Chapter 16

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

Assert

- `#include <cassert>`
- `assert( boolean expression );`
- If the value of the expression is false, assert prints an error message and calls function `abort`
Assert Example

```c
int totalDays( int days, int weeks )
{
    assert ( ( days < 0 ) || ( days > 7 ) )
    return ( 7 * weeks + days );
} // totalDays
```

Exceptions

- Up to now, we have checked for errors by placing if statements that checked for the condition or by using assert statements
- Exceptions give us more flexibility to deal with errors
Exception

- Indicates that something unexpected has occurred or been detected
- Allows the program to deal with the problem in a controlled manner
- Can be as simple or as complex as the program design requires

Exceptions – Terminology

- **Exception**
  - Object or value that signals an error
- **Throw an exception**
  - Send a signal that an error has occurred
- **Catch/Handle an exception**
  - Process the exception
  - Interpret the signal
Exceptions – Key Words

- **throw**
  - Followed by an argument, it is used to throw an exception

- **try**
  - Followed by a block {}, it is used to invoke code that throws an exception

- **catch**
  - Followed by a block {}, it is used to detect and process exceptions thrown in preceding try block
  - Takes a parameter that matches the type thrown

Exceptions – Flow of Control

1. A function that throws an exception is called from within a try block
2. If the function throws an exception, the function terminates and the try block is immediately exited
   A catch block, to process the exception, is searched for in the source code immediately following the try block
3. If a catch block is found that matches the exception thrown, it is executed then control goes to next statement
   - If no catch block matching the exception is found, the program terminates
   - If no exception is thrown, the catch block is skipped
Exceptions – Example

```cpp
int totalDays( int days, int weeks )
{ if ( ( days < 0 ) || ( days > 7 ) )
    throw "invalid number of days";
else
    return ( 7 * weeks + days );
} // totalDays
```

Exceptions – Example

```cpp
try
{ totDays = totalDays( days, weeks );
  cout << "Total days: " << totDays;
} // try
catch ( char *msg )
{ cout << "Error: " << msg;
} // catch
```
Exceptions – What Happens

1. The try block is entered and the totalDays function is called
2. If the first parameter is between 0 and 7, the total number of days is returned and the catch block is skipped over (no exception thrown)
3. If an exception is thrown, the function and try block are exited and the catch blocks are scanned for first one that matches that data type of the thrown exception
   - The catch block executes
   - The statement following is then executed

Exceptions - Notes

- Predefined functions, such as new, may throw exceptions
- The value that is thrown does not need to be used in the catch block
  - In this case, no name is needed in the catch parameter definition
  - The catch block parameter definition does need the type of exception being caught
Exceptions Not Caught?

- An exception **will not** be caught if:
  - It is thrown from outside a try block
  - There is no catch block that matches the data type of the thrown exception
- If an exception is not caught, the program terminates

Exceptions and Objects

- An exception **class** can be defined in a class and thrown as an exception by a member function
- An exception class may have:
  - No members – used only to signal an error
  - Members – pass error data to catch block
- A class can be more than one exception class
Exceptions – What Happens After a catch Block?

- Once an exception is thrown, the program cannot return to the throw point
  - The function executing the throw is terminated (does not return), other calling functions in the try block terminate, and this resulting in unwinding the stack
- If objects were created in the try block and an exception is thrown, they are destroyed

---

IntRange.h – Program

```cpp
#ifndef INTRANGE_H
#define INTRANGE_H
#include <iostream>
class IntRange {
    private:
        int input;
        int lower;
        int upper;
    public:
        class OutOfRange // Exception class
        {
            public:
                int value;
                OutOfRange( int i ) { value = i; }
        };
}; // IntRange
```
**IntRange.h – Program**

```cpp
IntRange ( int low, int high ) { lower = low; upper = high; }
int getInput ( void )
{ cin >> input;
  if ( input < lower || input > upper )
    throw OutOfRange( input );
  return input;
} // getInput
}; // IntRange.h

#endif
```

**IntRangeMain.cpp – Program**

```cpp
#include <iostream>
#include "IntRange.h"
using namespace std;

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

  cout << "Enter a value in the range 5 - 10: ";
```
try
{    userValue = range.getInput();
    cout << "You entered " << uservalue << endl;
} // try

catch ( IntRange::OutOfRange ex )
{    cout << "The value " << ex.value
        << " is out of range." << endl;
} // catch

return 0;
} // main

Enter a value in the range 5 - 10: 12[Enter]
The value 12 is out of range.
End of the program.
Function Templates

- A function template is a pattern for a function that can work with many data types
- When written, parameters are left for the data types
- When called, the compiler generates code for the specific data types in the function call

Function Template Example

```
template <class T>
T times10( T num )
{ return 10 * num;
} // time10
```

double times10 ( double num )
{ return 10 * num;
} // times10

<table>
<thead>
<tr>
<th>What gets generated when times10 is called with an int?</th>
<th>What gets generated when times10 is called with a double?</th>
</tr>
</thead>
</table>
| int times10( int num)
{ return 10 * num;
} // times10 | double times10 ( double num )
{ return 10 * num;
} // times10 |
Template Example

