Quiz – Creating Effective Data
Requires Knowing Data Types

Score 1 if you know the meaning, 0.5 if you kind of know the term, 0 for don’t know

- Abstract data type
- Array
- Bitmap
- Boolean
- B-tree
- Character variable
- Container class
- Double precision
- Elongated stream
- Enumerated type
- Floating point
- Heap
- Index
- Integer
- Linked list
- Named constant
- Literal
- Local variable
- Lookup table
- Member data
- Pointer
- Private
- Retroactive synapse
- Referential integrity
- Stack
- String
- Structured variable
- Tree
- Typedef
- Union
- Value chain
- Variant
Scoring

- 0-14
  - Beginning programmer, you should not be in this class
- 15-19
  - Intermediate programmer, or you forgot a lot
- 20-24
  - Expert programmer
- 25-29
  - Guru programmer
- 30-32
  - Pompous fraud
  - “Elongated Stream”, “Retroactive Synapse” and “Value Chain” are made up

Tips for Variable Declarations

- Use a template for variable declarations
  - When you need to declare new variables, pull the template into your file and edit it
    ```
    public * * // Comments
    private * * // Comments
    ```
  - Easier if you use a command-line style editor, then just a few keystrokes
  - Select line most similar to what you want and delete the rest
  - * guarantee a syntax error in case you forget to change the declaration
  - Empty comment reminds you to comment the variable as you declare it
- Chris' template
  ```
  * * /* Chris is a jerk! */
  ```
Implicit Declarations

• When you use a variable without declaring it
  – Allowed in some languages, e.g. Visual Basic, Fortran, PHP
• Considered a hazardous feature
  – Checking variable intCtr, but it has the wrong value, always seems to be zero?
  – Meant to use ctrInt variable instead
• Turn off implicit declarations if possible
  – Option explicit in VB
  – No option in PHP but can use error_reporting(E_ALL); to give warnings when an undefined variable is referenced

Initializing Data

• Improper data initialization another fertile source of errors
  – Good idea to always initialize variables
  – Always create default constructors that initialize class level variables
• Uninitialized variables
  – May be assigned a value or some languages will use whatever is in memory
  – Value is outdated, was valid at one point but is no longer valid
  – Part of the variable assigned a value and part not, e.g. an array of objects allocated, but each object hasn’t been created via new
• Initialize each variable as it’s declared
Initializing Data

- Check input parameters for validity
  - Before you assign input values to anything, make sure the values are reasonable
  - Applies to input from the user, or input within a method
- Initialize each variable close to where its first used
  - Some programmers do the following:
    ```
    ' Initialize all variables
    Idx = 0
    Total = 0
    Done = False
    ...
    ' Lots of code using Idx and Total
    ' Code using Done
    while not Done
    ...
    ```

Variable Initialization

- Better: Initialize variables closer to where they are used
  ```
  Idx = 0
  ' Code using Idx
  ...
  Total = 0
  ' Code using total
  ...
  Done = False
  ' Code using Done
  ```

- Why is this better?
  - Easier to reference variable in the code
  - Decreases chance of overwriting variable values later on as code is modified
Scope

- Code between references to a variable is a “window of vulnerability”
  - New code might be added or called that mucks up a variable’s value

```c
b = b / c;
span for b: (1 + 1)/ 2  = 1
b = a + 1;
```

