CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

Code for Today

- On your VM:
  - $ mkdir except
  - $ cd except
  - $ wget http://ee.usc.edu/~redekopp/cs104/except.tar
  - $ tar xvf except.tar

Recall

- Remember the List ADT as embodied by the 'vector' class
- Now consider error conditions
  - What member functions could cause an error?
  - How do I communicate the error to the user?

```
#ifndef INTVECTOR_H
#define INTVECTOR_H

class IntVector {
 public:
  IntVector();
   ~IntVector();
   void push_back(int val);
   void insert(int loc, int val);
   bool remove(int val);
   int pop(int loc);
   int& at(int loc) const;
   bool empty() const;
   int size() const;
   void clear();
   int find(int val) const;
};

#endif
```

int_vector.h

Insert() Error

- What if I insert to a non-existent location

```
#include "int_vector.h"

void IntVector::insert(int loc, int val)
{
  // Invalid location
  if(loc > size_){
      // What should I do?


  }
}
```

int_vector.cpp

insert(7, 99);

0	1	2	3	4	5	6	7
30	51	52	53	54	10		

We can hijack the return value and return an error code.

But how does the client know what those codes mean? What if I change those codes?

get() Error

- What if I try to get an item at an invalid location

```
#include "int_vector.h"

int IntVector::get(int loc)
{
   // Invalid location
   if(loc >= size_){
       // What should I do?


   }
   return data_[loc];
}
```

int_vector.cpp

get(7);

0	1	2	3	4	5	6	7
30	51	52	53	54	10		

I can't use the return value, since it's already being used.

Could provide another reference parameter, but that's clunky.
int get(int loc, int &error);

EXCEPTIONS

Exception Handling

- When something goes wrong in one of your functions, how should you notify the function caller?
  - Return a special value from the function?
  - Return a bool indicating success/failure?
  - Set a global variable?
  - Print out an error message?
  - Print an error and exit the program?
  - Set a failure flag somewhere (like “cin” does)?
  - Handle the problem and just don't tell the caller?

What Should I do?

- There's something wrong with all those options...
  - You should **always** notify the caller something happened. Silence is not an option.
  - What if something goes wrong in a Constructor?
    - You don't have a return value available
  - What if the function where the error happens isn't equipped to handle the error
- All the previous strategies are **passive**. They require the caller to actively check if something went wrong.
- You shouldn't necessarily handle the error yourself...the caller may want to deal with it?

The "assert" Statement

- The ***assert*** statement allows you to make sure certain conditions are true and immediately halt your program if they're not
  - Good sanity checks for development/testing
  - Not ideal for an end product

```
#include <cassert>
int divide(int num,  int denom)
{
  assert(denom != 0);
  // if false, exit program

  return(num/denom);
}
```

Exception Handling

- Use C++ Exceptions!!
- Give the function caller a choice on how (or if) they want to handle an error
  - Don't assume you know what the caller wants
- Decouple and CLEARLY separate the exception processing logic from the normal control flow of the code
- They make for much cleaner code (usually)

```
// try function call
int retVal = doit();
if(retVal == 0){
}
else if(retVal < 0){

}
else {

}
```

Which portion of the if statement is for error handling vs. actual follow-on operations to be performed.

The "throw" Statement

- Used when code has encountered a problem, but the current code can't handle that problem itself
- 'throw' interrupts the normal flow of execution and can return a value
  - Like 'return' but *special*
  - If no piece of code deals with it, the program will terminate
  - Gives the caller the opportunity to catch and handle it
- What can you give to the throw statement?
  - Anything (int, string, etc.)! But some things are better than others...

```
int main(){
  int x;  cin >> x;
  divide(5,x);
}
int divide(int num,int denom)
{ if(denom == 0)
      throw denom;
  return(num/denom);
}
```

The "try" and "catch" Statements

- try & catch are the companions to throw
- A try block surrounds the calling of any code that may throw an exception
- A catch block lets you handle exceptions if a throw does happen
  - You can have multiple catch blocks…but think of catch like an overloaded function where they must be differentiated based on ***number*** and ***type*** of parameters.

```
int divide(int num,int denom)
{
  if(denom == 0)
     throw denom;
  return(num/denom);
}
```

```
try {
   x = divide(numerator,denominator);
}
catch(int badValue){
  cerr << "Can't use value " << badValue << endl;
  x = 0;
}
```

