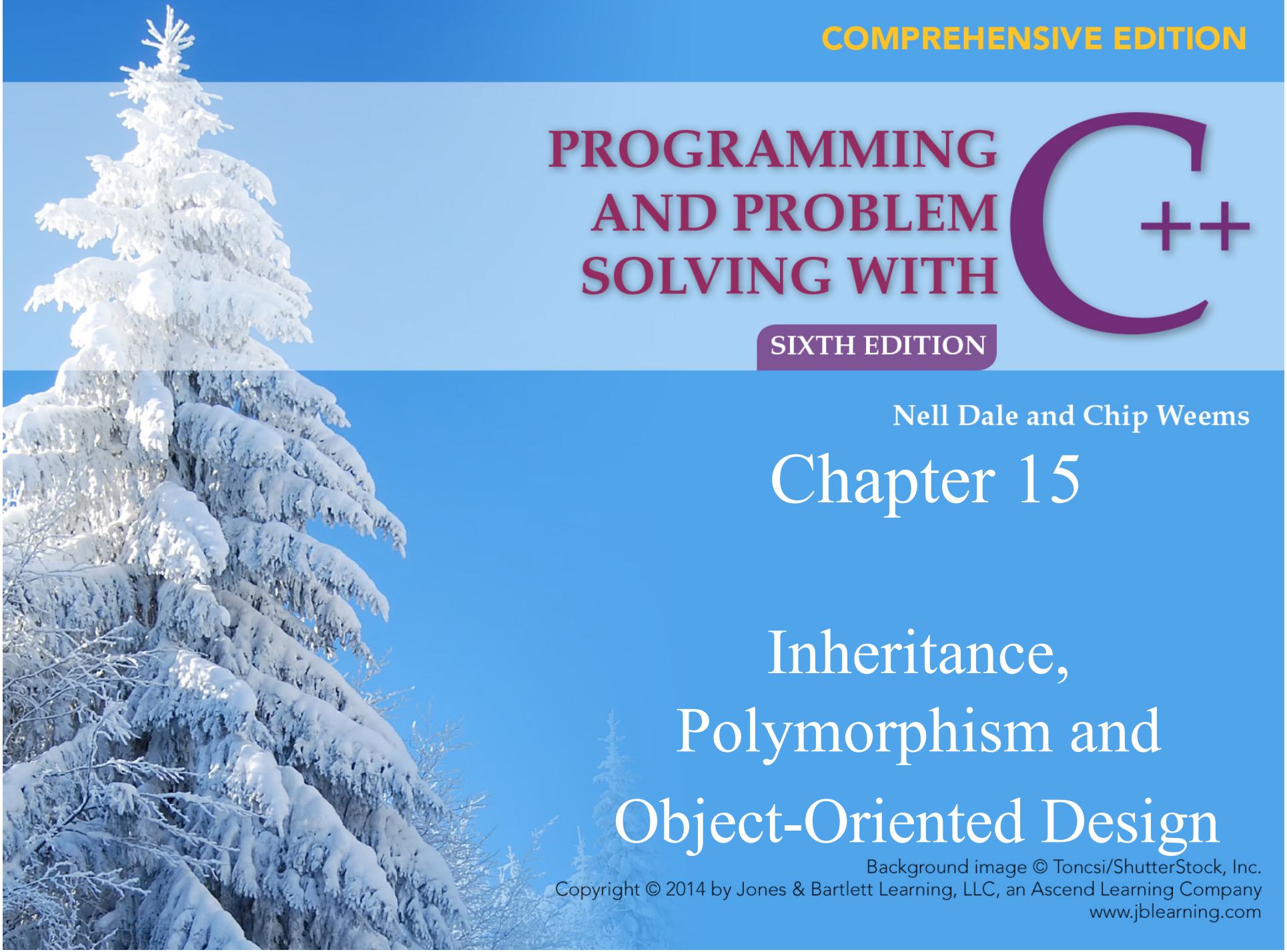


COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 15

Inheritance, Polymorphism and Object-Oriented Design

Chapter 15 Topics

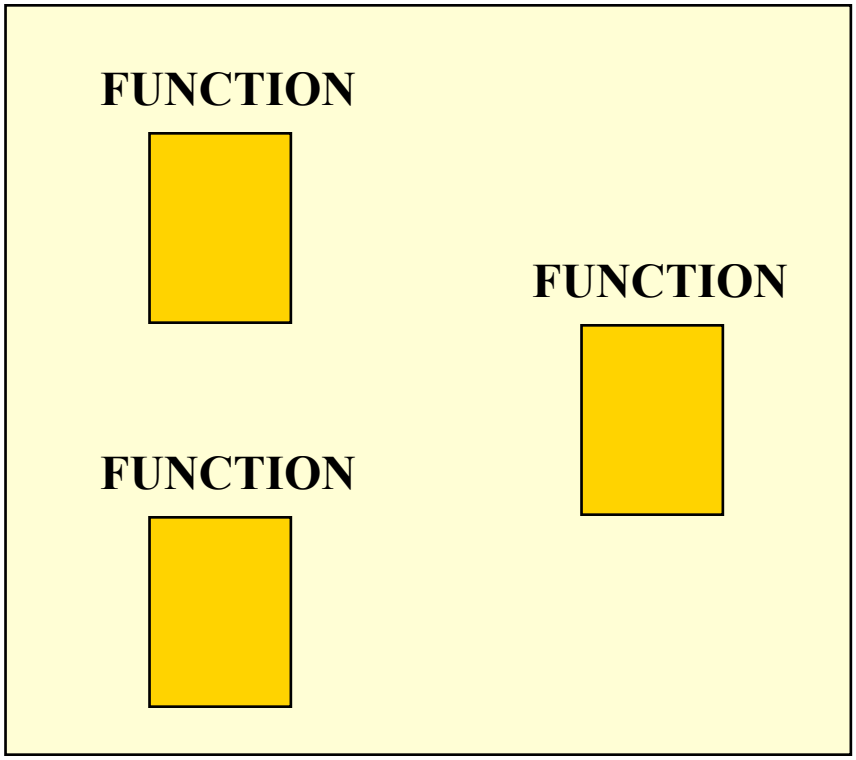
- **Structured Programming vs. Object-Oriented Programming**
