CS 370
High-Quality Routines
Introduction

- **Routine**
  - An individual method or procedure invocable for a single purpose
    - *Examples:* function in C++, method in Java, even a macro in C/C++
  - Greatest programming technique ever invented for saving space and improving performance

- However, avoiding code duplication is NOT the only reason to create a routine...
Valid Reasons to Create a Routine
Reasons to Create a Routine

- **Reduce complexity**
  - Software Primary Technical Imperative ➔ managing complexity!
  - Hide information/process so you don’t have to think about it later

- **Introduce an intermediate, understandable abstraction**
  - Put section of code into well-named routine ➔ documents its purpose

- **Avoid duplicate code**
  - Saves space
  - Also creates central point of control
Reasons to Create a Routine

- **Support subclassing**
  - Keeps writing functions in derived classes a lot easier if routines are short and well-written

- **Hide sequences**
  - If things have to be done in a certain order, better to hide that in a routine rather than spreading it all over the rest of the code!

- **Hide pointer operations**
  - Tend to be difficult to read and error-prone
  - Can change to something other than pointers later without breaking everything else
Reasons to Create a Routine

- **Improve portability**
  - Isolate OS/hardware-related/non-portable code to their own routines

- **Simplify complicated boolean tests**
  - *Example*: `canDoBattle()` rather than:
    - `If( unit.alive() && unit.hasWeapon() && unit.isConscious())`
  - Makes code more readable:
    - Details of test are out of the way
    - Descriptive function name summarizes purpose of test

- **Improve performance**
  - Optimize code in one place → again, central points of control!
Should Routines ALWAYS Be Small?

- No, a routine does not ALWAYS have to be small (nor should it be)
  - That said, small routines are easier to work with and usually indicate unity of purpose (good abstraction)
What If the Code Seems Too Simple?

- Often, it’s useful to break something off into a routine, even if it seems too simple.

   *Example:*
   - *Original code:*
     - degrees = 180f * radians / PI;
   - *New code:*
     - degrees = radiansToDegrees(radians);

- *Why?*
  - Self-documenting
  - May find that you need to expand it out into a more involved piece of code
    - *Examples:*
      - What if we always wanted to work with positive degrees?
      - What if we wanted to make sure the value was between 0 and 360?
Design at Routine Level
Cohesion

- **Cohesion (for routines)**
  - Refers to how closely the operations in a routine are related
    - E.g.,
      - \(\text{cosine}() \rightarrow\) high cohesion
      - \(\text{cosineAndTan}() \rightarrow\) less cohesion; trying to do more than one thing

- For classes, usually one refers to abstraction and encapsulation

- For routines, however, cohesion is still a very important concept
Cohesion is not a universally “good” attribute → depends on the *kind of cohesion*

**Functional cohesion**
- When a function performs ONE and ONLY ONE operation
  - Should correspond to name; if not, less cohesive and poorly named
- Strongest and best kind of cohesion
Less-Than-Ideal Cohesion

- **Sequential cohesion**
  - Operations in routine:
    - Must be performed in specific order
    - Share data from step to step
    - Don’t make up a complete function when done together
  - *Example*:
    - Calculate age from birth date
    - THEN use computed age calculate time to retirement
Less-Than-Ideal Cohesion

- Communicational cohesion
  - Operations in routine:
    - Make use of the same data
    - HOWEVER, aren’t related in any other way
  - Example:
    - Calculate age from birth date
    - Compute retirement time from birth date (NOT age)
  - Generally implies you should split function into two functions
Less-Than-Ideal Cohesion

- **Temporal cohesion**
  - Operations combined into routine because they are all done at the same time
  - *Example:* function called startup()
  - Not good if contains a hodge-podge of different code
  - Better if calls other functions that do specific things:
    - E.g., startup() orchestrates things by calling other functions
Bad Cohesion

- **The following assume that, apart from the kind of cohesion specified, nothing else links the operations.**

- **Procedural cohesion**
  - Operations must be done in a specified order, but no other reason to combine them
    - E.g., operations in order that employee data is entered from user

- **Logical cohesion**
  - Several operations in same routine → one is selected based on control flag that is passed in
    - Called “logical” → only has cohesion because everything in one big if or switch statement
  - *Exception:* if routine ONLY has if/switch statement and calls to other routines → e.g., event handler

- **Coincidental cohesion**
  - Operations have no discernable relationship to each other; just happen to be in same routine
Good Routine Names
Describe What It Does (But Don’t Be Silly)

- Describe everything the routine does
  - *Problem*: if you have lots of side effects in your routine...
    - E.g., compute report totals and open output file
      - `computeReportTotals()` → not descriptive enough
      - `computeReportTotalsAndOpenOutputFile()` → silly
  - *Solution*: try to make routines that make things happen directly and don’t have side effects

http://1.bp.blogspot.com/-Yw6bz1YMdV8/VH8UTXIGcJl/AAAAAAAAAD8k/FXdkiyPq_kU/s1600/monty%2Bpython%2Bgeneral%2Btoo%2Bsilly.jpg
Don’t Be Wishy-Washy!

