C++ Interlude 3: Exceptions

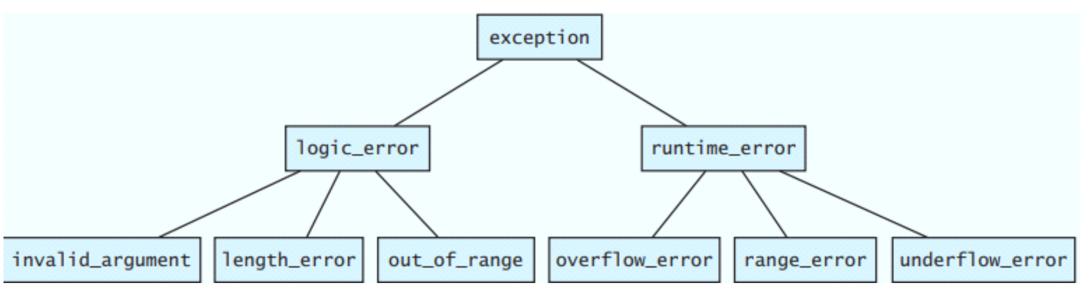
Thursday, February 18, 2021 6:47 PM

- Exception: an object that signals the rest of the program that something unexpected has happened
- Handle: we handle the exception when we detect and react to it

1. **ASSERTIONS**
   a. Assertion: a statement of truth about some aspect of a program's logic
      i. Use an *assert* statement to test a precondition or postcondition
   b. To use *assert*, include the following in your program: #include <cassert>
   c. To call the assert function: assert(someBooleanCondition);
   d. Example:

```
while (!found && (index < size))
{
   if (target == boxes[index].getItem())
      found = true;
   else
      index++;
}  // end while
assert(found);     // Verify that there is a box to return
return boxes[index];
// end findBox
```

1. **THROWING EXCEPTIONS**
   a. An alternate way of communicating or returning information to a function's client is to throw an exception - a thrown exception bypasses normal execution and control immediately returns to the client
   b. An exception can contain information about the error or unusual condition that helps the client resolve the issue and possibly try the function again.
   c. Format of a *throw* statement: throw *ExceptionClass(stringArgument);*
   d. Hierarchy of C++ exception classes:



2. **HANDLING EXCEPTIONS**
   a. NOTE: You CANNOT handle exceptions under *runtime_error* - you can only handle exceptions under *logic_error*.
   b. To handle an exception, use the try/catch blocks:

```
try
{
  < statement(s) that might throw an exception >
}
catch (ExceptionClass identifier)
{
  < statement(s) that react to an exception of type ExceptionClass >
}
```

   c. You can also have multiple catch blocks to handle different exceptions. The catch blocks must be ordered so that the most specific exceptions are caught before the more general exceptions.
   d. An uncaught exception propagates back to the main function and the program

execution terminates abnormally with an error message

3. PROGRAMMER-DEFINED EXCEPTION CLASSES

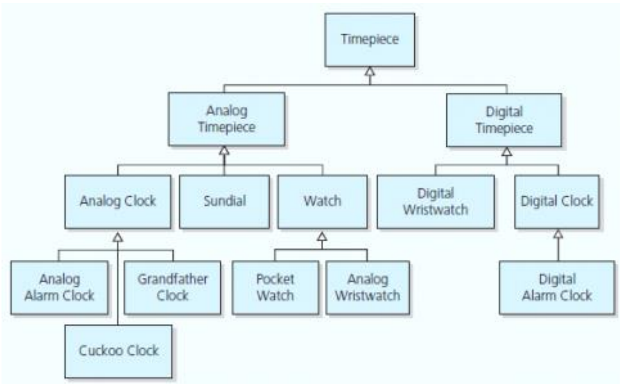
a. You can define your own exception class:

```
#include <stdexcept>
#include <string>
using namespace std;
class TargetNotFoundException: public exception
{
public :
   TargetNotFoundException(const string& message = "")
                 : exception("Target not found: " + message.c_str())
   {
   }  // end constructor
}; // end TargetNotFoundException
```

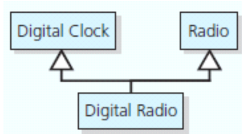
C++ Interlude 4: Class Relationships & Reuse

Tuesday, February 23, 2021 2:38 PM

4.1 Inheritance Revisited



- A class an derive the behavior and structure of another
- A digital alarm clock is a digital clock
- Inheritance enables the reuse of existing classes
- In C++, a derived class inherits all of the members of its base class, except the constructors and the destructor
- Multiple inheritance is when a derived class has more than one base class, for example:

