CSCI 3110 Advanced Data Structures

Review for C++ Class

C++ Class Review

CSCI 3110

C++ Class Review

CSCI 3110

Three Pillars of Object-oriented Programming

- Encapsulation
  - The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings
- Inheritance
  - A way to form new classes using classes that have already been defined.
  - The new classes, known as derived classes, take over (or inherit) attributes and behavior of the pre-existing classes, which are referred to as base classes.
  - It is intended to help reuse existing code with little or no modification.

Three Pillars of Object-oriented Programming

- Polymorphism
  - The characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a function to have more than one form.
  - The essence of polymorphism is to combine encapsulation with inheritance and thus hide the details of specialization

Encapsulation

- Encapsulation is critical to building large complex software which can be maintained and extended.
- Many studies have shown that the greatest cost in software is not the initial development, but the thousands of hours spent in maintaining the software.
- Well encapsulated components are far easier to maintain.

Encapsulation

- Practical rules of implementation which enhance the encapsulation of your objects:
  - Never put data in the public interface of your class
  - Create accessor methods (e.g., Get and Set methods) for all the data in your class
  - Keep helper methods out of the public interface

C++ Class Review

CSCI 3110

C++ Class Review

CSCI 3110

Class Members

- A class can contain member data and member functions.
- A member data or a member function can be either public, protected, or private
- Public member data/function can be accessed by all clients
- Private member data/function can only be accessed by the class itself

Class Member Data

- A member data can be
  - instance variables: Each object/instance of the class has its own copy
  - instance constant variables: Each object/instance of the class has its own copy of the constant
  - class variables: All objects/instances of the class share the same copy

    syntax: static int NumOfInstanceCopy;
  - class constants: Constants shared by all instances

    syntax: static const int MinDeposit;
- In C++11, all non-static member data and member data of static const integral type can be initialized where it is declared (in its class).

Constant Member Functions

- A constant member function is a member function followed by the keyword *const*
- Syntax: string getName() const;
- A constant member function is prohibited from changing any data stored in the calling object.
- Non-constant member functions cannot be invoked within a constant member function since they may change member data.

Static Member Functions

- A static member function is a member function following the keyword *static*
- Syntax: static Student create();
- A static member function cannot be declared with the keywords *virtual*, *const*.
- A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared.

Static Member Functions

- Static member functions exist even before any object is created.
- Static member functions can be invoked in the following form:

  ClassName::StaticMemberFuncName(func. argument list)
- The static member functions are not used very frequently in programs. But nevertheless, they become useful whenever we need to have functions which are accessible even when the class is not instantiated.
- For example: wrap a global function in a class, or provide a function instead of constructors to create an object/instance.

C++ Class Review

CSCI 3110

Constructor

- Constructor provides a chance to initialize member data after the creation of an object and before any interface function is invoked
- Constructor properties:
  - Using class name as its name
  - **NO** return type, even void
  - Can have parameters
  - May have multiple constructors
  - Invoked automatically and implicitly
  - Initializer list can be used and preferred (separated by ,)
    - A initializer list uses a functional notation that consists of a data member name followed by its initial value enclosed in parentheses.
    - Initializer list is more efficient
    - Initializer lists can only be used with constructor
  - Like any other member functions, constructors can be public or private.

Default Constructor

- Default Constructor: the constructor taking no parameters
  - The compiler generates a default constructor if NO constructor is defined
  - A compiler-generated default constructor may not initialize data members to values that you will find suitable.

Default Constructor

```
class Date
{
    Date(int year=2007, int month=1, int date=1);
    ...
};
```

What's the meaning of "assignment operator" in the above constructor?

