# V22.0474-001 Software Engineering
## Spring 2008
## Lecture 22: Code Optimization II

Clark Barrett, New York University

# Outline

- Project Status Check

- Code Tuning: Logic

- Code Tuning: Loops

- Code Tuning: Data Transformations

- Code Tuning: Other Techniques

**Sources for today's lecture:**

McConnell, Steve. *Code Complete*, Second Edition, ch. 26.

# Project Status Check

**Upcoming assignments**

- *April 10*: Code for testing (hw8) is due.

- *April 17*: Testing report (hw9) is due.

- *April 22*: Testing presentations will take place in class:
  - Send me the slides by 10am on April 22;
  - As before, each team has 20 minutes;
  - The presentation should be given by someone new;
  - Discuss unit, integration, and system/user-testing on your own project.

- *April 29*: Testing report for other team (hw10) is due.

# Project Status Check

**Upcoming assignments**

- *May 1*: Final project presentations
  - Each team will have 30 minutes to present their project;
  - Final presentations should present the final product as well as discuss what was accomplished and learned during the project as a whole.

- *May 7*: Spring Showcase
  - Demo your project for anyone who wants to see.
  - Refreshments will be served!
  - Invite your friends!

- *May 10*: Final project report and code is due

# Code Tuning

In this lecture, we will discuss *specific code-tuning techniques*.

The *warnings and guidelines* about performance tuning we discussed last time still apply.

In particular, code-tuning usually means trading off *readability and maintainability* for *speed*.

The *effectiveness* of a specific technique will depend on your development environment.

Thus, consider this a list of things to try, but you still need to *measure* performance before and after.

# Code Tuning: Logic

**Stop testing when you know the answer**

*Which is faster?*

```
if ((5 < x) && (x < 10)) {
  ...
}


if (5 < x) {
  if (x < 10) {
    ...
  }
}
```

In C, C++, and Java, there is no difference because these languages have *short-circuit* evaluation. In other languages, however, the second piece of code may be faster.

# Code Tuning: Logic

*How can the same principle be applied to the following code?*

```
negativeInputFound = false;
for (i = 0; i < count; i++) {
  if (input[i] < 0) {
    negativeInputFound = true;
  }
}
```

Once a negative value has been found, it is unnecessary to continue the for loop. Some possible ways to implement this include:

- Use `break` or `goto` to end the loop when `input[i] < 0`.

- Check for `negativeInputFound` as part of the loop terminating condition.

- Use a sentinel (more on this later).

# Code Tuning: Logic

*How could the following code be made faster?*

```
if (c == '+' || c == '=')
  ProcessMathSymbol(c);
else if (c >= '0' && c <= '9')
  ProcessDigit(c);
else if (c == ',' || c == '.' || c == '!' || c == '?')
  ProcessPunctuation(c);
else if (c == ' ')
  ProcessSpace(c);
else if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
  ProcessAlpha(c);
```

**Order Tests by Frequency**

By reordering the tests to put the common cases first, we can avoid many unnecessary tests.

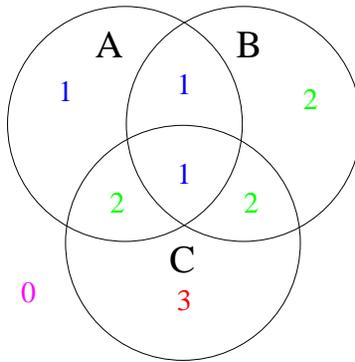**Compare Performance of Similar Logic Structures**

We could also rewrite the code using `switch` and `case`. Would this be faster? It's unclear. The only reliable way to find out is to measure it.

# Code Tuning: Logic

Suppose you wanted to categorize an object into one of four categories (0, 1, 2, 3) based on which groups A, B, or C it belongs to as follows.

*How would you do it?*

# Code Tuning: Logic

One solution:

```
if ((isA(x) && !isC(x)) || (isA(x) && isB(x) && isC(x))) {
  category = 1;
}
else if ((isB(x) && !isA(x)) || (isA(x) && isC(x) && !isB(x))) {
  category = 2;
}
else if (isC(x) && !isA(x) && !isB(x)) {
  category = 3;
}
else {
  category = 0;
}
```

