Inheritance

CPE 212

Derived Classes

- **Inheritance**
  - A mechanism that allows one to reuse existing debugged code by allowing a class to acquire properties, the data and operations, of another class
- **Multiple Inheritance**
  - Inheritance of properties from multiple classes

Terminology

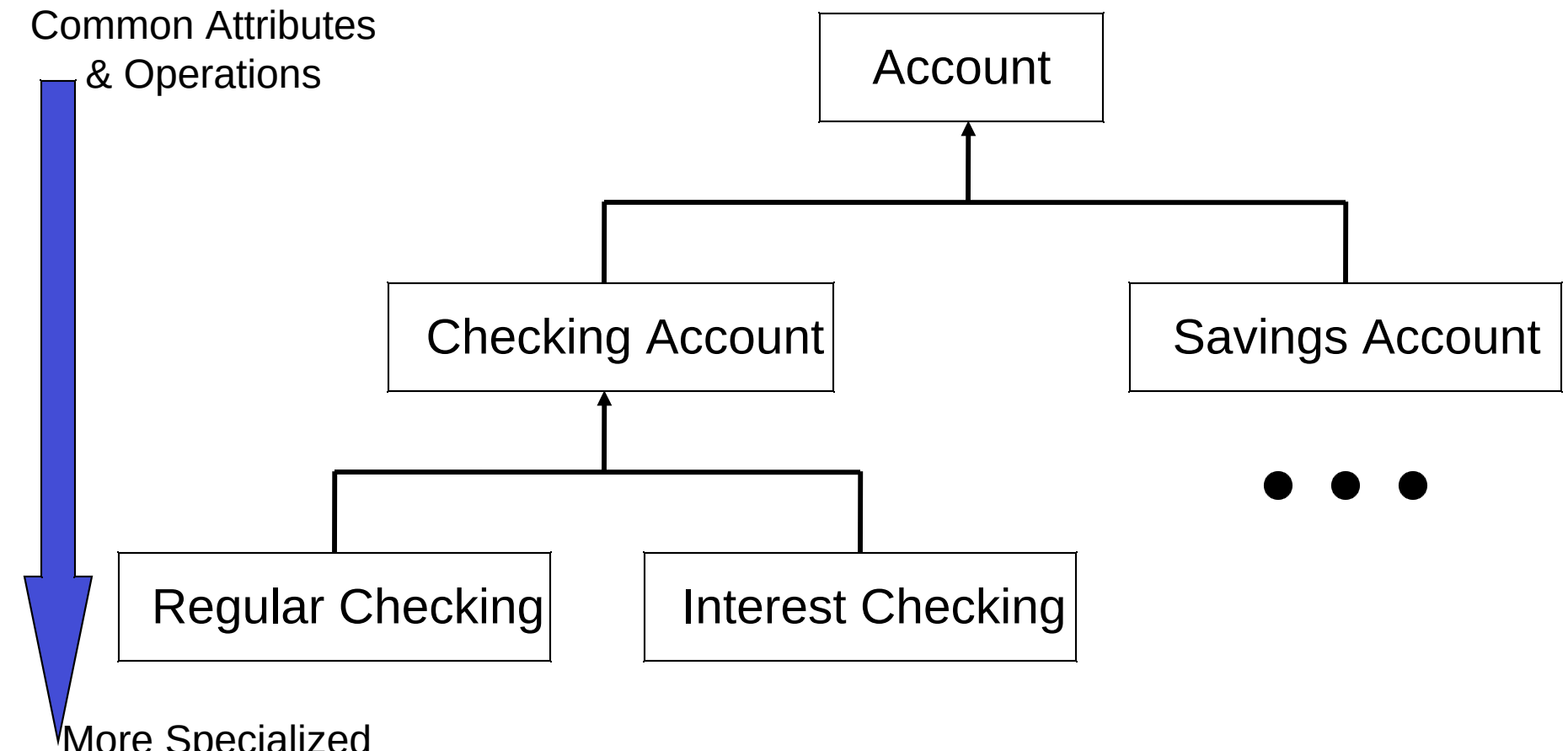
- **Base Class** (or parent class or superclass)
  - The class being inherited from
- **Derived Class** (or child class or subclass)
  - The class that inherits the properties of the Base Class

