SANTA CLARA UNIVERSITY
THE JESUIT UNIVERSITY IN SILICON VALLEY

Formal Specification and Advanced Data Structures

The Phases of Software Development

Learning Objectives

- Write precondition/postcondition contracts for small functions, and use the C++ assert facility to test preconditions
- Recognize quadratic, linear, and logarithmic running time behavior in simple algorithms, and write big-O expressions to describe this behavior
- Create and recognize test data that is appropriate for simple problems, including testing boundary conditions and fully exercising code

INTRODUCTION

The phase of Software Development

- Specification of the task
- Design of a solution
- Implementation (coding) of the solution
- Analysis of the solution
- Testing and debugging
- Maintenance and evolution of the system
- Obsolescence

SPECIFICATION, DESIGN, IMPLEMENTATION

Specification, design, implementation

- **Specification**: a precise description of the problem
- **Design:** design phase consists of formulating the steps to solve the problem
- **Implementation**: actual C++ code that carries out the design

Specification

- Example: Display a table for converting Celsius temperatures to Fahrenheit

- For a small problem, a sample of the desired output is a sufficient specification

```
CONVERSIONS FROM -50.0 to 50.0
   Celsius       Fahrenheit
    -50.0C
    -40.0C
    -30.0C
    -20.0C
    -10.0C
      0.0C
     10.0C
     20.0C
     30.0C
     40.0C
     50.0C
```

The actual Fahrenheit temperatures will be computed and displayed on this side of the table.

Design

- **Algorithm**:
  - A set of instructions for solving a problem
  - Example: An algorithm for the temperature problem will print the conversion table
- **Pseudocode**:
  - Details of a particular programming language can be distracting
  - **A mixture of English and a programming language**
  - When the C++ code for a step is obvious, then the pseudocode may use C++
  - When a step is clearer in English, then we will use English

Design Concept: Decomposing the Problem

- **Break down the problem into a few subtasks**
  - decompose each subtask into smaller subtasks
  - replace the smaller subtasks with even smaller subtasks, and so forth
- **Each subtask can be implemented as a separate piece of your program (C++ function)**
- Example: The temperature problem
  1. Converting a temperature from Celsius degrees to Fahrenheit
  2. Printing a line of the conversion table in the specified format

Pseudocode for the temperature problem

1. Do preliminary work to open and set up the output device properly
2. Display the labels at the top of the table
3. For each line in the table (using variables celsius and fahrenheit):
   a. Set celsius equal to the next Celsius temperature of the table
   b. fahrenheit = the celsius temperature converted to Fahrenheit
   c. Print the Celsius and Fahrenheit values with labels on an output line

What makes a good decomposition?

- Subtasks should help you produce short pseudocode

- No more than a page of succinct description to solve the entire problem, and ideally much less than a page

Consideration for selecting good subtasks

- **Code reuse**
  - A function is written with **sufficient generality** that it can be reused elsewhere

- **Future changes** to the program
  - Example: Our temperature program might be modified to convert to Kelvin degrees instead of Fahrenheit
  - If the conversion task is performed by a separate function, much of the modification will be confined to this one function

PRECONDITIONS AND POSTCONDITIONS

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.

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

```
void write_sqrt(double x)

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

   . . .
```

Example

```
void write_sqrt(double x)

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

}
```

The precondition and postcondition appear as comments in your program.

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

```
void write_sqrt( double x)

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

- The postcondition always indicates what work the function has accomplished. In this case, when the function returns the square root of **x** has been written.

Another Example

```
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 ?

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

true

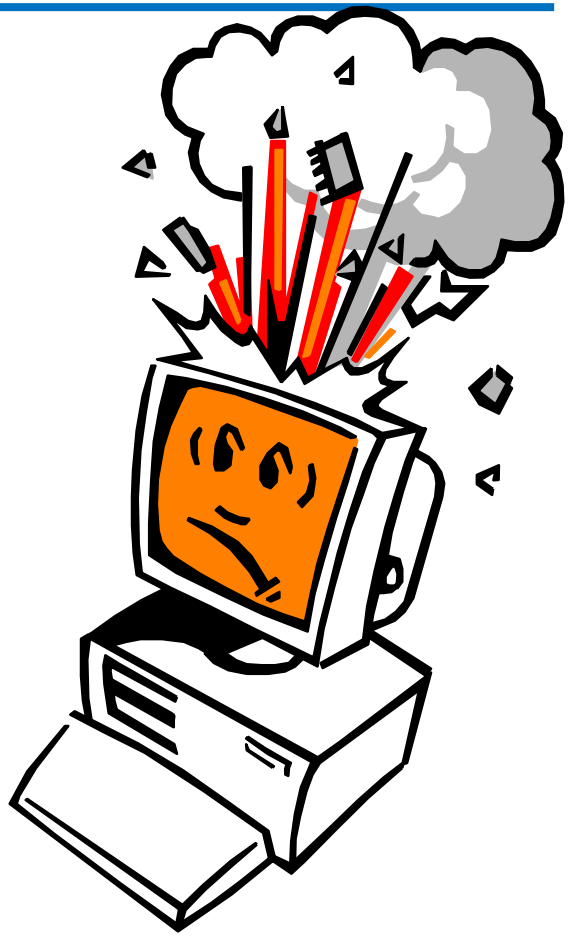
false

Nobody knows, because the precondition has been violated.

