Exceptions

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

- An **exception** is a condition that occurs at execution time and makes normal continuation of the program impossible.

- When an exception occurs, the program must either terminate or jump to special code for handling the exception.

- The special code for handling the exception is called an **exception handler.**

- **Example: unreliable division function:**

```
double divide(double numerator, double denominator)
{
   if (denominator = = 0)
   {
      cout < < "Error: Can not divide by zero!" < < endl;
      return 0;          //this is a problematic error signal
   }
   else
      return (numerator/denominator);
}
```

- **Note:** 0 is problematic as an error signal since 0 is a valid result of a division operation.

- **Exceptions - Key Words:**
  - **throw**
    followed by an argument, is used to signal an exception
  - **try**
    followed by a block { }, is used to invoke code that throws an exception.
  - **catch**
    followed by a block { }, is used to process exceptions thrown in the preceding **try** block. **catch** takes a parameter that matches the type thrown.

- **Sample Program:**

```
// This program illustrates exception handling.
#include <iostream>
using namespace std;

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

int main()
{
   int num1, num2;
   double quotient;

   cout << "Enter two numbers: ";
   cin >> num1 >> num2;
   try
   {
      quotient = divide(num1, num2);
      cout << "The quotient is " << quotient << endl;
   }
   catch (char *exceptionString)
   {
      cout << exceptionString;
   }

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

double divide(double numerator, double denominator)
{
   static char *str = "ERROR: Cannot divide by zero.\n";

   if (denominator == 0)
      throw str;
   else
      return (numerator/denominator);
}
```

Object Oriented Exception Handling with Classes

Throwing an Exception Class

- Instead of throwing a character string or some other value of a primitive type, the following program **throws an exception class.**

- The throw statement **causes an instance of the exception class to be created**. All that remains is for the catch block to handle the exception.

- **Class Definition File:**

```
class IntRange
{
  private:
    int input;                      // For user input
    int lower;                      // Lower limit of range
    int upper;                      // Upper limit of range
  public:
    // Exception class
    class OutOfRange
       { };                         // Empty class declaration

    // Member functions
    IntRange(int low, int high)     // Constructor
    {
       lower = low;
       upper = high;
    }

    int getInput()
    {
      cin > > input;
      if (input < lower || input > upper)
         throw OutOfRange();        //throw an exception class
      return input;
    }
};
#endif
```

- **Sample Program that Throws an Exception Class:**

```
// This program demonstrates the use of object-oriented
// exception handling.
#include <iostream>
#include "IntRange.h"
using namespace std;

int main()
{
   IntRange range(5, 10);
   int userValue;

   cout << "Enter a value in the range 5 - 10: ";
   try
   {
      userValue = range.getInput();
      cout << "You entered " << userValue << endl;
   }
   catch (IntRange::OutOfRange)
   {
      cout << "That value is out of range.\n";
   }

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

Multiple Exceptions

- In many cases a program will need to test for 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.

```
#ifndef INTRANGE2_H
#define INTRANGE2_H

#include <iostream>
using namespace std;

class IntRange2
{
  private:
    int input;                          // For user input
    int lower;                          // Lower limit of range
    int upper;                          // Upper limit of range
  public:
    // Exception classes.
    class TooLow
        { };
    class TooHigh
        { };
    // Member functions.
    IntRange2(int low, int high)    // Constructor
    {
         lower = low;
         upper = high;
    }
    int getInput()
    {
        cin >> input;
        if (input < lower)
            throw TooLow();
        else if (input > upper)
            throw TooHigh();
        return input;
    }
};
#endif
```

- **Sample Program that Throws Multiple Exceptions:**

```
// This program demonstrates the IntRange2 class.
#include <iostream>
#include "IntRange2.h"
using namespace std;

int main()
{
   IntRange2 range(5, 10);
   int userValue;

   cout << "Enter a value in the range 5 - 10: ";
   try
   {
      userValue = range.getInput();
      cout << "You entered " << userValue << endl;
   }
   catch (IntRange2::TooLow)
   {
      cout << "That value is too low.\n";
   }
   catch (IntRange2::TooHigh)
   {
      cout << "That value is too high.\n";
   }

