Routines and Defensive Programming

Chapter 7-8

McConnell

Identify two things that are bad

```
void MyClass::HandleStuff(CORP_DATA &inputRec, int crntQtr, EMP_DATA empRec,
        double &estimRevenue, COLOR_TYPE &prevColor, int expenseType)
{
int i;
for (i=0; i<100; i++) {
        inputRec.revenue[i] = 0;
        inputRec.expense[i] = corpExpense[crntQtr][i];
        }
UpdateCorpDatabase(empRec);
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if (expenseType==1) {
        for (i=0; i<12; i++) profit[i] = revenue[i] – expense.type1[i];
        }
else if (expenseType==2) {
        profit[i]=revenue[i] – expense.type2[i];
        }
}
```

Subroutines

- Perhaps the single greatest invention in Computer Science Programming
- “OK, I know routines are great and use them all the time…so what?”
  - Point is to understand that many valid reasons exist to make a routine and there are right and wrong ways to go about it
  - E.g. In CS A201 you might have been told that subroutines are used to share code and avoid duplication, period.
    - Better explanation…

Valid Reasons to Create Routines

- Reduce complexity
  - Single most important reason
  - Hide information so that you don’t have to think about it
  - Improves maintainability, correctness
  - Indicator: deep nesting of loops or conditionals

Valid Reasons to Create Routines

- Introduce an intermediate, understandable abstraction

```
if (node != NULL) {
        while (node.next != NULL) {
                node = node.next;
                leafName = node.name;
        }
else {
        leafName = “”;
}

OR

leafName = getLeafName(node)            // Self-documenting!
```

Valid Reasons to Create Routines

- Avoid duplicate code
- Support subclassing
- Hide sequences
  - Good idea to hide the order in which events happen to be processed
  - E.g. in Pop stack and decrement stack top
- Hide pointer operations
- Improve portability
- Improve performance
  - Easier to profile to find inefficiencies

Valid Reasons to Create Routines

- Simplify complicated boolean tests
  - Put complex boolean tests into subroutine, rarely need detail for understanding program flow
    - Details of test are out of the way
    - Descriptive function name summarizes purpose
  - Example with a bug:

```
do
{
        // Code here ….
        System.out.println(“Enter ‘A’ for option A, ‘B’ for option B”);
        s = keyboard.next();
        c = s.charAt(0);
} while ((c != ‘A’) || (c != ‘B’));
```

Easier to Read

```
do
{
        // Code here ….
        System.out.println(“Enter ‘A’ for option A, ‘B’ for option B”);
        s = keyboard.next();
        c = s.charAt(0);
} while (!validKey(c));


bool validKey(char c)
{
  if (c == ‘A’) return true;
  if (c == ‘B’) return true;
  return false;
}
```

Small Routines?

- Even a single line of code can be a valid subroutine
  - points = deviceUnits * (POINTS_PER_INCH / DeviceUnitsPerInch());
- Can turn into a inline function:
  - Function DeviceUnitsToPoints(deviceUnits Integer) : Integer

    DeviceUnitsToPoints = deviceUnits * (POINTS_PER_INCH / DeviceUnitsPerInch())
- More readable to use:
  - points = DeviceUnitsToPoints(deviceUnits)

More robust to maintenance

- One liner expanded to

```
Function DeviceUnitsToPoints(deviceUnits Integer) : Integer
    if (DeviceUnitsPerInch() != 0)
                DeviceUnitsToPoints = deviceUnits *
                 (POINTS_PER_INCH / DeviceUnitsPerInch() )
        else
                DeviceUnitsToPoints = 0
        End If
    End Function
```

If original line of code still in dozens of places, the test would be repeated many times...

Cohesion

- Read in text, but skipping (cover in 401)

Good Routine Names

- Describe everything the routine does
  - If this results in a ridiculously long silly name then your routine is probably doing too much
    - ComputeReportTotalsAndOpenOutputFile
- Avoid wishy-washy verbs
  - HandleCalculation, PerformService, ProcessInput, DoOutput
  - Either your name is bad or if it is appropriate then your subroutine is doing too many vague things
- Don't differentiate routines solely by number
  - Part1, Part2, Part3…

Good Routine Names

- Make names as long as necessary
  - Optimum length for a variable 9-15 chars
  - Routines more complex, so longer OK
- To name a function, use a description of the return value
  - Printer.IsReady()
  - Pen.CurrentColor()
  - Customer.NextID()
- To name a procedure (void method), use a strong verb followed by an object
  - PrintDocument()
  - CalcMonthlyRevenues()
  - Object not necessary in an OOP language since the object automatically tells you what the object is
    - Mydocument.Print()

Good Routine Names

- Use opposites precisely
  - Add/remove
  - Increment/decrement
  - Begin/end
- Establish conventions for common operations
  - E.g. if each object has a unique identifier, give a common id() method

How Long?