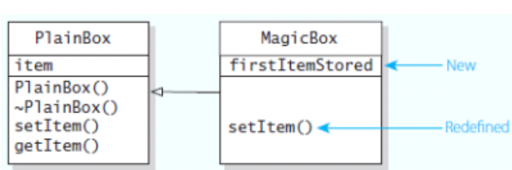


- However, a problem arises when the two base classes have similarly named methods, such as both *Digital Clock* and *Radio* have a method called *turnOn*.
- If multiple inheritance is used, the derived class inherits code from only one base class; any other base classes should be abstract base classes.
- A derived class can redefine inherited methods - a method in a derived class redefines a nonvirtual method in the base class if the two methods have the same name and parameter declarations.

LISTING C4-1 The class `PlainBox`, originally given in Listing C1-3

```
template<class ItemType>;
class PlainBox
{
private:
   ItemType item;

public:
   PlainBox();
   PlainBox(const ItemType& theItem);
   void setItem(const ItemType& theItem);
   ItemType getItem() const;
}; // end PlainBox
```



LISTING C4-2 The class `MagicBox`, originally given in Listing C1-7

```
template<class ItemType>;
class MagicBox : public PlainBox<ItemType>
{
private:
   bool firstItemStored;

public:
   MagicBox();
   MagicBox(const ItemType& theItem);
   void setItem(const ItemType& theItem);
}; // end MagicBox
```

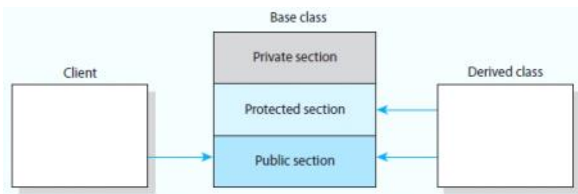
- An instance of a derived class has all the behaviors of its base class
- A derived class inherits private members from the base class, but cannot access them directly

More info on VMTs:

```
}; // end MagicBox
```

- An instance of a derived class has all the behaviors of its base class
- A derived class inherits private members from the base class, but cannot access them directly
- A derived class's methods can call the base class's public methods
- Clients of a derived class can invoke the base class's public methods
- Early (or static) binding can cause problems
- Late binding means that the appropriate version of a method is decided at execution time
- A polymorphic method has multiple implementations
- A virtual method is one that you can override
- A method that is virtual in a base class is virtual in any derived class
- Every class as a virtual method table (VMT), which remains invisible to the programmer. For each method in the class, the VMT contains a pointer to the actual instructions that implement the method's definition.
- For a virtual method, the compiler cannot complete the VMT. Instead, a call to a constructor during program execution sets the pointer. That is, the constructor establishes within the VMT to the versions of the virtual methods that are appropriate for the object. Thus the VMT is the mechanism that enables late binding.

PUBLIC, PRIVATE, AND PROTECTED SECTIONS OF A CLASS



- In addition to its public and private sections, a class can have a protected section which hides members from a class's clients but makes them available to a derived class. A derived class can reference the protected members of its base class directly, but clients of the base class or derived class cannot.
- How to begin the definition of the derived class:

`class DerivedClass : `*kindOfInheritance*` BaseClass`

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

- Public inheritance is the most important and the one that we will use the most often

***IS-A* AND *AS-A* RELATIONSHIPS**

- You should use public inheritance only when an *is-a* relationship exists between two classes of objects. Otherwise, do NOT use public inheritance.
- When public inheritance is inappropriate, if your class needs access to the protected members of another class or if you need to redefine methods in that class, you can form an *as-a* relationship, which is basically private inheritance.

```
class Stack : private List
```

4.2 Containment: *Has-a* Relationships

- *Has-a*, or containment, means a class has an object as a data member.
- Ex: A pen *has* a ball tip, but the pen itself isn't a ball.

```
class Pen
{
private:
    Ball point;
    ...
}; // end Pen
```

- When an instance of an object cannot exist independently of the containing class (when the pen is destroyed, so is the ball), this type of containment is called composition.
- Another form of containment is aggregation. In an aggregate relationship, the contained item can exist independently of the containing class (PlainBox class has an instance of ItemType named item, but if plain box is destroyed, we can still use item).
- Favor containment over inheritance

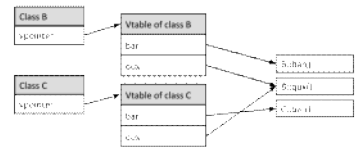
4.3 Abstract Base Classes Revisited

- A pure virtual method has an undefined body and is written as

`virtual `*prototype*` = 0;`

- A class that contains at least one pure virtual method is an abstract base class
- An abstract base class has descendants but no instances

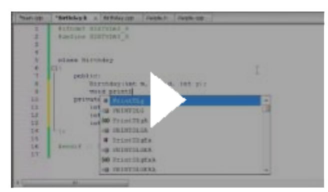
More info on VMTs:



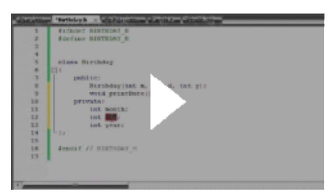
[Understanding Virtual Tables]

What is composition?

