Code::Dive

"...and why you care"

Bartek Wiklak i Krzysiek Czaiński

Principles

from the presentation by Venkat Subramaniam

Who writes bad code?

- Me!
- Everyone!
- But I noticed I'm good at spotting what's bad in someone else's code.
- So I'll just write my bad code and then review someone else's code, while waiting for the review of my code.
- But what is "bad" code?

Rules we can agree on
and then communicate in review easier

DRY

Don’t Repeat Yourself

Not just copy-paste. It’s duplicating effort we want to avoid.

- Solving a problem, that is already solved violates DRY.
- Having to make one change in many places violates DRY.

removing and erasing

```
auto pred = [](auto x) { …
}
for (int i = v.size() - 1; i >= 0; --i) {
  if (pred(v[i]))
    v.erase(v.begin() + i);
}
```

```
1 v.erase(remove_if(v.begin(), v.end(), pred), v.end());
```

```
2 boost::remove_erase_if(v, pred);
```

Remove-erase is an established idiom in C++. Don’t write the loop yourself, because you might do it poorly.
address_search_provider_view.cc

Similar algorithm rewritten

```
int TabStripBaseView::GetPinnedTabCount() const {
  int pinned_tabs = 0;
  for (auto tab = tabs_.begin(); tab != tabs_.end() && (*tab)->IsPinned();
       ++tab) {
    ++pinned_tabs;
  }
  return pinned_tabs;
}
```

```
1 count_if(tabs_.begin(), tabs_.end(), [](auto tab){ return tab->IsPinned(); });
```

```
2 count_if(tabs_, [](auto tab){ return tab->IsPinned(); });
3 size(tabs_ | filtered([](auto tab){ return tab->IsPinned(); });
```

The same for count_if

FindActiveModalDialog

```
// FindActiveModalDialog(ui::ModalType modal_type, bool has_to_be_visible)
for (std::list<DialogBaseView*>::const_iterator i = dialog_view_list_.begin();
    i != dialog_view_list_.end();
    ++i) {
  views::Widget* dialog_widget = (*i)->GetWidget();
  if (dialog_widget && (!has_to_be_visible || dialog_widget->IsVisible())) {
    if ((*i)->GetModalType() == modal_type)
      return *i;
  }
}
return NULL;
```

First, rage based for.

FindActiveModalDialog (2)

```
// FindActiveModalDialog(ui::ModalType modal_type, bool has_to_be_visible)
for (const auto dialog : dialog_view_list_) {
  views::Widget* dialog_widget = dialog->GetWidget();
  if (dialog_widget && (!has_to_be_visible || dialog_widget->IsVisible())) {
    if (dialog->GetModalType() == modal_type)
      return dialog;
  }
}
return nullptr;
```

Do you see the predicate here?

FindActiveModalDialog (3)

```
auto pred = [=](auto dialog) {
  views::Widget* dialog_widget = dialog->GetWidget();
  return dialog_widget &&
         (!has_to_be_visible || dialog_widget->IsVisible()) &&
         dialog->GetModalType() == modal_type;
};

for (const auto dialog : dialog_view_list_) {
  if (pred(dialog))
    return dialog;
}
return nullptr;
```

Oh! that’s something we are familiar with!

FindActiveModalDialog (4) - Find if

```
auto pred = [=](auto dialog) {...

for (const auto dialog : dialog_view_list_) {
  if (pred(dialog))
    return dialog;
}
return nullptr;

auto found_it = find_if(dialog_view_list_.begin(), dialog_view_list_.end(), pred);
// auto found_it = boost::find_if(dialog_view_list_, pred);
return found_it != dialog_view_list_.end() ? *found_it : nullptr;
```

Again, use algorithms.

child_at ;-(

```
for (int i = index - 1; i >= 0; --i) {
  if (address_bar_->child_at(i)->visible()) {
    return address_bar_->child_at(i);
  }
}
```

```
int child_count() const { return children_.size(); }

const View* child_at(int index) const { // DCHECK_GE, DCHECK_LT
  return children_[index];
}
View* child_at(int index) {
  return const_cast<View*>(const_cast<const View*>(this)->child_at(index));
}
```

It’s sad, that someone decided to invent a nonstandard interface for iterating over children (incompatible with standard and other ready-to-use algorithms).
addressbar_border_manager.cc

YAGNI

You Ain’t Gonna Need It (Yet)

Don’t do work you think you might need later. Instead, do work you know you need soon. You’ll avoid a lot of useless work.
We seem to be pretty good at this in Opera.

SRP

Single Responsibility Principle

- short functions
- SLAP
  - Single Layer of Abstraction Principle

(eg. “Or” or “And” in function name)

We have so many way-too-long functions, I won’t even show an example.
When trying to think of a good name for something, and the name you come up with contains “Or” or “And”, you’re probably violating SRP.

OCP

Open Closed Principle

All entities should be:

- open for extension,
- closed for modification.