Better?

```c
b = b / c;
span for c:  3
b = a + 1;
```

Live Time of a Variable

Excessively Long Live Times

```c
recordIndex = 0;
total = 0;
done = false;
...
while (recordIndex < recordCount) {
    recordIndex++; // Last reference to recordIndex
    ...
    if (total > projectedTotal) {
        done = true; // Last reference to done
    }
}```
Live Time of a Variable
Shorten Life Spans

25 recordIndex = 0;
26 while (recordIndex < recordCount) {
27    recordIndex++;
        // Last reference to recordIndex
...
68 total = 0;
69 done = false;
70 if (total > projectedTotal) {
71    done = true;
        // Last reference to total
        // Last reference to done

Variables to Watch

• Proximity Principle
  – Keep related actions together
  – Also applies to comments, loop setup, etc.
• Pay attention to counters and accumulators
  – i,j,k,sum, commonly not reset the next time used
• Initialize each variable as it’s declared
  – Not a substitute for initializing close to where they’re used, but a
good form of defensive programming
• Look at the compiler’s warning messages
• Use memory access tools to check for bad pointers or
memory leaks
  – 0xCC used to initialize in the debugger, makes it easier to find
access to uninitialized memory
Naming Variables

• Examples of poor variable names
  – \( X = X - XX; \)
  – \( XXX = XXX - \text{LateFee}(X1,X); \)
  – \( \text{Tom} = \text{Dick} + \text{Harry}; \)
• Examples of good variable names
  – \( \text{Balance} = \text{Balance} - \text{LastPayment}; \)
  – \( \text{Balance} = \text{Balance} - \text{LateFee}(<\text{CustomerID}, \text{Payment}>) ; \)
  – \( \text{MonthlyTotal} = \text{NewPurchases} + \text{SalesTax}; \)

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Good Names</th>
<th>Bad Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running total of checks written</td>
<td>RunningTotal, CheckTotal, nChecks</td>
<td>Written, CT, X</td>
</tr>
<tr>
<td>Velocity of a train</td>
<td>Velocity, TrainVelocity, VelocityMPH</td>
<td>V, Velt, Train, TV</td>
</tr>
<tr>
<td>Current Date</td>
<td>CurrentDate, CrtDate</td>
<td>CD, current, Date</td>
</tr>
<tr>
<td>Lines per page</td>
<td>LinesPerPage</td>
<td>LPP, Lines, L</td>
</tr>
</tbody>
</table>
Optimum Name Length?

• 1990 Study of COBOL programs
  – Effort required to debug was minimized when variables had names that averaged 10 to 16 characters
  – Names averaging 8-20 almost as easy to debug
  – Strive for medium-length variable names, definitely try to avoid too short variable names

• Short variable names not all bad
  – i,j, etc. good for loops, scratch values with limited scope
  – Longer names better for rarely used or variables with wide scope, variables used outside the loop
  – Shorter names better for local or loop variables

Looping

• Examples:

```plaintext
RecordCount:=0;
while (moreScores()) do
{
    RecordCount++;  
    Score[RecordCount] = getNextScore();
}
```

... Code that uses RecordCount

Nested Loop:
```plaintext
for TeamIndex:=1 to TeamCount do begin
    for EventIndex :=1 to EventCount[TeamIndex] do
        Score[TeamIndex,EventIndex]:=0;
end;
```

Common to confuse i,j if use as nested loop names
Qualifiers in Variable Names

- Many programs have variables with computed values
  - Total, average, maximum, etc.
- Modify name with qualifier
  - RevenueTtl, ScoreMax, etc.
  - Be consistent – put at beginning or end
    - Book recommends end
    - Also use opposites precisely
      - Add/Remove
      - Get/Set?
      - Get/Put?
    - Special case for num
      - numSales refers to total number of sales
      - SalesNum refers to the number of the sale
      - Use count or total if applicable

Naming Status Variables

- Use a better name than “flag”
  - Doesn’t say what the flag does
  - E.g. Flag, StatusFlag, PrintFlag, …
- Better names
  - DataReady, ReportType, CharacterType, RecalcNeeded
- Give boolean variable names that imply true or false
  - Bad booleans: Status, b
  - Good booleans: Done, Success, Ready, Found
  - Use positive names
    - If not NotFound …
Naming Conventions

• Some programmers resist conventions
  – Rigid and ineffective?
  – Destructive to creativity?
• But many benefits
  – Help you learn code more quickly on a new project rather than learning idiosyncrasies of other programmers
  – Reduce name proliferation, e.g. PointTtl and TtlPoints
  – Compensate for language weaknesses
    • E.g. emulate constants, enumerated types
  – Can emphasize relationships among related items
    • E.g. EmpAddr, EmpPhone, EmpName
  – Any convention is better than no convention!

When to have Naming Conventions

• Multiple programmers working on a project
• Plan to turn a program over to another programmer for modification or maintenance
• Program will be reviewed by others
• Program is so large you must think about it in pieces
• A lot of unusual terms that are common and you want to have standard terms or abbreviations in coding
Informal Naming Conventions

- Guidelines for a language-independent convention
  - Identifyglobals
    - e.g. g_OverallTotal
  - Identify module or class variables
    - e.g. m_Name;
    - VB.NET: For class variables, use Me.varName
      - E.g. this.varName
  - Identify type definitions
    - e.g. int_Count;
  - Identify Named Constants
    - e.g. all UPPERCASE
  - Identify in/out parameters
    - e.g. in_Name, out_Price

Typical prefixes for C

- char - c,ch
- Integer indices – i,j
- Number – n
- Pointer – p
- String – s
- Variables and routines in all_lower_case
- Constants in ALL_CAPS
- Underscore to separate; e.g.
  - first_name instead of firstname

- Example: char *ps_first_name;
Hungarian Naming Convention

• Formal notation widely used in C and with Windows programming
  – Names look like words in a foreign language
  – Charles Simonyi, who is Hungarian

• Three parts
  – Base Type
  – One or more prefixes
  – Qualifier

Hungarian Base Types

• Base Type specifies the data type of the variable being named
• Generally doesn’t refer to any predefined data types, only abstract types
• Example:
  – wn = Window
  – scr = Screen
  – fon = Font
  – pa = Paragraph
• Example:
  – WN wnMain=NULL;
  – FONT fonUserSelected = TIMES_NEW_ROMAN;
Prefixes

• Prefixes go in front of the base type and describe how the variable will be used
• Somewhat standard list:
  – a = Array
  – c = Count
  – d = Difference
  – e = Element of an array
  – g = Global variable
  – h = Handle
  – i = Index to array
  – m = Module-level variable
  – p(np, lp) = Pointer (near or long)
• Examples
  – Array of windows: awnDialogs
  – Handle to a window: hwnMyWindow
  – Number of fonts: cfon

Qualifiers

• The rest of the descriptive part of the name that would make up the variable if you weren’t using Hungarian
• Some standard qualifiers
  – Min = First element in an array or list
  – First = First element to process in an array
    • Similar to Min but relative to current operation rather than the array itself
  – Last = Last element to deal with in an array
  – Lim = Upper limit of elements to deal with in the array
  – Max = Last element in an array or other kind of list
Hungarian Examples

- achDelete
  - An array of characters to delete
- iach
  - Index to an array of characters
- ppach
  - Pointer to a pointer of an array of characters
- mhscrUserInput
  - Module-level handle to a screen region for user input
- gpachInsert
  - Global pointer to an array of characters to insert

Hungarian Advantages

- Standard naming convention
- Broad enough to use in multiple languages
- Adds precision to some areas of naming that are imprecise, e.g. Min/First
- Allows you to check abstract types before compiling
- Helps document types in weakly-typed languages
- Names can become compact
Hungarian Disadvantages

• Variable names not readable unless familiar with the notation
• Combines data meaning with data representation
  – If you later change something from an integer to a long, you might have to change the variable name as well
• “Abuse” of format - encourages some lazy variable names
  – Very common in windows: hwnd
  – We know it is a handle to a window, but is it a menu, dialog box, or ? Qualifiers often left off

Creating Readable Variables

• To create short names that are readable, here are some general guidelines
  – Remove nonleading vowels
    • Computer to cmptr
  – Use first letter or truncate after 1-3 letters
  – Remove useless suffixes –ing, ed, etc.
  – Keep the first and last letters of each word
  – Keep the most noticeable sound in each syllable
Variable Don'ts

• Don’t
  – Remove one character from a word, doesn’t justify the loss
  – Create unpronounceable names
    • XPos rather than XPstn
  – Use names with similar meanings
    • RecNum, NumRecs as two separate variables
  – Use similar names with different meanings
    • NumRecs, NumReps as very different values
  – Use numbers
    • Total1, Total2
  – Use misspelled names
    • hilight
  – Differentiate solely by capitalization
