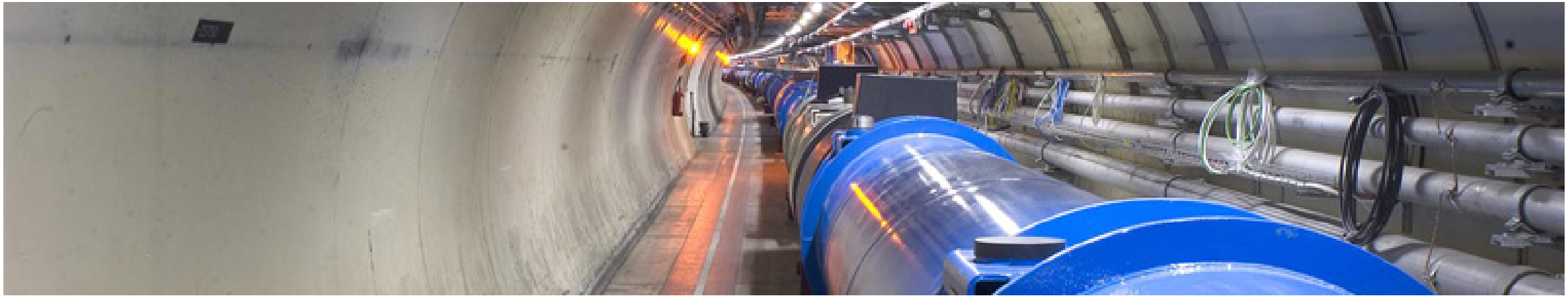
Parasol
Smarter computing.
Texas A&M University

C++11 Style – A Touch of Class

Bjarne Stroustrup

Texas A&M University

www.research.att.com/~bs



Abstract

We know how to write bad code: Litter our programs with casts, macros, pointers, naked new and deletes, and complicated control structures. Alternatively (or in addition), obscure every design decision in a mess of deeply nested abstractions using the latest object-oriented programming and generic programming tricks. For good measure, complicate our algorithms with interesting special cases. Such code is incomprehensible, unmaintainable, usually inefficient, and not uncommon.

But how do we write good code? What principles, techniques, and idioms can we exploit to make it easier to produce quality code? I will make an argument for type-rich interfaces, compact data structures, integrated resource management and error handling, and highly-structured algorithmic code. I will illustrate my ideas and motivate my guidelines with a few idiomatic code examples.

I will use C++11 freely. Examples include auto, general constant expressions, uniform initialization, type aliases, type safe threading, and user-defined literals. C++ features are only just starting to appear in production compilers, so some of my suggestions have the nature of conjecture. However, developing a “modern style” is essential if we don’t want to maintain newly-written 1970s and 1980s style code in 2020.

This presentation reflects my thoughts on what “Modern C++” should mean in the 2010s: a language for programming based on light-weight abstraction with a direct and efficient mapping to hardware, suitable for infrastructure code.

Parasol
Smarter computing.
Texas A&M University

What is C++?

Template meta-programming!

Too big!

Buffer overflows

A hybrid language

A multi-paradigm programming language

Embedded systems programming language

It's C!

Low level!

Generic programming

An object-oriented programming language

A random collection of features

C++

A light-weight abstraction programming language

Key strengths:

- software infrastructure
- resource-constrained applications

Parasol
Smarter computing.
Texas A&M University

No one size fits all

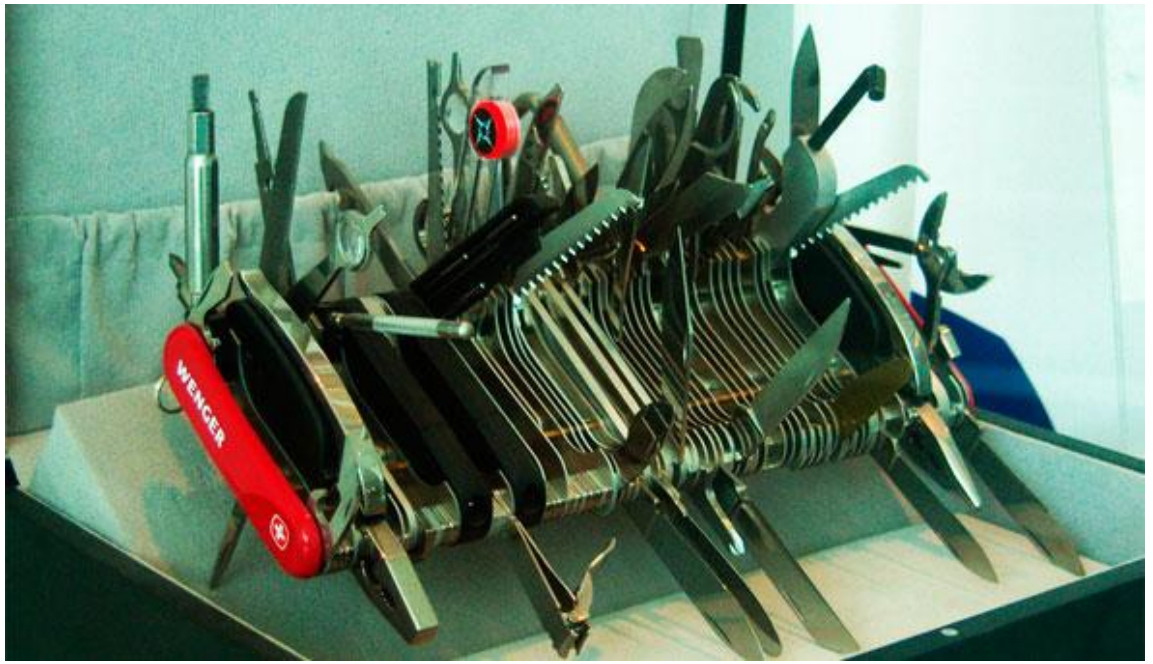
- Different projects have different constraints
  - Hardware resources
  - Reliability constraints
  - Efficiency constraints
    - Time
    - Power
  - Time to completion
  - Developer skills
- Extremes
  - All that matters is to get to the market first!
  - If the program fails, people die
  - A 50% overhead implies the need for another $50M server farm

“Multi-paradigm” is not good enough

Parasol
Smarter computing.
Texas A&M University

These styles were never meant to be disjoint:

- C style
  - functions and structures
  - Typically lots of macros, void*, and casts
- C++85 style (aka “C with Classes”)
  - classes, class hierarchies, and virtual functions
- “True OO” style
  - Just class hierarchies
  - Often lots of casts and macros
- Generic C++
  - Everything is a template



What we want

- Easy to understand
  - For humans and tools
  - correctness, maintainability
- Modularity
  - Well-specified interfaces
  - Well-defined error-handling strategy
- Effective Resource management
  - Memory, locks, files, …
- Thread safety
  - Unless specifically not
- Efficient
  - Compact data structures
  - Obvious algorithmic structure
- Portable
  - Unless specifically not

What we want

- A synthesis
  - And integrated set of features
  - C++11 is a significant improvement
- Articulated guidelines for use
  - What I call “style”

