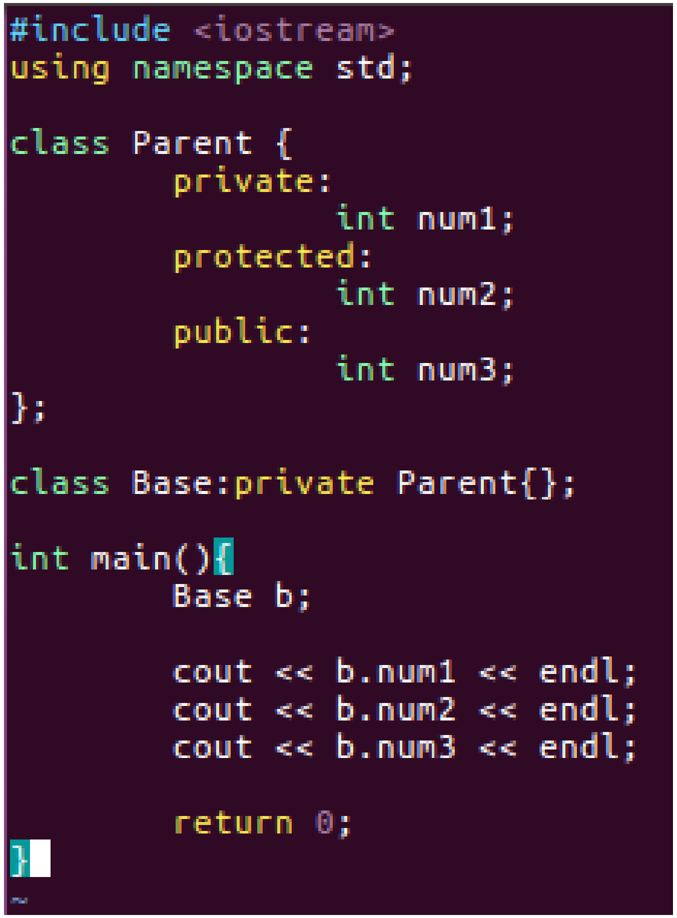
```
class daughter : --------- mother{
      private: double dPriv;
      public: void dFoo ( );
};
```

```
void daughter :: dFoo ( ){
      mPriv = 10;   //error
      mProc = 20;
};
```

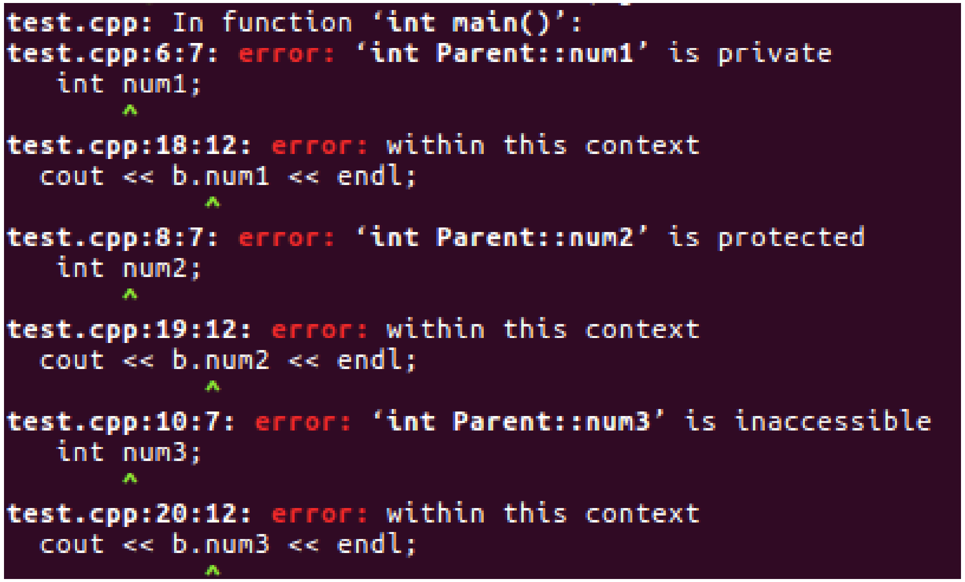
```
class grandDaughter : public daughter {
      private: double gPriv;
      public: void gFoo ( );
};
```

```
int main() {
      /*….*/
}
```

