Lecture 1: Introduction

Instructor: George Wolberg
Department of Computer Science
City College of New York
Outline of this lecture

- Course Objectives and Schedule
  - WHAT (Topics)
  - WHY (Importance)
  - WHERE (Goals)
  - HOW (Information and Schedule)

- The Phase of Software Development
  - Basic design strategy
  - Pre-conditions and post-conditions
  - Running time analysis
Topics (WHAT)

- Data Structures
  - specification, design, implementation and use of
    - basic data types (arrays, lists, queues, stacks, trees…)
- OOP and C++
  - C++ classes, container classes, Big Three
- Standard Template Library (STL)
  - templates, iterators
  - ADTs in our DS course cut-down version of STL
- Recursion, Searching and Sorting Algorithms
  - important techniques in many applications
Importance (WHY)

- Data Structures (how to organize data) and Algorithms (how to manipulate data) are the cores of today’s computer programming.

- The behavior of Abstract Data Types (ADTs) in our Data Structures course is a cut-down version of Standard Template Library (STL) in C++.

- Lay a foundation for other aspects of “real programming” – OOP, Recursion, Sorting, Searching.

George Wolberg, 2016
Goals (WHERE)

- Implement these data structures as classes in C++
- Determine which structures are appropriate in various situations
- Confidently learn new structures beyond what are presented in this class
- also learn part of the OOP and software development methodology

understand the data types inside out
Course Information (HOW)

- Objectives
  - Data Structures, with C++ and Software Engineering

- Textbook and References
  - Reference: *C++ How to Program* by Dietel & Dietel, 8th Ed., Prentice Hall 2011

- Prerequisites
  - CSc103 C++ (Intro to Computing), CSc 104 (Discrete Math Structure I)

- Assignments and Grading
  - 6-7 programming assignments roughly every 2 weeks (50%)
  - 2 in-class writing exams (50%)

- Computing Facilities
  - PCs: Microsoft Visual C++ ; Unix / Linux : g++
  - also publicly accessible at Computer Science labs
Tentative Schedule (HOW)

(14 weeks = 28 classes = 23 lectures + 3 reviews + 2 exams, 6-7 assignments)

- Week 1. The Phase of Software Development (Ch 1)
- Week 2. ADT and C++ Classes (Ch 2)
- Week 3. Container Classes (Ch 3)
- Week 4. Pointers and Dynamic Arrays (Ch 4)
- Reviews and the 1st exam (Ch. 1-4)
- Week 5. Linked Lists (Ch. 5)
- Week 6. Template and STL (Ch 6)
- Week 7. Stacks (Ch 7) and Queues (Ch 8)
- Week 8. Recursion (Ch 9)
- Reviews and the 2nd exam (Ch. 5-9)
- Week 9/10. Trees (Ch 10, Ch 11)
- Week 11. Searching and Hashing (Ch 12)
- Week 12. Sorting (Ch 13)
- Week 13. Graphs (Ch 15)
- Reviews and the 3rd exam (mainly Ch. 10-13)
Course Web Page

You can find all the information at

http://www-cs.ccny.cuny.edu/~wolberg/cs212/index.html

or via my web page:

http://www-cs.ccny.cuny.edu/~wolberg

-Come back frequently for the updating of lecture schedule, programming assignments and exam schedule

- Reading assignments & programming assignments
Piazza

- All class-related discussion will be done on Piazza.
- Ask questions on Piazza (rather than via emails)
- Benefit from collective knowledge of classmates
- Ask questions when struggling to understand a concept.
- You can even do so anonymously.

