An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 21

Exceptions

And Reading Inputs

Based on material developed by Prof. Abhiram G. Ranade

When Things Go Wrong

- Programs should be written to be robust against all possible inputs in all possible environments
  - "Foolproof"

> *A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.*
>
> - Douglas Adams

- So far, our programs have focused on the "normal cases" rather than the exceptions
- Today: Some examples of exceptions and handling them

Wrong Kind of Inputs

- Recall that std::cin helps with reading in formatted inputs
- If the input is not of the right kind, it will "silently fail"

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main() {
  int x, sum = 0;
  cout << "Enter non-negative numbers to sum (end with -1): ";
  for(cin >> x; x >= 0; cin >> x)
    sum += x;
  cout << "Sum is " << sum << endl;
}
```

- Goes into an infinite loop if a non-number input is included!

Input Errors

- iostream objects set internal flags when errors occur
  - They can be checked via public functions

```
cin.fail(); // or just !cin (i.e., !bool(cin))
            // true if >> failed: wrong/no input, or bad()
cin.bad();  // true if system-level IO failure
cin.eof();  // true if End-Of-File reached (next >> will fail)
```

  - Can try to recover from wrong input by discarding inputs

```
cin.clear();       // clear the fail flag (so that we can retry)

cin.ignore(n,ch); // discard ≤ n characters, but stopping after ch
                  // default for n is 1, for ch is EOF.
```

Checking for Errors After Input

- A possible fix: on any error, pretend that input ended
  - Ideally, should notify the user about the error

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main() {
  int x, sum = 0;
  cout << "Enter non-negative numbers to sum (end with -1): ";
  for(cin >> x; cin && x >= 0; cin >> x)
    sum += x;
  cout << "Sum is " << sum << endl;
}
```

Checking for Errors After Input

- A possible fix: on any error, pretend that input ended
  - Ideally, should notify the user about the error

```
#include <iostream>
using std::cin; using std::cout; using std::endl;
int main() {
  int x, sum = 0;
  cout << "Enter non-negative numbers to sum (end with -1): ";
  for(cin >> x; cin && x >= 0; cin >> x)
    sum += x;
  if(!cin)
    cerr << "There was an error while reading inputs." << endl;
  cout << "Sum is " << sum << endl;
}
```

Avoiding Errors: Peek Before Reading

- When an input format errors occur, it is not guaranteed that no input has been consumed
  - E.g., when reading into an int, if a ± symbol followed by a non-digit is presented, cin may consume the symbol
  - To safely read numbers and ±, peek ahead to ensure there is a digit

```
char c = (cin >> std::ws).peek();
```

Skips till the next whitespace

Returns the next character, without removing it from the stream

  - Coming up: A wrapper around an input stream's >> for this

Avoiding Errors: Peek Before Reading

```
char c = (inp >> std::ws).peek(); // inp is an std::istream object
bool plusminus = (c=='+' || c=='-'), number = (c>='0' && c<='9');
if(!number) {
    inp >> c; // read the non-digit symbol that we peeked
    char next = inp.peek(); // peek again w/o skipping whitespace
    number = (plusminus && next >= '0' && next <= '9');
}
if(number) {
    int x; inp >> x; // read the number
    if(plusminus && c=='-') x = -x;
    if(inp) handleNumber(x); // if inp has failed, don't use x
} else
    if(inp) handleSymbol(c); // if inp has failed, don't use c
return inp;
```

Example: Peeking to Quit

```
bool quit(const string& prompt, char match) {
  cout << prompt;
  char c = (cin>>std::ws).peek();
  return (!cin || (c==match)); // if cin failed, still quit
}

int main() {
  const string prompt = "Input (q to quit): ";
  while(!quit(prompt,'q')) {
    // handle the input
    // quit() function did not consume any non-ws input
  }
}
```

A Reverse Polish Notation Calculator

- Example for today: An RPN Calculator
- RPN is a "postfix" notation
  - E.g., 1 2 + (evaluates to 3), 1 2 * 3 4 * + (evaluates to 14)
- Parsing is simple as there are no parentheses!
- Evaluation is easy to implement using a stack
  - Push numbers into the stack
  - On seeing operators, pop two elements from the stack, apply the operator, and push the result back

A Reverse Polish Notation Calculator

- Our plan: integer inputs, but retain the answer as a rational number

```
class rational {
  int N, D;
  void reduce(); // remove gcd from N, D
public:
  rational(int num=0, int den=1);
  rational& operator+= (const rational& other) {
    N = N * other.D + other.N * D; D *= other.D;
    reduce(); return *this;
  }
  rational& operator-= (const rational& other);
  rational& operator*= (const rational& other);
  rational& operator/= (const rational& other);
  friend ostream& operator<< (ostream& out, const rational& r);
};
```

