CSCI 103 – Unit 5g
Exceptions

Thinking About Errors

- Consider the slightly modified `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

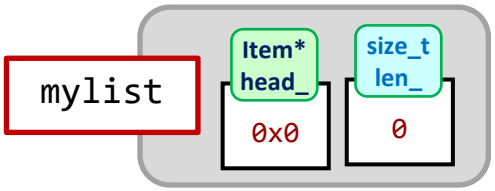
- Now consider the `ListInt` class
- Now consider error conditions
  - What member functions could cause an error?
  - How do I communicate the error to the user?

```
#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();
```



In this case (since it is void and the return value is not used), we could 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;
  }
}
```

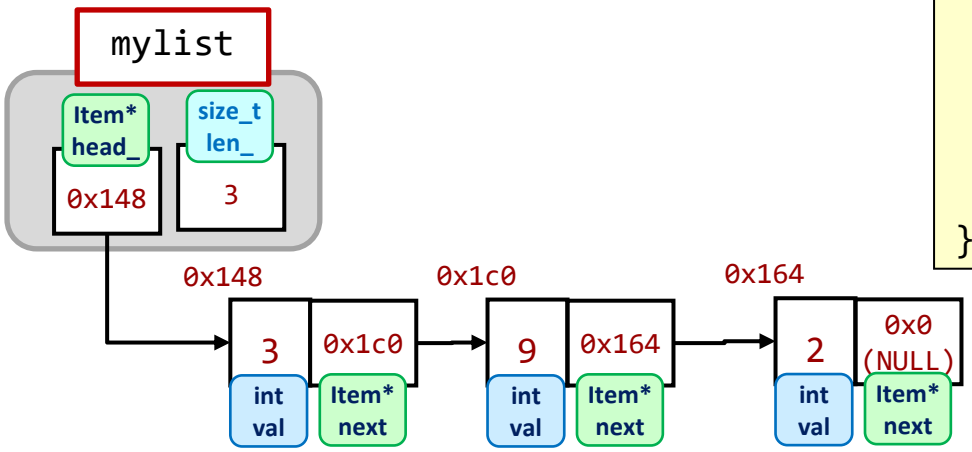
listint.cpp

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);
}
```

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 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).

It's often hard to tell looking at someone else's code.

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 1 or more lines of 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);
   cout << "Division result: " << x << endl;
}
catch( int badValue){
  cout << "Can't use denominator: " << badValue << endl;
  x = 0;
}

// Use x for more computation
```

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;
}
```