The "try" & "catch" Flow

- catch(...) is like an 'else' or default clause that will catch any thrown type
- This example is not good style...we would never throw something deliberately in our try block...it just illustrates the concept

```
try {
  cout << “This code is fine.” << endl;
  throw 0; //some code that always throws
  cout << “This will never print.” << endl;
}

catch(int &x) {
  cerr << “The throw immediately comes here.” << endl;
}
catch(string &y) {
  cerr << “We won't hit this catch.” << endl;
}
catch(...) {
  cerr << "Printed if the type thrown doesn't match";
  cerr << " any catch clauses" << endl;
}

cout << “Everything goes back to normal here.” << endl;
```

Catch & The Stack

- When an exception is thrown, the program will work its way up the stack of function calls until it hits a catch() block
- If no catch() block exists in the call stack, the program will quit

```
int divide(int num, int denom)
{
  if(denom == 0)
     throw denom;
  return(num/denom);
}
int f1(int x)
{
  return divide(x, x-2);
}

int main()
{
  int res, a;
  cin >> a;
  try {
    res = f1(a);
  }
  catch(int& v) {
    cout << "Problem!" << endl;
  }
}
```

Catch & The Stack

- When an exception is thrown, the program will work its way up the stack of function calls until it hits a catch() block
- If no catch() block exists in the call stack, the program will quit

```
int divide(int num, int denom)
{
  if(denom == 0)
     throw denom;
  return(num/denom);
}
int f1(int x)
{
  return divide(x, x-2);
}

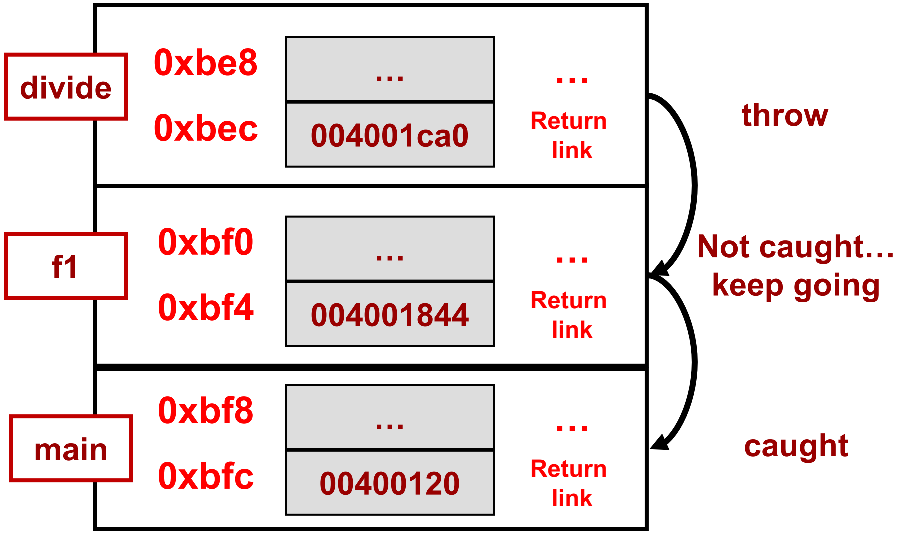
int main()
{
  int res, a = 2;
  try {
    res = f1(a);
  }
  catch(int& v) {
    cout << "Problem!" << endl;
  }
}
```

Catch & The Stack

- When an exception is thrown, the program will work its way up the stack of function calls until it hits a catch() block
- If no catch() block exists in the call stack, the program will quit

```
int divide(int num, int denom)
{
  if(denom == 0)
     throw denom;
  return(num/denom);
}
int f1(int x)
{
  return divide(x, x-2);
}

int main()
{
  int res, a;
  cin >> a;
  try {
    res = f1(a);
  }
  catch(int& v) {
    cout << "Caught here" << endl;
  }
}
```



Catch & The Stack

- You can use catch() blocks to actually resolve the problem

```
int divide(int num, int denom)
{
  if(denom == 0)
     throw denom;
  return(num/denom);
}
int f1(int x)
{
  return divide(x, x-2);
}

int main()
{
  int res, a;
  cin >> a;
  while(1){
    try {
      res = f1(a);
      break;
    }
    catch(int& v) {
      cin >> a;
    }
  }
}
```

What Should You "Throw"

