CS 370
REVIEW: The Pseudocode Programming Process

DR. MICHAEL J. REALE
FALL 2015
Pseudocode

“Pseudocode”
- Informal, English-like notation for describing how an algorithm/routine/class/program will work
- Streamlines creation of code in routines
Good Pseudocode

- **Good pseudocode:**
  - **English-like statements** to precisely describe specific operations
  - **Avoids syntactic elements** from target programming language
  - Balance between:
    - **High level**
      - Write at **level of intent** → MEANING rather than HOW
    - **Low level**
      - Generating code will be nearly automatic
  - Results in **useful comments**
Advantages of Good Pseudocode

- Makes design reviews easier
- Can catch errors before writing code
- Can change pseudocode more easily than code
- Minimizes commenting effort
- Pseudocode $\rightarrow$ good design documentation
  - Easier to maintain
  - More likely to be accurate
Routine Construction

There are five stages to constructing a routine:

- Design the routine
- Code the routine
- Check code
- Clean up loose ends
- Repeat as needed
Design the Routine

- Define:
  - Whether routine called for in **prerequisites**
  - **Problem** the routine will solve
  - Clear and unambiguous name

- Think about:
  - Testing
  - **Error handling** → what can go wrong?
  - **Efficiency** → critical or not?
  - **Data types** you need

- Research:
  - Available libraries
  - Existing **algorithms** and data types

- Write:
  - **Header comment** → describe routine
  - High-level **pseudocode**
  - **Recursively write** pseudocode → stop when at low enough level

- Review:
  - Check pseudocode
  - Make sure you understand it!
  - **Iterate** and try a few ideas
Code the Routine

- Write declaration
- Pseudocode → comments
- Fill in code beneath pseudocode comments
  - Writing metaphor actually works here:
    - Comments → “Outline”
    - Code → “Text”
- Each comment → block of code (complete thought expressed by comment)
  - Usually 2-10 lines of code per comment
  - If code seems too long:
    - 1) Move code into its own routine (reapply PPP)
    - 2) Apply PPP recursively → keep breaking it down
Check the Code

- **Check before compilation**
  - Look for:
    - Problem with design due to low-level implementation issue
    - Typing errors
  - Mentally run code
- **Compile the routine**
  - Why not compile earlier?
    - *Pros of compiling early*: clear out syntax errors
    - *Cons of compiling early*:
      - “Just One More Compile” syndrome → start rushing and hacking stuff together just to get rid of syntax errors
  - Eliminate causes of all error messages and warnings
    - Warnings → usually a sign of low-quality code
    - Important warnings → buried in “unimportant” warnings
Clean Up Loose Ends

• Take a look and check you have:
  ○ Good function interface
  ○ Central purpose
  ○ Good variables
  ○ Good statements and logic
  ○ Good layout
  ○ Accurate comments
  ○ Remove redundant comments
Alternatives to PPP

- **Test-first development**
  - Test cases written before writing any code

- **Refactoring**
  - Write code, but then iteratively improve through semantic-preserving transformations
    - Look for patterns of bad code (or “smells”, as Fowler refers to them)

- **Design by contract**
  - Define routine preconditions and postconditions

- **Hacking?**
  - Bad idea...