LECTURE 7

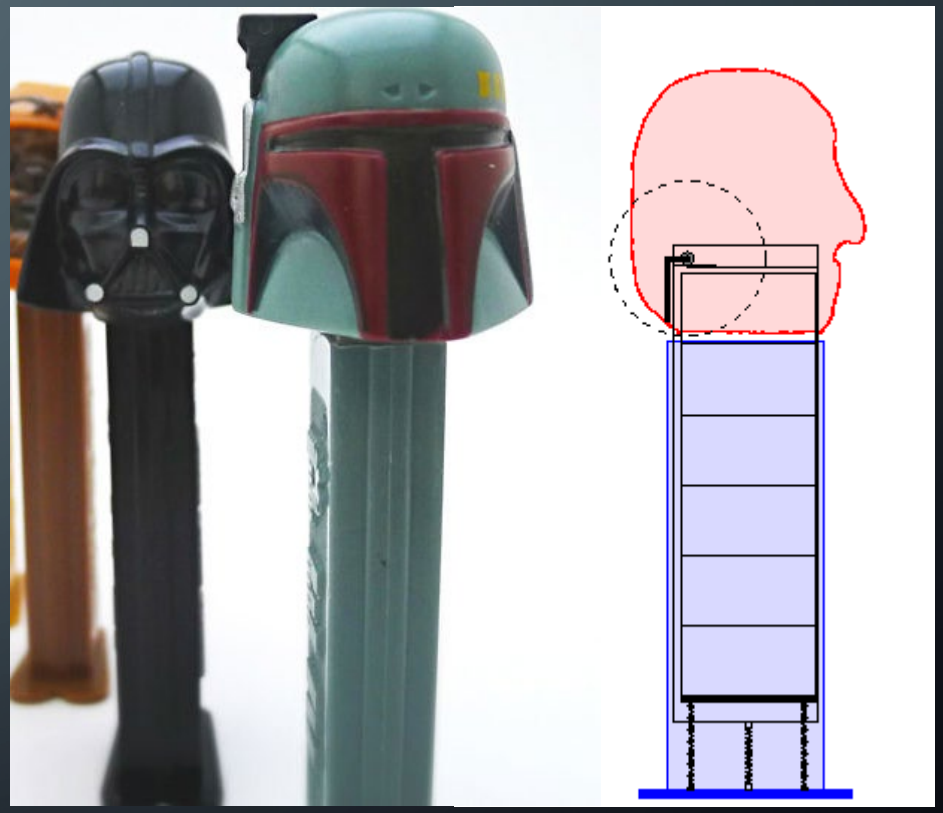
THE STACK ADT & EXCEPTIONS

TODAY'S CLASS:

- Stack ADT and how to handle exceptions

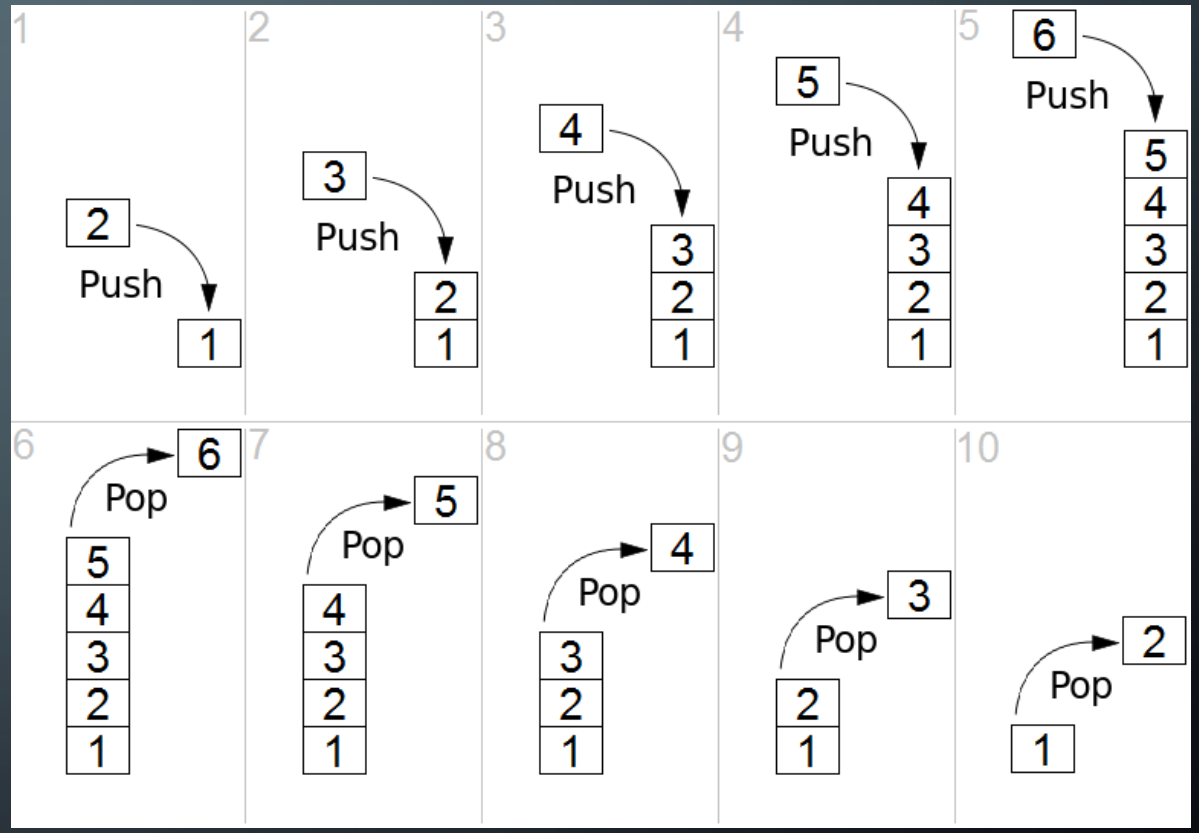
Stacks are … stacks of data

- What is a stack?
  - A list type where we can only add and remove from the first element
  - A data structure where accessing the most recent data element is easy