- Usually, don't throw primitive values (e.g. an “int”)
  - `throw 123;`
  - The value that is thrown may not always be meaningful
  - Provides no other context (what happened & where?)
- Usually, don't throw “string”
  - `throw “Someone passed in a 0 and stuff broke!”;`
  - Works for a human, but not much help to an application
- Use a class, some are defined already in <stdexcept> header file
  - ```
    throw std::invalid_argument("Denominator can't be 0!");
    throw std::runtime_error("Epic Fail!");
    ```
  - Serves as the basis for building your own exceptions
  - Have a method called “what()” with extra details
  - http://www.cplusplus.com/reference/stdexcept/
  - You can always make your own exception class too!
    ```

Exception class types

- exception
  - logic_error (something that could be avoided by the programmer)
    - invalid_argument
    - length_error
    - out_of_range
  - runtime_error (something that can't be detected until runtime)
    - overflow_error
    - underflow_error

```
#include <iostream>
#include <stdexcept>
using namespace std;
int divide(int num, int denom)
{
  if(denom == 0)
    throw invalid_argument("Div by 0");
  return(num/denom);
}
int f1(int x)
{
  return divide(x, x-2);
}

int main()
{
  int res, a;
  cin >> a;
  while(1){
    try {
      res = f1(a);
      break;
    }
    catch(invalid_argument& e) {
      cout << e.what() << endl;
      cin >> a;
    }
  }
}
```

cin Error Handling (Old)

```
#include <iostream>
using namespace std;
int main()
{
  int number = 0;
  cout << "Enter a number: ";
  cin >> number;
  if(cin.fail()) {
    cerr << "That was not a number." << endl;
    cin.clear();
    cin.ignore(1000,'\n');
  }
}
```

USC Viterbi School of Engineering

cin Error Handling (New)

```
#include <iostream>

using namespace std;

int main()
{
  cin.exceptions(ios::failbit); //tell "cin" it should throw
  int number = 0;
  try {
    cout << "Enter a number: ";
    cin >> number;       // cin may throw if can't get an int
  }
  catch(ios::failure& ex) {
    cerr << "That was not a number." << endl;
    cin.clear();

    // clear out the buffer until a '\n'
    cin.ignore( std::numeric_limits<int>::max(), '\n');
  }

}
```

Vector Indexing (Old Way)

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

int main()
{
  int index = -1;
  vector<int> list(5);

  if(index < 0 || index >= list.size()) {
    cerr << "Your index was out of range!" << endl;
  }
  else {
    cout << "Value is: " << list[index] << endl;
  }

}
```

Vector Indexing (New Way)

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

int main()
{
  int index = -1;
  vector<int> list(5);
  try {
    cout << "Value is: " << list[index] << endl;
  }
  catch(out_of_range &ex) {
    cerr << "Your index was out of range!" << endl;
  }

}
```

Notes

- Where does break go in each case?
- In 2nd option, if there is an exception, will we break?
  - No, an exception immediately ejects from the try {…} and goes to the catch {…}

```
do {
  cout << "Enter an int: ";
  cin >> x;
  if( ! cin.fail()){
   break;
  }
  else {
    cin.clear();
    cin.ignore(1000,'\n');
  }
} while(1);
```

```
do {
  cin.exceptions(ios::failbit);
  cout << "Enter an int: ";
  try {
    cin >> x;
    break;
  }
  catch(ios::failure& ex) {
    cerr << "Error" << endl;
    cin.clear();
    cin.ignore(1000,'\n');
  }
} while(1);
```

Other "throw"/"catch" Notes

- Do not use throw from a destructor. Your code will go into an inconsistent (and unpleasant) state. Or just crash.
- You can re-throw an exception you've caught
  - Useful if you want to take intermediate action, but can't actually handle the exception
  - Exceptions will propagate up the call hierarchy ("Unwinding the call stack")

```
#include <iostream>
#include <stdexcept>
using namespace std;
int divide(int num, int denom)
{
  if(denom == 0)
    throw invalid_argument("Div by 0");
  return(num/denom);
}
int f1(int x)
{
  int y;
  try { y = divide(x, x-2); }
  catch(invalid_argument& e){
    cout << "Caught first here!" << endl;
    throw;  // throws 'e' again
} }

int main()
{
  int res, a;
  cin >> a;
  while(1){
    try {
      res = f1(a);
      break;
    }
    catch(invalid_argument& e) {
      cout << "Caught again" << endl;
      cin >> a;
} } }
```

FUNCTION TEMPLATES

Overview

- C++ Templates allow alternate versions of the same code to be generated for various data **types**

How To's

- Example reproduced from: http://www.cplusplus.com/doc/tutorial/templates/
- Consider a max() function to return the max of two int's
- But what about two double's or two strings
- Define a generic function for any type, T
- Can then call it for any type, T, or let compiler try to implicitly figure out T

