CS 370
The Pseudocode Programming Process

DR. MICHAEL J. REALE
FALL 2015
Introduction

- At this point, you are ready to beginning programming at a lower level

- How do you actually write your classes and functions?
  - *One approach: Pseudocode Programming Process (PPP)*
Pseudocode

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

In the following slides, we will cover:
- How to write good pseudocode
- How to construct routines using pseudocode
- Other alternatives to pseudocode
Good Pseudocode
What Makes Pseudocode Good?

- Any arbitrary English description won’t be as useful

**Guidelines:**
- Use **English-like statements** to **precisely describe specific operations**
  - E.g., “Do stuff” is not very instructive
- **Avoid syntactic elements** from target programming language
  - Pseudocode should be higher level than actual code
  - E.g., “allocate using malloc” is specific to C
- Write at the **level of intent** → describe MEANING of approach rather than HOW it will be implemented
  - E.g., Say “Keep track of current number of resources” rather than “increment resource number by 1”
- Write at **low enough level** that generating code will be nearly automatic

**Pseudocode = balance between being:**
- Too high level → not instructive
- Too low level → no more instructive than the code itself

**Related question to ask:**
*Will this pseudocode result in good/useful comments?*
Example of Bad Pseudocode

- Increment resource number by 1
- Allocate a dlg struct using malloc
- If malloc() returns NULL then return 1
- Invoke Osrsrc_init to initialize a resource for the OS
- *hRsrcPtr = resource number
- Return 0

**Problems:**
- Poorly written
- Includes target coding language details (C-specific pointer notation and malloc())
- Focuses on how code will be written rather than meaning
- Coding details → whether function returns 1 or 0
Example of Good Pseudocode

- Keep track of current number of resources in use
- If another resource is available
  - Allocate a dialog box structure
  - If a dialog box structure could be allocated
    - Note that one more resource is in use
    - Initialize the resource
    - Store the resource number at the location provided by the caller
  - Endif
- Endif
- Return true if a new resource was created; else return false

Better because:
- Written entirely in English (no language-specific syntax, so could be implemented in any language)
- Written at level of intent → meaning easier to determine
- Precise and detailed enough to implement in code
- Will produce good comments
Advantages of Good Pseudocode

- **Makes reviews easier**
  - Can review low-level **design** without having to look at the actual code

- **Supports the idea of iterative refinement**
  - High-level design → *refine to* pseudocode → *refine to* source code
  - Catch errors earlier in pipeline

- **Makes changes easier**
  - Can change/refine/modify pseudocode a lot easier than actual code
  - Again, allows catching errors in design earlier

- **Minimizes commenting effort**
  - Pseudocode → becomes the comments!

- **Pseudocode is easier to maintain than other forms of design docs**
  - Right in line with code → more likely to be accurate (and easier to keep up to date)
Constructing Routines by Using the PPP
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

- **Check the prerequisites**
  - Is this routine part of the design?
  - Do the requirements call for (at least indirectly) this routine?

- **Define the problem the routine will solve**
  - In particular, define:
    - Information/process the routine will hide
    - Inputs
    - Outputs
    - Preconditions
    - Postconditions

- **Name the routine**
  - Name should be clear and unambiguous
    - If you can’t come up with such a name, maybe purpose of routine isn’t clear
Design the Routine

- Decide how to test the routine
  - Useful for unit testing (we’ll talk about this later)

- Think about error handling
  - What can go wrong?
  - What values could be invalid, and what will happen?
Design the Routine

- Think about efficiency
  - Either:
    - Efficiency not critical → keep good abstraction for interface and code readable
      - Can always update/optimize later
    - Efficiency IS critical → tune to meet your resource budgets
  - Usually, though, don’t go nuts with optimization until later (when you can assess where the real bottlenecks are)
Design the Routine

- Research functionality available in standard libraries
  - Make sure you’re not driving yourself nuts reinventing the wheel
  - May be a “buy vs. build” decision

- Research algorithms and data types
  - If you can’t find a library to do the job for you, check to see if there is an existing algorithm you can use
Design the Routine

- **Write the pseudocode**
  - Just write it in your code editor
  - Start general → work towards more specific
    - Write header comment that briefly and succinctly describes routine
    - Fill in high-level pseudocode
    - Recursively write pseudocode until you are at a low enough level
Design the Routine

- **Think about the data**
  - Are you doing a lot of data manipulation?
  - What data types do you need/want to make this process work well?
Design the Routine

- **Check the pseudocode**
  - REVIEW your pseudocode
  - Think about how you would explain it to someone else (or actually try to explain it to someone else)
  - Make sure you understand it!
    - If you don’t, no one else will.

- **Try a few ideas and keep the best (iterate)**
  - Once you start coding → emotionally invested → harder to throw away bad design
  - Decompose until you can see exactly how to code it
Code the Routine

- Write the routine declaration
- Turn the header description into a comment
- Turn the pseudocode into comments
  - At this point, turning this into code should feel natural
  - If it doesn’t, keep working on your pseudocode
- Fill in code beneath pseudocode comments
  - Writing metaphor actually works here:
    - Comments \(\rightarrow\) “Outline”
    - Code \(\rightarrow\) “Text”
- Each comment \(\rightarrow\) block of code (complete thought expressed by comment)
  - Usually 2-10 lines of code per comment
Too much code per comment?

- If you find that a given comment produces a lot of code, you have two options:
  - 1) Move code into its own routine (reapply PPP)
  - 2) Apply PPP recursively ➔ keep breaking it down
Check the Code

• Why check it now?
  ○ More expensive to fix later at testing
  ○ Design might have problem due to low-level implementation issue
  ○ Code could have a good-old-fashioned typing error
Check the Code

- Mentally check routine for errors
  - Walk through process → “run it in your head”
    - Another reason to keep routines small...
  - Make sure you understand why it works!
    - Less than 5% of errors → hardware, compiler, or OS errors
    - Suspect your own work first if something’s wrong
Check the Code

- Compile the routine
  - Debatable:
    - *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

- *Guidelines for compiling:*
  - Use pickiest compiler warning level
  - Use validators (e.g., lint)
    - ASIDE: Android Studio automatically runs lint
  - Eliminate causes of all error messages and warnings
    - Warnings → usually a sign of low-quality code
      - If you ignore them, important warnings might get buried in there and ignored as well!
Check the Code

- Step through the code in the debugger
- Test the code
  - Run test cases you’ve written
- Remove errors from the routine
  - If the routine is REALLY buggy, start over and rewrite it
Clean Up Loose Ends

- Take a look and check you have:
  - Good function interface
  - Central purpose
  - Good variables
    - Good names, proper initialization, no unused objects, etc.
  - Good statements and logic
    - Look for off-by-one errors, infinite loops, improper nesting, resource leaks
  - Good layout
    - E.g., good use of whitespace
  - Accurate comments
  - Remove redundant comments
    - May have pseudocode comments that are almost exactly the same as the code
Repeat As Needed

- If the routine is still awful after all this ➔ start over at pseudocode stage...
Alternatives to PPP
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...