Access Rights of Derived Classes

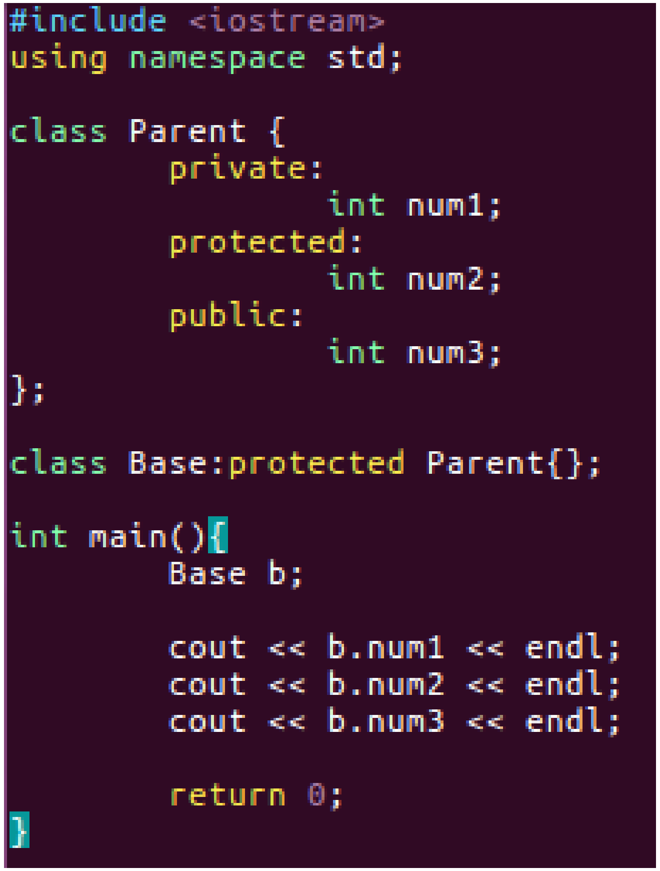


```
#include <iostream>
using namespace std;

class Parent {
        private:
                int num1;
        protected:
                int num2;
        public:
                int num3;
};

class Base:private Parent{};

int main(){
        Base b;

        cout << b.num1 << endl;
        cout << b.num2 << endl;
        cout << b.num3 << endl;

        return 0;
}
~
```

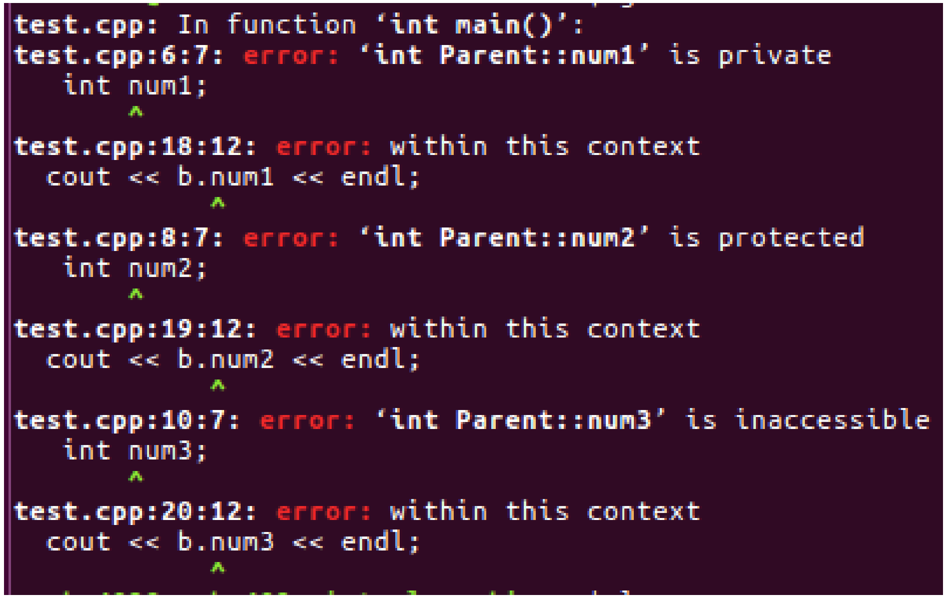


```
test.cpp: In function ‘int main()’:
test.cpp:6:7: error: ‘int Parent::num1’ is private
   int num1;
       ^
test.cpp:18:12: error: within this context
  cout << b.num1 << endl;
            ^
test.cpp:8:7: error: ‘int Parent::num2’ is protected
   int num2;
       ^
test.cpp:19:12: error: within this context
  cout << b.num2 << endl;
            ^
test.cpp:10:7: error: ‘int Parent::num3’ is inaccessible
   int num3;
       ^
test.cpp:20:12: error: within this context
  cout << b.num3 << endl;
            ^
```

