AN INTRODUCTION TO PROGRAMMING
THROUGH C++

with

Manoj Prabhakaran

Lecture 14
Review

Based on material developed by Prof. Abhiram G. Ranade

Today

- A quick recap and a peek ahead
- A couple of problems from the Lab Quiz
  - Sample solutions
  - And some variations

Recap

- Several programming concepts, so far

Data	Control Flow/Dynamics	Program Organization
Variables, expressions	Sequential execution	Statements, scope
Basic data types	(And sequence points)	`main()` and other functions
Internal representation	Conditional execution	Preprocessing
Reference variables	Conditional loops	Header files, Multiple C++ files
Structs	Function calls	Functions inside structs
Arrays	Lifetime of a variable	Function templates
From the Standard Library: I/O streams, `string`	Static variables	Namespaces

Sequence Points

- When evaluating `EXP1 + EXP2` or, `cout << EXP1 << EXP2`, or `f(EXP1,EXP2)`, there is no guarantee about the order in which the expressions EXP1 and EXP2 will be evaluated

- But when evaluating `(EXP1,EXP2)` or `(EXP1 && EXP2)` or `(EXP1 || EXP2)`, EXP1 is guaranteed to be evaluated first
  - These operators are sequence points: Expressions appearing before the point will be evaluated before evaluating the ones after it

- Statements, conditions in `if`, `while` and ternary conditional expressions, expressions in the `for` loop control, and each initialisation in a declaration (e.g., `int x = EXP1, y = EXP2;`) all have sequence points after them

Recap

- Several programming concepts, so far

Data	Control Flow/Dynamics	Program Organization
Variables, expressions	Sequential execution	Statements, scope
Basic data types	(And sequence points)	`main()` and other functions
Internal representation	Conditional execution	Preprocessing
Reference variables	Conditional loops	Header files, Multiple C++ files
Structs	Function calls	Functions inside structs
Arrays	Lifetime of a variable	Function templates
From the Standard Library: I/O streams, `string`	Static variables	Namespaces
Pointers	Recursion	Classes (a glimpse)
More from the Standard Library	Exception handling	

- Additional important concepts coming up! ⬆️

Two Examples from the Quiz

- Balanced parentheses
- Detecting a sub-sequence

Balanced Parentheses

- A sequence of ‘(‘ and ‘)’ is balanced or valid, if it can be obtained from a valid mathematical expression by erasing all other symbols.
  - E.g., `(1+2)*((3+4)*(1+2))` yields a balanced sequence `()(()())`. But `(()` or `)(` are not balanced
- Write a program to check if a sequence is balanced
- Formal definition of being balanced: A string is balanced iff it is:
  - The empty string, or
  - A string of the form (X) where X is balanced (and shorter), or
  - A string of the form XY, where both of X, Y are balanced (and shorter)

Balanced Parentheses

- A sequence of '(' and ')' is balanced or valid, if it can be obtained from a valid mathematical expression by erasing all other symbols.
  - E.g., `(1+2)*((3+4)*(1+2))` yields a balanced sequence `()(()())`. But `(()` or `)(` are not balanced
- Write a program to check if a sequence is balanced
- Hint: Print # unbalanced openings, Δ = No. of (s – No. of)s
- Clearly, to be balanced Δ should be 0 at the end
- But that is not enough: E.g. `)(` is not balanced
- Condition: Left to right, Δ is never negative and is 0 at the end

Balanced Parentheses

```
#include <iostream>
int main(){
    int delta = 0;  // Δ = #( - #) seen so far
    bool valid = true;  // In a left-right scan, imbalance already
    int n; std::cin >> n;  // number of symbols to read
    for (int i=0; i < n; i++){
        char ch; std::cin >> ch;
        ch == '(' ? ++delta : --delta ; // update Δ
        if(delta < 0)
            valid = false;  // unmatched ) at this point
    }
    if(delta > 0)
        valid = false;  // one or more ( unmatched at the end
    std::cout << (valid?"VALID ":"INVALID ") << delta << std::endl;
}
```