Parasol
Smarter computing.
Texas A&M University

Overview

- Ghastly style
  - qsort() example
- Type-rich Programming
  - Interfaces
  - SI example
- Resources and errors
  - RAII
  - Resource handles and pointers
  - Move semantics
- Compact data structures
  - List vs. vector
  - Vector of point
- Simplify control structure
  - Algorithms, lambdas
- Low-level != efficient
- Type-safe concurrency
  - Threads, async(), and futures

B. Stroustrup: *Software Development for Infrastructure*. IEEE Computer, January 2012

ISO C++11

- This is a talk about how to use C++ well
  - In particular, ISO C++11
  - C++ features *as a whole* support programming style
- This is not a talk about the new C++11 features
  - I use those where appropriate
  - My C++11 FAQ lists the new features
- Most C++11 features are already shipping
  - E.g. Clang, GCC, and Microsoft C++
    (the order is alphabetical ☺)
- The C++11 standard library is shipping
  - E.g. Boost, Clang, GCC, Microsoft C++

Ghastly Style

Memory to be sorted

Number of bytes in an element

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));
```

Number of elements in memory

Element comparison function

"It" doesn't know how to compare doubles?

```
void f(char* arr, int m, double* darr, int n)
{
        qsort(arr, m, sizeof(char *), cmpstringp);
        qsort(darr, n, sizeof(double), compare_double);
}
```

"It" doesn't know the size of a double?

"It" doesn't know the number of elements?

Ghastly Style

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));

static int cmpstringp(const void *p1, const void *p2)
{               /* The actual arguments to this function are "pointers to pointers to char */
                return strcmp(* (char * const *) p1, * (char * const *) p2);
}

static int compare_double(const void *p1, const void *p2)
{
                double p0 = *(double*)p;
                double q0 = *(double*)q;
                if (p0 > q0) return 1;
                if (p0 < q0) return -1;
                return 0;
}
```

Use inefficient indirect function call

Prevent inlining

Throw away useful type information

Ghastly Style

- **qsort()** implementation details
  - Note: I looked for implementations of **qsort()** on the web, but most of what I found were "educational fakes"

Swap bytes (POD only)

```
/* Byte-wise swap two items of size SIZE. */
#define SWAP(a, b, size) do { register size_t __size = (size); register char *__a = (a), *__b = (b); do { char __tmp = *__a; *__a++ = *__b; *__b++ = __tmp; } while (--__size > 0); } while (0)
/* … */
char *mid = lo + size * ((hi - lo) / size >> 1);
if ((*cmp) ((void *) mid, (void *) lo) < 0) SWAP (mid, lo, size);
if ((*cmp) ((void *) hi, (void *) mid) < 0) SWAP (mid, hi, size); else goto jump_over;
if ((*cmp) ((void *) mid, (void *) lo) < 0) SWAP (mid, lo, size);
jump_over:;
```

Lots of byte address manipulation

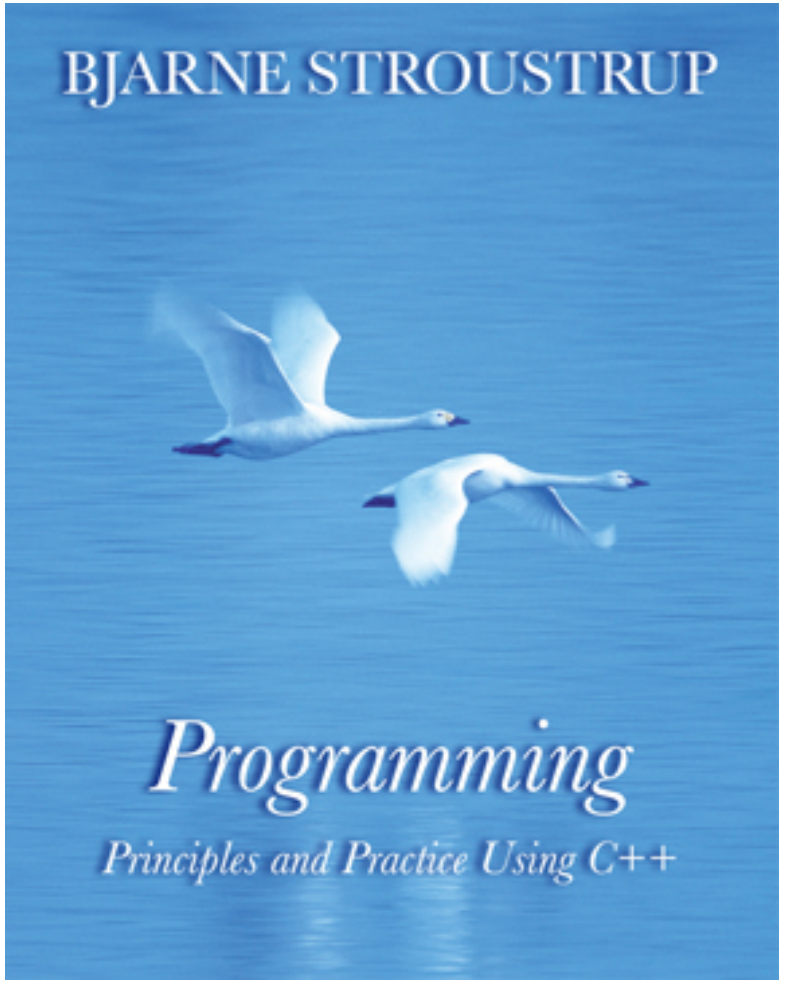
Lots of indirect function calls

Unfair? No!

- I didn't make up that example
  - it is repeatedly offered as an example of good code (for decades)
  - qsort() is a popular ISO C standard-library function
  - That qsort() code is readable compared to most low-level C/C++ code
- The style is not uncommon in production code
  - Teaching and academic versions often simplify to protect the innocent (fraud?)
- I see much worse on bulletin boards
  - Have a look, and cry
- Many students aim for that level of code
  - "for efficiency"
  - because it is cool (their idols does/did it!)
- It's not just a C/C++ problem/style
  - Though I see C and C-style teaching as the source of the problem

Does it matter? Yes!

- Bad style is the #1 problem in real-world C++ code
  - Makes progress relatively easy
  - Only relatively: bad code breeds more bad code
- Lack of focus on style is the #1 problem in C++ teaching
  - A “quick hack” is usually the quickest short-term solution
    - Faster than thinking about “design”
  - “They” typically teach poor style
  - Many are self-taught
    - Take advice from
      - Decades old books
      - Other novices
    - Imitate
      - Other languages
      - Bad old code



So what do I want?

