Section 3:Lecture 8

- Polymorphism

Object-Oriented Concept

- Encapsulation
  - Abstract Data Type (ADT), Object
- Inheritance
  - Derived object
- Polymorphism
  - Each object knows what it is

Polymorphism – An Introduction

- Definition
  - *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance

Before We Proceed…

- Inheritance – Basic Concepts
  - Class Hierarchy
    - Code Reuse, Easy to maintain
  - Type of inheritance : public, protected, private
  - Function overriding

class **Time** Specification

```
// SPECIFICATION   FILE                                    ( time.h)
        class  Time
        {
           public :
                void     Set ( int h, int m, int s ) ;
                void     Increment ( ) ;
                void     Write ( )  const ;
                Time     (int initH, int initM, int initS ) ;
                Time     ( ) ;
           protected :
                int              hrs ;
                int              mins ;
                int              secs ;
        } ;
```

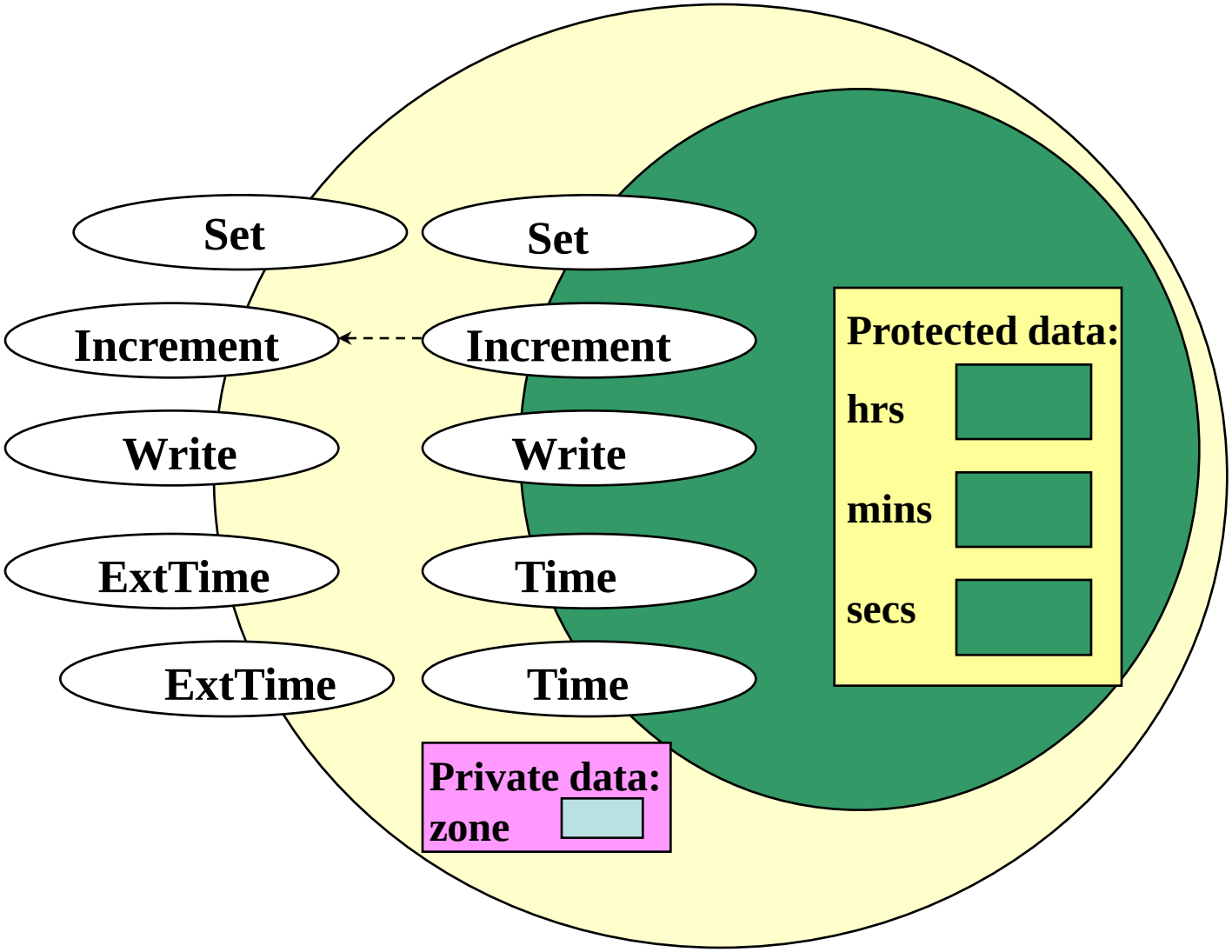
Derived Class **ExtTime**

```
// SPECIFICATION  FILE                              ( exttime.h)
#include  "time.h"
enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time
{
   public :
        void       Set ( int h, int m, int s, ZoneType timeZone ) ;
        void       Write ( )  const;    //overridden
        ExtTime    (int initH, int initM, int initS, ZoneType initZone ) ;
        ExtTime    ();
   private :
        ZoneType  zone ;        //  added data member
} ;
```