Class Interfaces

- Public
  - Interface to the rest of the world
- Private
  - Interface to member functions of the class
- Protected
  - Interface to derived classes

Big C++ , Horstmann and Budd

Inheritance Concept



Unified Modeling Language (UML) Class Diagram

"is-a" Relationship

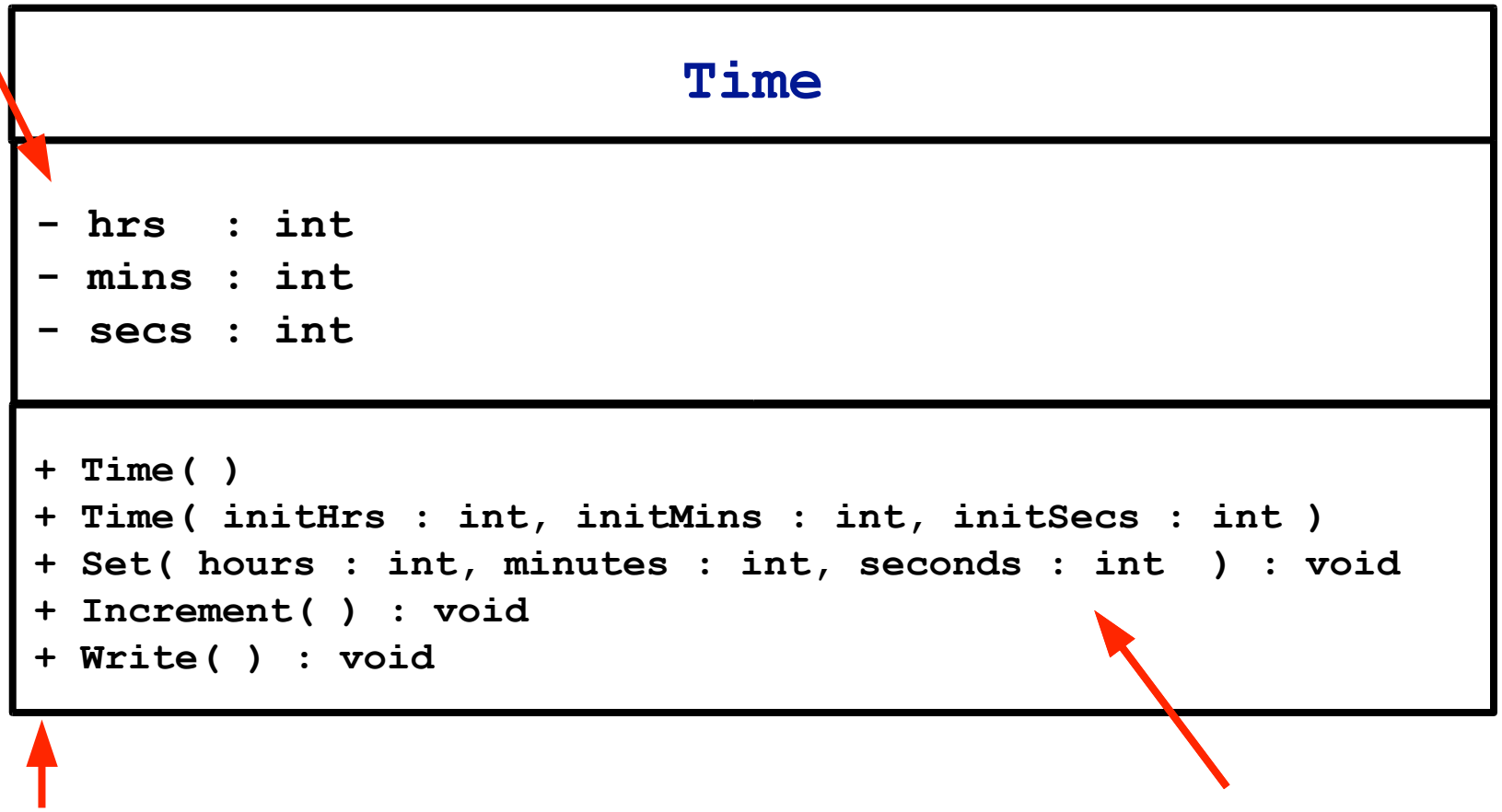
- Inheritance creates an "is-a" relationship between an object of a derived class and the parent class
- Derived class object inherits attributes and methods from parent but it also includes additional attributes or methods
- Examples: **BaseClass** **DerivedClass**
  - A **Student** **is a** kind of **Person**
  - A **RegularChecking** object **is a** kind of **CheckingAccount**

