BM 593 Numerical Methods & C Programming

10th week Object Oriented Programming with C++ Language

3 Major Properties of Object Oriented Programming

1. Encapsulation (Data+Methods put together in the class structure)
2. Inheritance (Hierarchy among classes)
3. Polymorphism (The same name or symbol can be used multiple times within different contexts)

```
enum Boolean {false, true};
struct Point {
   int X, Y;
   Boolean Visible;
   int GetX() {return X;}
};
struct Point {
   int X, Y;
   Boolean Visible;
   int GetX();
};
int Point::GetX(){return X;}
struct Point {
   int X, Y;
   Boolean Visible;
   int GetX(){return X;};
   Point (int NewX, int NewY); // Constructor Declaration
};
   Point (int NewX, int NewY){ // Constructor Definition
     X=NewX;
     Y=NewY;
     Visible=false;
   };
Point Origin(1,1)
Point::Point(int NewX=0,int NewY=0){
     X=NewX;
     Y=NewY;
     Visible=false;
```

```
}
Point Origin(5);
```

Inheritance with

private : can be accessed only by member functions declared within the same class

protected : can be accessed only by member functions within the same class and by member functions of classes that are derived from this class

public : can be accessed from anywhere within the same scope as the class definition

```
class Point {
  int X;
  int Y;
  public :
    int GetX();
    Point (int NewX, int NewY);
};
```

Class is private by default whereas struct is public.

```
enum Boolean {false, true};


class Employee {
  double salary;
  Boolean permanent;
  Boolean professional;

  public:
     char name[50];
     char dept_code[3];

  private:
    int Error_check(void);

  public :
     Employee(double salary, Boolean permanent,
   Boolean professional, char *name, char *dept_code);
};



// point.cpp
```

```
#include <iostream.h>

class Point {
  int X;
  int Y;
  public :

    Point (int InitX, int InitY){X=InitX; Y=InitY;}
    int GetX(){return X;}
    int GetY(){return Y;}
};

   int main(){

      int YourX, YourY;

      cout << "Set X coordinate ";
      cin >> YourX;

      cout << "Set Y coordinate ";
      cin >> YourY;

      Point YourPoint(YourX, YourY);

      cout << "X is " << YorPoint.GetX();
      cout << '\n';
      cout << "Y is " << YorPoint.GetY();
      cout << '\n';

      return 0;
   }

   // point.h


   enum Boolean {false, true};

   class Location {
```

Access in base class	Access Modifier	Inherited access in base
public	public	public
private	public	not accessable
protected	public	protected
public	private	private
private	private	not accessable
protected	private	private

```
    protected:
     int X;
     int Y;

    public:
     Location (int InitX, int InitY);
     int GetX();
     int GetY();
   };

   class Point: public Location { // X and Y are protected within Point

      protected:
        Boolean Visible;

      public:
         Point (int InitX, int InitY);
         void Show();
         void Hide();
         Boolean IsVisible();
         void MoveTo(int NewX, int NewY);
   };


// circle.cpp

#inclue "point.h"
#include <conio.h>  // for getch() function


class Circle : Point {
```

```
int Radius;

  public :

  Circle(int InitX, int InitY, int InitRadius);
  void Show(void);
  void Hide(void);
  void Moveto(int NewX, int NewY);


};

Circle::Circle (int InitX, int InitY, int InitRadius): Point(InitX,InitY)
  {
    Radius = InitRadius;
  }

void Circle::Show(void)
{
    Visible=true;
    circle(X,Y,Radius);
}

void Circle::Hide(void)
{
    Visible=false;
    circle(X,Y,Radius);
}

void Circle::MoveTo(int NewX, int NewY){

 Hide();
 X=NewX;
 Y=NewY;
 Show();
};

int main(){
  Circle MyCircle(100,200,50);
```

```
  MyCircle.Show();
  getch();
  MyCircle.MoveTo(200,250);
  getch();
  return 0;
}
```

Member function **Circle::Show** needs to access Boolean Visible from base its class **Point**. Now Visible is protected in **Point** and **Circle** is derived privately from **Point**. So Visible is **private** witin **Circle** and is accessible just like radius. If Visible had been defined as **private** in **Point** it would have been inaccessible to member functions of **Circle**. However, if you make Visible **public** in **Point** then Visible would be accessible by non-member functions as well. Since Visible is protected in **Point** and **Circle** is derived **private** from **Point**, member functions of **Circle** can access to Visible without exposing it to **public** abuse.