- Simple interfaces

```
void sort(Container&);      // for any container (e.g. vector, list, array)
                            // I can't quite get this is in C++ (but close)
```

- Simple calls

```
vector<string> vs;
// …
sort(vs);          // this, I can do
```

- Uncompromising performance
  - Done: **std::sort()** beats **qsort()** by large factors (not just a few percent)
- No static type violations
  - Done
- No resource leaks
  - Done (without a garbage collector)

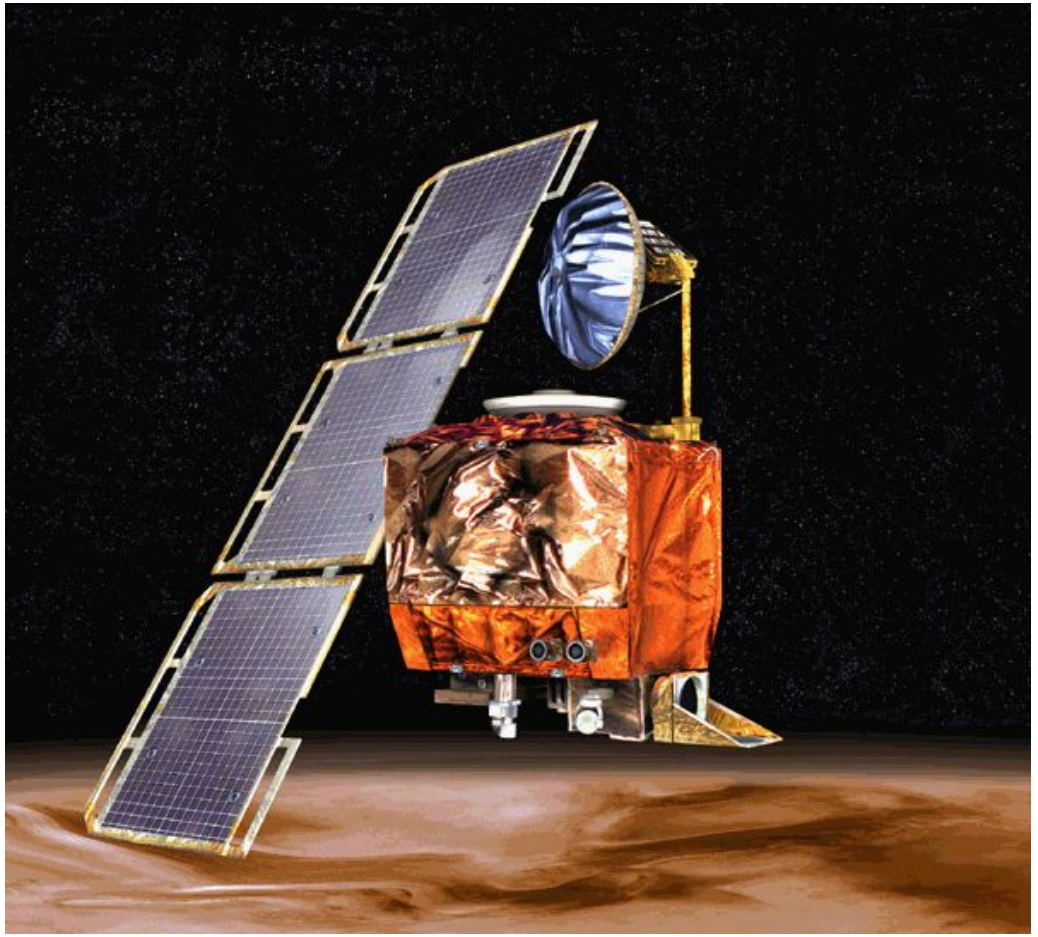
Type-rich Programming

- Interfaces
- SI-units

Focus on interfaces

- Underspecified / overly general:
  - **void increase_speed(double);**
  - **Object obj; … obj.draw();**
  - **Rectangle(int,int,int,int);**

- Better:
  - **void increase_speed(Speed);**
  - **Shape& s; … s.draw();**
  - **Rectangle(Point top_left, Point bottom_right);**
  - **Rectangle(Point top_left, Box_hw b);**



SI Units

- Units are effective and simple:

```
Speed sp1 = 100m/9.8s;          // very fast for a human
Speed sp2 = 100m/9.8s2;         // error (m/s2 is acceleration)
Speed sp3 = 100/9.8s;           // error (speed is m/s and 100 has no unit)
Acceleration acc = sp1/0.5s;    // too fast for a human
```

- They are also almost never used in programs
  - General-purpose languages generally don't directly support units
  - Run-time checking is far too costly

SI Units

- We can define Units to be handled at compile time:

```
template<int M, int K, int S> struct Unit {  // a unit in the MKS system
        enum { m=M, kg=K, s=S };
};

template<typename Unit>      // a magnitude with a unit
struct Value {
        double val;                          // the magnitude
        explicit Value(double d) : val(d) {}  // construct a Value from a double
};

using Speed = Value<Unit<1,0,-1>>;                  // meters/second type
using Acceleration = Value<Unit<1,0,-2>>;           // meters/second/second type
```

SI Units

- We have had libraries like that for a decade
  - but people never used them:

```
Speed sp1 = Value<1,0,0> (100)/ Value<0,0,1> (9.8);   // very explicit
Speed sp1 = Value<M> (100)/ Value<S> (9.8);           // use a shorthand notation
Speed sp1 = Meters(100)/Seconds(9.8);                 // abbreviate further still
Speed sp1 = M(100)/S(9.8);                            // this is getting cryptic
```

- Notation matters.

SI Units

- So, improve notation using user-defined literals:

```
using Second = Unit<0,0,1>;      // unit: sec
using Second2 = Unit<0,0,2>;     // unit: second*second

constexpr Value<Second> operator"" s(long double d)
        // a f-p literal suffixed by 's'
{
        return Value<Second> (d);
}

constexpr Value<Second2> operator"" s2(long double d)
        // a f-p literal  suffixed by 's2'
{
        return Value<Second2> (d);
}
```

SI Units

- Units are effective and simple:

```
Speed sp1 = 100m/9.8s;          // very fast for a human
Speed sp2 = 100m/9.8s2;         // error (m/s2 is acceleration)
Speed sp3 = 100/9.8s;           // error (speed is m/s and 100 has no unit)
Acceleration acc = sp1/0.5s;    // too fast for a human
```

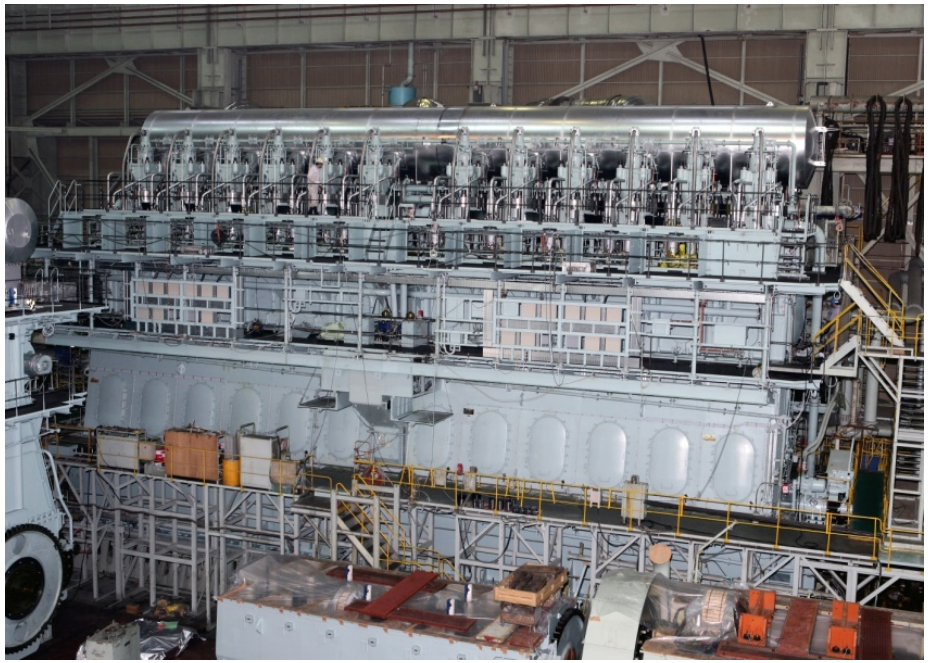
- and essentially free (in C++11)
  - Compile-time only
  - No run-time overheads