OCP, given:

```
// Access to generated version info about the current build.
class VersionInfo {
  enum Channel {
    CHANNEL_UNKNOWN, …, CHANNEL_DEV, CHANNEL_BETA, CHANNEL_STABLE
  };

  Channel GetChannel();

  // "developer", "beta", "Stable"
  std::string GetVersionStringModifier() const;

  // Whether this is a Stable "official" release build.
  bool IsOfficialBuild() const;
```

Suppose we are given this class VersionInfo, and...

Get URL with product info (OCP ✗)

```
string url = "https://download.opera.com/";
VersionInfo version_info;

switch (version_info.GetChannel()) {
  case CHANNEL_DEV:    url += "&product=Opera%20developer"; break;
  case CHANNEL_BETA:   url += "&product=Opera%20beta";      break;
  case CHANNEL_STABLE: url += "&product=Opera";             break;
  default:             NOTREACHED();                       break;
}
```

… we need to compute an URL with our product name. If we just write it like this, we violate OCP.

- It isn't open for extension: what if we want to append product info to other URLs?
- It isn't closed for modification: this switch requires modification if VersionInfo later supports another CHANNEL (also DRY).

Append URL with product info (OCP ✓)

```
string AppendUrlWithProductInfo(string url)
  string url = "https://download.opera.com/";
  VersionInfo version_info;

  switch (version_info.GetChannel()) {
    case CHANNEL_DEV:    url += "&product=Opera%20developer"; break;
    case CHANNEL_BETA:   url += "&product=Opera%20beta";      break;
    case CHANNEL_STABLE: url += "&product=Opera";             break;
    default:             NOTREACHED();                       break;
  }
  url += "&product=Opera"
  if (!version_info.IsOfficialBuild())
    url += "%20" + version_info.GetVersionStringModifier();
  return url;
}
```

To fix the problems:

- make it a function taking the URL,
- avoid switch here, because VersionInfo already does the switch.

LSP

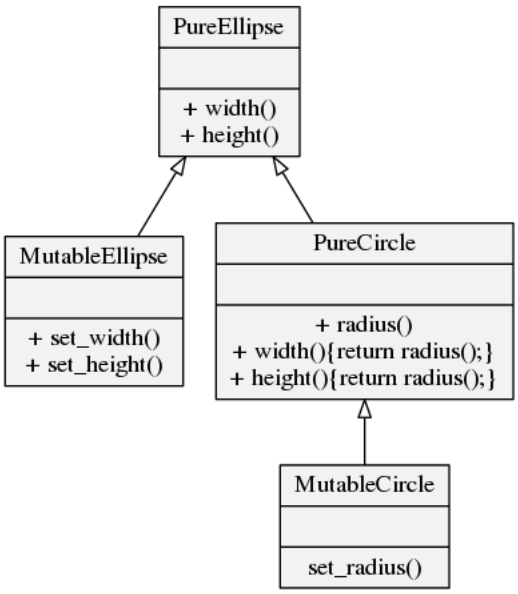
Liskov Substitution Principle

- BagOfApples is a (kind of) BagOfFruits **?**
- Circle is a (kind of) Ellipse **?**

The derived class objects must be *substitutable* for the base class objects.

A container of Thing is *not* a kind-of container of Anything even if a Thing is a kind-of an Anything
What if Ellipse derives from Circle? - what if Circle has radius() method? Ouch!

LSP: a PureCircle is a PureEllipse



Inheritance is the base class of all evil.
-- Sean Parent

I just couldn't stand the fact, that in math a Circle is an Ellipse, but not in programming. But when you think about it, an Ellipse in programming is actually a MutableEllipse. And now it is no surprise that a MutableCircle is not a MutableEllipse.

-- diagram source:

```
digraph hierarchy {
size="5,5"
node[shape=record,style=filled,fillcolor=gray95]
edge[dir=back, arrowtail=empty]

Ellipse[label = "{PureEllipse|| + width()\n+ height()}"]
Circle[label = "{PureCircle|| + radius()\n+ width()\{return radius();\}\n+ height()\{return radius();\}}"]
MutableEllipse[label = "{MutableEllipse|| + set_width()\n+ set_height()}"]
MutableCircle[label = "{MutableCircle||set_radius()}"]

Ellipse->MutableEllipse
Ellipse->Circle
Circle->MutableCircle
}
```

Principles summary

DRY,
YAGNI,
SRP, SLAP,
OCP,
LSP
Let’s use these names during code reviews.

AAA club

almost always auto

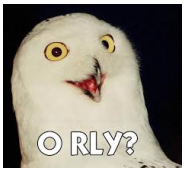
Join the club with Scott Meyers and Herb Sutter… maybe.

auto (1)

```
auto const widget = GetWidget();
DoSomething(widget);
```

```
DoSomething(GetWidget());
```

```
Widget* const widget = GetWidget();
DoSomething(widget);
```



Ok.
Now what if widget may be null...

