

Lecture 4

More C++

- Templates
- Functors
- string streams
- exceptions
- unit testing

Function Templates

a pattern for a function

Template Motivation

```
void Swap(string& x; string& y)
{ string tmp(move(x));
  x = move(y);
  y = move(tmp);
}

void Swap(vector<int>& x; vector<int>& y)
{ vector<int> tmp (move(x));
  x = move(y);
  y = move(tmp);
}
```

Almost identical.

Can sometimes avoid writing “same” code twice by defining
Swap(Object&, Object&)
and using
typedef string Object or
typedef vector<int> Object

We cannot do this if we want to swap strings and swap vectors in same program.

template

“a preset format for a document or file, used so that the format does not have to be recreated each time it is used: *a memo template.*”

from the dictionary on my computer

Brace
initializer does not
have a type...
e.g f({1,2,3}) doesn't
work since template
type deduction
fails.

The template
parameter cannot*
be deduced from the
return value
*There is one exception to this rule

Templates

Learn more at:
[http://www.cprogramming.com/
tutorial/templated_functions.html](http://www.cprogramming.com/tutorial/templated_functions.html)

- Logic doesn't depend on type.
- Not an actual function/class!
- Compiler instantiates the function template (one instantiation for each type used.) Compiler deduces the type.
- Used in *classes* and *functions*.

Syntax:

template<**class** Type>
function declaration

template<**class** Type>
class declaration

Have the compiler write the function!

```
void Swap( string& lhs, string& rhs)
{
    string tmp(move(lhs));
    lhs = move(rhs);
    rhs = move(tmp);
}
```

```
void Swap( vector<int>& lhs, vector<in>& rhs)
{
    vector<int> tmp(move(lhs));
    lhs = move(rhs);
    rhs = move(tmp);
}
```

A function template is sometimes called a parameterized function or an algorithm.

The compiler can deduce the parameter type from the arguments!

```
template <class Object>
void Swap( Object& lhs, Object& rhs)
{
    Object tmp(move(lhs));
    lhs = move(rhs);
    rhs = move(tmp);
}

int main()
{
    string f = 'Hello';
    string l = 'Hi';
    Swap(f, l);
    vector<int> a = {4, 5};
    vector<int> b = {1, 2};
    Swap(a, b);
    Swap(f, b);
    return 0;
}
```

Function Template
Design for a function.
Not an actual function.

Instantiation
Compiler generates a new function with the correct type. The compiler then checks to make sure the function works with this type.

How would you return the larger of two items?

```
const string& Max(const string& first,
                  const string& second)
{
    return (first > second)? first: second;
}
```

```
const int& Max(const int& first,
               const int& second)
{
    return (first > second)? first: second;
}
```

```
template<class Object>
const Object& Max(const Object& first,
                  const Object& second)
{
    return (first > second)?first: second;
}

int main()
{
    cout << Max( 7, 9);

    cout << Max(string("Hello"), string("World"));
    cout << Max(7, string("World"));

}
```

Finding an item in a vector

Template example: functions to find the largest item in a vector

```
int findMax(const vector<int> & a)
{
    int maxIndex = 0;

    for(int i=1; i < a.size(); i++)
        if (a[maxIndex] < a[i])
            maxIndex = i;
    return maxIndex;
}
```

```
int findMax(const vector<char> & a)
{
    int maxIndex = 0;

    for(int i=1; i < a.size(); i++)
        if (a[maxIndex] < a[i])
            maxIndex = i;
    return maxIndex;
}
```

```
int findMax(const vector<double> & a)
{
    int maxIndex = 0;

    for(int i=1; i < a.size(); i++)
        if (a[maxIndex] < a[i])
            maxIndex = i;
    return maxIndex;
}
```

```
/**
 * Return the index of the maximum item
 * in the array a
 * Assume a.size() > 0
 * Comparable objects must have
 * operator<
 */
template < class Comparable>
int findMax(const vector<Comparable> & a)
{
    int maxIndex = 0;

    for(int i=1; i < a.size(); i++)
        if (a[maxIndex] < a[i])
            maxIndex = i;
    return maxIndex;
}

int main()
{
    vector<char> a(5);
    ... // code to enter 5 items into a
    int i = findMax(a);
    cout << "...";

    vector<int> b(5);
    ... // code to enter 5 items into b
    int j = findMax(b);
    cout << "...";

    vector<IntCell> c(5);
    ... // code to enter 5 items into c

    int k = findMax(c);
    cout << "...";
}
```

When instantiating a function from a template the compiler checks to make sure the function works with this type.