Default Parameter

- Default parameter
  - A default parameter is a function parameter that has a default value provided to it.
  - If the user does not supply a value for this parameter, the default value will be used.
  - If the user does supply a value for the default parameter, the user-supplied value is used.
  - If a parameter is a default parameter, all parameter following it must be default parameters too.
  - If the declaration and definition is separated, no need to specify the default value in the definition.
- **NOTES**: Default parameters do **NOT** count towards the parameters that make the function unique.

```
void PrintValues(int nValue);
void PrintValues(int nValue1, int nValue2=20);
```

Copy Constructor

- Copy constructor:
  - The copy constructor lets you create a new object from an existing one by initialization
  - A constructor with one parameter: a reference to an object of this class
  - If the only parameter of a constructor is not a reference parameter, it is not a copy constructor.
- Invoked automatically and implicitly if necessary
- If no copy constructor is defined, a default copy constructor is generated by the compiler
  - member-by-member copy

Copy Constructor

- Invoked automatically and implicitly if necessary
  - A variable is declared which is initialized from another object

    Person q("Mickey"); // constructor used to build q
    Person r(q); // copy constructor used to build r
    Person p = q; // copy constructor used to init. in decla.
    p = q; // Assignment operator, no (copy) constructor
  - Class objects are passed by value, returned by value, or thrown as an exception

    f(p); //copy constructor initializes parameters
  - Question: Why the parameter of copy constructor must be a reference?

Copy Constructor: Guidance

- Don’t write a copy constructor if member-to-member copies are ok
- If you need a copy constructor, you also need a destructor and operator = (Related with dynamic memory allocation)

Copy Constructor vs. Assignment Operator

A copy constructor is used to initialize *a newly declared* variable from an existing object. This makes a deep copy like assignment, but it is somewhat simpler:

- There is no need to test to see if it is being initialized from itself.
- There is no need to clean up (eg, delete) an existing value (there is none)
- A reference to itself is not returned

Passing Arguments to Constructors

- When an object is created, a suitable constructor is chosen based on passed arguments to initialize the object.
- Pitfall – initialize objects by default constructor:
  string a;
  string b(); // Is b a string variable?
- Pitfall – Constructors with one argument serve double duty as type conversions. To avoid this, use keyword explicit.

```
class Array
{
public:
        Array(size_t count);
       // etc.;
};
void UseArray(const Array& array);

Array array = 123; // What happens here?
UseArray(123); // What happens here?
```

Copy Constructor vs. Assignment Operator

A copy constructor is used to initialize *a newly declared* variable from an existing object. This makes a deep copy like assignment, but it is somewhat simpler:

- There is no need to test to see if it is being initialized from itself.
- There is no need to clean up (eg, delete) an existing value (there is none)
- A reference to itself is not returned

Destructor

- Destructor provides a chance to release resources the object has, such as memory, files, locks, etc.
- Destructor properties:
  - Destructor name: followed by class name
  - NO return type, even void
  - Can**NOT** have parameters
  - Can**NOT** be overloaded
  - Invoked automatically and implicitly
  - Compiler generates a default destructor if no one is defined

Destructor

- Both constructor and destructor can be public or private
- If all explicitly declared constructors are private, clients cannot create an object through these constructors
- If the destructor is private, you cannot create an object of this class without using new operator.
- If the destructor is private, no object of the class can be deleted.

C++ Class Review

CSCI 3110

Create an object

- A class is a user-defined data type. It can be used like any other built-in types like int.
- Different ways to create an object
  - Date today;
    // default constructor is called
  - Date anniversary(1997, 10, 23);
    // constructor taking 3 int parameters is called
  - Date commencement = today;
    // copy constructor is called
  - Date ACMMeeting(commencement);
    // copy constructor is called
  - Date *currentEventDay = new Date;
    // default constructor is called
  - Date * currentEventDay = new Date();
    // equivalent to the previous one
  - Date *anniversary = new Date(1997, 10, 23);

Accessing Class Members

- Access public member data/functions:
  - By . (dot) operator
    Example: today.getYear()
  - By -> (selection operator) Example: currentEventDay->getYear()
    // currentEventDay must be a pointer
- Access class variables or constants
  - By :: (scope resolution operator)
    Example: Date::NumOfDateObjectCopy = 5;
    // NumOfDateObjectCopy is a class variable of class Date

    currentMonth = Date::Feb;
    // Feb is either a static constant or an enumerator of class Date
  - They can also be accessed like any other “normal” public member data
- Access static member functions
  - By :: (scope resolution operator)
    Example: Date::getNumOfCopy()
    // getNumOfCopy is a static function of class Date
  - They can also be accessed like any other “normal” public member functions

