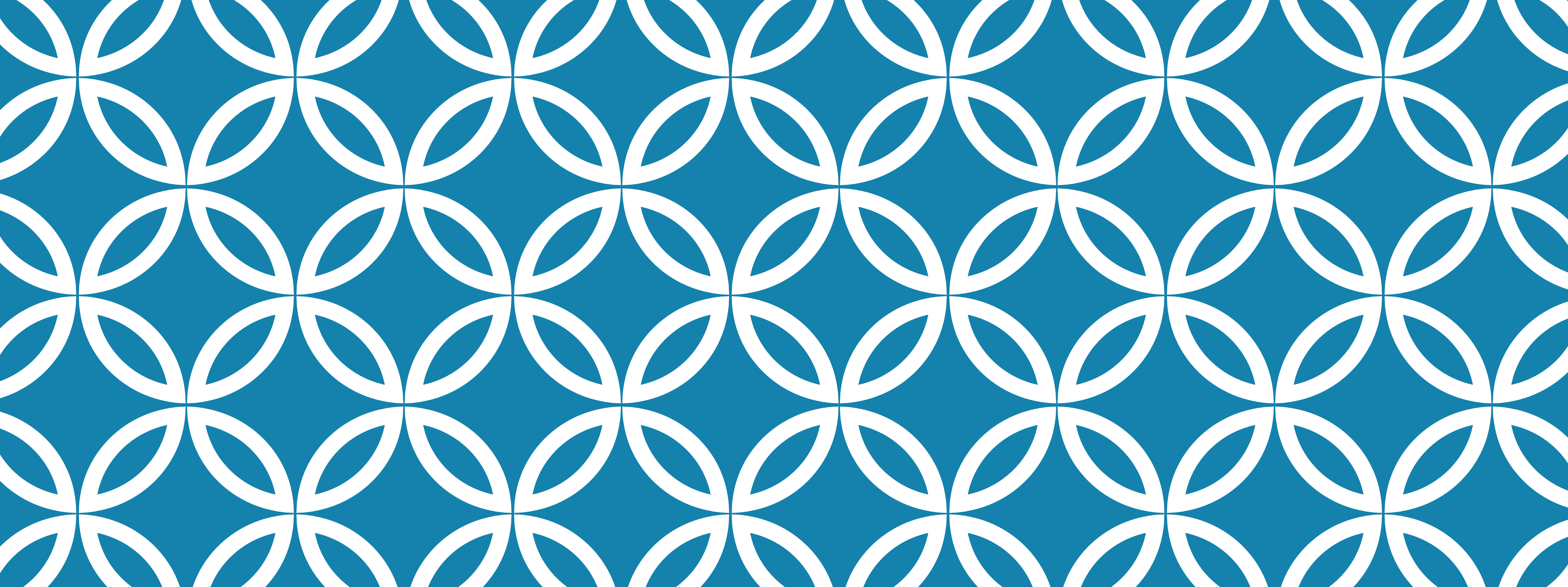


PROFILERS

Professor Ken Birman
CS4414 Lecture 11

OUR CHALLENGE TODAY

Can we pull these different threads together?

- We care a great deal about performance… but also elegance (and correctness, security, algorithmic efficiency)
- We are working with big systems that use big libraries
- The theme, throughout, centers on taking control of things: the hardware, Linux, the C++ compiler, your own code…

Can we visualize how all these elements interplay and use our vision to identify unexpected bottlenecks?

AN UNDERSTANDING OF CONST, CONSTEXPR AND TEMPLATES CLARIFIES PERFORMANCE ANALYSIS

Knowing how to visualize a program lets us assess performance:

- We need to learn how gprof and other profiling tools work, but then can use them to diagnose performance problems.
- With a clear mental image of how things *should* work, we can identify surprising overheads that may lead to insights about root causes of slow performance.

TOOLS VERSUS VISUAL THINKING

A tool is used to measure something. You learn some number.

The image in your mind shapes the questions you ask – the measurements you need. If you cannot visualize how something is working, you won't be effective at evaluating performance.

