**Lab 6**

**Object Composition**

**Objectives**

**Following programming skills will be acquired in this lab:**

* To understand the concept of **has-a** relationship.
* To understand the design of a **composite** class.
* To learn the use of **class member initializer** syntax.
* To understand use of **this** keyword.

**Composition**

In real-life, complex objects are often built from smaller, simpler objects. For example,

* A car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts.
* A personal computer is built from a CPU, a motherboard, some memory, etc.
* Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on.

***This process of building complex objects from simpler ones is called composition (object composition).***

More specifically, ***composition is used for objects that have a has-a relationship to each other***.

A car *has-a* metal frame, *has-an* engine, and *has-a* transmission.

A personal computer *has-a* CPU, a motherboard, and other components.

You *have-a* head, a body, some limbs.

So far, all the classes we have used in our examples have had member variables that are built-in data types (eg. int, float, char). While this is generally sufficient for designing and implementing small, simple classes, it quickly becomes burdensome for more complex classes, especially those built from many sub- parts. To facilitate the building of complex classes from simpler ones, C++ allows us to do object composition in a very simple way — by using classes as member variables in other classes.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object to implement features of your new class, but the user of your new class sees the interface you’ve defined rather than the interface from the original class. To do this, you follow the typical path of embedding **private** objects of existing classes inside your new class.

Occasionally, however, it makes sense to allow the class user to directly access the composition of your new class, that is, to make the member objects **public**. The member objects use access control themselves, so this is a safe thing to do and when the user knows you’re assembling a bunch of parts, it makes the interface easier to understand. A **Car** class is a good example.

# Example 6.1

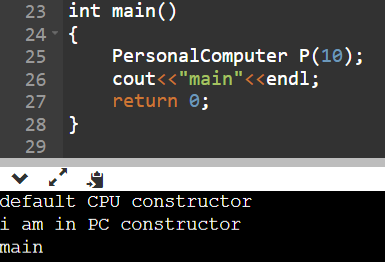
|  |  |
| --- | --- |
| # include<iostream>  using namespace std;  class **Engine**  {  public:  void **start**() const {}  void **rev**() const {}  void **stop**() const {}  };    class **Wheel**  {  public:  void **inflate**(int psi) const {}  };    class **Window**  { public:  void **rollup**() const  { cout <<"\nThe window has been rolled up"; }  void **rolldown**() const  { cout<<"\nThe window has been rolled down"; }  }; | class **Door** {  public:  Window window;  void **open**() const {}  void **close**() const {}  };  class **Car** {  public:  Engine engine;  Wheel wheel[4];  Door left, right; // 2 doors  };    void **main**() {  Car car;  car.left.window.rollup(); car.right.window.rolldown(); car.wheel[0].inflate(72); system("pause");    } |

Text

Description automatically generated

# Different variations of concept:

# 



# Initializing Class Member Variables

If we were designing a personal computer class, we might do it like this (assuming we’d already written a CPU, Motherboard, and RAM class):

|  |
| --- |
| #include "CPU.h"  #include "Motherboard.h"  #include "RAM.h"    class **PersonalComputer**  { private:  CPU mCPU;  Motherboard mMotherboard;  RAM mRAM; }; |

The preferred way to initialize class members is through initializer lists rather than assignment. So let’s write a constructor for our PersonalComputer class that uses an initialization list to initialize the member variables. This constructor will take 3 parameters: a CPU speed, a motherboard model, and a RAM size, which it will then pass to the respective member variables when they are constructed.

PersonalComputer::PersonalComputer(int nCPUSpeed, char \*strMboard, int nRAMSize) : mCPU(nCPUSpeed), mMotherboard(strMboard), mRAM(nRAMSize)

