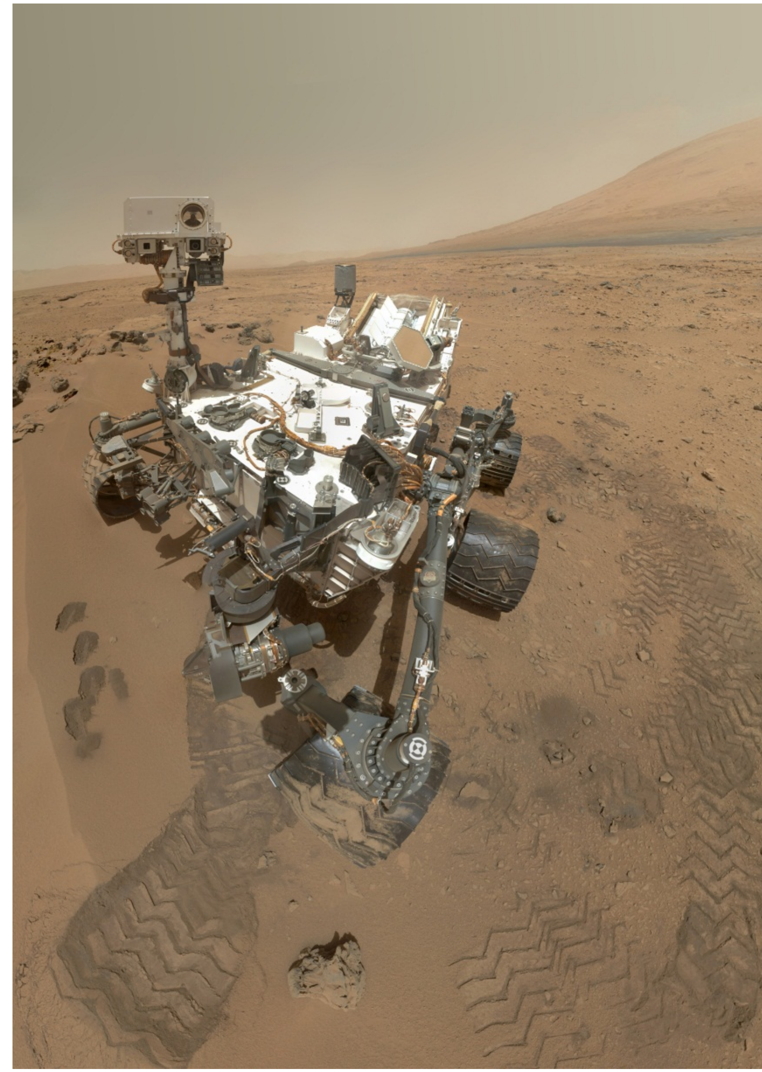
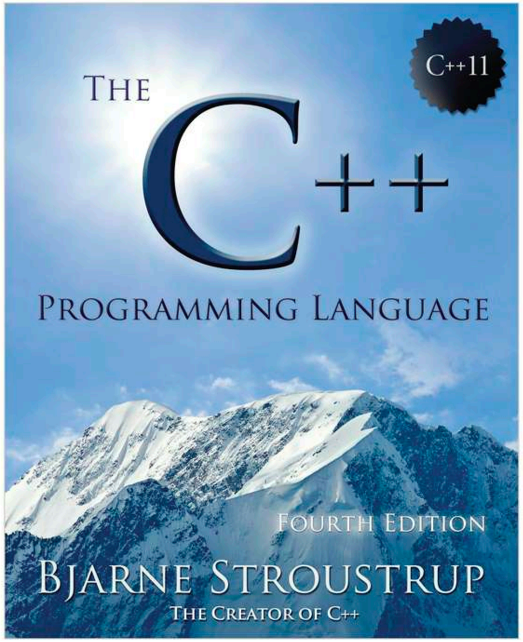
C++ Style for 2014 and beyond

Bjarne Stroustrup
Morgan Stanley,
Columbia University, TAMU
www.stroustrup.com



Overview

- C++
- Simplifying code using C++14
  - For, auto, lambda, UDL, concepts, vector, move, algorithms, …
- Overview
  - Language and libraries
- Where is the overhead?
  - Poor algorithms
  - Complicated data structures
  - Messy code



What do we want?