Error!!!

→ k = findMax(c);
cout << "...";

A class template is a type generator.

A class template is sometimes called a parameterized type or parameterized class.

Similarly, we define template classes:

```
template <class Object>
class MyClass
{ Object data[5];
  ...
}
```

Notice that the compiler cannot deduce the type!

```
int main()
{
  MyClass<int> x;
  MyClass<double> y;
  ...
}
```

Generating a class from a templated class is Specialization or template instantiation. Remember that in a templated class you need to provide the template arguments.

A simple MemoryCell Class:

```
class MemoryCell
{
public:
    explicit MemoryCell(int initialValue = 0):storedValue(initialValue){}

    int read() const {return storedValue;}
    void write(int x){storedValue = x;}

private:
    int storedValue;
};
```

Could we store a double
instead of an int?

How about storing a char
instead of an int?

```

class MemoryCell
{
public:
    explicit MemoryCell(double initialValue = 0)
        :storedValue(initialValue){}

    double read() const {return storedValue;}
    void write(double x){storedValue = x;}

private:
    double storedValue;
};

```

```

class MemoryCell
{
public:
    explicit MemoryCell(char initialValue = '')
        :storedValue(initialValue){}

    char read() const {return storedValue;}
    void write(char x){storedValue = x;}

private:
    char storedValue;
};

```

```

template<class Object>
class MemoryCell
{
public:
    explicit MemoryCell(const Object& initialValue = Object())
        :storedValue(initialValue){}
    //public member functions

    Object& read() const {return storedValue;}
    void write(const Object& x){storedValue = x;}

private:
    Object storedValue;
};

int main()
{
    MemoryCell<double> d;
    d.write( 5.0 );
    cout << "Cell contents are " << d.read() << endl;
    MemoryCell<char> c('a');
    cout << "Cell contents are " << c.read() << endl;
    return 0;
}

```

Typical layout for member implementation

```
template<class Object>
class MemoryCell
{
public:
    explicit MemoryCell(const Object& initialValue = Object()):storedValue(initialValue){}
    //public member functions
    const MemoryCell& operator=(const MemoryCell& rhs);
    const Object& read() const {return storedValue;}
    void write(const Object& x){storedValue = x;}

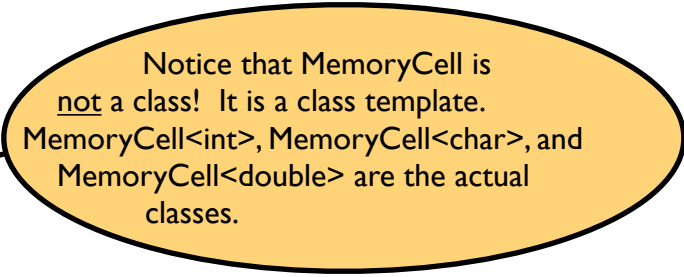
private:
    Object storedValue;
};

template<class Object>
const MemoryCell<Object>& MemoryCell<Object>::operator=(const MemoryCell<Object>& rhs)
{
    if (this != &rhs)
        storedValue = rhs.storedValue;
    return *this;
}

int main()
{
    MemoryCell<int> m;

    m.write( 5 );
    cout << "Cell contents are " << m.read() << endl;

    return 0;
}
```



Notice that MemoryCell is not a class! It is a class template. MemoryCell<int>, MemoryCell<char>, and MemoryCell<double> are the actual classes.

