Overview
- Coding
- Code tuning
  - Monday 10/25: design due - start your prototype
  - Prototype due on 11/5

When to create a routine
- reduce complexity
  - "procedural abstraction"
- avoid duplication ("code factoring")
- support subclassing
  - easier for short, well-designed routines
- hide sequencing
- hide pointer operations
- improve portability
- simplify boolean tests
- improve performance
  - can recode with better algorithm
Design at Routine Level

- **Functional cohesion**
  - routine performs exactly one function

- **Good routine names**
  - describe everything it does
  - avoid meaningless names (e.g., dealWithInput())
  - to name a function, use description of return value
  - to name a procedure, use verb followed by an object, e.g., PrintDocument()

- **Use precise opposites**
  - add / remove, begin / end, open / close etc.

- **Establish conventions for common names**
  - not employed to dealWithInput() GetId() etc.

Routine Size

- No unequivocal evidence that large size increases errors or vice versa

- **Rule of thumb**
  - no more than 200 lines
  - if longer, maybe can factor out code

Routine Parameters

- **Order them**
  - eg, IN - IN/OUT - OUT

- **Use same order across routines with similar parameters**
  - bad example C: printf / fprintf (file argument first) but puts / fputs (last)

- **Use all parameters**
  - unused parameters correlated with increased errors (Card et al., 1986)

- **Put status / error variables last**

- Don't use parameters as work variables
Routine Parameters (2)

- Document interface assumptions about parameters
- Limit the # of parameters (7 is a rule thumb)
- Establish a naming convention for IN / IN-OUT / OUT parameters
  - e.g., input_ inout_ out_
- Make sure actuals match formals

Special Considerations

- Functions versus procedures
  - primary purpose is to return a value, then use a function, otherwise a procedure
- Setting the return value
  - don’t return references or pointers to local data
  - make sure value is set on all paths

Macros & Inline Routines

- Fully parenthesize macros
  - #define Cube(a) a*a*a will break: Cube (x+1)
  - #define Cube(a) (a)* (a) *(a) still incorrect!
    - #define Cube(a) ((a)* (a) *(a))
- Use {} for multiple-line macros
  - use only sparingly, use procedures instead
- Use inline routines sparingly
  - violate encapsulation
  - turn on interprocedural optimization instead (i.e., use a good compiler)
Refactoring

“A change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior” (Fowler)

Refactoring Reasons

Kinds of Refactoring

- Data-level
- Statement-level
- Routine-level
- Class implementation
- Class interface
- System Level
Code Tuning

- Jackson's rule (1): "Don't do it"
- Jackson's rule (2): "Don't do it yet"
  - avoid doing tuning as you code
  - more important to be correct first
- 80/20 rule
  - often 80% of time spent in 20% of the code (aka Pareto principle)
    - use profiling to find and concentrate on those code parts

Sources of Inefficiency

- Wrong algorithm \(O(n^2)\) sorting versus \(O(n \log n)\)
- I/O operations
  - in memory access \(\sim 1,000\) times faster
- Memory hierarchy
  - cache effects: bad locality slowdowns of up to \(100x\)
- System calls
  - \(100s\) or \(1000\) of machine cycles

Get help from the Compiler

- Often compiler can improve execution time by \(2x\)
  - typical for gcc -O3 compared to -O1/-O0
- Coding style can help or obstruct the compiler
  
  ```
  for (i=0;i<LIMIT; i++)
    a[i] = b[i];
  ```

  ```
  for (p=a, q=b; p!= LIMIT; p++, q++)
    *p = *q;
  ```
Optimization Overview

- Optimization seeks to improve a program’s utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent, etc.
- Optimization should not alter what the program computes
  - The answer must still be the same
  - If they are different, most likely your program is wrong (and you were just lucky it didn’t break earlier)

Compiler Optimizations

- Work in two parts
  - program analysis
    - compute useful properties about the program, e.g., \( x = 5 \) (a constant)
    - harder if program is written in obfuscated style
    - pointers are particularly difficult to analyze
  - program transformation
    - transform into equivalent but better (faster) version
    - based on properties computed by analysis that guarantee the correctness