  – Use unrelated names
  – Use hard-to-read characters
    • e1ite, elite

Using Variables

• Coming up with a name is just the first step…
• Some guidelines for using variables
  – Minimize scope
  – Keep references together
    • If order doesn’t matter, keep references to the same variable in the same place instead of scattered throughout
  – Use a variable for one purpose only
  – Avoid global variables
    • Side-effects, Alias problems
Numbers in General

• Avoid magic numbers
  – Use constants instead
    • Easier to change
    • Code more readable
    • Helps describe history of the number
  – Magic numbers in contexts like 0xCAFEBABE ok
  – OK to hard-code 0’s and 1’s
• Don’t rely on implicit type conversions
  – Source of many errors
• Avoid mixed-type comparisons
  – If (i==x) where i=int, x=double

Numbers

• Check integer division
  – 7/10 = 0
• Check for integer overflow
  – 65535 for 16 bits
  – Consider overflow in intermediate calculations as well, not just
    the final calculation
• Avoid addition/subtraction of numbers with vastly
different magnitudes
  – 5,000,000.02 – 5,000,000.01 = 0 if not enough bits for precision
  – Process smallest numbers first, work way up to larger ones
• Avoid equality of floating point types
Strings

- Avoid magic characters
  - “empty”
  - “%@$”
  - Special characters to overload meaning
    - E.g. array of names, but in some cases want to associate a phone number, so use “^Name^Number”

- Arrays in C
  - Initialize strings to null
  - Use strncpy() instead of strcpy()
Arrays

• Make sure the array indexes are within the bounds
  – Check the end points of arrays
  – Can sometimes help to use arrays as sequential structures if
    doesn’t impact performance
• Multidimensional arrays
  – Make sure subscripts are used in correct order, e.g. Array[i][j]
    when mean Array[j][i]
• Nested loops
  – Watch for index cross talk, Array[i] when mean Array[j]
• Throw in an extra element at the end of the array
  – Common to be off by one at the end
  – Gives yourself a cushion
  – But doing this is pretty sloppy, consider what you are saying
    about yourself if you do this! But choose lesser of two evils

References and Pointers

• Address of an object or data in memory
• General tips
  – Isolate pointer operations in routines instead of
    scattering throughout the code
  – Check pointers before using them
    • Ensure contents are valid
    • E.g. if (ptr != NULL) { … }
  – Simplify complicated pointer expressions

Net[i] = Base[i]*Rates->Discounts->Factors->Net;

QuantityDiscount = Rates->Discounts->Factors->Net;
Net[i] = Base[i] * QuantityDiscount;
Organizing Straight-Line Code

• Pay attention to order in straight-line code
• Make it obvious If there are order dependencies
  
  ComputeMarketingExpenses();
  ComputeMISExpenses();
  ComputeAccountingExpenses();

  – If these methods rely on global data, there is a hidden dependency
  – Use parameters to make dependencies more clear, along with documentation
    ComputeMarketingExpenses(&ExpenseData);
    ComputeMISExpenses(&ExpenseData); // After Marketing
    ComputeAccountingExpenses(&ExpenseData); // After MIS

Order Doesn’t Matter?

• In some cases order doesn’t matter. Can you then put statements in any way you like?

  InitMarketing(MarketingData);
  InitMIS(MISData);
  InitAccounting(AccountingData);

  ComputeAccounting(AccountingData);
  ComputeMIS(MISData);
  ComputeMarketing(MarketingData);

  PrintMIS(MISData);
  PrintAccounting(AccountingData);
  PrintMarketing(MarketingData);
Group Related Code

• Localizes references to each variable, values used closer to when assigned

    InitMIS(MISData);
    ComputeMIS(MISData);
    PrintMIS(MISData);

    InitAccounting(AccountingData);
    ComputeAccounting(AccountingData);
    PrintAccounting(AccountingData);

    InitMarketing(MarketingData);
    ComputeMarketing(MarketingData);
    PrintMarketing(MarketingData);

Conditionals

• If-statements
  – Make sure that you branch correctly on equality
    • >, >=
  – Put the normal case after the if rather than after the else
    If (SomeTest)  if (!SomeTest) {
      { }              // lots of code here
    }                 
    else {            
      // lots of code here
    }
  – Write nominal path through the code first, then the exception
Nominal Case Mixed with Error Cases

OpenFile(Input, Status)
if Status = Error then
    ErrorType = FileOpenError
else
    ReadFile(InputFile, FileData, Status)
    if Status = Success then
        SummarizeFileData(FileData, SummaryData, Status)
        if Status = Error then
            ErrorType = DataSummaryError
        else
            PrintSummary(SummaryData)
            SaveSummaryData(SummaryData, Status)
            if Status = Error then
                ErrorType = SummarySaveError
            else
                UpdateAllAccounts
                ErrorType = None
            end if
        end if
    else
        ErrorType = FileReadError
    end if
end if

Process Nominal Case First

OpenFile(Input, Status)
if Status <> Error then
    ReadFile(InputFile, FileData, Status)
    if Status = Success then
        SummarizeFileData(FileData, SummaryData, Status)
        if Status <> Error then
            PrintSummary(SummaryData)
            SaveSummaryData(SummaryData, Status)
            if Status <> Error then
                UpdateAllAccounts
                ErrorType = None
            else
                ErrorType = SummarySaveError
            end if
        else
            ErrorType = DataSaveError
        end if
    else
        ErrorType = FileSaveError
    end if
else
    ErrorType = FileOpenError
end if
Consider the Else

- If just use a plain if, consider if you need an else
- GM study: only 17% of if statements had an else, but further analysis showed 50-80% should have had one
  - Useful to include to make sure all cases are covered
- One option - code the else clause with a null statement if necessary to show that the else case has been considered

Case Statements

- Order cases by
  - Alphabetical or Numerical order
  - Normal case first, decreasing frequency
- Don’t make up phony variables to use a case statement

