CPE 212 - Fundamentals of Software Engineering

•••

Inheritance

Reminder:

Project 01 due this Friday by 11:59pm

Objective:

Overview of the use of inheritance with C++ classes

Outline

- Defining Inheritance
- Relationships
- Example
- virtual Functions
- Abstract Classes

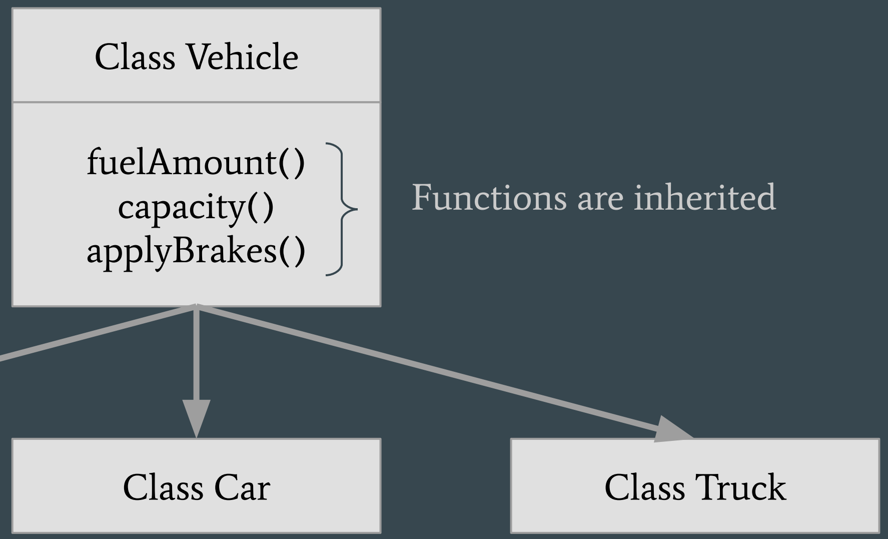
Derived Classes

- 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
- Derived Class
  - The class that inherits properties from another class
  - Also called a Sub Class or Child Class
- Base Class
  - The class whose properties are inherited by the derived class
  - Also called a Super Class

Class Interfaces

- Public
  - Interface to the everyone who uses the class
  - Accessible from anywhere outside the class but within the program
- Private
  - Interface to member functions of the class
  - Cannot be accessed, or even viewed, from outside the class
  - Only class and friend functions can access private members
- Protected
  - Interface to the derived classes
  - Similar to private members with the additional benefit of being able to be accessed from the derived class

Inheritance Concept



Class Vehicle

fuelAmount()
capacity()
applyBrakes()

Functions are inherited

Class Bus

Class Car

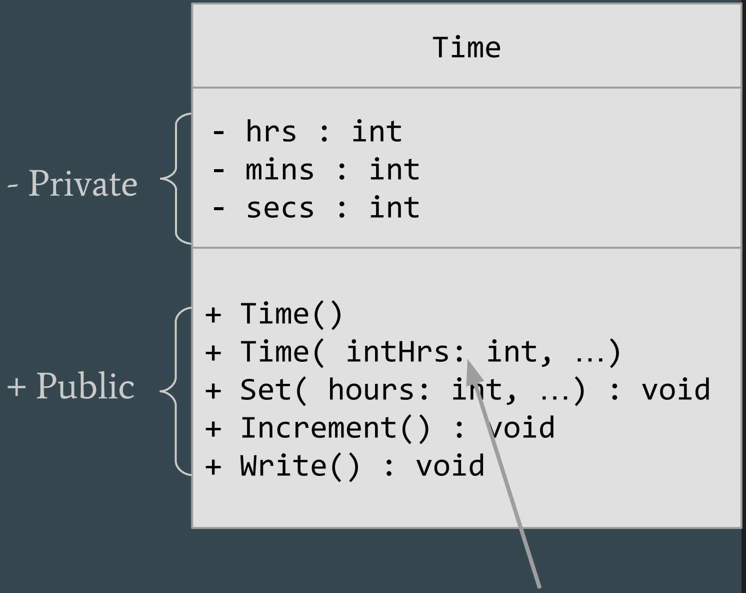
Class Truck

Unified Modeling Language (UML)
Class Diagram

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 Car is a kind of Vehicle
  - A SuperCar object is a kind of Car

Time Class - time.h



Time

- hrs : int
- mins : int
- secs : int

+ Time()
+ Time(intHrs: int, …)
+ Set(hours: int, …) : void
+ Increment() : void
+ Write() : void

