

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

Week 8

Exceptions



Exceptions

- ❑ An **exception** is a value or an object that indicates that an error has occurred
- ❑ 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**
- ❑ **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 a preceding **try** block. It takes a parameter that matches the type of exception thrown



Throwing an Exception

- ❑ Code that detects the exception must pass information to the exception handler.
- ❑ This is done using a **throw** statement:

```
throw string("Emergency!");  
throw 12;
```

- ❑ In C++, information thrown by the **throw** statement may be a value of any type

```
int x = 10;  
int y = 0;  
  
try  
{  
    if (y == 0)  
        throw "Error: Divide by 0\n";  
    cout << x/y << endl;  
}  
catch(const char *r)  
{  
    cout << r << endl;  
}
```



Catching an Exception

- ❑ Block of code that handles the exception is said to **catch** the exception and is called an **exception handler**
- ❑ An exception handler is written to catch exceptions of a given type: For example, the code

```
catch(string str)
{
    cout << str;
}
```

```
catch(int x)
{
    cerr << "Error: " << x;
}
```



- ❑ Every catch block is attached to a **try** block of code and is responsible for handling exceptions thrown from that block

```
try
{
    // code that may throw an exception
    // goes here
}
catch(char e1)
{
    // This code handles exceptions
    // of type char that are thrown
    // in this block
}
```



Execution of Catch Blocks

- ❑ The catch block syntax is similar to that of a function
- ❑ A catch block has a formal parameter that is initialized to the value of the thrown exception before the block is executed
- ❑ An example of exception handling is code that computes the square root of a number.
- ❑ It throws an exception in the form of a string object if the user enters a negative number

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



Flow of Control

1. The computer encounters a **throw** statement in a **try** block
2. The computer evaluates the **throw** expression, and immediately exits the **try** block
3. The computer selects an attached **catch** block that matches the type of the thrown value, places the thrown value in the catch block's formal parameter, and executes the catch block

```
int main( )
{
    try
    {
        double x;
        cout << "Enter a number: ";
        cin >> x;

        if (x < 0)
            throw string("Bad argument!");

        cout << "Square root of " << x
              << " is " << sqrt(x);
    }

    catch(string str)
    {
        cout << str;
    }

    return 0;
}
```



Exceptions

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

```
try
{
    totDays = totalDays(days, weeks);
    cout << "Total days: " << days;
}
catch (char *msg)
{
    cout << "Error: " << msg;
}
```

1) try block is entered. totalDays function is called.

2) If 1st parameter is between 0 and 7, total number of days is returned and catch block is skipped over (no exception thrown).

3) If exception is thrown, function and try block are exited, catch blocks are scanned for 1st one that matches the data type of the thrown exception. catch block executes.



Uncaught Exception

- ❑ An exception may be uncaught if
 - there is no **catch** block with a data type that matches the exception that was thrown, or
 - it was not thrown from within a **try** block
- ❑ The program will terminate in either case

```
int main( )
{
    try {
        double x;
        cout << "Enter a number: ";
        cin >> x;
        cout << "Square root of " << x
             << " is " << sqrt(x);
    }
    catch(int p) {
        cout << p;
    }
    return 0;
}
```

```
int main( )
{
    try
    {
        double x;

        cout << "Enter a number: ";
        cin >> x;

        if (x < 0)
            throw string("Bad argument!");

        cout << "Square root of " << x
             << " is " << sqrt(x);
    }

    catch(int p)
    {
        cout << p;
    }

    return 0;
}
```



Handling Multiple Exceptions

- ❑ Multiple catch blocks can be attached to the same block of code.
- ❑ The catch blocks should handle exceptions of different types

```
try {...}  
catch (int iEx){ }  
catch (string strEx){ }  
catch (double dEx){ }
```

```
void multi_catch() {  
    int x =10;  
    int y = 1;  
    try {  
        if (y == 0) throw "divide by zero";  
        if (y == 1) throw 99;  
  
        cout << x/y << endl;  
  
    }  
  
    catch(const char *r)  
    {  
        cout << r << endl;  
    }  
  
    catch(int n)  
    {  
        cout << "other error: " << n << endl;  
    }  
}
```

Handling Multiple Exceptions

```
void use_1_3()
{
    int x = 10;
    int y = 1;

    try
    {
        if (y == 0) throw "Error: Divide by Zero\n";
        if (y == 1) throw 99;

        cout << x/y << endl;

    }

    catch(...) // will receive any exception type

    {
        cout << "Error occurs" << endl;
    }
}
```