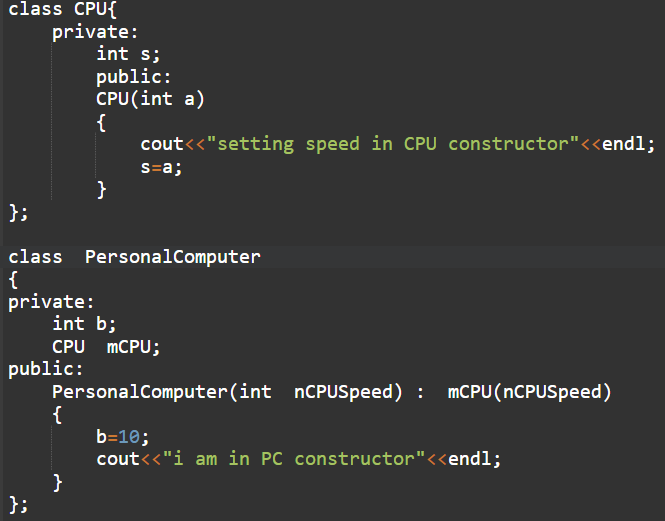
{

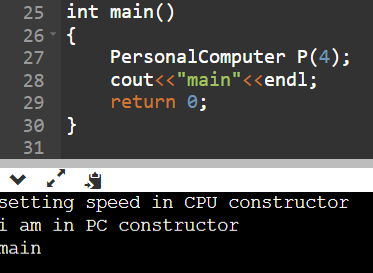
}

Now, when a PersonalComputer object is instantiated using this constructor, that PersonalComputer object will contain a CPU object initialized with nCPUSpeed, a Motherboard object initialized with strMotherboardModel, and a RAM object initialized with nRAMSize.

It is worth explicitly noting that composition implies ownership between the complex class and any subclasses. **When the complex class is created, the subclasses are created. When the complex class is destroyed, the subclasses are similarly destroyed.**

**Example:**





## Example 6.2

Many games and simulations have creatures or objects that move around a board, map, or screen. The one thing that all of these creatures/objects have in common is that they all *have-a* location. In this example, we are going to create a creature class that uses a point class to hold the creature’s location. Our creature is going to live in a 2d world, so our point class will have 2 dimensions, X and Y. We will assume the world is made up of discrete squares, so these dimensions will always be integers.

|  |
| --- |
| #include <iostream> #include <string> using namespace std;  class **Point2D**  {  private:  int mnX;  int mnY;  public:  Point2D() : mnX(0), mnY(0)  { }  Point2D ( int nX, int nY) : mnX(nX), mnY(nY)  {}  void **output** ()  {  cout << "("<< GetX() << ", " << GetY() << ")"<<endl;  }  void **SetPoint** (int nX, int nY)  { mnX = nX;  mnY = nY; }    int **GetX**() const { return mnX; }  int **GetY**() const { return mnY; }  }; |

|  |
| --- |
| class **Creature**  {  private:  string strName;  Point2D Location;  public:  Creature() { }  Creature(string Name, Point2D Location)  {  strName = Name;  Location = Location ;  }  void **show**()  {  cout <<strName << " is at " ;  Location.output(); }    void **MoveTo**(int nX, int nY)  { Location.SetPoint(nX, nY); }  }; |

|  |
| --- |
| int **main**()  {  cout << "Enter a name for your creature: "; string Name;  cin >> Name;  Creature cCreature(Name, Point2D(4, 7));  OR  *Point2D object (4,7);*  *Creature cCreature(Name, object);*    while (1)  {  cCreature.show();  cout << "Enter new X location for creature (-1 to quit): ";  int nX=0;  cin >> nX;  if (nX == -1)  break;    cout << "Enter new Y location for creature (-1 to quit): ";  int nY=0;  cin >> nY;  if (nY == -1)  break;    cCreature.MoveTo(nX, nY);  }  system("pause");  return 0;  } |

**Program Output:**

Enter a name for your creature: Marvin

Marvin is at (4, 7)

Enter new X location for creature (-1 to quit): 6

Enter new Y location for creature (-1 to quit): 12

Marvin is at (6, 12)

Enter new X location for creature (-1 to quit): 3

Enter new Y location for creature (-1 to quit): 2

Marvin is at (3, 2)

