Coding Techniques

Code Design and Tuning

Layout of Source Code

• Good layout affects how easy it is to understand the code, review it, and revise it months after it was written

• Picky details?
  – Attention to such details makes a difference in the quality and maintainability of the code!
  – Must be done during initial construction

• Team should agree on a style before coding begins
import java.io.*; class OutputStars {    public static void main(String[] args) throws Exception   {
    BufferedReader inFromUser= new BufferedReader(    new InputStreamReader(System.in)); String s;
    int n;
    int i,j;
    System.out.println("Enter a number greater than 1:");
    s = inFromUser.readLine();n = Integer.parseInt(s);
    if (n < 1) {
        System.out.println("Number must be positive.");    } else {
        // Fill in the code here to output the triangle of *'s for
        (i=1;i<=n; i++) {
            for(j=1;j<=i; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}

Layout

• Fundamental theorem of formatting
  – Good visual layout shows the logical structure of the program

  – Making the code look pretty is worth something, but less than showing the code’s structure

  – Lots of techniques, match up curly braces, group related items together, e.g. 3*4+4*5, indent code, use whitespace such as spacing and blank lines

  – Bad: if (x<3)
      
      a=1;
      b=2;
      c=3;
      d=4;
Religion Wars

- Layout approaches religion on how formatting should be done among programmers

- Key points:
  - Accurately represent the logical structure of the code
  - Consistently represent the logical structure of the code
  - Improve readability
  - Withstand modifications

Whitespace

- Could you read a book as easily if there were no whitespace? Consider the gutters and margins

- Blank Lines
  - Help group related statements
  - Study in 1990 found that the optimal number of blank lines in a program is 8-16%
    - Above 16% and debug time increased dramatically

- Alignment
  - Align elements that belong together, e.g. = in columns

- Indentation
  - 1983 study, subjects scored 20-30% higher on a comprehension test when programs had 2-4 space indentation scheme vs. no indentation
  - Second lowest scores on six-space indentation! Even though subjects thought it looked most pleasing, less effective than 2-4 space indentation
Layout Styles

- Pure blocks
  
  if (...) begin
  
  statement1;
  
  statement2;
  
  end if

- Shortened Pure blocks
  
  if (...) begin
  
  statement1;
  
  statement2;
  
  end

- Endline
  
  while (x==y) do {
  
  statement 1;
  
  statement 2;
  
  }

Using Only One Statement Per Line

- Several statements on one line
  - Takes up fewer screens of space
  - Might be OK for related statements, e.g. initialization

- Better to leave one statement per line
  - Doesn’t hide complexity of a line
  - Some optimizing compilers use syntax as clues for optimization
  - Programmer forced to read left-right and top-bottom
  - Harder to find errors if in the middle of the line
  - Harder to step through code with a debugger
  - Harder to comment out individual statements
Declaring Functions

- Recommendation: Put each parameter on a line

```c
int SomeFunction( int numEmployees,
                  EList Employees,
                  File InputFile,
                  Rec dataRecord )
```

Extra work but holds up better under modification

Debugging

- For many programmers, debugging is the hardest part of programming
- First bug, Mark I computer:
Some are better than others

- Study of twelve programmers with at least four years of experience
- Fastest three programmers:
  - Average debug time: 5 minutes
  - Average number of errors not found: 0.7
  - Average number of errors made correcting errors: 3.0
- Slowest three programmers:
  - Average debug time: 14.1 minutes
  - Average number of errors not found: 1.7
  - Average number of errors made correcting errors: 7.7
- Use errors as opportunities
  - Learn about the program, kinds of mistakes you make, how you fix errors

Ineffective Debugging

- Guessing
  - Scatter print statements and logic changes until it works
  - More exciting without making backups
  - Learn to use your integrated debugger!
- Don’t waste time trying to understand the problem
- Fix the error with the most obvious patch

```plaintext
X = Compute(Y);
If (Y == 17) X=$25.15;  // Was getting wrong answer for 17
```