Array of Objects

- Like any other data type in C++, we can define arrays of class objects.
- Example: InventoryItem inventory[40];
  - Defines an array of 40 *InventoryItem* objects
  - The name of the array is *inventory*
  - The default constructor is called for each object in the array
  - If no default constructor is defined in the class *InventoryItem*, a compile error occurs.
- If those constructors have special needs (large and expensive memory allocation), this can be a performance bottleneck and a waste of time if not all elements are to be used.
  - Solution: Use STL vector defined in <vector>
  - vector<InventoryItem> inventory;
    inventory.reserve(100);

Array of Objects: Initializer List

- Define an array of objects and initialize them by a constructor requiring parameters

```
InventoryItem inventory[3] = { InventoryItem("Hammer"),
                               InventoryItem("Wrench"),
                               InventoryItem("Pliers")
                             };
```

- Define an array of objects and initialize them by different constructors

```
InventoryItem inventory[3] = { InventoryItem("Hammer"),
                               InventoryItem("Wrench", 9.3, 100),
                               InventoryItem()
                             };
```

Passing Objects as Function Arguments

- By default, an object is passed by value.
  - If an object is passed by value, (default) copy constructor is invoked automatically to copy the object argument to the parameter
- An object can also be passed by reference
  - Syntax: Appending the symbol (&) to the class type
  - The address of the object is copied.
  - Any modification through the object parameter applies to the object argument.

C++ Class Review

CSCI 3110

Class Friends

- A class can declare the following entities as its friends
  - a global function
  - a member function of another class
  - another class
- A friend of a class can access ANY (even private) member data and functions
- Declare a class friend is dangerous because it violates the principle of encapsulation, and you have no control over your friends.
- Therefore, class friends are not encouraged.

Class Friends: Syntax


```
class Balance
{
public:
friend void printBalance(const Balance&); // a global function as a friend
friend double SavingAccount::getBalance(); // a member function as a friend
friend CheckingAccount; // a class as a friend
...
};
```

- friend is a key word
- The position of friend declarations does not matter. No difference if friend declaration is either in public, or in private range.

C++ Class Review

CSCI 3110

C++ Class Review

Function Overloading

- function overloading: the practice of declaring multiple functions with the same name.
- If a function is overloaded, the function name is not sufficient to decide which function is being called. It needs the number, type, and order of the parameters.
- Overloaded functions cannot differ only by their return type.
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Function Overloading

- The following function are overloaded:

```
int      f( int, float );
void     f( int, float, int );
string   f( int, string );
int      f( string, int );
```

- The following function are NOT overloaded:

```
int      f(int, float);
void     f(int, float);
```

A compile-time error occurs if the above two functions are defined in the same scope.

reason: Two function are not overloaded if they only differ in the return type since system has no idea which version should be called when the return value is ignored. For example: f(10, 2.4);

Function Overloading

- The compiler determines which function is being invoked by analyzing the parameters

```
float tryMe(int x) {
        return x + .375;
}

float tryMe(int x, float y) {
        return x*y;
}

result = tryMe(25, 4.32); // which tryMe is called?
```

- Which version of the function f (defined in previous slide) is called int the following statements:

```
f ( 10, 20.0 );
f ( 10, 10 );
f ("10", 20 );
f ( 10, "20.0");
f ( 10, 10.3, 12 );
```

Function Overloading & Default Parameters

- Default parameters do **NOT** count towards the parameters that make the function unique.
- Example:

```
void PrintValues(int nValue);

void PrintValues(int nValue1, int nValue2=20);
```

- What can be wrong in the above code?

