EE2331 Data Structures and Algorithms:

Object-Oriented C++ Review

Outline

- Classes and Objects
- Functions/Operators Overloading
- Friend Function
- NULL Pointer, Void Pointer and Function Pointer
- Dynamic Memory Allocation
- Templates
- Inheritance
- Virtual Function
- Static/Dynamic Binding

Classes

C++ Classes

- C++ supports data abstraction providing a built-in structured data types known as *struct* and *class*
- A *class* is similar to a struct but is nearly always designed so that its components (class members) include both data and functions that manipulate that data

```
struct  Time {
    int     hrs;
    int     mins;
    int     secs;
};
```

public by default

```
class Time {
    private:
          int       hrs;
          int       mins;
          int       secs;
    public:
          void      set( int, int, int );
          void      increment();
          void      write() const;
          bool      equal( Time ) const;
          bool      lessThan( Time ) const;
};
```

Class Declaration

- Class variables (referred to as class objects or class instances) are created by using ordinary variable declarations

```
    Time startTime;
    Time endTime;
```

- Any software that declares and manipulates ***Time*** objects is called a client of the class:
  - public: this constitutes the public interface, clients can access these class members directly
  - private: this is private information that is inaccessible to clients. If client code attempts to access a private item, the compiler will signal an error
  - const: this declares a constant member function which is not allowed to change any class members. So such use of *const* only makes sense, and is hence only allowed, for member functions.

Class Objects and Members

- Always remember that a class is a type, not a data object -- a pattern from which you create many objects of that type

- The declarations

  ```
  Time  time1;
  Time  time2;
  ```

  create two objects of the *Time* class: time1 and time2. Each object has its own copies of *hrs*, *mins*, and *secs*, the private data members of the class

- The C++ compiler generates just one physical copy of member functions. However we still say that there are two objects of the *Time* class, and each object has eight members

Built-In Operations on Classes

- With class types you can:
  - declare as many objects as you like
  - pass class objects as parameters to functions and return them as function values
  - declare arrays of class objects

- Most of the built-in operations **DO NOT** apply to classes:
  - you cannot use the + operator to add two *Time* objects
  - you cannot use the == operator to compare two *Time* objects for equality

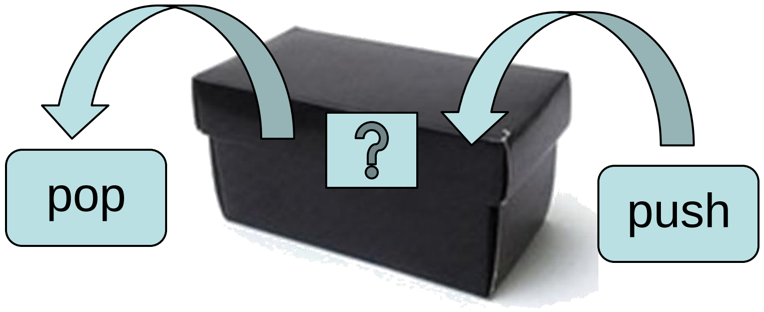
- Two important built-in operations on class objects:
  - member selection (.)
  - assignment (=)

Design by Contract

Ex3_1

- An approach for designing software
- Software designers should define formal, precise and verifiable interface specifications for software components. These specifications are referred to as "contracts"
- Interface provides an abstraction layer for interaction between client and underlying data representation
- Programs that work on the interface are unaffected by the changes of implementation, giving freedom to programmer to postpone decisions on implementation details

- The example contains three parts:
  - Specification File (time.h)
  - Implementation File (time.cpp)
  - Main Function



Class Specifications

```
// Precondition:
// 0 <= hours <= 23 && 0 <= minutes <= 59 && 0 <= seconds <= 59
// Postcondition:
// time is set according to the incoming parameters
// NOTE: This function MUST be called prior to any of the other member functions
void set (int hours, int minutes, int seconds);


// Precondition:
// The set function has been invoked at least once
// Postcondition:
// time has been advanced by one second, with 23:59:59 wrapping around to 0:0:0
void increment ();


// Precondition:
// The set function has been invoked at least once
// Postcondition:
// Time has been output in the form HH:MM:SS
void write () const;
```

Class Specifications

```
// Precondition:
// The set function has been invoked at least once for both this time and
// otherTime
// Postcondition:
// Function value == TRUE, if this time equals otherTime, otherwise Function
// value == FALSE
bool equal (Time otherTime) const;

// Precondition:
// The set function has been invoked at least once for both this time and
// otherTime && This time and otherTime represent times in the same day
// Postcondition:
// Function value == TRUE, if this time is earlier in the day than otherTime,
// otherwise Function value == FALSE
bool lessThan (Time otherTime) const;
```