Access Rights of Derived Classes

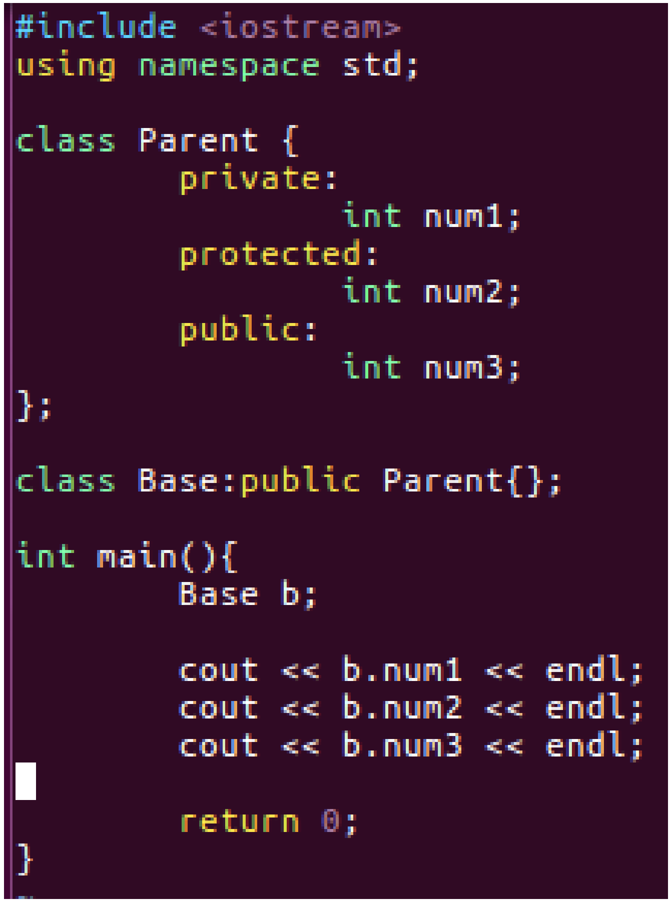


```
#include <iostream>
using namespace std;

class Parent {
        private:
                int num1;
        protected:
                int num2;
        public:
                int num3;
};

class Base:protected Parent{};

int main(){
        Base b;

        cout << b.num1 << endl;
        cout << b.num2 << endl;
        cout << b.num3 << endl;

        return 0;
}
```