Pair

```
template<class Type1, class Type2>
struct pair
{
public:
    Type1 first;
    Type2 second;
    pair (const Type1 & f=Type1(),const Type2 & s = Type2())
        : first(f), second( s){}
};
```

```
pair<int, int> mypair1;
mypair1.first = 3;
mypair1.second= 9;
cout << mypair1.first<< mypair1.second;

pair<int, string> mypair2;
mypair2.first = 3;
mypair2.second = "shoes";
cout << mypair2.first << " " << mypair2.second;
```

A functor is used for:

- * Customizing sorting algorithms
- * In comparing two items, determining what it means for two items to be equal
- * Customizing numerical algorithms
- * Customizing searching algorithms

Functors

Flexibility & Generality

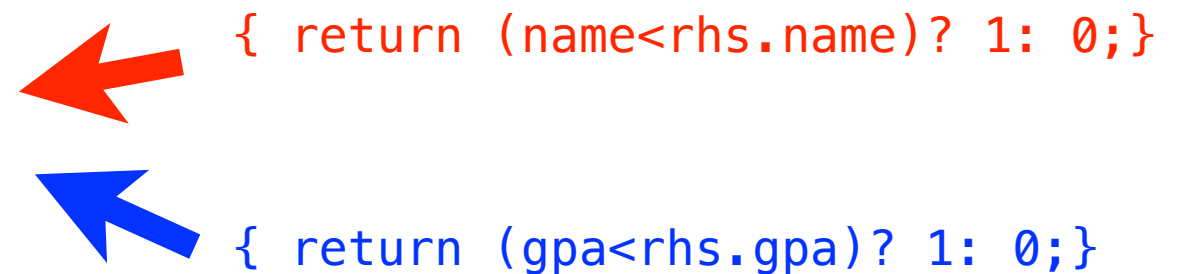
A functor is the main method of parameterization in the STL.

Functors (Function Objects)

- Class that can be useful with no data members and a single method
- Can pass a functor as a template parameter, just like other classes
- This effectively passes the member function as a parameter
- Example: pass **LessThanByGPA** or **LessThanByName** to findmax

Functor Motivation

```
class student
{
private:
    string name;
    double gpa;
    ...
public:
    string get_name();
    double get_gpa();
    bool operator<(const student& rhs){ ...};
    ...
};
```



```
{ return (name<rhs.name)? 1: 0;}
```

```
{ return (gpa<rhs.gpa)? 1: 0;}
```

```
template<class Comparable>
Const Comparable & findMax(const vector<Comparable> & a)
{
    Int maxIndex = 0;
    for( int i = 1; i < a.size( ); ++i )
        if (a[maxIndex] < a[i])
            maxIndex = i;
    return a[maxIndex];
}
```

Meaning of `student::operator<` fixed at compile time.

What if we want to compare students by `name` and by `gpa` at different points within the same program?

```
class LessThanByGPA
{ public:
    bool IsLessThan(const student& lhs, const student& rhs) const
    {return lhs.get_gpa() < rhs.get_gpa();}
};
```

```
class LessThanByName
{ public:
    bool IsLessThan(const student& lhs, const student& rhs) const
    {return lhs.get_name() < rhs.get_name();}
};
```

The second template parameter is a class which has a member function called isLessThan

```
// Generic findMax, with a function object.
```

```
// Precondition: a.size( ) > 0.
```

```
// class Comparator functor with method IsLessThan
```

```
template <class Object, class Comparator>
```

```
const Object & findMax( const vector<Object> & a, Comparator comp )
```

```
{ int maxIndex = 0;
```

```
  for( int i = 1; i < a.size( ); i++ )
```

```
    if( comp.isLessThan( a[ maxIndex ], a[ i ] ) )
```

```
      maxIndex = i;
```

```
  return a[ maxIndex ];
```

```
}
```

```
int main()
```

```
{ vector<student> a;
```

```
  ...
```

```
  student i = findMax(a, LessThanByName());
```

```
  student j = findMax(a, LessThanByGPA());
```

CS2134

```
  ...
```

```
}
```

Cleaner Syntax

- Overloaded `operator()` as a method in a function object
- This is the convention used in STL
- `IsLessThan(....)` is call to `IsLessThan.()`

