

OBJECT ORIENTED PROGRAMMING

LAB 5

- PUBLIC-PUBLIC SIMPLE INHERITANCE
- PUBLIC-PRIVATE SIMPLE INHERITANCE,
- CONSTRUCTOR INHERITANCE,
- CALL ORDER OF CONSTRUCTORS/DESTRUCTORS IN INHERITANCE,
- *THIS* POINTER

A few of the concepts used in this laboratory

1. Inheritance

Inheritance is the OOP specific means used to reuse code already written.

Why reuse code at all? If we would not reuse the code our programs would be larger in size (and require more expensive computing resources – which may not be a problem in modern PCs but it is still a problem in embedded/battery powered microcontroller & microprocessor devices). Our code could also have a bug in one of the implementations of a given functionality, but not in others, and with a source code base of thousands lines of code (LoC) or more this can become difficult to take into consideration/find/debug.

We typically make use of inheritance if we want to particularize an abstraction but keep some of the features (data + functionality) from an existent but more general abstraction. With this particularization we specialize the derived class versus its base.

Example base class	Example derived class(es)
Person	Pupil, Student, Teacher, Employee, ...
Shape	Line, Rectangle, Square, Circle, Pyramid, Cone, ...
Publication	NewsPaper, Magazine, Book, ScientificPaper, ...

In C++ terminology, the code we reuse is found in a class we call **BASE CLASS** (or **PARENT CLASS**) that has attributes and methods.

Terminology to identify the base class	Terminology used to identify the derived class
parent, base	derived, derivate, child, subclass

Code reuse requires writing one **DERIVATE** (or **CHILD**) **CLASS** (or multiple derivative classes) that will have both their own attributes and methods as well as reuse attributes /methods in the base class.

Note, there are also exceptions: elements of the class that are not inherited by derived classes include: constructors, destructors, friend functions. This has implications, for example, by defining a constructor in the base class, we don't have a corresponding constructor automatically defined in the derived class.

The number of member elements of the derived class is the sum between the number of elements in the base class (those inherited) and those belonging directly to the derived class.

The derived class “is a “ with regard to the base class. For a derived class “Magazine” if it is inherited from class “Book” we can say “Magazine is a Book.”

This relationship between classes is accompanied by an access modifier that can be one of: public, protected or private, describing the way through which the derived class inherits (has access to member elements of the base class).

The general syntax in C++ for inheritance is:

```
class name_of_derived: access_modifier name_of_base [, access_modifier name_of_baseB ...]  
{  
  //member elements for derived class  
  //similar class element composition  
};
```

Attention! Derived classes also end with “};”.

After the class keyword and the derived class name we use the “:” followed by the access modifier for inheritance and the name of the base class.

Attention! In the highlighted inheritance syntax above we do not use the scope resolution operator, but the “:” operator.

The inheritance access modifier changes how an element of the base class can be accessed for a derived object. The rule is to pick the most restrictive access type from combining the element access modifier in the base class with the derived class inheritance access modifier.

base \ inherited	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

2. Constructor inheritance

Constructors and destructors do not get inherited. But the Derived class constructor can call its correspondent from the base class.

We will use a simple example that we will modify slightly in a few iterations to showcase that constructors don't get inherited.

3. Call order of constructors and destructors in inheritance

We first call the base constructor, then the derived constructor.

For destructors is the reverse.

Let's make an analogy between constructing and destructing a derived object and layers of an onion.

Step 0. There is nothing.

Step 1. The core of the onion gets “brought into the world”. Analogue, the constructor of the base class gets called.

Step 2. The immediate outer layer in the shell of the onion gets “brought into the world”. The constructor of the derived class gets called.

When we “destroy” the onion we need to start peeling from the outer layer.

Step 0. The outer layer of the onion gets “peeled off” existence. Analogue, the derived class destructor gets called.

Step 1. The core of the onion gets “removed from existence”. Analogue, the destructor of the base class gets called.

Step 2. There is nothing left.

4. THIS Pointer

Any C++ object has access to its own memory address through the pointer `this`. This is passed as a silent argument to all member functions that are particular to the instance (object).

- Friend functions and not member functions so they do not have the pointer `this`.
- Static member functions are not particular to an iteration, so they don't have the pointer `this` either.