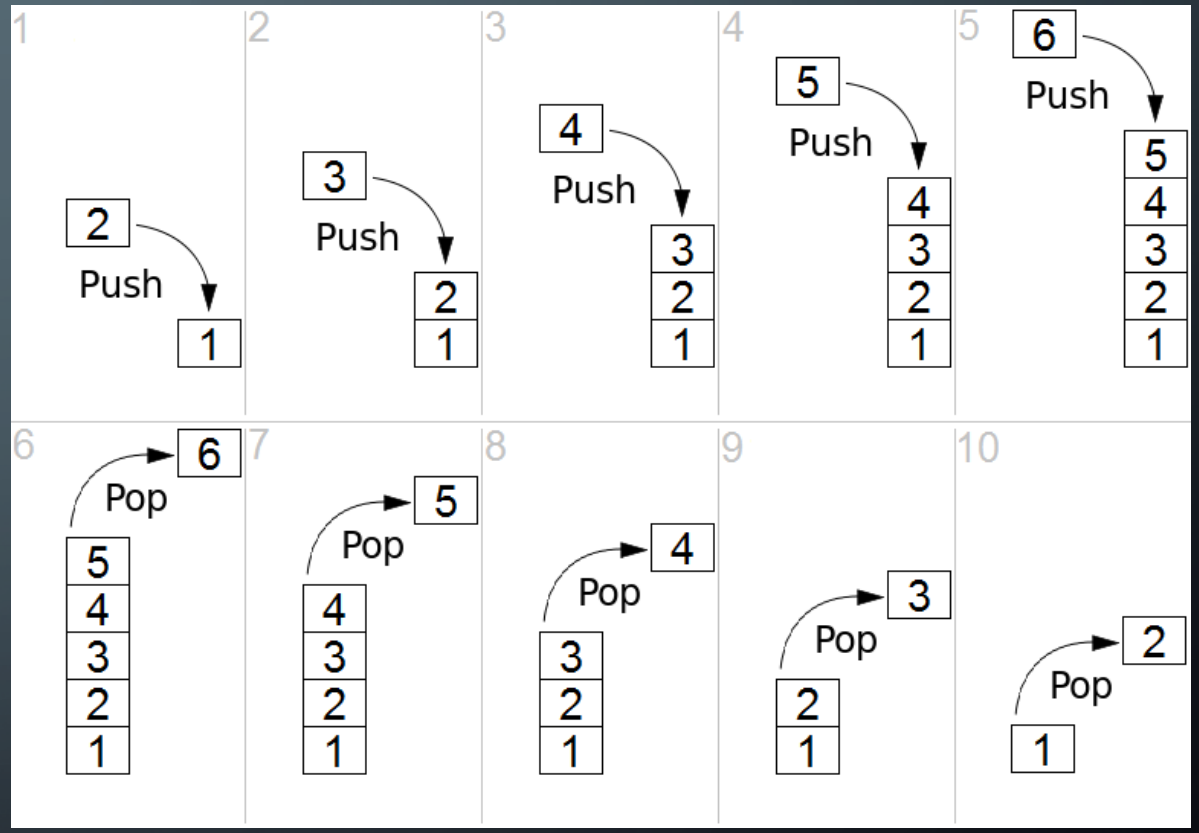
Stacking Stuff

- How do we interact with it?
  - PUSHing data onto the stack places new elements at the top
  - POPing data from the stack removes and returns elements at the top
  - At all times, the data is in last-in-first-out (LIFO) order



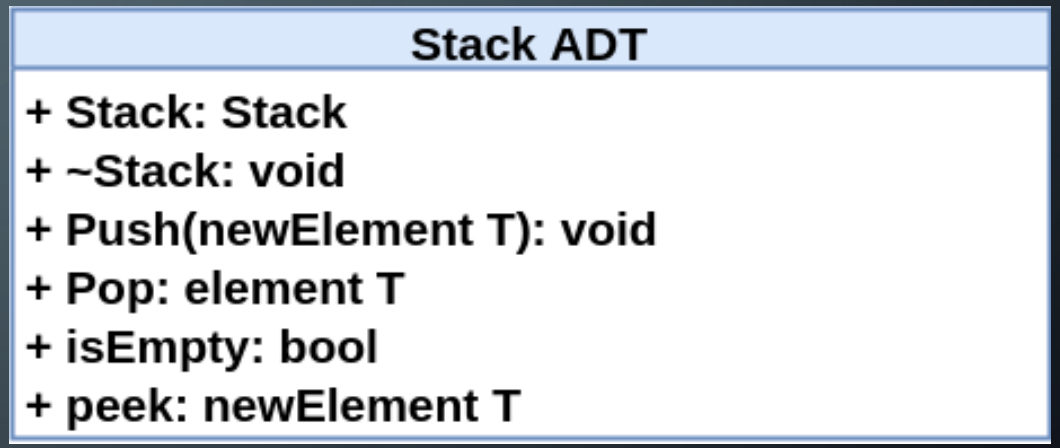
Core Functionality

- PUSH
  - Need to make sure there is room for new items
- POP
  - Need to check if the stack has items to remove
- Both cases:
  - Need a pointer to the top element, and a size