Style

- Keep interfaces strongly typed
  - Avoid very general types in interfaces, e.g.,
    - **int**, **double,** …
    - **Object,** …

Because such types can represent just about anything

  - Checking of trivial types find only trivial errors
  - Use precisely specified interfaces



Resources and Errors

- Resources
- RAII
- Move

Resources and Errors

```
//      unsafe, naïve use:

void f(const char* p)
{
        FILE* f = fopen(p,"r");      // acquire
        // use f
        fclose(f);                   // release
  }
```

Resources and Errors

```
//      naïve fix:

void f(const char* p)
{
        FILE* f = 0;
        try {
            f = fopen(p, "r");
            // use f
        }
        catch (...) {     // handle every exception
            if (f) fclose(f);
            throw;
        }
        if (f) fclose(f);
  }
```

RAII (Resource Acquisition Is Initialization)

```
// use an object to represent a resource
class File_handle { // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r)
                { p = fopen(pp,r); if (p==0) throw File_error{pp,r}; }
        File_handle(const string& s, const char* r)
                { p = fopen(s.c_str(),r); if (p==0) throw File_error{s,r}; }
        ~File_handle() { fclose(p); }          // destructor
        // copy operations
        // access functions
};

void f(string s)
{
        File_handle fh {s, "r“};
        // use fh
}
```

RAII

- For all resources
  - Memory (done by **std::string**, **std::vector**, **std::map**, …)
  - Locks (e.g. **std::unique_lock**), files (e.g. **std::fstream**), sockets, threads (e.g. **std::thread**), …

```
std::mutex m;      // a resource
int sh;            // shared data

void f()
{
    // …
    std::unique_lock lck {m}; // grab (acquire) the mutex
    sh+=1;                    // manipulate shared data
}                             // implicitly release the mutex
```

Resource Handles and Pointers

- Many (most?) uses of pointers in local scope are not exception safe

```
void f(int n, int x)
{
        Gadget* p = new Gadget{n};          // look I'm a java programmer! ☺
        // …
        if (x<100) throw std::run_time_error{"Weird!"};          // leak
        if (x<200) return;                                       // leak
        // …
        delete p;                           // and I want my garbage collector! ☹
}
```

  - No "Naked New"!
  - But, why use a "raw" pointer?

Resource Handles and Pointers

- A **std::shared_ptr** releases its object at when the last **shared_ptr** to it is destroyed

```
void f(int n, int x)
{
        shared_ptr<Gadget> p = new Gadget{n};        // manage that pointer!
        // …
        if (x<100) throw std::run_time_error{“Weird!”};        // no leak
        if (x<200) return;                                     // no leak
        // …
}
```

  - But why use a **shared_ptr**?
  - I’m not sharing anything.

Resource Handles and Pointers

- A **std::unique_ptr** releases its object at when the **unique_ptr** is destroyed

```
void f(int n, int x)
{
        unique_ptr<Gadget> p = new Gadget{n};
        // …
        if (x<100) throw std::run_time_error{“Weird!”};        // no leak
        if (x<200) return;                                     // no leak
        // …
}
```

  - But why use ***any*** kind of pointer ?
  - I’m not passing anything around.

Resource Handles and Pointers

- But why use a pointer at all?
- If you can, just use a scoped variable

```
void f(int n, int x)
{
        Gadget g {n};
        // …
        if (x<100) throw std::run_time_error{“Weird!”};        // no leak
        if (x<200) return;                                     // no leak
        // …
}
```

Resource Management Style

- Prefer classes where the resource management is part of their fundamental semantics
  - E.g., **std::vector**, **std::ostream**, **std::thread**, …
- Use "smart pointers" to address the problems of premature destruction and leaks
  - **std::unique_ptr** for (unique) ownership
    - Zero cost (time and space)
  - **std::shared_ptr** for shared ownership
    - Maintains a use count
  - But they are still pointers
    - "any pointer is a potential race condition – even in a single threaded program"

How to move a resource

- Common problem:
  - How to get a lot of data cheaply out of a function
- Idea #1:
  - Return a pointer to a **new**'d object

    Matrix* operator+(const Matrix&, const Matrix&);

    Matrix& res = *(a+b); // *ugly! (unacceptable)*

  - Who does the **delete**?
    - there is no good general answer

How to move a resource

- Common problem:
  - How to get a lot of data cheaply out of a function
- Idea #2
  - Return a reference to a **new**'d object

```
Matrix& operator+(const Matrix&, const Matrix&);
Matrix res = a+b;       // looks right, but …
```

  - Who does the **delete**?
    - What **delete**? I don't see any pointers.
    - there is no good general answer

How to move a resource

- Common problem:
  - How to get a lot of data cheaply out of a function
- Idea #3
  - Pass an reference to a result object

```
void operator+(const Matrix&, const Matrix&, Matrix& result);
Matrix res = a+b;                    // Oops, doesn't work for operators
Matrix res2;
operator+(a,b,res2);                 // Ugly!
```

  - We are regressing towards assembly code

How to move a resource

- Common problem:
  - How to get a lot of data cheaply out of a function
- Idea #4
  - Return a **Matrix**

    Matrix operator+(const Matrix&, const Matrix&);

    Matrix res = a+b;

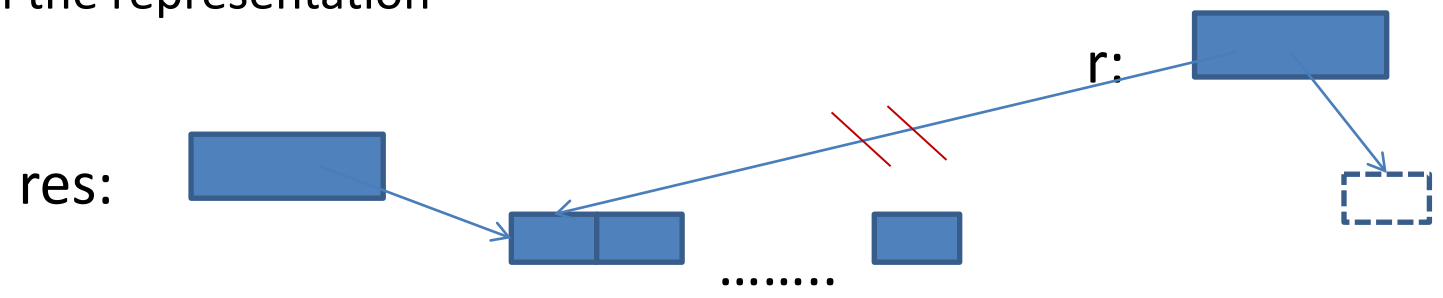
    - Copy?
      - expensive
    - Use some pre-allocated "result stack" of **Matrix**es
      - A brittle hack
    - Move the **Matrix** out
      - don't copy; "steal the representation"
      - Directly supported in C++11 through move constructors