- Reliability
- Speed (latency)
- Throughput
- Maintainability
  - Readability
  - Productivity
  - Portability

- Popularization
  - Teaching

- C++ is my main tool
  - Research
  - Development

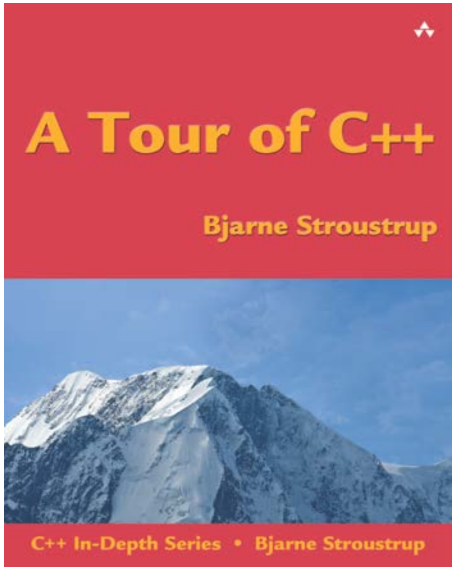
Language Myths

- There is a best language
  - For everybody and for every task
  - One size fits all

- Oh, no!
  - These myths confound
    - Education
    - Practice
    - Research
    - Language design
    - Management
    - Funding

What does C++ offer?

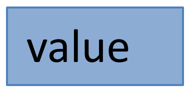
- What is C++?
  - Direct map to hardware
    - of instructions and fundamental data types
    - Initially from C
  - Zero-overhead abstraction
    - Classes with constructors and destructors, inheritance, generic programming, function objects
    - Initially from Simula



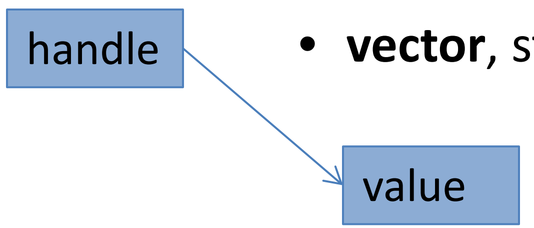
- Much of the inspiration came from operating systems
- What does C++ want to be when it grows up?
  - See above
  - And be better at it for more modern hardware and techniques

Map to Hardware

- Primitive operations => instructions
  - +, %, ->, [], (), …

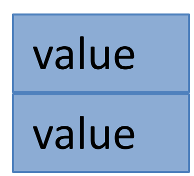


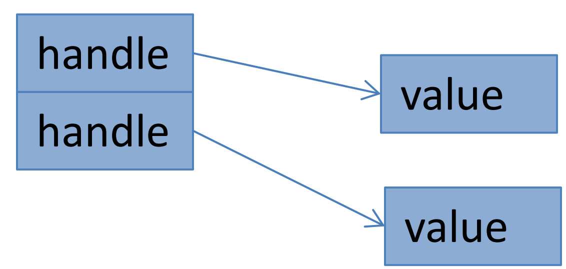
- **int**, double, complex<double>, Date, …



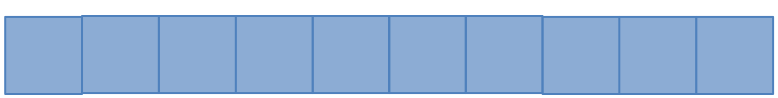
- **vector**, string, thread, Matrix, …

- Objects can be composed by simple concatenation:
  - Arrays
  - Classes/structs





- All maps to "raw memory"



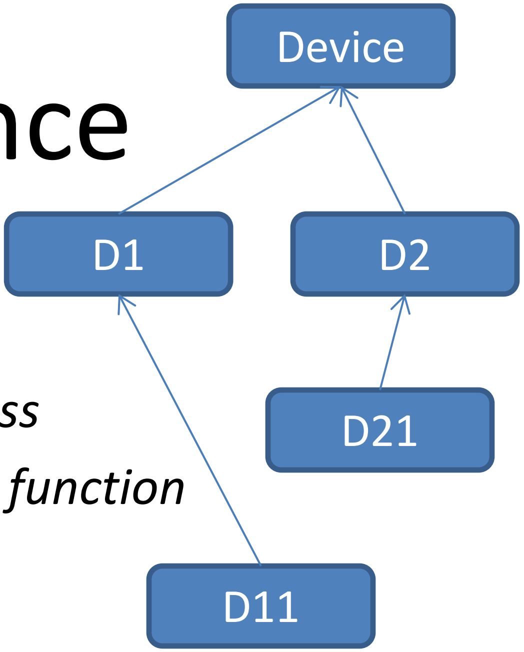
Classes: Construction/Destruction

```
class X {                 // a user-defined type called X
public:          // interface
     X(Something);        // constructor from Something
     ~X();                // destructor
     // …
private:         // implementation
     // …
};
```

"A constructor establishes the environment for the members to run in; the destructor reverses its actions."