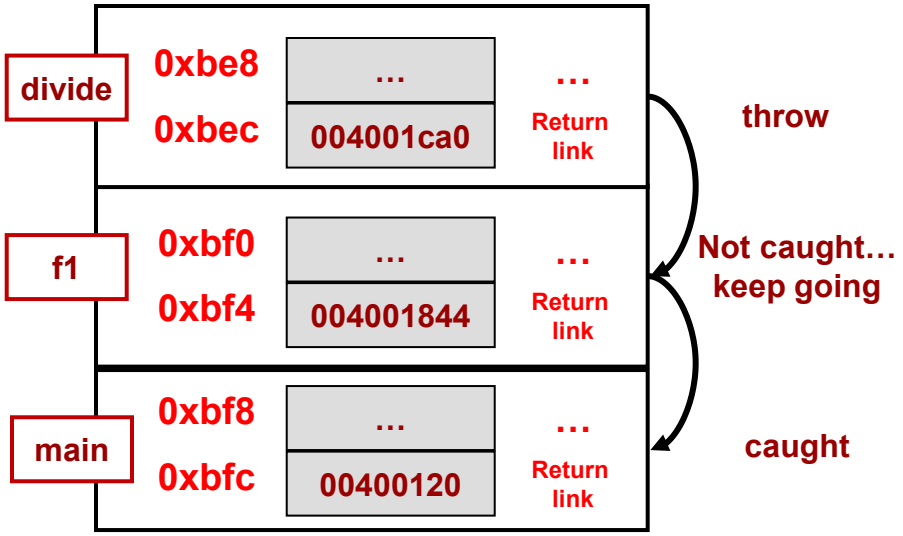
Catch Block Notes

- 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;
  }
}
```

Catch & The Stack

- You can use catch() blocks to actually 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

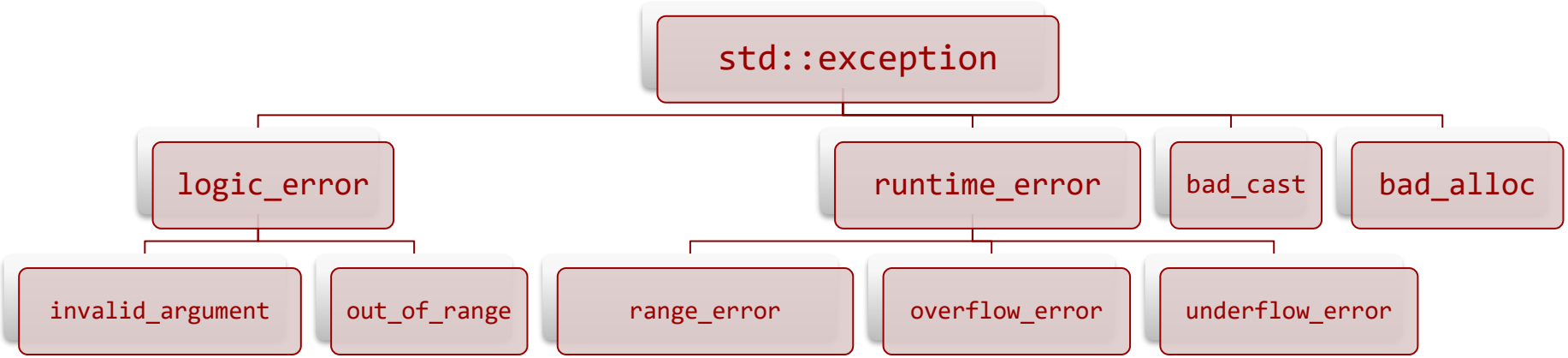
```
int divide(int num, int denom)
{
  if(denom == 0)
     throw denom;
  return(num/denom);
}
int f1(int x)
{
  return divide(x, x-2);
}

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



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

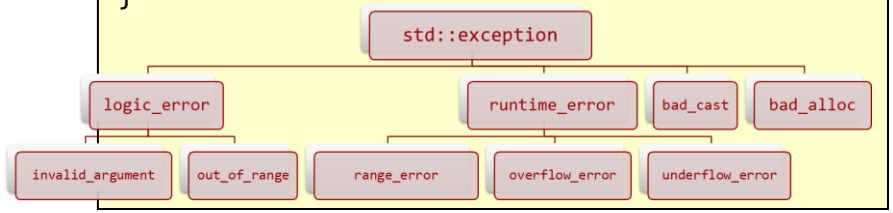
- Best practice: Order your catch statements from the **MOST** derived type first to the base type
- Why?
  - Recall: `DivByZero` is-a `range_error`
  - So a DivByZero can be passed to a range_error

```
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_;
}
```

A Look Ahead: 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 image or maze project. What's wrong with the code to the right where I throw if the argument is null or I can't open the file?

```
void save(char* filename)
{
   // performs dynamic allocation
   uint8_t*** image = allocateNewImage();




   if(filename == NULL) {
     throw invalid_argument("null ptr");
     // Do we still have an issue?

   }
   ofstream ofile(filename);
   if(ofile.fail() {
      throw runtime_error("can't open file");
      // Do we still have an issue?

   }
   //normal processing


   // deallocate the image array
}
```

A Look Ahead: 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 ImageDeleter {
  uint8_t*** img_;  int h_, w_;
  ImageDeleter(uint8_t*** img, int h, int w) :
      img_(img), h_(h), w_(w) {}
  ~ImageDeleter() {
     for(int i=0; i < h_; i++) {
       for(int j=0; j < w_; j++) {
         delete [] img_[i][j];
       }
       delete [] img_[i];
     }
     delete [] img_;
  }
};
```

```
void save(char* filename)
{
   // performs dynamic allocation
   uint8_t*** image = allocateNewImage();

   // How does this help solve the issue
   //  if we throw below
   ImageDeleter d(image, h_, w_);

   if(filename == NULL) {
     throw invalid_argument("null ptr");
     // Do we still have an issue?

   }
   ofstream ofile(filename);
   if(ofile.fail() {
      throw runtime_error("can't open file");
      // Do we still have an issue?

   }
   //normal processing


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

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');
  }

}
```

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]);
  ...
}
```