CSE 143

Principles of Programming and Software Engineering

Textbook: Chapter 1

CSE 143 C++ Programming Style Guide
(in course packet and on the web)

06/21/01 C-1

Programming is...

- ...just the beginning!
- Building good software is hard
  - Why?
  - And what does "good" mean? or "bad?"
- "Software engineering" = "techniques to facilitate development of computer programs"
- Problem-solving is more than just programming
- Today: some issues, terminology, and techniques
- Throughout the course: more and more techniques

06/21/01 C-2

Footnote on "Software Engineering"

- "Engineer" has a specific legal connotation in many professions
  - Licensing procedures
  - Legal implications
- That has not been true in software engineering
- That may be changing
  - Texas recently became the first state to license software engineers

06/21/01 C-3

The Software Lifecycle

- Big SW programs are expensive to develop, long-lived, and critical to their users
- Typical stages (iterate as needed):
  - Analysis and Specification
  - Design
  - Coding
  - Testing
  - Production
  - Maintenance
- You guess: which stage is the biggest?

06/21/01 C-4

Lifecycle in a Typical HW

- Analysis and Specification
  - Assignment Description
    - `May be ambiguous!`
  - Sample executable
- Design
  - Some of the design is implied by what you're given
  - Sometimes, part of your job is "reverse engineering"
- Coding
  - Make sure you do it in style – quality counts!
- Debugging

06/21/01 C-5

Software Lifecycle in HW

- Testing
  - We may provide some test data
  - You need make up data of your own
    - `Maybe with data errors, too.`
- Production
  - Who are the users: TAs while grading!
- Maintenance
  - Is there life for homework after turn-in??

06/21/01 C-6

Software Engineering Issues

Let's say you're given a LARGE program to modify...

What do you look for?

06/21/01 C-7

Software Engineering Issues

So when you design your programs...

Will you meet these criteria?

06/21/01 C-8

A Key Goal: Modularity

- "Module:" self-contained unit of code
- Large systems are viewed as composed of modules
- Ideally, modules are independent
  - Don't depend on each other except in clear-cut ways
  - Can be independently modified
  - Isolate errors
  - Can be developed separately
  - Can be reused

06/21/01 C-9

Achieving Modularity

- Easier said than done!
- Many ways a system could be divided into modules
  - not all are equally good
- Abstraction: separating the concept from the details of implementation
- Top-down programming
  - Hierarchy of functions
- Object-oriented Programming: identifying "objects" that contain both data and operations
  - more later

06/21/01 C-10

Down to Earth: Modules in C++

- Large C and C++ programs are written as lots of separate source and header files
- .cpp ("source" or "implementation") files
  - Contain a group of related functions
  - Later: methods (functions) from a `class`
- .h ("header" or "specification") files:
  - constant definitions
  - function prototypes
  - type definitions
  - Later: class declarations

06/21/01 C-11

Putting Pieces Together

- Each .cpp file has #includes for any .h files it needs.
- Each .cpp file is separately compiled
  - Each compilation creates an "object file"
    (May be part of a database kept by development system)
- A .h file may have #includes for other .h files
- A .h file does not contain #includes for .cpp files
- A .h file is not compiled by itself
- The linker combines:
  - all the object files of your project
  - any needed external object files or libraries

06/21/01 C-12

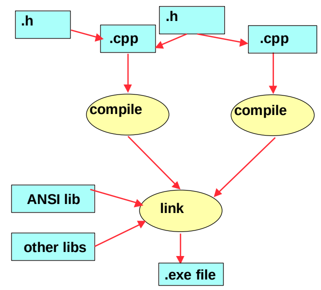
Building the Project

- Programmer has to define a "project"
  - specify which .cpp files are to be used
  - large projects may have dozens or hundreds of source files
- In modern development environments like MSCV...
  - you do this with mouse clicks and menus
  - many options and settings are available
  - "Build" button may automatically perform many steps of compilation and linking
- In command-line systems
  - Programming creates a "make file" to describe all the project files and how they are to be combined
- Eventual result is one big executable file

06/21/01 C-13

Build Steps

- Lots of individual steps happen when the project is built
- If no errors, result is one executable file



06/21/01 C-14

Testing

- How do you know the program is correct?
  - One way: Test it!
  - Microsoft is said to have one tester for every developer
- Try as many relevant "test cases" as you can
  - Many errors only show up in a few test cases
  - What is a “successful” test case?
- *Sad fact of life: It is difficult or impossible to construct a perfect set of test cases*