- Private

+ Public

Not C++ Syntax

# Protected

```
//********* 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
   // Constructor, initHrs:initMins:initSecs
   Time(int initHrs, int initMins, int 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
};
```

Time Class - 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;
```

Time Class - 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

- Private variables inherited

```
int hrs, mins, secs;
```

- Add new private attribute
- Add new methods
- Reimplement any methods necessary
- Create new constructors

Important!

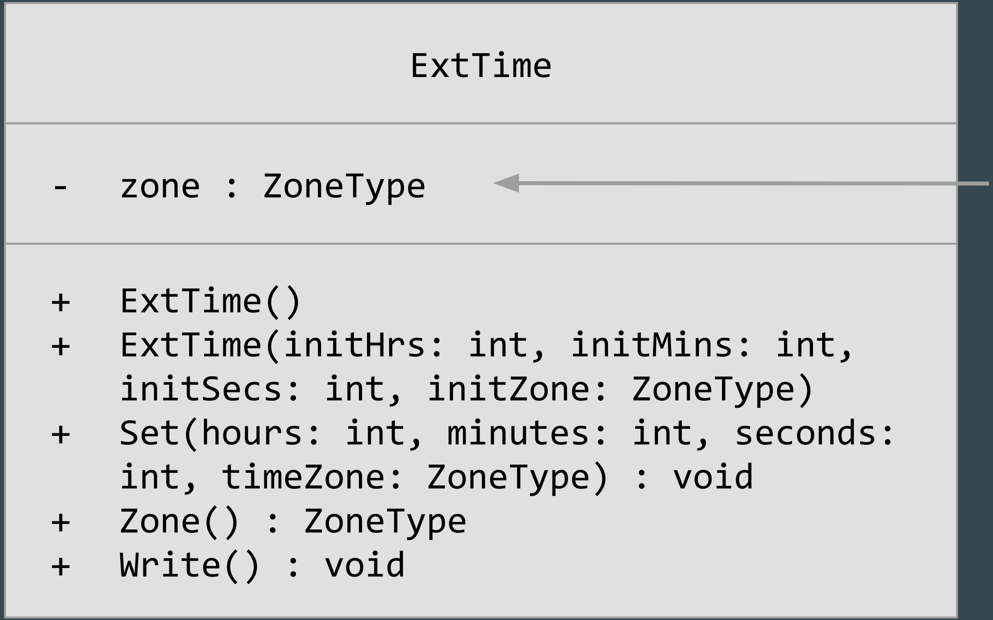
Constructors are not inherited!!

Inherited from Base

```
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;
```

```
// 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();
```

Class Diagram for ExtTime

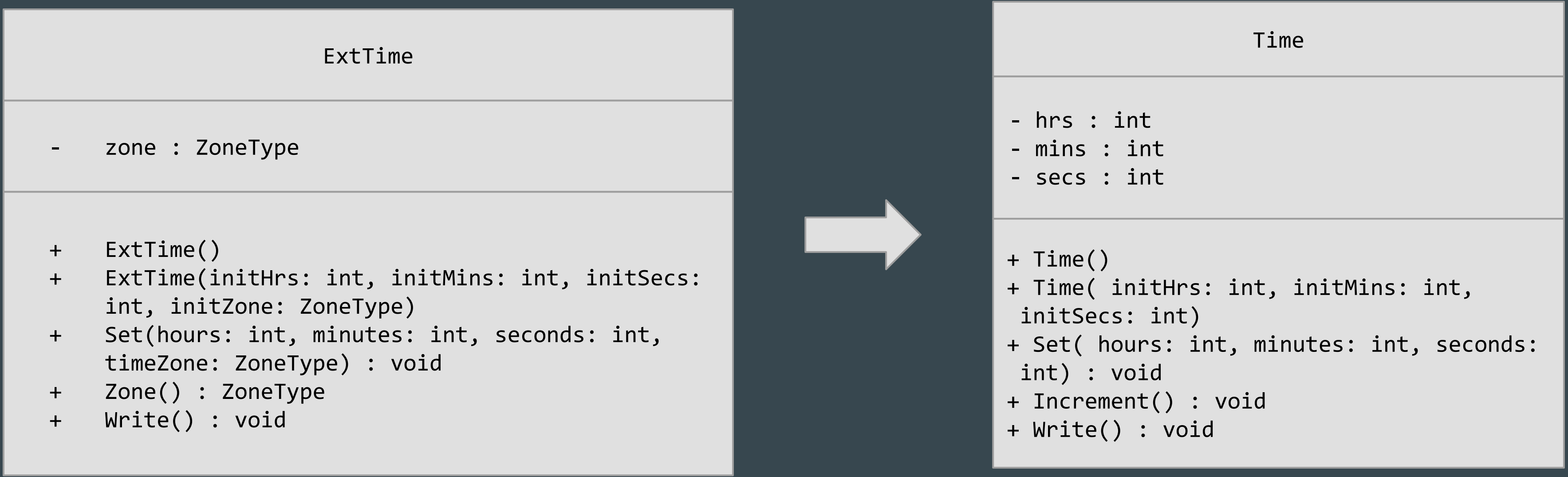