If you find auto less readable tham Widget*, then why do you find the one-line version readable?
In some sense, actually naming the pointer type Widget* makes it less readable. GetWidget and DoSomething are from the same framework. DoSomething can operate on whatever GetWidget returns. We don't care what the actual pointer or smart-pointer type it is, we just want to pass it from GetWidget to DoSomething.
And that's why the one-line version we are used to is good. That is also why the auto version is good.

auto (2)

```
auto const widget = GetWidget();
if ( widget )
  DoSomething(widget);
```

- Meh…
- Null widget lives outside the if...

```
if (auto const widget = GetWidget())
  DoSomething(widget);
```

- *But isn't this an assignment inside an if statement?*

~~Widget* widget;~~
~~if (widget = GetWidget())~~
~~ DoSomething(widget);~~

- Prefer variables to live as short as possible. Narrow down their scope. Declare late, and make them go out of scope as soon as not needed.
- Initializing inside if () is a very elegant and readable way of scoping the variable. It's easy to get used to. Please use it!

auto: declaring variables

```
// Classic C++ declaration order
const char* s = "Hello";
widget w = get_widget();

employee e{ empid };
widget w{ 12, 34 };
```

```
// Modern C++ style
auto s = "Hello";
auto w = get_widget();

auto e = employee{ empid };
auto w = widget{ 12, 34 };
```

```
auto lock = lock_guard<mutex>{ m };  // error
auto ai   = atomic<int>{};           // error
```

```
Matrix a, b;      // some type that does lazy eval
auto ab = a * b; // to capture the lazy-eval proxy
auto c = Matrix{ a * b }; // to force computation
```

but this is a `static_cast<Matrix>`!

http://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

Problems i found with Herb Sutters AAA style are in red frames

auto, but don’t hide static_cast

```
Matrix a;
auto x = 3;
auto da = a.size();
auto c = Matrix{ x * da }; // oops, ment { x * a }
```

- this will compile, assuming `Matrix` has an `explicit` constructor:
  `explicit Matrix(size_t size);`
- but this won’t compile:
  `Matrix c = x * da;`

Always use the weakest appropriate cast possible. Prefer implicit cast to static cast. (Prefer static cast to reinterpret cast. Never use C-style casts.)

auto: declaring variables - summary

```
// Classic C++ declaration order
const char* s = "Hello";
widget w = get_widget();

employee e{ empid };
widget w{ 12, 34 };
```

```
// Modern C++ style
auto s = "Hello";
auto w = get_widget();

auto e = employee{ empid };
auto w = widget{ 12, 34 };
```

```
auto const size = checked_cast<int>(v.size());
auto const npos = static_cast<uint32_t>(-1);
```

http://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

This is my choice of preferred initialization syntax.

- auto s = “Hello” - because I just want a string literal handle, and don’t care about its’ type;
- auto w - get_widget() - because DRE;
- when I want to commit to a type explicitly, I don’t want to confuse readers with auto, that I deduce something; besides, let’s be consistent with old code with this;
- when casting, DRE - use auto so as not to state the result type twice.

auto - details

```
auto [const] x = f();
```

- \+ it's safest to force a copy;
- \- if `f()` is a member function - getter - returning by `const&` to avoid a copy, this introduces a copy.

```
auto [const]& x = f();
```

- \+ proper solution if `f()` returns a reference;
- \- won't compile if `f()` returns by value.

```
auto&& x = f();
```

- \- use rarely in generic context;
- \+ doesn't force a copy, but works in both cases: return by value or ref.

```
decltype(auto) x = f(); // C++14
```

- you don't want to know… do you?

decltype(auto)

```
template<class Cont, class Idx>              // C++14
decltype(auto) grab(Cont&& cont, Idx&& i) {
  authenticateUser();
  return std::forward<Cont>(cont)[std::forward<Idx>(i)];
}
```

http://scottmeyers.blogspot.com/2013/07/when-decltype-meets-auto.html

auto - gotchas

```
auto const i = {5};  // what is the type here?

for (auto const x : vect) { ...  // Ouch, if copy is expensive!

auto const name = widget.name(); // Ouch, copy
```

`i` is an initializer_list<int>

why auto?

- safer

```
int k;
auto k = 9;  // can't declare uninitialized

unsigned n = a_vector.size(); // 64 bit vs 32 bit
```

- more efficient

```
function<int(int)> lin = [a,b](int x){ return a*x + b;}
auto lin = [a,b](int x){ return a*x + b;}

std::map<string,int> map;
for (std::pair<string,int> const& x : map ) { … // Ouch
```

- more maintainable
- more readable
- (less typing)

- n has a bogus value if a_vector has more elements that 2**32 (if that actually can happen in practice, the tests probably won't catch this);
- to stroe a lambda for later without auto, you need a function<>, which adds a level of indirection and may allocate;
- map::value_type doesn't match the for-loop's x, so there will be a copy upon each iteration;
- maintainable - the type can change as long as it can be used the same way;
- more readable - when we don't care what the actual type is

auto - summary

- Use auto, but think what you're doing (just like without auto).

```
auto x = ...;
auto const x = ...;

for (auto x : range) { ...
for (auto const x : range) { ...
...

auto const s = "Hello";
auto const w = get_widget();
```

```
auto& x = ...;
auto const& x = ...;

for (auto& x : range) { ...
for (auto const& x : range) {


employee e{ empid };
widget w{ 12, 34 };
```

```
auto const size = checked_cast<int>(v.size());
auto const npos = static_cast<uint32_t>(-1);
```

- …that said, readability first!