Another Example

What values will be returned by these function calls ?

```
is_vowel( '?' );
```



Violating the precondition might even crash the computer.

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.

THEN MY FUNCTION WILL EXECUTE,
AND WHEN IT IS DONE,
THE POSTCONDITION WILL BE TRUE.
I GUARANTEE IT.

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 40-day flood or other disaster?

- You
- The programmer who wrote that torrential function
- Noah

A Quiz

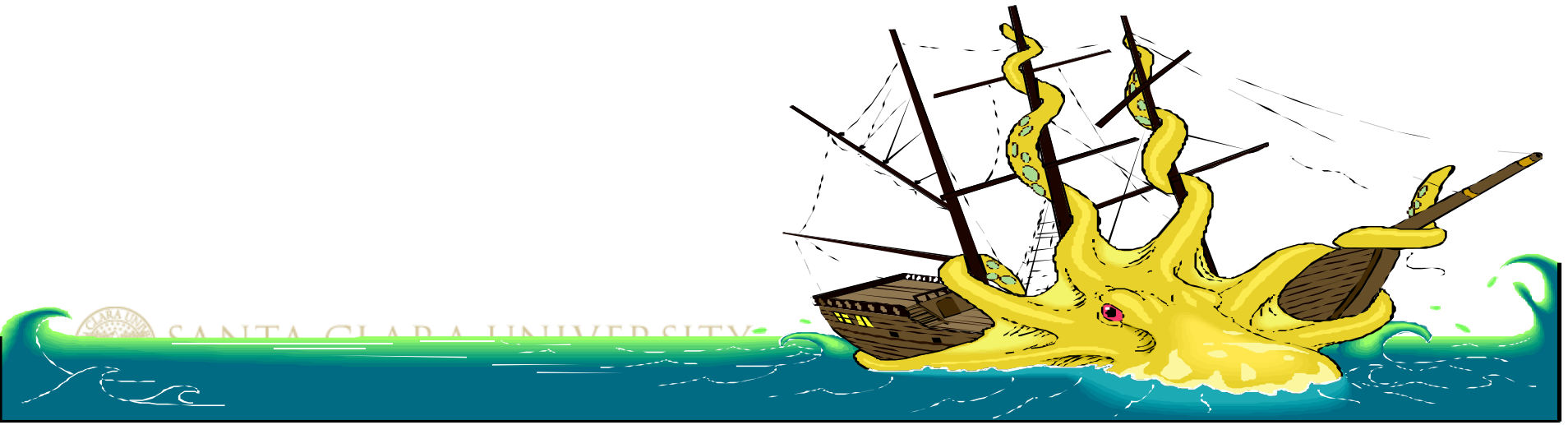
Suppose that you call a function, and you neglect to make sure that the precondition is valid.
Who is responsible if this inadvertently causes a 40-day flood or other disaster?

✶You

The programmer who calls a function is responsible for ensuring that the precondition is valid.

- 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 disaster.



Example

- The assert function is useful for detecting violations of a precondition.

```
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 preconditions/postconditions

- Succinctly 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.

Temperature Conversion Program

Standard Library and Standard Namespace

- **Standard Library aids programmers in writing portable code**
  - can be compiled and run with many different compilers
  - on different machines
- "include directive" at the top of the file
  - To use the usual C++ input/output facilities: **`#include <iostream>`**
  - Some additional input/output items require a second include directive: **`#include <iomanip>`**

Standard Namespace

Older Names for the Header Files

- Older compilers used iostream.h instead of simply iostream

Standard namespace, also called std

- All of the items in the new header files are part of a feature
- When you use one of the new header files, your program should also have this statement after the include directives:

```
using namespace std;
```

Use declared constants

❖ `const double TABLE_BEGIN = -50.0;`

- The value never be changed while the program is running
- Use all capital letters
  - ✓easy to identify such values within a program
  - ✓but well-known formulas may be more easily recognized in their original form (ex. F=9/5C+32)

Use Assert to Check a Precondition