A Reverse Polish Notation Calculator

```
rational::rational(int num, int den) : N(num), D(den) {
  if(D==0)
    throw std::domain_error("Zero Denominator");
  reduce();
}
rational& rational::operator/= (const rational& other) {
  if(other.N==0)
    throw std::domain_error("Division by zero");
  int sign = (other.N < 0) ? -1 : 1;
  N *= other.D * sign;
  D *= other.N * sign; // keep denominator positive
  reduce();
  return *this;
}
```

Causes the program to exit

Unless handled (coming up)

`throw`

- Syntax of the throw-expression: `throw` *expression*
- Can throw expression of any type
  - Typically, expressions thrown are of a class like `std::exception`
  - E.g., `std::domain_error`, `std::invalid_argument`, `std::runtime_error`, etc.
    - They are all "derived" from the "base class" `std::exception`
      - Note: An object of a derived class is also considered an object of the base class (with possibly extra members/features)

`throw`

- A throw statement results in "stack unwinding"
  - The function immediately terminates and its frame is removed from the stack (as if it returned), destructing all objects going out of scope
  - And on returning to the point where the function was called from, again the expression is thrown (recursively)
  - Until the program terminates
- Unless, the point of throw is inside a block that is "handled"

try - catch

- To be able to handle an exception that is thrown (possibly by a function that was called), the point where the throw occurs should be inside a "try block."Exception handled only if it matches catch type

```
try {
  // code that could potentially throw an exception
  // (possibly because a function call does it)
  int a, b;  cin >> a >> b;
  rational r(a,b); // our constructor can throw an exception if b==0
  cout << "It worked!" << endl;
} catch (std::exception& e) {
  // handle any thrown expression of a class derived from std::exception
  cerr << "Error: " << e.what() << endl;
}
```

If exception thrown above, any code here not executed

Message used while constructing e

try - catch

- Can have multiple catch blocks

```
try {
  // code that could potentially throw an exception
  // (possibly because a function call does it)
} catch (std::domain_error& e) {
  cerr << "Illegal value: " << e.what() << endl;
} catch (std::exception& e) {
  cerr << "Error: " << e.what() << endl;
} catch (std::string s) {
  cerr << "Someone threw a string: " << s << endl;
} catch (...) { // special syntax: catch everything thrown
  cerr << "Mysterious Error" << endl;
  throw; // re-throws exception being handled (to be handled by caller)
}
```

A Reverse Polish Notation Calculator

```
class RPNcalc {
  bool working = true;                        // till the calculator is "closed"
  std::stack<rational> stk;                    // the working stack
  void op(rational& a, rational b, char c); // sets a = a @ b, where c encodes @
public:
  void operator<< (const char& c);           // execute the operation for c
  void operator<< (const int& n);            // accept an integer input n
  operator bool() { return working; }
  friend ostream& operator<< (ostream& out, RPNcalc& calc); // print output (top)
};

istream& operator>> (istream& in, RPNcalc& calc); // a function to read inputs
```

A Reverse Polish Notation Calculator

```
class RPNcalc {
  bool working = true;                    // till the calculator is "closed"
```

```
int main() {
  const string prompt = ">>> Expression to evaluate (q to quit): ";
  while(!quit(prompt,'q')) {
    try {
      RPNcalc C;
      while(cin && C) {cin >> C;} // read till calculator or cin finished
      cout << "Output: " << C << endl;
    } catch (const std::exception& e) {
      cerr << "ERROR: " << e.what() << ". Skipping till '.'" << endl;
      cin.clear(); cin.ignore(std::numeric_limits<std::streamsize>::max(),'.');
    }
  }
}
```

A Reverse Polish Notation Calculator


```
void RPNcalc::operator<< (const char& c) {
  if(!working)
    throw std::invalid_argument("Input to closed stack");
  if(c=='.')
    working = false; // operator '.' to finish executing
  else {
    if(stk.size() < 2) throw std::invalid_argument("stkack underflow");
    rational b = stk.top(); stk.pop(); op(stk.top(),b,c);
  }
}

void  RPNcalc::operator<< (const int& n) {
  if(!working)
    throw std::invalid_argument("Input to closed stack");
  stk.push(rational(n));
}
```

Exercise

- Rewrite the RPNcalc class to use std::vector instead of std::stack internally. Make sure it works without any other changes outside the class.
- Change the '?' command to print the whole stack
- Add more commands (^ for power, comparisons,...)