1. Members X and Y are declared **protected** in **Location**.
2. **Point** specifies **public** derivation from **Location**, so **Point** also inherits the X and Y members as **protected**.
3. **Circle** is derived from **Point** using the default **private** derivation.
4. **Circle** therefore inherits X and Y as **private**. **Circle::Show** can access X and Y. Note that X and Y are still **protected** within **Location**.

```
// mcircle.cpp

#inclue "point.h"
#include <conio.h>  // for getch() function
#include <string.h>  // for string functions
class Circle : public Point {

  protected :
   int  Radius;
  public :
    Circle (int InitX, int InitY, int Radius);
    void Show(void);
};

class GMessage : public Location {
   char *msg;
```

```
   int Field;
   public:
   GMessage (int msgX, int msgY, int FieldSize, char *text);
   void Show(void);
};

class MCircle :Circle, GMessage {
public :
  MCircle(int mcircX, int mcircY, int mcircRadius, char *msg);
  void Show (void);
};
class MCircle : Circle , GMessage {
  public :
     MCircle(int mcircX, int mcircY, int mcircRadius, char *msg );
     void Show (void);
};

Circle::Circle (int InitX, int InitY, int InitRadius): Point(InitX,InitY){
    Radius = InitRadius;
}
void Circle::Show(void){
    Visible=true;
    circle(X,Y,Radius);
}

GMessage::GMessage (int msgX, int msgY, int FieldSize, char *text) : Location(msgX,msgY) {
  Font MsgFont;
  Field=FieldSize;
  msg=text;
};

void GMessage::Show(void){
   int size =Field/ (8*strlen(msg));
}


MCircle::MCircle(int mcircX, int mcircY,

int mcircRadius, int Font,char* msg) : Circle (mcircX, mcircY,mcircRadius),
```

```
GMessage(mcircX,mcircY,2*mcircRadius,msg)
{
}
void Mcircle::Show(void){
  Circle::Show();
  GMessage::Show();
}
```

Calling virtual Functions in a Base Class Constructor

```
#include<iostream.h>
class Base{
 public:
   Base(){
     cout << "Base: constructor. Calling clone()" << endl;
     clone();
   }
   virtual void clone(){
     cout << "Base::clone() called" << endl;
   }
};

class Derived: public Base{
  public:
     Derived(){
       cout << "Derived: constructor"<<endl;
     }
  void clone(){
    cout << "Derived::clone() called" << endl;
  }
};
void main(){
  Derived x;
  Base *p = &x;
  // call clone thru pointer to class instance
  cout << "Calling clone through instance pointer";
  cout << endl;
  p->clone();
}
```

The output is

Base: constructor. Calling clone()

Base::clone() called

Derived: constructor

Calling clone thru instance pointer

Derived::clone() called

When you create an instance of a derived class the compiler first calls the constructor of the base class. At this point the derived class is partilally initialized. Therefore, when the compiler calls the clone() virtual function it cannnot bind it to the virtual clone. It instead calls the base clone(). Once the object is created through a base class pointer's invoking the derived clone function the last two lines are produced. If you call a virtual function in a class constructor the compiler invokes the base class version of the function not the version defined for the derived class.

Virtual Base Class

```
#include <iostream.h>
class device{
    public:
       device();
       {cout << "device: constructor" << endl;}
};
class comm_device: public device{
  public:
     comm_device()
       {cout << "comm_device: constructor" << endl;}
};
class graphics_device: public device{
   public:
      graphics_device(){
        cout << "graphics_device: constructor" << endl;}
};
class graphics_terminal: public comm_device, public graphics_device{
   public:
     graphics_terminal();
     {cout << "graphics_terminal: constructor"<< endl;}
};
void main(){
  graphics_terminal gt;
}
```

On the Output screen

```
device: constructor
comm_device: constructor
graphics_device: constructor
graphics_terminal: constructor
```