EXAMPLE 1 – SIMPLE PUBLIC-PUBLIC INHERITANCE

The first example from this lab introduces the concept of inheritance with a simple public-public class inheritance into a single derivate class from a single base class.

```
#include <iostream>    //std::cout, std::cin, std::endl, std::string
using namespace std;

class BASE
{//create class BASE
    int x;//private member / attribute
    public://the following will be public
        void initX(int n)//setter for x
            {x=n;}
        void printX()//quasi-getter for x
            {cout<<x;}
};//always close the class with semicolon ;!

class DERIVATE:public BASE
{//create a derivate class
    int y;//private member / attribute
    public://the following will be public
        void initY(int n)//setter for y
            {y=n;}
        void printY()//quasi-getter for y
            {cout<<y;}
};//always close the class with semicolon ;!

int main()
{//create the derivate class object.
    DERIVATE D1;
    D1.initX(10);//set the value for x: OK(no errors)
    D1.initY(20);//set the value for y: OK(no errors)
    D1.printX();//display the value for x: OK(no errors)
    D1.printY();//display the value for y: OK(no errors)
    D1.x=30; //ERROR! X is not accessible because it is private in base class
            //even if inheritance is public
    D1.y=40; //ERROR! Y is not accessible because it is private in (derived) class
}
```

Conclusions:

- Methods `initX()` and `printX()` are accessible because `DERIVATE` publicly inherits `BASE`, and in `BASE` these methods are public.
- Attributes `x` and `y` are not accessible being private (`x` in `BASE`, `y` in `DERIVATE`), and the public inheritance access modifier makes no difference.

(Homework).5.1.

In the prior example what would happen if the DERIVED class still inherited publicly the BASE class, and the base class had a public attribute *z*? Would we be able to call it directly from main() like in the code sequence below?

```
[...]
class BASE
{
    //create class BASE
    int x;//private member / attribute
    public://the following will be public
        int z;
        void initX(int n)//setter for x
            {x=n;}
        void printX()//quasi-getter for x
            {cout<<x;}
}; //always close the class with semicolon ;!
[...]
int main()
{
    //create the derivate class object.
    DERIVATE D1;
    D1.z=30; //???
}
```

EXAMPLE 2 – PUBLIC-PRIVATE INHERITANCE

We change the prior example to have public methods in base class but private inherited in derived class.

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

class BASE
{
    //create class BASE
    int x;//private attribute
    public://the following are public
    void initX(int n)//Setter for x
    {
        x=n;
    }
    void printX()//non-standard method to display x
    {
        cout<<x;
    }
}; //always close class with };

class DERIVATE:private BASE
{
    //create a derivate class
    int y;//private member
    public://the following are public
    void initY(int n)//setter for y
    {
        y=n;
    }
    void printY()//non-standard method to display y
    {
        cout<<y;
    }
}
```