Signup: piazza.com/ccny.cuny/spring2016/csc212bc

Class link: piazza.com/ccny.cuny/spring2016/csc212bc/home
Outline

- Course Objectives and Schedule
  - Information
  - Topics
  - Schedule
- The Phase of Software Development
  - Basic design strategy
  - Pre-conditions and post-conditions
  - Running time analysis
Phase of Software Development

- Basic Design Strategy – four steps (Reading: Ch. 1)
  - Specify the problem - Input/Output (I/O)
  - Design data structures and algorithms (pseudo code)
  - Implement in a language such as C++
  - Test and debug the program (Reading Ch 1.3)

- Design Technique
  - Decomposing the problem

- Two Important Issues (along with design and Implement)
  - Pre-Conditions and Post-Conditions
  - Running Time Analysis
An important topic: **preconditions** and **postconditions**.

They are a method of specifying what a function accomplishes.
Preconditions and Postconditions

Frequently a programmer must communicate precisely what a function accomplishes, without any indication of how the function does its work.

Can you think of a situation where this would occur?
Example

- You are the head of a programming team and you want one of your programmers to write a function for part of a project.

  HERE ARE THE REQUIREMENTS FOR A FUNCTION THAT I WANT YOU TO WRITE.

  I DON'T CARE WHAT METHOD THE FUNCTION USES, AS LONG AS THESE REQUIREMENTS ARE MET.
What are Preconditions and Postconditions?

- One way to specify such requirements is with a pair of statements about the function.
- The **precondition** statement indicates what must be true before the function is called.
- The **postcondition** statement indicates what will be true when the function finishes its work.
Example

```c
void write_sqrt( double x)

    // Precondition:  x  >=  0.
    // Postcondition: The square root of x has
    // been written to the standard output.
```
void write_sqrt( double x)

// Precondition: \( x \geq 0 \).
// Postcondition: The square root of x has
// been written to the standard output.

- The precondition and postcondition appear as comments in your program.
- They are usually placed after the function’s parameter list.
Example

void write_sqrt( double x)

// Precondition: x >= 0.
// Postcondition: The square root of x has
// been written to the standard output.

- In this example, the precondition requires that
  x >= 0
  be true whenever the function is called.
Example

Which of these function calls meet the precondition?

write_sqrt(-10);
write_sqrt(0);
write_sqrt(5.6);
Example

Which of these function calls meet the precondition?

write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );

The second and third calls are fine, since the argument is greater than or equal to zero.
Example

Which of these function calls meet the precondition?

write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );

But the first call violates the precondition, since the argument is less than zero.
Example

```c
void write_sqrt( double x)
  // Precondition:  x >= 0.
  // Postcondition: The square root of x has
  // been written to the standard output.
```
Another Example

```cpp
bool is_vowel( char letter )
//   Precondition:  letter is an uppercase or lowercase letter (in the range 'A' ... 'Z' or 'a' ... 'z') .
//   Postcondition:  The value returned by the function is true if letter is a vowel;
//                  otherwise the value returned by the function is false.
```
Another Example

What values will be returned by these function calls?

```c
is_vowel( 'A' );
is_vowel(' Z' );
is_vowel( '?' );
```
Another Example

What values will be returned by these function calls?

```cpp
is_vowel('A');
is_vowel('Z');
is_vowel('?');
```

Nobody knows, because the precondition has been violated.
Who is responsible for the crash?

Violating the precondition might even crash the computer.

```cpp
write_sqrt(-10.0);
is_vowel('?');
```
Always make sure the precondition is valid . . .

- The programmer who calls the function is responsible for **ensuring that the precondition is valid** when the function is called.
so the postcondition becomes true at the function’s end.

- The programmer who writes the function counts on the precondition being valid, and ensures that the postcondition becomes true at the function’s end.

- The precondition is enforced in C++ through use of assert() function.
A Quiz

Suppose that you call a function, and you neglect to make sure that the precondition is valid. Who is responsible if this inadvertently causes a 1-day long blackout in NYC or other disaster?

- You
- The programmer who wrote that Power Supply function
- Mayor Bloomberg

Out of Penn Station
A Quiz

Suppose that you call a function, and you neglect to make sure that the precondition is valid. Who is responsible if this inadvertently causes a 1-day long blackout in NYC or other disaster?