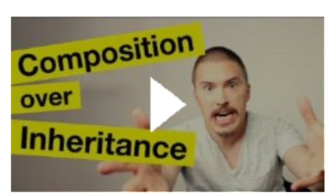
[Buckys C++ Programming Tutorials - 46 - Composition]



[Buckys C++ Programming Tutorials - 47 - Composition Part 2]



[Composition over Inheritance]



C++ Interlude 5: Overloaded Operators & Friend Access

Tuesday, February 23, 2021 2:38 PM

5.1 Overloaded Operators

- An operator with more than one meaning is overloaded and is an example of a simple form of polymorphism.
- To overload an operator, you define an operator method whose name has the following form where *symbol* is the operator you want to overload.
  `operator`*symbol*
- For example, for ==, name the method operator== and declare one argument: the object that will appear on the right-hand side of the operator. The current object represents the object on the left-hand side of the operator.
  ```
  bool operator==(const LinkedList<ItemType>& rightHandSide) const;
  ```
- Example of how this method start being implemented for a LinkedList:
  ```
  template <class ItemType>
  bool LinkedList<ItemType>::operator==(const
                                        LinkedList<ItemType>& rightHandSide) const
  {
     bool isEqual = true; // Assume equal

     // First check whether the number of items is the same
     if (itemCount != rightHandSide.getLength())
        isEqual = false;
  ```

OVERLOADING = FOR ASSIGNMENT

- Without an overloaded assignment operator, you get a shallow copy instead of a deep copy. A deep copy is necessary for a dynamically allocated data structure such as a chain of linked nodes.
- How to overload the assignment operator for a LinkedList
  ```
  template <class ItemType>
  LinkedList<ItemType>& LinkedList<ItemType>::operator=(const
                                              LinkedList<ItemType>& rightHandSide)
  {
     // Check for assignment to self
     if (this != &rightHandSide)
     {
        this->clear();                       // Deallocate left-hand side
        copyListNodes(rightHandSide);        // Copy list nodes
        itemCount = rightHandSide.itemCount; // Copy size of list
     } // end if

     return *this;
  } // end operator=
  ```

OVERLOADING + FOR CONCATENATION

- Link-based list:
  ```
  LinkedList<ItemType>&
                operator+(const LinkedList<ItemType>& rightHandSide) const;
  ```
  ```
  concatList = a new, empty instance of LinkedList
  concatList.itemCount = itemCount + rightHandSide.itemCount
  leftChain = a copy of the chain of nodes in this list
  rightChain = a copy of the chain of nodes in the list rightHandSide
  Set the last node of leftChain to point to the first node of rightChain
  concatList.headPtr = leftChain.headPtr
  return concatList
  ```
- Array-based list:
  - Decide on the capacity of the resulting list
  - Create a new ArrayList object and copy the entries from the first list into the first array elements and then place the entries in the second list into the elements that follow.

5.2 Friend Access and Overloading

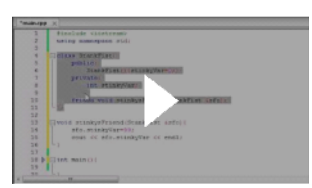
- Functions and classes can be friends of a class
- A class can provide additional access to its private and protected parts by declaring other functions and classes as friends.
  ```
  template<class friendItemType>
  friend ostream& operator<<(ostream& outputStream,
                             const LinkedList< friendItemType >& outputList);
  ```
- Define a node for the ADT list as follows:
  ```
  template<class ItemType>
  class ListNode            // A node on the list
  {
  private:
     ItemType item;         // A data item on the list
     Node<ItemType> *next;  // Pointer to next node

     Node();
     Node(const ItemType& nodeItem, Node<ItemType>* nextNode);

     // Friend class - can access private parts
  ```

How does the friend access specifier work?