```
int max(int a, int b)
{
  if(a > b) return a;
  else return b;

}

double max(double a, double b)
{
  if(a > b) return a;
  else return b;
}
```

Non-Templated = Multiple code copies

```
template<typename T>
T max(const T& a, const T& b)
{
  if(a > b) return a;
  else return b;
}
int main()
{
  int x = max<int>(5, 9); //or
  x = max(5, 9); // implicit max<int> call
  double y = max<double>(3.4, 4.7);
  // y = max(3.4, 4.7);
}
```

Templated = One copy of code

CLASS TEMPLATES

Templates

- We've built a list to store integers
- But what if we want a list of double's or string's or other objects
- We would have to define the same code but with **different types**
  - What a waste!
- Enter C++ Templates
  - Allows the one set of code to work for any type the programmer wants
  - The type of data becomes a parameter

```
#ifndef LIST_INT_H
#define LIST_INT_H
struct IntItem {
  int val; IntItem* next;
};
class ListInt{
 public:
   ListInt();   // Constructor
   ~ListInt();   // Destructor
   void push_back(int newval); ...
 private:
   IntItem* head_;
};
#endif
```

```
#ifndef LIST_DBL_H
#define LIST_DBL_H
struct DoubleItem {
  double val; DoubleItem* next;
};
class ListDouble{
 public:
   ListDouble();   // Constructor
   ~ListDouble();   // Destructor
   void push_back(double newval); ...
 private:
   DoubleItem* head_;
};
#endif
```

Templates

- Allows the type of variable in a class or function to be a parameter specified by the programmer
- Compiler will generate separate class/struct code versions for any type desired (i.e instantiated as an object)
  - LList<int> my_int_list causes an 'int' version of the code to be generated by the compiler
  - LList<double> my_dbl_list causes a 'double' version of the code to be generated by the compiler

```
// declaring templatized code
template <typename T>
struct Item {
  T val;
  Item<T>* next;
};

template <typename T>
class LList {
public:
   LList();   // Constructor
   ~LList();  // Destructor
   void push_back(T newval); ...
 private:
   Item<T>* head_;
};


// Using templatized code
//  (instantiating templatized objects)
int main()
{
  LList<int> my_int_list;
  LList<double> my_dbl_list;

  my_int_list.push_back(5);
  my_dbl_list.push_back(5.5125);

  double x = my_dbl_list.pop_front();
  int y = my_int_list.pop_front();
  return 0;
}
```

Templates

- Writing a template
  - Precede class with:

    template <typename T>

    Or

    template <class T>
  - Use T or other identifier where you want a generic type
  - Precede the definition of each function with template <typename T>
  - In the scope portion of the class member function, add <T>
  - Since Item and LList are now templated, you can never use Item and LList alone
    - You must use Item<T> or LList<T>

```
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
  T val; Item<T>* next;
};

template <typename T>
class LList{
 public:
   LList();   // Constructor
   ~LList();  // Destructor
   void push_back(T newval);
   T& at(int loc);
 private:
   Item<T>* head_;
};


template<typename T>
LList<T>::LList()
{ head_ = NULL;
}

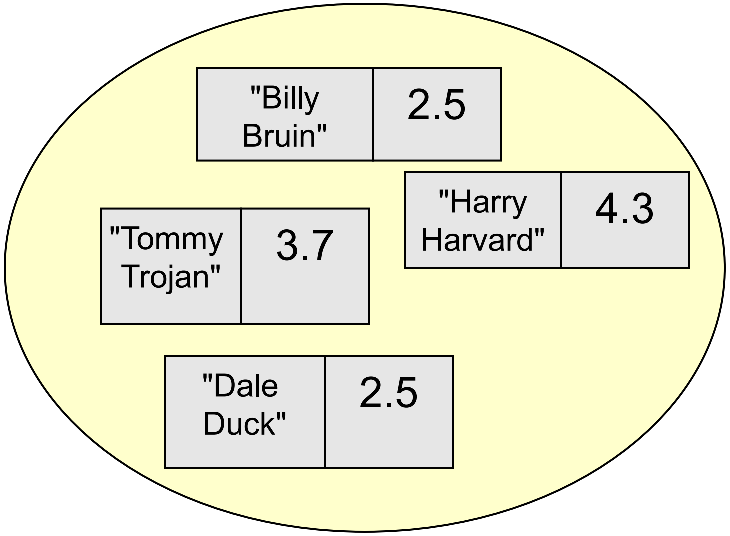
template<typename T>
LList<T>::~LList()
{ }

