# Exceptions, Templates, and the Standard Template LIbrary

Chapter 16

# Exceptions

- Exception
  - A value or object that indicates an error has occured
  - used to signal errors or unexpected events

# Exceptions

- Exception
  - A value or object that indicates an error has occured
  - used to signal errors or unexpected events

- When an exception occurs in a program, it must either terminate or jump to code designed to handle the exception

# Exceptions

- Exception Handler
  - The code used to handle exceptions

# Exceptions

- Exceptions in C++

# Exceptions

- Exceptions in C++
  - ‘throw’
    - followed by an argument
    - used to signal an exception

# Exceptions - Throw

```
int doSomething(int value)
{
	if (value == 1)
	{
		throw std::string("Cannot pass in a value of 1\n");
	}

	return value;
}
```

# Exceptions

- Exceptions in C++
  - ‘try’
    - followed by a block { }
    - used to invoke code that may throw an exception

# Exceptions

- Exceptions in C++
  - ‘try’
    - followed by a block { }
    - used to invoke code that may throw an exception
  - ‘catch’
    - followed by a parameter that matches the exception type
    - followed by a block { }
    - processes exceptions thrown by try block

# Exceptions - Try/Catch

```
try
{
	doSomething(1);
}
catch (std::string exception)
{
	printf("Caught exception: %s", exception.c_str());
}
```

# Exceptions

- The block of code that handles the exception is said to ‘catch’ the exception

# Exceptions

- The block of code that handles the exception is said to ‘catch’ the exception

- The exception handler is written to catch exceptions of a give type
    - catch (std::string exception)...
    - catch (char* exception) …
    - ...

# Exceptions

- Why would we use exception handling instead of just returning something?

# Exceptions

- Why would we use exception handling instead of just returning something?
    - What about a function that calculates the square root of a number?

# Exceptions

- Why would we use exception handling instead of just returning something?
  - What about a function that calculates the square root of a number?
    - And what if the function is called with a negative number?
    - How can you tell the error from a good return value?

# Exceptions

```
int main( ) {
  try  {
    double x;
    cout << "Enter a number: ";
    cin >> x;
    if (x < 0) throw "Bad argument!";
    cout << "Square root of " << x <<  " is " << sqrt(x);
  }
  catch(char *str)  {
      cout << str;
  }
  return 0;
}
```

# Exceptions

- For exceptions, there is a special flow of control

# Exceptions

- For exceptions, there is a special flow of control
    - When a throw statement is reached
        - Skip the rest of the function
        - The try block is exited
        - If there is a catch block that matches the exception type
            - The catch case is executed

# Exceptions

- If an exception was not caught
  - No catch block that matches the data type
  - The throw happened outside if a try block

# Exceptions

- If an exception was not caught
  - No catch block that matches the data type
  - The throw happened outside if a try block

- Both cases cause the program to terminate

# Exceptions

- There are times when it is possible for different types of ‘throws’ to take place

# Exceptions

- There are times when it is possible for different types of ‘throws’ to take place
  - For this we can use multiple catch blocks

# Exceptions

- There are times when it is possible for different types of ‘throws’ to take place
  - For this we can use multiple catch blocks

```
try {
	...
}
catch (int intException) { … }
catch (char* strException) { … }
catch (double doubleException) { … }
...
```

# Exceptions

- Try blocks can also be nested inside of another try block

# Exceptions

- Try blocks can also be nested inside of another try block

```
try {
        try {
                ...
        }
        catch (double doubleException) { … }
}
catch (int intException) { … }

...
```

# Exception Classes

- So far, we have seen exceptions using primitive data types, but it is also possible to define and throw an ‘exception class’

# Exception Classes

- For Exception Classes
  - The catch block must be written to handle the class
  - The class can contain a lot more information about the error via data members
  - The classes are regular classes used to hold exceptions

# Exception When Calling ‘new’

- Although we rarely talk about it, the ‘new’ operation can fail

# Exception When Calling ‘new’

- Although we rarely talk about it, the ‘new’ operation can fail
  - The exception is of type ‘bad_alloc’
    - Use #include <new>
    - detects that memory was NOT allocated

# Exception When Calling ‘new’

```
#include <limits.h>

try
{
	int* pHugeArray = new int[ULONG_MAX]; // comes from limits.h
	delete [] pHugeArray;
}
catch (std::bad_alloc e)
{
	printf("Bad allocation: %s\n", e.what());
}
```

# Unhandled Exception

- The compiler tries to find a handler to an enclosing ‘try’ block in the same function
- If none is found, it terminates execution of the function, and continues searching for a handler starting at the point of the call in the calling function

# Unhandled Exception

- Unhandled exceptions propagate backwards up the stack

# Unhandled Exception

- Unhandled exceptions propagate backwards up the stack
  - For example, if foo() called foo1() called foo2() called … and the exception happened at the bottom

# Unhandled Exception

- Unhandled exceptions propagate backwards up the stack
  - For example, if foo() called foo1() called foo2() called … and the exception happened at the bottom
  - The catch block searching would propagate backwards during its search
    - checks foo3(),  checks foo2() where the function call to foo3 happened, checks foo1() where the function call to foo2 happened, ...

