Software Implementation

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr,
  EMP_DATA empRec, float & estimRevenue, float ytdRevenue,
  int screenX, int screenY, COLOR_TYPE & newColor,
  COLOR_TYPE & prevColor, STATUS_TYPE & status,
  int expenseType )
{
for ( int i = 1; i <= 100; ++i ) {
  inputRec.revenue[ i ] = 0;
  inputRec.expense[ i ] = corpExpense[ crntQtr, i ];
  }
UpdateCorpDatabase( EmpRec );
estimRevenue = ytdRevenue * 4.0 / (float)crntQtr ;
newColor = prevColor;
status = Success;
if ( expenseType == 1 ) {
                        for ( int i = 1; i <= 12; ++i )
                        profit[ i ] = revenue[ i ] - expense.type1[ i ];
                        }
else if ( expenseType == 2 )  {
                              profit[ i ] = revenue[ i ] - expense.type2[ i ];
                              }
else if ( expenseType == 3 )
  {
  profit[ i ] = revenue[ i ] - expense.type3[ i ];
  }
}
```

Reasons for Creating Routines

- One of our primary tools in writing “good code” is knowing why and when to create new subroutines

- Classes are abstractions that represent the “things” in a system
- Subroutines (or methods) are abstractions that represent the “algorithms” in a system

- There are many reasons to create routines; we will focus on a few:
  - Top-down decomposition of algorithms
  - Avoiding code duplication
  - Avoiding deep nesting
  - Recursion

Strong Cohesion

- Just like classes, routines should be highly cohesive
- A cohesive routine does one and only one thing, and has a name that effectively describes what it does
  - GetCustomerName, EraseFile, CalculateLoanPayment
- Routines that do too much become obvious if we name them properly
  - DoDishesAndWashClothesAndSweepFloor

Algorithm Decomposition

- Long routines become hard to read and understand
- Long routines should be divided into subparts, with each subpart being implemented as a separate, well-named subroutine
- The original routine becomes a "driver" that calls the subroutines (it also becomes much shorter)
- Decomposition continues until subroutines are too simple to subdivide further
- In OO design, an “algorithm” is often distributed across multiple subroutines on multiple classes

Comments on comments

- If you feel a need to comment a section of code, consider putting that section of code in a routine of its own with a descriptive name
- This will frequently do away with the need for the comment, and often results in highly-readable code
- If a routine is heavily commented, it might be because it hasn't been decomposed far enough

Avoiding code duplication

- Avoiding code duplication is one of the most important principles of software design
- Duplicated code makes software maintenance difficult and error-prone
- If the same code is needed in multiple places, put the code in a routine that can be called wherever the code is needed

Avoiding code duplication

- This will simplify software maintenance by centralizing the code; it will also encourage software reuse
- Inheritance in OO languages is another tool for achieving code reuse (inherit shared code from a superclass)
- If your design requires code duplication, your design needs to be fixed

Good Routine Names

- A routine name should clearly and completely describe what the routine does
  - PrintReportAndInitPrinter rather than just PrintReport
- Procedure names (no return value)
  - Verb or verb phrase
- Function names (has return value)
  - A function name should describe the function's return value
  - IsPrinterReady, CurrentPenColor, NextCustomerId

Good Routine Names

- Avoid meaningless verbs
  - HandleCalculation, PerformServices, DealWithInput
  - If a routine is hard to name, it is probably not well designed because it is hard to describe what it does
- Make routine names long enough to be easily understandable (don’t abbreviate too much)
- Establish conventions for naming routines
  - Boolean functions - IsReady, IsLeapYear, ...
  - Getters/setters - GetName, SetName, ...
  - Initialization - Init/Deinit, Setup/Cleanup, ...

Parameters

- Use all of the parameters
- The more parameters a routine has, the harder it is to understand
- The fewer parameters the better
- A common rule-of-thumb is that you should limit parameters to no more than 7, and even that many should be rare

Guidelines for initializing data

- Improper data initialization is one of the most fertile sources of error in computer programming
- Initialize variables when they're declared
- Declare variables close to where they're used
  - Variables don't have to be declared at the top of the method
- Check for the need to reinitialize a variable
  - Counters, accumulators, etc.
- Compiler warnings can help find un-initialized variables (-Wall)

Code Layout

- The physical layout of the code strongly affects readability
- Good visual layout makes the logical structure of a program clear to the reader
- Good layout helps avoid introducing bugs when the code is modified
- Pick a style that you like, and consistently use it
- Layout Examples
  - Example 1
  - Example 2

Whitespace

- Use whitespace to enhance readability
  - Spaces, tabs, line breaks, blank lines
- Organize routines into "paragraphs"
  - Paragraph = a group of closely related statements
  - Separate paragraphs with one or more blank lines
- Indentation
  - Use indentation to show the logical structure (i.e., nesting)
- Align elements that belong together
  - Sequence of variable declarations (align names)
  - Sequence of assignments (align ='s)
  - Wrapped subroutine calls (indent wrapped parameters)

Expressions

- Over-parenthesize arithmetic expressions
  - Enhance readability
  - Make clear the order of operator evaluation
  - Don't assume that everyone has the operator precedence table memorized
- Insert extra spaces between operands, operators, and parentheses to enhance readability