- Routine size inversely correlated with errors, as the size increased (up to 200 lines) the number of errors per line decreased
  - Basili and Perricone 1984
- Routine size not correlated with errors, although complexity/data were
  - Shen 1985
- Routines < 32 lines of code not correlated with lower cost or fault
  - Card/Church/Agresti 1986
- Small routines had 23 percent more errors per line of code than larger routines but were 2.4 times less expensive to fix
  - Selby and Basili 1991
- Most error prone routines were larger than 500 lines of code
  - Jones 1986
- Issues like cohesion, complexity more important than size, but probably no more than 200 lines

Parameters

- Put parameters in input-modify-output order
  - Instead of random or alphabetical
  - Use in/out keywords if language supports it
  - Can define yourself in some languages

```
#define IN
#define OUT

void Invertmatrix(
        IN StringCase desiredCase,
        IN OUT Sentence *sentenceToEdit)
);
```

Disadvantages?

Parameters

- Use all the parameters
- Put status or error variables last
  - Common convention, incidental to main purpose of the routine and output only
- Don't use parameters as working variables

```
int  Sample(int inputVal) {
        inputVal += CurrentFactor( inputVal);
        inputVal *=  Multiplier( inputVal);
        …
        return inputVal;
}
```

Can use **const** keyword in C++

Parameters

- Document interface assumptions about parameters
  - Units, input only, range of expected values, etc.
- Limit the number of a routine's parameters to about seven
  - Magic number for people's comprehension
  - Could use composite data type to pass more

Pass Objects or Primitives?

- Given a class Foo:
  - Exposes data through ten accessors
  - GetA(), GetB(), GetC() … GetJ()
    - Yes, bad names, but only for purposes of the exercise ☺
- If you are writing a method "FooCalc" in another object that requires access to A, B, and C from Foo, how would you design this method?
  - FooCalc(ValA, ValB, ValC) ?
  - FooCalc(Foo obj) ?

Functions vs. Procedures

- A common practice is to have a function operate as a procedure and return a status value

  If (FormatOutput(data) == SUCCESS) then…
- Alternative

  FormatOutput(data, outputStatus)

  If (outputStatus == SUCCESS) then…
- Which is better?

Defensive Programming

- What is it?

Exercise

- How would you handle the error scenario where too many items were added to the array?

```
public void AddItem(int num, String name)
{
        // Say that data Array's size is 100



        dataArray[numItems++] = new Item(num, name);
}
```

Defensive Programming

- Protect yourself from the cold cruel world of
  - Invalid data passed to routines
  - Events that can “never” happen
  - Bad code written by some other programmer
- Invalid Input – Handling Garbage In
  - Check the values of all data from external sources
    - Numeric values within tolerance
    - Strings short enough to handle
    - Strings are valid (e.g. injected SQL)
  - Check the values of all routine input parameters
  - Decide how to handle bad inputs

Assertions

- Assertions are small lines of code that can be used to check if everything is operating as expected
  - otherwise an error results and the program terminates
  - For errors that should never occur in the code
- Assertion takes an input that’s supposed to be true, and a message to display if it isn’t
  - assert (denominator != 0) : “Denominator not zero”
- Use to document assumptions made in code and flush out error conditions

Sample Assert Usage Scenarios

- Input value of a parameter in proper range
- File or stream open/closed
- Value of an input only variable not changed
- Pointer is not null
- Array contains expected number of elements
- Container is empty/full
- Verify preconditions and postconditions
- Results from an optimized routine match the slower but clearly written routine
- For real-world programs, both assertions and error-handling code might be used to address the same error

Assert Example (C++)

```
// Approximates the square root of n using Newton's Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
        double answer = 1;
        int i = 0;
        assert((n > 0) && (num_iterations > 0));
        while (i < num_iterations)
        {
                answer = 0.5 * (answer + n / answer);
                i++;
        }
        return answer;
}
```

Handling Expected Errors

- Return a neutral value
  - Might continue operating but return a neutral value known to be harmless
    - Empty string, 0, etc.
- Substitute next piece of valid data
  - Typically when processing a stream, e.g. reading from a file or sampling data
- Return the same answer as the previous time
  - E.g. temperature reading software sampling
- Substitute the closest legal value

Handling Expected Errors

- Log a warning message to a file
- Return an error code
- Set a status variable
- Problems associated with these?