- Use auto or auto& whenever appropirate. Understand, which one you need and why.
- Always use auto& in range for loops.
- Use auto when you don't care what the actual type is.

Const (principle)

- prefer immutable instances
- use `const` **everywhere you can**
- (prefer `constexpr` to `const`)

mutable vs. immutable instances

```
auto w = GetWidget();
if (some_cond)
  w = GetOtherWidget();
```

```
auto const w = some_cond ? GetOtherWidget() : GetWidget();
```

Const before or const after?

- let the flame wars begin…

```
const auto size = v.size();      auto const size = v.size();
const int* p = &x;               int const* p = &x;
???                              int* const p = &x;

                 typedef int* PInt;
const PInt p = &x;               PInt const p = &x;
const int* p = &x;               int* const p = &x;

                    int      *      a[5];
              [const]int[const]*[const] a[5];
```

Which is the “right” way?

- const before?
- const after?
- const consistently?
- doesn’t matter?

I think “const before” is the only wrong answer. The only way to be consistent is “const after”, because you can’t always use “const before”.
In C++ you don’t read type declarations from left to right. You read them starting from the middle and going left/right according to operator’s priority.
‘a’ is an array of 5 [const] pointers to [const] int.

To const or not to const?

Mark `const` all *variables*, that aren't going to change.

- locals:
```
auto const size = v.size();
int const x = (any_expr);
Widget const* const widget = GetWidget();
```
- passing by &:
```
void f(Widget const& w);
```
- member functions:
```
int get_x() const;
```

Marking things const

1) documents that it won't change,
2) compiler enforces that.

To const or not to const? Exception

Except, don't mark `const`:

- Nontrivial locals, that are going to be moved from or returned by value:

```
auto ~~const~~ w = MakeWidget();
LOG(INFO) << w;
return w;
```

- By-value return types

const kills move

To const or not to const? By value

Are these 2 distinct overloads of function `f()`?

```
void f(int x, Widget w, Gadget g);
void f(int const x2, Widget const w2, Gadget g2);

void f(int const x, Widget w, Gadget g) {
  w.set_x(x);
  GadgetSink(std::move(g));
}
```

For **by value** parameters, I say:

- use `const` in the implementation only (.cc files);
- don't use `const` in the interface (declarations or in-class functions).

Top-level const of function argument types is ignored just like the argument names, so don't use it in declarations.
However, (not only) top-level const of function argument types matters in the functions implementation - just like the argument names, so use const there.

Const (principle) summary

- use const wherever possible,
  - except returned/moved-from locals,
  - except by-value return types,
- const after is better than const before,
  - but any const is way better than no const,
- if you can make something immutable without loss of clarity, do it and mark it const.