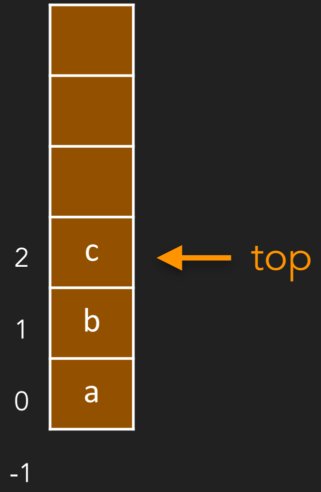
Balanced Parentheses: Multiple Kinds

- What if we use two kinds of brackets: [ ] and ( ) ?
- What doesn't work: one counter for each kind
  - Consider input starting with ([vs. with [(
    - If followed by]) the first one is valid and the second invalid
    - Just counting (and [can't differentiate between the two!
- What works: **using a stack**
  - Push open brackets into a stack
  - When a closing bracket arrives, pop and check for match
  - Also at the end, check if the stack is empty

Recap

A Simple Stack Implementation

```
const int stCap = 1000;  // capacity
struct charStack {
    char St[stCap];
    int top; // top = -1 for empty stack
    bool pop(char& i) { // if stack is empty, return false
        if (top == -1) return false;
        i = St[top--];
        return true;
    }
    bool push(char i) { // if stack is full, return false
        if (top == stCap-1) return false;
        St[++top] = i;
        return true;
    }
    void clear() { top = -1; }
};
```



Balanced Parentheses

```
int main(){
    char ch; charStack S; S.clear();
    bool valid = true;  // In a left-right scan, no imbalance so far
    int n; std::cin >> n;  // number of symbols to read
    for (int i=0; i < n; i++){
      std::cin >> ch;
      if(ch=='(' || ch=='[') S.push(ch);  // TODO: handle overflow
      else {
        char top;
        if(!S.pop(top) ||
           !( (top=='[' && ch==']') || (top=='(' && ch==')') ))
          valid = false;  // mismatch at this point
      }
    }
    if(S.pop(ch)) valid = false;  // stack not empty
    std::cout << (valid?"VALID":"INVALID") << std::endl;
}
```

Detect Subsequence

- $(a_1,\ldots,a_n)$ is a ***sub-sequence*** of $S_1,S_2,\ldots$ iff there are n indices $k_1 < k_2 < \ldots < k_n$ such that $(a_1,\ldots,a_n) = (S_{k_1},\ldots,S_{k_n})$

- Problem: Given an input sequence, check if it is a sub-sequence of an algorithmically generated (infinite) sequence that is <u>monotonically increasing</u>

  – Specifically, the Fibonacci sequence
  $F(0) = 0$, $F(1) = 1$, and for all $n > 1$, $F(n) = F(n-1) + F(n-2)$

  – $F(n)$ has a closed form expression, but will not need/use it

Detect Subsequence: 2 Approaches

Pseudocode

```
for each input
  seek a match in Fibonacci seq
  if Fib. seq overshoots input
    output "false"

if all inputs matched
  output "true"
```

```
read first input
for each element in Fibonacci seq
  if it equals current input
    if no more inputs
       output "true" and stop
    else
       read next input
  else if it overshoots input
    output "false" and stop
```

Detect Subsequence

Approach 1, Version 1

```
for each input
  seek a match in Fibonacci seq
  if Fib. seq overshoots input
    output "false"

if all inputs matched
  output "true"
```

```
#include <iostream>
int main(){
  int i, M; std::cin >> M;
  int f = 1, g = 0;             // initializing f = F(-1), g = F(0)
  for(i=0; i < M; i++) {      // for each input
    int x; std::cin >> x;       // read the input
    do {                      // seek a match in fib. seq
        std::swap(f,g); g += f; // advance f, g
    } while (f < x);            // until input found or overshot
    if (f != x)   // if fib. seq overshoots (no match)
        break;    // then leave i<M, to output false
  }
  std::cout << (i==M?"true":"false") << std::endl;
}
```

It is OK to have a longer program, perhaps with a bit of repeated code (e.g., using while instead of do-while)