But sometimes you have "is it A or B?" questions. Tools can help!

IDEA MAP FOR TODAY

What determines performance?

90% - 10% rule

Visualize the expected behavior of your program!

Use performance analysis tools and methods to confirm your expectations or to discover surprises

Aim for the biggest wins with the smallest changes to the code

Should we do extensive hand-optimization, or can we shape the "system" to achieve our goals through elegant, higher-level methods?

RULES OF THUMB

Start by trying to understand the big picture!

There is always some big thing that dominates performance. Your job is simply to identify it. And this should be easy because whatever it is, the program spends a lot of time on it!

RULES OF THUMB

Simple suggestions often work well even if overly vague

Performance analysis is like measuring with your thumb: more like an art than a science!

TODAY: ARE THERE SIMPLE RULES FOR PERFORMANCE ANALYSIS?

Suppose you are given a big program written by a team. It has 25,000 lines of really hard to understand code.

Your job: it runs for 10 secs, and your boss thinks this is too slow.

How much speedup is possible? Could you get a 10x or better speedup? How will you approach this?

IDEA OF A SIMPLE RULE: HOW DO WE GET CODE TO COMPILE?

Always fix the first

line the compiler complains about!

A single issue can cause dozens or hundreds of error messages, so each compilation issue you fix might “clear” many more.

IDEA OF A SIMPLE RULE: HOW DO WE DEBUG OUR CODE?

Fix the very first thing that goes wrong in any execution.

Even if the first issue is “obscure”, fixing it will shed light on any systematic mistakes you might be making. It will also get rid of secondary effects.

Unit test all your components, one method a time. Issues with basic components can confuse you: you might think the bug is in the code *using* that method, not in the method itself.

SIMPLE RULE FOR PERFORMANCE ANALYSIS

Speed up the thing the program is doing the most.

Ignore the rest. Why mess with stuff that works?

25,000 LINES OF CODE…

Probably 250,000 machine instructions.

Your computer runs at about 1 billion instructions per second. So, each CPU can execute 10B instructions in 10s. Our word count program ran for about 13 seconds of user time.

But is it plausible that every line of the program was really executed 5.2M times?

TO CONSUME 13 SECONDS OF CPU TIME…

The program must be (1) in some form of loop, or (2) running a recursive algorithm that has high complexity. [Note: Some programs might also be (3) waiting for the kernel, a lot]

Unless the entire program is a massive loop, it is far more likely that just part of the program is responsible.

A great many studies confirm this intuition!

EVERY PROGRAM LOOKS SIMILAR WHEN VIEWED FROM A MILE UP

Most programs have a lot of non-executing code, and a lot of code used just when starting up, or just when printing output.

A program generally spends 90 to 99% of its compute time in just 10% or less of the code, and often much less.

BUT FIRST YOU NEED TO KNOW IF THE PROGRAM IS SPENDING A LOT OF WAITING

Linux has many tools that can help.

With “top” you can watch when the program is executing. Is it keeping one or more cores busy?

If a process is fully busy on 8 cores, it shows as 800% loaded

IOSTAT, VMSTAT

If a program runs for a long time but has low CPU utilization, it must be waiting for the Linux kernel to “do something”.

Iostat monitors I/O activity and can help you understand if your application is overwhelming the file system.

Vmstat monitors paging.   High paging rates will slow you down

EXOTIC TOOLS

Mpstat: “Multi-processor” statistics. Extremely useful for programs that are multi-threaded. Try “mpstat –P ALL”

Sar: This command is often used to create periodic reports that it will mail to the system administrator. For more automated uses.

EXOTIC TOOLS

CoreFreq: An Intel tool for collecting information about the cores on a NUMA machine, including cache stalls, instruction prefetch stalls, etc. Requires a special setup first, to disable a few Linux features that might otherwise confuse the output.

Htop: Similar to top, but some people prefer the output format, and it has a few options top lacks.

perf stat -p 2087

EXOTIC TOOLS

Perf: A bit like CoreFreq, but tracks a wide variety of process behaviors and reports on them. You can ask it to "focus" on a particular process this way:

perf stat -p 1234 // replace 1234 with the process-id

YOUR TASK?

