Advanced C++ Topics

• Inheritance
• Virtual methods and late binding
• Friend classes and methods
• Class templates
• Overloaded operators
• Iterators
Inheritance Revisited

• Inheritance is useful to
  – explicitly represent relationships among program components
  – reuse as much design and implementation effort as possible
  – avoid parallel implementations that are error prone since they are hard to keep synchronized

• Class hierarchies represent *shared* and *distinct* relationships between classes
  – derived (sub) class inherits base (super) class properties
    • all member data and functions except constructors and destructors
Inheritance – Basic Concepts

• Superclass or base class
  – a class from which another class is derived

• Subclass, derived class, or descendant class
  – a class that inherits all members of another class
  – can add new members to those it inherits
  – can redefine an inherited method of its base class, if the two methods have the same parameter declarations
Inheritance – Basic Concepts

• The base class’s public methods can be called by
  – An instance of the base class
  – An instance of the derived class
  – The derived class’s methods

• A derived class inherits all of the base class’s members (except constructors and destructor)
  – An instance of a derived class has all the behaviors of its base class (can call the base class’s public methods)
  – A derived class cannot access the base class’s private data and methods directly by name
Inheritance Revisited

The class Sphere

displayStatistics()

mySphere.displayStatistics();
myBall.displayStatistics();

The class Ball

displayStatistics()
Inheritance – Syntax

- class derivedClass: `access-modifier` baseClass
- access modifier describes access semantics of base class components inherited by derived class
- Public methods can be used by any code
  - client, class member functions, derived classes
- Private members
  - class member functions and `friends`
- Protected
  - class members, friends, derived classes
Kinds of Inheritance

• Apply most restrictive access based on base access type and inheritance access modifier

• Public inheritance
  – Public/Protected $\rightarrow$ Public/Protected derived members

• Protected inheritance
  – Public/Protected $\rightarrow$ Protected derived members

• Private Inheritance
  – Public/Protected $\rightarrow$ Private derived members

• Private base class members remain private under all inheritance types
Figure 8-1 Inheritance: Relationships among timepieces
Inheritance – Example

• **Sphere** serves as a base class for **Ball**
  – some routines inherited, some new, and some are redefined (e.g. display Statistics())

<table>
<thead>
<tr>
<th>Sphere</th>
<th>Ball</th>
</tr>
</thead>
<tbody>
<tr>
<td>theRadius</td>
<td>theName</td>
</tr>
<tr>
<td>Sphere()</td>
<td>Ball()</td>
</tr>
<tr>
<td>~Sphere</td>
<td>~Ball()</td>
</tr>
<tr>
<td>setRadius()</td>
<td>setName()</td>
</tr>
<tr>
<td>getRadius()</td>
<td>getName()</td>
</tr>
<tr>
<td>getDiameter()</td>
<td>resetBall()</td>
</tr>
<tr>
<td>getCircumference()</td>
<td></td>
</tr>
<tr>
<td>getArea()</td>
<td>displayStatistics()</td>
</tr>
<tr>
<td>getVolume()</td>
<td></td>
</tr>
<tr>
<td>getDisplayStatistics()</td>
<td></td>
</tr>
</tbody>
</table>

- New
- Redefined
Multiple Inheritance

- Multiple inheritance
  - a derived class can have more than one base class
  - we will not study this kind of inheritance
Inheritance Revisited

• In general, a class’s data members should be private
Is-a Relationships

- Public inheritance should imply an *is-a* relationship
- Object type compatibility
  - you can use an instance of a derived class anywhere you can use an instance of the base class (but not the other way around)
- Example
  - A *ball* is a *sphere*
  - Given the following function declaration:
    void displayDiameter(Sphere thing);
    The following statements are valid:
    Ball myBall(5.0, “Volleyball”);
    displayDiameter(myBall);

see C8-Sphere.cpp, C8-Ball.cpp
Sphere/Ball Example

• Ball class (derived from Sphere)
  – both constructors call base class constructors to handle private radius data
  – getName() is a new method
  – setName() gives access to Ball data
  – resetBall() uses both Sphere and Ball access routines as the data is private in both classes
    • Ball could access name directly
Sphere/Ball Example – 2

• Ball class
  – displayStatistics() redefines the name in the derived class as a local method
    • to display Ball’s unique data element “theName”
    • uses full class::member scope resolution syntax to call Sphere’s displayStatistics()
  – instance of Ball has two data members
    • theName and theRadius (inherited)
    • since Sphere::theRadius is defined private (rather than protected or public) it can only be accessed through the public (get/set)Radius() methods
Has-a Relationships

• If the relationship between two classes is not is-a, do not use public inheritance
• Has-a relationship (also called *containment*)
  – a class has an object as a data member
  – cannot be implemented using inheritance
• Example: A has-a relationship between a pen and a ball
  
  ```cpp
  class Pen {
    ...
    private:
      Ball point;
  };
  ```
As-a Relationships

• Uses private inheritance
  – Example: Implement a stack as a list
    
    class Stack: private List
    
    • stack can manipulate the items on the stack by using List’s methods
    • the underlying list is hidden from the clients and descendants of the stack

• Private inheritance is useful when
  – a class needs access to the protected members of another class, or
  – if methods in a class need to be redefined
As-a relationship – Private Inheritance

• All public, protected and private elements of the base class are *private* in the derived class.