- Suppose we want to write a generic function that prints the elements of an array regardless of the type of the element

PrintArray.cpp – Program

```cpp
#include <iostream>
#include <cstring>
using namespace std;

template< class T>
void printArray ( const T *array, const int size )
{  for ( int i = 0; i < size; i++ )
    { if ( !( i % 10 ) )
        cout << endl;
        cout << array[i] << " ";
    } // for
    cout << endl;
} // printArray
```
int main ( void )
{
    const int aSize = 5, bSize = 7, cSize = 6, dSize = 3;
    int a[ aSize ] = { 1, 2, 3, 4, 5 };
    float b[ bSize ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
    char c[ cSize ] = "Hello";
    string d[ dSize ] = { "I", "love", "CS" };

    printArray( a, aSize );
    printArray( b, bSize );
    printArray( c, cSize );
    printArray( d, dSize );
    return 0;
} // main

printArray.cpp – Output

1 2 3 4 5
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Hello
I love CS
Generic swap

```c++
template <class T>
void swap ( T &var1, T &var2 )
{  T temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}  // swap
```

Template with Multiple Types

```c++
template <class T1, class T2>
void swap ( T1 &var1, T2 &var2 )
{  T1 temp;
    temp = var1;
    var1 = static_cast<T1>( var2 );
    var2 = static_cast<T2>( temp );
}  // swap
```
Operators in Template

- If arithmetic or relational operators are used in templates, they must be defined for the different types of the templates.
- If the operators are not defined for a particular type, the compiler generates an error.

Overloading Function Template

- Function templates may be overloaded.
- Each template must have a unique parameter list.

```
template <class T>
  T sumAll( T num ) ...
```

```
template <class T1, class T2>
  T sumAll( T1 num1, T2 num2 ) ...
```
Overloading Function Templates

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

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

Overloading Class Template

- All data types specified in the template prefix must be used in the template definition
- Function calls must pass parameters for all data types specified in the template prefix
- Like regular functions, function templates must be defined before being called
More on Function Templates

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition

Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop the function using usual data types first, then convert it 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

- Classes can also be represented by templates
  - When a class object is created, type information is supplied to define the type of data members of the class
- Unlike functions, classes are instantiated by supplying the type name (int, double, string, etc.) at object definition

Generic Class

- Suppose you wanted to have a class that checked the bounds on arrays
- We want this class to work with different data types for the array
- We could create a different class for each data type, but there is a better solution
- We will use a class template
```cpp
#ifndef SIMPLEVECTOR_H
#define SIMPLEVECTOR_H
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

template <class T>
class SimpleVector
{
private:
    T *aptr;
    int arraySize;
    void memError( void );
    void subError( void );
public:
    SimpleVector ( void ) { aptr = NULL; arraySize = 0; };
    SimpleVector ( int s )
    { arraySize = s;
        try
        { aptr = new T[ s ];
        } // try
        catch ( bad_alloc )
        { memError( );
        } // catch
        for ( int i = 0; i < arraySize; i++ )
            *( aptr + i ) = 0;
    } // SimpleVector::SimpleVector
    ~SimpleVector ( void )
    { if ( arraySize > 0 )
        delete [] aptr;
    } // SimpleVector::~SimpleVector
}; // SimpleVector
```
template <class T>
SimpleVector<T>::SimpleVector ( const SimpleVector &obj )
{ arraySize = obj.arraySize;
aptr = new T [ arraySize ];
if ( aptr == NULL )
    memError( );
for ( int i; i < arraySize; i++ )
    *( aptr + i ) = *( obj.aptr + i );
} // SimpleVector::SimpleVector

template <class T>
T &SimpleVector<T>::operator[]( const int &sub )
{ if ( sub < 0 || sub >= arraySize )
    subError( );
    return aptr[sub];
} // SimpleVector::operator[]

template <class T>
T &SimpleVector<T>::getElementAt( const int &sub )
{ return operator[]( sub );
} // SimpleVector<T>::getElementAt

template <class T>
void SimpleVector<T>::memError( void )
{ cout << "ERROR: Cannot allocate memory.\n";
    exit( 1 );
} // SimpleVector<T>::memError

template <class T>
void SimpleVector<T>::subError( void )
{ cout << "ERROR: Subscript out of range.\n";
    exit( 1 );
} // SimpleVector<T>::subError
Declaring Objects of Class Templates

- The declaration of class templates is a little different
- Consider the following examples:
  - SimpleVector<int> intTable ( 10 );
  - SimpleVector<float> floatTable ( 10 );
- In these, the parameter inside the angle brackets replaces the T in the previous declarations

Class Templates and Inheritance

- Class templates can inherit from other class templates:
  ```
  template <class T>
  class SimpleVector
  { ... };
  class SearchableVector : public SimpleVector<T>
  { ... };
  ```
- Must use type parameter T everywhere the base class name is used in the derived class
#ifndef SEARCHABLEVECTOR_H
#define SEARCHABLEVECTOR_H
#include "SimpleVector.h"

template <class T>
class SearchableVector : public SimpleVector<T>{  public:
    SearchableVector ( int s ) : SimpleVector<T>( s ) { }
    SearchableVector ( SearchableVector & );
    SearchableVector ( SimpleVector<T> &obj ) :
        SimpleVector<T>( obj ) { }
    int findItem ( T );
}; // SearchableVector