Guaranteed Initialization with Class Constructors

Ex3_2

- The ***Time*** class we have been discussing has a weakness. It depends on the client to invoke the set function before calling any other member function. If the client fails to invoke the set function first, the precondition of the member functions is false and the function implementation is broken
- We should not rely on the user to initialize the data!
- C++ provides a mechanism, called a class constructor, to guarantee the initialization of a class object
  - A constructor function has an unusual name: the name of the class itself. Let’s add two class constructors to Time class
  - The first constructor has three int parameters which are used to initialize the private data when the object is created (see example)
  - The second constructor is parameterless and initializes the time to some default value, such as 0:0:0. This parameterless constructor is known in C++ as a default constructor (see example)

Guaranteed Initialization with Class Constructors

```
class Time
{
        private:
                int     hrs;
                int     mins;
                int     secs;
        public:
                void    set(int, int, int);
                void    increment();
                void    write() const;
                Boolean equal(Time) const;
                Boolean lessThan(Time) const;
                Time(int, int, int);     // Constructor
                Time();                  // Constructor
};
```

Guaranteed Initialization with Class Constructors

```
// Constructor
// Precondition:
// 0 <= initHrs <= 23 && 0 <= initMins <= 59 && 0 <= initSecs <= 59
// Postcondition:
// hrs == initHrs && mins == initMins && secs == initSecs
Time::Time (int initHrs, int initMins, int initSecs )
{
         hrs = initHrs;
         mins = initMins;
         secs = initSecs;
}

// Default constructor
// Postcondition:
// hrs == 0 && mins == 0 && secs == 0
Time::Time ( )
{
         hrs = 0;
         mins = 0;
         secs = 0;
}
```

Initialization list

- Initialization list can be used to initialize member variables
- Guaranteed initialization even **before** constructor body is executed (recommended!)

```
Time::Time( ) : hrs(0), mins(0), secs(0) {
    // do something
}

Time::Time (int initHrs, int initMins, int initSecs) :
          hrs(initHrs), mins(initMins), secs(initSecs) {
    // do something
}
```

Same as:

```
Time::Time( ) {
    hrs = mins = secs = 0;
    // do something
}

Time::Time (int initHrs, int initMins, int initSecs) {
    hrs = initHrs; mins = initMins; secs = initSecs;
    // do something
}
```

Invoking a Constructor

- A constructor is never invoked using dot (.) notation. A constructor is automatically invoked whenever a class object is created:

  ```
  Time lectureTime(10, 30, 0);
  ```

  the first constructor is automatically invoked initializing the private data to 10:30:0

  ```
  Time startTime;
  ```

  the default constructor is implicitly invoked initializing startTime's private data to the time 0:0:0

Guidelines for Using Constructors

- A constructor cannot return a function value
- A class may provide several constructors. The choice depends on the number and data types of the parameters to the constructor
- Parameters to a constructor are placed immediately after the name of the class object

- If a class object is declared without a parameter list, then:
  - if the class does not have constructors at all, private data members are created in an uninitialized state
  - if the class has constructors, then the default constructor is invoked if there is one. If there is no default constructor, a syntax error occurs
- If a class has at least one constructor, and an array of class objects is declared then one of the constructors must be the default constructor. This constructor is invoked for each element of the array.

```
SomeClass arr[10];    // invoke the parameterless default constructor
```

Class Destructors

- Just as a constructor is implicitly invoked when a class object is created, a destructor (a special member function) is implicitly invoked when a class object is destroyed e.g. *when control leaves the block in which a local object is declared*.
- A class destructor is named the same as a constructor except that the first character is a tilde(~):

```
class SomeClass    {
    public:
        SomeClass();  // Constructor
        ~SomeClass(); // Destructor
    private:
        int* arr;
};
```

```
// Destructor
// Postcondition
// Object pointed to by this is no
// longer on memory
SomeClass::~SomeClass() {
         cout << “Goodbye World”;
}
```

- Should be used for releasing all resources before coming out of the program, e.g, closing files, releasing memories etc.

