Programming in the Middle

• How should functions be organized?
• Short functions easy to debug and test
• If you can't briefly describe what a function does, break it up!
• Essential question: how do functions communicate with each other? That is, what are their interfaces?
• Use data structures such as a struct to package information and simplify interfaces
Interface Design

- What arguments are needed?
- Which are input or output?
- Pass input arguments by value, or `const` reference if copying is expensive
- Output arguments should be returned (if only one) or passed by reference
- Use multiple interfaces (by overloading or default arguments) for expert or novice usage

Top-down and Bottom-up

- Top-down design: start with high-level description of task, break it down into subtasks, and repeat on subtasks
- Bottom-up design: implement operations on low-level data structures, then build functionality on top of them
- Can start in the middle, work both ways
- Recommended: begin with top-down to understand high-level structure, then use bottom-up to build components
Don't Hard-wire!

- **Hard-wiring** is the practice of imposing fixed choices on the design of the code
- Re-wire code by *refactoring*: changing design of the code, not what it does
- Hard-wired: integrating \( \cos(x) \) using the trapezoidal rule with 20 points
- Re-wired: a function that implements the trapezoidal rule, passing the integrand and number of points as arguments

What to Comment

- Interfaces! These comments are "contracts" with users of your code
- Loop invariants and assertions: document conditions that are guaranteed to be true
- Warnings about unusual behavior
- What the code is trying to do
- What not to comment: don't simply restate what the code is doing!
Documentation

• User documentation: how to use the code, but not how it works
• Developer documentation: for other programmers, to change code for their purposes
• Document only what is necessary; long documents won't be read
• Be sure to keep documentation in sync with your code

Cross-Language Development

• Code is rarely translated into other languages (too much time!)
• Applications often mix languages
• To help, compiled languages are translated into same object code
• Must account for conventions in:
  – naming functions
  – passing arguments
  – layout of data structures
Cross-Language Strategies

- Most features of languages are implemented in ways that limit compatibility: classes, input/output, etc.
- Calls to functions in other languages should be restricted to simple exchange of data only
- Write routines to convert data structures
- Java provides JNI (Java Native Interface) to call compiled code
- Fortran 2003 provides C types

Procedural vs Object-Oriented

- In **procedural** programming, code and data are independent
- In **object-oriented** programming, code and data are packaged into objects
- C is procedural; C++ is both
- Procedure more efficient, similar to assembly language
- Object-oriented supports abstraction and modularity through data hiding
- Can be more efficient using **inline**
Layered Software

- Large software projects are typically divided into layers that correspond to (higher or lower) levels of functionality
- Example: LAPACK is built on top of BLAS, which contains low-level functions that work directly with matrices, while LAPACK focuses on algorithmic details
- Code should not reach across layers; communicate only through interfaces

Next Time

- Calling C++ from MATLAB
- And vice versa!