- **Using Inheritance to Create a New C++ `class` Type**
- **Using Composition (Containment) to Create a New C++ `class` Type**

Chapter 15 Topics

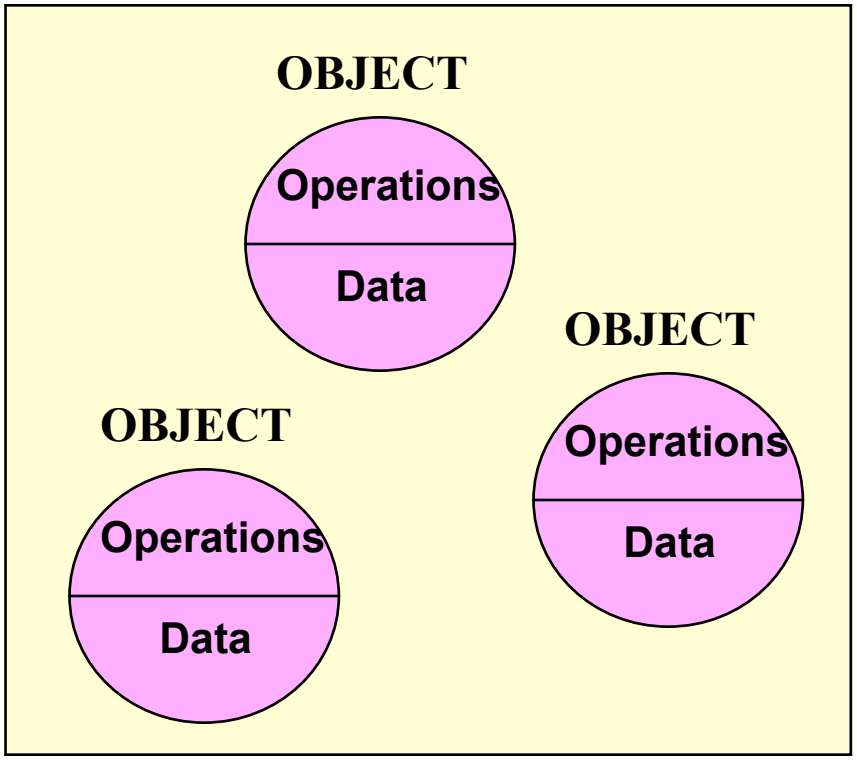
- **Static vs. Dynamic Binding of Operations to Objects**
- **Virtual Member Functions**