Time Class Example - time.h

```
//********* time.h Standard CPE212 Class Header Here ************
//********* Note: simplified for this example ******************
class Time
{
  private:
    int hrs;    // Valid range 0-23 inclusive
    int mins;   // Valid range 0-59 inclusive
    int secs;   // Valid range 0-59 inclusive


  public:

    Time();     // Default constructor, Time is 0:0:0
    Time(int initHrs, int initMins, int initSecs); // Constructor, initHrs:initMins:initSecs

    void Set(int hours, int minutes, int seconds ); // Set time
    void Increment();  // Add one second and wrap if necessary

    void Write() const;   // Output time in HH:MM:SS form
};
```

Class Diagram Entry for **Time** Class

- Private



+ Public

Not C++ Syntax

 # Protected

Time Class Example - Time.cpp

```
//********* time.cpp Standard CPE212 Implementation Header Here ********
#include <iostream>
#include "time.h"

using namespace std;

// Private members of Time class declared in Time.h:   int hrs, mins, secs;

Time::Time()                          // Default constructor
{
    hrs = 0;
    mins = 0;
    secs = 0;
} // End Time::Time()

Time::Time(int initHrs, int initMins, int initSecs )   // Constructor
{
    hrs = initHrs;
    mins = initMins;
    secs = initSecs;
} // End Time::Time(…)

void Time::Set(int hours, int minutes, int seconds )   // Set to
    hours:minutes:seconds
{
    hrs = hours;
    mins = minutes;
    secs = seconds;
} // End Time::Set(…)
```

Time Class Example - Time.cpp

```
void Time::Increment()          // Add one second and wrap around if necessary
{
    secs++;
    if (secs > 59)
    {
        secs = 0;
        mins++;
        if (mins > 59)
        {
            mins = 0;
            hrs++;
            if (hrs > 23)
                hrs = 0;
        }
    }
} // End Time::Increment()

void Time::Write() const        // Write state to stdout
{
    if (hrs < 10)
        cout << '0';
    cout << hrs << ':';
    if (mins < 10)
        cout << '0';
    cout << mins << ':';
    if (secs < 10)
        cout << '0';
    cout << secs;
} // End Time::Write()
```

