# CSE 143

## Principles of Programming and Software Engineering

Textbook: Chapter 1

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

6/26/00 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
- Later: more and more techniques

6/26/00 C-2

## Footnote on "Software Engineering"

- "Engineer" has a specific legal connotation in many profession
  - 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

6/26/00 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?

6/26/00 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
  - Your job!
  - Make sure you do it in style – quality counts!
- Debugging -- your job, too.

6/26/00 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??

6/26/00 C-6

## Software Engineering Issues

- Correctness (of course!)
- Modularity
  - Module: a piece which has some independence
- Ease of maintenance
- Fail-safe programming
- Style

- All of these influence modifiability, debugging, testing, user (and programmer!) satisfaction, etc.

- By the way... where is efficiency in all this??



## What is a "Correct" Program?

- One that meets its specification
  - What is the spec is incomplete or incorrect?
- OK, how do we know it's correct?
- Techniques for getting it correct
  - Inspection
    - Looking at it carefully
    - Mentally executing
    - Having a peer review it
  - Testing
  - Debugging
  - Invariants



## 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



## 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



## Down to Earth: Modules in C++

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

```
//math.cpp
#include <math.h>
double sqrt(double) {…}
…
```

```
//math.h
double sqrt(double) ;
…
```



## 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 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!!

## 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

6/26/00 C -

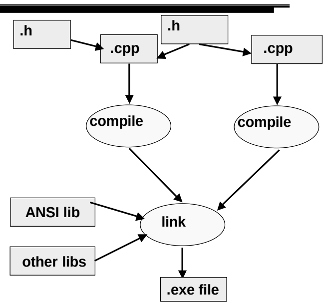
## Building the Project

- 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 systems 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
- Eventual result is one big executable file

6/26/00 C-14

## Build Steps

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




6/26/00 C-15

### A Linker Error in MSVC: “unresolved external”

main.obj : error **LNK**2001: unresolved **external** symbol "**bool __cdecl load_data(char * const, struct team * const, int *)**"

(?load_data@@YA_NQADQAUteam@@PAH@Z)

hw1.exe : fatal error **LNK**1120: 1 unresolved externals

6/26/00 C-16

## 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*

6/26/00 C-17

## 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 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!!`

6/26/00 C-18

## Testing Concepts

- White-box testing
  - look at your code, make sure you test all of it
    - `e.g., test both sides of every if statement`
    - `make sure every function is called, etc.`
- Black-box testing
  - Don't look at code
    - `One person codes, another person tests`
  - Imagine test cases weird enough to break your program
- Regression testing
  - Run the same test cases after every program change
  - Make sure you don't introduce new bugs!



## Testing Incomplete Programs

- Stubs
  - Very simple implementation of part of program
  - Allows you to test another part of program
- Drivers
  - Test one module of program in isolation



## Some Testing Advice

- Use stubs and drivers as appropriate
- 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



## 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???



## 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: Properties of (related) variables that should hold true at all times.



## Example: Search

```
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;
}
```

## 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

6/26/00 C-25

## 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 there is a bug in the program?

6/26/00 C-26

## 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
```

6/26/00 C-27

## 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
  - Required in all homework from now on

6/26/00 C-28

## Use *assert*( ) to aid debugging

- Use *assert* liberally in the programming projects
  - Test preconditions especially, in as much detail as practical
  - Test invariants and postconditions when reasonable
- Don't worry about the overhead
  - Think of your programs as still under debug, even when turned in.
  - It is possible to disable assertion checking in "production" code.
  - MSVC -- automatically disabled in "release" mode

6/26/00 C-29

## Assert vs. Error Checking

- Use *asserts* to catch programming errors
- Use explicit error checking to catch bad data from user.
  - User input should never trigger an assert in production code
  - Ideally, a program should always detect and recover from bad input

`Even if "recover" just means a graceful exit`

6/26/00 C-30

## Masking vs. Reporting Errors

- Think of programs as collections of functions
- When one of these functions is executing and detects an error, what should it do?
- Two main choices
  - 1. "Mask" the error. Fix things up so that it looks to the rest of the program as if no error occurred
  - 2. Report the error

```
Usually, report it to the calling function.
We'll highlight several options for doing this.
Calling function must be prepared to handle the
reported error.
```

6/26/00 C-31

## Option 1: Return a Flag

- "Flag" - boolean variable indicated success/error
- Example:

*bool readMoreData (params....)*

  - The return value simply means "function succeeded/function found an error"
- Advantage
  - simple to check if it's OK

6/26/00 C-32

## Option 2: Return a Special Value

- Special value should be one you don't normally return!
- Example: -1 if normal values are positive
- Advantage
  - fits well if you're already returning something else
- Disadvantage
  - can't use if you could return anything on success!

6/26/00 C-33

## Option 3: Status Functions

- Stream example
  - if (cin.good()) ...
  - if (cin.bad()) ...
  - if (cin.eof()) ...
- Advantage
  - can do several operations, then check for an error
- Disadvantage
  - may not discover error soon enough

6/26/00 C-34

## Option 4: Error Parameter

- Used in textbook
  - see `listClass` functions in chapter 3
  - `void listClass::ListDelete(int Position, bool & Success);`
  - sets success to false if error while deleting

    `e.g. position is invalid`
- Advantages
  - works even if you're already using the return value for something else
  - can use the same error flag for several calls
- Disadvantages
  - can be cumbersome

6/26/00 C-35

## Option 5: Exceptions!

- Very clean way to do error handling
- Basic idea: when error is detected, throw an exception with information about what went wrong
- Client code can "catch" exception and react appropriately (recover, terminate, etc.)
- Kind of complicated in C++
  - Java does it (a bit) better
- We probably won't have a chance to use exceptions in CSE143 – but know the idea

6/26/00 C-36

## 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

6/26/00 C-37

## 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 code structure
- Assertions used to check invariants
- No global variables, goto

6/26/00 C-38