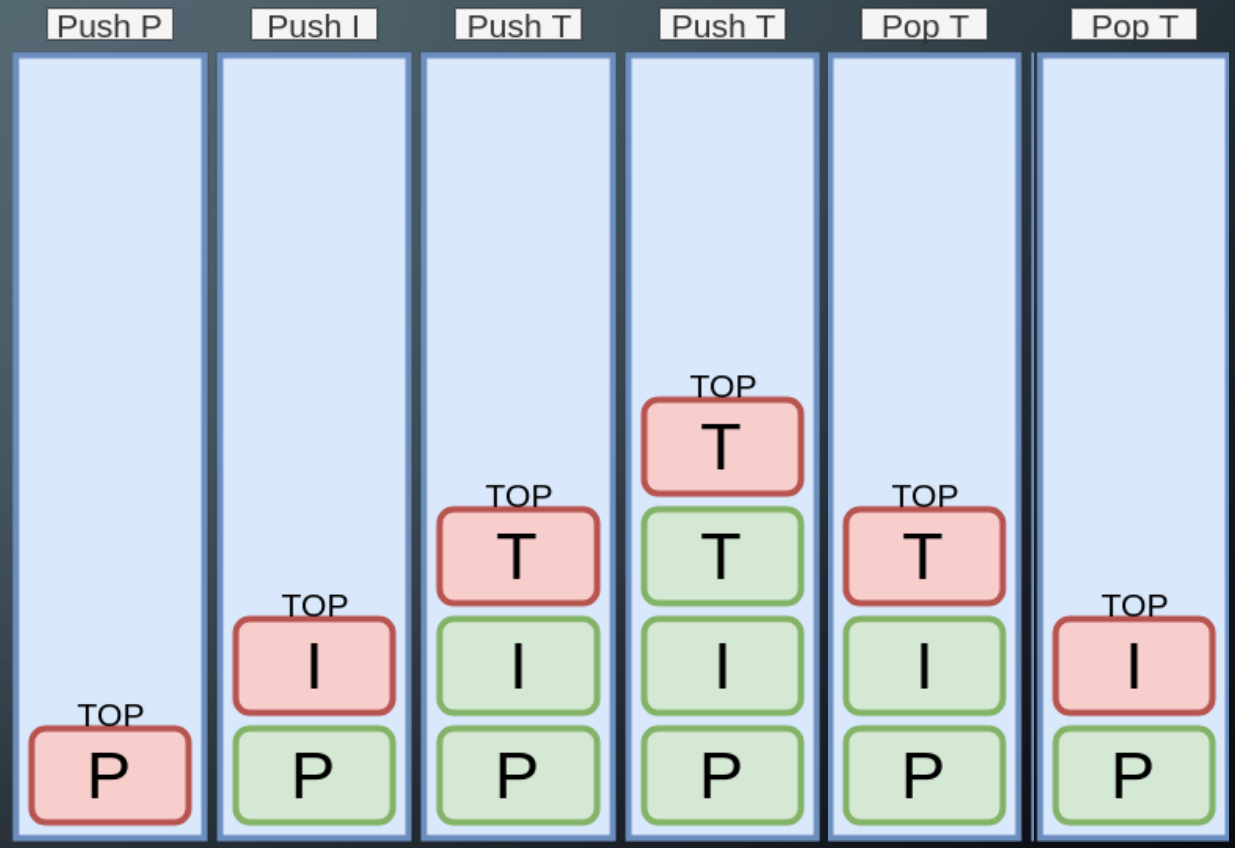
The Stack ADT

- This UML diagram shows all the core methods we need to implement
- Our ADT enforces that the user can only ever access the top element
- We have added PEEK, which allows the user to peek at the top element without removing it.



Use Case Example: Backspace

- When we are typing, our text processor allows us to delete characters in the reverse order from when we typed them
- This can be implemented via a stack:
  - As keys are pressed, push characters onto the stack
  - When backspace is pressed, pop characters off the stack



A Few Considerations…

- What happens when we POP an empty stack?
- What does PEEK do for an empty stack?
- If push and pop return bools rather than values, we can simply return true if successful and false otherwise
  - Return value by parameter?
- Peek might return a default value, NULL, or nullptr

Use Case Exercise: Palindrome Tester

- Given a string of characters as input, use a stack to test if that string is a palindrome
- Use only a stack ADT and its methods push, pop, and peek
- Assume push and pop return a bool, and peek returns a value or null

Use Case Exercise: Palindrome Tester

- What core methods do we need?
- What corner cases have we handled?
- What corner cases have we missed?

```
bool checkPalindrome(string s){
  find center of the string

  for each element from 0-center do{
    push the element onto the stack
  }
  if the string is odd then{
    skip ahead 1
  }
  while there are letters and the
  stack is not empty do{
    if the top != the current letter then{
      return false
    }
    go to the next letter
    pop the top item of the stack
  }
  if the stack is not empty then{
    return false
  }
  return true
}
```

