CSCI-3081: Program Design and Development

CODING STYLE & SELF-DOCUMENTING CODE

```
x = 3+4 * 2+7;
```

```
/* Use the insertion sort technique to sort the "data" array in ascending order.
This routine assumes that data[ firstElement ] is not the first element in data and
that data[ firstElement-1 ] can be accessed. */ public void InsertionSort( int[]
data, int firstElement, int lastElement ) { /* Replace element at lower boundary
with an element guaranteed to be first in a sorted list. */ int lowerBoundary =
data[ firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /* The elements in
positions firstElement through sortBoundary-1 are always sorted. In each pass
through the loop, sortBoundary is increased, and the element at the position of the
new sortBoundary probably isn't in its sorted place in the array, so it's inserted
into the proper place somewhere between firstElement and sortBoundary. */ for (
int sortBoundary = firstElement+1; sortBoundary <= lastElement; sortBoundary++ )
{ int insertVal = data[ sortBoundary ]; int insertPos = sortBoundary; while (
insertVal < data[ insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1; } data[ insertPos ] = insertVal; } /* Replace original
lower-boundary element */ data[ firstElement-1 ] = lowerBoundary; }
```

```
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed. */
public void InsertionSort( int[] data, int firstElement, int lastElement ) {
/* Replace element at lower boundary with an element guaranteed to be first in a
sorted list. */
int lowerBoundary = data[ firstElement-1 ];
data[ firstElement-1 ] = SORT_MIN;
/* The elements in positions firstElement through sortBoundary-1 are
always sorted. In each pass through the loop, sortBoundary
is increased, and the element at the position of the
new sortBoundary probably isn't in its sorted place in the
array, so it's inserted into the proper place somewhere
between firstElement and sortBoundary. */
for (
int sortBoundary = firstElement+1;
sortBoundary <= lastElement;
sortBoundary++
) {
int insertVal = data[ sortBoundary ];
int insertPos = sortBoundary;
while ( insertVal < data[ insertPos-1 ]) {
data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1;
}
data[ insertPos ] = insertVal;
}
/* Replace original lower-boundary element */
data[ firstElement-1 ] = lowerBoundary;
}
```

```
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed.
*/

public void InsertionSort(int[] data, int firstElement, int lastElement ) {
   // Replace element at lower boundary with an element guaranteed to be
   // first in a sorted list.
   int lowerBoundary = data[ firstElement-1 ];
   data[ firstElement-1 ] = SORT_MIN;

   /* The elements in positions firstElement through sortBoundary-1 are
   always sorted. In each pass through the loop, sortBoundary
   is increased, and the element at the position of the
   new sortBoundary probably isn't in its sorted place in the
   array, so it's inserted into the proper place somewhere
   between firstElement and sortBoundary.
   */
   for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
      sortBoundary++ ) {
      int insertVal = data[ sortBoundary ];
      int insertPos = sortBoundary;
      while ( insertVal < data[ insertPos - 1 ] ) {
         data[ insertPos ] = data[ insertPos - 1 ];
         insertPos = insertPos - 1;
      }
      data[ insertPos ] = insertVal;
   }

   // Replace original lower-boundary element
   data[ firstElement - 1 ] = lowerBoundary;
}
```

Good Code vs. Bad Code

Take in a single (real) value, use it to evaluate two different polynomials (we’ll choose a straight line and a quadratic), decide if the line value is greater than the quadratic value and return TRUE/FALSE accordingly.

Good Code vs. Bad Code

```
FUNCTION comppoly(x)
float y1, y2
float a1=0.1, b1=0.3, a2=2.1, b2=5.3, c=0.22
y1 = a1*x + b1
y2 = a1*x^2 + b2*x + c
return(y2>y1)
END FUNCTION
```

```
FUNCTION ComparePolynomials(x)
//DECLARE VARIABLES, PARAMETERS
float  y_line, y_quadratic
float  lineParam = [0.1, 0.3]
float  quadParam = [2.1, 5.3, 0.22]

//CALCULATE THE LINE AND QUADRATIC VALUES AT X
y_line      = lineParam[0]*x + lineParam[1]
y_quadratic = quadParam[0]*x^2 + quadParam[1]*x + quadParam[2]

//COMPARE THE FUNCTIONS, RETURNING A LOGICAL
return(y_line > y_quadratic)
END FUNCTION
```

