# Efficiency, Reliability, and Design

## CSE250 Lecture Notes Weeks 3–4+

Kenneth W. Regan
University at Buffalo (SUNY)

September 20, 2010

# Efficiency and Reliability

- Historically, a tradeoff.
- But both are helped by *declarativeness*.
- Means representing concepts, categories, and logical properties directly in code, rather than in your brain (only).
- Examples: *classes* for category nouns, `const` for logical constants, separate functions for separate operations, exception classes (in Java) for particular errors…
- Representing *properties* is a current challenge…
- …Hence burden currently falls on *comments*, *assertions*, *annotations*, *invariants*, and *requires/ensures*… (Lectures will have examples along lines of pp166–170 and Ch. 7.)

# Examples of the Tradeoff

- Named functions versus assembler or "spaghetti code."
- Use of `goto` considered harmful—though good for optimizing some loops.
- Recursive functions are often easier to reason about (ch. 7), but have high calling overheads (less so on newer compilers?).
- Strict expression semantics (as in Java) impedes some optimizations.
- Greater indirection, as in Java, loses some time but reduces code dependencies.
- Use of objects and high-level coding constructs in general...

Smarter compiler technology is reducing all these drawbacks—and C++ templates were designed to *eliminate* the last!

# Walking And Chewing Gum At The Same Time. . .

A "mini" case is returning a value and causing a change in data at the same time. Some authorities warn against it, and avoiding its pitfalls is a main idea of "Functional Languages" (to come in CSE305). Examples:

1. `return elements->at(rearItem++);     //implicitly pops it`
2. `while(getline(*INFILEp,line)) { ... //reads and tests`
3. 
```
out += sq->pop() + " " + sq->pop(); //which pop is first?
```

The last is bad because the order of the two changes in one expression is not defined in C++. Java does mandate left-to-right order here, but even so such expressions are considered *Programmer Errors*.

The first two, however, are *fine*. Indeed they are common idioms. When we define "glorified pointers" called *iterators*, we will use `*itr++` all the time—and this is just a direct translation of Java's `next()` operator.

# Program and Design Efficiencies

- *Program* efficiencies usually make at most a *constant-factor* difference in running time. E.g. if you save 3 statements in a `for(int i = 0; i < n; i++)` loop that originally had 14 statements, then your new running time in the loop is 11/14 of the old running time.
- Smarter compilation also usually saves at most a constant factor. Ditto faster processors, or doubling the number of cores!
- For statements not within loops the savings is just an *additive* constant.
- Hence to distinguish *greater* efficiencies that result from good *design*, it is convenient to have a notation that ignores constant factors and additive terms.

# Big-$O$ Notation

Suppose that on problems with $n$ data items (counting chars or small ints/doubles), your program takes at most $t(n)$ steps. Let $g(n)$ stand for a performance target. Then

$$t(n) = O(g(n)),$$

meaning your *program design* achieves the target, if there are constants $c > 0$ and $n_0 \geq 0$ such that:

$$\text{for all } n \geq n_0,\ t(n) \leq cg(n).$$

Here $c$ is called "the constant in the $O$" and should be estimated and minimized as well, even though "$t = O(g)$" does not depend on it. Having $n_0$ be not excessive is also important. (Often we think of "$c$" as being $\geq 1$.)

# Principal Constant

- Actually, the value of $c$ which you use to satisfy the definition of $O$-notation is hard to make best-possible. So I say a particular choice is "reported."
- E.g. $g(n) = n^2$, $t(n) = 5n^2 + 20n - 10$.
- If you "report" $c = 10$, then since $5n^2 + 20n - 10 \leq 10n^2$ whenever $5n^2 - 20n + 10 \geq 0$, so you get $n_0 = (20 + \sqrt{400 - 200})/10$ up to int, $= 3$.
- But if you try $c = 6$, you get the bigger $n_0 = (20 + \sqrt{400 - 160})/2$ up to int, $= 18$.
- You can do it with $c = 5.1$, or any $c = 5 + \epsilon$, but ironically you can never satisfy the definition with $c = 5$ exactly!
- Still 5, the coefficient of the leading term, is "the truth," so we call it the *principal constant*.

tead.

