Top-Down Design

- **Hierarchical structure**
  - Define problem at highest level of functionality, then break it down further and further into small subroutines
  - Each subroutine is easy to code and document
- A.k.a. *structured programming*

![Diagram](image.png)
Advantages of Top-Down Design

- Can solve complex problems
  - Break the problem into sequence of smaller tasks
  - Break those tasks into sub-tasks
  - Soon, each task is easy to do!
- Breaking down problem helps us clarify what needs to be done
- Parts of the solution may turn out to be re-usable
- More than one person can work on solution
  - Because there are many separable tasks with defined interfaces (inputs/outputs)
How to Tackle Class Projects

• Class projects tends to be pretty small...
  • Much less documentation, much less verification/QA, working independently

• ...but the basic methodology of how to develop correct code (without banging your head against your desk in frustration or giving up in despair) will apply for industry-scale projects too
Getting Started

• Start early!
  • The problem is nearly always more complex than you think when you first read it

• Study first
  • If you don't understand the course material, fix that before diving into the code

• **READ every word of the directions!**
  • Don't assume
  • Take notes

• Break down the problem into pieces (top-down, object-oriented, etc.)

• **Start on paper (or at a whiteboard)**
  • Have a plan before you implement anything!
Develop Incrementally

• Don't attempt to implement the entire program at once
  • Start with a small, workable part

• After each sub-problem: compile, fix syntax errors, re-compile, execute (test), debug
  • Then add another small part, etc.

• If you hit an error, it probably came from the newly-introduced code
  • Want that segment of newly-introduced code to be as small as possible
Dealing with Compiler (Syntax) Errors

• Once it encounters an error, the compiler can't make much sense of anything that follows
  • **Fix the first one or two errors, then re-compile!**
  • The later ones might be garbage
• If the error sounds like a foreign language, Google the error message (but use caution...)
• Check for the common mistakes
  • Missing semi-colons
  • Misplaced { }, ( )
  • Backwards << or >>
  • Misspelled variable names
Runtime Errors (a.k.a "Bugs")

- Program compiles, but it gives the wrong output (or segfaults, or runs out of memory, ...) when it is executed

- **Debugging**: figuring out why program failed to execute as expected

Some ways to debug...

- Add print/cout statements in strategic places
  - Print variable values frequently, see what's changing when
  - Trace execution path by printing on function enter/exit or anywhere else you're unsure if a statement is reached
  - *Print debugging is not failproof, but in my experience it's enough to find 99% of coding errors*

- Debuggers (e.g., GDB)
Testing

• What is software testing?
  • Running a program (or part of a program) to find bugs
  • Or to increase confidence that there are no bugs

• What is software testing NOT?
  • Static analysis: examining the program's code looking for bugs, without running the program
    • (Also important to do... it just isn't testing)

• Why do software testing?
  • Ideally, we could prove (using formal techniques) program correctness
  • Difficult for even some trivial or small programs, usually impractical or impossible
Testing Questions

• Which executions of a program should we run?
  • We can rarely feasibly test *ALL* possible inputs
  • In this class, we can strategize about *boundary conditions*:
    • Empty arrays, full arrays, first element, last element, etc.
    • Smallest and largest valid values
    • Values used in if or while conditions
  • In large projects: statement/decision/path coverage, constrained randomization... but still focus on boundary conditions

• How do we know if an execution reveals a bug?
  • In this class, we can manually check program output against what we expect
  • In large projects: reference models, assertions, monitors and checkers...
Developing/Executing a Test Plan

- Identify the boundary/edge cases
  - Do this before you write any code!
  - The edge cases often fall straight out of the specification, not the code
  - Come up with test case(s) for each boundary case
  - Sometimes this process will uncover edge cases you didn't know about, and your code will be better

- During implementation, refine your list of test cases

- Code a test driver
  - This often lives in main() for small projects
  - Test each function separately before testing the whole

- Run your test code