Elements of Coding Style

- Naming conventions
- File names and content, directory layout
- Layout of declarations and implementations
- Indentation and placement of braces
- Use of whitespace
- Placement of pointer and reference symbols
- Conventional use of loops and conditionals

General Naming Conventions

- Differentiate between variable names and routine names
  - variableName vs. RoutineName() — not typically in C++
- Differentiate between classes and objects
  - Widget widget;
  - LongerWidget longerWidget;
- Identify global variables
  - g_RunningTotal
- Identify member variables
  - m_ prefix

General Naming Conventions (2)

- Identify type definitions
  - might use ALL_CAPS or a t_ prefix
- Identify named constants
  - LINES_PER_PAGE_MAX
- Identify elements of enumerated types
  - Prefix based on the type, e.g. Color_ or Planet_
- Format names to enhance readability
  - gymnasticsPointTotal rather than GYMNASTICSPOINTTOTAL

C++ Naming Conventions

i and *j* are integer indexes.

p is a pointer.

Constants, typedefs, and preprocessor macros are in *ALL_CAPS*.

Class and other type names are in *MixedUpperAndLowerCase*.

Variable and function names use lowercase for the first word, with the first letter of each following word capitalized—for example, *variableOrRoutineName*.

The underscore is not used as a separator within names, except for names in all caps and certain kinds of prefixes (such as those used to identify global variables).

Listing 31-68: C++ example of good formatting with restraint.

The lightness of this line compared to the line of asterisks visually reinforces the fact that the routine is subordinate to the class.

```cpp
//**********************************************************************
// MATHEMATICAL FUNCTIONS
//
// This class contains the program's mathematical functions.
//**********************************************************************

//----------------------------------------------------------------------
// find the arithmetic maximum of arg1 and arg2
//----------------------------------------------------------------------
int Math::Max( int arg1, int arg2 ) {
   if ( arg1 > arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}

//----------------------------------------------------------------------
// find the arithmetic minimum of arg1 and arg2
//----------------------------------------------------------------------
int Math::Min( int arg1, int arg2 ) {
   if ( arg1 < arg2 ) {
      return arg1;
   }
   else {
      return arg2;
   }
}
```

CSCI-3081: Program Design and Development

SELF-DOCUMENTING CODE

Main Points on Documenting Code

- Best documentation is often in the code itself.
- Taking this to the next level, the best documentation often is the code itself.
  - But, only if you write it in a self-documenting way.
- If you're going to take something away from this class that helps you better program as part of a team, this might be the idea to take with you.
- Why?
- I can't remember the last time I "went back" to write in-line documentation about my code.

Poor Documentation from Bad Programming Style

```
for ( i = 2; i <= num; i++ ) {
meetsCriteria[ i ] = true;
}
for ( i = 2; i <= num / 2; i++ ) {
j = i + i;
while ( j <= num ) {
meetsCriteria[ j ] = false;
j = j + i;
}
}
for ( i = 1; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets criteria." );
}
}
```

Documentation without Comments

```
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
   int factorableNumber = factor + factor;
   while ( factorableNumber <= num ) {
      isPrime[ factorableNumber ] = false;
      factorableNumber = factorableNumber + factor;
   }
}

for ( primeCandidate = 1; primeCandidate <= num; primeCandidate++ ) {
   if ( isPrime[ primeCandidate ] ) {
      System.out.println( primeCandidate + " is prime." );
   }
}
```

McConnell

- Entertaining discussion in McConnell, 32.3 “To Comment or Not to Comment”

"Does anyone have any other issues before we get back to work?" Socrates asked.

"I want to suggest a commenting standard for our projects," Thrasymachus said. "Some of our programmers barely comment their code, and everyone knows that code without comments is unreadable."