# Unhandled Exception

- The process of tracing backwards for catch blocks is called ‘unwinding the call stack’

# Unhandled Exception

- The process of tracing backwards for catch blocks is called ‘unwinding the call stack’

- If the unwinding propagates out of main, then the program is terminated

# Unhandled Exception

- Sometimes it may be needed to do some tasks in an exception handler and then continue the throw up the stack

# Unhandled Exception

- Sometimes it may be needed to do some tasks in an exception handler and then continue the throw up the stack

- Calling ‘throw;’ with no arguments can be used within an exception handler to pass the exception up

# Templates

# Function Templates

- How would we go about creating a square function for integers?

# Function Templates

- What if we wanted it to support other data types?

# Function Templates

- What if we wanted it to support other data types?
    - ints
    - char*
    - floats
    - structs
    - classes
    - ...

# Function Templates

- This can be accomplished using function templates

# Function Templates

- A ‘Function Template’ is a pattern for creating definitions of functions that differ only in the type of data they manipulate

# Function Templates

- A ‘Function Template’ is a pattern for creating definitions of functions that differ only in the type of data they manipulate
  - i.e. Defining a function to allow it to work for many different data types

# Function Templates

- A ‘Function Template’ is a pattern for creating definitions of functions that differ only in the type of data they manipulate
  - i.e. Defining a function to allow it to work for many different data types
- This is better than overloading many functions because the code is only written once

# Function Templates

- Consider two functions

```
void swap(int& x, int& y){
        int temp = x;
        x = y;
        y = temp;
}
void swap(char& x, char& y){
        int temp = x;
        x = y;
        y = temp;
}
```

# Function Templates

- Consider two functions

```
void swap(int& x, int& y){
        int temp = x;
        x = y;
        y = temp;
}
void swap(char& x, char& y){
        int temp = x;
        x = y;
        y = temp;
}
```

They both perform the same operations but on different data types

# Function Templates

- Using templates, the code is simpler

# Function Templates

- Using templates, the code is simpler

```
template<class T>
void swap(T& x, T& y){
        T temp = x;
        x = y;
        y = temp;
}
```

The ‘template<class T>’ indicates that some unknown class will be used in place of ‘T’.

This is similar to variables in math.

# Function Templates

- For a function template, the compiler create the actual definition from the template by inferring the type of the type parameters from the arguments in the call

# Function Templates

- For a function template, the compiler create the actual definition from the template by inferring the type of the type parameters from the arguments in the call
  - int i = 1; int j = 2; swap(i,j)
    - Forces the compiler to instantiate the template with type int in place of the ‘T’

# Function Templates

- More than one generic type can be used in a function template

# Function Templates

- More than one generic type can be used in a function template

```
template <class T1, class T2, class T3>
void someFunction(T1 a, T2 b, T3 c)
{
	…
}
```

# Function Templates

- More than one generic type can be used in a function template

```
template <class T1, class T2, class T3>
void someFunction(T1 a, T2 b, T3 c)
{
	…
}
```

- Each type parameter declared must be used in the template definition

# Function Templates

- For templates
  - The function template is a pattern

# Function Templates

- For templates
  - The function template is a pattern
  - No code is generated until the function is called

# Function Templates

- For templates
  - The function template is a pattern
  - No code is generated until the function is called
  - Function templates use no memory
    - An actual instance of the function is created in memory once the function is used

# Function Templates

- For templates
    - The function template is a pattern
    - No code is generated until the function is called
    - Function templates use no memory
    - When passing a class, always make sure that all operators used in the function are defined or overloaded in the class definition

# Function Templates

- For templates
  - The function template is a pattern
  - No code is generated until the function is called
  - Function templates use no memory
  - When passing a class, always make sure that all operators used in the function are defined or overloaded in the class definition
  - Function templates can be overloaded

# Function Templates

- For templates
  - The function template is a pattern
  - No code is generated until the function is called
  - Function templates use no memory
  - When passing a class, always make sure that all operators used in the function are defined or overloaded in the class definition
  - Function templates can be overloaded
  - Function templates must be defined before use

# Function Templates

- When should we use templates?