Two Programming Paradigms

Structural (Procedural) PROGRAM



Object-Oriented PROGRAM

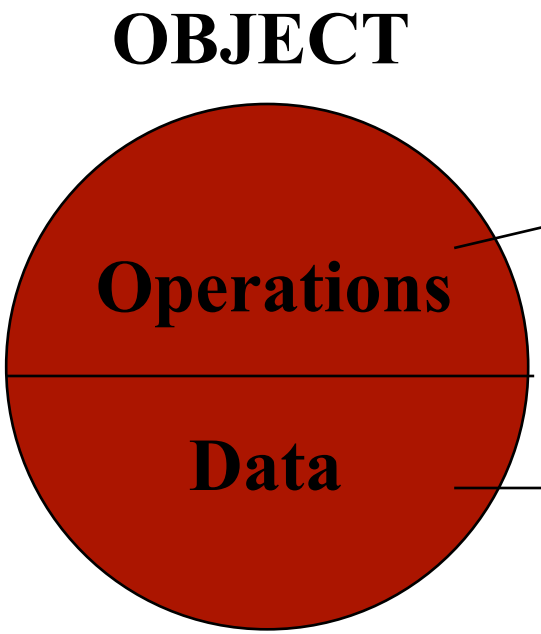


Object-Oriented Programming Language Features

1. Data abstraction
2. Inheritance of properties
3. Dynamic binding of operations to objects

OOP Terms	C++ Equivalents
Object	Class object or class instance
Instance variable	Private data member
Method	Public member function
Message passing	Function call (to a public member function)

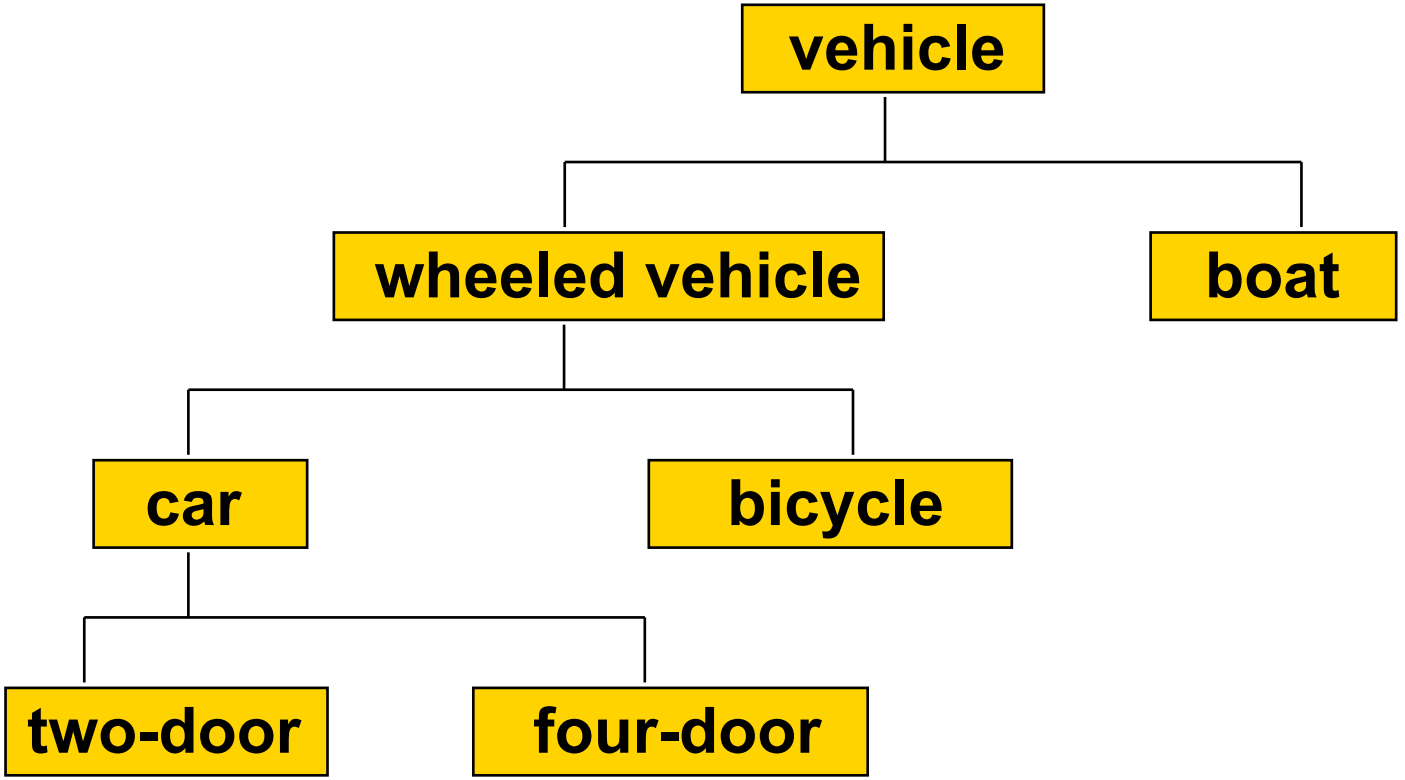
What is an object?