A Classification of Optimizations

- There are several granularities of optimizations
  - Local optimizations
    - Apply to a basic block in isolation
  - Global optimizations
    - Apply to a control-flow graph (method body) in isolation
  - Inter-procedural optimizations
    - Apply across method boundaries
- Almost all compilers do (1), most do (2) and some do (3)
Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification

Algebraic Simplification

- Some statements can be deleted
  \[ x := x + 0 \]
  \[ x := x \times 1 \]
- Some statements can be simplified
  \[ x := x \times 0 \Rightarrow x := 0 \]
  \[ y := y \times 2 \Rightarrow y := y \times y \]
  \[ x := x \times 8 \Rightarrow x := x << 3 \]
  \[ x := x \times 15 \Rightarrow t := x << 4; x := t - x \]

Constant Folding

- Operations on constants can be computed at compile time.
- In general, if there is a statement
  \[ x := y \text{ op } z \]
  - And \( y \) and \( z \) are constants
  - Then \( y \text{ op } z \) can be computed at compile time
- Example: \[ x := 2 + 2 \Rightarrow x := 4 \]
- Example: if \( 2 < 0 \) jump \( L \) can be deleted
Flow of Control Optimizations

- Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or "fall through" from a conditional
  - Such basic blocks can be eliminated
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster
    - Due to memory cache effects (increased spatial locality)

Common Subexpression Elimination

- Assume
  - Basic block is in single assignment form
  - A definition $x :=$ is the first use of $x$ in a block
- If any assignment have the same rhs, they compute the same value
- Example:
  
  $x := y + z$
  $\vdash x := y + z$
  $\vdash$ etc.
  $w := y + z$

  (the values of $x$, $y$, and $z$ do not change in the code)

Copy Propagation

- If $w := x$ appears in a block, all subsequent uses of $w$ can be replaced with uses of $x$
- Example:
  
  $b := z + y$
  $a := b$  $\Rightarrow$  $a := b$
  $x := 2 * a$  $\Rightarrow$  $x := 2 * b$

- This does not make the program smaller or faster but might enable other optimizations
  - Constant folding
  - Dead code elimination
Copy Propagation and Constant Folding

- Example:
  \[ a := 5 \]
  \[ x := 2 \ast a \Rightarrow x := 10 \]
  \[ y := x + 6 \Rightarrow y := 16 \]
  \[ t := x \ast y \Rightarrow t := x << 4 \]

Copy Propagation and Dead Code Elimination

- If \( w := \text{rhs} \) appears in a basic block
  - \( w \) does not appear anywhere else in the program
  - Then the statement \( w := \text{rhs} \) is dead and can be eliminated
    - Dead = does not contribute to the program’s result

- Example: (\( a \) is not used anywhere else)
  \[ x := z + y \]
  \[ a := x \Rightarrow a := x \Rightarrow x := 2 \ast x \]

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations interact
  - Performing one optimization enables other opt.
  - Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - The optimizer can also be stopped at any time to limit the compilation time
An Example

■ **Initial code:**

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

■ **Algebraic optimization:**

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

■ **Algebraic optimization:**

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```
An Example

- **Copy propagation:**
  
  ```
  a := x * x
  b := 3
  c := x
  d := c * c
  e := b << 1
  f := a + d
  g := e * f
  ```

An Example

- **Copy propagation:**
  
  ```
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 << 1
  f := a + d
  g := e * f
  ```

An Example

- **Constant folding:**
  
  ```
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 << 1
  f := a + d
  g := e * f
  ```
An Example

- **Constant folding:**
  
  a := x * x  
  b := 3  
  c := x  
  d := x * x  
  e := 6  
  f := a + d  
  g := e * f

An Example

- **Common subexpression elimination:**
  
  a := x * x  
  b := 3  
  c := x  
  d := x * x  
  e := 6  
  f := a + d  
  g := e * f
An Example

- Copy propagation:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

- Copy propagation:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + a \]
  \[ g := 6 \times f \]

An Example

- Dead code elimination:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + a \]
  \[ g := 6 \times f \]
An Example

- Dead code elimination:
  
  \[ a := x \times x \]

  \[ f := a + a \]
  \[ g := 6 \times f \]

- This is the final form