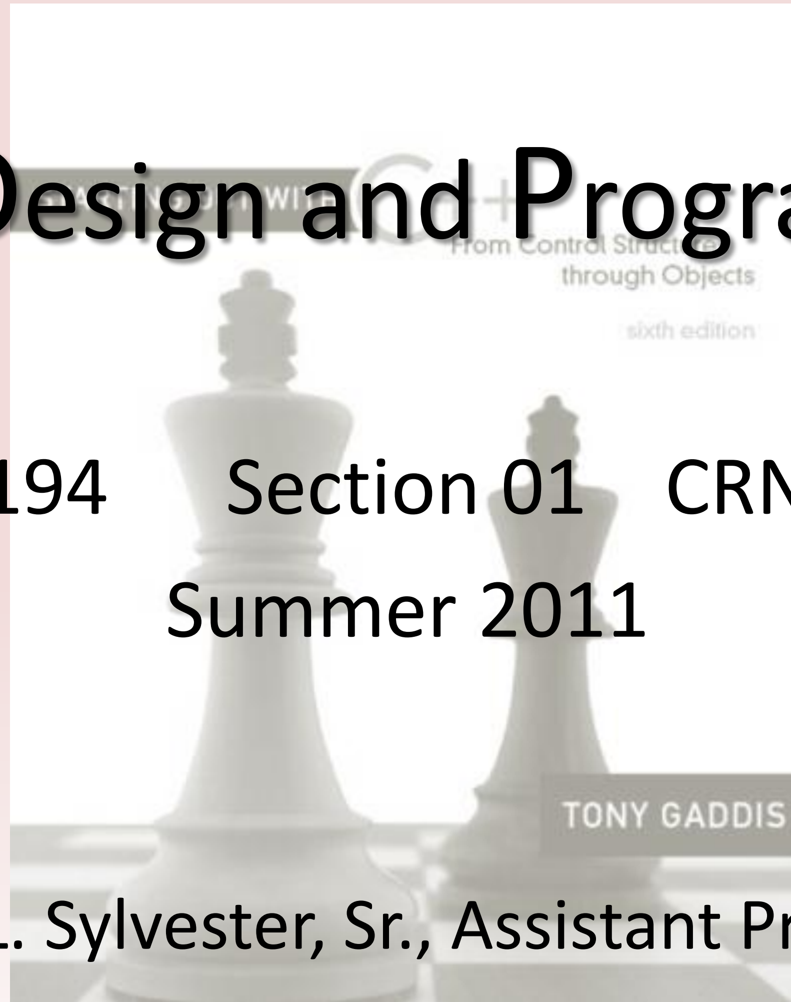
System Design and Programming II



CSCI – 194 Section 01 CRN: 32476

Summer 2011

David L. Sylvester, Sr., Assistant Professor

Chapter 16

Exceptions, Templates, and the Standard Template Library (STL)

Exceptions

Exceptions are used to signal errors or unexpected events that occur while a program is running.

Error testing is usually the strateforward process involving IF statements or other control mechanisms.

Ex:
```
if (denominator  == 0)
        cout << “Error:  cannot divide by zero.\n”;
else
         quotient  = numerator / denominator;
```

But what if similar code is part of a function that returns the quotient.

Ex:

```
// An unreliable division function
double divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
            cout << “ERROR:  Cannot divide by zero.\n”;
            return 0;
     }
            else
                return static_cast<double>(numerator) / denominator;
}
```

Functions commonly signal error conditions by returning a predefined value. This example returns 0 when division by zero is attempted. Even though the function displays an error message, the part of the program that calls the function will not know when an error has occurred. Problems like these require sophisticated error handling techniques.

Throwing an Exception

One way of handling complex error conditions is with exceptions. An exception is a value or an object that signals an error. When the error occurs, an exception is "thrown." The following example of the previous code is modified to throw the exception when division by zero has be attempted.

Ex:

```
// An unreliable division function
double divide(int numerator, int denominator)
{
    if (denominator == 0)
        throw "ERROR:  Cannot divide by zero.\n";
    else
        return static_cast<double>(numerator) / denominator;
}
```

Causes the exception to be thrown

The **throw** key word is followed by an argument, which can be any value. The line containing a throw statement is known as the throw point. When a throw statement is executed, control is passed to another part of the program known as an exception handler. When an exception is thrown by a function, the function aborts.

Handling and Exception

To handle an exception, a program must have a **try/catch** construct.