# Extra Notation $\Omega, \Theta, o$ (not in text)

- If $f(n) = O(g(n))$, then we can also write $g(n) = \Omega(f(n))$. In full this means that there are $c > 0$ and $n_0$ such that

$$\text{for all } n \geq n_0,\ g(n) \geq \frac{1}{c} f(n).$$

  Compare $f = O(g)$ meaning ... $f(n) \leq cg(n)$.

- If it goes in reverse, so that $g(n) = O(f(n))$ as well, then we say *$g(n)$ and $f(n)$ have the same growth order*, and we write $g(n) = \Theta(f(n))$.
- If $\lim_{n \longrightarrow \infty} f(n)/g(n) = 0$, then we can say something even stronger than "$f = O(g)$." We write $f = o(g)$ to signify that $f$ has a strictly lower growth rate.

# Analogy to $<, =, >$

- The real numbers enjoy a property called trichotomy: for all $a, b$, either $a < b$ or $a = b$ or $a > b$.
- Functions $f, g : \mathbf{N} \longrightarrow \mathbf{N}$ do not, e.g. $f(n) = \lfloor n^2 \sin n \rfloor$ and $g(n) = n$ [a quick hand-drawn graph was enough to show this in class].
- However, the British mathematicians Hardy and Littlewood proved that *for all real-number functions $f, g$ built up from $+, -, *, /$ and $\exp, \log$ only*,

$$f = o(g) \qquad \text{or} \qquad f = \Theta(g) \qquad \text{or} \qquad g = o(f).$$

- Thus common functions fall into a nice linear order by growth rate (see chart from text).

## Extra Slide—looking ahead. . .

- The notion of "trichotomy" is generally useful for reasoning about custom-made $<$ comparisons that are compound or not even numeric.
- If you test `x < y` and `y < x` and both of those return `false`, are you allowed to deduce that `x == y`?
- The K-W text does this with binary search trees at the bottom of page 471. It can do so because that code requires that all items in the tree be distinct.
- When you infer "==" from the $<$ and $>$ tests failing, you are said to be "assuming trichotomy."
- An example where you can't [which was mentioned in class prior to this slide] is the relation "southwest of" for two `Point` objects `p1,p2` as defined by

```
bool operator<(Point p1, Point p2) {
   return p1.x < p2.x && p1.y < p2.y;
}
```

# L'Hôpital's Rule and Little-oh

- When $f = o(g)$, sometimes it's not immediately obvious that $\lim_{n \longrightarrow \infty} f(n)/g(n) = 0$.
- E.g. $f(n) = n^3$, $g(n) = 2^n$. In that case use **L'Hôpital's Rule**: If $f(n)$ and $g(n)$ both go to $\infty$ or to 0, then

$$\lim_{n \longrightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \longrightarrow \infty} \frac{f'(n)}{g'(n)}$$

  *provided* the latter limit exists at all.
- Here $f'(n) = 3n^2$ and $g'(n) = 2^n(\ln 2)$, and we still don't know. So iterate: $f''(n) = 6n$, $g''(n) = 2^n(\ln 2)^2$; $f'''(n) = 6$, $g'''(n) = 2^n(\ln 2)^3$.
- Now it's obvious that $\lim_{n \longrightarrow \infty} f'''(n)/g'''(n) = 0$. Working backwards, the Rule means the same is true of $\lim f''(n)/g''(n)$, $\lim f'(n)/g'(n)$, and finally $\lim f(n)/g(n)$, so $f = o(g)$.

# The Factorial Case

- The function $f(n) = n!$ comes up in sorting and problems involving permutations.
- It's bigger than $2^n$. How much bigger?
- *Stirling's Formula*

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n + g(n),$$

  where $g(n)/n! \longrightarrow 0$ (so we can "asymptotically ignore" $g(n)$).
- The "$\sqrt{n}$" in front prevents us from making a simpler "Theta" relation. However:

$$\begin{aligned} \log n! &= n(\log(n) - \log(e)) + \frac{1}{2}(\log n + \log(2\pi)) \\ &= \Theta(n \log n). \end{aligned}$$

- This rigorous $\Theta$ relation can be used to prove that any method of sorting $n$ items via comparisons must take $\Omega(n \log n)$ time.
- It also justifies the "hazy" notation "$n! \approx 2^{n \log n}$."