```
ExtTime

-   zone : ZoneType

+   ExtTime()
+   ExtTime(initHrs: int, initMins: int,
    initSecs: int, initZone: ZoneType)
+   Set(hours: int, minutes: int, seconds:
    int, timeZone: ZoneType) : void
+   Zone() : ZoneType
+   Write() : void
```

Inherited attributes are not listed

Inherited methods are not listed unless they must be reimplemented

Time Class Diagram



```
ExtTime

-   zone : ZoneType

+   ExtTime()
+   ExtTime(initHrs: int, initMins: int, initSecs:
    int, initZone: ZoneType)
+   Set(hours: int, minutes: int, seconds: int,
    timeZone: ZoneType) : void
+   Zone() : ZoneType
+   Write() : void
```

```
Time

- hrs : int
- mins : int
- secs : int

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

Modes of Inheritance

- Public
  - If sub-class is derived from a public base class then the public members of the base class will become public
  - Protected members will become protected
- Protected
  - If sub-class is dervied from a protected base class then both the public and protected members will be protected in the derived class
- Private
  - If sub-class is derived from a private base class then both public and protected members of the base class will become private

Base class Member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible	Not accessible	Not accessible

Source: https://www.geeksforgeeks.org/inheritance-in-c/

Constructors and Destructors

- Base Class Constructor will be called BEFORE the Derived Class Constructor
- Derived Class Destructor will be called BEFORE the Base Class Destructor
- Omitting the Base Class results in execution of the default Base Class Constructor

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

Class Access

```
// C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object
class A {
public:
   int x;
protected:
   int y;
private:
   int z;
};
 class B : public A {
   // x is public
   // y is protected
   // z is not accessible from B
};
 class C : protected A {
   // x is protected
   // y is protected
   // z is not accessible from C
};

 class D : private A { // 'private' is default for classes
   // x is private
   // y is private
   // z is not accessible from D
};
```

Source: https://www.geeksforgeeks.org/inheritance-in-c/

ExtTime Class
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

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

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

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

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

ExtTime Class
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
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 the ExtTime Class?

- What happens when the following declarations appear in a client of ExtTime?

```
ExtTime  someTime1;
ExtTime  someTime2(8, 35, 0, PST);
```