C++ *this* Pointer

- Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, **this** may be used to refer to the invoking object (self).
- When you call a non-static member function, it is converted internally:

```
void Time::set(int hours, …)          // void Time::set(const Time* this, int hours, …)
{
    this->hrs = hours;
    ……
}
```

The first parameter will be a constant pointer (i.e. cannot make it point to something else)

```
Time time;
time.set(5, 30, 20);                  // set(&time, 5, 30, 20);
```

Passing the address of the time object

Function Overloading

- In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```
int compute (int a, int b) {
  return (a*b);
}

double compute (double a, double b) {
  return (a*b);
}

int main () {
  int x=5, y=2;
  double n=5.5, m=2.5;
  cout << compute (x, y) << endl;
  cout << compute (n, m) << endl;
  return 0;
}
```

Operator Overloading

Ex3_3

- C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, e.g. ***struct*** and ***class***.
- Operators are overloaded by means of operator functions, which are regular functions with special names: their name begins by the operator keyword followed by the *operator sign* that is overloaded. The syntax is:

```
type operator sign (parameters) { /*... body ...*/ }
```

```
// overload operator to perform "lessThan" function
bool Time::operator < (const Time& otherTime) {
    return (hrs < otherTime.hrs ||
        hrs == otherTime.hrs && mins < otherTime.mins ||
        hrs == otherTime.hrs && mins == otherTime.mins
        && secs < otherTime.secs);
}
```

Friend Function

- In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.
- *Friends* are functions or classes declared with the friend keyword.
- A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword ***friend***:

```
friend ostream& operator << (ostream& os, Time& time);
```

Include this header in the Time class

```
ostream& operator<<(ostream& os, Time& time) {
    if ( time.hrs < 10 )
        os << '0';
    os << time.hrs << ':';
    if ( time.mins < 10 )
        os << '0';
    os << time.mins <<':';
    if ( time.secs < 10 )
        os << '0';
    os << time.secs;
    return os;
}
```

This function is a non-member function of the class Time.

Advanced Pointer and Dynamic Memory Allocation

Null Pointer

- In C++, there is only one literal pointer constant: the value 0. The pointer constant 0, called the null pointer, points to absolutely nothing.

  The statement `intPtr = 0;`
  stores the null pointer into intPtr.

- The constant name **NULL** can be used instead of 0 as a null pointer

  ```
  intPtr = NULL;
  ```

- The null pointer is used only as a special value that a program can test for:

  ```
  if (intPtr == NULL)
      DoSomething();
  ```

- If during program execution to the null pointer is dereferenced, it results in segmentation fault
- Null pointer should be used as the error return value of functions that return pointers

Void Pointer

Ex3_4

- The void type of pointer is a special type of pointer. In C++, void represents the absence of type. Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
- This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters. In exchange, they have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.
- One of its possible uses may be to pass generic parameters to a function. For example:

```
void swap (void *vp1, void *vp2, const int size) {
           char *buffer = (char *)malloc(sizeof(char)*size);
           memcpy(buffer, vp1, size);
           memcpy(vp1, vp2, size);
           memcpy(vp2, buffer, size);
           free(buffer);
}

int main() {
           int a = 10, b = 20;
           swap(&a, &b, sizeof(int));       // swap two integers

           double x = 11.1, y = 22.2;
           swap(&x, &y, sizeof(double));    // swap two doubles
}
```

Function Pointer

Ex3_5

- C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function. Pointers to functions are declared with the same syntax as a regular function declaration, except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```
int addition (int a, int b) {
    return (a+b);
}

int subtraction (int a, int b) {
    return (a-b);
}

int doSomething (int x, int y,
                 int (*callback)(int,int)) {
    int g;
    g = (*callback)(x,y);
    return g;
}
```

```
int main () {
    int m,n;
    int (*minus)(int,int) = subtraction;

    m = doSomething (7, 5, addition);
    n = doSomething (20, m, minus);
    cout << n;
    return 0;
}
```

Memory Allocation

- Static allocation from stack
  - Implicitly managed by system and released at function exit
- Dynamic allocation from heap
  - Explicitly acquired and released by user during runtime
- Dynamic data structure is a structure of memory blocks that change in size as memory is allocated and de-allocated from the heap
  - No upper limit for your data size nor a guarantee on maximum memory usage
- Risks with Using Dynamic Data
  - Dangling Pointer: A pointer that points to a variable that has been deallocated
  - Inaccessible Object (Orphan): A dynamic variable on the heap without any pointer pointing to it
  - Memory Leak: The loss of available memory space that occurs when dynamic data is allocated but never deallocated
- C++ operations: new and delete

New Operation

- The new operation returns the address of the acquired memory block. It has two forms, one for allocating a single variable and one for allocating an array:

Syntax:

```
new DataType
new DataType[IntExpression]
```

Example:

```
int* intPtr;
char* nameStr;
intPtr = new int;
nameStr = new char[6];
```

- Variables created by new are said to be on the free store (or heap), a region of memory set aside for dynamic variables. The new operator obtains a chunk of memory from the heap

Delete Operation

- Dynamic data (from new operation) can be destroyed at any time during the execution of a program using the delete operator

Syntax:

```
delete Pointer
delete[] Pointer
```

Example:

```
int* intPtr = new int;
char* nameStr = new char[6];
delete intPtr;
delete[] nameStr;
```

- The delete operator returns the obtained chunk of memory to the heap
- The deleted variable must be created with new or be a null pointer
- The lifetime of a dynamic variable starts at new and ends at delete
- Before exit remember to deallocate all object created!