```
while (((startPath+pos)<=length(pathName))&&
        pathName[startPath+pos]!=';') {
   …
}

while ( ( (startPath + pos) <= length( pathName ) ) &&
        pathName[startPath + pos] != ';' ) {
   …
}
```

Expressions

- Put separate conditions on separate lines

```
If (('0' <= inChar && inChar <= '9') || ('a' <= inChar &&
   inChar <= 'z') || ('A' <= inChar && inChar <= 'Z')) {
   …
}
```

```
If (('0' <= inChar && inChar <= '9') ||
    ('a' <= inChar && inChar <= 'z') ||
    ('A' <= inChar && inChar <= 'Z')) {
   …
}
```

Expressions

- Put expressions, or pieces of them, in well-named subroutines

```
If (IsDigit(inChar) || IsLowerAlpha(inChar) || IsUpperAlpha(inChar)) {
   …
}
```

- Or, even better

```
If (IsAlphaNumeric(inChar)) {
   …
}

Bool IsAlphaNumeric(char c) {
   return (IsDigit(c) || IsLowerAlpha(c) || IsUpperAlpha(c));
}
```

Placing curly braces

```
for (int i=0; i < MAX; ++i) {
    values[i] = 0;
}

for (int i=0; i < MAX; ++i)
{
    values[i] = 0;
}

for (int i=0; i < MAX; ++i)
{   values[i] = 0;
}

for (int i=0; i < MAX; ++i)
    {
    values[i] = 0;
    }
```

Placing curly braces

- What about this?

```
for (int i=0; i < MAX; ++i)
    values[i] = 0;
```

Routine parameters

- Use spaces to make routine parameters readable

```
webCrawler->Crawl(rootURL,outputDir,stopWordsFile);

webCrawler->Crawl( rootURL, outputDir, stopWordsFile );

webCrawler->Crawl(rootURL, outputDir, stopWordsFile);
```

One statement per line

- Declare each variable on a separate line
  - More robust under modification
  - Easier to understand

```
int * p, q;  // oops!
```

```
int * p;  // correct
int * q;
```

- Don't put multiple statements on the same line

```
x = 0; y = 0;
```

```
x = 0;
y = 0;
```

Wrapping long lines

- When should you wrap long lines?
  - When they won't fit on the screen?
  - When they won't fit on a printed page?
- Wrapping between 80 and 90 characters is common
  - Lines longer than that are hard to read
  - It discourages deep nesting
- Align continuation lines in a way that maximizes readability

Dangerously deep nesting

- Excessive nesting of statements is one of the chief culprits of confusing code
- You should avoid nesting more than three or four levels
- Creating additional subroutines is the best way to remove deep nesting

Pseudo-Code

- When writing an algorithmically complex routine, write an outline of the routine before starting to code
- Use English-like statements to describe the steps in the algorithm
- Avoid syntactic elements from the target programming language
  - Design at a higher level than the code itself
- Write pseudo-code at the level of intent
  - *What* more than *how* (the code will show how)
- Write pseudo-code at a low enough level that generating code from it is straightforward
  - If pseudo-code is too high-level, it will gloss over important details

Example of bad pseudo-code

```
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 operating system
*hRsrcPtr = resource number
return 0
```

- Intent is hard to understand
- Focuses on implementation rather than intent
- Includes too many coding details
- Might as well just write the code

Example of good pseudo-code

```
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
```

- Written entirely in English
- Not programming language specific
- Written at level of intent
- Low-level enough to write code from

Choose Good Variable Names

- Too Long
  - NumberOfPeopleOnTheUSOlympicTeam
- Too Short
  - N
- Just Right
  - NumTeamMembers
- Are short variable names always bad? NO
  - Loop control variables: i, j, k, idx
  - Temporary variables: tmp
  - Names that are naturally short: x, y, z

C++ naming conventions

- Separating words in identifiers
  - "Camel-case"
    - WebCrawler, documentMap
  - Separate words with underscores
    - Web_crawler, document_map
- First char of class name is usually upper-case
- First char of method name can be either upper or lower case, but be consistent
- First char of variable name is usually lower-case
- Constant names are usually all upper-case

Other useful naming conventions

- Distinguish global, object, local, and parameter variables
  - g_GlobalVariable
  - m_MemberVariable
  - localVariable
  - _parameter

Creating readable names

- Names matter more to readers of the code than to the author of the code
- Don't use names that are totally unrelated to what the variables represent (e.g., “Thingy”)
- Don't differentiate variable names solely by capitalization
  - int temp;
  - char Temp;

Creating readable names

- Avoid words that are commonly misspelled
- Avoid characters that are hard to distinguish (1 and l)
- Avoid using digits in names (e.g., File1 and File2)
- Avoid variables with similar names but different meanings
  - int temp;
  - Mountain timp;

Abbreviation guidelines

- Only abbreviate when you have to
- Remove non-leading vowels (Computer -> Cmptr)
- Or, Use the first few letters of a word (Calculate -> Calc)
- Don't abbreviate by removing just one character from a word (use "name" instead of "nam")
- Create names that you can pronounce
- Abbreviate consistently