# Function Templates

- When should we use templates?
  - when we would otherwise create multiple functions that perform the same task but with different data types

# Function Templates

- When should we use templates?
  - when we would otherwise create multiple functions that perform the same task but with different data types
- When creating template functions
  - develop the function using normal data types, then convert to template
    - add template prefix
    - replace data types with generic names, ie ‘T’

# Class Templates

- What if we wanted to create a class that supported multiple types?
  - list
  - stack
  - ...

# Class Templates

- The same ability that allows us to create function templates can be used on classes

# Class Templates

- The same ability that allows us to create function templates can be used on classes
- Unlike function templates, a class template must be instantiated by supplying the type name at object creation.
  - Student<int> myClass;

# Class Templates

- Example

```
template <class T>
class MyList {
        public:
                MyList();
                ...
                void insert(T value);
};
```

# Class Templates

- Example

```
template <class T>
class MyList {
    public:
        MyList();
        ...
        void insert(T value);
};
MyList<int> intList;
MyList<float> floatList;
...
```

# Class Templates

- Example

```
template <class T>
class MyList {
        public:
                MyList();
                ...
                void insert(T value);
};
MyList<int> intList;
MyList<float> floatList;
...
```

Lets implement our own version of this using integers, then modify to work with templates

# Class Templates and Inheritance

- Template classes can also be used with inheritance

# Class Templates and Inheritance

- Template classes can also be used with inheritance
- Inheritance works for
  - non template classes inheriting from a template class
    - the base class template must be instantiated and then inherited from

# Class Templates and Inheritance

- Template classes can also be used with inheritance
- Inheritance works for
  - non template classes inheriting from a template class
    - the base class template must be instantiated and then inherited from
  - template class from a template class
  - ...

# Class Templates and Inheritance

```
template <class T>
class MyClass{
public:
        MyClass();
        void insert(T item);
        T getItem(int pos);
};
class MyBetterClass : public MyClass<int> {
public:
        MyBetterClass()
        :  MyClass() { ... }
};
```

# Standard Template Library

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

# Standard Template Library

- There are two important data structures in the STL
  - Containers
    - classes for storing data and imposing some organization

# Standard Template Library

- There are two important data structures in the STL
  - Containers
    - classes for storing data and imposing some organization
  - iterators
    - similar to pointers
    - allow accessing of elements in a container

# STL Containers

- There are two important containers types in the STL
    - Sequential containers
        - data is access and organized sequentially
        - vector, list, ...

# STL Containers

- There are two important containers types in the STL
  - Sequential containers
    - data is access and organized sequentially
    - vector, list, …
  - Associative containers
    - keys are used to allow data to be accessed quickly
    - set, multiset, map, multimap, ...

# STL Iterators

- An iterator is a generalization of a pointer used to access information in containers

# STL Iterators

- An iterator is a generalization of a pointer used to access information in containers
  - types
    - forward (operator++)
    - bidirectional (operator++ and --)
    - random-access
    - input (usable with cin and istream)
    - output (usable with cout and ostream)

# STL Iterators

- Each container class defines an iterator
  - list<int>::iterator pIt;
  - vector<int>::iterator pIt;
  - ...

# STL Iterators

- Each container class defines an iterator
  - list<int>::iterator pIt;
  - vector<int>::iterator pIt;
  - ...
- Each container class defines a way to get an iterator
  - begin()
  - end()
  - ...

# STL Iterators

- Iterators support pointer-like operations
  - Dereferencing
    - *pIt would give you the item that pIt refers to

# STL Iterators

- Iterators support pointer-like operations
  - Dereferencing
    - *pIt would give you the item that pIt refers to
  - Advancing
    - pIt++ can advance to the next item in the container

# STL Iterators

- Iterators support pointer-like operations
  - Dereferencing
    - *pIt would give you the item that pIt refers to
  - Move forward through the container
    - pIt++ can advance to the next item in the container
  - Move backward through the container
    - pIt-- can move to the previous item in the container

# STL Algorithms

- The STL also contains algorithms
  - requires the algorithm header
- Some of the algorithms include
  - binary_search
  - for_each
  - max_element
  - random_shuffle
  - count
  - find

# STL Algorithms

- Some of the STL algorithms can manipulate containers based off of a begin and end iterator

# STL Algorithms

- Some of the STL algorithms can manipulate containers based off of a begin and end iterator
  - max_element(iterator1, iterator2)
    - finds the max element in the portion of the container delimited by iterator1 and iterator2
  - min_element(iterator1, iterator2)
    - same as above, but minimum

# STL Algorithms

- Some of the STL algorithms can manipulate containers based off of a begin and end iterator
  - random_shuffle(iterator1, iterator2)
    - randomly reorders the portion of the container
  - sort (iterator1, iterator2)
    - sorts the portion of the container

# STL Algorithms

- How can we use max_element on a list?