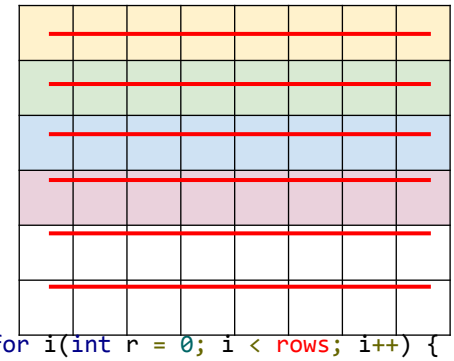
CPU Caches

and why you care

from the presentation by Scott Meyers

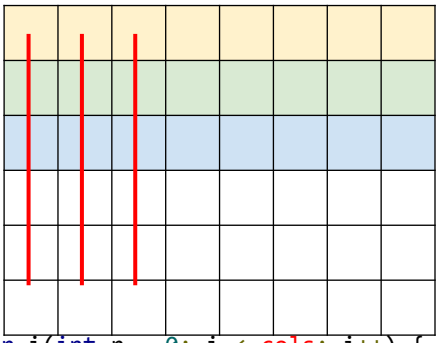
Simple problem - Matrix traversal

ROW MAJOR



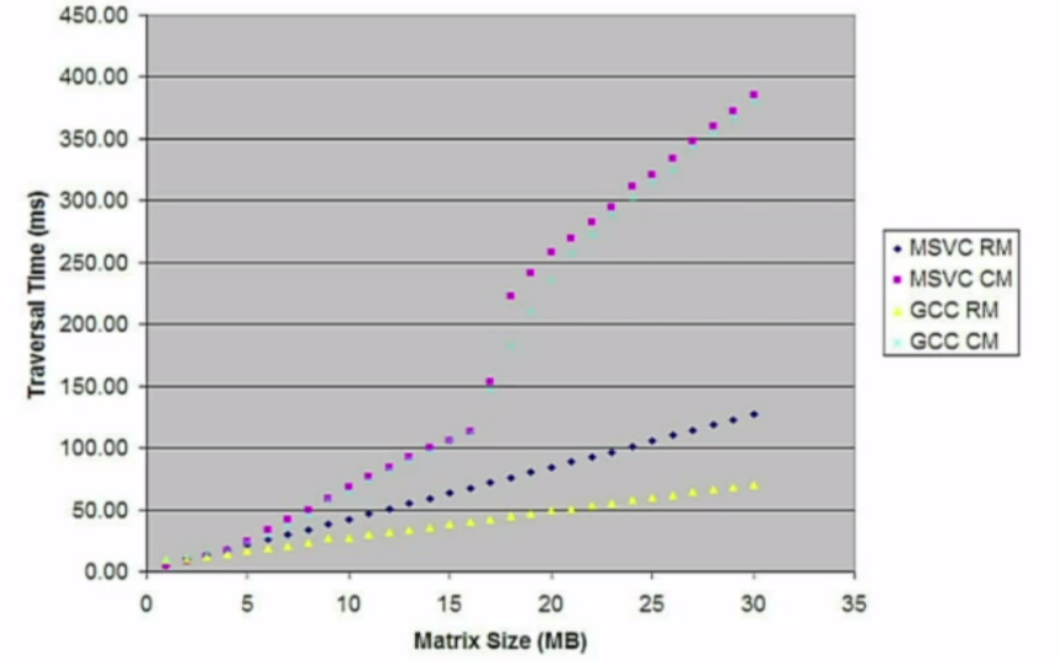
```
for i(int r = 0; i < rows; i++) {
  for (int c = 0; j < cols; j++) {
    auto data = vec[r][c];
  }
}
```

COLUMN MAJOR



```
for i(int r = 0; i < cols; i++) {
  for (int c = 0; j < rows; j++) {
    auto data = vec[r][c];
  }
}
```

- The same algorithm for both situations.
- We hit the same amount of memory the same number of times.
- Both algorithms have the same complexity.



Scott Meyers “Cpu caches and why you care”

Yet the results are strikingly different!
What’s more: for two independent compilers we’re getting similar strange behaviour.

BUT WHY?

Jeff Dean's Numbers every engineer should know (revised)

- L1 cache reference ~4 cycles - ~1 ns
- L2 cache reference ~11 cycles - 3 ns
  - Branch mispredict 5 ns
- L3 cache reference ~39 cycles - 12 ns 9xL2
- Main memory reference **107** cycles 27xL1
  - Mutex lock/unlock 100 ns
  - Compress 1K bytes with Zippy 10,000 ns
  - Send 2K bytes over 1 Gbps network 20,000 ns
  - Read 1 MB sequentially from memory 250,000 ns
  - Round trip within same datacenter 500,000 ns
  - Disk seek 10,000,000 ns
  - Read 1 MB sequentially from network 10,000,000 ns
  - Read 1 MB sequentially from disk 30,000,000 ns
  - Send packet CA->Netherlands->CA 150,000,000 ns

Caches are very fast...

Intel Core i7-9xx

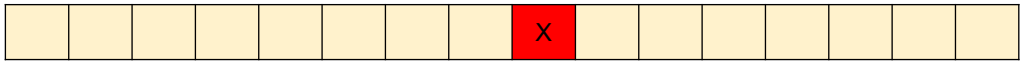
- 32KB L1 D-Cache, 32KB I-Cache per core
- 256KB L2 cache per core (I and D)
- 8MB L3 cache per 4 cores

But also very small.
D-Cache - is data cache
I-Cache - is instruction cache

Cache line

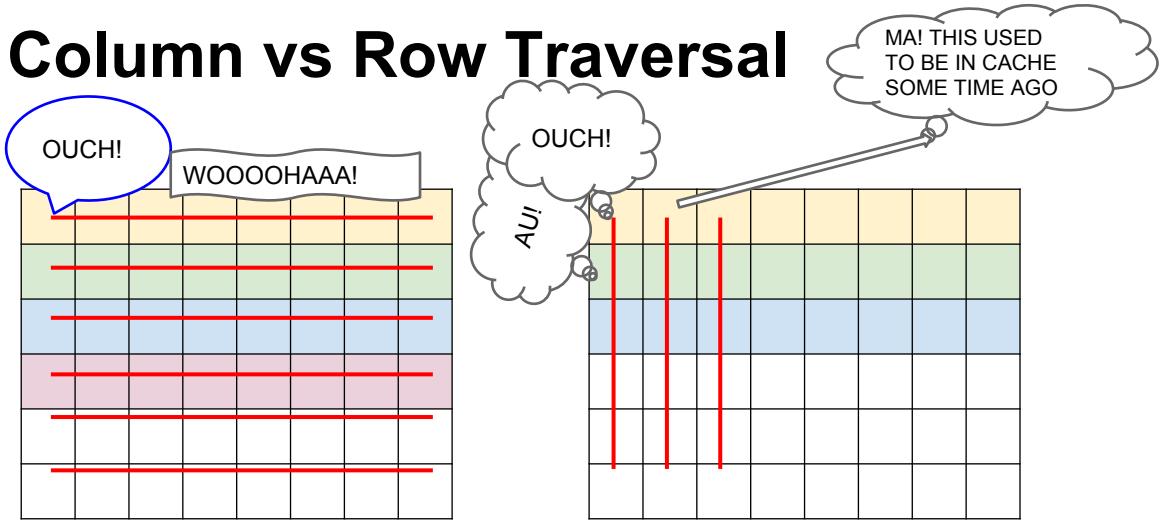
On Core i7 64 bit line

16 32bit values



Caches are acquired in 64bit cache lines. When you refer a bit X whole 64 bit line is placed into the cache.
That means you have all of them “for free”. All you have to do is to take advantage of them.