**Example 6.3**

|  |  |
| --- | --- |
| #include <iostream>  #include <string>  using namespace std;  class **Date**  { public:  Date( int = 1, int = 1, int = 1900 );  void print() ;  ~Date();  private:  int month;  int day;  int year;  };  Date::**Date**( int mn, int dy, int yr )  {  month = mn;  year = yr;  day = dy;  }  void Date::**print**() const  { cout << month << '/' << day << '/' << year; }  Date::~**Date**()  { cout << "Date object destructor for date "; cout << endl; }    class **Employee**  {  public:  Employee(string, string, Date, Date );  void print() ;  ~Employee();  private:  string firstName;  string lastName;  Date birthDate;  Date hireDate;  }; | |
| Employee::**Employee**( string first, string last, **Date** dateofBirth, **Date** dateofHire): birthdate (dateofBirth) , hireDate (dateofHire)  {  firstName = first;  lastName = last;  }  OR  Employee::**Employee**( string first, string last, **Date** dateofBirth, **Date** dateofHire)  {  firstName = first;  lastName = last;  birthdate = dateofBirth;  hireDate = dateofHire;  }    void Employee::**print**()  { cout << lastName << ", " << firstName << " Hired: "; hireDate.print(); cout << " Birthday: "; birthDate.print(); cout << endl;  }    Employee::~**Employee**()  { cout << "Employee destructor: " << lastName << ", " << firstName << endl; }    int **main**()  {  Date birth( 7, 24, 1949 );  Date hire( 3, 12, 1989 );    Employee manager( "A", "B", birth, hire );  cout << endl;  manager.print();    return 0;  } | |
|  | |



# The keyword this

The keyword this represents a pointer to the object whose member function is being executed.

***It is a pointer to the object itself.***

One of its uses can be to check if a parameter passed to a member function is the object itself.

## Example 6.4

|  |  |
| --- | --- |
| #include <iostream> using namespace std;    class **Dummy** {  public:  int isItMe (Dummy& param);  }; | **Output**:  yes, &a is b |
| int Dummy:: **isItMe** (Dummy& param)  {  if (&param == this)  return true;  else  return false;  }  int **main** () {  Dummy a;  Dummy\* b = &a;  if ( b-> isItMe (a) )  cout << "yes, &a is b";  return 0;  } |  |

***Refer this when you will learn operator overloading concept***

It is also frequently used in operator= member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an operator= function similar to this one:

|  |
| --- |
| Vector& Vector::operator= (const Vector& param)  {  x=param.x; y=param.y; return \*this;  } |

**Practical Exercises:**

**Exercise 6.1**

Using the **Point** class write a composite class called **Triangle**. The Triangle class will have three point objects as its data members (to store three vertices of a triangle object). The class should have a constructor, a function to compute area of a triangle and a print function.

**Code:**

#include <iostream>

using namespace std;

class Point {

int x;

int y;

public:

Point(int a=0, int b=0) {

setX(a);

setY(b);

}

void setX(int a) {

x = a;

}

void setY(int a) {

y = a;

}

int getX() const {

return x;

}

int getY() const {

return y;

}

};

class Triangle {

float area;

Point p1, p2, p3;

public:

Triangle(Point a, Point b, Point c) : p1(a), p2(b),p3(c) {

area = computeArea();

}

float computeArea() {

float a = (p1.getX() \* (p2.getY() - p3.getY())) +

(p2.getX() \* (p3.getY() - p1.getY())) +

(p3.getX() \* (p1.getY() - p2.getY()));

a = abs(a);

float tempArea = a/2;

area = tempArea;

return area;

}

void display()const {

cout << "Area of the triangle is : " << area;

}

};

int main() {

int x, y;

cout << "Enter X and Y for vertex 1 : " << endl;

cin >> x >> y;

Point p1(x, y);

cout << "Enter X and Y for vertex 2 : " << endl;

cin >> x >>y;

Point p2(x, y);

cout << "Enter X and Y for vertex 3 : " << endl;

cin >> x >>y;