*How can this be improved?*

# Code Tuning: Logic

**Substitue Table Lookups for Complicated Expressions**
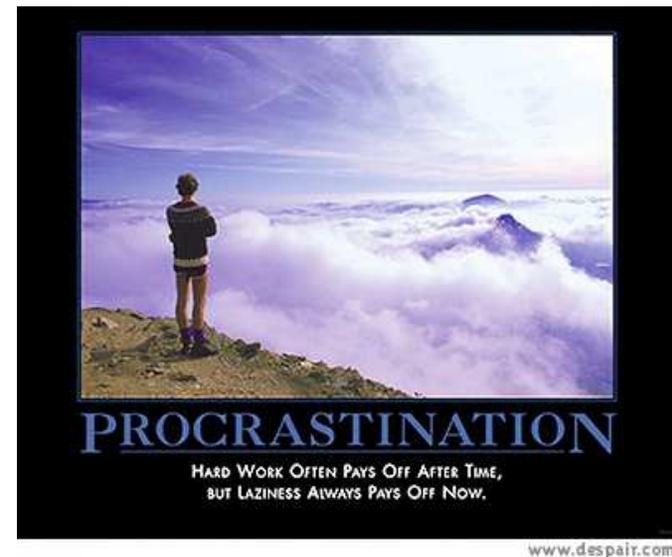
Here is a faster solution:

```
// define categoryTable
static int categoryTable[2][2][2] = {
   // !b!c !bc  b!c  bc
      0,   3,   2,   2,   // !a
      1,   2,   1,   1    // a
};
...
category = categoryTable[isA(x)][isB(x)][isC(x)];
```

A *lookup table*, if well-documented, is faster and easier to maintain than a complicated chain of if-then-else.

# Code Tuning: Logic



*How can you apply this principle in programming?*

# Code Tuning: Logic

**Use Lazy Evaluation**

In programming, laziness means waiting to do work until you know it will be needed.

> For example, suppose you have a table of $5000$ values that might be used, but only a few hundred are actually used in any run of the program. Rather than pre-computing all $5000$ values, you may want to wait and compute only the values you need when you need them.
>
> Note that if most of the $5000$ values are used, or if values are used many many times, it may be faster to pre-compute everything.

As usual, you have to *know your application* and *measure*.

# Code Tuning: Loops

**Unswitching**

*How can this code be made faster?*

```
for (i = 0; i < count; i++) {
  if (sumType == SUMTYPE_NET)
    netSum += amount[i];
  else
    grossSum += amount[i];
}
```

The test `sumType == SUMTYPE_NET` is repeated on each iteration. The loop can be rewritten as follows:

# Code Tuning: Loops

**Unswitching**

*How can this code be made faster?*

```
for (i = 0; i < count; i++) {
  if (sumType == SUMTYPE_NET)
    netSum += amount[i];
  else
    grossSum += amount[i];
}
```

The test `sumType == SUMTYPE_NET` is repeated on each iteration. The loop can be rewritten as follows:

```
if (sumType == SUMTYPE_NET) {
  for (i = 0; i < count; i++)
    netSum += amount[i];
}
else {
  for (i = 0; i < count; i++)
    grossSum += amount[i];
}
```

# Code Tuning: Loops

**Jamming or "Loop Fusion"**

*How would you speed up the following code?*

```
for (i=0; i < employeecount; i++) {
  employeeName[i] = "";
}
for (i=0; i < employeecount; i++) {
  employeeEarnings[i] = 0;
}
```

# Code Tuning: Loops

**Jamming or "Loop Fusion"**

*How would you speed up the following code?*

```
for (i=0; i < employeecount; i++) {
  employeeName[i] = "";
}
for (i=0; i < employeecount; i++) {
  employeeEarnings[i] = 0;
}
```

The two loops can be "fused" as follows:

```
for (i=0; i < employeecount; i++) {
  employeeName[i] = "";
  employeeEarnings[i] = 0;
}
```

*What do you have to worry about before doing loop fusion?*

If there are dependencies between the two loops, you have to check that the fused loop still does the right thing.

Also, note that many compilers perform this optimization automatically.

# Code Tuning: Loops

This technique can be extended. Loops can be unrolled *twice*, *three times*, etc.

Additional unrolls *complicate* the code significantly for *decreasing* performance gains.

As with loop fusion, loop unrolling is done *automatically* by many compilers.

# Code Tuning: Loops

**Unrolling**

Recall that last time we gave an example of a loop that iterated 10 times. The "unrolled" code was 10 times larger but also a bit faster. This principle can be applied to any loop. Consider the following:

```
i = 0;
while (i < count) {
  a[i] = i;
  i++;
}
```

Here is the loop after one unrolling:

```
i = 0;
while (i < count - 1) {
  a[i] = i;
  a[i+1] = i+1;
  i += 2;
}
if (i == count) {
  a[count-1] = count-1;
}
```

# Code Tuning: Loops

**Minimizing the Work Inside Loops**

*How could you make this code faster?*

```
for (i = 0; i < rateCount; i++) {
  netRate[i] = baseRate[i] * rates->discounts->factors->net;
}
```

# Code Tuning: Loops

**Minimizing the Work Inside Loops**

*How could you make this code faster?*

```
for (i = 0; i < rateCount; i++) {
  netRate[i] = baseRate[i] * rates->discounts->factors->net;
}
```

We can move the complicated expression outside the loop. In this case, the result improves both readability and (in many cases) performance.

```
quantityDiscount = rates->discounts->factors->net;
for (i = 0; i < rateCount; i++) {
  netRate[i] = baseRate[i] * quantityDiscount;
}
```

# Code Tuning: Loops

**Sentinel Values**

Consider the following loop:

```
found = false;
i = 0;
while ((!found) && (i < count)) {
  if (item[i] == testValue) {
    found = true;
  }
  else {
    i++;
  }
}

if (found) {
  ...
}
```

*How could you reduce the number of conditions checked for each loop?*

# Code Tuning: Loops

By using a *sentinel*, we can combine all three tests into one:

```
// Make sure there is space for at least count+1 objects
item[count] = testValue;

i = 0;
while (item[i] != testValue) {
  i++;
}

// Check if item was really found
if (i < count) {
  ...
}
```

Sentinels can be applied to nearly any situation in which you use linear search.