Both classes have the overloaded operator()

```
class LessThanByGPA
```

```
{ public:
```

```
    bool operator( )(const student& lhs, const student& rhs) const  
    {return lhs.get_gpa() < rhs.get_gpa();}
```

```
};
```

```
class LessThanByName
```

```
{ public:
```

```
    bool operator( )(const student& lhs, const student& rhs) const  
    {return lhs.get_name() < rhs.get_name();}
```

```
};
```

```
int main()
```

```
{ vector<student> a;
```

```
    ...
```

```
    student i = findMax(a, LessThanByName());
```

```
    student j = findMax(a, LessThanByGPA());
```

```
    ...
```

```
}
```

```

template <class Object, class Comparator>
const Object & findMax( const vector<Object> & a, Comparator comp )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( comp( a[ maxIndex ], a[ i ] ) )
            maxIndex = i;

    return a[ maxIndex ];
}

class LessThanByGPA
{ public:
    bool operator()(const student& lhs, const student& rhs) const
    {return lhs.get_gpa() < rhs.get_gpa();}
};

class LessThanByName
{
public:
    bool operator()(const student & lhs, student & rhs) const
    {return lhs.get_name() < rhs.get_name();}
};

int main()
{
    vector<student> classList;
    //some code to fill the classList, etc
    student st1 = findMax( classList, LessThanByName() );
    student st2 = findMax( classList, LessThanByGPA() );
}

```

function object examples

less

```
template <class Object>
class less
{ public:
    bool operator()(const Object& lhs, const Object& rhs) const
    {return lhs < rhs;}
};
```

greater_equal

```
template <class T>
class greater_equal
{
public:
    bool operator() (const T& lhs, const T& rhs) const
    {return lhs >= rhs;}
};
```

```

template <class Object, class Comparator>
const Object & findMax( const vector<Object> & a, Comparator comp )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( comp( a[ maxIndex ], a[ i ] ) )
            maxIndex = i;

    return a[ maxIndex ];
}
template <class Object>
class less
{ public:
    bool operator()(const Object& lhs, const Object& rhs) const
        {return lhs < rhs;}
};

template <class T>
class greater
{
    public:
        bool operator() (const T& lhs, const T& rhs) const
            {return lhs > rhs;}
};

int main()
{
    vector<double> exam_scores;

    //some code to fill the exam_scores, etc
    double max = findMax( exam_scores, less<double>() );
    double min = findMax( exam_scores, greater<double>() );
}

```



```

template <class Object, class Comparator>
const Object & findMax( const vector<Object> & a, Comparator comp )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( comp( a[ maxIndex ], a[ i ] ) )
            maxIndex = i;

    return a[ maxIndex ];
}

```

```

template <class Object>

```

```

class less

```

```

{ public:
    bool operator()(const Object& lhs, const Object& rhs) const
    {return lhs < rhs;}
};

```

```

template <class Object >
const Object & findMax( const vector<Object> & a )
{
    return findMax( a, less< Object >() );
}

```

```

int main()
{
    vector<double> exam_scores;
    //some code to fill the exam_scores, etc
double max = findMax( exam_scores, less<double>() );
    double max = findMax( exam_scores );
}

```

Functor Example

Modified from http://www.stroustrup.com/bs_faq2.html#this

```
class Sum {  
    int val;  
public:  
    Sum(int i=0) :val(i) { }  
    operator int() const { return val; }    // extract value  
  
    int operator()(int i) { return val+=i; } // application  
};
```

functor

Capable of maintaining a state.
The state can be examined
from the outside (static variables
cannot be examined from the
outside.)

Conversion operator is a member
function. It cannot modify the member
variables. Note that the syntax is odd.
It has no return type:
operator type()const;

stringstream

#include <sstream>

- a stream object “wraps around” a device to allow >> (extraction operator) and << (insertion operator) to be used
- a stream object can convert the data to the “right” type automatically!

“The simplest method of tokenizing strings in C++ is to use the standard iostream capabilities. The std::getline() function has a very rudimentary capacity to break strings up using a single delimiter character each time you call the function.”