Templates

What is C++ Template?

- One can pass “value” as parameter to a function/class for computations. How about passing “type” as parameter to them?
- Templates allow functions and classes to be parameterized so that the type of data being operated upon is received via a parameter**.**
- Templates provide a mean to reuse code — one template definition can be used to create multiple instances of a function or class, each operating on a different type of data.
- The template mechanism is important and powerful. It is used throughout the Standard Template Library (STL) to achieve genericity.
- There are two forms of template:
  - **Function template**
  - **Class template**

Function Templates

- Main reason for using functions: Make pieces of code reusable by encapsulating them within a function
- Example: Interchange the values of two int variables x and y. Instead of writing this snippet every time:

```
int temp = x;
x = y;
y = temp;
```

write a function:

```
/* Function to swap two int variables.
 Receive:   int variables first and second
 Pass back: first and second with values interchanged */

void swap(int& first, int& second){
  int temp = first;
  first = second;
  second = temp;
}
```

closely bind with the specific int type

Template Mechanism

- Declare a type parameter and use it in the function instead of a specific type. This requires a different kind of parameter list:

```
/* Swap template for exchanging the values of any two objects of the same type
Receive:   first and second, two objects of same type
Pass back: first and second with values swapped
Assumes:   Assignment (=) defined for type DataType */

template <typename DataType>      // declare type parameter
void swap(DataType& first, DataType& second)
{
      DataType temp;
      temp = first;
      first = second;
      second = temp;
}
```

generalized with the type parameter DataType

Template Instantiation

- `<typename DataType>` names `DataType` as a type parameter — value will be determined by the compiler from the type of the arguments passed when swap() is called. For example:

```
int main() {
       int        i1, i2;
       double     d1, d2;
       char       c1, c2;
       ...
       // Compiler generates definitions  of Swap() with DataType replaced
       swap(i1, i2);         //       by int
       swap(d1, d2);         //       by double
       swap(c1, c2);         //       by char
}
```

- A function template is only a pattern that describes how individual functions can be built from the given actual types. This process of constructing a function is called instantiation
- We instantiated `swap()` three times — with types int, double and char. In each instantiation, the type parameter is said to be bound to the actual type passed
- A template thus serves as a pattern for the definition of an unlimited number of instances

General Form of Template Declaration

- ***TypeParam*** is a type-parameter naming the "generic" type of value(s) on which the function operates.

function template

```
template <typename TypeParam> or template <class TypeParam>
// …… definition of the function using type TypeParam
```

```
template <typename TypeParam> or template <class TypeParam>
class SomeClass {
      // …… members of SomeClass ...
};
```

class template

- More than one type parameter may be specified:

```
template <typename TypeParam1,…, typename TypeParamn>
class SomeClass {
      // …… members of SomeClass ...
};
```

Instantiating Class Templates

- To use a class template in a program/function/library, one can instantiate it by using a declaration of the following form to pass ***Type*** as an argument to the class template definition:

```
ClassName<Type> object;
```

- Examples:

```
List<int>    intList;
List<double> doubleList;
```

- Compiler will generate two distinct definitions of List — two instances, one for int and one for double

Simple List Template

```
// SimpleList.h

#include <iostream>
using namespace std;

#ifndef ListT
#define ListT

const int LIST_CAPACITY = 100;

template <typename ListElement>
class List {
public:
        void print();
        void insert(ListElement item);
        int length();
        List();

private:
        ListElement myArray[LIST_CAPACITY];
        int size;
};
```