template<typename T>
void LList<T>::push_back(T newval)
{  ... }

#endif
```

Exercise

- Recall that maps/dictionaries store key,value pairs
  - Example: Map student names to their GPA
- How many key,value type pairs are there?
  - string, int
  - int, double
  - Etc.
- Would be nice to create a generic data structure
- Define a Pair template with two generic type data members



Templates

- Usually we want you to write the class definition in a separate header file (.h file) and the implementation in a .cpp file
- **Key Fact:** Templated classes must have the implementation <u>**IN THE HEADER FILE!**</u>
- **Corollary**: Since we don't compile .h files, <u>you cannot compile a templated class separately</u>
- Why? Because the compiler would have no idea what type of data to generate code for and thus what code to generate

```
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
  T val; Item<T>* next;
};

template <typename T>
class LList{
 public:
   LList();   // Constructor
   ~LList();   // Destructor
   void push_back(T newval);
private:
   Item<T>* head_;
};
#endif
```

List.h

```
#include "List.h"

template<typename T>
LList<T>::push_back(T newval)
{
  if(head_ = NULL){
    head_ = new Item<T>;
    // how much memory does an Item
    //  require?
  }
}
```

List.cpp

USC Viterbi School of Engineering

Templates

- The compiler will generate code for the type of data in the file where it is instantiated with a certain type

Main.cpp

```
#include "List.h"

int main()
{
  LList<int> my_int_list;
  LList<double> my_dbl_list;

  my_int_list.push_back(5);
  my_dbl_list.push_back(5.5125);

  double x = my_dbl_list.pop_front();
  int y = my_int_list.pop_front();
  return 0;
}

// Compiler will generate code for
LList<int> when compiling main.cpp
```

List.h

```
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
  T val; Item<T>* next;
};

template <typename T>
class LList{
 public:
   LList();   // Constructor
   ~LList();  // Destructor
   void push_back(T newval);
   T& at(int loc);
 private:
   Item<T>* head_;
};

template<typename T>
LList<T>::LList()
{ head_ = NULL;
}

template<typename T>
LList<T>::~LList()
{ }

template<typename T>
void LList<T>::push_back(T newval)
{  ... }

#endif
```

The devil in the details

C++ TEMPLATE ODDITIES

Templates & Inheritance

- For various reasons the compiler may have difficulty resolving members of a templated base class
- When accessing members of a templated base class provide the full scope or precede the member with this->

```
#include "llist.h"
template <typename T>
class Stack : private LList<T>{
 public:
   Stack();  // Constructor
   void push(const T& newval);
   T const & top() const;
};

template<typename T>
Stack<T>::Stack() : LList<T>()
{ }

template<typename T>
void Stack<T>::push(const T& newval)
{  // call inherited push_front()
   push_front(newval); // may not compile
   LList<T>::push_front(newval); // works
   this->push_front(newval);     // works
}

template<typename T>
T const &Stack<T>::top() const
{ // assume head is a protected member
  if(head) return head->val; // may not work
  if(LList<T>::head)          // works
     return LList<T>::head->val;
  if(this->head)              // works
     return this->head->val;
}
```

"typename" & Nested members

- For various reasons the compiler may have difficulty resolving nested types of a templated class whose template argument is still generic (i.e. T vs. int)
- Precede the nested type with the keyword 'typename'

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

template <typename T>
class Stack {
public:
  void push(const T& newval)
    { data.push_back(newval); }
  T& top();
private:
  std::vector<T> data;
};

template <typename T>
T& Stack<T>::top()
{
  vector<T>::iterator it = data.end();
  typename vector<T>::iterator it = data.end();
  return *(it-1);
}

int main()
{
  Stack<int> s1;
  s1.push(1); s1.push(2); s1.push(3);
  cout << s1.top() << endl;
  return 0;
}
```

It's an object, it's a function...it's both rolled into one!