This code was inspired from Patrick's (our TA) code for splitting a string

```
vector<string> fields;  
string data = "101,,Van Cortlandt Park - 242 St,,40.889248,-73.89858";  
stringstream datastream(data);  
string field;  
  
getline(datastream, field, ',');  
fields.push_back(field);  
  
getline(datastream, field, ','); //empty string  
fields.push_back(std::move( field ));  
  
cout << fields[] << fields[0]<< endl;
```

This could be put into a loop!
`while (getline(datastream, field, ','))
{ // add your code here }`

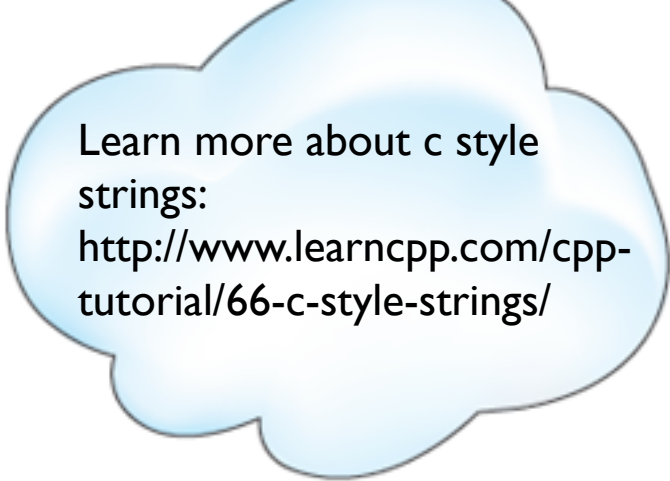
For more information:

<http://www.cplusplus.com/faq/sequences/strings/split/>

String to double?

```
string aString = "3.14159";
```

```
double d = atof( aString.c_str() );
```



Learn more about c style strings:
<http://www.learncpp.com/cpp-tutorial/66-c-style-strings/>

strings

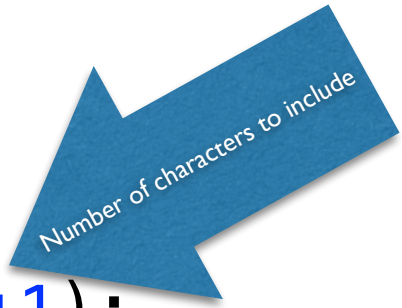
string::find_first_of

string::substr

```
vector<string> fields(10);
string s = "101,,Van Cortlandt Park - 242 St,,40.889248,-73.898583,,
int pos1 = s.find_first_of(',');
fields[0] = s.substr(0,pos1);

int pos2 = s.find_first_of(',', pos1+1);
fields[1]= s.substr(pos1+1, pos2-pos1-1); //notice this is the
//empty string

cout << fields[1];
cout << fields[0]<< endl;
```



Number of characters to include

For more information:

http://www.cplusplus.com/reference/string/string/find_first_of/

<http://www.cplusplus.com/reference/string/string/substr/>

Exception Handling

- mechanism to handle exceptional (unusual behavior)
 - I/O problem
 - subscript out of range
 - violation of a precondition
- Allow clean design of reliable code (avoids cluttering the “usual” code with lots of special cases)
- in C++ exceptions thrown in a “try” block are caught in a “catch” statement or propagated to block in which this is nested, or propagated to the caller

Exception Handling

- Try blocks
 - * enclose a throw expression/call to a function that throws an exception inside a try block
- Catch blocks
 - * follows a try block or call to a function that has a try block
- Throw expressions
 - * flag an unusual situation
 - * is of type void

C++ exception objects

- exceptions are objects
- hierarchy of exception classes inheriting from class exception
- an exception of class C is caught by a handler for class C or any of its ancestors
- handler can apply member functions of class C (or its ancestors) to the exception object.
- Example of underflow exception and its use in `vector::pop_back()`
- see try/catch on next page

```
try
{
    ...
    v.pop_back();
    ... // assuming pop_back was OK
}
catch (const UnderFlowException & e)
{
    cout << e.what() << endl;
    ....
}
```