[Buckys C++ Programming Tutorials - 48 - friend]



```
template<class ItemType>
class ListNode            // A node on the list
{
private:
   ItemType item;         // A data item on the list
   Node<ItemType> *next; // Pointer to next node

   Node();
   Node(const ItemType& nodeItem, Node<ItemType>* nextNode);

   // Friend class - can access private parts
   friend class LinkedList<ItemType>;
}; // end ListNode
```

- Friend methods can access the private and protected parts of the class.
- Friend methods are not members of the class.
- When a class is declared as a friend of a class C , all of its methods have access to the private and protected parts of the class C.
- Friendship is not inherited. The private and protected members declared in a derived class are not accessible by friends of the base class.

C++ Interlude 6: Iterators

Tuesday, February 23, 2021 2:38 PM

6.1 Iterators

COMMON ITERATOR OPERATIONS

- An iterator is a program component that enables you to traverse a collection of data, such as the data in a list, beginning with the first entry.
- During iteration:
  - Each data item is considered once
  - You can modify the collection as you traverse it by adding, removing, or changing entries
- We are familiar with iteration because we have written for loops
- Input iterators are simple iterators that traverse a collection of items, retrieve an item in a collection, and compare two iterators to determine whether they access the same entry in the collection.

Note: Common iterator operations

Operation	Description
*	Return the item that the iterator currently references
++	Move the iterator to the next item in the collection
--	Move the iterator to the previous item in the collection (used only for bidirectional or random iterators)
==	Compare two iterators for equality
!=	Compare two iterators for inequality

- To overload the above operators for your iterator class, you must derive your iterator class from the C++ template class *iterator* . This template is used to identify the category of iterator you are creating by using an iterator category tag as the template type.

For example, to declare an input iterator for the class `LinkedList`, we would use the lines:

```
template <class ItemType>
class LinkedIterator : public iterator<input_iterator_tag, int>
```

input_iterator_tag indicates that this iterator implements input-iterator functionality. The *int* identifies the type of value used to measure the distance between two iterators. The distance between two iterators is the number of elements of positions between the current positions of the two iterators.

Note: C++ iterator categories

All of the following iterators provide operations that copy or assign (=) and increment (++).

Category	Tag	Operation
Input iterator	`input_iterator_tag`	Equality/inequality (==, !=), access collection entry (*)
Output iterator	`output_iterator_tag`	Change a collection entry (*)
Forward iterator	`forward_iterator_tag`	Same as the input and output iterators and has a default constructor
Bidirectional iterator	`bidirectional_iterator_tag`	Same as the forward iterator, but also can traverse the collection backward (--)
Random-access iterator	`random_iterator_tag`	Same as the bidirectional iterator and adds support for arithmetic (+, -, +=, -=) and relational (<, <=, >, >=) operations between iterators. Supports the [] operator to directly access collection entries.

USING ITERATOR OPERATIONS

- Standard containers in C++ implement two special methods, *begin* and *end*, that return an iterator to the first entry and last entry respectively. These iterators should have an order of magnitude in performance of O(1) as they move from entry to entry.
- Normally, to display all the entries in a list, you need to perform this O(n) operation:

```
int currentPosition = 1;
while (currentPosition <= myList.getLength())
{
   cout << myList.getEntry(currentPosition); // O(n) operation
   currentPosition++;
}  // end while
```

- Rewrite the code above using *LinkedIterator* objects:

```
LinkedIterator<string> currentIterator = myList.begin(); <-- points to the first entry in the list
while (currentIterator != myList.end())
{
   cout << *currentIterator; // O(1) operation<-- dereference the iterator
   ++currentIterator;                         to access entry at position
}  // end while
```

^ use prefix operator ++*currentIterator* to differentiate from an arithmetic increment postfix ++

- *Begin()* method:

What is a standard container?

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

https://www.cplusplus.com/reference/stl/#:~:text=Standard%20Containers,the%20types%20supported%20as%20elements.

```
template <class ItemType>
LinkedIterator<ItemType> LinkedList<ItemType>::begin()
{
   return LinkedIterator<ItemType>(this, headPtr);
}  // end begin
```

- *End()* method:

```
template <class ItemType>
LinkedIterator<ItemType> LinkedList<ItemType>::end()
{
    return LinkedIterator<ItemType>(this, nullptr);
}  // end end
```

IMPLEMENTING AN ITERATOR

- *LinkedIterator* is a distinct class separate from *LinkedList*.
- The constructor has two parameters:
  - The list traversed by the iterator
  - An initial node for the iterator to reference