• Client code reference to a public base class routine through the derived class instance is illegal.

• Derived class completely wraps base class elements.

• Derived class is implemented using or is implemented in terms of the base class.

see C8-StackL.cpp, C8-List.cpp
Stack as-a List Example

• A stack is *not* a type a list since it has unique semantics

• Stack semantics *can* be implemented in terms on List semantics

• Private inheritance strongly conceals any list related semantics from the clients
  – only exposes push()/pop()
  – cannot access insert()/remove()

• Contrast with the Stack *has-a* List implementation from Chapter 4
  – just as good; both work
Inheritance & Class Relationships

• Public inheritance
  – extend or specialize an existing class
  – most common
  – *is-a* relationship between base/derived classes

• Protected inheritance
  – not very useful; not often used

• Private inheritance
  – to implement one class in terms of another
  – *as-a* relationship
Class Relationships

• IS-A: derived class is special kind of base class
  – public inheritance used to implement in C++
• AS-A: derived class implemented in terms of base class
  – private inheritance can be used in C++
• HAS-A: object A includes instance of object B as part of its implementation
  – encapsulation is just as good as inheritance
Virtual Functions

• Derived class sometimes need to modify or completely replace actions of a base class method
  – derived class is said to override the inherited method
• Base class must give permission for redefinition
  – by declaring the method as virtual
• Redefinition is permitted but not required
• Redefined methods must have exactly the same signature as the inherited base class methods
• Derived functions do not need to use the virtual keyword
• Friend and constructor functions cannot be virtual, but destructors can be
Virtual Functions – 2

- Base class Animal gives permission to override `breathe()` and `move()`
- Derived class Fish overrides both, but denies permission to further override `breathe()`
- WalkingCatFish overrides `move()` but uses inherited `breathe()`

```cpp
class Animal {
public:
  ... 
  virtual void breathe(); // uses a nose
  virtual void move(); // uses feet
  ... 
};
class Fish: public Animal {
public:
  ... 
  void breathe(); // uses gills
  virtual void move(); // uses fins
  ... 
};
class WalkingCatFish: public Fish {
  void move(); // uses fins as feet
};
```
class baseClass{
public:
    virtual void print(){
        cout << “BaseClass”;}
};

class firstClass : public baseClass{
public:
    void print(){
        cout << “firstClass”;}
};

class secondClass : public baseClass{
};

class thirdClass : public baseClass{
public:
    void print(){
        cout << “thirdClass”;}
};

class fourthClass : public thirdClass{
};

see C8-staticBinding.cpp
Name Binding Time

• Binding time can be compile-time or run-time
  – controls what methods are called in some cases
• see C8-staticBinding2.cpp
• Direct access using $b$ and $d$ instance variable
  obviously compile time
• But what if they point to a different type object??
  – bp1 is legal (but bogus)
  – dp2 illegal type conversion
Virtual Methods and Late Binding

• Methods declared as virtual are tracked at runtime by a virtual method table (VMT)
  – methods not declared as virtual can be redefined
  – methods declared as virtual are overridden

• Use of non-virtual methods is determined at compile-time and references to the function are compiled-in
  – lower overhead reference method

• Use of virtual methods is determined at runtime by consulting the VMT of the object accessed
  – pointer to calling object (this) given to every method call
  – higher overhead, but avoids some undesirable behaviors
Virtual Methods and Late Binding

• Late, or dynamic, binding
  – the appropriate version of a polymorphic method is decided at execution time
  – a polymorphic method has multiple meanings and overrides a method of the superclass
  – the outcome of an operation depends upon the objects on which it acts

• Defining class methods as virtual preserves flexibility at the cost of slightly higher VMT overhead.
Virtual Methods and Late Binding

• In general, define a class’s methods virtual, unless you do not want derived class to override them.

• Any class that contains a virtual method is called a **polymorphic** class
  – and is extensible, i.e., can add capabilities to a derived class without access to the ancestor’s source code

• Constructors cannot be virtual

• Destructors can and should be virtual

• A virtual method’s return type cannot be overridden
Abstract Base Class

• Classes may declare a virtual function prototype without providing an implementation for it
  – used as a placeholder for an API element, which derived classes are obligated to implement
• Method functions declared but not defined in a class are pure virtual

  virtual type func_name(param_list) = 0;

• A base class containing a pure virtual function is called an abstract base class
• No instances of an abstract class can exist since at least one method lacks an implementation.
Perfect adherence to object encapsulation is not always convenient or clear
– access to private, and protected, members of a class by collaborating classes can simplify implementation
– *friend* definition is a mechanism for granting other exceptions

Functions and classes can be *friends* of a class
Friend functions can access *private* and *protected*
Friend functions of a class are not class members
Friends of base class are not friends of derived classes
Friend Methods/Classes

• *Friend* functions permit input and output routines to have access to private and protected class data
  – general input/output routines can be friends of objects they are creating and initializing