- Debugging by superstition
  - Full moon?
  - Re-type program, mysterious whitespace?
  - Compiler’s fault?
Scientific Debugging

• In general
  – Gather data through repeatable experiments
  – Form a hypothesis that accounts for as much relevant data as possible
  – Design an experiment to test your hypothesis
  – Prove or disprove the hypothesis
  – Repeat as needed

• For programming
  – Stabilize the error
  – Locate the source of the error
  – Fix the error
  – Test the fix
  – Look for similar errors

Tips on Finding Errors

• Refine the test cases that produce the error
• Reproduce the error several different ways
• Use the results of negative tests
• Brainstorm for hypothesis
• Narrow the suspicious region of code
• Be suspicious of routines that have had errors before
• Check code that’s changed recently
• Expand the suspicious region of code
• Integrate incrementally
• Check for common errors
• Talk to someone else about the problem
• Take a break
Tips of Fixing Errors

• Understand the problem before you fix it
• Understand the program, not just the problem
• Confirm the error diagnosis
• Relax
• Save the original source code
• Fix the problem, not the symptom
• Make one change at a time
• Check your fix
• Look for similar errors

Code-Tuning Strategies

• Code tuning is the practice of modifying correct code to make it run more efficiently
• Less of a factor in many of today’s systems, particularly business software
• Problem: Efficient code isn’t necessarily better code
Code Tuning Misconceptions

• Reducing the lines of code in a HLL improves the speed of the resulting machine code
  – FALSE
  – Usually more lines is faster due to pipelining
  – Example:
    for (i=0; i<5; i++) a[i]=i;  Time:  0.379
    vs.
    a[0]=0;  Time: 0.051
    a[1]=1;
    a[2]=2;
    a[3]=3;
    a[4]=4;

Code Tuning Misconceptions

• Certain operations are probably faster or smaller than others
  – FALSE!
  – No room for probably, changes with compilers and languages
  – Can reduce portability
• You should optimize as you go
  – FALSE!
  – Almost impossible to identify bottlenecks before a program is working
  – Focusing on performance detracts from other objectives
• A faster program is just as important as a correct one
  – FALSE!
  – Easy to make a fast program that is not correct
Pareto Principle

• 80/20 Rule
  – You can get 80 percent of the result with 20 percent of the effort

• Also applies to program optimization
  – Usually the part that needs to be perfected and optimized is quite small
  – Working toward perfection may prevent completion

• Measurement
  – It pays to measure your code to find the hot spots
  – Don’t assume the hot spots are in a particular place

Matrix Summation Example

• C example of straightforward code

```c
sum = 0;
for (row=0; row<rowCount; row++)
{
  for (col = 0; col < colCount; col++)
  {
    sum += matrix[row][column];
  }
}
```

Every access to a 2D array requires computing base + row*sizeof(row)+column
For a 10x10 matrix, that is 100 multiplications and additions plus loop overhead!

Speedup results? 0. Even with bigger matrices. Compiler had already optimized the first code well enough to match the second.
Common Sources of Inefficiency

- I/O operations
  - Move to random if sequential? Cache?
- Formatted print routines
- Floating point ops
  - Use integers when possible
- Paging

Consider a machine that stores data by rows, with 1K pages

\[
\begin{align*}
\text{for col:=1 to 1000} & \quad \text{for row:=1 to 5} \\
\text{for row:=1 to 5} & \quad \text{for col:=1 to 1000} \\
\text{table[row,col]:=0;} & \quad \text{table[row,col]:=0;}
\end{align*}
\]

- System calls

Code-Tuning Techniques

- Loops
  - Good source of hotspots since loops may run many times
  - Unswitching
    - Make a decision outside the loop if possible
    - Usually means turning the loop inside-out
    - Example:

\[
\begin{align*}
\text{for (i=0; i<count; i++)} & \quad \text{Unswitched loop: (Disadvantages?)} \\
\quad \text{if (sumtype==net)} & \quad \text{if (sumtype==net)} \\
\qquad \text{for (i=0; i<count; i++)} & \quad \text{for (i=0; i<count; i++)} \\
\qquad \text{netSum += amount[i];} & \quad \text{netSum += amount[i];} \\
\text{else} & \quad \text{else} \\
\qquad \text{grossSum += amount[i];} & \quad \text{grossSum += amount[i];}
\end{align*}
\]
Loop Optimization