```
};
int main()
{//create an object of type DERIVATE
    DERIVATE d1;
    //d1.initX(10);//Error: initX not accessible (private from main() perspective)
    d1.initY(20);//can set Y because initY is public
    //d1.printX();//Error: printX not accessible (private from main() perspective)
    d1.printY();//display Y without errors because printY is public
}
```

Errors that appear in this scenario are similar to:

Moodle C++ Error	<pre>test.cpp: In function 'int main()': test.cpp:34:16: error: 'void BASE::initX(int)' is inaccessible within this context d1.initX(10);//Error: initX not accessible (private from main() perspective) ^ test.cpp:8:10: note: declared here void initX(int n)//Setter for x ^~~~~~ test.cpp:34:16: error: 'BASE' is not an accessible base of 'DERIVATE' d1.initX(10);//Error: initX not accessible (private from main() perspective) ^ test.cpp:36:15: error: 'void BASE::printX()' is inaccessible within this context d1.printX();//Error: printX not accessible (private from main() perspective) ^ test.cpp:12:10: note: declared here void printX();//non-standard method to display x ^~~~~~ test.cpp:36:15: error: 'BASE' is not an accessible base of 'DERIVATE' d1.printX();//Error: printX not accessible (private from main() perspective)</pre>
DevC++ Error	<pre>D:\POO\LAB5\15.1.cpp In function 'int main()': 9 8 D D:\POO\LAB5\15.1.cpp [Error] 'void BASE::initX(int)' is inaccessible 29 13 D D:\POO\LAB5\15.1.cpp [Error] within this context 29 13 D D:\POO\LAB5\15.1.cpp [Error] 'BASE' is not an accessible base of 'DERIVATE' 11 8 D D:\POO\LAB5\15.1.cpp [Error] 'void BASE::printX()' is inaccessible 31 10 D D:\POO\LAB5\15.1.cpp [Error] within this context 31 10 D D:\POO\LAB5\15.1.cpp [Error] 'BASE' is not an accessible base of 'DERIVATE'</pre>

As we can see the effect of the private inheritance modifier is that object d1 of class type DERIVATE does no longer have access to methods initX and printX even if they were public in BASE class.

EXAMPLE 2 (FIXED) – InitXY, PrintXY - PUBLIC-PRIVATE INHERITANCE

We wish to find a solution through which , by keeping the private inheritance modifier, object d1 to be able to set through *init* functions and display through *print* functions values for both attribute x of BASE class as well as attribute y of DERIVATE class.

Solution consists in one of:

- Building a new set of methods
 - initXY() that also calls initX; setting both x and y
 - initX() in the scope of BASE has access to the x attribute
 - printXY() that also calls printX; printing both x and y
 - printX() in the scope of BASE has access to the x attribute

Implementation in problem 5.2 (VPL).

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

```

class BASE
{
//create class BASE
    int x;//private attribute
    public://the following are public
    void initX(int n)//Setter for x
    {
        x=n;
    }
    void printX()//non-standard method to display x
    {
        cout<<x<<endl;
    }
};

class DERIVATE:private BASE
{
//create a derivate class
    int y;//private member
    public://the following are public
    void initXY(int n, int m)//setter for y, and x after calling initX
    {
        initX(n);//x=n;
        y=m;
    }
    void printXY()//non-standard method to display y& x after calling printX
    {
        printX();
        cout<<y<<endl;
    }
};

int main()
{
//create an object of type DERIVATE
    DERIVATE d1;
    d1.initXY(10, 20);//ok
    d1.printXY();//ok
}

```

Behavior in main: object *d1* calls method *initXY* with 2 parameters, *initXY* in turn calls *initX* with the first parameter. *initX* initializes *x* with the first parameter value. Back in *initXY* *y* is initialized with the second parameter value.

Object *d1* calls method *printXY* which on execution, calls *printX* to display *x*, then directly displays *y* (*y* being an attribute in *DERIVATE*).

Conclusion: private inheritance does not mean totally “locking” access into *BASE*, but by calling indirectly (in cascade) we were able to access these methods of the *BASE* class.

(Homework) 5.2.

Homework: Apply the technique in EXAMPLE 2 in the case then we have a private inheritance on multiple levels. Ex: Class *C* inherits privately class *B* which inherits privately class *A*. *A* holds attribute *x*, *B* holds attribute *y*, *C* holds attribute *z*. Init *x* with 10, *y* with 20, *z* with 30 and print.

Solve in problem placeholder *Homework 5.2 A,B,C (VPL)*.

EXAMPLE 3 – CONSTRUCTOR IN INHERITANCE

We wish to analyze the behavior of constructors in case of class inheritance. We use program 5.3 (VPL).

```
#include <iostream> //std::cin, std::cout, std::endl, <string> => std::string
using namespace std;
class BASE
{
    int a;
public:
    BASE()
    {
        a=4;
        cout << "IMPLICIT CONSTRUCTOR CALLED" << endl;
    }
    BASE(int aa)
    {
        a=aa;
        cout << "CONSTRUCTOR WITH 1 ARG. CALLED"<< a <<endl;
    }
};