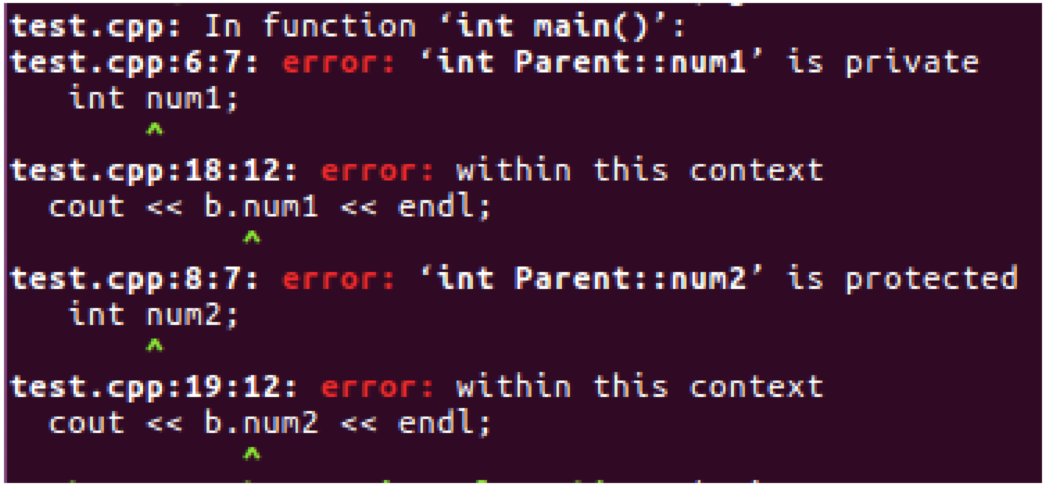


```
test.cpp: In function ‘int main()’:
test.cpp:6:7: error: ‘int Parent::num1’ is private
   int num1;
       ^
test.cpp:18:12: error: within this context
  cout << b.num1 << endl;
            ^
test.cpp:8:7: error: ‘int Parent::num2’ is protected
   int num2;
       ^
test.cpp:19:12: error: within this context
  cout << b.num2 << endl;
            ^
test.cpp:10:7: error: ‘int Parent::num3’ is inaccessible
   int num3;
       ^
test.cpp:20:12: error: within this context
  cout << b.num3 << endl;
            ^
```

Access Rights of Derived Classes



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

class Parent {
        private:
                int num1;
        protected:
                int num2;
        public:
                int num3;
};

class Base:public Parent{};

int main(){
        Base b;

        cout << b.num1 << endl;
        cout << b.num2 << endl;
        cout << b.num3 << endl;

        return 0;
}
```



```
test.cpp: In function ‘int main()’:
test.cpp:6:7: error: ‘int Parent::num1’ is private
   int num1;
       ^
test.cpp:18:12: error: within this context
  cout << b.num1 << endl;
            ^
test.cpp:8:7: error: ‘int Parent::num2’ is protected
   int num2;
       ^
test.cpp:19:12: error: within this context
  cout << b.num2 << endl;
            ^
```

What to inherit?

- **In principle, every member of a base class is inherited by a derived class**
  - just with different access permission
- **However, there are exceptions for**
  - constructor and destructor
  - operator=() member
  - friends

Since all these functions are class-specific

Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
   public:
      A ( )
         {cout<< “A:default”<<endl;}
      A (int a)
         {cout<<“A:parameter”<<endl;}
};
```

```
class B : public A
{
   public:
      B (int a)
            {cout<<“B”<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Constructor Rules for Derived Classes

You can also specify an constructor of the base class other than the default constructor

DerivedClassCon (derivedClass args) : BaseClassCon (baseClass args)
{ DerivedClass constructor body }

```
class A {
  public:
    A ( )
      {cout<< “A:default”<<endl;}
    A (int a)
      {cout<<“A:parameter”<<endl;}
};
```

```
class C : public A {
  public:
    C (int a) : A(a)
         {cout<<“C”<<endl;}
};
```

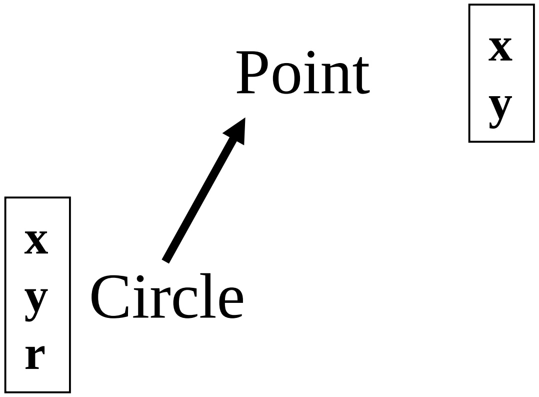
```
C test(1);
```

output:

```
A:parameter
C
```

Define its Own Members

The derived class can also define its own members, in addition to the members inherited from the base class



```
class Circle : public Point{
    private:
            double r;
    public:
            void set_r(double c);
};
```

```
class Point{
    protected:
        int x, y;
    public:
        void set(int a, int b);
};
```