set of methods (public member functions)

internal state (values of private data members)

Inheritance Hierarchy Among Vehicles



Every car is a wheeled vehicle.

Inheritance

- **Inheritance** is a mechanism by which one class acquires (inherits) the properties (both data and operations) of another class

- The class being inherited from is the **Base Class** (Superclass)

Inheritance, cont...

- The class that inherits is the **Derived Class** (Subclass)
- The derived class is specialized by adding properties specific to it

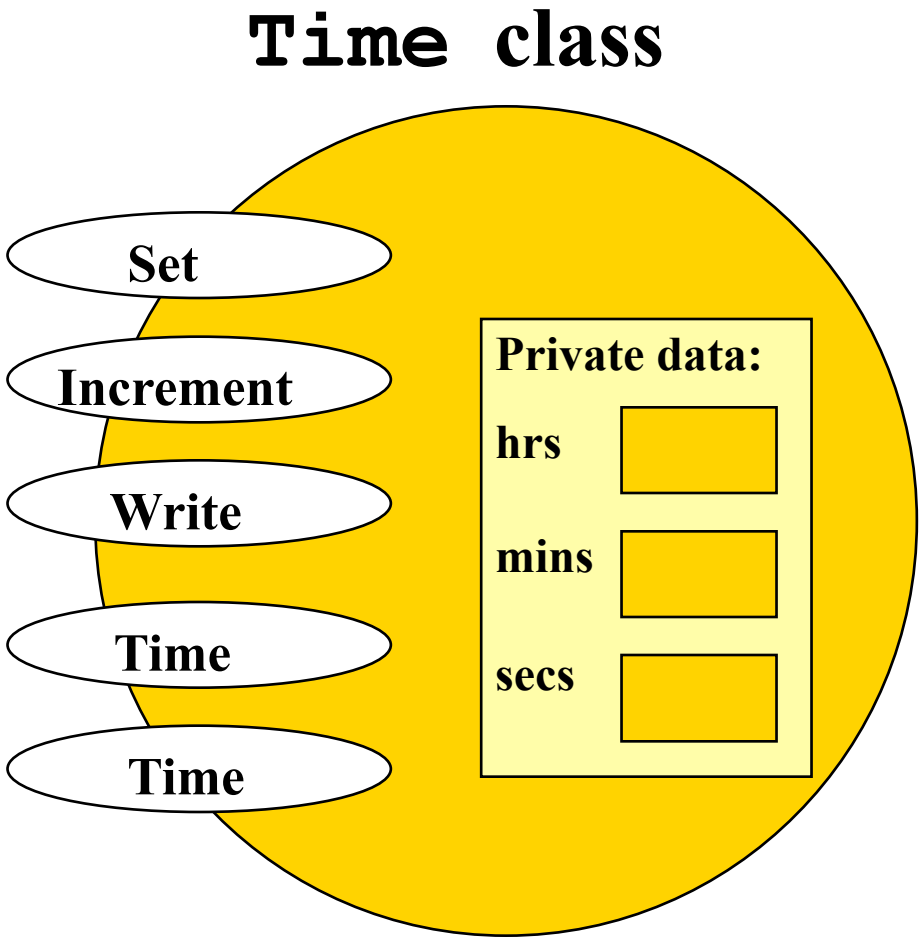
class Time Specification

```
// Specification file ("time.h")

class  Time
{

public:
     void Set ( int hours, int minutes, int seconds);
     void Increment ();
     void Write ()  const;
     Time ( int initHrs, int initMins, int initSecs);
     // Constructor
     Time ();              // Default constructor
```

class Time Specification

```
private:

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

Class Interface Diagram



Using Inheritance to Add Features