```
#include <cassert>
    ⋮
assert(c >= MINIMUL_CELSIUS);
```

- ***assert*** facility is a good approach to detecting invalid data at an early point
- To use ***assert***, the program include a directive
- After testing and debugging, assertion can be turned off by placing #define NDEBUG before the include directives

`return EXIT_SUCCESS;`

- Defined in cstdlib (or stdlib.h)
- Ends the main program
- And also sends the value of the constant EXIT_SUCCESS back to your computer's operating system
- The return value of EXIT_SUCCESS tells the operating system that the program ended normally, and the operating system can then proceed with its next task
- **zero** for most operating systems (so you may have used return 0 in other programming)

Exercises

Prototype:

int date_check (int year, int month, int day);

Preconditions/Postconditions:

// Precondition: The three parameters are a legal year, // month, and day of the month. //
Postcondition: If the given date has been reached on // or before today, then the function returns 0. //
Otherwise, the value returned is the number of days // until the given date will occur.

1. date_check (2017, 4, 14)
2. date_check (2017, 4, 3)

Exercises

1. What is the ***using*** statement that must appear before using any of the items from the C++ Standard Library?

2. What is an ***assert*** statement that checks whether precondition is valid?

RUNNING TIME ANALYSIS

Running Time Analysis

- Time analysis consists of reasoning about an algorithm speed:
  1. Does the data structure or algorithm work fast enough for my needs?
  2. How much longer does the method take when the input gets larger?
  3. Which of several different methods is fastest?

Running Time Analysis (cont.)

- **Not the actual time taken to run the program**
  - Because the number of seconds can depend on too many extraneous factors (e.g., processor speed, the effect of other processes)

- **The analysis counts the number of operations required**
  - An operation:
    - ✓ As simple as the execution of a single program statement
    - ✓ Arithmetic operation (addition, multiplication, etc.)
  - For most programs, **the number of operations depends on the input size**
    - ✓ For example, a program that sorts a list of numbers is quicker with a short list than with a long list

Big-O Notation

- Often enough to know in a rough manner **how the number of operations is affected by the input size**
- Express in a format called **big-O notation which is the order of an algorithm**
- The order analysis is often enough to compare algorithms to estimate how running time is affected by changing data size
- For example, the complexity of insertion into a linked list is O(1), while the complexity of insertion into a sorted array is O(n)

Linear algorithms

- Largest term in a formula is a constant times *n*
- **big-O of *n***
- *O(n), and the algorithm is called* ***linear***

- Example: What is the big-O notation for this formula?

  $100n+5$

  $=> O(n)$

Quadratic algorithms

- Largest term in a formula is no more than a constant times n^2
- **big-O of *n2***, and written $O(n^2)$
- The algorithm is called **quadratic**

- What is the big-O notation for this formula?

$n^2 + 5n$:

$=> O(n^2)$

Logarithmic algorithms

- Largest term in a formula is a constant times a logarithm of *n*
- **big-O of the logarithm of n**, and written $O(\log n)$
- The algorithm is called **logarithmic**

- What is the big-O notation for this formula?

$$1000 \log n + 5 \quad :$$

$$=> O(\log n)$$

Guessing Game

- **Worst-case analysis**: determines the maximum number of operations required for the inputs of a given size *n*

- **Average-case analysis**: determines the average number of operations required for the inputs of a given size *n*

- **Best-case analysis**: determines the fewest number of operations required for the inputs of a given size *n*

TEST AND DEBUGGING

Program testing

- Program testing occurs when you run a program and observe its behavior
- **A set of test inputs that is likely to discover errors**

- Choosing Test Data
  1. You must know what output a correct program should produce for each test input
  2. The test inputs should include those inputs that are most likely to cause errors

Boundary Values

- **Identifying and testing inputs called boundary values**
- Boundary values of a problem **are particularly apt to cause errors**

- Example:

```
int time_check(int hour);
// Precondition: hour lies in the range
// 0 <= hour <= 23
```

  - Two boundary values for *time_check* are hour equal to 0 and hour equal to 23

Boundary Values (cont.)

- If you cannot test all possible inputs, at least test the boundary values
  - For example, if legal inputs range from 0 to 1000000, then be sure to test input 0 and input 1000000
  - It is a good idea also to consider 0, 1, and –1 to be boundary values whenever they are legal input

Fully Exercising Code

- Make sure that **each line of your code is executed at least once** by some of your test data
- If there is some part of your code that is sometimes skipped altogether, then make sure that **there is at least one test input that actually does skip this part** of your code

- **Profiler**
  - a software tool that many compilers have
  - generate a listing indicating how often each statement of your program was executed
  - This can help you spot parts of your program that were not tested

Debugging

- Fixing the errors in programming
- Debugging Tip:
  1. Never start changing suspicious code on the hope that the change "might work better"
  2. Instead, discover exactly why a test case is failing and limit your changes to corrections of known errors
  3. After correcting a known error, all test cases should be rerun
- Use a software tool called a **debugger** to help track down exactly why an error occurs

Summary

- One good method for specifying what a function is supposed to do is to provide a **precondition and postcondition for the function**
- Understanding and using the **C++ Standard Library can** make program development easier
- **Time analysis** is an analysis of how many operations an algorithm requires
- Three important examples of big-*O* analyses are **linear** ($O(n)$), **quadratic** ($O(n^2)$)**, and logarithmic** ($O(\log n)$)
- An important testing technique is to identify and test **boundary values**
- A second important testing technique is to ensure that your test cases are **fully exercising the code**

References

1. C++: Classes and Data Structures, by Jeffrey Childs
2. http://en.cppreference.com
3. http://www.cplusplus.com
4. https://isocpp.org

Copyright Notice

Part of this lecture was adapted from the slides of Dr. Behnam Dezfouli