Use Case Exercise: Postfix Calculator

- Given a string of characters representing a valid postfix notation, output the resultant value
- Postfix uses two values and an operand:
  - Example: 1 2 + yields 3
  - Example: 1 2 + 3 - yields 0
  - Example: 1 2 + 3 - 1 + yields 1

Use Case Exercise: Postfix Calculator

- What core methods do we need?
- What corner cases have we handled?
- What corner cases have we missed?

```
int simplePostfix(string s){
  for each character c in s do{
    if c is an operator then{
      operand2 := top of stack
      pop the stack
      operand1 := top of stack
      pop the stack
      result := compute(c, operand1,operand2)
      push result onto the stack
    }
    else{
      push c onto the stack
    }
  }
  return top of stack
}
```

Stack Implementation

- We can implement a stack with any of the internal data structures we have seen so far
- Given the use cases we have seen, which makes more sense:
  A. Use a fixed-size array to hold stack data
  B. Use a singly-linked list to hold stack data
  C. Use a doubly-linked list to hold stack data
  D. Use a dynamically-sized array to hold stack data

Switching Contexts: Error Handling

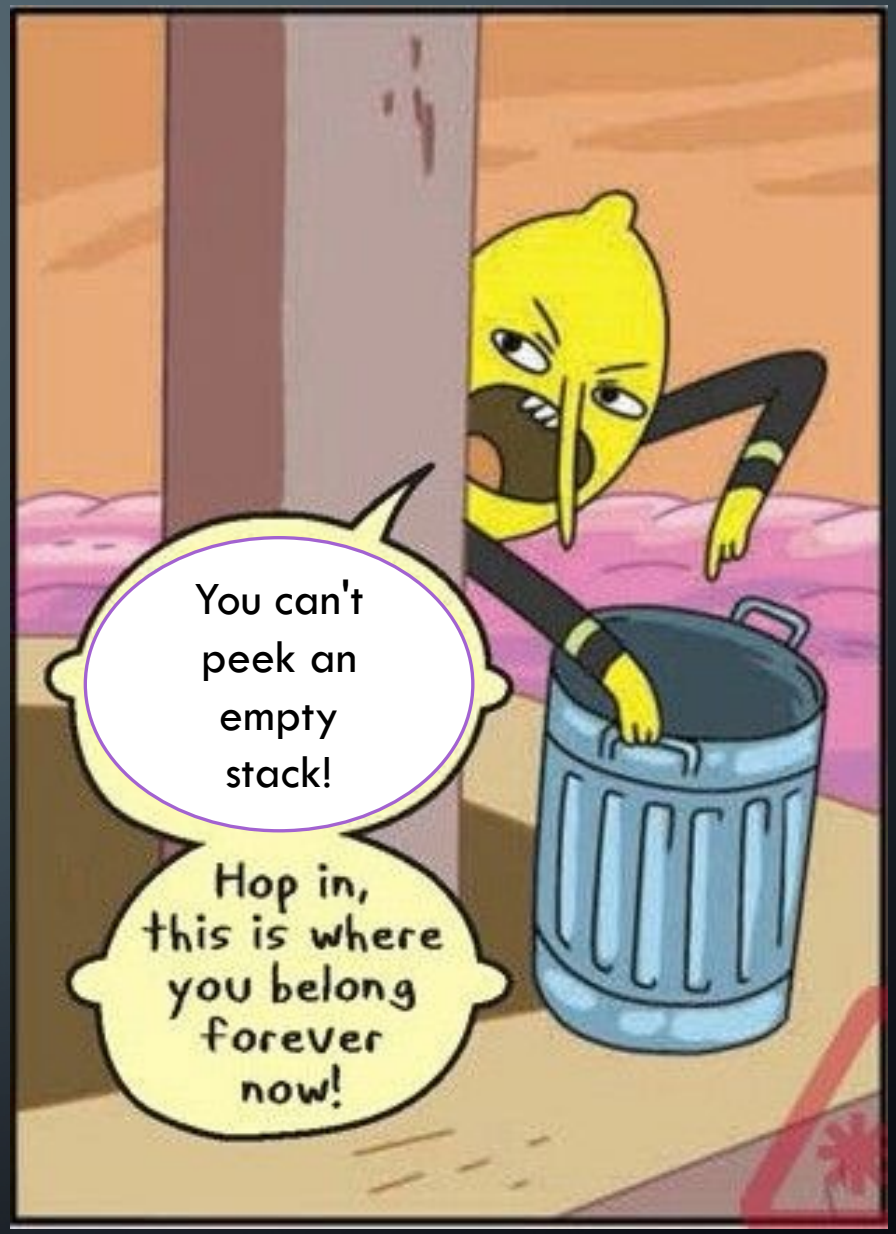
- Do we have to handle errors?
  - There are so many things that can go wrong.
  - Unless you are doing formal analysis, your code is never bulletproof
- In the past, a variety of approaches have been tried out:
  - Do nothing – bad if you plan to have other people use your code
  - Error Flags – Easy to signal different kinds of errors, hard to isolate
  - Returning Errors – Again, easy to implement, hard to isolate
  - Exceptions