```
// Specification file (“exttime.h”)
#include   “time.h”
enum ZoneType{EST, CST, MST, PST, EDT, CDT, MDT, PDT};

class ExtTime : public Time // Time is the base class
```

Using Inheritance to Add Features

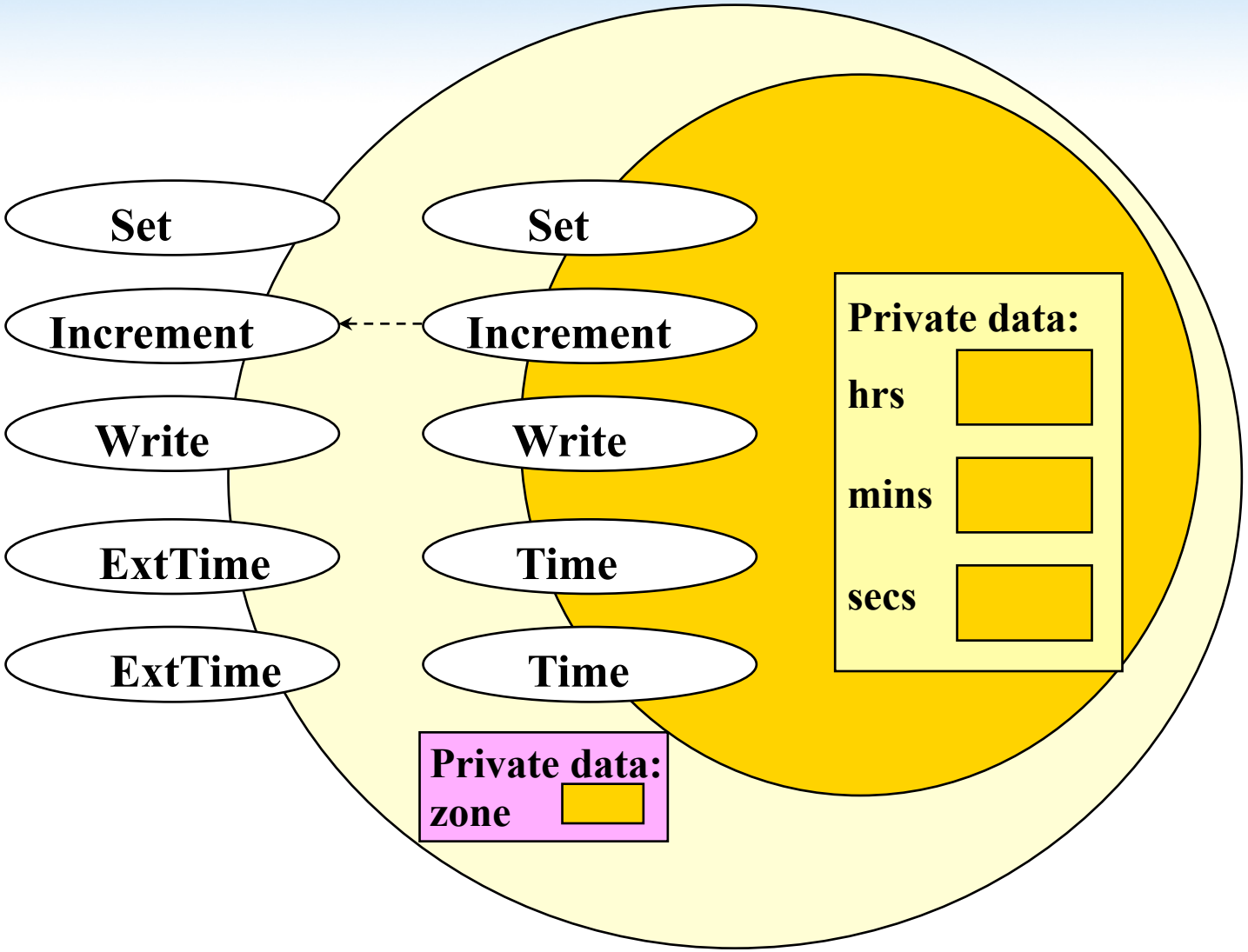
```
{
public:
    void Set(int hours, int minutes, int seconds,
             ZoneType  timeZone);
    void Write ()  const;
    ExtTime ( int  initHrs,  int  initMins,
              int  initSecs, ZoneType  initZone);
    ExtTime ();
private:
     ZoneType  zone; // Additional data member
};
```

`class ExtTime:public Time`

- **Says class Time is a public base class of the derived class ExtTime**
- **As a result, all public members of Time (except constructors) are also public members of ExtTime**
- **In this example, new constructors are provided, new data member zone added, and member functions Set and Write overridden**

Class Interface Diagram

`ExtTime` class



Client Code Using `ExtTime`