- Call a centralized error processing routine
  - Hard to reuse though in other programs
- Display an error message
- Shut down

Throwing Exceptions

- throw
  - Error happened, someone else deal with it
- try-catch-finally
  - Main problem is exceptions may not be caught properly, or just everything caught

    catch (Exception e)
  - Behavior varies among languages if not caught (nothing or terminates)

Exceptions

- Use exceptions to notify other parts of the program about errors that should not be ignored
  - Strength of exceptions is they are un-ignorable; if you want the possibility to ignore then use status codes
- Throw an exception only for conditions that are truly exceptional
  - Tradeoff for complexity and exception strength; exceptions weaken encapsulation by require calling code to know which exceptions might be thrown
- Don’t use an exception to pass the buck
  - If can handle it locally, do it

Exceptions

- Avoid throwing exceptions in constructors and destructors
  - Usually not caught anywhere
- Throw exceptions at the right level of abstraction
  - Routine should present a consistent abstraction in its interface and so should a class

```
class Employee {
        …
        public TaxId GetTaxId() throws EOFException {
        …
        }
}
```

Exceptions

- Avoid empty catch blocks

```
try {
        …
} catch (Exception e) {
}
```

What does this say about the code throwing the exception?

- Better:

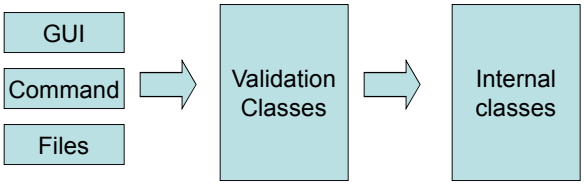
```
try {
        …
} catch (Exception e) {
    LogError("Unexpected exception " + e.toString() );
}
```

Consider alternatives to exceptions

- Some programmers just use exceptions because that is what the language supports for errors
- Always consider the full set of error handling alternatives
  - Handling the error locally
  - Propagation the error with an error code
  - Logging debug information to a file
  - Shutdown
  - Ignore
  - Sometimes the best response to a serious run-time error is to release all acquired resources and abort. Let the user rerun the program with proper input.
    - Bjarne Stroustrup

Barricades

- Damage containment strategy
  - E.g. double-hulled oil tanker
- Can use idea similar to a firewall
  - Designate certain interfaces to boundaries as “safe” areas
  - Check data crossing the boundaries for validity and respond sensibly if the data isn’t valid



Debugging Aids

- Common assumption
  - Developer version can be slow, but production version must be fast, stingy with resources
  - Assumption not always true
- Microsoft Word
  - Code in the idle loop that checks the integrity of the Document object
  - Helps detect data corruption more quickly and easier error diagnosis / recovery
- Consider if your production application really needs the extra speed, or if there is going to be much speed in removing error condition checks

Offensive Programming

- Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running
- Make sure asserts abort the program
  - Don't allow programmers to get in the habit of hitting enter to bypass known problems, make it painful so it will get fixed
- Completely fill any memory allocated so you can detect memory allocation errors
- Completely fill any files or streams allocated to flush out file format errors
- Be sure the code in each case statement's default clause fails hard (aborts) or is impossible to overlook
- Fill an object with junk before it is deleted
- Set up the program to email error log files to yourself so you can see the errors occurring in the released software

Plan to Remove Debugging Aids

- If you need to remove debugging code from the production code, plan for it from the beginning
- Use constants or preprocessor as a debug flag

```
#define DEBUG
#if defined(DEBUG)
    // debugging code
```

Remove Debugging

- Use debugging stubs
  - If you call a debugging subroutine, you can replace the complicated routine with a stub for the production version
  - Incurs small performance penalty, but debug code still available if needed

```
CheckPointer( pointer );

void CheckPointer(void *pointer) {
   // No code, just return
}
```

How Much to Leave?

- How much defensive code should be left in the production version?
  - Leave in code that checks for important errors
  - Remove code that checks for trivial errors
  - Remove code that results in hard crashes
    - Replace with graceful crash
  - Leave in code that helps the program crash gracefully
  - Log errors
  - Make sure error messages left in are friendly

Social Defensive Programming

- Not seen in public API's but can be used internally
- Idea: Documentation might be ignored, but invoking the function name or variable can't
  - ReferenceType MyClass::GetPointerDoNotDelete()
    - Tells the user not to delete this object because it is deleted elsewhere
  - SafeHandle.DangerousGetHandle()
    - Actual example from the CLR where the developers thought it important enough to call out a function that can be dangerous if misused (and hopefully make the users read the docs to figure out why it's dangerous)
  - m_dontUseMe
    - A class member was often misused as an ID (because it seemed to be unique while it wasn't really) and that caused endless problems. At the end, the mere name change to m_dontUseMe reduced the misuse considerably…

Easy Quiz

- An _____ statement is used to check for errors that should not normally occur
- A _____ is used as a buffer area between internal classes that do real work and the external API exposed to users
- A good order for the parameter types of (output / input / modify) is
  - First _____
  - Second _____
  - Third _____