Move semantics

- Return a **Matrix**

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix r;
    // copy a[i]+b[i] into r[i] for each i
    return r;
}
Matrix res = a+b;
```

- Define move a constructor for **Matrix**
  - don't copy; "steal the representation"

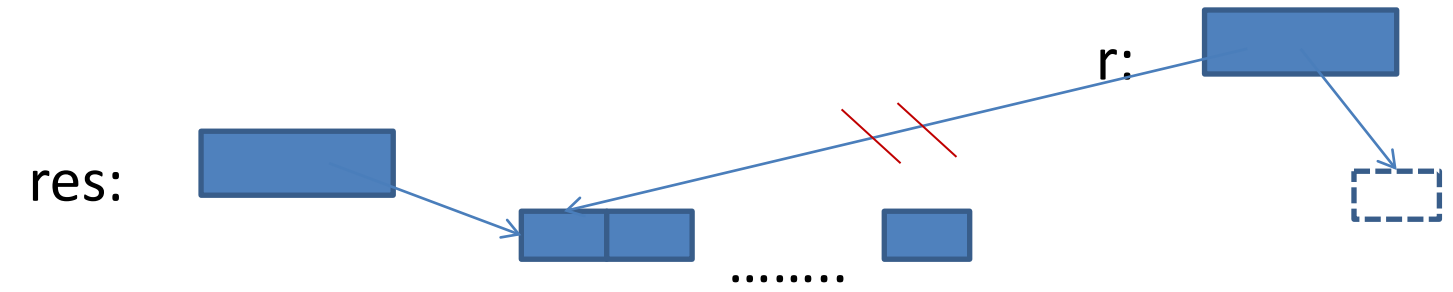


Move semantics

- Direct support in C++11: Move constructor

```
class Matrix {
    Representation rep;
    // …
    Matrix(Matrix&& a)      // move constructor
    {
        rep = a.rep;        // *this gets a's elements
        a.rep = {};         // a becomes the empty Matrix
    }
};