Error Handling

- So far, we have tried to anticipate problematic situations and corner cases in our code
- Unfortunately, we cannot handle all the possible things that could go wrong
- Part of defining our data structures is determining a set of assumptions that we will make – if these assumptions fail, we need a way to signal for help

Exceptions

- Recall that we ran into a snag with our stack ADT: what do we do if the user peeks on an empty stack?
- We have provided the user tools to make sure that doesn't happen (isEmpty), but we can't force them to check
- This is a perfect situation to raise an exception to signal a problem

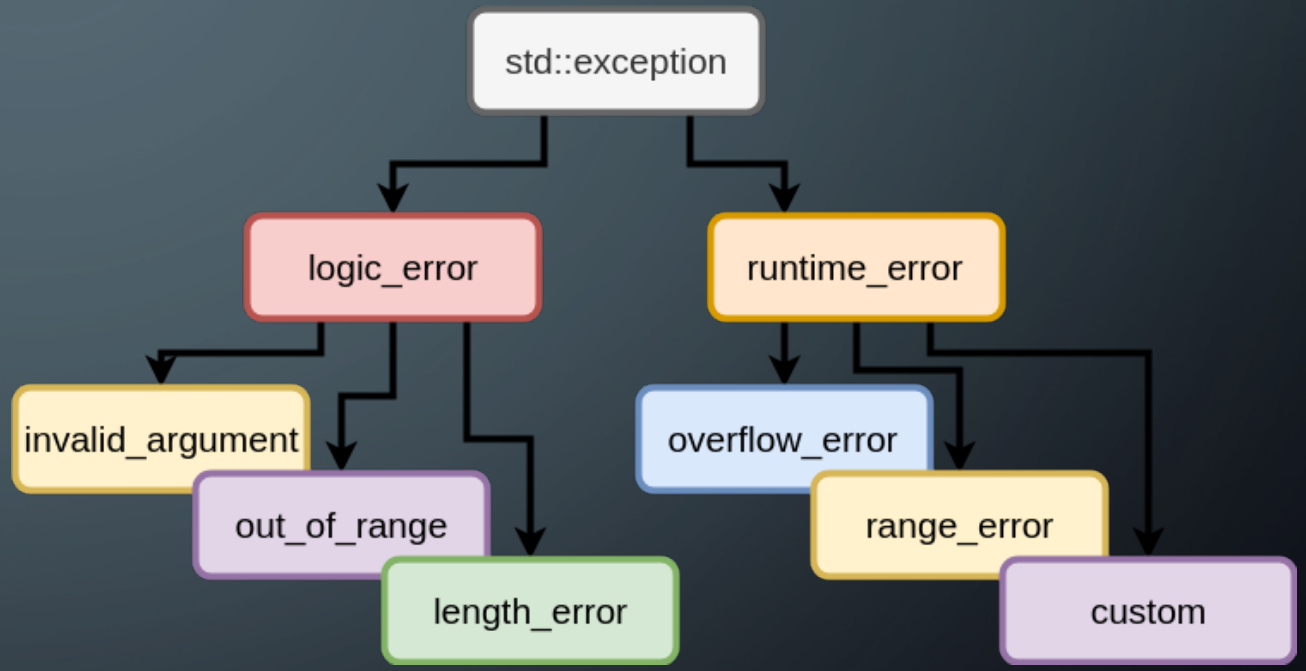


Exceptions

- C++ uses a special class of objects called exceptions, which interrupt the normal flow of a program's execution
- Common exception types implement the base classes defined in the standard library <exception> class header
- Example:
  - Overflow_error – Arithmetic overflow of arithmetic types (int, long)
  - Out_of_range – Accessed a data element outside of the valid range
  - Invalid_argument – The argument used for a function was invalid/improper

Exceptions

- As you might have guessed, we can write our own custom exceptions which derive the base exception class
- This is very useful for real software – the base classes simply indicate what went wrong with a string, but we may wish to just handle the problem internally. In this case, being able to capture state in a custom exception can help us resolve the problem



Syntax and Use Examples

- Classes and functions can throw exceptions, which call the constructor for the exception and interrupt the program
- We can throw specific exceptions or general ones. In general, we want to be as specific as possible...

```
updateItemN(T *dataArray, int length, int position){
  if(position >= length){
    throw(
      std::out_of_range(
        "position greater than array length!"
      )
    );
  }
  if(length < 0){
    throw(
      std::invalid_argument(
        "length must be non-negative"
      )
    );
  }
  if(position < 0){
    throw(
      "General error in updateItemN()"
    );
  }
}
```

Handling Exceptions

- If we can completely handle errors internally, we can catch exceptions and stop our program from halting completely

```
int getIOData(socket *s){
  // Potentially bad code goes in try clause
  try{
    int dataItem = s.read(1);
    return dataItem;
  }