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

Extracting Information from the Exception Class

- Sometimes we may want an exception to pass information back to the exception handler. This can be accomplished by giving the exception class **member variables** in which the information can be stored.

```
#ifndef INTRANGE3_H
#define INTRANGE3_H

#include <iostream>
using namespace std;

class IntRange3
{
  private:
    int input;                          // For user input
    int lower;                          // Lower limit of range
    int upper;                          // Upper limit of range
  public:
    // Exception class.
    class OutOfRange
    {
      public:
        int value;
        OutOfRange(int i)               // Exception class constructor
        {  value = i; }
    };
    // Member functions.
    IntRange3(int low, int high)        // Constructor
    {
        lower = low;
        upper = high;
    }
    int getInput()
    {
        cin >> input;
        if (input < lower || input > upper)
            throw OutOfRange(input);      //pass info to the handler
        return input;
    }
};
#endif
```

- **Sample Program that Passes Info to the Exception Handler:**

```
// This program demonstrates the IntRange3 class.
#include <iostream>
#include "IntRange3.h"
using namespace std;

int main()
{
   IntRange3 range(5, 10);
   int userValue;

   cout << "Enter a value in the range 5 - 10: ";
   try
   {
      userValue = range.getInput();
      cout << "You entered " << userValue << endl;
   }
   catch (IntRange3::OutOfRange ex)
   {
      cout << "That value " << ex.value
           << " is out of range.\n";
   }

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

Unwinding the Stack

- If an exception is thrown in a try block that has a catch block capable of handling the exception, control transfers from the throw point to the catch block. Assuming the catch block runs to completion, execution will continue at the first statement after the sequence of catch blocks attached to the try.

- If the function does not contain a catch block capable of handling the exception, control passes out of the function, and **the exception is automatically re-thrown at the point of the call in the calling function**. (See an example of a nested try block on the next page.)

- By this process, **an exception can propagate backwards along the chain of function calls until the exception is thrown out of a try block that has a catch block than can handle it.**

- If no such try block is ever found, the exception will eventually be thrown out of the main function, causing the program to be terminated.

- This process of propagating un-caught exceptions from a function to its caller is called unwinding the stack of function calls.

Re-throwing an Exception

- **Example of Nested try blocks:**

```
// This program demonstrates a nested try block.
#include <iostream>
using namespace std;

int main()
{
   try
   {
      doSomething();
   }
   catch (exception1)
   {
      code to handle exception1
   }
   catch (exception2)
   {
      code to handle exception2
   }
   return 0;
}

void doSomething()
{
   try
   {
      code that can throw exceptions 1, 2 and 3
      exception2 will be handled by main()
   }
   catch (exception1)
   {
      throw;              //re-throw exception1
   }
   catch (exception3)
   {
      code to handle exception3
   }
}
```

Function Templates

- A **function template** is a **generic** function that can work with different data types.

- The programmer writes the specifications of the functions, but substitues parameters for data types.

- When the compiler encounters a call to the function, it generates code to handle the specific data type(s).

- Function templates are better than overloaded functions, since **the code defining the algorithm of the function is only written once**.

- **Sample code:**

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

- **Function Template:**

```
template <class T>          //template prefix
T square(T number)          //T is a generic data type
{
   return (number * number);
}
```

- A template prefix begins with the **keyword template**, followed by **angle brackets containing one or more generic data types** (separated bo commas) used in the template. After this, the function definition is written as usual except the generic type parameters are used instead of the actual data type names.

- **Sample 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()
{
   cout << setprecision(5);

   // Get an integer and compute its square.
   cout << "Enter an integer: ";
   int iValue;
   cin >> iValue;

   // The compiler creates int square(int) at the first
   // occurrence of a call to square with an int argument.
   cout << "The square is " << square(iValue) << endl;

   // Get a double and compute its square.
   cout << "\nEnter a double: ";
   double dValue;
   cin >> dValue;

   // The compiler creates double square(double)at the first
   // occurrence of a call to square with a double argument.
   cout << "The square is " << square(dValue) << endl;

   return 0;
}
```

The swap Function Template

- A generic library swap function template exists and can be invoked by using #include <algorithm>.