Class Interface Diagram

ExtTime class



Why Polymorphism?--Review:
Time and *ExtTime* Example by Inheritance

```
void  Print  (Time   someTime )                    //pass an object by value
{
        cout  <<  "Time is  " ;
        someTime.Write (  ) ;                      // Time :: write()
        cout  <<  endl ;
}
```

CLIENT CODE

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

Time is 08:30:00
Time is 10:45:00

Static Binding

- When the type of a formal parameter is a parent class, the argument used can be:

  the same type as the formal parameter,

  or,

  any derived class type.
- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter
- When pass-by-value is used, static binding occurs

Can We Do Better?

```
void  Print  (Time   someTime )                    //pass an object by value
{
        cout  <<  "Time is  " ;
        someTime.Write (  ) ;                      // Time :: write()
        cout  <<  endl ;
}
```

CLIENT CODE

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00
```

Polymorphism – An Introduction

- Definition
  - *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance
- Allows run-time interpretation of object type for a given class hierarchy
  - Also Known as "Late Binding"
- Implemented in C++ using virtual functions

Dynamic Binding

- Is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference

Virtual Member Function

```
//  SPECIFICATION FILE                    ( time.h )

class  Time
{
public :
    .  .  .
    virtual  void  Write ( ) ;          //  for dynamic binding
    virtual ~Time();                    // destructor

private :

    int             hrs ;
    int             mins ;
    int             secs ;
} ;
```

This is the way we like to see...

```
void  Print  (Time *   someTime )
{
   cout  <<  "Time is  " ;
   someTime->Write (  ) ;
   cout  <<  endl ;
}
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00  CST
```

CLIENT CODE

```
Time        startTime( 8, 30, 0 ) ;
ExtTime     endTime(10, 45, 0, CST) ;

Time *timeptr;
timeptr = &startTime;
Print ( timeptr ) ;