LISTING C6-1 The header file for the class `LinkedIterator`

```
#ifndef _LINKED_ITERATOR
#define _LINKED_ITERATOR

#include <iterator>
#include "Node.h"

template<class ItemType>
class LinkedList;
```

<--provide a forward declaration of *LinkedList* class to resolve circular references

```
template <class ItemType>
class LinkedIterator : public iterator<input_iterator_tag, int>
{
private:
   // ADT associated with iterator
   const LinkedList<ItemType>* containerPtr;

   // Current location in collection
   Node<ItemType>* currentItemPtr;

public:
   LinkedIterator(const LinkedList<ItemType>* someList,
                  Node<ItemType>* nodePtr);

   /** Dereferencing operator overload.
    @return  The item at the position referenced by iterator. */
   const ItemType operator*();

   /** Prefix increment operator overload.
    @return  The iterator referencing the next position in
       the list. */
   LinkedIterator<ItemType> operator++();

   /** Equality operator overload.
    @param LinkedList  The iterator for comparison.
    @return  True if this iterator references the same list and
       the same position as rightHandSide, false otherwise. * /
   bool operator==(const LinkedIterator<ItemType>& rightHandSide) const;

   /** Inequality operator overload.
    @param LinkedList  The iterator for comparison.
    @return  True if this iterator does not reference the same
       list and the same position as rightHandSide,
       false otherwise. */
   bool operator!=(const LinkedIterator<ItemType>& rightHandSide) const;
}; // end LinkedIterator

#include "LinkedIterator.cpp"
#endif
```

What are forward declarations?

refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program).

https://www.geeksforgeeks.org/what-are-forward-declarations-in-c/

LISTING C6-2 The implementation file for the class `LinkedIterator`

```
#include "LinkedIterator.h"

template <class ItemType>
LinkedIterator<ItemType>::
LinkedIterator(const LinkedList<ItemType>* someList,
               Node<ItemType>* nodePtr):
               containerPtr(someList), currentItemPtr(nodePtr)
{
}  // end constructor

template <class ItemType>
const ItemType LinkedIterator<ItemType>::operator*()
{
   return currentItemPtr->getItem();
}  // end operator*

template <class ItemType>
LinkedIterator<ItemType> LinkedIterator<ItemType>::operator++()
{
```

```
    currentItemPtr = currentItemPtr->getNext();
    return *this;
} // end prefix operator++

template <class ItemType>
bool LinkedIterator<ItemType>::operator==(const
                    LinkedIterator<ItemType>& rightHandSide) const
{
    return ((containerPtr == rightHandSide.containerPtr) &&
            (currentItemPtr == rightHandSide.currentItemPtr));
} // end operator==

template <class ItemType>
bool LinkedIterator<ItemType>::operator!=(const
                    LinkedIterator<ItemType>& rightHandSide) const
{
    return ((containerPtr != rightHandSide.containerPtr) ||
            (currentItemPtr != rightHandSide.currentItemPtr));
} // end operator!=
```

- NOTE: An alternative is to use only iterators that have access to public methods of your data structure, but then you lose the efficiencies gained by directly accessing the structure. The best approach is to design an iterator for your class at the same time that you design your ADT, so that you can coordinate the features and ensure that the iterator does not change the class's structure.

6.2 Advanced Iterator Functionality

- The use of the iterator in a number of C++ standard functions simplifies the processing of many common algorithms, such as displaying the items in a collection, searching a collection, and counting the number of occurrences of an item in the collection.

Note: Some useful C++ functions that use iterators

- Process entries in a collection from *start_iterator* position to *end_iterator* position using the function *function_to_perform*:

  `for_each(`*start_iterator*`,` *end_iterator*`,` *function_to_perform*`);`

- Return an iterator to the position of the first occurrence of *target* between *start_iterator* and *end_iterator*:

  `iteratorType someIterator = find(`*start_iterator*`,` *end_iterator*`,` *target*`);`

- Return the number of occurrences of *target* between *start_iterator* and *end_iterator*:

  `int numberOccur = count(`*start_iterator*`,` *end_iterator*`,` *target*`);`

- Compare entries in collection 1 from *start1_iterator* through *end1_iterator* to those in collection 2 beginning at *start2_iterator*:

  `bool result = equal(`*start1_iterator*`,` *end1_iterator*`,` *start2_iterator*`);`

- Move *someIterator* from its current position forward *distanceToAdvance* positions:

  `advance(`*someIterator*`,` *distanceToAdvance*`);`

- Determine the distance or number of positions from *someIterator* to *anotherIterator*:

  `int theDistance = distance(`*someIterator*`,` *anotherIterator*`);`