- **Sample Program:**

```
#include <iostream>
#include <string>
#include <algorithm>          //needed for swap
using namespace std;

int main()
{
   // Get and swap two chars.
   char firstChar, secondChar;
   cout << "Enter two characters: ";
   cin >> firstChar >> secondChar;
   swap(firstChar, secondChar);
   cout << firstChar << " " << secondChar << endl;

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

   // Get and swap two strings.
   string firstString, secondString;
   cout << "Enter two strings: ";
   cin >> firstString >> secondString;
   swap(firstString, secondString);
   cout << firstString << " " << secondString << endl;

   return 0;
}
```

Using Operators in Function Templates

- Always remember that templates will only work with types that support the operations used by the template.

- **Sample Program:**

```
// This program illustrates the use of function templates.
#include <string>
#include <iostream>
using namespace std;

// Template for minimum of an array.
template <class T>
T minimum(T arr[ ], int size)
{
   T smallest = arr[0];
   for (int k = 1; k < size; k++)
   {
      if (arr[k] < smallest)
         smallest = arr[k];
   }
   return smallest;
}

int main()
{
   // The compiler creates int minimum(int [], int)
   // when you pass an array of int.
   int arr1[] = {40, 20, 35};
   cout << "The minimum number is " << minimum(arr1,3)
      << endl;

   // The compiler creates string minimum(string [], int)
   // when you pass an array of string.
   string arr2[] = {"Zoe", "Snoopy", "Bob", "Waldorf"};
   cout << "The minimum string is " << minimum(arr2, 4)
      << endl;

   return 0;
}
```

Function Templates with Multiple Types

- **Sample Program:**

```
// This program illustrates the use of function templates
// with multiple types.
#include <iostream>
#include <string>
using namespace std;

//template function
template <class T1, class T2, class T3>
void echoAndReverse(T1 a1, T2 a2, T3 a3)
{
   cout << "Original order is: "
      << a1 << " " << a2 << " " << a3 << endl;
   cout << "Reversed order is: "
      << a3 << " " << a2 << " " << a1 << endl;
}

int main()
{
   echoAndReverse("Computer", 'A', 18);

   echoAndReverse("One", 4, "All");

   return 0;
}
```

Overloading Function Templates

- Function templates can be overloaded.

- **Sample Program:**

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

template <class T>
T sum(T val1, T val2)
{
   return val1 + val2;
}

template <class T>
T sum(T val1, T val2, T val3)
{
   return val1 + val2 + val3;
}

int main()
{
   double num1, num2, num3;

   cout << "Enter two values: ";
   cin >> num1 >> num2;
   cout << "Their sum is " << sum(num1, num2) << endl;

   cout << "Enter three values: ";
   cin >> num1 >> num2 >> num3;
   cout << "Their sum is " << sum(num1, num2, num3) <<endl;

   return 0;
}
```

Notes on Function Templates

- A function template is a pattern (or prototype).
- No actual code is generated by the compiler until the function named in the template is called in the code.
- A function template uses no memory.
- When passing a class object to a function template, ensure that all operators referred to in the template are defined or overloaded in the class definition.
- All data types specified in a template prefix must be used in the template definition.
- Function calls must pass parameters for all data types specified in the template prefix.
- Function templates can be overloaded – need different parameter lists.
- Like regular functions, function templates must be defined before being called.
- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix,
  - convert data type names in the function to a type parameter (i.e., a T type) in the template.

Class Templates

- Templates can be used to create generic classes and abstract data types.

- Unlike functions, a class template is instantiated by supplying the type name (int, float, string, etc.) at object definition.

- **Sample Class Template:**