Multiple Exceptions

- ❑ a method that throws more than one exception type

```
void method() throw (int, const char *, runtime_error)
{
    int *ptr = nullptr;

    if (ptr == nullptr) throw runtime_error("NULL Pointer\n");
    *ptr = 10;
}
```

- ❑ If the method below throws an exception other than int, the method calls **std::unexpected** instead of looking for a handler or calling **std::terminate**.

```
void exception_method_2() throw()
{
    int *ptr = nullptr;

    if (ptr != nullptr) throw 99;
    if (ptr == nullptr) throw "none-int exception\n";
}
```



Throwing an Exception Class

- ❑ An **exception class** can be defined and thrown
- ❑ A catch block must be designed to catch an object of the exception class
- ❑ The exception class object can pass data to the exception handler via data members

```
class MException
{
    string m_msg;
public:
    MException()=default;
    MException(string msg):m_msg(msg) { }
    const char *print(){ return m_msg.c_str();}
};
```

```
int main () {
    int x =10;
    int y = 0;
    try {
        if (y == 0) throw MException("Divide by 0");
        cout << x/y << endl;
    }
    catch(MException &x)
    {
        cout << "Error: "<< x.print() << endl;
    }
}
```

Exception When Calling new

- ❑ If **new** cannot allocate memory, it throws an exception of type **bad_alloc**
- ❑ Must **#include <new>** to use **bad_alloc**
- ❑ You can invoke **new** from within a **try** block, then use a **catch** block to detect that memory was not allocated.

```
void standard_exceptions() {  
    try {  
        int* myarray= new int[1000];  
    }  
    catch (exception& e) {  
        cout << "Standard exception: "  
            << e.what() << endl;  
    }  
}
```

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr



Where to Find an Exception Handler?

- ❑ The compiler looks for a suitable handler attached to an enclosing **try** block in the same function
- ❑ If there is no matching handler in the function, it terminates execution of the function, and continues the search for a handler starting at the point of the call in the calling function.



Unwinding the Stack

- ❑ An unhandled exception propagates backwards into the calling function and appears to be thrown at the point of the call
- ❑ The computer will keep terminating function calls and tracing backwards along the call chain until it finds an enclosing **try** block with a matching handler, or until the exception propagates out of **main** (terminating the program).
- ❑ This process is called **unwinding the call stack**



Rethrowing an Exception

- ❑ Sometimes an exception handler may need to do some tasks, then pass the exception to a handler in the calling environment.
- ❑ The statement

`throw;`


- ❑ with no parameters can be used within a catch block to pass the exception to a handler in the outer block

Nested Exception Handling

- ❑ Once an exception is thrown, the program cannot return to throw point.
- ❑ The function executing throw terminates (does not return), other calling functions in try block terminate, resulting in unwinding the stack.
- ❑ If objects were created in the try block and an exception is thrown, they are destroyed.
- ❑ try/catch blocks can occur within an enclosing try block
- ❑ Exceptions caught at an inner level can be passed up to a catch block at an outer level

```
int x = 10;
int y = 0;

Try
{
    try
    {
        if (y == 0) throw "divide by zero";
        cout << x/y << endl;
    }
    catch(const char * e)
    {
        cout << e << endl;
        throw;
    }
}
catch(exception &e)
{
    cout << "second catch "
        << e.what()
        << endl;
}
```



Standard Exceptions

- ❑ Two generic exception classes are **derived** from **exception** class to report errors:
- ❑ **logic_error**: error related to the internal logic of the program
- ❑ **runtime_error**: error detected during runtime

```
catch(logic_error &e) {  
    cout << e.what() << endl;  
}  
catch(runtime_error &e) {  
    cout << e.what() << endl;  
}  
catch(exception &e) {  
    cout << e.what() << endl;  
}
```

```
void exception_derived () {  
    int x = 10, y = 1;  
    try {  
        if (y == 0) throw runtime_error(" runtime exception");  
        if (y == 1) throw logic_error("logical error");  
        cout << x/y << endl;  
    } catch (exception &e) {  
        cout << e.what() << endl;  
    }  
}
```



Expanding the exception class

```
class myexception: public exception
{
private:
    string message;

public:
    myexception(){
        message = "Generic error\n";
    }
    myexception(string s){
        message = s;
    }

    virtual const char* what() const throw ()
    {
        return message.c_str();
    }
};
```

Override the what method in exception

```
try {
    throw myexception("user def. err.");
}
catch (exception& e)
{
    cout << e.what() << '\n';
}
```

```
try {
    throw myexception("user def. err.");
}
catch (myexception& e)
{
    cout << e.what() << '\n';
}
```