• Jamming or Fusion
  – Combining two loops that operate on the same set of elements

```c
for (i=0; i<count; i++) {
    name[i] = "";
}
...
for (i=0; i<count; i++) {
    num[i] = 0;
}
...
```

Dangers of jamming? Relatively small time increase, up to 4%

---

Loop Optimization

• Loop Unrolling
  – Directly compute code that would normally be done by the loop
  – Good way to exploit parallelism, pipelining

```c
for (i=1; i<count; i++) {
    a[i] = i;
}
```

```c
for (i=1; i<count-4; i+=4) {
    a[i] = i;
    a[i+1]=i+1;
    a[i+2]=i+2;
    a[i+3]=i+3;
}
for (; i<count; i++)
    a[i]=i; // Catch leftovers
```

21-28% increase in speed
Loop Optimization

- **Minimize Work Inside Loops**
  - Precompute as much as possible

```plaintext
for (i=1; i<count; i++) {
    a[i] = i*j*k*l*m*n;
}
```

```plaintext
z = j*k*l*m*n;
for (i=1; i<count; i++) {
    a[i] = i*z;
}
```

Generally small increase in performance, most compilers can do a similar optimization on its own

- **Also can try strength reduction**

```plaintext
for (i=1; i<count; i++) {
    a[i] = i*j*k*l*m*n;
}
```

```plaintext
increment = j*k*l*m*n;
incAmount = increment;
for (i=1; i<count; i++) {
    a[i] = incAmount;
    incAmount += increment;
}
```

Logic

- **Stop testing when you know the answer**

```plaintext
for (i=1; i<count; i++) {
    if (a[i] == target)
        found=true;
}
```

```plaintext
for (i=1; i<count; i++) {
    if (a[i] == target)
        found=true;
        break;
}
```

- **Order tests by frequency**

```plaintext
switch (inputChar) {
    case '+': ...  
    case '-': ...  
    ...
}
```

Applies to switch and if-then-else
Particularly noticeable inside loops
Logic

- Substitute Table Lookup for Complicated Expressions
- Example:

```c
if ((A && !c) || (A && B && C))
    Class = 1;
else if ((B && !A) || (A && C && !B))
    Class = 2;
else if (C && !A && !B)
    Class = 3;
else
    Class = 0;
```

```c
static int ClassTable[2][2][2] =
/*  !B!C   !B   B!C  BC */
{  0,      3,   2,     2,    /*  !A */
  1,      2,    1,     2 }; /*  A */
```

Arrays

- Minimize Array References

```c
for (discount = 0; discount < numD; discount++)
{
    for (rate = 0; rate < numR; rate++)
    {
        rateTbl[rate] = rateTbl[rate] * discountTbl[discount];
    }
}
```

discountTbl[discount] is unchanged the entire inner loop:

```c
for (discount = 0; discount < numD; discount++)
{
    thisDiscount = discountTbl[discount];
    for (rate = 0; rate < numR; rate++)
    {
        rateTbl[rate] = rateTbl[rate] * thisDiscount;
    }
}
Precompute

- Initialize at compile time, reduce strength, eliminate common sub expressions

```java
for (i=1; i<count; i++) {
    a[i] = Math.pow(log(i) / log(3), 2);
}
```

```java
const LOG3 = log(3);
for (i=1; i<count; i++) {
    a[i] = (log(i) / LOG3) * (log(i) / LOG3);
}
```

```java
const LOG3 = log(3);
for (i=1; i<count; i++) {
    double unSquared = (log(i) / LOG3);
    a[i] = unSquared * unSquared;
}
```

Assembly

- Use inline assembly for critical routine
- Generally preferred to avoid hard-to-read code