SearchableVector<T>::SearchableVector ( SearchableVector &obj)
    : SimpleVector<T>( obj.size() )
    { for ( int i = 0; i < this->size(); i++ )
        this->operator[]( i ) = obj[ i ];
    } // SearchableVector

template <class T>
int SearchableVector<T>::findItem ( T item )
{ for ( int i = 0; i < size(); i++ )
    if ( getElementAt( i ) == item )
        return i;
    return -1;
} // findItem
Introduction to the Standard Template Library (STL)

- The Standard Template Library contains many templates for frequently used algorithms and data structures
- We’ve seen these earlier in the book
  - The vector data type
- STL is not supported in many older compilers

Abstract Data Types

- The most important data structures in STL are containers and iterators:
  - Container – a class that stores data and organizes it in some fashion
  - Iterator – similar to a pointer and is used to access the individual data elements in a container
Container Classes

- **Sequence** – organizes data in a sequential fashion similar to an array
  - Includes: vector, deque, and list

- **Associative** – uses keys to rapidly access elements in the data structure
  - Includes: set, multiset, map, and multimap

Sequence Containers

<table>
<thead>
<tr>
<th>Container Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>An expandable array. Values may be added to or removed from the end or middle.</td>
</tr>
<tr>
<td>deque</td>
<td>Like a vector, but also allows values to be added to or removed from the front.</td>
</tr>
<tr>
<td>list</td>
<td>A doubly-linked list of data elements. Values may be inserted in or removed from any position.</td>
</tr>
</tbody>
</table>
## Associative Containers

<table>
<thead>
<tr>
<th>Container Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>Stores a set of keys. No duplicate values are allowed.</td>
</tr>
<tr>
<td>multiset</td>
<td>Stores a set of keys. Duplicates are allowed.</td>
</tr>
<tr>
<td>map</td>
<td>Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed.</td>
</tr>
<tr>
<td>multimap</td>
<td>Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed.</td>
</tr>
</tbody>
</table>

## Iterators

<table>
<thead>
<tr>
<th>Iterator Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Can only move forward in a container (uses the ++ operator).</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>Can move forward or backward in a container (uses the ++ and -- operators).</td>
</tr>
<tr>
<td>Random-Access</td>
<td>Can move forward and backward, and can jump to a specific data element in a container.</td>
</tr>
<tr>
<td>Input</td>
<td>Can be used with cin to read information from an input device or a file.</td>
</tr>
<tr>
<td>Output</td>
<td>Can be used with cout to write information to an output device or a file.</td>
</tr>
</tbody>
</table>
More on Iterators

- Iterators are associated with containers
  - The type of container you have determines the type of iterator you use
- For example, vectors and deques require random-access iterators
- Lists, sets, multisets, maps, and multimaps require bidirectional iterators

Algorithms

- Algorithms are implemented as function templates
- They perform various operations on elements of containers
- They require algorithm header file
- Includes:
  - binary_search, for_each, find_if, min_element, sort, count, find, max_element, random_shuffle and others
**Description of Some Algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binary_search</code></td>
<td>Performs a binary search for an object and returns true if the object is found. For example, consider <code>binary_search(iter1, iter2, value)</code>; The statements performs a binary search in the range <code>iter1</code> to <code>iter2</code> looking for <code>value</code>.</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Returns the number of times a value appears in a range. For example, consider <code>iter3 = count(iter1, iter2, value)</code>; The statements returns the number of times <code>value</code> is found in the range <code>iter1</code> to <code>iter2</code>.</td>
</tr>
<tr>
<td><code>for_each</code></td>
<td>Executes a function for each element of a container. For example, consider <code>for_each(iter1, iter2, func)</code>; The statement calls the function <code>func</code> for each element in the range from <code>iter1</code> to <code>iter2</code>, passing the element as the argument.</td>
</tr>
</tbody>
</table>

---

**Using find – Program**

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main( void )
{
    vector<int> numbers;
    vector<int>::iterator iter;
    for ( int x = 0; x < 10; x++ )
        numbers.push_back( x );
    cout << "The numbers in the vector are:\n";
    for ( iter = numbers.begin( ); iter != numbers.end( ); iter++ )
        cout << *iter << endl;
    cout << endl;
    iter = find( numbers.begin( ), numbers.end( ), 7 );
    cout << *iter << endl;
    return 0;
} // main
```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void doubleValue ( int &val )
{
    val *= 2;
} // doubleValue

void main ( void )
{
    vector<int> numbers;
    vector<int>::iterator iter;

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

    cout << "The numbers in the vector are:
    
    for ( iter = numbers.begin(); iter != numbers.end(); iter++ )
        cout << *iter << endl;
    cout << endl;

    for_each ( numbers.begin(), numbers.end(), doubleValue );

    cout << "The numbers again are:
    
    for ( iter = numbers.begin(); iter != numbers.end(); iter++ )
        cout << *iter << endl;
    cout << endl;
    } // main