– BS 1979 (slightly rephrased)

Classes and Inheritance



- Insulate the user from the implementation

```
struct Device {                              // abstract class
    virtual int put(const char*) = 0;        // pure virtual function
    virtual int get(const char*) = 0;
};
```

- A class can be the root of a hierarchy of derived classes
  - The derived classes supply the implementation (code and possibly data)
- Abstract classes: No data members, all data in derived classes
  - "not brittle"
- Manipulate through pointer or reference
  - Typically allocated on the free store ("dynamic memory")
  - Typically requires some form of lifetime management
    - use resource handles

Parameterized Types and Classes

- Templates
  - Essential: Support for generic programming
  - Secondary: Support for metaprogramming

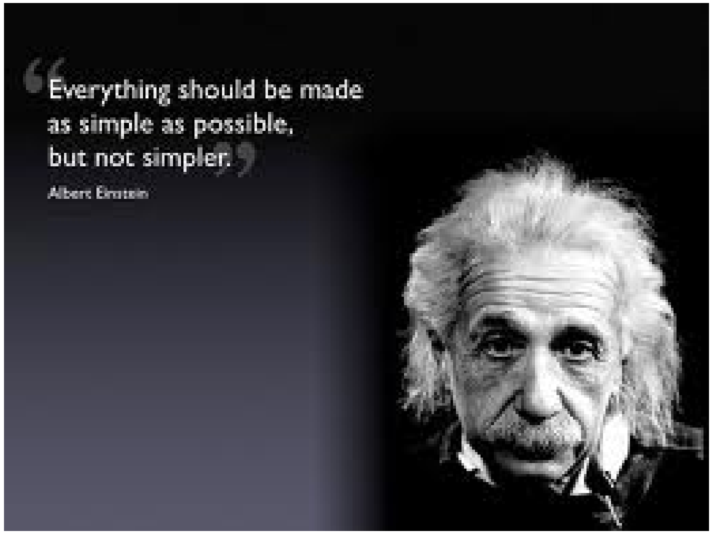
```
template<typename T>
class vector { /* … */ };                    // a generic type

vector<double> constants = {3.14159265359, 2.54, 1, 6.62606957E-34, };   // a use
```

C++14 Concept

```
template<Sortable Seq>
void sort (Seq& c) { /* … */ }        // a generic function taking a sortable sequence

sort(constants);                      // a use
```

Make simple things simple!

- What does “simple” mean?



- Make the code directly express intent
  - Simple code is easier to understand
  - Simple code is often beautiful
  - Simple code is often fast
  - Not all code can be simple
    - Hide the complexity behind a simple interface

A simple example:

Compute the mean of a sequence of numbers

```
auto mean(const Sequence& seq)
{
    auto n = 0.0;
    for (x : seq)
        n += x;
    return n / seq.size();
}

cout << mean({1,2,3,4,5,6,7,8,9,0});

cout << char(mean("this is also a sequence"));
```

Similar to other languages

Python

```
def mean(seq):
   n = 0.0
   for x in seq:
      n += x
   return n / len(seq)
```

C++

```
auto mean(const Sequence& seq) {
   auto n = 0.0;
   for (x : seq)
      n += x;
   return n / seq.size();
}
```

We can simplify further

- **def mean(seq):**
  return sum(seq) / len(seq)

```
auto mean(const Sequence& seq) {
        return accumulate(seq,{}) / seq.size();
}
```

- Nil of the appropriate type (Value_type<Sequence>)

- Libraries can make most things simple to use

Resource management

- A resource is something that must be acquired and released
  - explicitly or implicitly
- Examples: memory, locks, file handles, sockets, thread handles

```
void f(int n, string name)
{
        vector<int> v(n);               // vector of n integers
        fstream fs { name, "r"};        // open file <name> for reading
        // …
} // memory and file released here
```

- We must avoid manual resource management
  - We don't want leaks
  - We want to minimize resource retention

Value types

- The key types of C++ are value types
  - The built-in types
  - The standard-library types
  - Many of your own types
- Assignment yields two independent objects that compare equal