class DERIVATE:public BASE
{//compiler will offer an implicit constructor for DERIVATE
};
int main()
{
    DERIVATE D1; //OK, using compiler created constructor for DERIVATE
    return 0;
}
```

In general, in C++, upon object creation of the type of a derived class, it is first necessary to call the constructor of the base class and then follows the call of a constructor of the derived class. In the example above, to create object D1 it would have been necessary to create a constructor in the base class, specifically an implicit constructor, which we only have for BASE class, not for DERIVATE class.

However the above code compiles and runs without errors. We could say that the constructor is inherited in DERIVED and used upon creation of D1, but this is not true. What really happens is the compiler creates for us an implicit constructor and uses it to create object D1. We will prove a bit later (**) in this lab that **constructors do not get inherited**.

EXAMPLE 4 – DERIVATE DOES NOT INHERIT CONSTRUCTOR

To verify the way the constructor gets created for DERIVATE, we change the code (we copy from 5.3(VPL) into 5.4(VPL) and make the following change in main()).

```
int main()
{
    DERIVATE D1 (40); //NOT OK, no constructor with parameters in DERIVATE
    return 0;
}
```

It gives an error since constructors don't get inherited. BASE had a constructor with 1 parameter.

In this case we should call an implicit constructor in base class (already written) through a constructor with 1 parameter in derived class. Since there is no constructor with 1 parameter in the DERIVATE class, the program returns an error. The compiler can only create implicit constructors. We get an error because the DERIVATE(int) is missing:

```
15.5.cpp: In function 'int main()':
15.5.cpp:24:15: error: no matching function for call to 'DERIVATE::DERIVATE(int)'
  DERIVATE D1(40); //NOT OK, no constructor with parameters in DERIVATE
                ^
15.5.cpp:19:7: note: candidate: DERIVATE::DERIVATE()
  class DERIVATE:public BASE
```

This error proves that a constructor with 1 parameter is sought, and one can't be created by the compiler, it can only create an implicit compiler (due to *candidate: DERIVATE: DERIVATE ()* line).

In order to fix this issue, we modify in 5.4(VPL) the DERIVATE class from empty to have a constructor with 1 argument (and also an attribute) like in the code snippet below.

```
class DERIVATE:public BASE
{
//compiler will NOT offer an implicit constructor for DERIVATE.
//we created more than 0 constructors in our code
int b;
public:
    DERIVATE(int bb){
        b=bb;
        cout <<"CONSTRUCTOR WITH 1 ARG. CALLED - IN DERIVATE" << endl;
    }
};
```

Compilation will succeed and we will have as output:

--- Program output ---

```
IMPLICIT CONSTRUCTOR CALLED
CONSTRUCTOR WITH 1 ARG. CALLED - IN DERIVATE
```

The above messages prove (**) that upon creation of a derived class object one uses a constructor with 1 parameter in derived class and an implicit constructor in base class.

We will expand on this example in future labs.

EXAMPLE 5 – CALL ORDER FOR CONSTRUCTORS IN INHERITANCE

Starting from example 5.4 (VPL) source code we copy over the working source code into in 5.5(VPL) modify the definition of D1 in main () back to call the implicit constructor upon creation of D1.

```
int main()
{
DERIVATE D1; //NOT OK, no implicit constructor created by the compiler in DERIVATE
return 0;
}
```

We again observe the error message in the compiler.

```
15.5.cpp: In function 'int main()':
15.5.cpp:30:10: error: no matching function for call to 'DERIVATE::DERIVATE()'
  DERIVATE D1; //NOT OK, no implicit constructor created by the compiler in DERIVATE
      ^~
15.5.cpp:23:5: note: candidate: DERIVATE::DERIVATE(int)
  DERIVATE(int bb){
```

This time the compiler looks for an implicit constructor, but does not create it because we created a constructor with parameters in DERIVATE.

This again proves what we told before (**)**constructors don't get inherited.**

To fix the error we modify 5.5 (VPL) to have both an implicit constructor and a constructor with 1 parameter in class DERIVATE.

```
class DERIVATE:public BASE
{
int b;
public:
  DERIVATE(){
    b=5;
    cout <<"IMPLICIT CONSTRUCTOR CALLED - IN DERIVATE"<<endl;
  }
  DERIVATE(int bb){
    b=bb;
    cout <<"CONSTRUCTOR WITH 1 ARG. CALLED - IN DERIVATE" << endl;
  }
};
```

This allows us to create 2 objects in main.