Column vs Row Traversal



That's why column traversal differs so much from row traversal.
In row traversal, when you refer first byte in a row, you have next 16 bytes "for free".
On the other hand in column traversal you hit a cache miss every step.
You could do a lot of things during these 107 cycles!

Lessons learned

- Locality counts
- Predictable access patterns count
- Linear traversals are very fast
  - linear search can beat $log_2 n$ heap alloc BSTs
  - binary search on sorted array can beat O(1) of heap based hash tables
  - hardware cache wins for small n
- Small=Fast (insertion sort vs QS)

Size does matter

Multiple threads

http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206

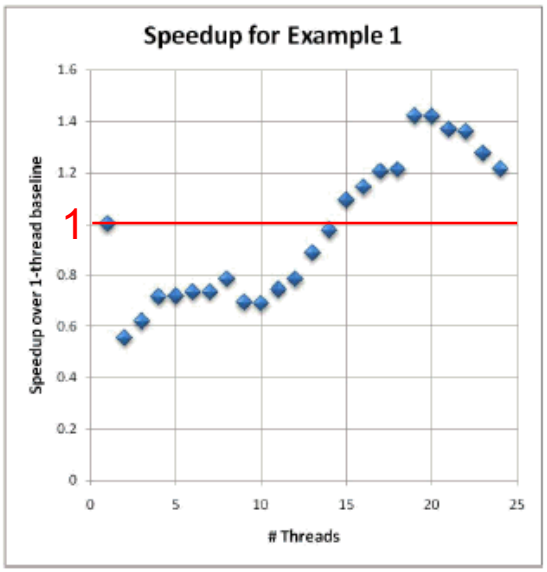
```
int result[P];

for( int p = 0; p < P; ++p )
  pool.run( [&,p] {
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p];
  } );
pool.join();

odds = 0;
for( int p = 0; p < P; ++p )
  odds += result[p];
```

Simple algorithm, we just read the data from matrix and count the number of off odd elements.
Perfect candidate for parallelization. No data sharing, no mutexes.

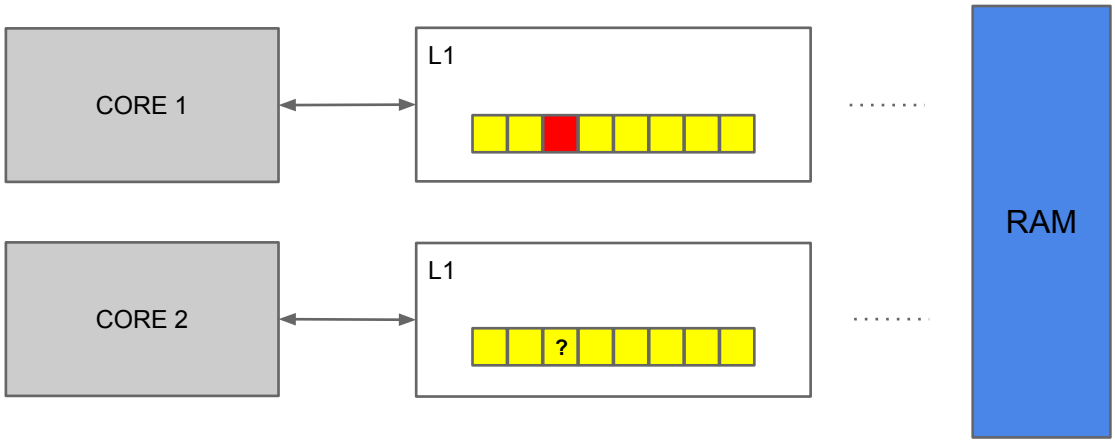
Multiple threads



http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206

The performance and scalability is rather depressing… but why?