# Intuitive Meaning of Growth Comparisons

Let $g(n)$ stand for an exact performance target, $f(n)$ for some other definite function, $t(n)$ for the actual running time of your program, and $u(n)$ for a rival methodology.

1. $t(n) = \Theta(f(n))$ means, "I know the asymptotic performance of my program pretty well."
2. $t(n) = O(g(n))$ means: your methodology is fine, and you can probably tweak your code with constant-factor improvements and/or better hardware to make your exact target.
3. $t(n) = o(u(n))$ means: your program will eventually slay its rival.
4. $u(n) = \Omega(f(n))$ means: $f(n)$ is a growth lower bound on the innate capability of the (other) design.

Example of the last: sorting via comparisons needs time $\Omega(n \log n)$. *Proving* other believed examples is the hardest problem in theoretical computer science, prize $1,000,000.

# Tradeoff (aka. Crossover) Points

- But when $t(n) = o(u(n))$, don't get cocky: for "small $n$" the other program may still beat you.
- [Show chart from text again, but this time note that the lower-growing functions are actually higher at the left end.]
- Interestingly, this purely-math phenomenon shows up in real code.
- [Show demo of Insertion Sort with $u(n) = \Theta(n^2)$, versus recursive Merge Sort with $t(n) = \Theta(n \log n)$...]
- [...But aaaaaaaarrrghhh!, computers today are so freakin' fast that I can no longer show the tradeoff with a millisecond timer!—at least iterating the code just once...]
- Note Merge Sort is "asymptotically best-possible"—but other $\Theta(n \log n)$ sorts (to come in Ch. 10) tend to beat its implementations on the principal constant.
- [Given definite time functions $t(n)$ and $u(n)$, calculating the (last) crossover point $n_1$ is like finding "$n_0$" to verify $O$-notation.]

# Reliability Factors

- The text covers many good software-design and coding factors in chapters 1–2 and 7, and throughout...
- Several should be familiar from previous CS courses.
- Of all we will emphasize:

  Modularity

  and (my umbrella term)

  Logic Commenting.
- Commenting is called *annotating* when comments are in a standard format that a postprocessor (e.g. `javadoc`), or even the compiler itself, can analyze.
- The text @nnotates parameter names...we will try to systematize other logical properties of methods and classes and relationships.

# Modularity

- Definition is hard to pin down.
- Abstraction and Information Hiding are key (sect. 1.2).
- ADTs (sect. 1.4) are necessary but not sufficient:
  - The “Zillions of Little Classes” problem…
  - Coupling of classes…
  - Inheritance can make code non-modular.
- A stab at a definition: modularity is the organization of code into components so that dependencies among components are sparse, and implementations of components can be changed without affecting neighbors.

# Modularity and Testing: Classes

- Example: `bigint.h` by Rossi-Vinokur, and my `FibonacciTimes.cpp` client.
- Can switch implementation from `RossiBigInt` to `VinBigInt` by changing a single C++ `typedef` line.
  - (It might be even better if the implementations were in separate files and the switch line were in a separate "gateway" file...)
- [Demo in class of running times and then tracking down a `Segmentation fault` error.]
- Rotating the two modules gave some confidence that the fault was not in either class.
- A "stub" (text, pp156–157) can be for a whole class or package as well as a function or method. The empty body doesn't give the same confidence as an alternative implementation, but it can help for testing other code, and is important in *prototyping*.