WHAT THE "FUNCTOR"

Who you gonna call?

- Functions are "called" by using parentheses () after the function name and passing some arguments
- Objects use the . or -> operator to access methods of an object
- Calling an object doesn't make sense
  - You call functions not objects
  - Or can you?

```
class ObjA {
 public:
  ObjA() {}
  void action();
};

int main()
{
  ObjA a;
  ObjA *aptr = new ObjA;
  // This makes sense:
  a.action();
  aptr->action();

  // This doesn't make sense
  a();

  // a is already constructed, so
  // it can't be a constructor call
  // So is it illegal?


  return 0;
}
```

Operator()

- Calling an object does make sense when you realize that () is an operator that can be overloaded
- For most operators their number of arguments is implied
  - operator+ takes an LHS and RHS
  - operator-> takes no args
- You can overload operator() to take any number of arguments of your choosing

```
class ObjA {
 public:
  ObjA() {}
  void action();
  void operator()() {
    cout << "I'm a functor!";
    cout << endl;
  }
  void operator()(int &x) {
    return ++x;
  }
};

int main()
{
  ObjA a;
  int y = 5;
  // This does make sense!!
  a();
  // prints "I'm a functor!"

  // This also makes sense !!
  a(y);
  // y is now 6
  return 0;
}
```

Functors: What are they good for?

- I'd like to use a certain class as a key in a map or set
- Maps/sets require the key to have…
  - A less-than operator
- Guess I can't use ObjA
  - Or can I?

```
class ObjA {
 public:
  ObjA(...) {}
  void action();
  int getX() { return x; }
  string getY() { return y; }
 private:
  int x; string y;
};
```

obja.h – Someone else wrote it

```
int main()
{
  // I'd like to use ObjA as a key
  // Can I?
  map<ObjA, double> mymap;

  ObjA a(5,"hi");
  mymap[a] = 6.7;
  return 0;
}
```

Functors: What are they good for?

- Map template takes in a third template parameter which is called a "Compare" object
- It will use this type and assume it has a functor [i.e. operator()] defined which can take two key types and compare them

```
class ObjA {
 public:
  ObjA(...) {}
  void action();
  int getX() { return x; }
  string getY() { return y; }
 private:
  int x; string y;
};
```

obja.h – Someone else wrote it

```
struct ObjAComparer
{
  bool operator()(const ObjA& lhs,
                  const ObjA& rhs) const
  { return lhs.getX() < rhs.getX(); }
};

int main()
{
  // Now we can use ObjA as a key!!!!
  map<ObjA, double, ObjAComparer> mymap;

  ObjA a(5,"hi");
  mymap[a] = 6.7;
  return 0;
}
```

More Uses

- Functors can act as a user-defined "function" that can be passed as an argument and then called on other data items
- Below is a modified count_if template function (from STL <algorithm>) that counts how many items in a container meet some condition

```
template <typename InputIterator, typename Cond>
int count_if2 (InputIterator first,
               InputIterator last,
               Cond pred)
{ int ret = 0;
  for( ; first != last; ++first){
    if ( pred( *first ) )
      ++ret;
  }
  return ret;
}
```

More Uses

- Functors can act as a user-defined "function" that can be passed as an argument and then called on other data items
- You need to define your functor struct [with the operator()], declare one and pass it to the function

```
struct NegCond {
  bool operator(int val) { return val < 0; }
};

int main()
{ std::vector<int> myvec;

  // myvector: -5 -4 -3 -2 -1 0 1 2 3 4
  for (int i=-5; i<5; i++)
     myvec.push_back(i);
  NegCond c;
  int mycnt = count_if2 (myvec.begin(),
                         myvec.end(),
                         c);
  cout << "myvec contains " << mycnt;
  cout << " negative values." << endl;
  return 0;
}
```

Final Word

- Functors are all over the place in C++ and STL
- Look for them and use them where needed
- References
  - http://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html
  - http://stackoverflow.com/questions/356950/c-functors-and-their-uses

Practice

- SlowMap
  - wget http://ee.usc.edu/~redekopp/cs104/slowmap.cpp
- Write a functor so you can use a set of string*'s and ensure that no duplicate strings are put in the set
  - http://bits.usc.edu/websheets/index.php?folder=cpp/templates
  - strset