```
int main()
{
DERIVATE D1; // OK, we have also implicit constructor in DERIVATE
DERIVATE D2(300);
return 0;
}
```

The program output will be:

```
IMPLICIT CONSTRUCTOR CALLED
IMPLICIT CONSTRUCTOR CALLED - IN DERIVATE
IMPLICIT CONSTRUCTOR CALLED
CONSTRUCTOR WITH 1 ARG. CALLED - IN DERIVATE
```

Program displays output from the 2 objects:

- for D1 it calls the implicit constructor in BASE, then the implicit one in DERIVATE,
- for D2 it calls the implicit constructor in BASE, then the one with 1 parameter in DERIVATE

EXAMPLE 6 – CALL ORDER FOR CONSTRUCTORS AND DESTRUCTORS

We wrap up this sequence of examples with 5.6 (VPL) in which we also include destructors for BASE and DERIVATE.

The purpose of the exercise is to check out the order of calls for constructors and destructors for an object based on an inherited class.

```

#include <iostream>
using namespace std;
class BASE
{
    int a;
public:
    BASE()
    {
        a=4;
        cout << "IMPLICIT CONSTRUCTOR CALLED" << endl;
    }
    BASE(int aa)
    {
        a=aa;
        cout << "CONSTRUCTOR WITH 1 ARG. CALLED"<< a <<endl;
    }
    ~BASE(){
        cout << "DESTRUCTOR CALLED - IN BASE"<<endl;
    }
};

class DERIVATE:public BASE
{
    int b;
public:
    DERIVATE(){
        b=5;
        cout <<"IMPLICIT CONSTRUCTOR CALLED - IN DERIVATE"<<endl;
    }
    DERIVATE(int bb){
        b=bb;
        cout <<"CONSTRUCTOR WITH 1 ARG. CALLED - IN DERIVATE" << endl;
    }
    ~DERIVATE(){
        cout << "DESTRUCTOR CALLED - IN DERIVATE"<<endl;
    }
};

int main()
{
    DERIVATE D1; //OK, using constructors created by programmer
    return 0;
}

```

The program will output:

```

IMPLICIT CONSTRUCTOR CALLED
IMPLICIT CONSTRUCTOR CALLED - IN DERIVATE
DESTRUCTOR CALLED - IN DERIVATE
DESTRUCTOR CALLED - IN BASE

```

As we can see, **upon object creation the program will first call constructor in the base class then the constructor in derived class**, and upon object destruction, **the order is reversed, first destructor in derived class, then destructor in base class.**

EXAMPLE 7 – A MORE COMPLEX PUBLIC-PUBLIC EXAMPLE

This exercise wishes to reunite elements from prior labs with a simple inheritance. The problem statement clearly defines what your code should implement as well as format for input and output data as one would in a verification type of situation.

Create a program implementing a class SQUARE, inherited by a class PYRAMID, with the following characteristics:

- SQUARE having:
 - Attributes (*side* – of an integer type, *color* – of a word type).
 - Getters and Setters for the 2 attributes
 - Implicit constructor, a version of constructor with 2 parameters and a copy constructor
 - Destructor
 - Non-standard member function *Area*
 - Friend functions *Perimeter*,
- With PYRAMID, inheriting class SQUARE publicly, having:
 - Attribute (*height* – of a fractional type available in C++)
 - Setter, getter for *height*
 - Non-standard method *Volume*
- With main(), all fractional number printing will use 2 digits after separator
 - Object SQUARE S1, calling implicit constructor;
 - Reading *side* from standard input
 - Reading *color* from standard input
 - Setting the *side* and *color* for the object
 - Printing "side: " followed by *side* and " color: ", followed by *color*, followed by newline
 - Object SQUARE S2, calling constructor with 2 parameters
 - Reading *side* from standard input
 - Reading *color* from standard input
 - Setting the *side* and *color* for the object
 - Printing "side: " followed by *side* and " color: ", followed by *color*, followed by newline
 - Object SQUARE S3, copying elements from S2;
 - Object PYRAMID PD1, calling implicit constructor;
 - Reading *height* from standard input
 - Setting *side* from *side* of S1
 - Setting *color* from *color* of S2
 - Printing "side: " followed by *side*, " color: ", followed by *color*, " height: ", followed by *height*, then followed by newline
 - Displaying perimeter for S1, followed by newline ($P = 4 * side$)
 - Displaying area for S2, followed by newline, ($A = side^2$)
 - Displaying volume for PD1, followed by newline ($V = A * height / 3$)

Example of interaction with console (standard input and output).

Input	Explanation
1 red 2 blue 3.0	<i>side</i> and <i>color</i> for S1 <i>side</i> and <i>color</i> for S2 <i>Height</i> for PD1
Output	Explanation
<i>side</i> : 1 <i>color</i> : red <i>side</i> : 2 <i>color</i> : blue <i>side</i> : 1 <i>color</i> : blue <i>height</i> : 3.00 4 4 1.00	Output for object S1 Output for object S2 Output for PD1 (<i>side</i> for S1, <i>color</i> from S2, <i>height</i> from input) Perimeter for S1: $4 * 1 = 4$ Area for S2: $2^2 = 4$ Volume for PD1: Area of S1 = $1^2 = 1$. Height = 3. => $V = 1.00$

Note this problem is closer to what you can expect to get in terms of requirements for a final verification/colloquium at the OOP course.