C++ Class Review

Operator Overloading

- Operator overloading provides a way to define and use operators such as +, -, *, /, and [] for user-defined types such as classes. By defining operators, the use of a class can be made as simple and intuitive as the use of intrinsic data types.
- Operator overloading makes life easier for the users of a class, not for the developer of the class.
- Example:

```
int i;
char c;
string s;
cin > > i > > c > > s; // > > is overloaded
i = 5 + 4;
s = "string1" + "string2"; // + is overloaded
```

Operator Overloading

- Operator overloading
  - Can overload any C++ operator except **. .* :: ?: sizeof**
  - CanNOT define new operators by overloading symbols that are not already operators in C++
  - CanNOT change standard precedence of a operator or its number of operands
  - At least one operand of an overloaded operator must be an instance of a class

Operator Overloading:Syntax

- Overload an operator globally (not a class member function)
  - Syntax to overload a binary operator:
    ReturnType operator X (OperandType1 firstOperand, OperandType2 secondOperand);
    where X is the binary operator to be overloaded
- Overload a binary operator as a class member:
  - The first operand is always the “current object”, through which the operator is invoked.
  - Syntax to overload a binary operator:
    ReturnType operator X (OperandType2 secondOperand);
    where X is the binary operator to be overloaded
- ReturnType can be a constant, or a reference. The same to parameters.

Operator Overloading:Example

- Assume we have a class *Balance* representing bank account balance.
- The following are the syntax to overload different operators

```
class Balance
{
public:
    // other functions
    Balance operator + (const Balance& right); //overload binary operator +
    Balance operator - (); // overload unary operator -
    const Balance& operator = (const Balance&); //overload assignments
    bool operator == (const Balance&); //overload equal operator
    ...
};
```

- Notes: **operator** is a keyword.

Operator Overloading:Example

- With overloaded operator in class *Balance*, we can write the following statements:

```
Balance bal1(25.98);
Balance bal2(33.45);
Balance bal3; // assume corresponding constructors exist
bal3 = bal2; //equivalent to: bal3.operator=(bal2)
bal3 = bal1 + bal2; // equivalent to: bal3.operator=( bal1.operator+(bal2) )
bal3 = bal2 + bal1; // equivalent to: bal3.operator=( bal2.operator+(bal1) )
bal1 = -bal2;  // equivalent to: bal1.operator-(bal2)
if ( bal1 == bal2 )  //equivalent to: bal1.operator-(bal2)
    cout « "they are equal" « endl;
```

Overloading Function Call Operator

- Overloading function call operator ()
  - To create objects which behave like functions,
  - For classes that have a primary operation.
  - The function call operator must be a member function, but has no other restrictions - it may be overloaded with any number of parameters of any type, and may return any type.
  - A class may also have several definitions for the function call operator.
- Syntax: ReturnType operator() (function parameter list);

Overloading Function Call Operator

- Example:

```
class Sorting
{
public:
    void operator () (int intVal[], size_t size);
    void operator () (Iterator begin, Iterator end);
    void operator () (void);
    ...
};
```

- Usage

```
int value[8]={1, 3, 67, 7, 15, 10, 2, 100};
Sorting sort;
sort(value, 8);
sort(); // correct
```

Overloading ++ and −− operators

- Both ++ and −− operators have two forms: prefix and postfix

```
class Balance
{
public:
    // other functions
    Balance& operator ++ (); //overload prefix operator ++
    Balance operator ++ (int); // overload postfix operator ++
    ...
};
```

- Overloaded prefix ++ operators takes no parameters, while overloaded postfix ++ operator takes a DUMB integer parameter. Similar to the −− operator.
- Typically, overloaded prefix ++ operator returns a reference of the object, while overloaded postfix ++ operator returns a copy of the object. Similar to the −− operator.
- From the implementation, you can see that prefix operators are more efficiency (See example given in the class).

Overloading > > and < < Operators

- We can overload > > and < < as a class member function, but it does not behave as we expect:

```
class Balance
{
public:
    // other functions
    ostream& operator < < (ostream&); //overload < <
    istream& operator > > (istream&); //overload > >
    ...
};

Balance bal(33.12);
cout < < bal;  //compile error since it is equivalent to cout.operator<<(bal)
bal < < cout; // correct, but is this what we want?
bal < < bal < < cout; // correct: bal.operator<<( bal.operator<<(cout) )
```

Overloading > > and < < Operators

- To solve this problem, we have to overload > > and < < as global functions

```
class Balance
{
public:
    // other functions
    friend ostream& operator < < (ostream&, const Balance&);
    friend istream& operator > > (istream&, Balance&);
    ...
};

ostream& operator < < (ostream& outs, const Balance& value) //overload
< <
{
    //implementation
}
istream& operator > > (istream& outs, Balance& value) //overload > >
{
    //implementation
}
```

Overloading >> and << Operators

- Questions in the overloading of >> and << operators
  - Why is the second parameter in overloaded << operator is a const reference, and a reference in overloaded >> operator?
  - Why the return type of overloaded >> and << operators is a reference to istream and ostream, respectively?