First, understand if the process is genuinely busy. If it is, you need to find that core portion that loops so heavily.

Studies show that it will rarely be more than 10% of the code, and often might be as little as 1% of the code!

This will reduce your performance-optimization task to a set of 250 to 2,500 lines out of the original 25,000!

GPROF

If you compile your program with the –pg flag, and if it exits normally (main returns), a profile file will be written.

Then when you execute gprof myprog, grprof prints pages of output that can narrow the exact spot down in your code that spends so much time looping!

HOW IT WORKS

There are two aspects

- Linux kernel helps by using timers to build a histogram of where the PC pointed as the program executes.
- g++ -pg helps by including some additional method-call tracing data in a dedicated register. This allows it to count how often each method was called, by whom, and to compute the total time per call.

EXAMPLE OF A GPROF OUTPUT

In this example, Ns_DStringNAAppend is responsible for 17% of the total runtime

```
Flat profile:

 %      cumulative self    self    total
 time    seconds   seconds calls ms/call ms/call name
 17.7     3.72     3.72 13786208 0.00    0.00    Ns_DStringNAppend [8]
  6.1     5.00     1.28   107276 0.01    0.03    MakePath [10]
  2.9     5.60     0.60  1555972 0.00    0.00    Ns_DStringFree [35]
  2.7     6.18     0.58  1555965 0.00    0.00    Ns_DStringInit [36]
  2.3     6.67     0.49  1507858 0.00    0.00    ns_realloc [40]
```

GPROF ALSO TRACKS CALLER INFORMATION

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
....
```

ISSUE SPECIFIC TO C++

With C++ templates, the method names can become so long that gprof may sometimes have formatting issues or even crash!

The heavy use of libraries more or less guarantees that most time will be shown as being in various libraries. But you want to understand time spent in the user code that *called* these libraries, which is trickier!

EXAMPLE: FAST-WC

Ken and Sagar's word count programs are nearly impossible to profile with these tools.

Gprof sometimes even crashes due to the long symbol names!

But just the same, I tried a few different "configurations" and found a case where it was able to run…

SOME OF THE GPROF OUTPUT FROM FAST_WC

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 55.39      1.03     1.03                             wcounter(int)
 18.28      1.37     0.34      412   825.44   825.44  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, int>, std::_Select1st<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, int> >, std::less<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, int> > >::_M_erase(std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const, int> >*)
 15.60      1.66     0.29                             std::map<std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >, int, DefineSortOrder, std::allocator<std::pair<std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > const, int> > >::operator[](std::pair<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > const&)
```

(CLEANED UP TO BE LEGIBLE)

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
55.39	1.03	1.03				wcounter(int) // **Ken's word count main loop**
18.28	1.37	0.34	412	825.44	825.44	_M_erase(…)
15.60	1.66	0.29				operator[]()
6.99	1.79	0.13				M_erase(…)
2.15	1.83	0.04				__tcf_1
1.08	1.85	0.02				__tcf_0
0.54	1.86	0.01	1168849	0.01	0.01	M_get_insert_hint_unique_pos(…)
0.00	1.86	0.00	8	0.00	0.00	_M_get_insert_unique_pos(…)
0.00	1.86	0.00	1	0.00	0.00	_GLOBAL__sub_I_lock
0.00	1.86	0.00	1	0.00	0.00	M_get_insert_unique_pos(…)

... etc

EXAMPLE: FAST-WC

Fast-wc used a std::map, and the core operation was to increment the count in an element.

We actually see the calls to the indexing operation: "operator[]", although gprof doesn't have a count for how often it was called. (C++ library wasn't compiled with –pg!)

It looks like most of the time is spent in "word counter", which is good

WHAT ABOUT M_ERASE AND _M_ERASE?