Templates



Polymorphism

- ❑ Inheritance: sub-type, run-time, dynamic Polymorphism
- ❑ Templates: generics, compile time Polymorphism, parametric Polymorphism
- ❑ Overloading: ad-hoc Polymorphism
- ❑ Casting: coercion (by force) Polymorphism



Function Templates

- ❑ **Function template:** A pattern for creating definitions of functions that differ only in the type of data they manipulate. It is a generic function
- ❑ They are better than overloaded functions, since the code defining the algorithm of the function is written once
- ❑ When called, compiler generates code for specific data types in function call is only written once

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

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

```
template <class T>
```

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



Using a Template Function

- ❑ When a function defined by a template is called, the compiler creates the actual definition from the template by inferring the type of the type parameters from the arguments in the call
- ❑ 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 referred to in the template are defined or overloaded in the class definition

```
20 10
3.14 10.3
```

```
#include <iostream>

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

int main()
{
    int x=10, y=20;
    swap(x,y);
    std::cout << x << " "
               << y << std::endl;

    float f=10.3f, g=3.14f;
    swap(f,g);
    std::cout << f << " "
               << g << std::endl;
    return 0;
}
```



Function Template Notes

- ❑ All data types specified in template prefix must be used in 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

```
#include <iostream>

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

int main()
{
    int x=10, y=20;
    swap<int>(x,y);
    std::cout << x << " "
               << y << std::endl;

    float f=10.3f, g=3.14f;
    swap<float>(f,g);
    std::cout << f << " "
               << g << std::endl;
    return 0;
}
```



Where to Start When Defining Templates

- ❑ 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 in the template

```
int max(int x, int y) {  
    return ((x>y)?x:y);  
}
```

```
float max(float x, float y) {  
    return ((x>y)?x:y);  
}
```

```
char max(char x, char y) {  
    return ((x>y)?x:y);  
}
```

```
double max(double x, double y) {  
    return ((x>y)?x:y);  
}
```

```
template <class T>  
T max (T x, T y) {  
    return ((x>y)?x:y);  
}
```

```
int main()  
{  
    cout << max<int>(6,5)  
        << endl;  
    cout << max(3.0f,2.0f)  
        << endl;  
    cout << max(1.0, 2.0)  
        << endl;  
    cout << max('a','g')  
        << endl;  
    return 0;  
}
```

Where to Start When Defining Templates

```
template<class T>
void print(vector<T> &v)
{
    copy(begin(v), end(v), ostream_iterator<T>(cout, "::"));
    cout << endl;
}
```

Copies the elements in the vector to the output stream.

```
int main()
{
    vector<int> vi= {8 , 2 , 7 , 4 , 3 , 6 , 5};
    print(vi);

    vector<double> vd = {3.0 , 4.1 , 5.8};
    print(vd);

    return 0;
}
```

```
8::2::7::4::3::6::5::
3::4.1::5.8::
```



Where to Start When Defining Templates

```
template <class T>
bool Equals(T x, T y)
{
    return x == y;
}

int main ( )
{
    cout << boolalpha << Equals(2,3)      << endl;
    cout << boolalpha << Equals(3.0, 3.0) << endl;
    cout << boolalpha << Equals('f','t')   << endl;

    return 0;
}
```



Function Template Notes

- ❑ Can define a template to use multiple data types
- ❑ T1 and T2 will be replaced in the called function with the data types of the arguments
- ❑ Function templates can be overloaded Each template must have a unique parameter list

```
template<class T1, class T2>  
double mpg(T1 miles, T2 gallons)  
{  
    return miles / gallons;  
}
```

```
template <class T>  
T sumAll (T num)  
{  
    :  
}  
  
template <class T1, class T2>  
T1 sumall (T1 num1, T2 num2)  
{  
    :  
}
```



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
- ❑ Pass type information to class template when defining objects:

`grade<int> testList[20];`

`grade<double> quizList[20];`

- ❑ Use as ordinary objects once defined

```
template <class T>
class grade
{
    private:
        T score;
    public:
        grade(T);
        void setGrade(T);
        T getGrade()
};
```



Class Templates

```
template <class T>
class Grade
{
private:
    T score;
public:
    Grade()=default;
    Grade(T g) {
        score = g;
    }
    void setGrade(T g) {
        score = g;
    }
    T getGrade() {
        return score;
    }
};
```

```
int main()
{
    Grade<int> gi;
    gi.setGrade(20);
    cout<< gi.getGrade() << endl;

    Grade<double> gd;
    gd.setGrade(95.5);
    cout<<gd.getGrade() << endl;

    return 0;
}
```