Overloading Type Conversion Operator

- Data type conversion happens “behind the scenes” with the built-in data types.

  For example:

```
int i=10;
double d = i;
```

- The same functionality can also be given to class objects by writing special operator functions to convert a class object to any other type.

- For example, the following overloaded operator can convert a Balance object into a double.

```
class Balance
{
public:
    operator double();
    ...
};
```

Overloading Type Conversion Operator

- Please notice that
  - No return type is specified since it is a Balance-to-double conversion function. It will always return a double.
  - Similarly, there is no parameters.
  - *double* can be replaced by any built-in or user-defined data types.

- Then, the following code is legal:

```
Balance bal(33.23);
double d = bal;
double current = double(bal);
```

C++ Class Review

CSCI 3110

C++ Class Review

Function Template

- Generic Programming: programming using types as parameters
- Function Template
  - A technique allowing us to create “one” function such that its functionality can be adapted to more than one type or class without repeating the entire code for each type.
  - Syntax:
    template<template parameter list>
    ReturnType FunName(function parameter list);
  - Template parameter list: a sequence of type parameters and value parameters separated by commas.
  - type parameter declaration:
    class T ;
  - value parameter:
    int minLimit ;

Template Parameters

- Template parameter list: a sequence of type parameters and value parameters separated by commas.
- type parameter declaration:
  class T ;
- value parameter:
  int minLimit ;

Function Template:Example


```
template<class  T>
T max( T first, T second, T third )
{
    T lager;
    lager = (first < second)? second : first;
    return (larger < third)? third : larger;
}
```

Invocation examples:

```
max(10, 30, 15);
max(2.34, 12.45, 0.33);
max(bal1, bal2, bal3);
//Assume Balance is a class overloading the operator <,
// and bal1, bal2, bal3 are Balance objects.
```

Notice: When a template function is called, the types of the function arguments determine which version of the template is used; that is, the template arguments are deduced from the function arguments. Also the following function call are also correct.

```
max<int>(10, 30, 15);
max<double>(2.34, 12.45, 0.33);
max<Balance>(bal1, bal2, bal3);
```

Template Instantiation

- How function template works?
  - When the compiler encounters a call to a template function (For example: max<int>(10, 30,45)), it uses the template to automatically generate a function replacing each appearance of **T** by the type passed as the actual template parameter (**int** in this case) and then calls it.
  - This process is automatically performed by the compiler and is invisible to the programmer.
  - The process of generating a function declaration from a function template is called template instantiation.

Function Template

- The keyword **class** can be replaced by another one **typename** without making any difference.
- Template parameter can be used just like regular types.
- - When the compiler encounters a call to a template function (For example: max<int>(10, 30,45)), it uses the template to automatically generate a function replacing each appearance of **T** by the type passed as the actual template parameter (**int** in this case) and then calls it.
  - This process is automatically performed by the compiler and is invisible to the programmer.
  - The process of generating a function declaration from a function template is called template instantiation.

C++ Class Review

Class Template

- Similar to function template, class templates are used where we have multiple copies of code for different data types with the same logic.

- Syntax and example

```
//Sample code for C++ Class Template
template  <class T, int Max> class MyQueue
{
    T m_data[Max];
    int m_tail; //index to the last item public:
    void Add(T const &d);
    void Remove();
    void Print();
};
```

Class Template

- Implementation of a member function:

```
template <class T, int Max>
Queue<T,Max>::Queue()
    m_tail(-1)
{
}

template <class T, int Max>
void Queue<T,Max>::add(const T& item)
{
    if (m_tail == Max -1)
        throw runtime_error("No more space");
m_data[++m_tail] = item;
}
```

Class Template: Benefits

- One C++ Class Template can handle different types of parameters.
- Compiler generates classes for only the used types.
- If the template is instantiated for int type, compiler generates only an int version for the c++ template class.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

Class Template: Key Concepts

- A C++ class template is used to specify a class in terms of a data-type parameter
- When you (the client) declare instances of a class template, you must specify the actual data type that the class template will use.
- Any operation required of the actual data type should be clearly documented in the class template.
- Templates can have more than one data-type parameter.
- If a set of functions or classes have the same functionality for different data types, they becomes good candidates for being written as Templates.

