CSCI 104 Exceptions

Mark Redekopp

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?

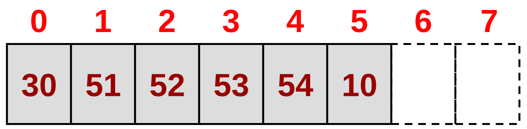
```
#ifndef INTVECTOR_H
#define INTVECTOR_H

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

insert(7, 99);



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?

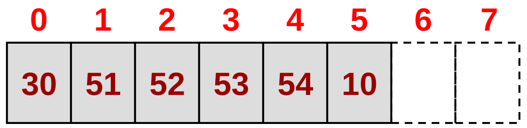
```
#include "int_vector.h"

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


  }
}
```

int_vector.cpp

get() Error

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

get(7);



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);

```
#include "int_vector.h"

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


  }
  return data_[loc];
}
```

int_vector.cpp

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

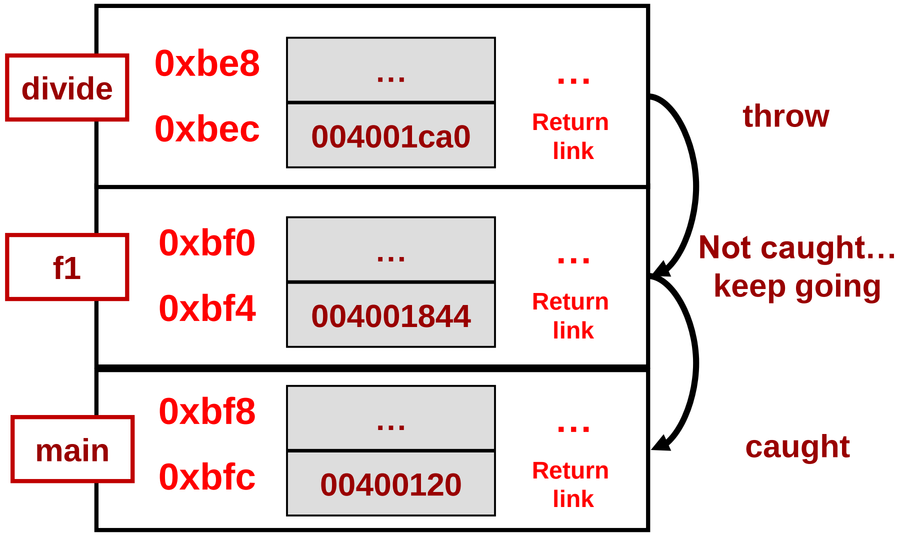
- When an exception is thrown, the program will work its way up the stack of function calls until it hits a catch() block
- If no catch() block exists in the call stack, the program will quit

```
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');
  }
}
```

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;
} } }
```

Other Exceptions Notes

- Think about where you want to handle the error
  - If you can handle it, handle it…
  - If you can't, then let the caller

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

int f1(char* filename)
{
  ifstream ifile;
  ifile.exceptions(ios::failbit);
  // will throw if opening fails
  ifile.open(filename);

  // Should you catch exception here
  // Or should you catch it in main()
}

int main(int argc, char* argv[])
{
  readFile(argv[1]);
  ...
}
```