Ex:

```
try
{
     // code here calls functions or object member
     // functions that might throw an exception.
}
catch(exceptionParameter)
{
      // code here handles the exception
}
// repeat as many catch blocks as needed.
```

The first part of the construct is the try block. This starts with the key word try and is followed by a block of code executing any statements that might directly or indirectly cause an exception to be thrown. The try block is immediately followed by one or more catch blocks, which are the exception handlers. A catch block starts with the key word catch, followed by a set of parentheses containing the definition of an exception parameter.

Ex:

```
try
{
      quotient = divide(num1, num2);
      cout << "The quotient is " << quotient << endl;
}
catch(char *exceptionString)
{
       cout << exceptionString);
}
```

Because the divide function throws an exception whose value is a string, there must be an exception handler that catches a string. The catch block shown catches the error message in the exceptionString parameter, and then displays it with cout.

Sample Program

```
// This program demonstrates an exception
// being thrown and caught.
#include <iostream>
using namespace std;

// Function prototype
double divide(int, int);

int main()
{
  int num1, num2;  // To hold two numbers
  double quotient; // To hold the quotient of the numbers

  // Get two numbers.
  cout << "Enter two numbers: ";
  cin >> num1 >> num2;

  // Divide num1 by num2 and catch any
  // potential exceptions.
  try
  {
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
  }
  catch (char *exceptionString)
  {
    cout << exceptionString;
  }

  cout << "End of the program.\n";
  return 0;
}
```

```
//*****************************************
// The divide function divides numerator by        *
// denominator. If denominator is zero, the        *
// function throws an exception.                   *
//*****************************************

double divide(int numerator, int denominator)
{
  if (denominator == 0)
    throw "ERROR: Cannot divide by zero.\n";

  return static_cast<double>(numerator) / denominator;
}
```

If the exception is a string, the program jumps to this catch clause

After the catch block is finished, the program resumes here.

What if an Exception is Not Caught

There are two possible ways for a thrown exception to go uncaught. The first possibility is for the try/catch construct to contain no catch blocks with an exception parameter of the right data type. The second possibility is for the exception to be thrown from outside a try block. In either case, **the exception will cause the entire program to abort execution.**

Object-Oriented Exception Handling

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"
//*********************************************
// setWidth sets the value of the member variable width.   *
//*********************************************

void Rectangle::setWidth(double w)
{
  if (w >= 0)
    width = w;
  else
    throw NegativeSize();
}

//*********************************************
// setLength sets the value of the member variable length.  *
//*********************************************

void Rectangle::setLength(double len)
{
  if (len >= 0)
    length = len;
  else
    throw NegativeSize();
}
```

In the setWidth function, the parameter w is tested by the if statement. If w holds a negative number, the statement **throw NegativesSize();** is executed.

This way of reporting errors is much more graceful than simply aborting the program. Any code that uses the Rectangle class must simply have a catch block to handle the negativeSize exceptions that the Rectangle class might throw.

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
  int width;
  int length;

  // Create a Rectangle object.
  Rectangle myRectangle;

  // Get the width and length.
  cout << "Enter the rectangle's width: ";
  cin >> width;
  cout << "Enter the rectangle's length: ";
  cin >> length;

  // Store these values in the Rectangle object.
  try
  {
    myRectangle.setWidth(width);
    myRectangle.setLength(length);
    cout << "The area of the rectangle is "
         << myRectangle.getArea() << endl;
  }
  catch (Rectangle::NegativeSize)
  {
    cout << "Error: A negative value was entered.\n";
  }
  cout << "End of the program.\n";

  return 0;
}
```

Multiple Exceptions

When writing programs, you may have the need to test several different types of errors and signal which one has occurred. C++ allows you to throw and catch multiple exceptions. The only requirement is that each different exception be of a different type. You then code a separate catch block for each type of exception that may be thrown in the try block.

The previous example only tested for a negative number with no specifications as to what value was negative. To expand the rectangle class so it throws one type of exception when a negative value is specified for the width, and another type of exception when a negative value is specified for the length, we would first declare two different exception classes.