  catch(const networkIOException& theException){
    // Catch clause only runs when the specific
    // exception is encountered.

    // Prints the string generated by the throw call
    std::cerr << theException.what();

    // You can run arbitrary code to handle the error
    // or provide more information
    std::cerr <<
      "Network read failed, check socket"
      << std::endl;
    return DEFAULTVALUE;
  }
}
```

FAIL VS. EPIC FAIL

- Opening files for reading and writing

```
std::ifstream ifs("afile.txt");
// what if afile.txt does not exist?
```

- parsing: example

```
int value = std::stoi("123!");
```

- memory allocation

```
std::vector data(1000);
// what if allocation fails inside vector?
```

For each line of code you write, think: How could this fail?

std::exception

- If you define your own exceptions, derive them from `std::exception`
  See example `derivedexept.cpp`
- They make the code much cleaner
- They reduce the number of sequential checks that go on

They also allow you to handle errors where you can (sometimes) do something about it.

Problems with exceptions

- They complicate resource management
- They can be somewhat hard to reason about.
- They make binaries larger

Overall exceptions are worth the effort. Be sure they are exceptions.

Never use exceptions for normal program flow.

All errors are not exceptions, in particular anything that is triggered by user input.

HOW TO HANDLE ERRORS

- handle it if you can
  - retry a network connection for example
- Notify the user
- Cleanup resources
- Die with dignity.

If you are out of memory anything in the exception handling code cannot allocate!

This includes things like `std::string`.

EXCEPTION SPECIFIERS AND SAFETY

- It is possible in C++ to say that a method may throw an exception.

```
void mymethod throw(A,B)
```

  - Does not work well, not recommended.
  - Document possible exceptions in the comment for the method.
- Related: the `noexcept` keyword
  - States a function does not throw an exception.
  - This is good practice if you can guarantee it will not.
  - Such code is called *exception-safe*.

EXAMPLE

- Parse a IP address in the form 192.168.0.1 into a 32 bit integer.
  - Version using exceptions: `parse_with_exceptions.cpp`

Assignment/Homework

- Reading: Carrano pp. 253-261, 265-286, 291-306
- HW3 due Today
- ICE 4 due Tomorrow
- P2 (Courseweb) due on Friday
- HW4 and ICE 5 will be released today