timeptr = &endTime;
Print ( timeptr ) ;
```

Time::write()

ExtTime::write()

Virtual Functions

- Virtual Functions overcome the problem of run time object determination
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- How ?
  - Define a virtual function in the base class. The word virtual appears only in the base class
  - If a base class declares a virtual function, it must implement that function, even if the body is empty
  - Virtual function in base class stays virtual in all the derived classes
  - It can be overridden in the derived classes
  - But, a derived class is not required to re-implement a virtual function. If it does not, the base class version is used

Polymorphism Summary

- When you use virtual functions, compiler store additional information about the types of object available and created
- Polymorphism is supported at this additional overhead
- Important :
  - virtual functions work only with pointers/references
  - Not with objects even if the function is virtual
  - If a class declares any virtual methods, the destructor of the class should be declared as virtual as well.

Abstract Classes & Pure Virtual Functions

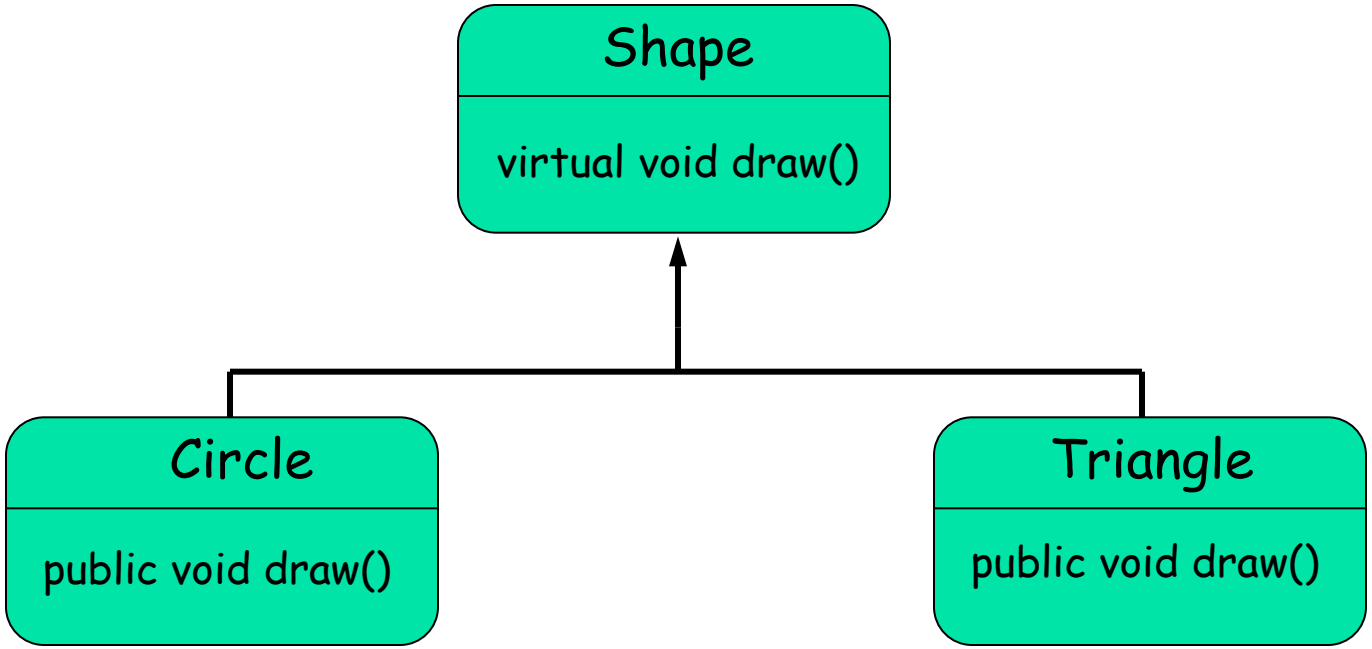
- Some classes exist logically but not physically.
- Example : Shape
  - Shape s; // Legal but silly..!! : "Shapeless shape"
  - Shape makes sense only as a base of some classes derived from it. Serves as a "category"
  - Hence instantiation of such a class must be prevented

```
class Shape    //Abstract
{
  public :
  //Pure virtual Function
  virtual void draw() = 0;
}
```

- A class with one or more pure virtual functions is an Abstract Class
- Objects of abstract class can't be created

Shape s; // error : variable of an abstract class

Example


Shape
virtual void draw()
Circle
public void draw()
Triangle
public void draw()

- A pure virtual function not defined in the derived class remains a pure virtual function.
- Hence derived class also becomes abstract

```
class Circle : public Shape {                    //No draw() - Abstract
    public :
    void print(){
      cout << "I am a circle" << endl;
    }
class Rectangle : public Shape {
    public :
    void draw(){                                 // Override Shape::draw()
      cout << "Drawing Rectangle" << endl;
    }
```

```
Rectangle r;          // Valid
Circle c;             // error : variable of an abstract class
```

Pure Virtual Functions: Summary

- Pure virtual functions are useful because they make explicit the abstractness of a class
- Tell both the user and the compiler how it was intended to be used
- Note : It is a good idea to keep the common code as close as possible to the root of you hierarchy

Summary – Cont'd

- It is still possible to provide definition of a pure virtual function in the base class
- The class still remains abstract and functions must be redefined in the derived classes, but a common piece of code can be kept there to facilitate reuse
- In this case, they can not be declared inline

```
class Shape { //Abstract
public :
  virtual void draw() = 0;
};

// OK, not defined inline
void Shape::draw(){
 cout << "Shape" << endl;
}
```

```
class Rectangle : public Shape
{
  public :
   void draw(){
     Shape::draw(); //Reuse
     cout <<"Rectangle"<< endl;
}
```

Take Home Message

- Polymorphism is built upon class inheritance
- It allows different versions of a function to be called in the same manner, with some overhead
- Polymorphism is implemented with virtual functions, and requires pass-by-reference