```
#include  "exttime.h"
          .
          .
          .
ExtTime  thisTime ( 8, 35, 0, PST);
ExtTime  thatTime; // Default constructor called

thatTime.Write(); // Outputs 00:00:00  EST
cout  <<  endl;
```

Client Code Using `ExtTime`

```
thatTime.Set (16, 49, 23, CDT);
thatTime.Write(); // Outputs 16:49:23  CDT
cout  << endl;

thisTime.Increment ();
thisTime.Increment ();
thisTime.Write (); // Outputs 08:35:02  PST
cout  << endl;
```

Constructor Rules for Derived Classes

- At run time, the base class constructor is implicitly called first, before the body of the derived class's constructor executes
- If the base class constructor requires parameters, they must be passed by the derived class's constructor

Implementation of `ExtTime` Default Constructor

```
ExtTime::ExtTime ( )
// Default  Constructor
// Postcondition:

//     hrs == 0   &&   mins == 0   &&  secs == 0

//      (via an implicit call to base class default
//       constructor)

//      &&  zone == EST
```

Implementation of `ExtTime` Default Constructor

```
{
    zone  =  EST;
}
```

Implementation of Another `ExtTime` Class Constructor

```
ExtTime::ExtTime  ( /* in */ int initHrs,
                    /* in */ int initMins,
                    /* in */ int initSecs,
                    /* in */ ZoneType initZone)

          : Time (initHrs, initMins, initSecs)
```

Implementation of Another `ExtTime` Class Constructor

```
// Constructor initializer
// Pre: 0 <= initHrs <= 23 && 0 <= initMins <= 59
//      0 <= initSecs <= 59 && initZone is
//      assigned
// Post: zone == initZone && Time set by base
//      class constructor

{
    zone  = initZone;
}
```

Implementation of `ExtTime::Set` function

```
void  ExtTime::Set  ( /* in */ int initHrs,
                      /* in */ int initMins,
                      /* in */ int initSecs,
                      /* in */ ZoneType initZone)

// Pre: 0 <= initHrs <= 23 && 0 <= initMins <= 59
//      0 <= initSecs <= 59 && initZone is assigned
// Post: zone == timeZone  && Time set by base
//      class function
```

Implementation of `ExtTime::Set` function

```
{
    Time::Set (hours, minutes, seconds);
    zone  = timeZone;
}
```

Implementation of ExtTime::Write Function

```
void ExtTime::Write (  )  const

//  Postcondition:
// Time has been output in form HH:MM:SS  ZZZ
//      where  ZZZ is the time zone abbreviation
```

Implementation of ExtTime::Write Function

```
{
    static string zoneString[8] = { “EST”, CST”,
        MST”, “PST”, “EDT”, “CDT”, “MDT”, “PDT” };
    Time::Write ();
    cout  << ‘ ‘ <<  zoneString[zone];
}
```

Responsibilities

- **Responsibilities** are operations implemented as C++ functions
- **Action responsibilities** are operations that perform an action
- **Knowledge responsibilities** are operations that return the state of private data variables

What responsibilities are Missing?

The Time class needs

- **int Hours()**
- **int Minutes()**
- **int Seconds()**

The ExtTime class needs

- **ZoneType zone()**

Composition (or Containment)

- **Composition (containment) is a mechanism by which the internal data (the state) of one class includes an object of another class**

An Entry Object

```
#include "Time.h"
#include "Name.h"
#include <string>
 class Entry
{
public:
     string NameStr() const;
     // Returns a string made up of first
     // name and last name
     string TimeStr() const;
     // Returns a string made up of hour,
     // colon, minutes
```

An Entry Object

```
Entry();
    // Default constructor
    Entry(……)
    // Parameterized constructor

private:
    Name name;
    Time time;
}
```

Implementation of ExtTime::Write Function

```
void ExtTime::Write (  )  const

//  Postcondition:
//Time has been output in form HH:MM:SS  ZZZ
//      where  ZZZ is the time zone abbreviation
```

Implementation of ExtTime::Write Function

```
{
    static string zoneString[8] = { “EST”, CST”,
        MST”, “PST”, “EDT”, “CDT”, “MDT”, “PDT” };
    Time::Write ();
    cout  << ‘ ‘ <<  zoneString[zone];
}
```

Order in Which Constructors are Executed

Given a class X:

- **If X is a derived class its base class constructor is executed first**
- **Next, constructors for member objects, if any, are executed (using their own default constructors if none is specified)**
- **Finally, the body of X's constructor is executed**

In C++ . . .

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 descendant class type**

Static Binding

- **Static binding** is the **compile-time determination** of which function to call for a particular object based on the type of the formal parameter(s)
- When pass-by-value is used, static binding occurs

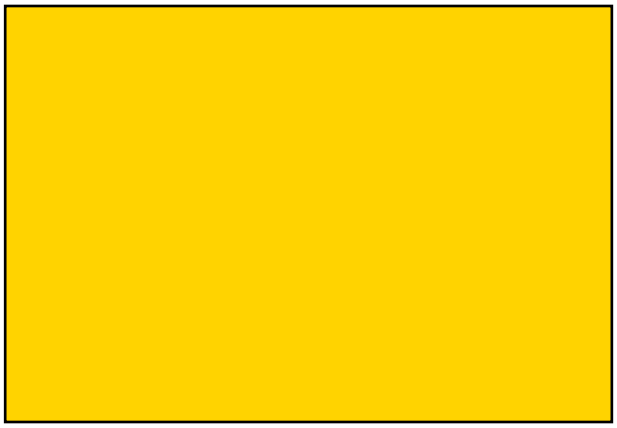
Static Binding Is Based on Formal Parameter Type

```
void  Print ( /* in */ Time someTime)
{
 cout  <<  "Time is ";
 someTime.Write ( );
 cout  <<  endl;
}
```

CLIENT CODE

```
Time startTime(8, 30, 0);
ExtTime endTime(10,45,0,CST);
Print ( startTime);
   Time is 08:30:00
Print ( endTime);
   Time is 10:45:00
```

OUTPUT



Dynamic Binding

- **Dynamic binding** is the **run-time determination** of which function to call for a particular object of a descendant 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

Virtual Member Function

```
// Specification file ( "time.h")

class  Time
{

public:

     .  .  .

     virtual void Write () const;
     // Forces dynamic binding

     .  .  .
```

Virtual Member Function

```
private:
    int  hrs;
    int  mins;
    int  secs;
};
```

Dynamic binding requires pass-by-reference

```
void Print  ( /* in */ Time& someTime)
{
 cout  <<  "Time is  ";
 someTime.Write ( );
 cout  <<  endl;
}
```

CLIENT CODE

```
Time startTime ( 8, 30, 0);
ExtTime endTime (10,45,0,CST);
Print ( startTime);
Print ( endTime);
Time is 10:45:00 CST
```

OUTPUT

```
Time is 08:30:00
```

Using virtual functions in C++

- Dynamic binding requires **pass-by-reference** when passing a class object to a function

- In the declaration for a virtual function, the word **virtual appears only in the base class**

Using virtual functions in C++

- If a base class declares a virtual function, it must implement that function, even if the body is empty
- A derived class is not required to re-implement a virtual function; if it does not, the base class version is used

Object-Oriented Design

- Identify the Objects and Operations
- Determine the relationship among objects
- Design and Implement the driver

Implementation of the Design

- Choose a suitable data representation
  - Built-in data type
  - Existing ADT
  - Create a new ADT
- Create algorithms for the abstract operations
  - Top-down design is often the best technique to create algorithms for the function tasks

Case Study

- Beginning of the Appointment Calendar Program
- Class Entry composed of a Name class and a Time class
- Current Time class represents active time (seconds with Increment)
- Need a static Time class for time in the appointment calendar

Is inheritance appropriate?