Inheritance Example

Base Class

Time

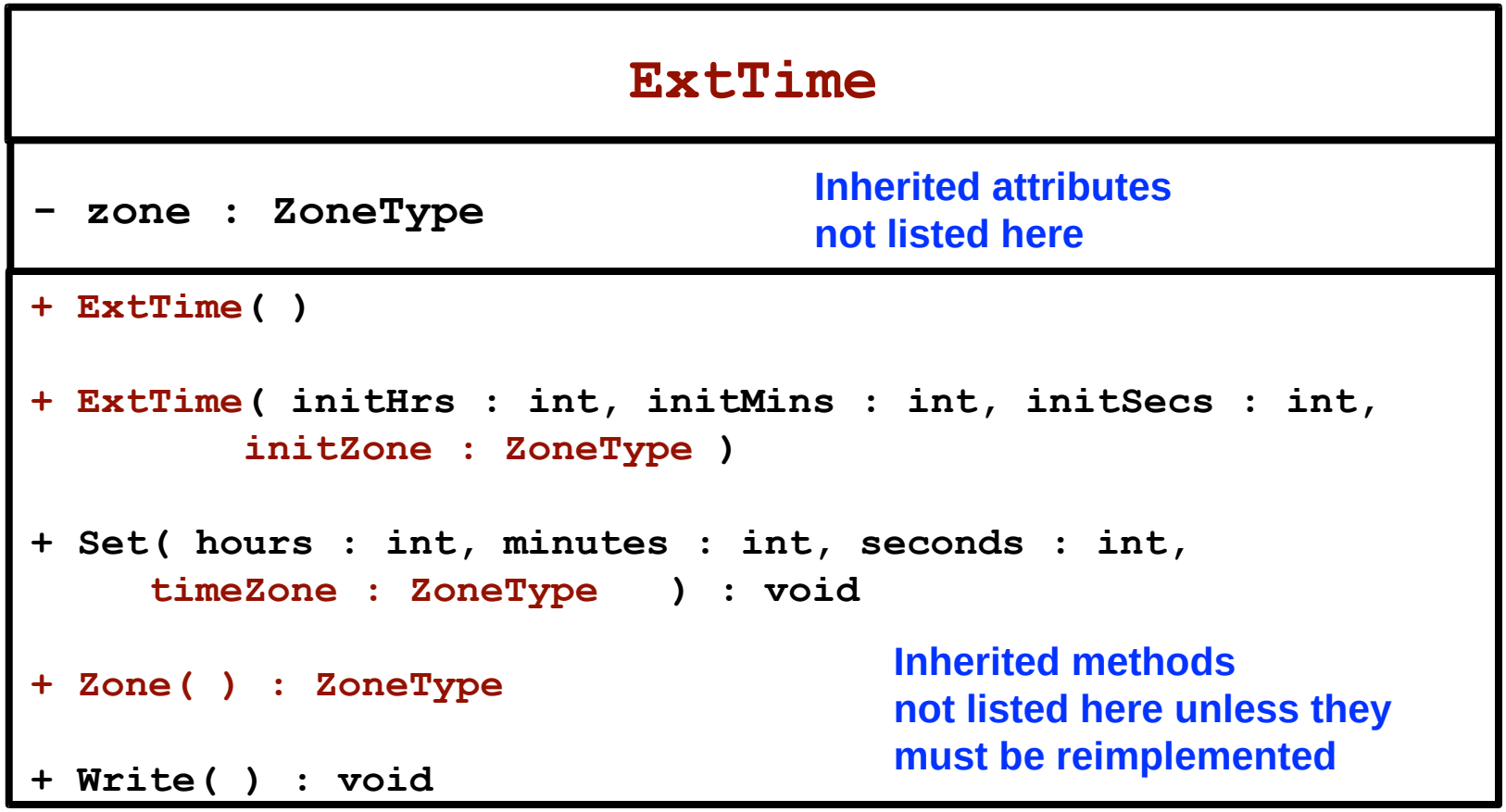
```
private:
    int hrs;
    int mins;
    int secs;

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

Derived Class

```
Inherit these private attributes from Time
    int hrs, mins, secs;

Add a new private attribute
    ZoneType zone;    // Use the enumerated type ZoneType

Add new methods:
    ExtTime();        // Default constructor 0:0:0 EST
    ExtTime(int initHrs, int initMins, int initSecs, ZoneType initZone);
    ZoneType Zone() const;    // Returns timezone

Reimplement these methods:
    void Set(int hours, int minutes, int seconds, ZoneType timeZone);
    void Write() const;    // Must also print out timezone