ExtTime Class Driver exttimedriver.cpp

```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

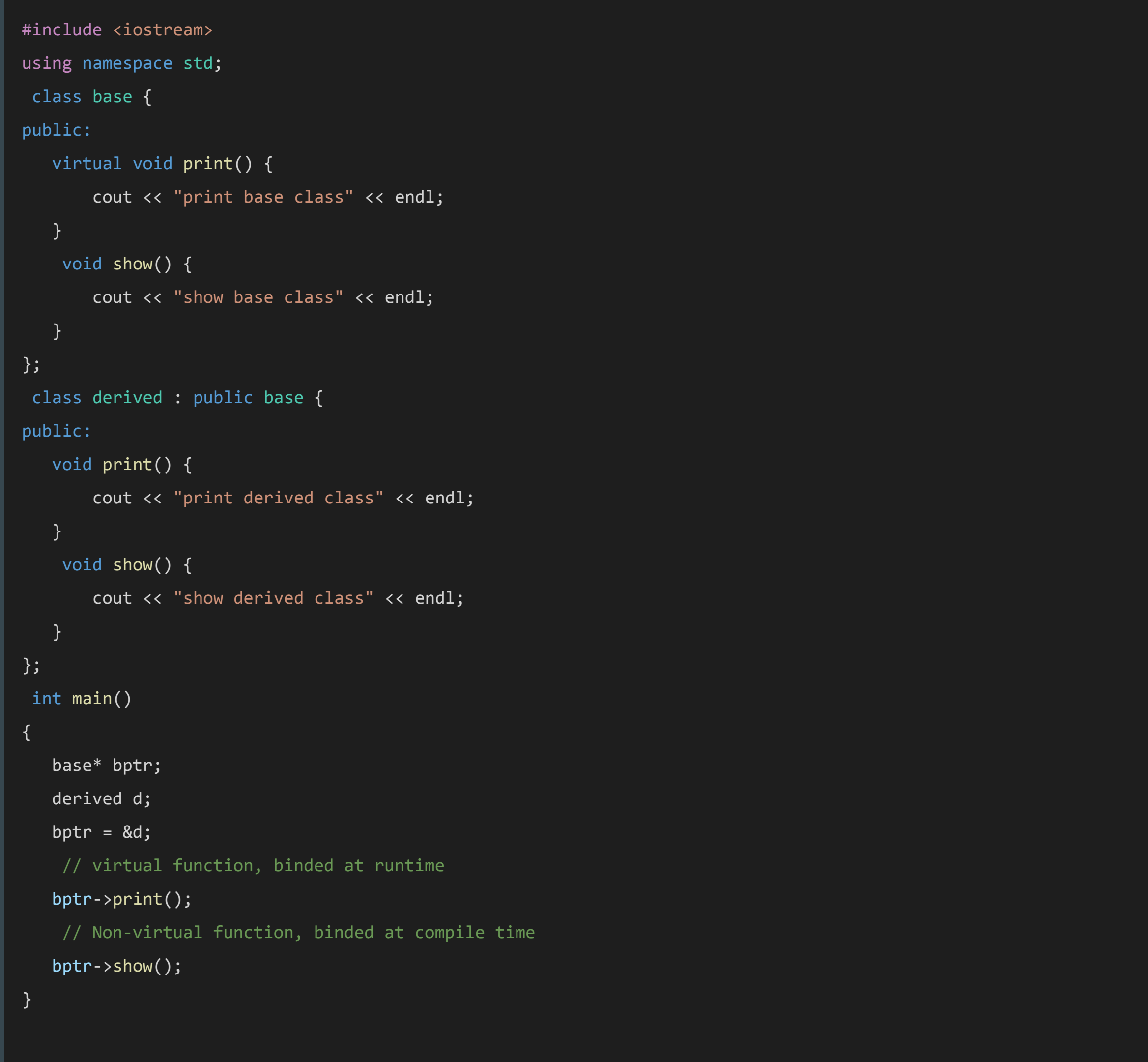
- What would happen if the client needed access to both the Time class and the ExtTime class?

- What happens when you add the following to client.cpp?

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

virtual Functions

- A member function which is declared within a base class and is re-defined by a derived class.
- Bindings
    - Compile-time (early binding) is possible through the use of a pointer to the base class type
    - Run-time (late binding) is done through the use of the content of the pointer rather than the type of pointer



```
#include <iostream>
using namespace std;
 class base {
public:
   virtual void print() {
       cout << "print base class" << endl;
   }
    void show() {
       cout << "show base class" << endl;
   }
};
 class derived : public base {
public:
   void print() {
       cout << "print derived class" << endl;
   }
    void show() {
       cout << "show derived class" << endl;
   }
};
 int main()
{
   base* bptr;
   derived d;
   bptr = &d;
    // virtual function, binded at runtime
   bptr->print();
    // Non-virtual function, binded at compile time
   bptr->show();
}
```

Time Class Example

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

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

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

Question

- What is the output?

```
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);
```

Slicing

- 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

Polymorphism

- The ability to determine which of several operations with the same name is appropriate
- Polymorphic Operation is 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 Function Implementation

- 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`

- 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

- 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 may 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

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

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

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
  - Not considered class members
  - Outside the scope of the class
- More examples to come later in the course

```cpp
// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
   friend void ChangePrivate( Point & );
public:
   Point( void ) : m_i(0) {}
   void PrintPrivate( void ){cout << m_i << endl; }

private:
   int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
  Point sPoint;
  sPoint.PrintPrivate();
  ChangePrivate(sPoint);
  sPoint.PrintPrivate();
// Output: 0
//         1
}
```

https://docs.microsoft.com/en-us/cpp/cpp/friend-cpp?view=vs-2019