```
class Circle{
    protected:
        int x, y;
    private:
        double r;
    public:
        void set(int a, int b);
        void set_r(double c);
};
```

Even more …

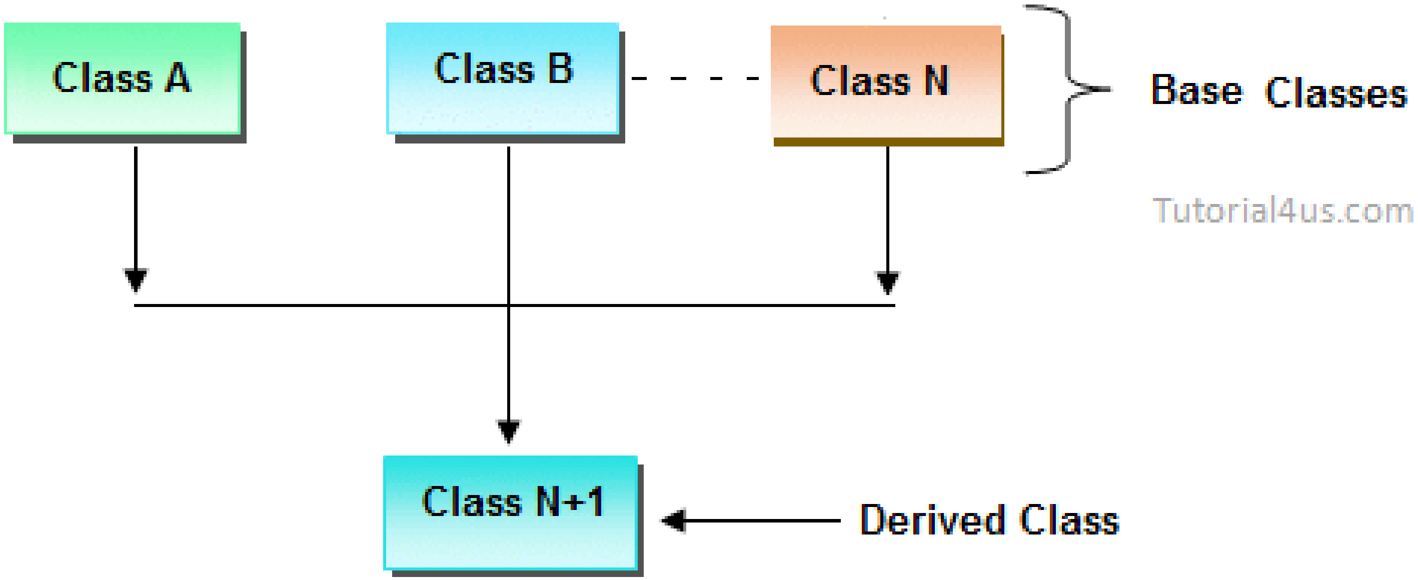
- **A derived class can override methods defined in its parent class. With overriding,**
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

```
class A {
  protected:
    int x, y;
  public:
    void print ()
            {cout<<“From A”<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
        {cout<<“From B”<<endl;}
};
```

Even more …

- Multiple Inheritance



Access Method

```
class Point{
      protected:
         int x, y;
      public:
         void set(int a, int b)
               {x=a; y=b;}
         void foo ();
         void print();
};
```

```
class Circle : public Point{
  private:  double r;
  public:
      void set (int a, int b, double c) {
            Point :: set(a, b); //same name function call
            r = c;
      }
      void print();  };
```

```
Point A;
A.set(30,50);  // from base class Point
A.print(); // from base class Point
```