🌟 You

The programmer who calls a function is responsible for ensuring that the precondition is valid.
On the other hand, careful programmers also follow these rules:

- When you write a function, you should make every effort to detect when a precondition has been violated.
- If you detect that a precondition has been violated, then print an error message and halt the program.
On the other hand, careful programmers also follow these rules:

- When you write a function, you should make every effort to detect when a precondition has been violated.
- If you detect that a precondition has been violated, then print an error message and halt the program...
- ...rather than causing a chaos.

The famous skyline was dark on Aug 14th, 2003.
Example

void write_sqrt( double x)
// Precondition: x >= 0.
// Postcondition: The square root of x has
// been written to the standard output.
{
    assert(x >= 0);

    // ...
Advantages of Using Pre- and Post-conditions

- Concisely describes the behavior of a function... 
- ... without cluttering up your thinking with details of how the function works.
- At a later point, you may reimplement the function in a new way ...
- ... but programs (which only depend on the precondition/postcondition) will still work with no changes.
## Summary of pre- and post-conditions

<table>
<thead>
<tr>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The programmer who calls a function ensures that the precondition is valid.</td>
<td>- The programmer who writes a function ensures that the postcondition is true when the function finishes executing.</td>
</tr>
<tr>
<td>- The programmer who writes a function can bank on the precondition being true when the function begins execution. Careful programmers enforce this anyway!</td>
<td></td>
</tr>
</tbody>
</table>

George Wolberg, 2016
Phase of Software Development

- Basic Design Strategy – four steps (Reading: Ch.1)
  - Specify Input/Output (I/O)
  - Design data structures and algorithms
  - Implement in a language such as C++
  - Test and debug the program (Reading Ch 1.3)

- Design Technique
  - Decomposing the problem

- Two Important Issues (along with design and Implement)
  - Pre-Conditions and Post-Conditions
  - Running Time Analysis
Running Time Analysis – Big O

- Time Analysis
  - Fast enough?
  - How much longer if input gets larger?
  - Which among several is the fastest?
Example: Stair Counting Problem

- How many steps?

1789 (Birnbaum)
1671 (Joseph Harriss)
1652 (others)

1665 (Official Eiffel Tower Website)

- Find it out yourself!
Example : Stair Counting Problem

- Find it out yourself!
  - Method 1: Walk down and keep a tally
    - Each time a step down, make a mark
  - Method 2: Walk down, but let Judy keep the tally
    - Down+1, hat, back, Judy make a mark
  - Method 3: Jervis to the rescue
    - One mark per digit

There are 2689 steps in this stairway (really!)

Eiffel Tower
Example: Stair Counting Problem

- How to measure the time?
  - Just measure the actual time
    - vary from person to person
    - depending on many factors
  - Count certain operations
    - each time walk up/down, 1 operation
    - each time mark a symbol, 1 operation

George Wolberg, 2016
Example: Stair Counting Problem

- Find it out yourself!
  - Method 1: Walk down and keep a tally
    \[2689 \text{ (down)} + 2689 \text{ (up)} + 2689 \text{ (marks)} = 8067\]
  - Method 2: Walk down, let Judy keep tally
    \[
    \begin{align*}
    \text{Down: } & 3,616,705 = 1+2+\ldots+2689 \\
    \text{Up: } & 3,616,705 = 1+2+\ldots+2689 \\
    \text{Marks: } & 2,689 = 1+1+\ldots+1
    \end{align*}
    \]
    \[\Rightarrow 7,236,099!\]
  - Method 3: Jervis to the rescue
    only 4 marks!
Example: Stair Counting Problem

- Size of the Input: n
  - Method 1: Walk down and keep a tally
    \[3n\]
  - Method 2: Walk down, let Judy keep tally
    \[n + 2(1 + 2 + \ldots + n) = n + (n+1)n = n^2 + 2n\]
    - Trick: Compute twice the amount and then divided by two
  - Method 3: Jervis to the rescue
    - The number of digits in \(n\) = \([\log_{10} n] + 1\)
Example: Stair Counting Problem