Detect Subsequence

Approach 1, Version 1

```
for each input
  seek a match in Fibonacci seq
  if Fib. seq overshoots input
    output "false"

if all inputs matched
  output "true"
```

```
#include <iostream>
int main(){
  int i, M; std::cin >> M;
  int f ~~= 1, g = 0;             // initializing f = F(-1), g = F(0)~~
  for(i=0; i < M; i++) {        // for each input
    int x; std::cin >> x;         // read the input
    do {                        // seek a match in fib. seq
        ~~std::swap(f,g); g += f; // advance f, g~~
    } while (f < x);              // until input found or overshot
    if (f != x)   // if fib. seq overshoots (no match)
        break;    // then leave i<M, to output false
  }
  std::cout << (i==M?"true":"false") << std::endl;
}
```

Make it **modular**?
Can we keep the specifics of the Fibonacci sequence separate?

Detect Subsequence

Approach 1, Version 2

```
for each input
  seek a match in Fibonacci seq
  if Fib. seq overshoots input
    output "false"

if all inputs matched
  output "true"
```

```
#include <iostream>
int next();  // Each call to it will return the next element in the sequence.
int main(){
  int i, M; std::cin >> M;
  for(i=0; i < M; i++) {       // for each input
    int f, x; std::cin >> x;   // read the input
    do f=next(); while (f<x); // seek a match in the sequence
    if (f != x) break; // if fib. seq overshoots, leave i<M, to output false
  }
  std::cout << (i==M?"true":"false") << std::endl;
}
```

A ***modular*** solution:
Can readily replace Fibonacci
with other sequences.

```
int next() {                        // implements fibonacci sequence
  static int f1 = 1, f2 = 0;        // initialize f1,f2 to "F(-1)",F(0)
  std::swap(f1,f2); f2 += f1;       // (f1, f2) ← (f2, sum)
  return f1;
}
```

Detect Subsequence

```
for each input
  seek a match in Fibonacci seq
  if Fib. seq overshoots input
    output "false"

if all inputs matched
  output "true"
```

Approach 1, Version 3

```cpp
#include <iostream>
#include "fib.h"    // struct Fibonacci defined here
int main(){
  Fibonacci fib; fib.init(); // initialise the struct before accessing
  int i, M; std::cin >> M;
  for(i=0; i < M; i++) {         // for each input
    int x, f;
    std::cin >> x;               // read next input
    do f=fib.next(); while (f<x); // seek a match in fib. seq
    if(f != x) break; // if fib. seq overshoots, leave i<M, to output false
  }
  std::cout << (i==M?"true":"false") << std::endl;
}
```

A ***more modular*** solution.
OK to use the fibonacci sequence
in many places in a program.

fib.h

```cpp
struct Fibonacci {
  int f1, f2;
  void init() { f1=0; f2=1; }
  int next() { int f = f1; std::swap(f1,f2); f2 += f1; return f;}
};
```

Detect Subsequence: 2 Approaches

Pseudocode

```
for each input
  seek a match in Fibonacci seq
  if Fib. seq overshoots input
    output "false"

if all inputs matched
  output "true"
```

```
read first input
for each element in Fibonacci seq
  if it equals current input
    if no more inputs
       output "true" and stop
    else
       read next input
  else if it overshoots input
    output "false" and stop
```

Detect Subsequence

```
read first input
for each element in fib. seq
  if it equals current input
    if no more inputs
       output "true" and stop
    else
       read next input
  else if it overshoots input
    output "false" and stop
```

Approach 2

Version with structs

```
int main(){
  intInputs in; in.init();
  Fibonacci f; f.init();
  int x, y;
  bool read_ok = in.read(x); // read input; returns false if inputs over
  for( y = f.next(); read_ok; y = f.next() ) { // for each y in fib. seq
    if(x==y)                     // if y matched with input
        read_ok = in.read(x);// read next input; exit loop if input over
    else if (x<y)                // if y overshoots input
        break;                   // then exit loop, leaving read_ok true
  }
  std::cout << (!read_ok?"true":"false")
            << std::endl;
}
```

```
struct intInputs {
  int toRead; // how many to read
  void init() { std::cin >> toRead; }
  bool read(int& x) {
    if (toRead <= 0) return false;
    std::cin >> x; toRead --;
    return true;
  }
};
```