- Avoid meaningless, vague, or wishy-washy verbs
  - *Examples:*
    - HandleCalculation()
    - PerformServices()
    - OutputUser()
    - ProcessInput()
    - DoStuff()
  - Doesn’t tell you what it actually does!
  - Two causes:
    - Could just be the name, not the routine: FormatAndPrintOutput() is better than HandleOutput()
    - Routine itself is wishy-washy and vague
Don’t differentiate routine names solely by number

- Don’t do this:
  - OutputUserPart1()
  - OutputUserPart2()
  - ...

- The numbers don’t really tell you what the routine means
Function Name Length

- Make names of routines as long (or short) as necessary to make them understandable
  - “Optimum” average length: 9 to 15 characters
  - Remember: if routine attached to object, part of name already there
    - E.g., report.print()
  - If name getting ridiculously long → routine might be doing too much
If routine returns a value, make the **name a description of the return value**

- Good examples:
  - cos()  
  - customerID.next()  
  - printer.isReady()  
  - pen.CurrentColor()
Use a strong verb followed by an object
- Strong cohesion → usually performs operation on object
- Examples:
  - PrintDocument()
  - CheckOrderInfo()
- Again, if routine in class, may not need object in name
  - E.g., document.Print()
Opposite Day!

- Use opposites precisely
  - *Examples:*
    - `Open() / Close() → make sense`
    - `FileOpen() / _lclose() → not symmetrical and confusing`
Conventions

- Establish conventions for common operations
  - *Example:*
    - How do we get the ID for each object?
      - `employee.id.name()`
      - `dependent.getID()`
      - `candidate.id()`
    - Especially true when multiple people on project
How Long Should a Routine Be?

- Controversial question
  - Theoretically, approximately 50-150 lines / one screen / 1-2 printed pages
    - IBM → used to limit routines to 50 lines
    - TRW → used to limit routines to two pages
  - Lots of research, but not all of it is still applicable
    - ...and some of it contradictory

- “Line” = non-comment, nonblank line of code
How Long Should a Routine Be?

- 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
Parameters

- 39% of all errors → errors in communication between routines
  - (Basili and Perricone, 1984)

- Good idea, then, to have some **consistency** with how you layout and use parameters for your routines
Ordering of Parameters

- Put parameters in **input-modify-output** order
  - *First* → input-only
  - *Second* → input-and-output
  - *Third* → output-only

- *Example*: void blurImage(Mat origImage, Mat &blurredImage);

- If you prefer reverse ordering (output first, a la C), that’s fine, but **be consistent**
If several routines use similar parameters, put similar parameters in a consistent order

- *Example:* in C, `fprintf()` similar to `printf()`
- *Another example:*
  - *Do:* 
    - void gaussianBlur(Mat origImage, int maskSize, Mat &outputImage);
    - void medianFilter(Mat origImage, int maskSize, Mat &outputImage);
  - *...NOT...*
    - void gaussianBlur(Mat origImage, int maskSize, Mat &outputImage);
    - void medianFilter(int maskSize, Mat origImage, Mat &outputImage);
Use All Parameters

- If you pass in a parameter, make sure you are using it.

- If you are NOT using it → generally, should remove it:
  - Less error prone
  - Cleans up code → less confusing
Good convention to put status/error variables last in parameter list
- Incidentally to main purpose of routine
- Output-only parameters anyway
- Example:
  - void computeAverage(int values[], int cnt, int &errorStatus);
Parameters != Working Variables

- Don’t use routine variables as working variables
  - Use local variables instead
  - E.g., DON’T do this:
    ```
    int addOne(int inputVal) {
        inputVal = inputVal + 1;
        return inputVal;
    }
    ```

- If input parameter, good idea to prevent it from changing if your language supports it
  - C++ → `const` keyword
Document interface assumptions about parameters