Matrix res = a+b;
```

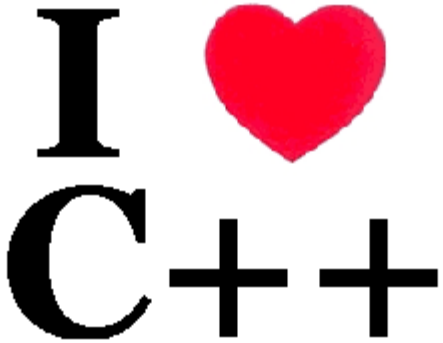


Move semantics

- All the standard-library containers have move constructors and move assignments
  - **vector**
  - **list**
  - **forward_list** (singly-linked list)
  - **map**
  - **unordered_map** (hash table)
  - **set**
  - …
  - **string**
- Not **std::array**

Style

- No naked pointers
  - Keep them inside functions and classes
  - Keep arrays out of interfaces (prefer containers)
  - Pointers are implementation-level artifacts
  - A pointer in a function should not represent ownership
  - Always consider **std::unique_ptr** and sometimes **std::shared_ptr**
- No naked **new** or **delete**
  - They belong in implementations and as arguments to resource handles
- Return objects “by-value” (using move rather than copy)
  - Don’t fiddle with pointer, references, or reference arguments for return values



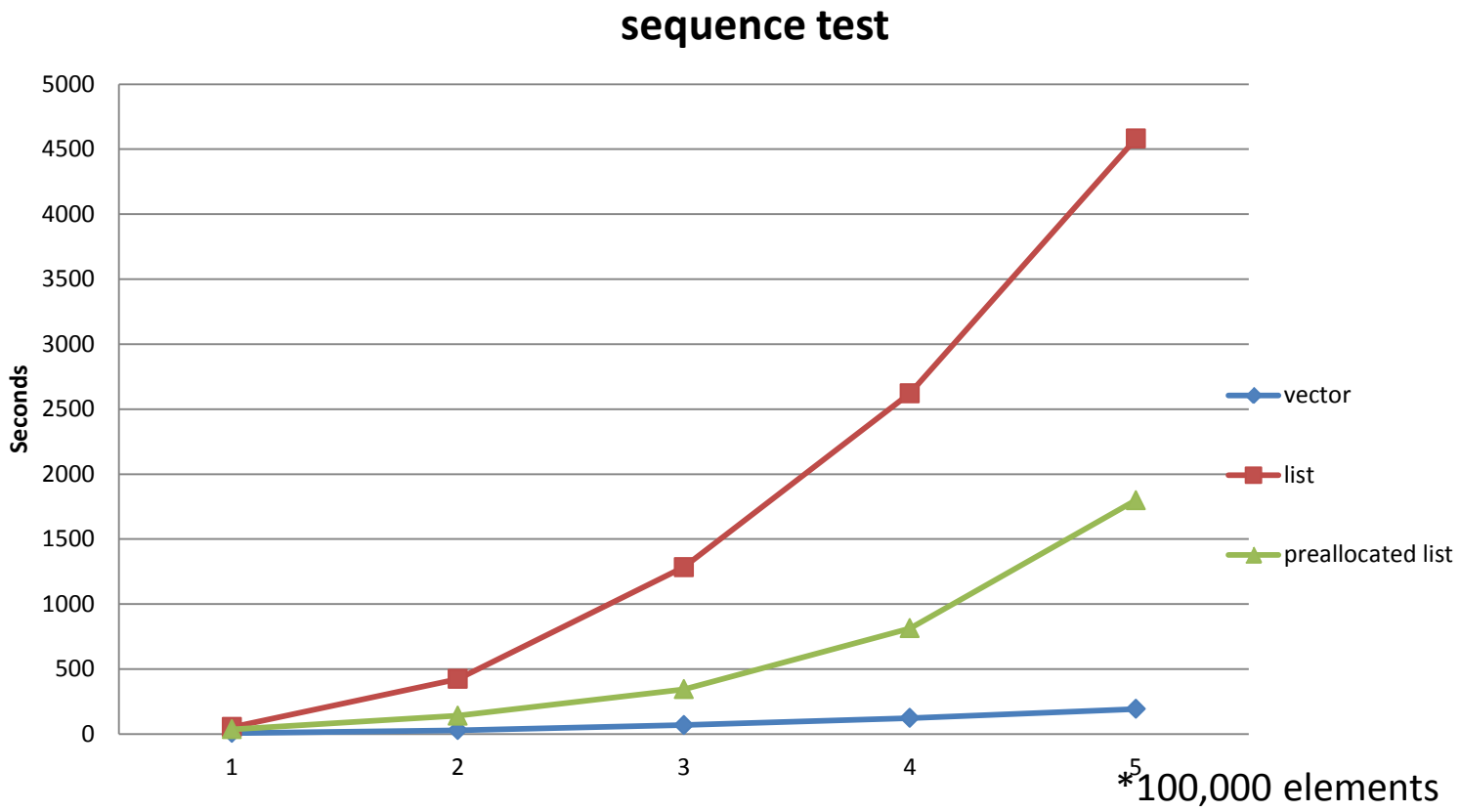
Use compact data

- Vector vs. list
- Object layout

Vector vs. List

- Generate N random integers and insert them into a sequence so that each is inserted in its proper position in the numerical order. **5 1 4 2** gives:
  - **5**
  - **1 5**
  - **1 4 5**
  - **1 2 4 5**
- Remove elements one at a time by picking a random position in the sequence and removing the element there. Positions **1 2 0 0** gives
  - **1 2 4 5**
  - **1 4 5**
  - **1 4**
  - **4**
- For which N is it better to use a linked list than a vector (or an array) to represent the sequence?

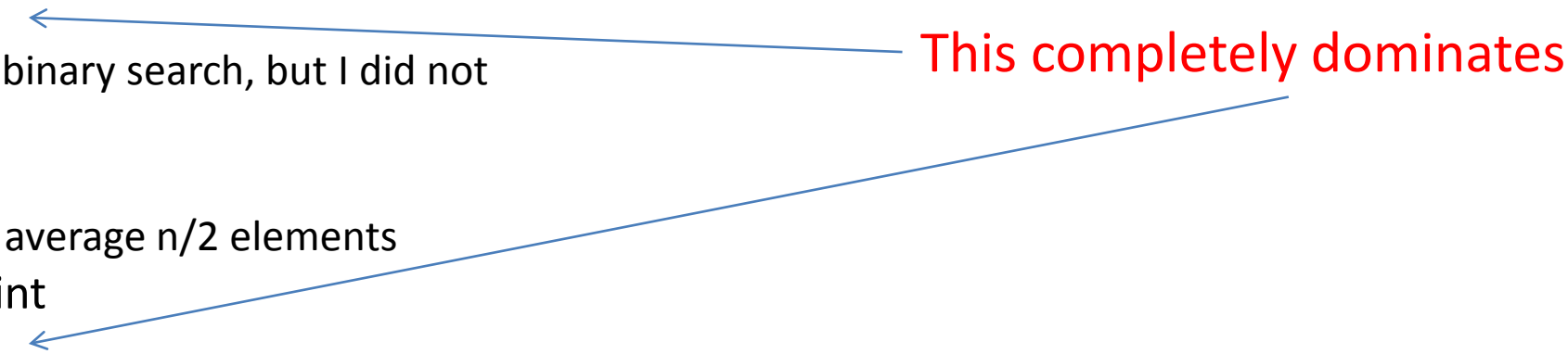
Vector vs. List



- Vector beats list massively for insertion and deletion
  - For small elements and relatively small numbers (up to 500,000 on my machine)
  - Your mileage ***will*** vary

Vector vs. List

- Find the insertion point
  - Linear search
  - Vector could use binary search, but I did not
- Insert
  - List re-links
  - Vector moves on average n/2 elements
- Find the deletion point
  - Linear search
  - Vector could use direct access, but I did not
- delete
  - List re-links
  - Vector moves on average n/2 elements
- Allocation
  - List does N allocations and N deallocations
  - The optimized/preallocated list do no allocations or dealloations
  - Vector does approximately log2(N) allocations and log2(N) deallocations
  - The optimized list does 1 allocation and 1 deallocation



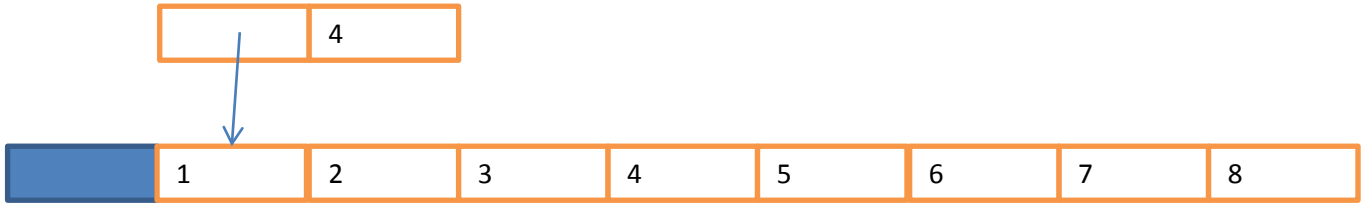
Vector vs. List

- The amount of memory used differ dramatically
  - List uses 4+ words per element
    - it will be worse for 64-bit architectures
    - 100,000 list elements take up 6.4MB or more (but I have Gigabytes!?)
  - Vector uses 1 word per element
    - 100,000 list elements take up 1.6MB or more
- Memory access is relatively slow
  - Caches, pipelines, etc.
  - 200 to 500 instructions per memory access
  - Unpredictable memory access gives many more cache misses
- Implications:
  - Don’t store data unnecessarily.
  - Keep data compact.
  - Access memory in a predictable manner.

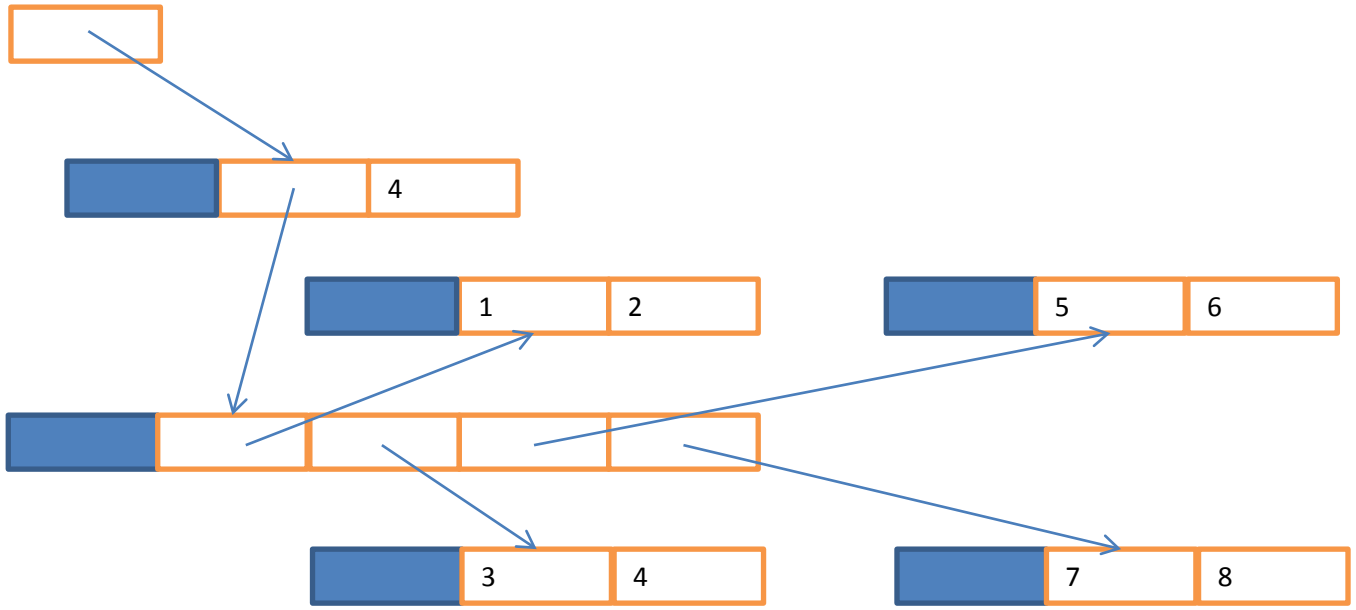
Use compact layout

vector<Point> vp = { Point{1,2}, Point{3,4}, Point{5,6}, Point{7,8} };

C++:



"True OO" style:



Simplify control structure

- Prefer algorithms to unstructured code

Algorithms vs. “Code”

- Problem: drag item to an insertion point
- Original solution (after cleanup and simplification):
  - 25 lines of code
    - one loop
    - three tests
    - 14 function calls
- Messy code
  - Is it correct?
    - who knows? try lots of testing
  - Is it maintainable?
    - Probably not, since it is hard to understand
  - Is it usable elsewhere?
    - No, it’s completely hand-crafted to the details of the problem
- The author requested a review
  - Professionalism!

Algorithms vs. "Code"

- Surprise!
  - it was a simple **find_if** followed by moving the item