```
#include <iostream>
#include <string> //even if your C++ compiler includes string via iostream other environments don't
#include <math.h> //for pow()
#include <iomanip> //for setprecision and fixed

using namespace std;

#ifdef VERBOSE_PROGRAM //if uncommented it will enable printouts in my methods, to improve debugging

class SQUARE
{ //attributes will be private
    int side; //is private because we don't want it to be modified directly
    string color; //is private
public: //and methods will be public
    //setter generally returns void
    void SetSide(int dim);
    void SetColor(string clr);

    //getter will have the return type identical to the attribute
    int GetSide();
    string GetColor();

    //constructors following next
    //they have no return type
    //and always have the same name as the class
    SQUARE(); //declaration (prototype) for the implicit constructor

    //a constructor with parameters doesn't need to have the same
    //number of parameters as the class has attributes
    //some attributes can be set implicitly or through setter
    //dim and clr have local visibility in the object
    SQUARE(int dim, string clr); //declaration (prototype) for a constructor with parameters

    //copy ct. declaration
    //copy ct. get a reference to a SQUARE
    SQUARE(const SQUARE &s);

    ~SQUARE(); //destructor declaration

    //friend function for class SQUARE
    //friend functions have access to all elements of a class
    //including PROTECTED and PRIVATE
    //if we erase the reserved keyword friend from function Perimeter
    //it no longer has access via arrow -> to private attribute "side"
    friend int Perimeter(SQUARE *SP);

    int Area();
}; //class always ends with "};"

//outside class we define methods for class SQUARE with resolution operator

//definition (implementation) for setter for "side"
void SQUARE::SetSide(int dim)
{
    side=dim;
}

//definition (implementation) for setter for "color"
void SQUARE::SetColor(string clr)
{
    color=clr;
}

//definition (implementation) for getter for "side"
```

```

int SQUARE::GetSide()
{//always returns attributes. not local variables
    return side;
}

//definition (implementation) for getter for "color"
string SQUARE::GetColor()
{//always returns attributes. not local variables
    return color;
}

//definition (implementation) for implicit constructor
SQUARE::SQUARE()
{//implicit constructor sets default values for attributes
//for objects created without parameters
    side=10;
    color="green";
#ifdef VERBOSE_PROGRAM
    cout<<"Created an implicit object"<<endl;
#endif
}

//definition (implementation) for constructor with parameters
SQUARE::SQUARE(int dim, string clr)
{//constructor with parameters sets values for attributes with values given upon object creation
    side=dim;
    color=clr;
#ifdef VERBOSE_PROGRAM
    cout<<"Created an object with parameters"<<endl;
#endif
}

SQUARE::SQUARE(const SQUARE &s)
{//each attribute (eg: side) gets the corresponding value
//of the same attribute in the source object (eg: s.side)
    side=s.side;
    color=s.color;
#ifdef VERBOSE_PROGRAM
    cout<<"Copied an object"<<endl;
#endif
}

SQUARE::~~SQUARE()
{//destructor to erase objects from memory when
//reaching end of scope for object
//program the destructor to display message
#ifdef VERBOSE_PROGRAM
    cout<<"Destroyed an object"<<endl;
#endif
}