Cache coherency

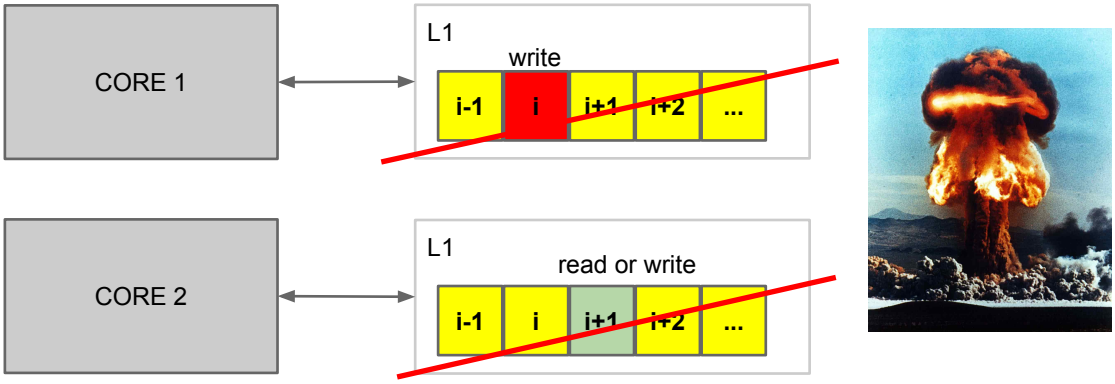


Imagine you have the same cache line on multiple cores, and one core mutates a byte in this cache line.
What happens?

Cache coherency

The short answer is: you don’t have to worry, through the “Magic of Hardware” all cache lines will be marked “dirty” and the hardware will fetch correct cache line for all threads.

Cache coherency



“You don’t have to worry” - probably …

Imagine that many threads on multiple cores what the read some value $k \neq i$, $|k - i| < \varepsilon$ that happens to be in the same cache line.
If just one thread mutates value $k = i$, all other cache lines are marked dirty and must be re fetched.
This is horrible waste of cache speed.

Now, let’s revise our first solution, we had a small vector `result` and all threads mutated it aggressively. It doesn’t matter that none of them shared the same value in it.

False sharing

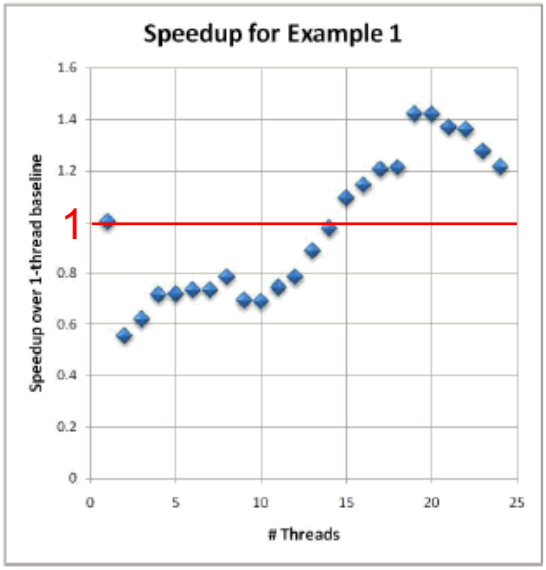
```
pool.run( [&,p] {
  result[p] = 0;
  int count = 0;
  int chunkSize = DIM/P + 1;
  int myStart = p * chunkSize;
  int myEnd = min( myStart+chunkSize, DIM );
  for( int i = myStart; i < myEnd; ++i )
    for( int j = 0; j < DIM; ++j )
      if( matrix[i*DIM + j] % 2 != 0 )
        ++count;
  result[p] = count;
}
```

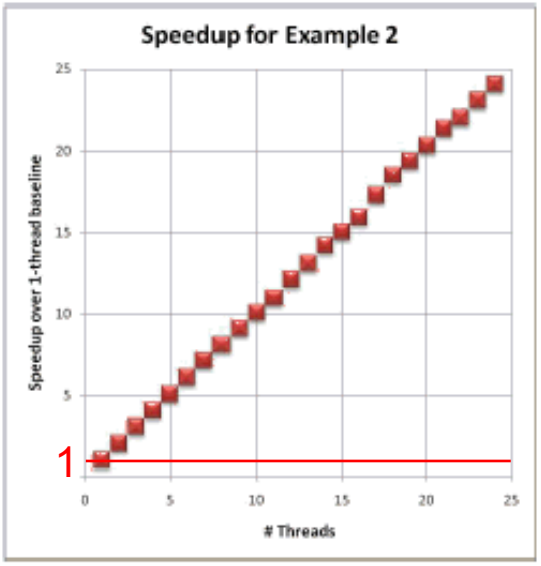
In this example there's rather easy solution to this problem.

To eliminate false sharing we introduce a local variable `count` and we update the `result` vector only once for thread.
In terms of computational complexity we actually do **more** work.

False sharing

http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206





Yet the result is perfect scalability.

More things we can do

- separate frequently used class members from those use rarely (data driven progr.)
- be nice for instruction cache
  - avoid branching
  - sort vector of pointers to derived classes by type AAABBBC
- PGO & WPO

Cache associativity

```
void UpdateEveryKthByte(std::array<uint8_t,N>& arr, int const k) {
    auto const kNumOfIterations = 1024*1024;
    int p = 0;
    for (int i = 0; i < kNumOfIterations; i++) {
        arr[p]++;
        p += k;
        if (p >= arr.size()) p = 0;
    }
}
```