```
void drag_item_to(Vector& v, Vector::iterator source, Coordinate p)
{
        Vector::iterator dest = find_if(v.begin(), v.end(), contains(p));        // find the insertion point
        if (source < dest)
                rotate(source, source+1, dest);        // from before insertion point
        else
                rotate(dest, source, source+1);        // from after insertion point
}
```

- It's comprehensible (maintainable), but still special purpose
  - **Vector** and **Coordinate** are application specific

Algorithms vs. “Code”

- Why move only one item?
  - Some user interfaces allow you to select many

```
template < typename Iter, typename Predicate>
pair<Iter, Iter> gather(Iter first, Iter last, Iter p, Predicate pred)
        // move elements for which pred() is true to the insertion point p
{
        return make_pair(
                stable_partition(first, p, !bind(pred, _1)),    // from before insertion point
                stable_partition(p, last, bind(pred, _1))       // from after insertion point
        );
}
```

- Shorter, simpler, faster, general (usable in many contexts)
  - No loops and no tests

Style

- Focus on algorithms
  - Consider generality and re-use
- Consider large functions suspect
- Consider complicated control structures suspect

Stay high level

- When you can
- Most of the time

Low-level != efficient

- Language features + compiler + optimizer deliver performance
  - You can afford to use libraries of algorithms and types
  - **for_each()**+lambda vs. for-loop
    - Examples like these give identical performance on several compilers:

```
sum = 0;
for(vector<int>::size_type i=0; i<v.size(); ++i)          // conventional loop
        sum += v[i];

sum = 0;
for_each(v.begin(),v.end(),                               // algorithm + lambda
        [&sum](int x) {sum += x; });
```

Low-level != efficient

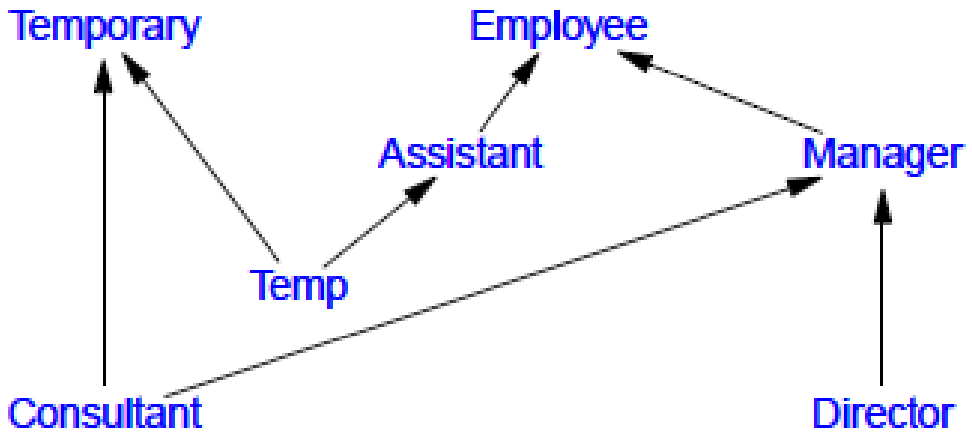
- Language features + compiler + optimizer deliver performance
  - **sort()** vs. **qsort()**
  - Roughly : C is 2.5 times slower than C++
    - Your mileage ***will*** wary
- Reasons:
  - Type safety
    - Transmits more information to the optimizer
    - also improves optimization, e.g. type-bases anti-aliasing
  - Inlining
- Observations
  - Performance of traditional C-style and OO code is roughly equal
  - Results vary based on compilers and library implementations
    - But sort() is typical

Low-level != efficient

- Don’t lower your level of abstraction without good reason
- Low-level implies
  - More code
  - More bugs
  - Higher maintenance costs

Inheritance

- Use it
  - When the domain concepts are hierarchical
  - When there is a need for run-time selection among hierarchically ordered alternatives



- Warning:
  - Inheritance has been seriously and systematically overused and misused
    - "When your only tool is a hammer everything looks like a nail"

Parasol
Smarter computing.
Texas A&M University

Concurrency

- There are many kinds
- Stay high-level
- Stay type-rich

Type-Safe Concurrency

- Programming concurrent systems is hard
  - We need all the help we can get
  - C++11 offers type-safe programming at the threads-and-locks level
  - Type safety is hugely important
- threads-and-locks
  - is an unfortunately low level of abstraction
  - is necessary for current systems programming
    - That's what the operating systems offer
  - presents an abstraction of the hardware to the programmer
  - can be the basis of other concurrency abstractions

Threads

```
void f(vector<double>&);            // function

struct F {                          // function object
    vector<double>& v;
    F(vector<double>& vv) :v{vv} { }
    void operator()();
};

void code(vector<double>& vec1, vector<double>& vec2)
{
    std::thread t1 {f,vec1};        // run f(vec1) on a separate thread
    std::thread t2 {F{vec2}};       // run F{vec2}() on a separate thread
    t1.join();
    t2.join();
    // use vec1 and vec2
}
```

Thread – pass argument and result