20
95.5



Class Templates

- ❑ If a member function is defined outside of the class, then the definition requires the template header to be prefixed to it, and the template name and type parameter list to be used to refer to the name of the class

```
int main() {  
    Grade<int> gi;  
    gi.setGrade(20);  
    cout<< gi.getGrade()  
        << endl;  
  
    Grade<double> gd;  
    gd.setGrade(95.5);  
    cout<<gd.getGrade()  
        << endl;  
  
    return 0;  
}
```

20

95.5

```
template <class T>  
class Grade  
{  
private:  
    T score;  
public:  
    Grade();  
    Grade(T);  
    void setGrade(T);  
    T getGrade();  
};
```

```
template<class T>  
Grade<T>::Grade()=default;
```

```
template<class T>  
Grade<T>::Grade(T g){  
    score = g;  
}
```

```
template<class T>  
void Grade<T>::setGrade (T g)  
{  
    score = g;  
}
```

```
template<class T>  
T Grade<T>::getGrade() {  
    return score;  
}
```



Class Template and Inheritance

- ❑ Class templates can inherit from other class templates.
- ❑ Must use type parameter T everywhere base class name is used in derived class

```
int main()
{
    Base<int> b;
    b.set(20);

    Derived<int> d;
    d.set(30);

    return 0;
}
```

```
template <class T>
class Base{
public:
    void set(const T& v) {data = v;}
private:
    T data;
};
```

```
template <class T>
class Derived: public Base<T> {
public:
    void set(const T& v);
};
```

```
template <class T>
void Derived<T>::set(const T& v){
    Base<T>::set(v);
}
```



Template function specialization

template <class T>

```
bool Equals(T lhs, T rhs){  
    cout << "Equals<T>:";  
    return lhs == rhs;  
}
```

Generic template function

template<>

```
bool Equals<string> (string lhs, string rhs) {  
    cout << "Equals<string>:";  
    return lhs.compare(rhs);  
}
```

No template type is specified

template<>

```
bool Equals<float> (float lhs, float rhs) {  
    cout << "Equals<float>:";  
    return fabs(lhs-rhs)<0.00001;  
}
```

Type to exclude

```
int main() {  
    cout << Equals(4,3) << endl;  
    cout << Equals (0.42f-0.02,0.02f) << endl;  
    cout << Equals ("abcd","abcd") << endl;  
    return 0;  
}
```

This will invoke the
special template <>



Template class specialization

```
template <class T>
class Double {
public:
    Double () = default;
    Double (T d) {
        data = d*2;
    }
    T getData(){
        return data;
    }
private:
    T data;
};
```

```
template <>
class Double<string> {
public:
    Double () = default;
    Double (const string d) {
        data = d;
        data.append(d);
    }
    string getData(){
        return data;
    }
private:
    string data;
};
```

```
int main() {
    Double<int> i(3);
    cout << i.getData() << endl;

    Double<double> d(4.0);
    cout << d.getData() << endl;

    Double<string> s("4");
    cout << s.getData() << endl;

    return 0;
}
```

6
8
44

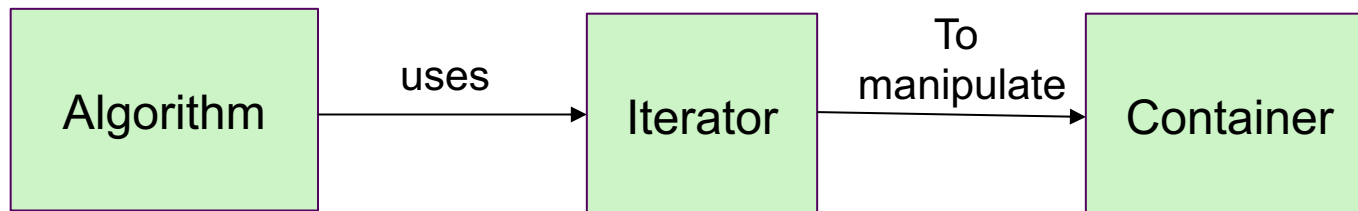


STL: Standard Template Library



Introduction to the Standard Template Library

- ❑ Standard Template Library (STL): a library containing templates for frequently used **data structures** and **algorithms**
- ❑ Not supported by many older compilers
- ❑ STL contains :
 - ❑ **containers**: classes that stores data and imposes some organization on it
 - ❑ **iterators**: like pointers; mechanisms for accessing elements in a container
 - ❑ **algorithms**: implemented as function templates to perform operations on containers