Inherit this method unmodified:
    void Increment();
```

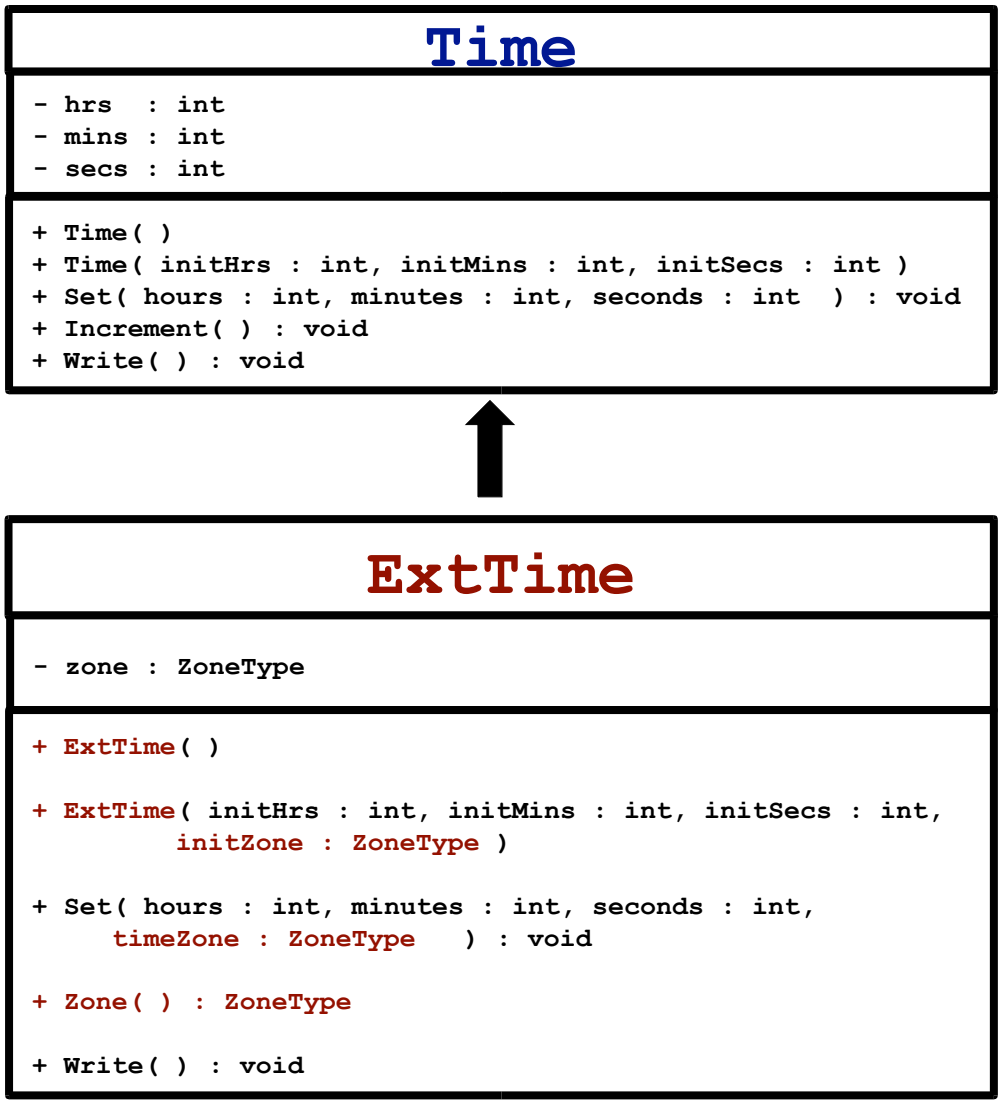
Constructors are not inherited!!

Class Diagram Entry for **ExtTime**



Overall Class Diagram

Base Class



Derived Class

Important Observations - 1

- Private members of the **Base Class** are present within objects of the **Derived Class** but are ***NOT*** directly accessible within the **Derived Class** code
  - There is a **Base Class** object nested within each **Derived Class** object
  - This embedded **Base Class** object contains all inherited members
  - Any private inherited members must be accessed via the public member functions inherited from the **Base Class**

Important Observations - 2

- **protected** members of the **Base Class** are directly accessible within objects of the **Derived Class**

- Use **protected** members only with careful consideration since it may make your code more difficult to maintain

Constructors and Destructors -1

- ***Base Class Constructor*** will be called before the ***Derived Class Constructor***

- ***Derived Class Destructor*** will be called before ***Base Class Destructor***

Constructors and Destructors - 2

- Syntax for Derived Class Constructor

```
DerivedClassName::DerivedClassName(parameter_list) :
                        BaseClassName(argument_list)
{
   // Derived Class Constructor Statements
}
```

- Omitting the Base Class results in execution of the default Base Class Constructor

ExtTime Class Specification - exttime.h

```
//********* exttime.h Standard CPE212 Class Header Here ************
#include "time.h"

enum ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT};    // Eight US time zones

class ExtTime : public Time   // "public" makes Time a public base class of ExtTime
// So all public members of Time (except constructors) are public members of ExtTime
{
  private:
    ZoneType  zone;       // Represents time zone


  public:
    ExtTime();       // Default constructor, Time is 0:0:0 EST

    ExtTime(int initHrs, int initMins, int initSecs, Zonetype initZone); // Constructor
                    // Time is initHrs:initMins:initSecs  initZone

    void Set(int hours, int minutes, int seconds , ZoneType timeZone); // Set time & zone

    void Write() const;  // Output time in HH:MM:SS  TimeZone form

    ZoneType  Zone() const;   // Returns time zone
};
```

ExtTime Class Implementation - exttime.cpp