- **Big-O Notation** – the order of the algorithm
  - Use the largest term in a formula
  - Ignore the multiplicative constant

- **Method 1**: Linear time
  \[3n \Rightarrow O(n)\]

- **Method 2**: Quadratic time
  \[n^2 + 2n \Rightarrow O(n^2)\]

- **Method 3**: Logarithmic time
  \[[\log_{10} n] + 1 \Rightarrow O(\log n)\]
## A Quiz

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Big-O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n^2+5n)</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>(100n+n^2)</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>((n+7)(n-2))</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>(n+100)</td>
<td>(O(n))</td>
</tr>
<tr>
<td>number of digits in 2n</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>
## Big-O Notation

- The order of an algorithm generally is more important than the speed of the processor

<table>
<thead>
<tr>
<th>Input size: n</th>
<th>O(log n)</th>
<th>O (n)</th>
<th>O (n²)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of stairs: n</td>
<td>[log₁₀n]+1</td>
<td>3n</td>
<td>n²+2n</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>30</td>
<td>120</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>300</td>
<td>10,200</td>
</tr>
<tr>
<td>1000</td>
<td>4</td>
<td>3000</td>
<td>1,002,000</td>
</tr>
</tbody>
</table>

George Wolberg, 2016
Time Analysis of C++ Functions

- Example- Quiz (5 minutes)
  - Printout all item in an integer array of size N

    ```cpp
    for (i=0; i< N; i++)
    {
      val = a[i];
      cout << val;
    }
    ```

  - 2 C++ operations or more?

- Frequent linear pattern
  - A loop that does a fixed amount of operations N times requires O(N) time
Another example

Printout char one by one in a string of length N

```
for (i=0; i< strlen(str); i++)
{
    c = str[i];
    cout << c;
}
```

$O(N^2)!$

What is a single operation?

- If the function calls do complex things, then count the operation carried out there
- Put a function call outside the loop if you can!
Another example

Printout char one by one in a string of length N

```
N = strlen(str);
for (i=0; i<N; i++) {
    c = str[i];
    cout << c;
}
```

What is a single operation?

- If the function calls do complex things, then count the operation carried out there
- Put a function call outside the loop if you can!
Time Analysis of C++ Functions

- Worst case, average case and best case
  - search a number x in an integer array a of size N

```cpp
for (i=0; (i< N) && (a[i] != x); i++ );
if (i < N) cout << “Number ” << x << “is at location ” << i << endl;
else cout << “Not Found!” << endl;
```

- Can you provide an exact number of operations?
  - Best case: 1+2+1
  - Worst case: 1+3N+1
  - Average case: 1+3N/2+1
Testing and Debugging

- Test: run a program and observe its behavior
  - input -> expected output?
  - how long ?
  - software engineering issues

- Choosing Test Data : two techniques
  - boundary values
  - fully exercising code (tool: profiler)

- Debugging… find the bug after an error is found
  - rule: never change if you are not sure what’s the error
  - tool: debugger
Summary

- Often ask yourselves FOUR questions
  - WHAT, WHY, WHERE & HOW
  - Topics – DSs, C++, STL, basic algorithms
  - Data Structure experts
  - Schedule – 23 lectures, 7 assignments, 2 exams
  - some credits (10) for attending the class
  - Information – website

- Remember and apply two things (Ch 1)
  - Basic design strategy
  - Pre-conditions and post-conditions
  - Running time analysis
  - Testing and Debugging (reading 1.3)
Reminder …

Lecture 2: ADT and C++ Classes

Reading Assignment before the next lecture:

Chapter 1

Chapter 2, Sections 2.1-2.3

Office Hours:

Wed 2:00 pm - 3:00 pm
(Location: NAC 8/202N)

(check website for details)
THE END