```
double* f(const vector<double>& v);    // read from v return result
double* g(const vector<double>& v);    // read from v return result

 void user(const vector<double>& some_vec)  // note: const
{
        double res1, res2;
        thread t1 {[&]{ res1 = f(some_vec); }};        // lambda: leave result in res1
        thread t2 {[&]{ res2 = g(some_vec); }};        // lambda: leave result in res2
        // …
        t1.join();
        t2.join();
        cout << res1 << ' ' << res2 << '\n';
}
```

async() – pass argument and return result

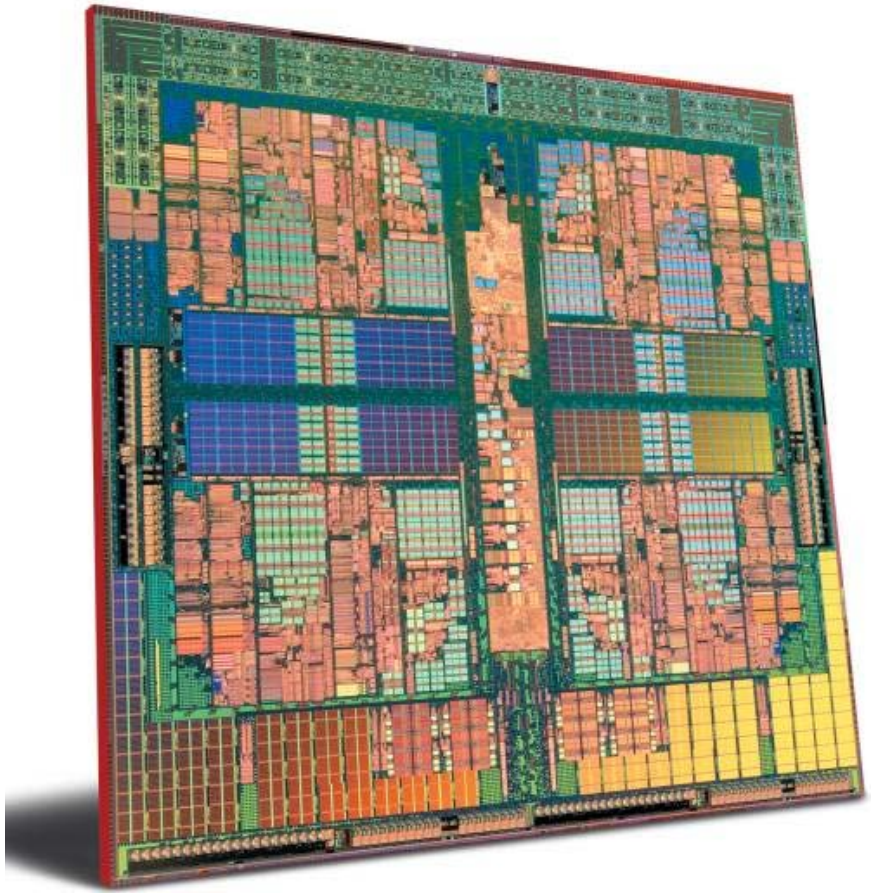
```
double* f(const vector<double>& v);    // read from v return result
double* g(const vector<double>& v);    // read from v return result

void user(const vector<double>& some_vec)   // note: const
{
        auto res1 = async(f,some_vec);
        auto res2 = async(g,some_vec);
        // …
        cout << *res1.get() << ' ' << *res2.get() << '\n';          // futures
}
```

- Much more elegant than the explicit thread version
  - And most often faster

C++ Style

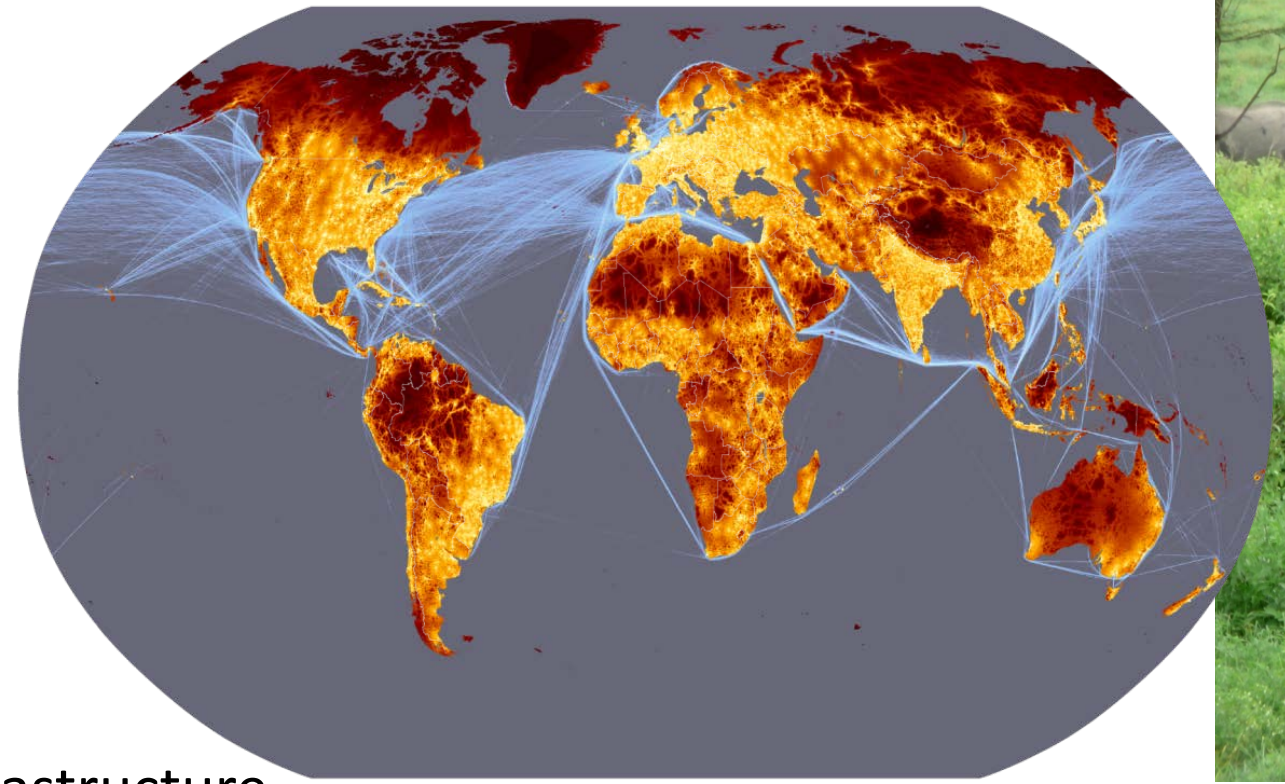
- Practice type-rich programming
  - Focus on interfaces
  - Simple classes are cheap – use lots of those
  - Avoid over-general interfaces
- Integrate Resource Management and Error Handling
  - By default, use exceptions and RAII
  - Prefer move to complicated pointer use
- Use compact data structures
  - By default, use **std::vector**
- Prefer algorithms to “random code”
- Build and use libraries
  - Rely on type-safe concurrency
  - By default, start with the ISO C++ standard library



Parasol
Smarter computing.
Texas A&M University

Questions?

C++: A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications