CSCI 103 – Unit 5g Exceptions

CSCI 103L Teaching Team

Thinking About Errors

- Consider the `vector<T>` class
- Now consider error conditions
  - What member functions could cause an error?
  - How do I communicate the error to the user?

```
#ifndef VECTOR_H
#define VECTOR_H

template <typename T>
class vector {
 public:
  vector();
  ~vector();
  bool empty() const;
  int size() const;

  void push_back(const T& val);

  void insert(size_t loc, const T& val);
  void erase(size_t loc);

  T& at(size_t loc);
  const T& at(size_t loc) const;
  ...
};
#endif
```

Vector Class
(Slightly modified from actual C++ version)

Thinking About Errors

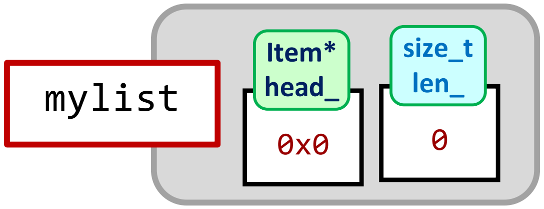
- Now consider the `ListInt` class
- Now consider error conditions
  - What member functions could cause an error?
  - How do I communicate the error to the user?

```
#ifndef LISTINT_H
#define LISTINT_H

struct Item {
  int val;
  Item* next;
};

class ListInt {
public:
  ListInt();
  ~ListInt();
  void push_back(int v);
  void pop_back();
  void pop_front();
  int front() const;
  int back() const;

  // Get the value at the i-th location
  int& get(size_t i);
  const int& get(size_t i) const;

private:
  Item* head_;
  size_t len_;
};
#endif
```

pop_front() Error

- What if I erase a non-existent location

```
mylist.pop_front();
```



We can use 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 "listint.h"

void ListInt::pop_front()
{
  // Empty list check?
  if(head_ == NULL){
      // What should I do?

  }
  else {
      Item* temp = head_;
      head_ = head_->next;
      delete temp;
  }
}
```

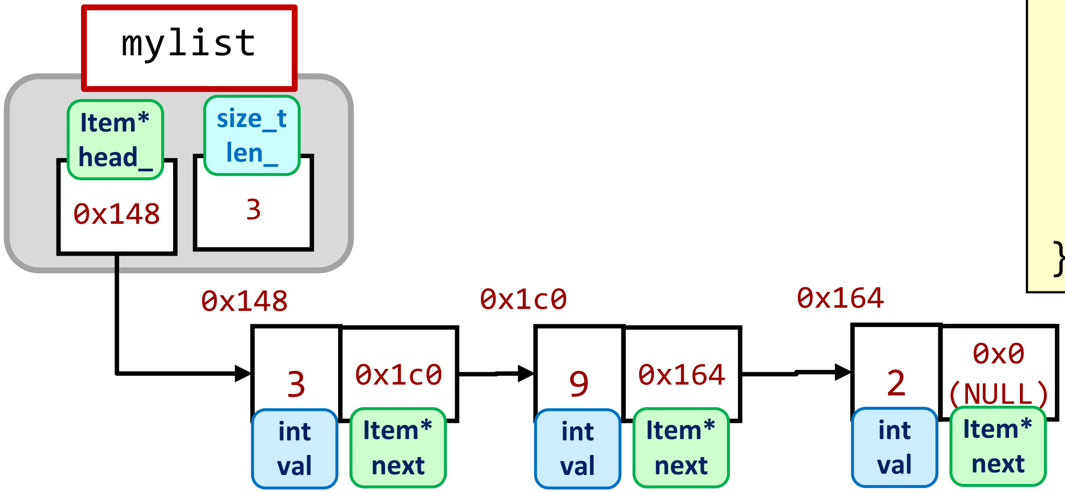
listint.cpp

get() Error

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

```
#include "listint.h"

int ListInt::get(size_t i) const
{
  // is i a valid index?
  if( i >= len_  ){
      // What should I do?
  }
  else {
      Item* temp = head_;
      while(i != 0) {
        temp = temp->next;
        i--;
      }
      return temp->val;
  }
}
```

mylist.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);
```

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);
}
```

Topics

- What are exceptions
  - When/where to use them
- Exception syntax in C++
  - try, throw, catch
- Processing (handling) exceptions
  - Uncaught exceptions
  - Unexpected exceptions
- Stack unwinding
- Exception objects

What are exceptions

- An exception is something exceptional
  - Not expected to happen frequently
- In programming an exception (error) occurs when a problem happens that is not handled by the normal flow of your program
  - Classic examples: divide by zero, out-of-memory
- How to deal with exceptions?

Terms

- Exception
  - Something has (or would go wrong)
- Signaling (throwing)
  - Indicating that something has gone wrong
- Handling (catching)
  - Dealing with the fact that something has gone wrong

Dealing with exceptions

- When writing programs we should expect *some* errors to occur
- We can:
  - Ignore them
    - Not appropriate for “real” software
  - Prevent them
    - Validating all inputs in all cases is very hard
    - Problems outside our control (e.g. out of memory)
  - Use error codes, return error values
    - Incurs processing overhead
    - Like validation, hard to code for all possible cases
  - Use exceptions and exception handling
    - C++ Exceptions

C++ Exceptions

- Standardized way to process errors
  - Works across interface boundaries (classes, functions), compatible with encapsulation/isolation
- Defines syntax and semantics for signaling an error has occurred (throw)
- Defines syntax and semantics for detecting and handling errors (catch)
  - These are separate parts of the program

Programmatic error handling

- Code structure without exceptions

```
err = doTask1()

if err: process error

err2 = doTask2()

if err2: process error

err3 = doTask3()

if err3: process error

. . .
```

- Intermingled code/error processing makes code:
  - Hard to read
  - Hard to debug
  - Hard to update/maintain
  - Incurs processing overhead
    - Checking for errors when errors *should* be rare

With exceptions

- Regular or “main line” code does not expect errors, but signals when they do occur
- Main-line code and exception handlers when separate are easier to read and maintain
  - Main line code detects and error, throws and then lets someone else deal with it
- Separate error handling into dedicated exception handlers
- Similar to a classes, ”users” of code (handlers) are separate from “implementors” (throwers of exceptions)
- User decides to handle:
  - No exceptions
  - All exceptions
  - All exceptions of a type
  - All “related” exceptions
- “Handling” can be
  - Ignore the exception
  - Recover/restart
  - Pass exceptions “up the stack”
  - Filtering exceptions

Exceptions Design Pattern

- Better/easier to assume exceptions never happen
  - Write your program assuming no errors
- Then add code to detect and signal exceptions
- Then add code (if necessary) to handle exceptions
- Don’t overuse exceptions – reserve for exceptional cases
  - Shouldn’t turn into alternate for regular control-flow
  - Appropriate data validation is OK
  - If local code can easily handle the error, don’t throw

Why add handlers “if necessary”?

- Code reuse is often major goal of software projects
- What to do with an error often depends on who is ”using” your code
- If you’re writing a class or a library of functions you don’t know if an exception is truly an error or not
  - So when something goes wrong, we signal that there is an exception
- If you’re using a class or a library you can decide what to do with the error
  - Throwing vs. catching are different operations that might be separated by time or across different teams, etc.
- When detecting errors we don’t want to dictate how they are handled
- If the function is used in different programs, or different ways in the same program, each use might require different actions to be taken when an exception occurs

Exception Handling

- 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 status = doit();
if(status == 0){
   // Code A
}
else if(status < 0){
   // Code B
}
else {
   // Code C
}
```

Which portion of the if..else statement is the normal case(s) and which are the error-handling case(s)

Basic C++ Exception Syntax

- C++ uses three keywords for exceptions
  - try, catch, throw
  - ”try” this code, “throw” an error, “catch” that error

```
//somewhere in main…
try {
    //main line code
    int val = f1();
    //f1() *could* have an error
}
catch (Ex e) {
    //if an error occurs execution jumps here
    //so we can process it
}
```

```
//somewhere else in your code…
int f1()
{
    //regular code here
    //oh no! something is wrong
    throw Ex();
}
```

The "throw" Statement

- Use the **throw** statement when code has encountered a problem, but cannot handle that problem itself
- **throw** HALTS the function and returns an "error" value
  - Like 'return' but *special*. ***Immediately ENDS the executing function!***
  - 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 "return" with the throw statement?
  - Anything (int, string, etc.)! But some things are better than others...
  - Doesn't have to match the return type

```
int main()
{
  int x;
  cin >> x;
  cout << divide(5,x) << endl;
  return 0;
}

int divide(int num, int denom)
{
  if(denom == 0)
     throw "Denom is 0";
  // normal case
  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;
  // normal case
  return(num/denom);
}
```

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

// use x
```

Multiple Errors (throws and catches)

- A function can have multiple throw statements (though it will exit when the first executes) and each can throw a different type
- A try block can have multiple catch statements (one per type)

```
void swap(int arr[], int len, int i, int j)
{
   if(i >= len) {
     // throw a string for no good reason
     throw string("bad index i");
   }
   if(j >= len) {
     // throw an int for no good reason
     throw -1;
   }
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}
```

```
int main()
{
   int data[5] = {1,2,3,4,5};
   int i, j;
   cin >> i >> j;
   try {
      swap(data, 5, i, j);

      for(int i=0; i < 5; i++) {
        cout << data[i] << " ";
      }
      cout << endl;
   }
   catch (string& e) {
      cout << e << endl;
   }
   catch (int e) {
      cout << "Bad j – " << e << endl;
   }
   return 0;
}
```

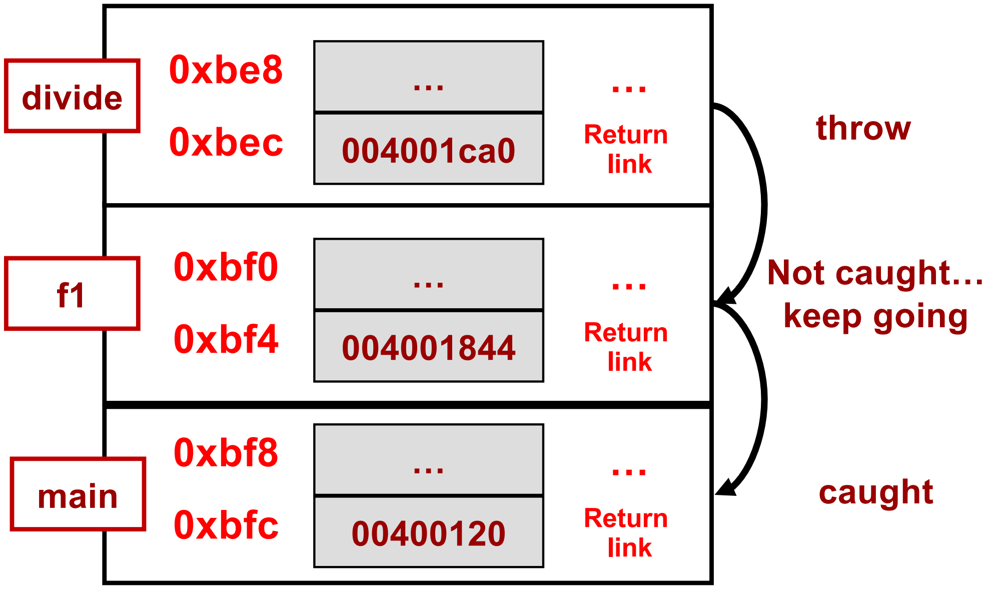
Catch Block Notes

- Should catch by reference (avoid a copy)
- Will try the catch blocks of a try statement in order until it matches the type of what is thrown
  - More about this when inheritance is used with the thrown exception types
- **catch(…)** is like an 'else' or default clause that will catch any thrown type

```
int main()
{
   int data[5] = {1,2,3,4,5};
   int i, j;
   cin >> i >> j;
   try {
      swap(data, 5, i, j);

      for(int i=0; i < 5; i++) {
        cout << data[i] << " ";
      }
      cout << endl;
   }
   catch (string& e) {
      cout << e << endl;
   }
   catch (int e) {
      cout << "Bad j – " << e << endl;
   }
   catch (...) {
      cout << "Unknown exception" << endl;
   }
   return 0;
}
```

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 string("div-by-0");
  return(num/denom);
}

// some arbitrary "middle" function
int f1(int x)
{
  return divide(x, x-2); // arbitrary
}

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

Stack Unwinding

- When an exception is not caught in the same scope as the throw we have to “unwind” the stack
  - In the DivByZero example we threw in fdivide() but caught in main (different scopes)
- We go down the stack looking for a matching catch() {} block
  - If we find one we “unwind” all of the intervening functions (local variables go out of scope, destructors called)
- If we never find one (i.e. we get all the way back to main() we have an “uncaught exception”
  - Stack is *not* unwound
  - terminate() is called, program ends

terminate()

- Special function that terminates (stops) your program with a message
- Used when things go wrong with exception handling

Catch & The Stack

- You can use catch() blocks to resolve the problem
- The while loop and the cin in the catch statement will cause the program to keep getting new inputs until f1(a) does NOT throw

```
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;
    }
  }
  // We know we have a good result
  ...
}
```

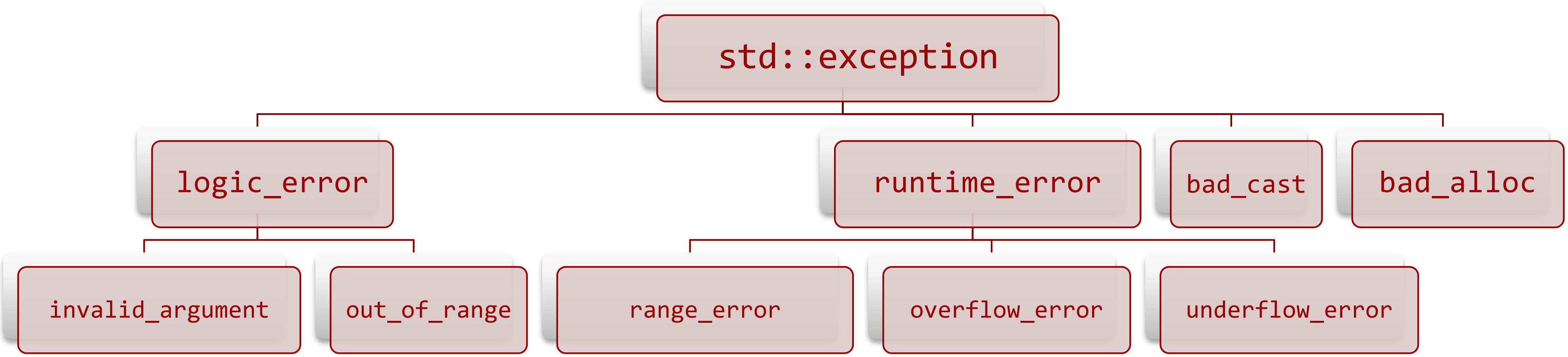
What Should You "Throw"

- Usually, don't throw primitive values (e.g. an int, double, etc.) or a string
  - `throw 123;`
    - The value that is thrown may not always be meaningful and provides little context
  - `throw "Someone passed in a 0 and stuff broke!";`
    - Easy for humans to read but hard for computer to understand
- 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!");
    ```
  - http://www.cplusplus.com/reference/stdexcept/
  - Serves as the basis for building your own exceptions
  - You can always make your own exception class too!

C++ Exception Hierarchy

- Using an inheritance hierarchy is recommended and C++ provides one in <stdexcept>
- All exceptions are derived from `std::exception`
  - `bad_alloc` is thrown by new if not enough memory is available
  - `out_of_range` is thrown by `vector::at` if bad index is given
  - `logic_error`: errors that the programmer should have been able to avoid
  - `runtime_error`: errors that could not detected until the program runs



Standard C++ Exception Practice

- Exceptions are instances of a class
  - Usually derived from C++ standard exceptions
  - Not just whacky control flow, thrower gets to send an object to the handler
- If code detects an error
  - "throw" an appropriate instance depending on what went wrong
- Calling code (or using in the case of classes) can choose to "catch" exceptions they care about
  - Matching based on the exception instance type

Why throw classes?

- Technically you can "throw" anything (any type)
- Throwing a specific class allows you to pack detailed information about what went wrong in the instance data members
- Catching code can match based on the class type and will know exactly what went wrong

C++ Exception Hierarchy

- std::exception defines a what() function that returns a message the that can be given to the constructor of a derived exception and retrieved when caught

```
class exception {
public:
  exception ();
  exception (const exception&);
  exception& operator= (const exception&);
  virtual ~exception();
  virtual const char* what() const;
}
```

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

int divide(int num, int denom)
{
  if(denom == 0)
    throw range_error("Div by 0");
  return(num/denom);
}

int main()
{
  int res, n, d;
  cin >> n >> d;
  while(1){
    try {
      res = divide(n,d);
      cout << "Result is " << res << endl;
      break;
    }
    catch(range_error& e) {
      cout << e.what() << endl;
      cin >> n >> d;
    }
  }
  return 0;
}
```

You Can/Should Define Your Own

- You can define your own exceptions
- Because catch statements execute based on the **TYPE** of exception thrown, it is recommended to make your own exception types (structs/classes)
- It is recommended you inherit from std::exception or one of its subclasses

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

struct DivByZero : public std::range_error {
   DivByZero(const char* what) :
     range_error(what) { }
};

int divide(int num, int denom) {
  if(denom == 0)
    throw DivByZero("Div by 0");
  return(num/denom);
}

int main() {
  int res, n, d;
  cin >> n >> d;
  while(1){
    try {
      res = divide(n,d);
      break;
    }
    catch(DivByZero& e) {
      cout << e.what() << endl;
      cin >> n >> d;
    }
  }
  return 0;
}
```

You Can/Should Define Your Own

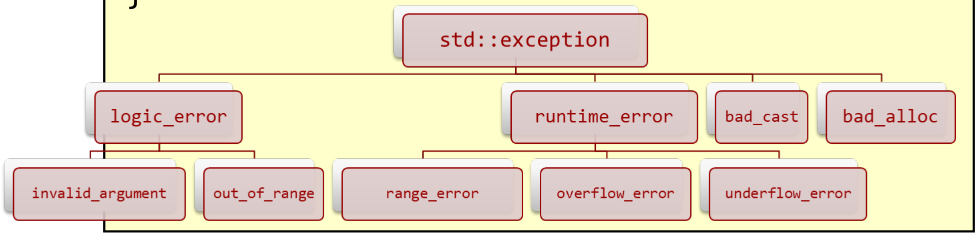
- Best practice: Order your catch statements from the **MOST** derived type first to the base type
- Why?
  - Recall: DivByZero is-a range_error
  - So a DivByZero can be passed to a range_error

```
try {
   doTask();
}
catch(range_error& e) {
  // Handle a more generic range_error
}
catch(DivByZero& e) {
  // Handle divide by 0
}
...
```

Incorrect catch ordering

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

int main()
{
    try {
       doTask();
    }
    catch(DivByZero& e) {
      // Handle divide by 0
    }
    catch(range_error& e) {
      // Handle a more generic range_error
    }
    catch(exception& e) {
      // Handle any error derived
      //   from std::exception
    }
    catch(...) {
      // Handle any exception not derived
      //   from std::exception
    }
    return 0;
}
```

Correct catch ordering



Re-Throwing Exceptions

- You may want to catch an exception to take some intermediate action, but you can't fully process the error and so you can **re-throw** it.
  - May want to log some error in the intermediate function but then throw it again to be handled by the higher level software

```
#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;
} } }
```

NEVER Throw from a Destructor

- Do not use throw from a destructor. Your code will go into an inconsistent (and unpleasant) state. Or just crash.
  - Because data member or base class destructors may not have the chance to run

```
class Base {
public:
  Base() { bptr_ = new int; *bptr_ = 0; }
  virtual ~Base() { delete bptr_; }
private:
  int* bptr_;
}

class Composite : public Base {
public:
  Composite() {
    sptr_ = new string("hi");
    inUse_ = true;
  }
  ~Composite() {
     if(inUse_ == true) {
        throw std::logic_error(
             "Should not be in use anymore");
     }
     // If we throw, do we ever do this code?
     delete sptr_;
  }

private:
  string* sptr_;
  bool inUse_;
}
```