- Two types of container classes in STL:
 - **sequence containers**: organize and access data sequentially, as in an array. These include **array**, **vector**, **deque**, and **list**
 - **associative containers**: use keys to allow data elements to be quickly accessed. These include **set**, **multiset**, **map**, and **multimap**
- **Container Adaptors** : provide a different interface for sequential containers (queue, priority_queue, and stack).

```
#include<iostream>
#include<iterator>
#include<vector>
using namespace std;
int main()
{
    vector<int> ar = { 3, 1, 2, 5, 4 };

    for (unsigned int i=0; i<ar.size(); i++)
        cout << ar[ i ] << " ";

    return 0;
}
```



Iterators

- ❑ **Generalization of pointers.** They are used to access information in containers
- ❑ Many types:
 - ❑ **forward** (uses **++**)
 - ❑ **bidirectional** (uses **++** and **--**)
 - ❑ **random-access**
 - ❑ **input** (can be used with **cin** and **istream** objects)
 - ❑ **output** (can be used with **cout** and **ostream** objects)
- ❑ Each container class defines an iterator type, used to access its contents
- ❑ The type of an iterator is determined by the type of the container:

list<int>::iterator x;
list<string>::iterator y;

x is an iterator for a container of type **list<int>**



Iterators

- ❑ Iterators support pointer-like operations. If **iter** is an iterator, then
 - ❑ ***iter** is the item it points to: this **dereferences** the iterator
 - ❑ **iter++** advances to the next item in the container
 - ❑ **iter--** backs up in the container
- ❑ The **end()** iterator points to past the end: it should never be dereferenced

```
#include<iterator>
#include<vector>
using namespace std;
int main()
{
    vector<int> ar = { 3, 1, 2, 5, 4 };
    cout << "The vector elements are : ";
    vector<int>::iterator it ar.begin();
    while (it != ar.end()) {
        cout << *it << " ";
        it++;
    }
    return 0;
}
```

begin: Return iterator to beginning
end: Return iterator to end
rbegin: Return reverse iterator to reverse beginning
rend: Return reverse iterator to reverse end
cbegin: Return const_iterator to beginning
cend: Return const_iterator to end
crbegin: Return const_reverse_iterator to reverse beginning
crend: Return const_reverse_iterator to reverse end

Some vector Class Member Functions

Function	Description
front(), back()	Returns a reference to the first, last element in a vector
size()	Returns the number of elements in a vector
capacity()	Returns the number of elements that a vector can hold
clear()	Removes all elements from a vector
push_back(value)	Adds element containing value as the last element in the vector
pop_back()	Removes the last element from the vector
insert(iter, value)	Inserts new element containing value just before element pointed at by iter



Algorithms

- ❑ STL contains **algorithms** implemented as function templates to perform operations on containers.
- ❑ STL provides means for various operations for the contents of the containers.
- ❑ Requires **algorithm** header file
- ❑ **algorithm** includes

binary_search	count
for_each	find
find_if	max_element
min_element	random_shuffle
sort	and others



Using STL algorithms

- ❑ Many STL algorithms manipulate portions of STL containers specified by a begin and end iterator
- ❑ **max_element(iter1, iter2)** finds max element in the portion of a container delimited by **iter1, iter2**
- ❑ **min_element(iter1, iter2)** is similar to above
- ❑ **random_shuffle(iter1, iter2)** randomly reorders the portion of the container in the given range
- ❑ **sort(iter1, iter2)** sorts the portion of the container specified by the given range into ascending order

```
int main()
{
    vector<int> vec;

    for (int k = 1; k <= 5; k++)
        vec.push_back(k*k);

    random_shuffle(vec.begin(),vec.end());

    vector<int>::iterator p = vec.begin();
    while (p != vec.end())
    {
        cout << *p << " "; p++;
    }

    return 0;
}
```



Using STL algorithms

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
using namespace std;
int main()
{
    srand(unsigned(time(nullptr)));
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(rand()%100);

    for (auto i = g1.begin();
         i != g1.end();
         ++i)
        cout << *i << " ";

    cout << endl;
```

```
sort(g1.begin(), g1.end());
```

```
    for (auto i = g1.begin();
         i != g1.end();
         ++i)
        cout << *i << " ";
    cout << endl;
```

```
    return 0;
}
```



Using STL algorithms

```
#include<iostream>
#include<set>
#include<string>
using namespace std;
int main() {
    set<int> setNum;

    setNum.insert(3);
    setNum.insert(1);
    setNum.insert(2);
    setNum.insert(3);

    cout<<"Set Size = "
         <<setNum.size()
         <<std::endl;

    for (set<int>::iterator it=setNum.begin();
         it!=setNum.end();
         ++it)
        std::cout << ' ' << *it;
    std::cout<<"\n";
    return 0;
}
```



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