```java
char action = command[0];  // Command is a string
switch (action) {
  case 'c': copy();
            break;
  case 'd': delete();
            break;
  case 'h': help();
            break;
  default:  PrintErrorMessage();
}
```
Better Practice

• May have problem with mapping to the phony variable
  – E.g. add a “Clear” command, both start with c
• Use if-then-else with actual values

```c
if (!strcmp(command,"copy"))
    copy();
else if (!strcmp(command,"delete"))
    delete();
else …
```

Case Statements

• Use the default clause only to detect legitimate defaults
  – If there is only one case left, you might decide to use that case as the default
  – But loses documentation provided by case labels and breaks down under modification
• Use the default clause to detect errors
• Don’t forget the break statement if needed
Loops

• Use appropriate type
  – while to test at beginning
  – Do-while to test at end
  – For generally for counting
    • Sometimes preferred over while since all loop control code is in one place
    • But don’t abuse the for loop

```c
for (rewind(inFile), recCount = 0; !feof(inFile); recCount++)
    { fgets(InputRec[recCount], MAX_CHARS, inFile) }
```

What is wrong with the above?
```
for (rewind(inFile), recCount = 0; !feof(inFile);
    fgets(InputRec[recCount], MAX_CHARS, inFile))
    { recCount++; }
```

Loop Conditions

• Make the loop condition clear
• Avoid too much processing in the loop boolean condition
```
while (fgets(InputRec[recCount++], MAX_CHARS, inFile)!=NULL)
    { }
```

• If the body is empty, the loop is probably poorly constructed
```
while (!feof(inFile))
    { fgets(InputRec[recCount], MAX_CHARS, inFile);
        recCount++; }
```
Loop Behavior

• Keep housekeeping chores at the beginning or the end of the loop
  – e.g. i=i+1

• Make each loop perform only one function

• Make loop termination conditions obvious
  – Don’t fool around with goto’s or break’s or continue’s if possible
  – Don’t monkey around with the loop index
    for (i=0; i<100; i++) {
      // Code here
      if (SomeCondition) i=100;
    }

Loop Behavior

• Avoid code that depends on the loop index’s final value
  – Instead copy to another variable

for (i=0; i < MaxRecords; i++)
{
  if (entry[i] == target) break;
}
...
if (i<MaxRecords)
entry[i] = newValue;

for (i=0, index= -1; i < MaxRecords; i++)
{
  if (entry[i] == target)
  {
    index = i;
    break;
  }
}
...
If (index != -1)
entry[i] = newValue;
Break/Continue

• Use break and continue with caution
• Be wary of a loop with a lot of break’s scattered in it
  – Can indicate unclear thinking about the structure of the loop
• Use break statements rather than boolean flags in a while loop
  – Can remove several layers of indentation by using the break, actually easier to read
• Use continue for tests at the top of a loop
  – Use with caution, but as with break continue can eliminate an extra layer of nesting

Continue Example

```c
finished = false;
while (!finished && !feof(file))
{
    ReadRecord(record, file);
    if (record.type == targetType)
    {
        // Process record
    }
}
```

Use break to eliminate finished boolean

```c
while (!feof(file))
{
    ReadRecord(record, file);
    if (record.type != targetType)
    continue;
    // Process record
}
```
Loops, Continued

• Use meaningful index variable names for nested loops, helps avoid crosstalk, easier to read

```plaintext
for i:=1 to 55 do
  for j:=1 to 12 do
    for k:=1 to num do
      sum:=sum+Transaction[j,i,k];

for PayCodeIdx:=1 to NumPayCodes do
  for Month:=1 to 12 do
    for DivisionIdx:=1 to NumDivisions do
      sum:=sum+
        Transaction[Month, PayCodeIdx, DivisionIdx];
```

Loop Length

• Make your loops short enough to view all at once
  – Helps give context into how the loop operates
  – Usually less than 20 lines
• Limit nesting to three levels
  – Yourdon study in 1986 showed the comprehension of programmers for loop behavior deteriorates significantly beyond three levels
• Make long loops especially clear