Exception Safety

- Be careful WHEN you throw an exception that you don't leave the code in a bad state or leak resources
- Recall your maze search. What's wrong with the code to the right where I throw if I don't find an 'S'?

```
int maze_search(char** maze, int r, int c)
{
   int numS = 0;
   bool** explored = new bool*[r];
   // allocate the rest of the 2D explored
   // array

   for(int i=0; i < r; i++) {
     for(int j=0; j < c; j++) {
        if(maze[r][c] == 'S')
          numS++;
     }
   }
   if(numS != 1) {
     throw runtime_error("Expected 1 S");
     // Any issue here?
   }
   ...



   // deallocate 2D explored array
}
```

Using the Stack To Help

- Recall: Objects declared on the stack AUTOMATICALLY have their destructors called when the function ends (whether by a normal return or **BY A THROW**)
- Read more: C++-11 shared_ptr, unique_ptr, etc.

```
struct ExploredDeleter {
  ExploredDeleter(bool** explored, int nr) {
     exp_ = explored;
     rows_ = nr;
  }
  ~ExploredDeleter() {
     for(int i=0; i < rows_; i++) {
       delete [] exp_[i];
     }
     delete [] exp_;
  }
  bool** exp_;
  int rows_;
};
```

```
int maze_search(char** maze, int r, int c)
{
   int numS = 0;
   bool** explored = new bool*[r];
   for(int i=0; i < rows_; i++) {
       exp_[i] = new bool[c];
   }
   // How does this help solve the issue
   //  if we throw below
   ExploredDeleter expdel(explored, r);

   for(int i=0; i < r; i++) {
     for(int j=0; j < c; j++) {
        if(maze[r][c] == 'S')
          numS++;
     }
   }
   if(numS != 1) {
     throw runtime_error("Expected 1 S");
     // Do we still have an issue?
   }
   ...


   // Removed the deallocation code
   ~~// deallocate 2D explored array~~
}
```

Classic Example: Divide by Zero

```
#include <stdexcept>
using namespace std::runtime_error;
//Derive a runtime error to indicate a divide by zero error
class DivByZeroException : public runtime_error {
public:
   DivByZeroException();
};
DivByZeroException::DivByZeroException() :
    runtime_error(”divide by zero exception occurred”) //runtime_error is std::string
    {} //empty constructor body
```

DivByZero.h

Divide by Zero

```
#include <iostream>
#include “DivByZero.h”
using namespace std;
double fdivide(int n, int d)
{
    if (d == 0) throw DivByZeroException(); //create instance of our exception and “throw” it
    return (double)n/d;
}
int main(int argc, char* argv[])
{
    int x, y;
    double q;
    cout << ”Enter two integers (x and y) to divide:” << endl;
    while( cin >> x >> y )
    {
        //try block contains the code that
        //*could* have an error
        try {
            q = fdivide(x,y);
            cout << ”Result: “ << q << endl;
        }
        catch ( DivByZeroException &e) { //match a DivByZeroException by reference
            cout << “Uh-oh! Exception! ” << e.what() << endl; //call .what() to get the error string
            cout << “y can not be zero, try again. << endl; //provide error specific feedback
        }
         cout << ”Enter two integers (x and y) to divide:” << endl;
    }
}
```

Something went wrong, let my caller handle it

Main line code, assumes no error

Error handling

noexcept (C++ 11)

- In C++ 11 and later we can label a function “noexcept”
- Optimization opportunity for the compiler
  - supporting stack unwinding has some overhead
- If myFunc() calls a function that does throw, terminate() is called immediately
  - Stack is not unwound

```
int myFunc(double v, int x) noexcept
{
   //code that *will not* throw an exception
}
```

STD LIBRARY EXCEPTION USAGE

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');
  }

}
```

Why use exceptions?

- Some programs *must* continue to execute after an error occurs
  - Mission-critical or life/safety software
  - Business critical (downtime = $$$)
- Developers of libraries and classes can concentrate on main-line development
  - Let users deal with errors
- If no exception occurs, main-line code executes with minimal overhead
  - Remember exceptions are rare!

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]);
  ...
}
```