06/21/01 C-15

An Approach to Testing

- Testing should be a controlled experiment to verify that the program works as intended
- Implications
  - Design first – know what you expect to happen
  - Record the design in comments so you (and consultants, TAs, instructors) can understand what you’re trying to do and check that against actual code
  - Develop tests as (actually **before**) you develop code
- No!
  - Changing code randomly to see if things get “better”
  - “I’ll add the comments once it works”

    `WASTE OF TIME – GUARANTEES MORE DEBUGGING!!`

06/21/01 C-16

Some Testing Advice

- Test normal cases
  - "live" data is nice when available
- Test extreme cases
  - Very small data sets
  - Very large data sets
  - Situations that are peculiar but legal
  - Even if a situation is unlikely in the real world, it can help find bugs

    `Takes unusual paths through the program`
- Test error cases
  - To make the program more robust

06/21/01 C-17

Debugging

- cout at appropriate points
  - show key variables
  - trace execution flow
- Debugger tool
  - Execute code one line at a time
  - Run to a particular program point, then stop
  - Look at variable values anywhere in program
  - Truly an amazing tool... how can you live without it?? Why would you want to???

06/21/01 C-18

Invariants

- Another tool for correctness
- "Invariant": something that must be true at a particular point in a program
- Three especially common code invariants
  - "Precondition": must be true on entry to a function (or the function is not guaranteed to work)
  - "Postcondition": must be true on exit from a function (the function promises this)
  - "Loop invariant": must be true on every iteration in a loop
- Data invariants: Property of a variable (or set of related variables) that should hold true at all times.

06/21/01 C-19

Example: Find Invariants for this Search Function

```
int findMax(int array[], int arraySize)
{
 int max = array[0];

 for (int i = 1; i < arraySize; ++i)

     if (max < array[i])

         max = array[i];

 return max;
}
```

06/21/01 C-20

Writing Invariants

- It's a good habit to form!
- Often should be recorded as comments
- Maybe be translated into code (manually)
  - e.g. as "sanity-checking" code
- In C/C++, simple (boolean) invariants can be coded as "asserts"
  - checked at run-time
  - error message given if assertion fails
  - poor user interface, but terrific debugging tool

06/21/01 C-21

Checking Preconditions

- Example: Average a list of numbers

```
double average(int nums[], int len);
// PRE: len > 0
// POST: Returns average of
//       nums[0]..nums[len-1]
```

- What happens if `len <= 0`?
  - `average` makes no sense!
  - Need to make sure precondition always holds
- Clients (callers) should never call average with `len <= 0`
  - But what if they do?

06/21/01 C-22

The `assert` macro

```
#include <assert.h>

double average(int nums[], int len)
{
  assert(len > 0);
  int sum = 0;
  for (int j = 0; j < len; j++)
     sum = sum + nums[j];
  return ((double) sum / (double) len);
}
```

- If an error occurs, program exits, printing:

```
Assertion failed:  len > 0
file main.cpp, line 23
```

06/21/01 C-23

Assert: Verifying Correctness

- Value of the `assert` macro
  - Double-checks that your program is correct
  - Finds errors early
  - Identifies the buggy part of your program
- Use it for all machine-checkable invariants

06/21/01 C-24

Assert vs. Error Checking

- Use *asserts* to catch programming errors
- Use explicit error checking to catch bad data from user.
  - Ideally, a program should always detect and recover from bad input
    - Even if "recover" just means a graceful exit

06/21/01 C-25

Do it with style, too!

- Other people will read your programs
  - If they can't understand your program, that's bad...
  - (especially if they're your TA! – or boss!!)
- You will read your program
  - (6 months later when you've forgotten it all)
- Your program will change
  - Ever try to reorganize someone else's mess?
- Good style reduces bugs

06/21/01 C-26

What Style?

- See the homework style guide on web!
- Comments to show what program is doing
  - e.g., preconditions & postconditions
- Descriptive names
- Many small functions
  - Less than 1 page long
- Use formatting to show structure
- Assertions used to check invariants
- Avoid global variables

06/21/01 C-27

Commenting: Bottom Line

- File heading - name/date/contents
  - For CSE143, also id #, section
- Function heading comments
  - Everything caller needs to know to use function
  - Must include description of parameters
  - Include pre/post conditions if you have them
- Description of major variables and data structures
  - What's in them, not how they're used
  - Describe relationship between separate variables
    - Often useful: data invariants
- Comments in code as needed to describe sequence of statements, non-obvious algorithms, etc.

06/21/01 C-28