```
//********* exttime.cpp Standard CPE212 Implementation Header Here ********
#include <iostream>
#include "exttime.h"
using namespace std;
// Private members of ExtTime class declared in exttime.h: ZoneType zone;

ExtTime::ExtTime() // Default constructor
// base class default constructor implicitly called before derived class constructor
{
    zone = EST;
} // ExtTime::ExtTime()

ExtTime::ExtTime( int      initHrs,
                  int      initMins,
                  int      initSecs,
                  ZoneType initZone )  : Time(initHrs, initMins, initSecs)
// Parameterized constructor with a constructor initializer
{
    zone = initZone;
} // ExtTime::ExtTime(…)

void ExtTime::Set( int      hours,
                   int      minutes,
                   int      seconds,
                   ZoneType timeZone )    // ExtTime::Set()
{
    Time::Set(hours, minutes, seconds);
    zone = timeZone;
} // End ExtTime::Set()
```

ExtTime Class Implementation - exttime.cpp

```
ZoneType ExtTime::Zone() const      // Zone()
{
    return zone;
} // End ExtTime::Zone()


void ExtTime::Write() const    // Write()
{
    static string zoneString[8] =
    {
        "EST", "CST", "MST", "PST", "EDT", "CDT", "MDT", "PDT"
    };

    Time::Write();
    cout << ' ' << zoneString[zone];
} // End ExtTime::Write(…)
```

Questions

- What are the private members of ExtTime?

More Questions

- What happens when the following declarations appear in a client of ExtTime?
  - ExtTime someTime1;
  - ExtTime someTime2(8, 35, 0, PST);

ExtTime Class Client - exttimedriver.cpp

```
//********* exttimedriver.cpp Header Here ***************
// >>> Incomplete ExtTimeDriver Program <<<
#include <iostream>
#include "exttime.h"

using namespace std;

int main()
{
    ExtTime time1(5,30,0,CDT); // Test parameterized constructor
    ExtTime time2;         // Test default constructor

    cout << "time1: ";
    time1.Write();         // Writes time1: 05:30:00 CDT to stdout
    cout << "time2: ";
    time2.Write();         // Writes time1: 00:00:00 EST to stdout

    time1.Increment();
    cout << "New time1: ";
    time1.Write();         // Writes New time1: 05:30:01 CDT to stdout

    time2.Set(23,59,59,PST);
    cout << "New time2: ";
    time2.Write();         // Writes New time2: 23:59:59 PST to stdout

    return 0;
} // End main()
```

Questions about Client

- What would happen if the client needed access to both the Time class and the ExtTime class?
  - If one added the following to client.cpp, what would happen?

```
#include “time.h”
#include “exttime.h”
```

Avoiding Multiple Header File Inclusion

```
// Rewrite time.h as follows using Include Guards

#ifndef TIME_H
#define TIME_H

class Time
{
  …
};
#endif

// If preprocessor identifier TIME_H is not already
// defined, then define it and pass the Time class
// declaration to the compiler
//
// If a subsequent #include "time.h" is encountered, the test
//  ifndef TIME_H will fail and the class declaration is
// skipped.
```

virtual Functions - 1

- **Static Binding**
  - Compile-time determination of which function to call for a particular object
- **Dynamic-Binding**
  - Run-time determination of which function to call for a particular object

virtual Functions - 2

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

startTime.Write();
cout << endl;
endTime.Write();
cout << endl;
```

Static binding here ensures that the correct version of the Write function is called -- either **`Time::Write()`** or **`ExtTime::Write()`**

virtual Functions - 3

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

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

- The output for endTime is incomplete since the time zone has been omitted

virtual Functions - 4

- **Slicing Problem**
  - You wish to pass an object of a derived class by value to a function
  - The function parameter data type is that of the base class -- not the derived class
  - Since the base class does not have the extra members that were added to the derived class, only the subset of derived class members shared with the base class are passed
  - For the previous example, the time zone attribute of endTime is sliced off and not handed to the function Print
  - Also a problem with member-by-member copy with =

- ***Passing by reference*** eliminates the slicing problem since the value is not copied, but static binding of Write to Time::Write() means the time zone is still not output

virtual Functions - 5

- **Polymorphism**
  - The ability to determine which of several operations with the same name is appropriate
- **Polymorphic Operation**
  - An operation that has multiple meanings depending upon the data type of the object to which it is bound at run time

- We can use a virtual function to make Write() a polymorphic operation and ensure that the correct version of the function is invoked
  - Using a virtual Time::Write() guarantees dynamic-binding
  - The decision of which version of Write() to invoke is delayed until runtime
  - At runtime, the data type of the argument determines the version of Write() invoked

virtual Functions - 6

```
class Time
{
   public:
      virtual void Write() const;
};

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

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

virtual Functions - 7

- If **virtual** appears in the function prototype, it does not appear in the heading of the function definition

- By declaring a member function of the **base class** as **virtual**, any redefined versions of the function in derived classes are also **virtual**

virtual Functions - 8

- Suppose one uses a reference parameter called **someParam** to pass an object to a function which will invoke a member function called **SomeMethod()** on that object:

```
someParam.SomeMethod();
```

- Case 1: SomeMethod() is not virtual
  - Data type of parameter determines method invoked
- Case 2: SomeMethod() is virtual
  - Data type of ***argument*** determines method invoked

pure virtual Functions and Abstract Classes - 1

- An ***abstract class*** may created to model an abstract concept that cannot be instantiated as a object
- The ***abstract class*** is used as a ***base class***
- An ***abstract class*** will include one or more ***pure virtual*** methods which have no function body
- An attempt to create an object of the abstract class type will result in a compile-time error
- To create objects of the ***derived class*** type, the ***derived class*** that inherits from an ***abstract class*** **MUST** ***reimplement all pure virtual methods***

pure virtual Functions and Abstract Classes - 2

```
class  Character_Device
{
   public:
      virtual  void  help( );              // Virtual function
      virtual  void  open( )  = 0;         // Note the pure virtual function body
      virtual  void  close( )  = 0;        // Note the pure virtual function body
      ...
};

Character_Device      d;                   // Compile-time error
```

Abstract Class Example - 1

```
class Base
{
  public:
    Base()  { x = 0; }

    void b1()  { x++; }

    virtual void b2()  { x--; }   // Virtual method
                                  // May be reimplemented in derived class

    virtual void b3() = 0;        // Pure virtual without implementation
                                  // Must be reimplemented in derived class to allow creation of objects of derived class type

    virtual void b4() = 0;        // Pure virtual with implementation
                                  // Must be reimplemented in derived class to allow creation of objects of derived class type
                                  // Derived class code can invoke base class implementation of this method on an object
                                  // of the derived class type

  private:
    int x;
};

void Base::b4()
{
    cout << "x = " << x << endl;
}

int main( )
{
    Base* b;              // Compiles

    Base   bb;            // Error: cannot declare variable ‘bb’ to be of abstract type ‘Base'
}
```

Abstract Class Example - 2

```
class Derived  :   public  Base
{
  public:
    Derived() { y = 1; }

    void d1()  { y++; }

    void b3()  { y = 0; }       // Required to reimplement this pure virtual method to create objects of type Derived

    void b4()  { }              // Required to reimplement this pure virtual method to create objects of type Derived

  private:
    int y;
};

int main( )
{
    Derived* d;             // Compiles

    Derived dd;             // Compiles because b3( ) and b4( ) have been reimplemented in Derived
}
```

inline Functions

- inline Function in C++
  - These functions are treated differently by the compiler
  - They may improve or degrade system performance depending upon the system
  - **Use of an inline function in this course will result in no credit (0) on that programming assignment**

Function Definitions within a Class Declaration

- In this course, you are not allowed to embed your function definition (the code that performs the function's task) within the class declaration
  - **Use of this technique this course will result in no credit (0) on that programming assignment**

Friend vs Member Function

- **Member Function**
  - Accessed using member selector operator
- **Friend Function**
  - A normal non-member function which has access to private members of the class
- Which one should you use?
  - Use a member function if the task involves only one object
  - Use a non-member friend function if the task involves two objects
- We will see examples of this later in the course