Of course, if you can avoid linear search, that's even better...

# Code Tuning: Loops

**Putting the Busiest Loop on the Inside**

*Which is faster?*

```
for (row = 0; row < 100; row++) {
  for (column = 0; column < 5; column++) {
    sum += table[row][column];
  }
}
```

or

```
for (column = 0; column < 5; column++) {
  for (row = 0; row < 100; row++) {
    sum += table[row][column];
  }
}
```

# Code Tuning: Loops

**Putting the Busiest Loop on the Inside**

*Which is faster?*

```
for (row = 0; row < 100; row++) {
  for (column = 0; column < 5; column++) {
    sum += table[row][column];
  }
}
```

or

```
for (column = 0; column < 5; column++) {
  for (row = 0; row < 100; row++) {
    sum += table[row][column];
  }
}
```

Total loops executed: $600$ in first case, $505$ in second case.

*But what about cache performance?*

It's hard to say which is faster. You would have to *measure*.

# Code Tuning: Loops

**Strength Reduction**

Suppose you know that multiplication is much more expensive than addition. How could you rewrite this loop?

```
for (i=0; i < saleCount; i++) {
  commission[i] = (i+1) * revenue * baseCommission * discount;
}
```

# Code Tuning: Loops

**Strength Reduction**

Suppose you know that multiplication is much more expensive than addition. How could you rewrite this loop?

```
for (i=0; i < saleCount; i++) {
  commission[i] = (i+1) * revenue * baseCommission * discount;
}
```

You can replace multiplication with addition as follows:

```
incrementalCommission = revenue * baseCommission * discount;
cumulativecommission = incrementalCommission;
for (i=0; i < saleCount; i++) {
  commission[i] = cumulativeCommission;
  cumulativeCommission += incrementalCommission;
}
```

# Code Tuning: Data Transformations

**Use Integers Rather Than Floating-Point Numbers**

Integer addition and multiplication tend to be faster than floating point. Even changing a loop index from floating point to integer can save significant time.

**Use the Fewest Array Dimensions Possible**

If you can rewrite a multi-dimensional array as a one-dimensional array, it may save some time (at the expense of readability).

**Minimize Array References**

Every time you access an array, you have to do some pointer arithmetic and a dereference. If you can avoid it, this may speed things up. For example, you could store a commonly used array value in a temporary variable.

# Code Tuning: Data Transformations

**Use Supplementary Indexes**

- *String-Length Index*

  Instead of calling `strlen` every time you want to compute the length of a `char*`, store the length as an extra variable with the `char*`. The `std::string` class does this for you. You can apply this principle to other similar situations.

- *Independent, Parallel Index Structure*

  Suppose you have big data structures and each one is identified by a unique integer id. If you are searching or maniuplating the data structures, it may be easier to manipulate the indexes instead: Create an auxilliary data structure containing just the index and a pointer to the big data structure and manipulate that instead. This is a standard database technique.

# Code Tuning: Data Transformations

**Use Caching**

The idea behind caching is to save the result of common expensive computations so you don't have to compute them again.

For example, suppose you are computing the height of a binary tree:

```cpp
class Tree {
private:
  Tree* left;
  Tree* right;
public:
  ...
  int height(Tree* t);
};

int Tree::height(Tree* t)
{
  if (t->isLeaf()) return 1;
  else return 1+max(t->left()->height(),t->right()->height());
}
```

# Code Tuning: Data Transformations

*If we know there will be many calls to* `height`*, how can we speed up this routine?*

# Code Tuning: Data Transformations

*If we know there will be many calls to* `height`*, how can we speed up this routine?*

Here's one possible version, using caching:

```cpp
class Tree {
  ...
  int savedHeight;
public:
  Tree(Tree* l, Tree *r) : ... savedHeight(-1) {}
  ...
};

int Tree::height(Tree* t)
{
  if (savedHeight == -1) {
    if (t->isLeaf()) savedHeight = 1;
    else savedHeight =
      1 + max(t->left()->height(), t->right()->height());
  }
  return savedHeight;
}
```

# Code Tuning: Other Techniques

- Exploit algebraic identities

- Precompute values that are known at compile time

- Be wary of system libraries

- Use inlining

- Recode in a low-level language

33