“…English is less precise than Java or Visual Basic and makes for a lot of excess verbiage. Programming-language statements are short and to the point. If you can't make the code clear, how can you make the comments clear? Plus, comments get out of date as the code changes…”

"I agree with that," Glaucon joined in. "Heavily commented code is harder to read because it means more to read. I already have to read the code; why should I have to read a lot of comments, too?"

"Wait a minute," Ismene said, putting down her coffee mug to put in her two drachmas' worth. "I know that commenting can be abused, but good comments are worth their weight in gold. I've had to maintain code that had comments and code that didn't, and I'd rather maintain code with comments. I don't think we should have a standard that says use one comment for every *x* lines of code, but we should encourage everyone to comment."

Mystery Routine Number One

```
// write out the sums 1..n for all n from 1 to num
current = 1;
previous = 0;
sum = 1;
for ( int i = 0; i < num; i++ ) {
   System.out.println( "Sum = " + sum );
   sum = current + previous;
   previous = current;
   current = sum;
}
```

Number 2, includes comments

```
// set product to "base"
product = base;

// loop from 2 to "num"
for ( int i = 2; i <= num; i++ ) {
   // multiply "base" by "product"
   product = product * base;
}
System.out.println( "Product = " + product );
```

Number 3

```
// compute the square root of Num using the
// Newton-Raphson approximation
r = num / 2;
while ( abs( r - (num/r) ) > TOLERANCE ) {
   r = 0.5 * ( r + (num/r) );
}
System.out.println( "r = " + r );
```

Kinds of Comments

- Repeat of the code
  - basically useless
- Explanation of the code
  - usually explain complicated, tricky, or sensitive pieces of code. Sometimes useful, but if the code is so complicated that it needs to be explained, then might be better to improve the code.
- Marker in the code
  - return NULL; // ***** NOT DONE! FIX BEFORE RELEASE!!!
- Summary of the code
  - summaries a few lines of code.. a lot better than repeating the code
- ****Description of the code's intent
  - // get current employee information (intent) vs.
  - // update employeeRecord object (summarizes the solution)
- Info that is impossible to put into code
  - copyright, confidentiality, version numbers, notes about design, references, etc.

Comment Efficiently

```
//  Variable        Meaning
//  --------        -------
//  xPos .......... XCoordinate Position (in meters)
//  yPos .......... YCoordinate Position (in meters)
//  ndsCmptng...... Needs Computing (= 0 if no computation is needed,
//                                   = 1 if computation is needed)
//  ptGrdTtl....... Point Grand Total
//  ptValMax....... Point Value Maximum
//  psblScrMax..... Possible Score Maximum
```

Commenting Efficiently (2)

```
/**********************************************************************
 * class:  GigaTron (GIGATRON.CPP)                                    *
 *                                                                    *
 * author: Dwight K. Coder                                            *
 * date:   July 4, 2014                                               *
 *                                                                    *
 * Routines to control the twenty-first century's code evaluation     *
 * tool. The entry point to these routines is the EvaluateCode()      *
 * routine at the bottom of this file.                                *
 **********************************************************************/
```

vs.

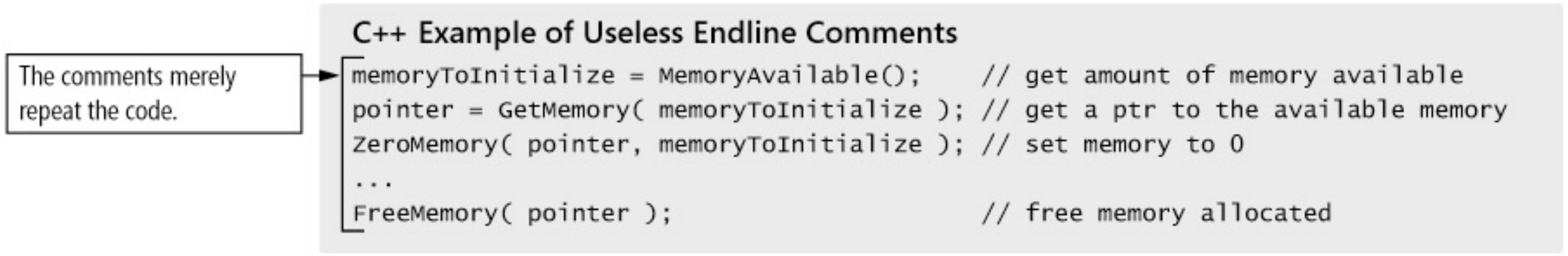
```
/**********************************************************************
   class: GigaTron (GIGATRON.CPP)

   author: Dwight K. Coder
   date:   July 4, 2014

   Routines to control the twenty-first century's code evaluation
   tool. The entry point to these routines is the EvaluateCode()
   routine at the bottom of this file.
 **********************************************************************/
```