# Modularity and Testing: Functions

- The `FibonacciTimes.cpp` client has several different ways of computing big Fibonacci numbers.
- One was unusable for big numbers (double-branch recursion), but single-branch recursion and a non-recursive function were fine.
- Fault disappeared when the non-recursive version was used.
- This exonerated the big-int classes completely, and pinned trouble on the recursion.
- Turns out asking for 50,000 recursions (to get the 100,000th Fibonacci number) exhausted the memory map for simultaneous activaton frames on `timberlake`—the limit for this method seems to be in the 41,000s (it varies).
- Case where a seg-fault was **not** a bad pointer.
- Ironic that the "dumb" function could do well over 50,000—indeed millions—of recursions to compute $F_{30}$ with no fault. . . because no more than 29 were *activated* at any one time.

# Logic Comments: Why?

- Not all important properties and relationships can be expressed or enforced by code statements themselves.
- Even something as simple as `fib n` needing $n \geq 0$:
  - Enforcing by declaring `n` as `unsigned` rather than `int` is discouraged. (E.g. in Microsoft .NET, `uint32` is not "CLS-compliant.") Mixing `int` and `unsigned` can be a pain...
  - Hey, "`n`" is a command-line argument: the user CAN type "-1"!
  - The langauge Ada in the 1980s tried to standardize this by having a `subtype natural` of the integers, but that didn't stop an Ariane rocket control program from malfunctioning when the underlying hardware wordsize was doubled (and exceptions left on)...
  - ...and it wouldn't have helped the loss of a Mars probe because one team thought units were `kph`, the other `mph`!
- Undecidability results (CSE396) *may* mean that sentient beings will *never* escape the need for ad-hoc logic comments.
- Hence current emphasis on *writing* them...and even *systematizing* them.

# Requires, Ensures, Maintains

- REQ and ENS are the same as PRE and POST, but emphasizing communication between methods.
  - REQ is also more specific to methods than what the text calls a "requirements specification" on pp68–69.
  - The names come from Bertrand Meyer's "Design By Contract" and Eiffel language.
- *Class invariants* (CLASS INV) are properties maintained by a class that are essential for its interpretation and run-time operation.
- *Loop invariants* (LOOP INV) are features that stay constant while other things change in a loop.
- *Recursion invariants* (REC INV) hold between recursive calls.
- AOK to abbreviate these three to just INV.

# LOOP INV and PRE + POST

A loop invariant abbreviates PRE and POST for a loop body—but may be more useful during the body too. E.g. for Insertion Sort:

```
for (int i = 1; i < n; i++) {
   // LOOP INV: vec[0..i) is sorted.
   [body]
}
```

abbreviates

```
for (int i = 1; i < n; i++) {
   // PRE: vec[0..i) is sorted.
   [body]
   // POST: vec[0..i+1) is sorted
}
```

- LOOP INV should be true as the loop is entered.
- Truth on exit (e.g. for $i = n$) should imply the goal.

# Checking Logic By Assertions

- “Simple” properties can be checked at runtime by assertions `assert(e)` where `e` is a Boolean expression.
  - at top of a method for REQ/PRE;
  - at bottom for ENS/POST.
  - on constructor exit for a CLASS INV, or anytime.
- Example: `merge(left,right,target)` requires

```
target.size() == left.size() + right.size()
```

  (or with “`>=`” in place of “`==`”).
- Easier to `assert` this with `vector` than with raw C/C++ arrays.
  - REQ: `left` and `right` are sorted
  - ENS: `target` is sorted.
  - Checking these assertions takes an extra $\Theta(n)$ time—on each call!
- `mergeSort(left)`, `mergeSort(right)` yield a REC INV.

# Class Invariants

Can be brief, expressing just the most important, least-obvious points. Examples:

- `StringStack.cpp`: `top` designates the first free space above the top element.
- With `vector` and other STL containers: `begin()` indexes the first element, but `end()` always means one place *past* the last element.
  - Just like "0" and "n" in a `for(int i = 0; i < n; i++){...}` loop.
- My `CPUTimer.h` timer class maintains duplicate copies `timestamp` and `prevStamp` of the last clock reading...
  - ...so that the very first line in the new-reading method gets the time, which overwrites `timestamp`.
  - You need the *difference* of two clock readings to measure a duration.