Ex:

```
// Exception class for a negative width
class NegativeWidth
        { };

// Exception class for a negative length
class NegativeLength
        { };
```

Sample of Multiple Exceptions

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"

//*******************************************************
// setWidth sets the value of the member variable width.    *
//*******************************************************

void Rectangle::setWidth(double w)
{
  if (w >= 0)
    width = w;
  else
    throw NegativeWidth();
}

//*******************************************************
// setLength sets the value of the member variable length.  *
//*******************************************************

void Rectangle::setLength(double len)
{
  if (len >= 0)
    length = len;
  else
    throw NegativeLength();
}
```

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{    int width;
  int length;

  // Create a Rectangle object.
  Rectangle myRectangle;

  // Get the width and length.
  cout << "Enter the rectangle's width: ";
  cin >> width;
  cout << "Enter the rectangle's length: ";
  cin >> length;

  // Store these values in the Rectangle object.
  try
  {
    myRectangle.setWidth(width);
    myRectangle.setLength(length);
    cout << "The area of the rectangle is "
         << myRectangle.getArea() << endl;
  }
  catch (Rectangle::NegativeWidth)
  {
    cout << "Error: A negative value was given "
         << "for the rectangle's width.\n";
  }
  catch (Rectangle::NegativeLength)
  {
    cout << "Error: A negative value was given "
         << "for the rectangle's length.\n";
  }
  cout << "End of the program.\n";
  return 0;  }
```

Function Templates

A function template is a “generic” function that can work with any data type. The programmer writes the specifications of the function, but substitutes parameters for data types. When the compiler encounters a call to the function, it generates code to handle the specific data types(s) used in the call.

Overloaded functions make programming convenient because only one function name must be remembered for a set of functions that perform similar operations. Each of the functions, must still be written individually, even if they perform the same operations.

Ex:

```
int square(int number)
{
    return number * number;
}
```

```
double square(double number)
{
    return number * number;
}
```

The only differences between these two functions are the data types of their return values and there parameters.

In situations like this, it is more convenient to write a function template than an overloaded function. Function templates allow you to write a single function definition that works with many different data types, instead of having to write a separate function for each data type used.

A function template is not an actual function, but a “mold” the compiler uses to generate one or more functions. When writing a function template, you do not have to specify actual types for the parameters, return value, or local variables. Instead, you use a type parameter to specify a generic data type. When the compiler encounters a call to the function, it examines the data types of its arguments and generates the function code that will work with those data types.

Ex:

```
template <class T>
T square(T number)
{
	return number * number;
}
```

The beginning of a function template is marked by a template prefix, which begins with the keyword template. Next is a set of angled brackets that contains one or more generic data types used in the template. A generic data type starts with the key word class followed by a parameter name that stands for the data type. The example only uses one parameter name, which is named T. If there are more parameters, you would list them separated by a comma. After this, the function definition is written as usual, except the type parameter are substituted for the actual data type names.

Ex:

```
T square(T number)
```

T is the type parameter or generic data type. The header defines square as a function that returns a value of type T and uses a parameter called number, which is also of type T. The compiler examines each call to square and fills in the appropriate data type for T.

Sample Template Program

```
// This program uses a function template.
#include <iostream>
#include <iomanip>
using namespace std;

// Template definition for square function.
template <class T>
T square(T number)
{
   return number * number;
}

int main()
{
   int userInt;        // To hold integer input
   double userDouble;  // To hold double input

   cout << setprecision(5);
   cout << "Enter an integer and a floating-point value: ";
   cin >> userInt >> userDouble;
   cout << "Here are their squares: ";
   cout << square(userInt) << " and "
        << square(userDouble) << endl;
   return 0;
}
```

Square is being called with int parameter and returns an int

Square is being called with double parameter and returns a double

Because the compiler encountered two calls to square, each with a different parameter type, it generated the code for two instances of the function: one with and int parameter and int return type, the other with a double parameter and double return type.

Notice that the template appears before the all calls to square. As with regular functions, the compiler must already know the template's contents when it encounters a call to the template function. **Note: Templates should be placed near the top of the program or in a header file.**

Sample Template Program (by Reference)

```
// This program demonstrates the swapVars function
template.
#include <iostream>
using namespace std;

template <class T>
void swapVars(T &var1, T &var2)
{
  T temp;

  temp = var1;
  var1 = var2;
  var2 = temp;
}

int main()
{
  char firstChar, secondChar;      // Two chars
  int firstInt, secondInt;         // Two ints
  double firstDouble, secondDouble; // Two doubles

  // Get and swapVars two chars
  cout << "Enter two characters: ";
  cin >> firstChar >> secondChar;
  swapVars(firstChar, secondChar);
  cout << firstChar << " " << secondChar << endl;

  // Get and swapVars two ints
  cout << "Enter two integers: ";
  cin >> firstInt >> secondInt;
  swapVars(firstInt, secondInt);
  cout << firstInt << " " << secondInt << endl;

  // Get and swapVars two doubles
  cout << "Enter two floating-point numbers: ";
  cin >> firstDouble >> secondDouble;
  swapVars(firstDouble, secondDouble);
  cout << firstDouble << " " << secondDouble << endl;
  return 0;
}
```

Introduction to the Standard Template Library (STL)

The Standard Template Library contains many templates for useful algorithms and data structures. This library contains numerous generic templates for implementing abstract data types (ADTs) and algorithms.

