CS 370
REVIEW:
High-Quality Routines

DR. MICHAEL J. REALE
FALL 2015
Routine Defined

- **Routine**
  - An individual method or procedure invocable for a single purpose
  - Greatest programming technique ever invented for saving space and improving performance
Reasons to Create a Routine

- **Reduce complexity**
  - Hides:
    - Information/processes
    - Sequences
    - Pointer operations
    - Non-portable code → improves portability
    - Complicated boolean tests
  - Intermediate, understandable abstraction
    - Use self-documenting name

- **Avoid duplicate code**
  - Saves space
  - Also creates central point of control

- **Improve performance**
  - One place to optimize code
Routine Creation

- May make routine even from seemingly simple line(s) of code:
  - Self-documenting
  - May have to expand in the future
Cohesion

Cohesion (for routines)
- Refers to how closely the operations in a routine are related
- Not universally “good” attribute → depends on type of cohesion
  - Good → functional cohesion
    - Performs ONE and ONLY ONE operation
    - Should correspond to name
  - Less-than-ideal:
    - Operations share data from step to step
    - Operations use same data (unrelated otherwise)
    - Operations all run at same time
      - Bad if hodge-podge of different code
      - Better if operations = other functions
  - Bad (operations not related otherwise):
    - Operations in specific (effectively arbitrary) order
    - Disparate operations executed based on control flag (giant switch statement)
      - Good exception: calling other routines → event handler
    - No relationship; just in same routine for no reason
A good routine name:
- Describes everything the routine does
  - Length → as long as necessary, BUT:
    - If routine attached to object, part of name already there
    - If name ridiculously long → may have too many side effects in routine
  - Avoid meaningless, vague, or wishy-washy verbs
  - Doesn’t differentiate routine names solely by number
  - Describes return value if routine returns a value
  - Uses a strong verb followed by an object
    - Again, if routine in class, may not need object in name
  - Uses a convention consistent with:
    - Complimentary/opposite routine(s)
    - Other routines in project
Routine Length

- “Line” = non-comment, nonblank line of code

- Most functions will be short
  - E.g., setter/getter functions

- For complex algorithms, 100-200 lines reasonable
  - However, beyond 200 lines, be careful
  - Do NOT have routines that are thousands of lines long!

- In general, want to use cohesion, nesting, # of variables, general complexity, etc. to determine how long the routine is
  - A hard cap on length is less useful
Routine Parameters

- Be consistent in layout/use of parameters
  - Use consistent ordering ⇒ e.g., **input-modify-output** order
  - **Routines with similar parameters** ⇒ consistent order
  - Put **status/error variables last** in parameter list
    - Incidentally to main purpose of routine
    - Output-only parameters anyway

- Good use of parameters
  - **Use all parameters** ⇒ otherwise, remove it
  - **Don’t use routine variables as working variables**
  - **Prevent change of input** variables (if possible)

- Try to limit the # of parameters to about 7+-2

- Consider a naming convention for parameters (e.g., prefix “i_” for input-only)
Document Assumptions

- **Document interface assumptions about parameters**
  - Input-only, modified, output-only
  - Units of measure
  - Status code/error value meaning
  - Ranges of expected values
  - Specific values that should NEVER appear
Variable Passing

• Passing in individual parameters vs. objects
  ○ 1) Case for passing in individual parameters
     – Weaker connection → loose coupling
     – Easier to understand
     – Easier to reuse
  ○ 2) Case for passing in objects
     – Interface of function more stable → can internally get other data from object if it needs to
     – Doesn’t violate encapsulation, since it doesn’t expose the 3 things the routine needs
Variable Passing

- Best choice → depends on **interface abstraction**
  - Specifies certain values (not necessarily from same object type) → pass in values
  - Tied to object itself → pass in object

- **Warning signs:**
  - Should be using *individual parameters* if:
    - Create an object
    - Populate the three fields you need
    - Call routine (passing in object)
    - Extract fields you need
  - Should be *passing in object* if:
    - Changing the parameter list frequently
    - Data comes from the same object
Actual and Formal Parameters

- **Formal parameters** → variables declared in routine definition
  - E.g., `int subtract(int a, int b)` → “a” and “b” are the formal parameters

- **Actual parameters** → values/variables/expressions passed in during routine calls
  - E.g., `x = subtract(3,2)` → 3 and 2 are the actual parameters

- Make sure you pass in the right type of parameters!
  - Don’t want compiler to convert it for you automatically without you knowing it
  - E.g., `x = subtract(3, 2.9)` → will convert 2.9 (float) to 2 (int)
Returning Values

- Return value if name/purpose implies it should
  - *Examples*:
    - `getDistance(Point a, Point b)` → probably should return distance
    - `formatReport(Report r)` → probably shouldn’t return anything

- Can return status variable
  - However, separate test of variable from function call

- Check all possible return paths
  - Good idea to initialize return variable first (default value)

- Don’t return references or pointers to local data!
  - *Local to routine*:
    - Exit routine → variables go out of scope!
  - *Class data*:
    - Store changes inside class → use accessor functions to get data afterwards
Macros

- **Macro**
  - Named piece of code
  - Blind code replacement by preprocessor

- **Using macros:**
  - Fully parenthesize macro expressions!
  - Use curly braces with multi-statement macros
  - Should not be used as a replacement for a routine unless absolutely necessary
  - Should ALWAYS be handled with care
  - Useful for *conditional compilation*
Inline Routines

- Inline routine
  - Instead of calling function, replace function with code from function
  - Unlike macro, inline function = actual function
  - Good for short functions

- **Advantages:**
  - Faster, more efficient → not using calling stack

- **Disadvantages:**
  - Larger code size
  - Requires code to be in header file → violates encapsulation!