• Useful when one class (A) contains an instance of another class (B) because A HAS-A B as part of its implementation
  – remember implementing stacks and queues using different mappings onto the List ADT operations
Friends – Example

• ‘write’ can access private variables in ‘Base’
  – cannot access private variables in Derived

```cpp
class Base{
public:
    friend void write(Base& b);
private:
    double bvalue;
};
void write(Base& b){
    cout << b.bvalue << endl;
}

class Derived:public Base{
public:
private:
    double dval;
};
int main(){
    Base b; Derived d;
    write(b); write(d);
}
```
Friend Methods/Classes

• A class List can be a friend of the class ListNode
  – List can access ListNode’s private and protected members)

```cpp
class ListNode
{
private:
... // define constructors and data
  // members item and *next
friend class List;
};
```
ADTs List and Sorted List Revisited

• Section 8.4 in Carrano provides a good discussion of applying these new concepts to familiar examples
  • abstract Base Class (BasicADT)
  • pure Virtual Functions
  • virtual Functions
• Three level class hierarchy
Class Templates

• Way to describe commonality among different solution components.
  – Situation: same basic structure with different data components
  – lists, stacks, queues, trees, etc.
  – same data structures, different data

• Parameterized class definition
  – data types are the parameters

```cpp
template <typename T>
class NewClass
{
    public:
        NewClass();
        NewClass(T initialData);
        void setData(T newData);
        T getData();
    private:
        T theData;
};
```
Class Templates – 2

• Declarations and methods must specify the type parameters in creating an instance

```cpp
int main() {
    NewClass<int> first;
    NewClass<double> second(4.8);
    first.setData(5);
    cout << second.getData() << endl;
}
```

```cpp
template <typename T>
void NewClass<T>::setData(T newData) {
    theData = newData;
}
```

```cpp
template <typename T>
T NewClass<T>::getData() {
    return theData;
}
```

see C8-templareEx1.cpp
Class Templates – 3

• Precede class definition template with
  template <typename T>
• Precede each method template with
  template <typename T>

see C8-ListT.cpp
• Data Type parameter <T> takes the place of typedef <listitemtypespec> ListItemType;
• Default constructor for the list node cannot initialize the data element
  – abstract definition of <T> has no information about what the type will be (it can be any type)
  – but copy constructor can
• Note the inclusion of the implementation source file at the end of the header file.
• Templat List class supports client code creating two standard lists holding double and char data
Class Templates – 4

- Abstract nature of the class description can lead to problems with apparently “standard” operations
- Everything the class does must be valid for every possible type specified for T
- For example, specified type should overload the “<” operator to support output
- Class template specification must thoroughly document all such operations and other assumptions made by the class implementation – must be satisfied by all classes provided as T
Class Templates – 5

• Compiler must know the class specified for T before it can compile the template instance

• Class template header and source files are defined as normal
  – but the implementation file is not compiled separately ahead of time
  – implementation file #included at the end of the header file

• All client code using the template thus includes it, making the definition available to compiler when it compiles instances for specific T
Overloaded Operators

• Overloaded operator has more than one meaning
• Overload common operators for classes to enable a particular operator to work correctly on instances of a class
• Example: a list
  – Define the equality operator for a list
    ```
    virtual bool operator == (const List& rhs) const
    ```
  – Overload the assignment operator to get a deep copy of a list
    ```
    virtual List& operator = (const List& rhs);
    ```
Overloading Operators -- Guidelines

- Can overload any operator except: ., .*, ::, ?, :, sizeof
- Cannot define new operators by overloading symbols that are not already operators in C++
- Cannot change the standard precedence of a C++ operator or the number of its operands
- At least one operand of an overloaded operator must be an instance of a class
- Cannot change the number of arguments for an overloaded method
- A typical class should overload the assignment, equality, and relational operators (= == != < <= > >=)
Iterators

• An iterator is an object that traverses a collection of like objects

• Common iterator operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Return the item that the iterator currently references</td>
</tr>
<tr>
<td>++</td>
<td>Move the iterator to the next item in the list</td>
</tr>
<tr>
<td>--</td>
<td>Move the iterator to the previous item in the list</td>
</tr>
<tr>
<td>==</td>
<td>Compare two iterators for equality</td>
</tr>
<tr>
<td>!=</td>
<td>Compare two iterators for inequality</td>
</tr>
</tbody>
</table>
Iterators

• A header file for the class `ListIterator`

```cpp
#include "ListNode.h"

class ListIterator
{
public:
  ListIterator(const List *aList, ListNode *nodePtr);
  const ListItemType & operator*();
  ListIterator operator++();
  bool operator==(const ListIterator& rhs) const;
  bool operator!=(const ListIterator& rhs) const;
  friend class List;

private:
  const List *container //ADT associated with iterator
  ListNode   *cur;      //current location in collection
};
```
Summary

• Inheritance
• Virtual methods and late binding
• Friend classes and methods
• Class templates
• Overloaded operators
• Iterators

• All these are ways to express desired semantics in C++ to specify algorithmic solution to a problem.