C++ Class Review

CSCI 3110

Why Iterator

- A great deal of programming has to do with manipulating sequences of elements
- You usually do the same sorts of things to a sequence of elements regardless of its type: traversing, searching, sorting, inserting, deleting, and so on
- Without a common interface, you would have to do the same things to different data structures in different ways.
- This would be a sad state of affairs

What's an Iterator

- Iterator is a concept that allows two parties–"client code", and "library code"–to communicate without concern for the other's internal details
- Intent of iterator
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- An iterator should allow its consumers to:
  - Move to the beginning of the range of elements
  - Advance to the next element
  - Return the value referred to, often called the referent
  - Interrogate it to see if it is at the end of the range

How C++ Meets Iterator Requirements

- Move to the beginning of the range of elements
  - Providing a begin member function, which returns an iterator that refers to the first element in the container, if it exists.
- Advance to the next element
  - Using Pointer semantics. The ++ operator (both prefix and postfix forms) moves the iterator to the next element. (Some categories of iterators also support − −, -=, +=, +, and -.)
- Return the value referred to, often called the referent
  - Using pointer semantics. The dereference operator * is used to return the referent.
- Interrogate it to see if it is at the end of the range
  - Providing an end member function on all standard containers that returns an iterator referring to one past the end of the container. **Dereferencing this iterator yields undefined behavior, but you can compare the value returned by end to the current iterator to test if you are done iterating through the range**.

C++ Iterator Usage Example

Example 1:

```
void printCont(const vector<int>& v)
{
    vector<int>::const_iterator  it;
    for (it = v.begin(); it != v.end(); ++it)
        cout < <  *it < <  endl;
}
```

Example 2:

```
list<string> x;
x.push_back("peach");
x.push_back("apple");
x.push_back("banana");
// Make y the same size as x. This creates empty strings in y's elements.
list<string> y(x.size());
// This is std::copy...
copy(x.begin(), x.end(), y.begin());
```

C++ Iterator Categories

- An iterator category is a set of requirements that defines a certain type of behavior.
- Five different categories of iterators: input, output, forward, bidirectional, and random access.

Table 2. Iterator categories and required member functions

Member Function	Input	Output	Forward	Bidirectional	Random Access
`Iter x(y)`	Y	Y	Y	Y	Y
`Iter x = y`	Y	Y	Y	Y	Y
`x == y`	Y	N	Y	Y	Y
`x != y`	Y	N	Y	Y	Y
`x++, ++x`	Y	Y	Y	Y	Y
`x--, --x`	N	N	N	Y	Y
`*x`	As rvalue	As lvalue	Y	Y	Y
`(*x).f`	Y	N	Y	Y	Y
`x->f`	Y	N	Y	Y	Y
`x + n`	N	N	N	N	Y
`x += n`	N	N	N	N	Y
`x - n`	N	N	N	N	Y
`X -= n`	N	N	N	N	Y
`X[n]`	N	N	N	N	Y

C++ Containers

- Different standard containers offer different types of iterators, depending on what can be efficiently supported, based on the type of data structure that is used by the container
- For example, a list provides bidirectional iterators because lists are usually implemented as a doubly-linked list, which makes it efficient to iterate forward and backward one element at a time. list does not provide random access iterators, though, not because it's impossible to implement, but because it can't be implemented efficiently
- Each standard container supports the category of iterator it can implement efficiently
- A list of the standard containers and the category of iterator each supports.