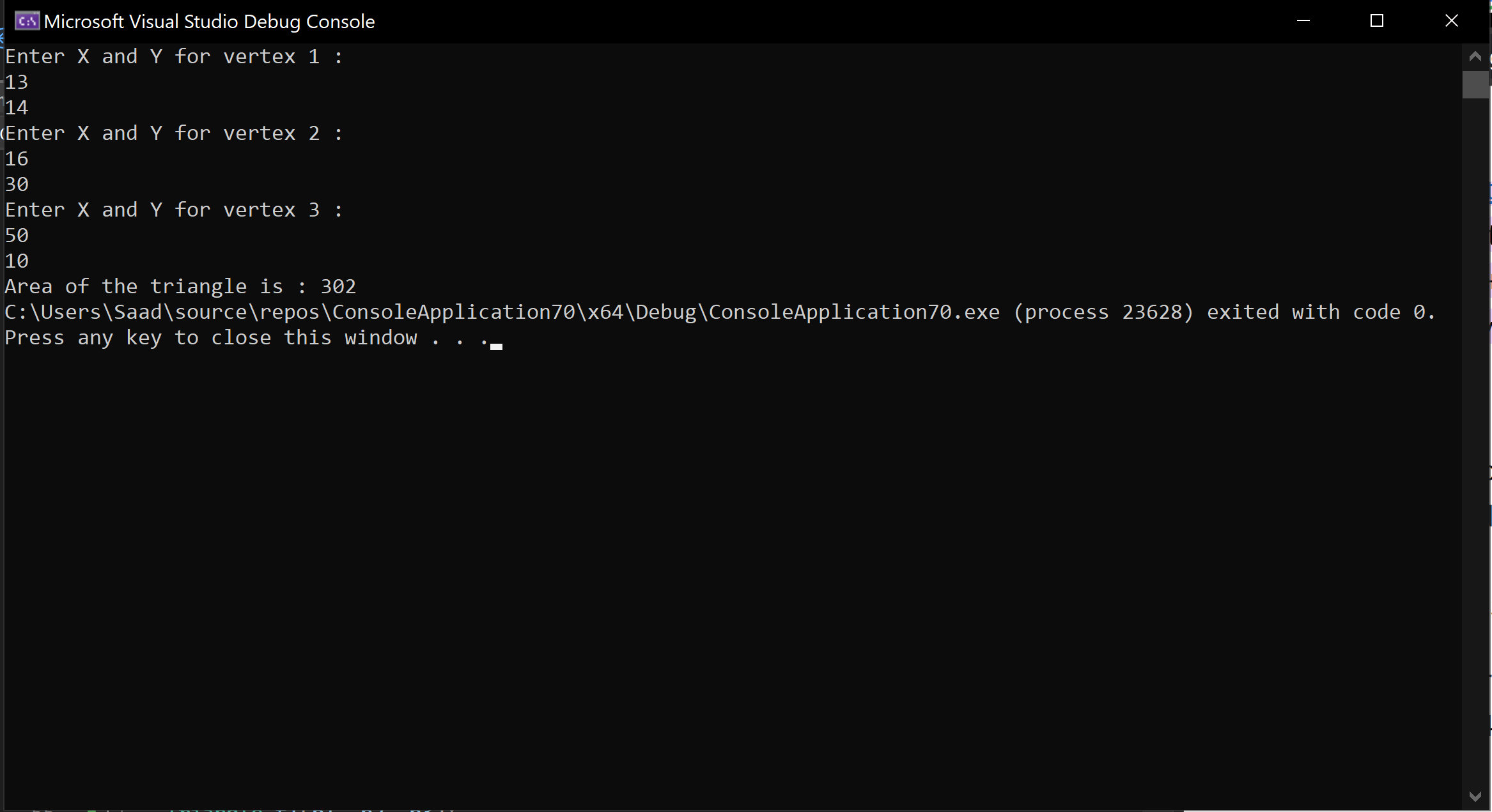
Point p3(x, y);

Triangle t1(p1, p2, p3);

t1.display();

}

**Output:**

****

**Exercise 6.2**

Write a program to display the name and birthday of a Person. Create a class Person. It has first name and last name. Create a class Date. It has day, month and year. Using composition, create a class called **BirthdayInfo**.

Write a test program which creates an instance of the BirthdayInfo class, and displays the name and birthday for that person.

**Code:**

#include <iostream>

using namespace std;

class Person {

string firstName;

string lastName;

public :

Person(string a = " ", string b = " ") {

setFname(a);

setLname(b);

}

void setFname(string a) {

firstName = a;

}

void setLname(string a) {

lastName = a;

}

void display() const {

cout << "Name : " << firstName << " ";

cout << lastName << endl;

}

};

class Date {

int day;

int month;

int year;

public:

Date(int a, int b, int c) {

setDay(a);

setMonth(b);

setYear(c);

}

void setDay(int a) {

day = a;

}

void setMonth(int a) {

month = a;

}

void setYear(int a) {

year = a;

}

void display()const {

cout << "Birthdate : " << day << "/" << month << "/" << year << endl;

}

};

class BirthdayInfo{

Person p;

Date d1;

public:

BirthdayInfo(Person a, Date b) : p(a), d1(b) {

display();

}

void display() const {

p.display();

d1.display();

}

};

int main() {

Person p1("Saad", "Ahmad");

Date d1(14, 3, 2004);

BirthdayInfo b1(p1, d1);

}

**Output:**