```
template <class T>
class Joiner
{
  public:
    T combine(T x, T y)
    {
       return (x + y);
    }
};
```

- **Example: Using Class Template:**

```
Joiner<double> jd;    //instantiates an object of type double

Joiner<string> sd;    //instantiates an object of type string

cout << jd.combine(3.0, 5.0);          //prints: 8.0

cout << sd.combine("Hi ", "Ho");       //prints: Hi Ho
```

Class Templates and Inheritance

- Templates can be combined with inheritance.

- You can derive:
  - Non template classes from a template class: instantiate the base class template and then inherit from it.
  - A template class from a template class
  - Other combinations possible

The Standard Template Library (STL)

- The Standard Template Library (STL) is a library containing templates for frequently used data structures and algorithms.

- The most important data structires included are **containers** and **iterators**.

- A **container** is a class that stores data and organizes it in some fashion.

- An **iterator** is an object that works like a pointer and allows access to items stored in containers.

- **Sequence Containers:**
  Organize and access data sequentially, as in an array:
  - vector
    A sequence of items implemented as an array. Items can be efficiently added and removed from the vector at its end.
  - deque
    A sequence of items that has a front and a back. Items can be efficiently added and removed from the ends.
  - list
    A sequence of item that allows quick additions and removals from any position.

- **Associative Containers:**
  Use keys to allow data elements to be quickly accessed:
  - set
    Stores a set of keys. No duplicate values allowed.
  - multiset
    Stores a set of keys. Duplicate values allowed.
  - map
    Maps a set of keys to data elements. Each key is associated with a unique data element. No duplicates.
  - multimap
    Maps a set of keys to data elements. The same key may be associated with multiple values.

- **Iterators:**
  Iterators are a generalization of pointers used to access information in containers

- **Iterator Types:**
  - forward
    can only move forward in a container (uses + + operator)
  - bidirectional
    can only move forward or backward in a container (uses + + and -- operators)
  - random-access
    can move forward and backward, and can jump to a specific data element in a container.
  - input
    can be used with cin to read information from an input device or a file.
  - output
    can be used with cout to write information to an output device or a file.

```
// This program is a simple demonstration of the vector STL template.
#include <iostream>
#include <vector>       // needed for vectors (see text for details)
using namespace std;

int main()
{
   vector<int> vect;       // Create a vector of int

   for (int x = 0; x < 10; x++)
      vect.push_back(x*x);

   //print everything using iterators.
   vector<int>::iterator iter = vect.begin();
   while (iter != vect.end())
   {
      cout << *iter << " ";
      iter ++;
   }
   return 0;
}
```

STL Algorithms

- STL contains algorithms implemented as function templates to perform operations on containers.

- Requires algorithm header file (#include <algorithm>)

- Collection of algorithms includes (see text for details):
  - boolVar = binary_search(iter1, iter2, value)
    Performs a binary search for an object (value) in a container in a range of elements iter1 to iter2. Returns true if found; otherwise returns fasle.
  - number = count(iter1, iter2, value)
    Returns the number of times a value appears in a container in the range of elements iter1 to iter2.
  - for_each(iter1, iter2, func)
    Executes a specified function (func) for each element in a container in the range of elements iter1 to iter2.
  - iter3 = find(iter1, iter2, value)
    Finds the first object in a container that matches a value. If value is found, the function eturns an iterator to it; otherwise, it returns the iterator iter2.
  - iter3 = max_element(iter1, iter2)
    Finds max element in the portion of a container delimited by the range of elements iter1 to iter2. Returns an iterator.
  - iter3 = min_element(iter1, iter2)
    Finds min element in the portion of a container delimited by the range of elements iter1 to iter2. Returns an iterator.
  - random_shuffle(iter1, iter2)
    Randomly reorders the portion of the container in the given range of elements iter1 to iter2. Returns an iterator
  - sort(iter1, iter2)
    Sorts into ascending order the portion of the container specified by the given range of elements iter1 to iter2.