Techniques

- Use endline comments appropriately.
- Bad examples:


The comments merely repeat the code.

C++ Example of Useless Endline Comments

```
memoryToInitialize = MemoryAvailable();    // get amount of memory available
pointer = GetMemory( memoryToInitialize ); // get a ptr to the available memory
ZeroMemory( pointer, memoryToInitialize ); // set memory to 0
...
FreeMemory( pointer );                     // free memory allocated
```

```
For rateIdx = 1 to rateCount                          ' Compute discounted rates
   LookupRegularRate( rateIdx, regularRate )
   rate( rateIdx ) = regularRate * discount( rateIdx )
Next
```

- Good use of endline comments:

```
int boundary = 0;          // upper index of sorted part of array
String insertVal = BLANK; // data elmt to insert in sorted part of array
int insertPos = 0;         // position to insert elmt in sorted part of array
```

Techniques (2)

- Comment at the paragraph of code level:

```
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

- Focus comments on Why not How:

```
// if account flag is zero (HOW)
if ( accountFlag == 0 ) ...

// if establishing a new account (WHY)
if ( accountFlag == 0 ) ...

// establish a new account (COMMENT FOR SECTION HEADER, CODE TELLS WHY)
if ( accountType == AccountType.NewAccount ) {
   ...
}
```

Techniques (3)

- Document a surprise:

```
for ( element = 0; element < elementCount; element++ ) {
   // Use right shift to divide by two. Substituting the
   // right-shift operation cuts the loop time by 75%.
   elementList[ element ] = elementList[ element ] >> 1;
}
```

- Document a workaround:

```
blockSize = optimalBlockSize( numItems, sizePerItem );