The most common data structures in the STL are containers and iterators. A container is a class that stores data and organizes it in some fashion. An iterator is an object that behaves like a pointer it is used to access the individual data elements in a container.

There are two type of container classes in the STL: sequence and associative.

Sequence Containers	
Container Name	Description
vector	An expandable array. Values may be added to or removed from the end or middle of a vector.
deque	Like a vector, but allows values to be added to or removed from the front.
list	A doubly linked list of data elements. Values may be inserted to or removed from any position.

Performance Differences Between vectors, deques and lists

There is a difference in performance between vectors, queues, and lists. When choosing one of these templates to use in your program, remember the following points:

- A vector is capable of quickly adding values to its end. Insertion to other points are not so efficient.
- A deque is capable to quickly adding values to its front and its end. deques are not efficient at inserting values at other positions, however
- A list is capable of quickly inserting values anywhere in its sequence. lists do not, however, provide random access.

An associative container uses keys to rapidly access elements. If you have eve used a relational database, you are probably familiar with the concept of keys.

Associative Containers	
Container Name	Description
set	Stores a set of keys. No duplicate values are allowed
multiset	Stores a set of keys. Duplicates are allowed.
map	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed.
multimap	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed

Types of Iterators	
Iterator type	Description
Forward	Can only move forward in a container (uses the ++ operator)
Bidirectional	Can move forward or backward in a counter (uses the ++ and – operators).
Random-access	Can move forward and backward, and can jump to a specific data element in a container
Input	Can be use with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

Algorithms	
Algorithms	Description
binary_search	Performs a binary search for an object and returns true if the object is found. Ex: **binary_search(iter1, iter2, value);**
count	Returns the number of time a value appears in a range. Ex: **iter3 = count(iter1, iter2, value);**
Find	Finds the first object in a container that makes a value and returns an iterator to it. Ex: **iter3 = find(iter1, iter2, value);**
for_each	Executes a function for each element in a container. Ex: **for_each(iter1, iter2, func);**
max_element	Returns an iterator to the largest object in a range. Ex: **iter3 = max_element(iter1,iter2);**
min_element	Returns an iterator to the smallest object in a range. Ex: **iter3 = min_elemen(iter1, iter2);**
random_shuffle	Randomly shuffles the elements of a container. Ex: **ramdom_shuffle(iter1, iter2);**
Sort	Sorts a range of elements. Ex: **sort(iter1, iter2);**

Member Functions of Vectors	
Member Function	Description
size( )	Returns the number of elements in a vector.
push_back( )	Accepts as an argument a value to be inserted into the vector. The argument is inserted after the last element.
pop_back( )	Removes the last element from the vector.
operator[ ]	Allows array-like access of existing vector elements. The vector must already contain elements for this operator to work. It cannot be used to insert new values into the vector.

Iterators

```
// This program provides a simple iterator demonstration.
#include <iostream>
#include <vector>    // Include the vector header
using namespace std;

int main()
{
  int count;  // Loop counter

  // Define a vector object
  vector<int> vect;

  // Define an iterator object
  vector<int>::iterator iter;

  // Use push_back to push values into the vector.
  for (count = 0; count < 10; count++)
  vect.push_back(count);

  // Step the iterator through the vector,
  // and use it to display the vector's contents.
  cout << "Here are the values in vect: ";
  for (iter = vect.begin(); iter < vect.end(); iter++)
  {
    cout << *iter << " ";
  }

  // Step the iterator through the vector backwards.
  cout << "\nand here they are backwards: ";
  for (iter = vect.end() - 1; iter >= vect.begin(); iter--)
  {
    cout << *iter << " ";
  }
  return 0;
}
```

Iterators may be used to access and manipulate container elements. The definition of an iterator is closely related to the definition of the container it is to be used with.

- defines the vector as integer
- Creates an iterator specifically for a vector of ints
- Allows the iterator to step through the vector
- Iterator points to first element