//function which is NOT a member of the class, so DON'T use resolution operator
//no longer using the keyword friend outside the class
//definition for friend function
int Perimeter(SQUARE *S)
{//perimeter computation
    int perimeter;
    perimeter=4*S->side;//S is pointer, using -> to access attribute
    //side is private, but friend function also has access to PROTECTED and PRIVATE elements
    return perimeter;
}

//Area is member function in class, so we define it with the resolution operator ::
int SQUARE::Area()
{//compute area
    int area;
    area=pow(side,2);
    return area;
}

```

```

//create derived class PYRAMID
//PYRAMID inherits elements of SQUARE
class PYRAMID : public SQUARE//using operator : , not operator ::
{
    float height;
public://declare and define within the class scope ...
    void SetHeight(float dim) //setter
    {
        height=dim;//dim is local variable in function, can be reused
    }
    float GetHeight() //getter
    {
        return height;
    }
    float Volume() //non-standard method
    {
        float volume;
        //method Volume has access to Area() because public-public
        //inheritance allows it
        volume=Area()*height/3;
        return volume;
    }
};//derived class as any class ends with "};"

int main()
{
int _side; string _color; float _height;

    SQUARE S1;//no params=> calling implicit constructor

    cin >> _side >> _color; //read from input
    S1.SetSide(_side); S1.SetColor(_color); //set attributes for object
    //print side and color
    cout << "side: " << S1.GetSide() << " color: " << S1.GetColor()<<endl;

    SQUARE S2 (15, "blue");//2 params (int, string)
        //=> calling the proper constructor with parameters

    cin >> _side >> _color; //read from input
    S2.SetSide(_side); S2.SetColor(_color); //set attributes for object
    //print side and color
    cout << "side: " << S2.GetSide() << " color: " << S2.GetColor()<<endl;

    SQUARE S3=S1;//obj copy -> calling copy constructor
    //we were not required to display anything for it

    //create derived class PYRAMID object
    PYRAMID PD1;//compiler will make me an implicit constructor
        //because I have no other constructors in my class PYRAMID

    cin >> _height; PD1.SetHeight(_height);
    PD1.SetSide(S1.GetSide());//PYRAMID object has access to setters from class SQUARE
    PD1.SetColor(S2.GetColor());
    cout <<setprecision(2) <<fixed;
    cout << "side: " << PD1.GetSide() << " color: " << PD1.GetColor() << " height: " <<
    PD1.GetHeight()<<endl;

    cout <<Perimeter(&S1)<<endl ;
    cout <<S2.Area() <<endl;
    cout <<PD1.Volume()<<endl;
}

```

(Homework) 5.3.

Adapt the Square Pyramid example for the Moodle/VPL homework with base class *Circle* and derived class *Cone*.

- With Circle having:
 - Attributes (*radius* – of a fractional type, *color* – of a word type).
 - Getters and Setters for attributes
 - Implicit constructor, constructor with 2 parameters and a copy constructor
 - Destructor
 - Non-standard member function *Area*
 - Friend functions *Perimeter*,
- With Cone having:
 - Attribute (*height* – of a fractional type available in C++)
 - Setter, getter for *height*
 - Non-standard method *Volume*
- With main():
 - Object Circle ci1;
 - Reading radius from standard input
 - Reading color from standard input
 - Object Cone c2;
 - Reading height from standard input
 - Displaying perimeter for ci1 with 2 decimals
 - Displaying area for ci1 with 2 decimals
 - Displaying volume for c2 with 2 decimals

Indications:

No other text displayed or VPL will fail.

Circle perimeter: $P=2\pi R$, Circle area: $A=\pi R^2$, Cone Volume: $V = A \frac{h}{3}$.

For π you can use constant `M_PI` from `math.h`.

To represent fractional numbers with 2 decimal precision you can review the prior lab material.

Example of interaction with console (standard input and output).

Input	Explanation
1.234	Radius for ci1
blue	Color for ci1
5.678	Height for c2
Output	Explanation
7.75	Perimeter of ci1
4.78	Area of ci1
9.05	Volume of c2