- Helps (future) you and others understand how to use the routine
- Can do this in comment before routine

Things that are useful to document:

- Whether params are input-only, modified, or output-only
- Units of measure (is this is feet, meters, fathoms, etc.)
- Meaning of status codes/error values (if not using enums)
- Ranges of expected values
- Specific values that should NEVER appear
Limit Number of Params

- Limit the number of parameters to about 7
  - Based on 7+-2 number

- If passing same data to many different routines → group routines into class?
Consider a naming convention for parameters:

- Add prefix depending on type:
  - i_ → input parameter
  - m_ → modified parameter
  - o_ → output parameter

- Or:
  - Input_
  - Modify_
  - Output_
Variable Passing

- **Example:**
  - Have object with 10 access routines for data
  - Have routine that needs 3 of the data elements from the object

- **Two different schools of thought:**
  - 1) Don’t pass in whole object; just make 3 parameters
    - Weaker connection \(\rightarrow\) loose coupling
    - Easier to understand
    - Easier to reuse
  - 2) Pass in whole object
    - Interface of function more stable \(\rightarrow\) 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

- Which is better?
Variable Passing

- Pass variables or objects that routine needs to maintain interface abstraction
  - If only need 3 elements, and they just HAPPEN to be provided by that object → pass in 3 parameters
  - If abstraction has to do with 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)
Functions and Procedures
Functions and Procedures

- Technically:
  - **Function** → returns a value
  - **Procedure** → does not return a value

- **Key question:** when should a routine return a value?

- **Short answer:** if the routine name/purpose implies it should return a value

- **Examples:**
  - `getDistance(Point a, Point b)` → probably should return distance
  - `formatReport(Report r)` → probably shouldn’t return anything
What About Status Variables?

- Not the end of the world to return a status variable (even if the routine is basically a “procedure”)
  - if( report.FormatOutput(formattedReport) == Success) { ... }

- That said, good idea to separate test from function call:
  - outputStatus = report.FormatOutput(formattedReport);
  - if(outputStatus == Success) { ... }
All Roads (Might) Lead to Rome...

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

```c
int compareValues(int a, int b) {
    int cmp = UNKNOWN;

    if(a > b) {
        cmp = FIRST_GREATER;
    }
    else if(a < b) {
        cmp = SECOND_GREATER;
    }

    return cmp;
}
```
Bad References

- 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 and Inline Routines
Macros and Inline Routines

- The following rules mostly apply to the C++ preprocessor
  o However, applicable to other languages as well

- Macro
  o Named piece of code
  o If preprocessor finds name, (blindly) replaces name with macro code before compilation
  o Blind code replacement
  o Example: #define SQUARE(a) ((a)*(a))
    - int x = SQUARE(3);
    - ...becomes...
    - int x = (3)*(3);

- Inline function
  o Instead of calling function, replace function with code from function
  o Actual function, so only can be done where a function call is appropriate
  o Faster execution (not using calling stack)
  o Expands size of code
  o Good for short functions
Macros and Parentheses

- Fully parenthesize macro expressions
  - **Bad Example:**
    - #define Cube(a)   a*a*a
    - Cube(x+1) → x+1*x*1*x+1
  - **Better, but Still Wrong Example:**
    - #define Cube(a)   (a)*(a)*(a)
    - x = 3
    - ++Cube(x) → ++(x)*(x)*(x)++ → 4*4*4 = 64
  - **Correct Example:**
    - #define Cube(a)   ((a)*(a)*(a))
Multiple-Statement Macros

- If you have multiple statements in your macro \(\rightarrow\) use curly braces!
  - *Bad Example:*
    - #define GO_LEFT( x )
      - \(x = x + 1;\) \(\backslash\)
      - \(x = \text{min}(x, \text{RIGHT\_BOUNDARY});\)
  
  - *Later...*
    - for(int i = 0; i < 100; i++)
      - GO_LEFT(x);
  
  - *Fixed Example:*
    - #define GO_LEFT( x ) {
      - \(x = x + 1;\) \(\backslash\)
      - \(x = \text{min}(x, \text{RIGHT\_BOUNDARY});\) \(\backslash\)
      - }

- NOTE: Not a good idea to use macros in place of actual function calls
MacroNamingConventions

Conventional in C++ ➔ name macros in all caps
- Book suggests using naming convention like routines IF you can replace the macro later
- Problematic: using side effects like ++ and -- can go badly when accidentally using macros
Macros in Summary

- **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

- **Advantages:**
  - Faster, more efficient

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