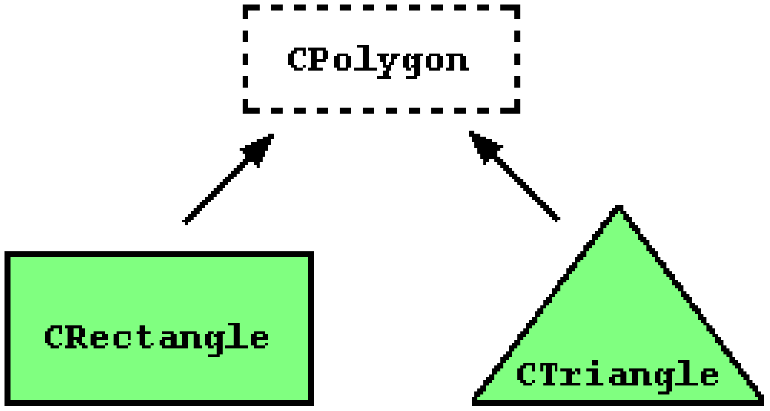
```
template <typename ListElement>
List<ListElement>::List(){
           size = 0;
}

template <typename ListElement>
void List<ListElement>::print(){
        for (int i=0; i<size; i++) {
                cout << myArray[i] << "  ";
        }
}

template <typename ListElement>
void List<ListElement>::insert(ListElement item){
           myArray[size++] = item;
}

template <typename ListElement>
int List<ListElement>::length(){
           return size;
}

#endif
```

Using Simple List Template

```
#include “SimpleList.h”

int main(){

    List<int> myList1;
    myList1.insert(10);
    myList1.insert(20);
    myList1.insert(30);

    myList1.print();
    cout << myList1.length();


    List<char> myList2;
    myList2.insert('A');
    myList2.insert('B');

    myList2.print();
    cout << myList2.length();
}
```

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a *base class* and a *derived class*: The *derived class* inherits the members of the *base class*, on top of which it can add its own members.
- For example, let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles. These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a height and a width (or base).



Inheritance Syntax

- The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

- The **public** access specifier may be replaced by any one of the other access specifiers (protected or private). This access specifier limits the most accessible level for the members inherited from the base class. The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.
- For example, if daughter were a class derived from mother that we defined as:

```
class Daughter: protected Mother;
```

- This would set **protected** as the less restrictive access level for the members of Daughter that it inherited from Mother. That is, all members that were public in Mother would become protected in Daughter.

Inheritance Example

Ex3_6

- The objects of the classes Rectangle and Triangle each contain members inherited from Polygon. These are: width, height and setValue.
- In principle, a publicly derived class inherits access to every member of a base class except:
  - its constructors and its destructor
  - its assignment operator members (operator=)
  - its friends
  - its private members

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void setValue (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };
```

```
class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.setValue (4,5);
  trgl.setValue (4,5);
  cout << rect.area() << '\n';   // 20
  cout << trgl.area() << '\n';   // 10
  return 0;
}
```

Pointers to Base Class

- One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. ***Polymorphism*** is the art of taking advantage of this simple but powerful and versatile feature.

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon* ppoly1 = &rect;
  Polygon* ppoly2 = &trgl;
  ppoly1->setValue (4,5);
  ppoly2->setValue (4,5);
  cout << ppoly1->area() << '\n';          // error
  cout << ppoly2->area() << '\n';          // error
  cout << rect.area() << '\n';   // 20
  cout << trgl.area() << ‘\n’;   // 10
  return 0;
}
```

- Because the type of ppoly1 and ppoly2 is pointer to Polygon, only the members inherited from Polygon can be accessed. That is why the program above accesses the **area()** members of both objects using **rect** and **trgl** directly, instead of the pointers; the pointers to the base class cannot access the area members.

Ex3_7

Virtual Function

- A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the **virtual** keyword:

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void setValue (int a, int b)
      { width=a; height=b;}
    virtual int area () { return 0; }
 };
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->setValue (4,5);
  ppoly2->setValue (4,5);
  ppoly3->setValue (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

Abstract Base Classes

- A virtual function in the base class will have an implementation.
- A virtual function without an implementation is called a pure virtual function. Notice that area() has no definition; this has been replaced by =0.

```
class Polygon {
  protected:
    int width, height;
  public:
    void setValue (int a, int b)
      { width=a; height=b;}
    virtual int area () =0;      // pure virtual
 };
```

- Pure virtual function in C++ corresponds to abstract method in Java. Classes that contain at least one *pure virtual function* are known as *abstract base classes*.
- Abstract base classes cannot be used to instantiate objects.
- But an *abstract base class* is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities.

```
Rectangle rect;
Polygon* ppoly1 = &rect;                  // valid
cout << ppoly1->area() << '\n';           // valid
```

Static vs Dynamic Binding of Function Calls

- In the previous example, the member function area() has been declared as virtual in the base class because it is later redefined (overridden) in each of the derived classes.
- Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

- Static binding (bind to base type in compile-time) is used
  - if the member function is invoked from object's value directly, **OR**
  - the member function in the base class is not virtual
- Dynamic binding (bind to derived type in runtime) is used
  - if the member function is invoked from object's reference or pointer, **AND**
  - the member function is defined virtual in the base class.