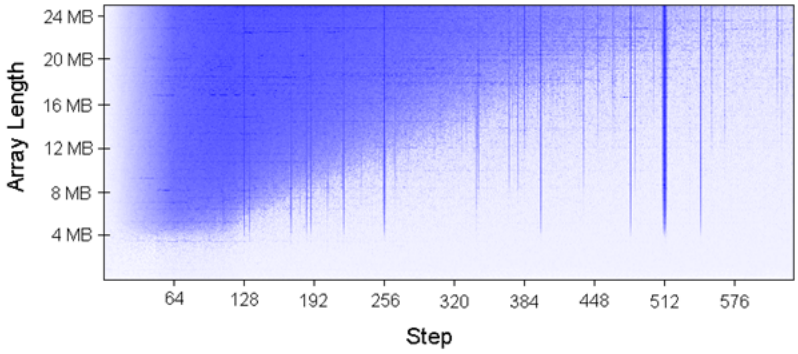
http://igoro.com/archive/gallery-of-processor-cache-effects/

Our mental model for how caches work is wrong.

Let’s make a simple test, we will mutate every k’th element of the array.

Cache associativity

blue - slow, white-fast



http://igoro.com/archive/gallery-of-processor-cache-effects/

Some steps are really bad.
Cache is divided into N slots,
Imagine a programmer who wants to choose a constant for arr size, what number would it be (256, 512)?

Simple example from our codebase

```
auto it = find(list.begin(), list.end(), index);
if (it != list.end())
  list.erase(it);
list.push_front(index);    // O(1)

// USE VECTOR
auto it = find(vect.begin(), vect.end(), index);
if (it != vect.end())
  vect.erase(it);
vect.insert(vect.begin(), index); // O(n) - don't care
```

Use rotate algorithm

```
auto it = find(vect.begin(), vect.end(), index);
if (it != vect.end())
  rotate(vect.begin(), it, next(it));
```

Very simple example from our code - use vector instead of list and prefer algorithms over explicit step by step code.
Computational complexity of using vector is O(n) here, for list it's just O(1). For small vectors (e.g. < 100 ints) don't bother!

```
list<int> l = { 1, 2 … 26 };  vector<int> v = { 1, 2 … 26 };
using clock = chrono::high_resolution_clock;
auto elapsed = [](clock::duration d) {
  return chrono::duration_cast<chrono::milliseconds>(d).count();
};
auto startL = clock::now();
for (long i = 0; i < 10000000; i++) {
  int b = l.back();
  auto it = find(l.begin(), l.end(), b);
  l.erase(it); l.push_front(b);
}
auto endL = clock::now();
cout << "duration list: " << elapsed(endL - startL) << "ms" << endl;

auto startV = clock::now();
for (long i = 0; i < 10000000; i++) {
  int b = v.back();  // worst case for vector
  auto it = find(v.begin(), v.end(), b);
  rotate(v.begin(), it, next(it));
}
auto endV = clock::now();
cout << "duration vector: " << elapsed(endV - startV) << "ms" << endl;
```

1653 ms

967 ms

Do not believe me, measure! I prepared a simple program to measure this two data structures. Run it yourself!

Error reporting

from the presentation by Andrzej Krzemieński

```
double CalculateWeight() {
  if (auto impl = GetImpl()) {
    return impl->CalculateWeight();
  } else {
    // TODO handle error condition
  }
}
```

This function's purpose is to return the weight of a loaded aircraft. It's part of a module used before the aircraft is allowed to take off. If it fails to give the answer, the crew will have to check it manually.

the TODO: It's unspecified/undefined behavior, but it's OK, we'll come back to the error condition when we have time later. Yeah, right ;-)

```
constexpr double kMaxWeight = 1000;

bool IsTooHeavy() {
  return CalculateWeight() > kMaxWeight;
}
```

- Tests pass.
- Time to ship it.
- Let's just search for TODO's.

Wow, we almost shipped a very dangerous bug!

```
constexpr double kBogusWeight = -666;

double CalculateWeight() {
  if (auto impl = GetImpl()) {
    return impl->CalculateWeight();
  } else {
    return kBogusWeight;
  }
}
```

We've changed unspecified/undefined behavior to specified.

```
constexpr double kMaxWeight = 1000;

bool IsTooHeavy() {
  return CalculateWeight() > kMaxWeight;
}
```

The tests still pass and the bug is still here.

```
optional<double> CalculateWeight() {
  if (auto impl = GetImpl()) {
    return impl->CalculateWeight();
  } else {
    return none;
  }
}
```

Now at least the failure case is part of the interface, but...

```
constexpr double kMaxWeight = 1000;

bool IsTooHeavy() {
  return CalculateWeight() > kMaxWeight;
}
```

The tests still pass and the bug is still here.

Conclusion (1)

```
double CalculateWeight() {
  if (auto impl = GetImpl()) {
    return impl->CalculateWeight();
  } else {
    // TODO handle error condition
    std::terminate();
  }
}
```

Don’t leave TODO’s with undefined behavior. Make them a defined terminate if you don’t know what to do with them.

Conclusion (2)

- Use exceptions, when appropriate.
- Let GetImpl throw.

```
double CalculateWeight() {
  return GetImpl().CalculateWeight();
}
```

Or terminate.