```
Circle C;
C.set(10,10,100);   // from class Circle
C.foo ();  // from base class Point
C.print(); // from class Circle
```

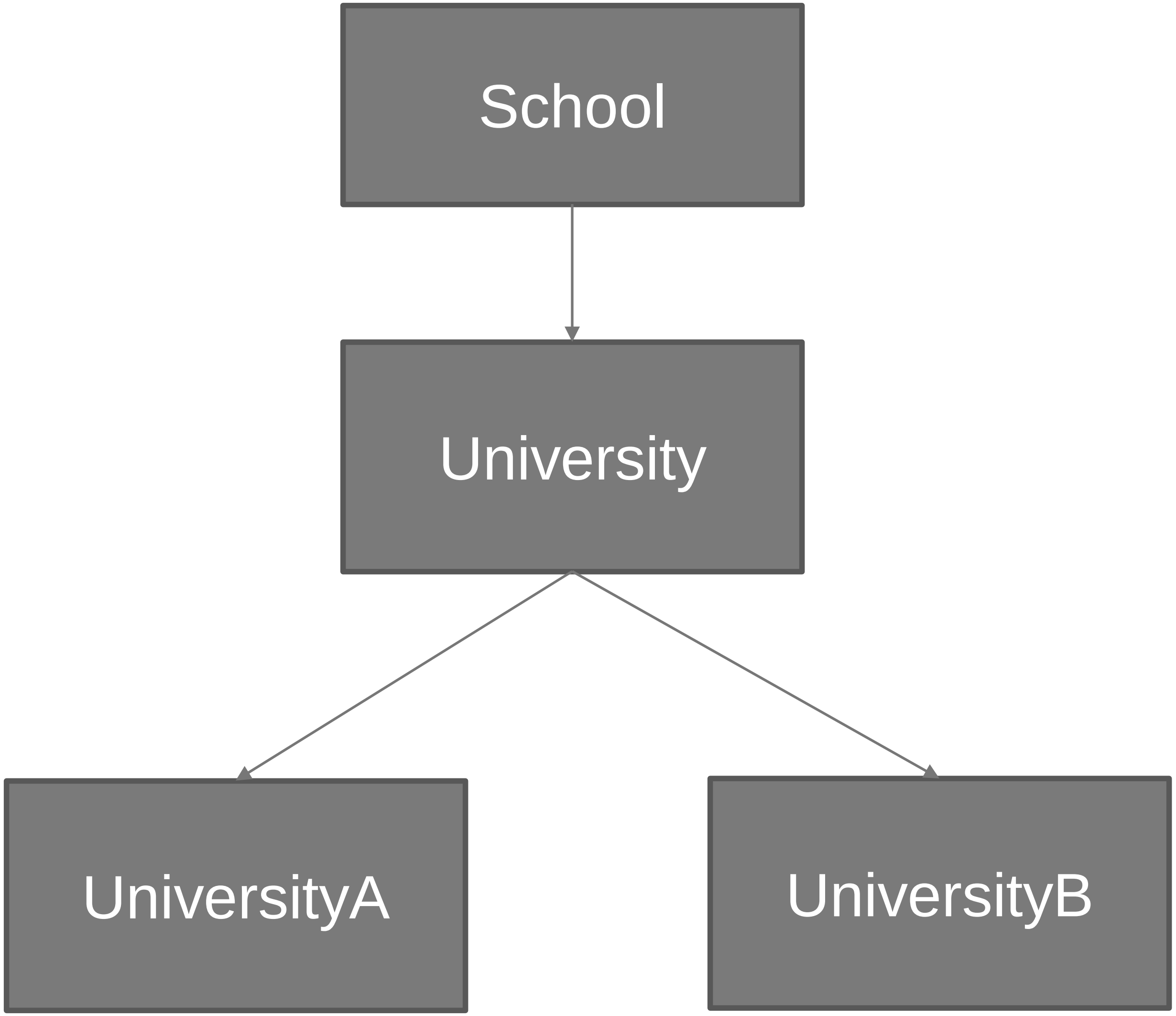
[Lab – Practice #1]

- **Calculate Average**

Class : student
Variables : float math, science, english, korean, average

Class : school
Variables : student students[10]
Methods : void calc_avg()
void print_result()

Class : university
Variables : int m_credit, s_credit, e_credit, k_credit
Methods : void calc_avg()


School
University
UniversityA
UniversityB

<score>

```
students[i].math = i * 5 + 20
students[i].science = i * 5 + 30
students[i].english = i * 5 + 40
students[i].korean = i * 5 + 50
```

<main>

```
int main(){
        universityA univ_a(3, 4, 1, 2);
        universityB univ_b(2, 1, 4, 3);
}
```

<output>

```
University A
student 0, math : 20, science : 30, english : 40, korean : 50, average : 32
….
University B
student 0, math : 20, science : 30, english : 40, korean : 50, average : 38
```