Table 3. Iterator categories for standard containers

Container	Iterator Category
`basic_string`	Random access
`deque`	Random access
`list`	Bidirectional
`map, multimap`	Bidirectional
`set, multiset`	Bidirectional
`vector`	Random access

Homemade Iterators

- Since a C++ iterator provides a familiar, standard interface, at some point you will no doubt want to add one to your own classes.
- Homemade iterator allows you to plug-and-play with standard library algorithms and it lets consumers of your data structure use the same iterator interface they're probably already familiar with.
- Before you begin implementing your own iterator, you should decide which category you can support.
- Use the same design parameter as the standard library and only support an iterator category that you can implement efficiently.
- If you want to use homemade iterators in STL functions defined in <algorithm>, inherit it from the class iterator defined in <iterator>
- See example SQueue.h, SQueue.cpp.

C++ Class Review

CSCI 3110

Exception


```
int Quotient ( int numer, int denom )
{
        if ( denom != 0)
                return numer / denom;
        else
                // What to do??
}
```

- Do nothing and silently return to the calling code
- Print an error message, and return an arbitrary integer value
- Return a special value such as -9999 to indicate an error
- Rewrite the function with a 3rd parameter to indicate success or failure
- Print an error message and halt the program

Exception

- An exception is an unusual event, often an error that occurs during the execution of a program that disrupts the normal flow of instructions.
- A function can indicate that an error has occurred by throwing an exception
- Exception handler: The code that deals with the exception

Throw Exceptions

- syntax:
  throw Expression ; // throw an exception
  // throw is a C++ reserved word
  // Expression can be a constant value, variable, or object

- Examples:
  throw 5; // The constant value 4 is thrown
  throw x; // The value of the variable x is thrown
  throw str; // The object str is thrown
  throw string("Exception Found"); // An anonymous string object with the string "Exception Found" is thrown
  throw UserDefinedException(); // An anonymous UserDefinedException object is thrown

- Notes: If an anonymous object is thrown, a constructor is invoked to initialize it.

Exception

try-catch statement:

```
try {
        statements or function call that may throw an exception
} catch ( ExceptionType1 variableName1 ) {
        exception handler 1
} catch ( ExceptionType2 variableName2 ) {
        exception handler 2
}
...
} catch ( ExceptionTypen variableNamen ) {
        exception handler n

}
```

Exception

- If try-clause statements do not throw an exception, control transfers to the statement following the entire try-catch statement.
- If an exception is thrown by a statement in the try-clause
  - The remaining statements in that *try* block are ignored.
  - The program searches the *catch* blocks in the order they appear after the *try* block and looks for an appropriate exception handler – The type of thrown exception matches the parameter type in the *catch* block.
  - The control jumps to first statement of the matched exception handler
  - Other remaining *catch* blocks are ignored.
- If an exception thrown by a try-clause statement is not catched by corresponding catch statement(s), the function returns **IMMEDIATELY** and pass the exception back to its **caller**. This sequence of returns continues back through the chain of function calls until either a matching exception handler is found or control reaches *main*. If *main* fails to catch it, the system terminates the program.

C++ Predefined Exceptions

Type	Header
exception	stdexcept, exception
logic_error	stdexcept
length_error	stdexcept
domain_error	stdexcept
out_of_range	stdexcept
invalid_argument	stdexcept, bitset
runtime_error	stdexcept
range_error	stdexcept
overflow_error	stdexcept
underflow_error	stdexcept
bad_alloc	new
bad_cast	typeinfo
bad_typeid	typeinfo
bad_exception	exception
ios_base::failure	ios, iostream, fstream

Exception

- To indicate the exceptions that will be thrown by a function, include a **throw** clause in the functions's header:

```
class DivByZero // Exception class
{};

int Quotient ( int numer, int denom ) throw(DivByZero)
{
        if ( denom != 0)
                return numer / denom;
        else
                throw DivByZero();
}
```

Exception


```
int main ()
{       ...
        try
        {
                int x = Quotient ( numer1, denom1 );
        }
        catch ( DivByZero anObject )
                cout < <  "*** Denominator can't be 0" < <  endl;
        }
        ...
}
```

For more exception examples, see exception.cpp

Advantages of Exception

- Separating Error-Handling Code from "Regular" Code

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

- Propagating Errors Up the Call Stack