These turn out to be methods used inside the std::map class, which uses a B+ tree data structure

M_Erase was a "wrapper" method that just called _M_Erase

Not called very often yet turned out to be 17% of the runtime for fast-wc. **By building my own home-made tree class I could probably get rid of that overhead… hmm…**

STATEMENT COUNTS

With the –A option, gprof will print line by line execution frequency… if it doesn't crash.

On a C++ program, however, this feature is unstable and may not produce any output at all.

```
               ulg updcrc(s, n)
               uch *s;
               unsigned n;
          2 ->{
                register ulg c;
                static ulg crc = (ulg)0xffffffffL;

          2 ->    if (s == NULL) {
          1 ->        c = 0xffffffffL;
          1 ->    } else {
          1 ->        c = crc;
          1 ->        if (n) do {
        756 ->            c = crc_32_tab[...];
756,1,756 1 ->        } while (--n);
                  }
          2 ->    crc = c;
          2 ->    return  c ^ 0xffffffffL;
          2 ->}
```

STATEMENT COUNTS

With the –A option, gprof will

tion

n't crash.

owever,

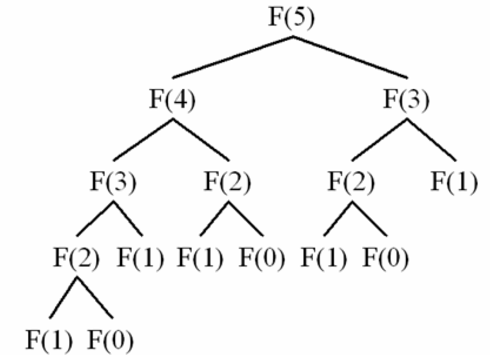
this feature is unstable and may not produce any output at all.

Tells us how many times the test was evaluated, how many times it was true, how many times it was false

```
            ulg updcrc(s, n)
            uch *s;
            unsigned n;
        2 ->{
             register ulg c;
             static ulg crc = (ulg)0xffffffffL;

        2 ->   if (s == NULL) {
        1 ->       c = 0xffffffffL;
        1 ->   } else {
        1 ->       c = crc;
        1 ->       if (n) do {
      756 ->           c = crc_32_tab[...];
756,1,755 1 ->     } while (--n);
               }
        2 ->   crc = c;
        2 ->   return  c ^ 0xffffffffL;
        2 ->}
```

THOUGHT QUESTION



int fibonacci(int n) { return n < 2? n: fibonacci(n-1)+fibonacci(n-2); }

This is a pretty inefficient recursion. But as a constexpr, called with a constexpr argument, C++ could compute it at compile time.

How would gprof output differ for the actual function versus the constexpr version? What about statement counts?

VALGRIND

A very popular tool because it reveals possible memory leaks (objects that were allocated but never properly freed). It also can create a profile of execution time, like gprof.

Use it this way: "valgrind --tool=callgrind ./(Your binary)"

"callgrind_annotate callgrind.out.pid" shows the output collected

GOOGLE PROFILING TOOLS

Google has a new collection of powerful tools called

google-perftools

These seem quite popular, but we haven't tried them at Cornell.

OTHER OPTIONS

In a debugger, just pause the program a few times.

If it really spends 90% of its time in some part of the code, that part of the code will be where it stops!

This can work when all else fails…

OTHER OPTIONS

You can also just insert code to time chunks of your logic:

1) Call the Linux "gettimeofday" method
2) Run the logic you are timing [perhaps: "many times"]
3) Call gettimeofday again, and subtract (1)

Big advantage is that this method won't cause any slowdown.

WHAT DO YOU DO WITH THIS RAW DATA?

The intellectually interesting task is to match what you observe against what you would anticipate based on understanding the program.

So there is always a back-and-forth:

- How did I design this to work?
- How is it actually working, and are there any big surprises?

WITH THESE REPORTS…

You'll see which methods are consuming most of the time

But also, who is calling those methods. The issue is often a higher level method that doesn't use a lot of time "itself" but actually accounts for almost all the time when you also include its children.

Gprof is showing you exactly this information!

ONCE YOU'VE NARROWED IT DOWN

At this point, you do need to read the code (even if you didn't write it), to understand what the method is trying to do.