/* The following code is necessary to work around an error in
WriteData() that appears only when the third parameter
equals 500. '500' has been replaced with a named constant
for clarity.
*/
if ( blockSize == WRITEDATA_BROKEN_SIZE ) {
   blockSize = WRITEDATA_WORKAROUND_SIZE;
}
WriteData ( file, data, blockSize );
```

Techniques (4)

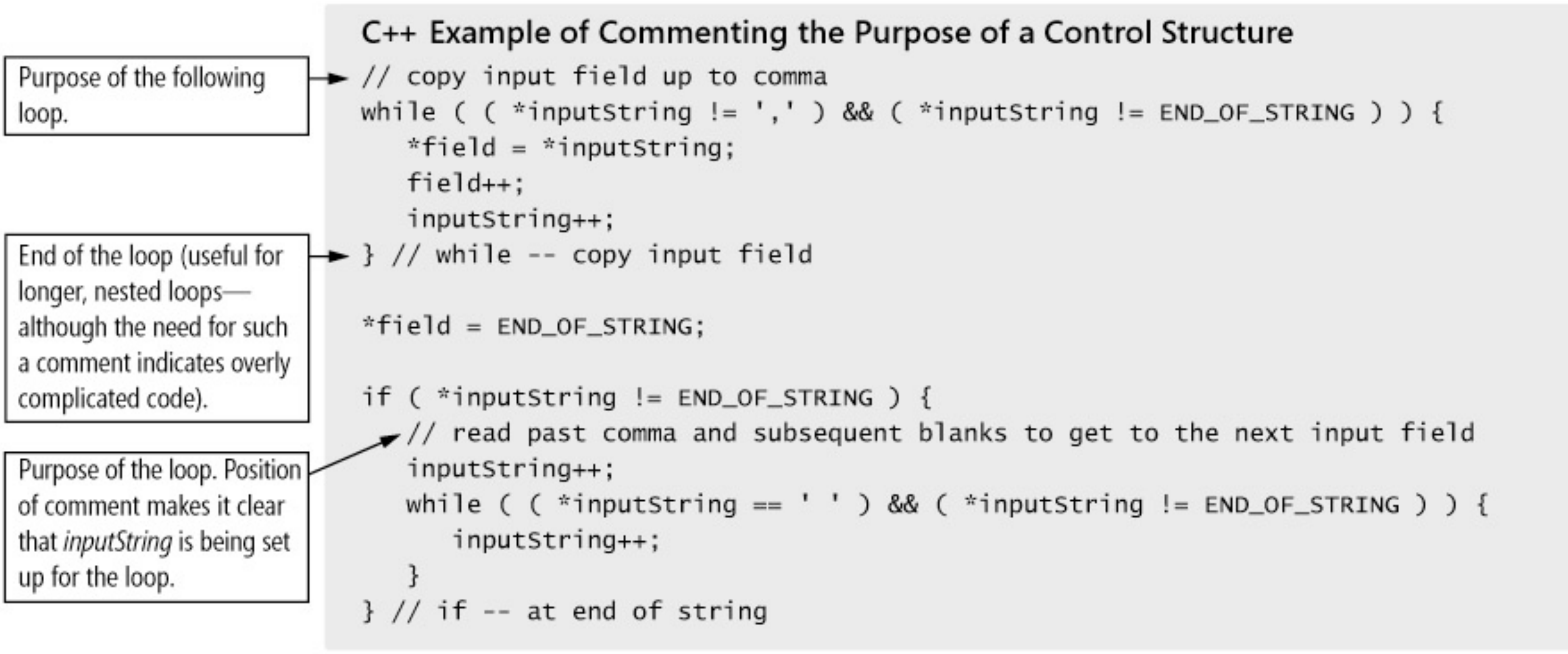
- Rewrite rather than comment tricky code (this is real):

```
// VERY IMPORTANT NOTE:
// The constructor for this class takes a reference to a UiPublication.
// The UiPublication object MUST NOT BE DESTROYED before the DatabasePublication
// object. If it is, the DatabasePublication object will cause the program to
// die a horrible death.
```

- You might say comments that mention death are a good indication your code might need to be revisited.

Techniques (5)

- Commenting control structures:



C++ Example of Commenting the Purpose of a Control Structure

Purpose of the following loop.

End of the loop (useful for longer, nested loops—although the need for such a comment indicates overly complicated code).

Purpose of the loop. Position of comment makes it clear that *inputString* is being set up for the loop.

```
// copy input field up to comma
while ( ( *inputString != ',' ) && ( *inputString != END_OF_STRING ) ) {
   *field = *inputString;
   field++;
   inputString++;
} // while -- copy input field

*field = END_OF_STRING;

if ( *inputString != END_OF_STRING ) {
   // read past comma and subsequent blanks to get to the next input field
   inputString++;
   while ( ( *inputString == ' ' ) && ( *inputString != END_OF_STRING ) ) {
      inputString++;
   }
} // if -- at end of string
```

Techniques (6)

- Documenting routines. If I had to put this in front of every routine I write, my program would consist of one routine!

```
'*********************************************************************
' Name: CopyString
'
' Purpose:      This routine copies a string from the source
'               string (source) to the target string (target).
'
' Algorithm:    It gets the length of "source" and then copies each
'               character, one at a time, into "target". It uses
'               the loop index as an array index into both "source"
'               and "target" and increments the loop/array index
'               after each character is copied.
'
' Inputs:       input    The string to be copied
'
' Outputs:      output   The string to receive the copy of "input"
'
' Interface Assumptions: None
'
' Modification History: None
'
' Author:       Dwight K. Coder
' Date Created: 10/1/04
' Phone:        (555) 222-2255
' SSN:          111-22-3333
' Eye Color:    Green
' Maiden Name:  None
' Blood Type:   AB-
' Mother's Maiden Name: None
' Favorite Car: Pontiac Aztek
' Personalized License Plate: "Tek-ie"
'*********************************************************************
```