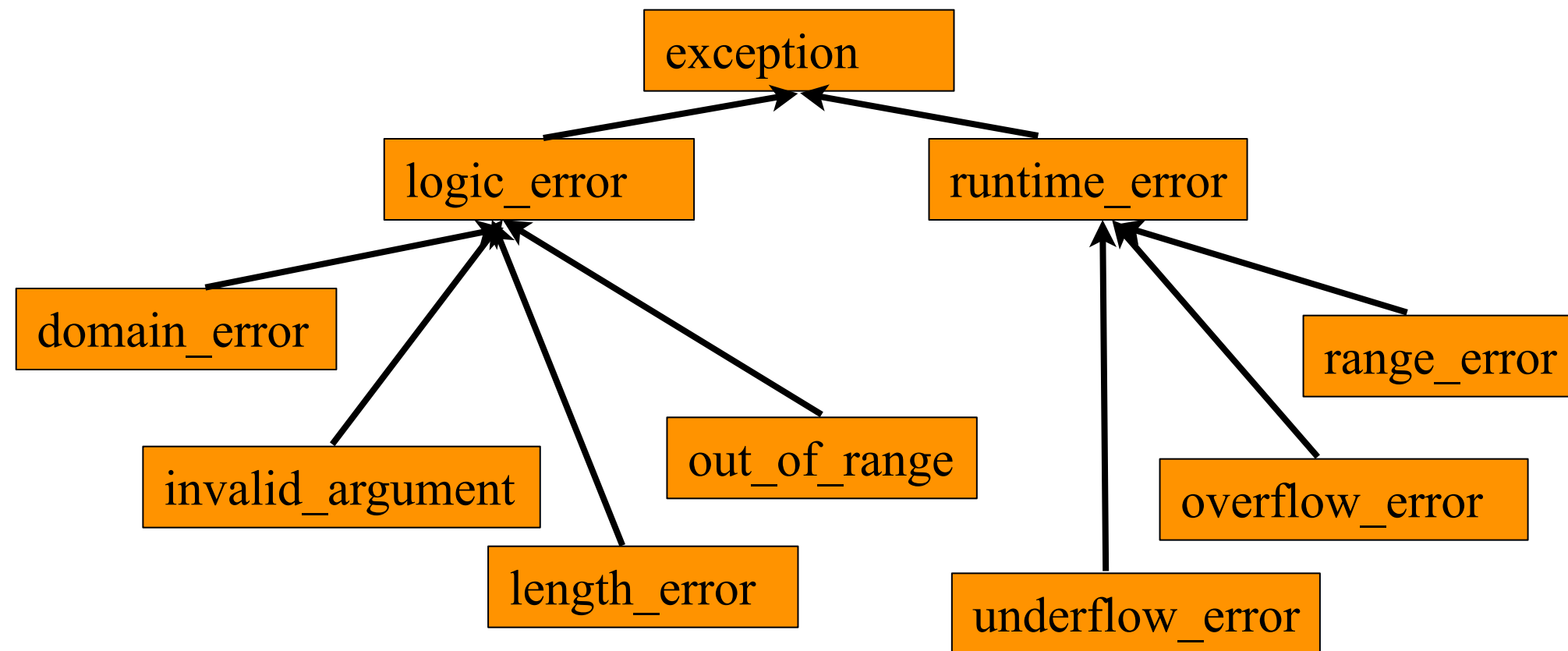
<http://www.cplusplus.com/doc/tutorial/exceptions/>