Sometimes you'll quickly realize that this code was poorly written. For example, it might be looking up the same information again and again in a list – a simple "memoizer" that caches some recent results can have a magical impact.

PERHAPS C++ NEEDS SOME HELP…

Recall our discussion about the vectorization features of C++

For those to work, your code needs to have a very clean structure, loops need to have "fixed" termination bounds, etc.

Often code can be rewritten, just a little, and will suddenly run much faster (remember to use –O3 when compiling)

THINK OUT OF THE BOX!

The developers of this code probably were using C++ intelligently, but they may never have realized that so much time would be spent at this one spot.

Once you really understand what this code is doing, you can ask whether it is doing it in the fastest possible way.

MANY THINGS CAN BE UNEXPECTEDLY COSTLY IN AN OBJECT-ORIENTED LANGUAGE

Objects might be getting created and freed very frequently.

Constructors might be more costly than you realize (and maybe more costly than needed)

A developer might be using some sort of standard pattern in a very inefficient way

MORE BIG PICTURE THINGS TO CONSIDER

Perhaps the larger code structure doesn't even need to be doing this expensive thing.

Sometimes people take shortcuts when coding without thinking much about it, and later those pieces of code get used heavily.

A relatively minor thing the developer didn't have time to do could make a huge difference.

THIS IS WHY LOOKING UP AT THE CALL STACK REALLY MATTERS!

Sure, a mysterious method named Ns_DStringNAppend is consume a lot of compute time.

The puzzle is that this might be internal to the C++ std libraries. But when you look up the calls stack a few levels, you see code from this program that is responsible for those calls.

EXAMPLE OF THE KIND OF THING WE HAVE IN MIND

In a photo processing application, it could be tempting to treat each pixel of the image as an object with RGB or HCL values.

Then an operation like "rotate and tilt" could be expressed very elegantly. But the resulting code will be accessing fields of these objects frequently…

REMEMBERING THE INTEL ADVICE ON VECTORIZATION

To vectorize nicely, we need dense arrays for each color. For example, a 760 x 1024 image could be represented as 3 matrices each for a distinct color in the RGB case.

Then we would have dense arrays with clean array indexing. Our loops might vectorize extremely well.

GHASTLY CODE EDITING!

This representation (as objects) will be used everywhere in the photo processing application

Sounds like we would need to rewrite all accesses to the photo in the image processing application.

… *or does it?* Any ideas?

… AN IDEA!

What about keeping the existing objects, but changing them into getter/setter objects that access into these 3 matrices?

Now the existing code will all continue to work! But we also have this great "raw" representation available

For very costly actions, we could recode against the raw data.

A SECOND, SIMILAR IDEA

We could also construct the dense array data when we need it, which won't be incredibly slow if it occurs rarely.

Then we can operate on it to do our expensive image operation.

At the end, we could "copy the results" back into the objects.

HOW TO DECIDE?

Adopt an approach that minimizes your effort.

Aim for the really, really expensive "thing" and optimize at that level.

Had we dived deep and optimized Ken's wcounter main loop, and perhaps figured out what causes M_Erase to be called, his word-count program could have been even faster!

SUMMARY

```
class GameOverScene : public cocos2d::CCScene {

public:
    GameOverScene():_layer(NULL) {};
    ~GameOverScene();
    bool init();

    //SCENE_NODE_FUNC(GameOverScene);

    static GameOverScene* node()
    {
        GameOverScene *pRet = new GameOverScene();

        //Error: undefined reference to `GameOverScene::init()'
        if (pRet && pRet->init())
        {
            pRet->autorelease();
            return pRet;
```

C++

In many courses you learn one technique, or one tool, and it is separable from other techniques or other tools.

In systems programming, our challenge is that we are really programming "the system" – which has many layers, including the C++ compiler, the STL, the Linux kernel, and even hardware elements like the network interface or the storage unit.

A true master of systems programming is able to visualize all these moving parts, then can use tools like gprof to "zero in" on issues