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

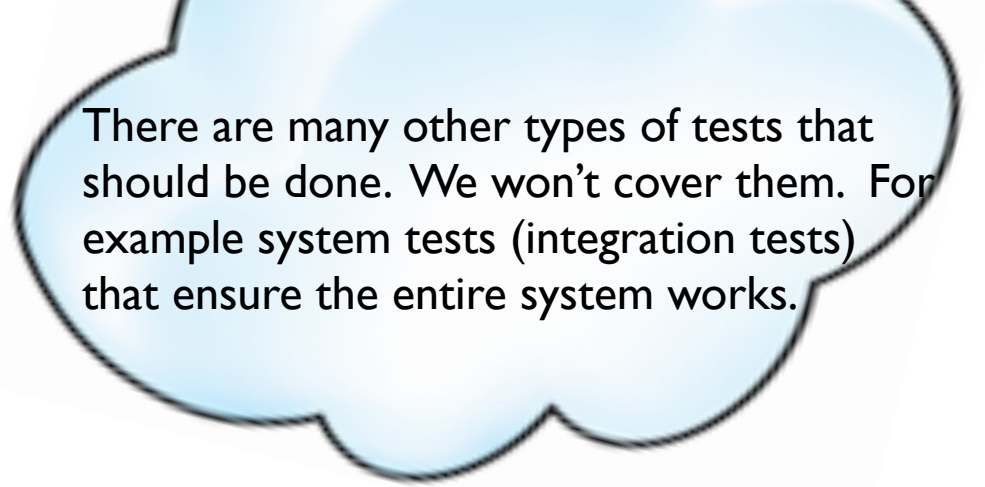
int main () {
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << "An exception occurred. Exception Nr. " << e << endl;
    }

    try {
        // code here
    }
    catch (int param) { cout << "int exception"; }
    catch (char param) { cout << "char exception"; }
    catch (...) { cout << "default exception"; }

    return 0;
}
```



Programming Principles and Practice Using C++
by Bjarne Stroustrup



There are many other types of tests that should be done. We won't cover them. For example system tests (integration tests) that ensure the entire system works.

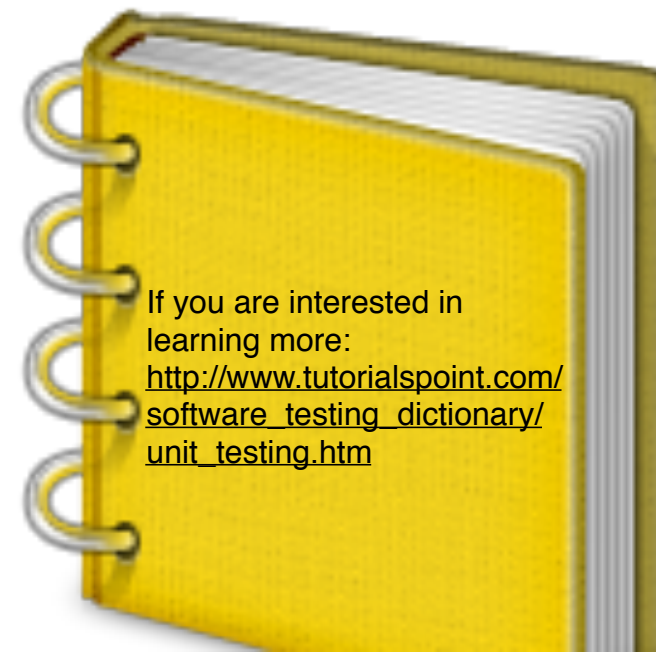
Unit Testing...

When you define a new class or function it is important to test your implementation for the conditions it is expected to be used for.

Untested code is often incorrect!

How to make sure your code works

- Stare at your code
- Prove that it is correct
- Wait till the graders tell you if it is correct
- or



Test your code

- Before coding, figure out the requirements (otherwise how can you know you succeeded!) and write the tests for each method/function/class (i.e. unit) before you start coding.
- Test each unit in isolation (thus when the entire code is put together it is more likely to work!)
- To test a unit - call it with sample inputs (including the null case!) to make sure it produces the correct result

Tips

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”

Dijkstra (1972 Turing Award Lecture)

- Start testing right away for each part of the code you write
- Test often. After you make a change, make sure everything that worked before still works...
- Remember to do the “corner cases”

What if I am lazy...

- You are in a CS class - get the computer to do the tests!
- We will use a ready to use a well tested standard testing framework

Using Catch

- copy catch.hpp into the same project directory that your main.cpp file can be found in.

https://raw.githubusercontent.com/philsquared/Catch/master/single_include/catch.hpp

catch.hpp is a header file. It provides preprocessor directives to manage the testing environment

- The line:

```
#define CATCH_CONFIG_MAIN
```

sets a variable that changes which parts of catch.hpp are executed by the preprocessor (thus end up in your compiled code)