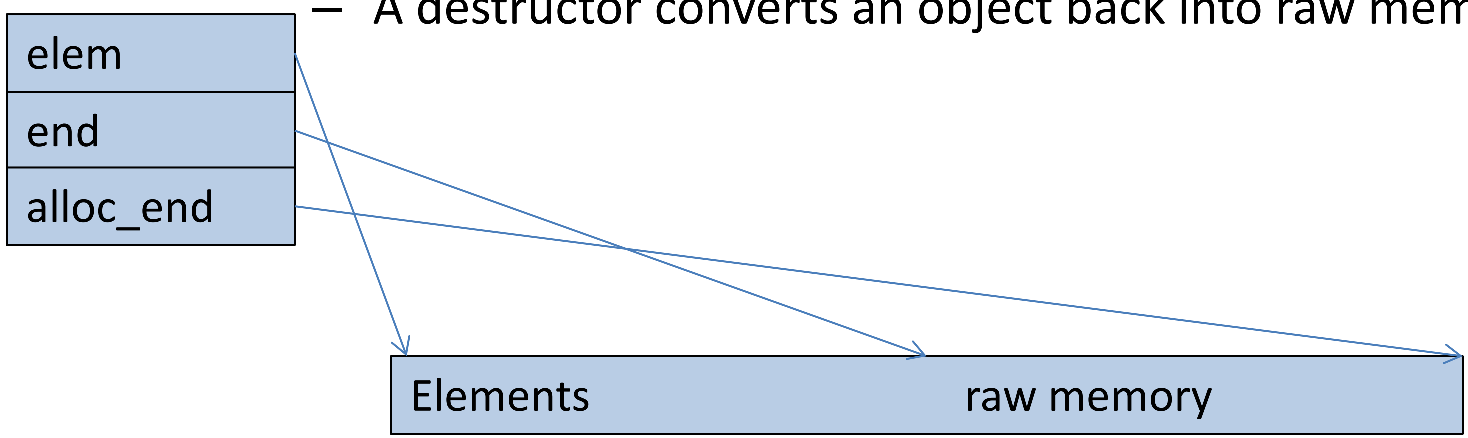
```
int i1 {7};
int i2 {i1};                // i1==i2
++i1;                       // i1==8; i2==7
vector<int> v1  {1,2,3,4};
vector<int> v2 {v1};        // v1==v2
++v1[0];                    // v1== {2,2,3,4}; v2=={1,2,3,4}
```

- Value types are a key to simplicity and performance

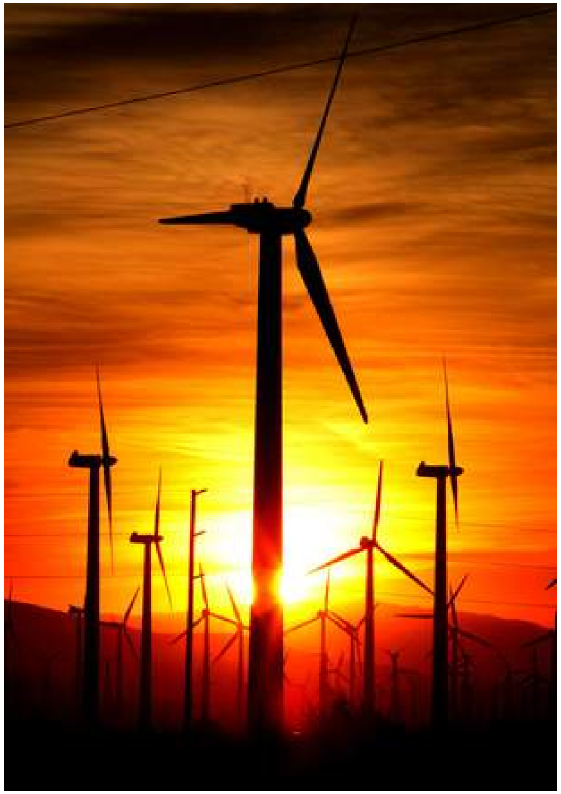
Vector representation

- Not seen by end-users ("private")
  - **vector_rep** takes care of untyped memory
  - **vector** takes care of typed objects
  - Direct access to "raw" memory
    - C++ is (among other things) a systems programming language
  - A constructor converts raw memory to an object
  - A destructor converts an object back into raw memory

vector_rep:



Control

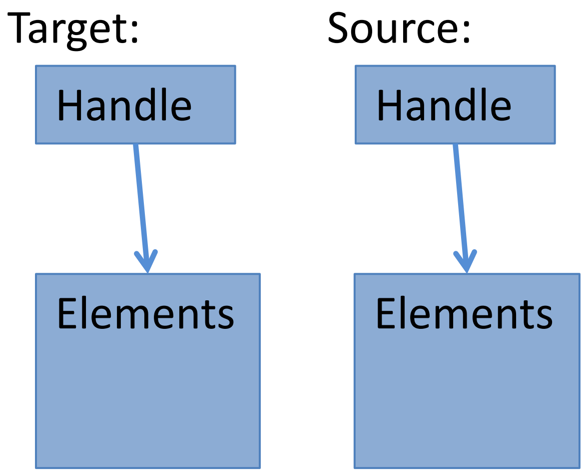


- We control object lifetime/life-cycle
  - Creation of objects: constructors
  - Destruction of objects: destructors
  - Movement of objects
    - Construction and assignment
    - from on scope to another
  - Copying of objects
    - Construction and assignment
    - from on scope to another
  - Access to representation
- At no cost compared to low-level hand coding

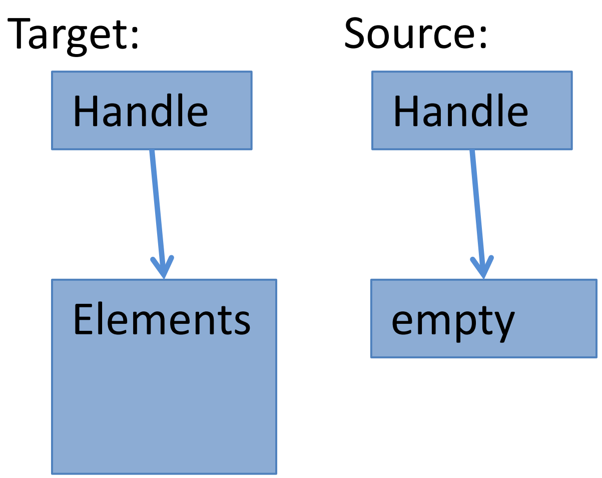
Copy and move

```
vector user(vector<int>& x)
{
        vector<int> y = x; // copy
        // …
        return y;          // move, don't copy
}

Vector<int> z = algo(some_vec);

// copy constructor and copy assignment
// move constructor and move assignment
```

copy:



move:



C++11/14 overview

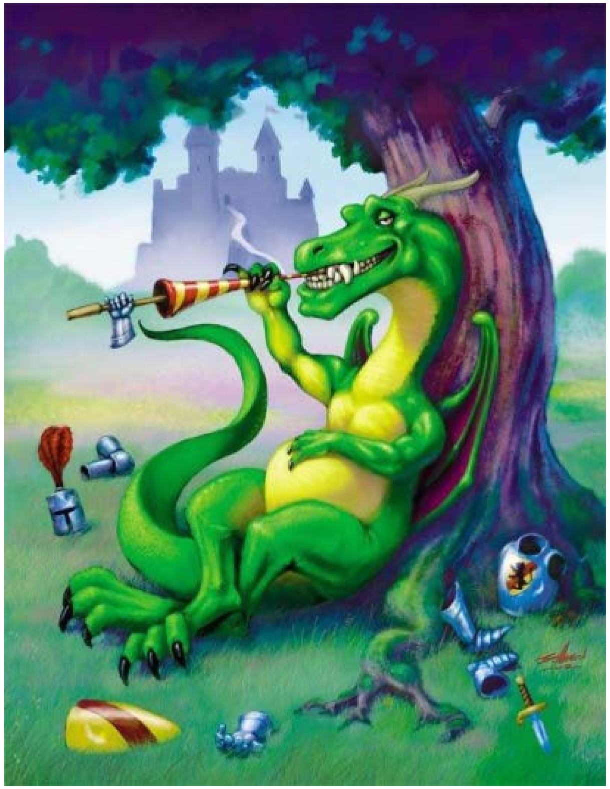
- Language features
  - Move, uniform initialization, range-for, auto, lambdas, variadic templates, user-defined literals, digit separators, forwarding constructors, in-class initialization, generalized constant expressions, template aliases, …
  - Memory model
- Library components
  - Type-safe threads, mutexes
  - lock-free programming
  - futures
  - Random numbers
  - Regular expressions
  - Emplace operations, move semantics, initializer lists, …
  - Type traits

Where does the time go?

- I mean, in addition to
  - System calls
  - Net accesses
  - Disk read/writes

  of course.

- A partial answer
  - Too much data
  - Poorly structured data
  - Unpredictable access to data
  - Messy code
  - Too many run-time decisions



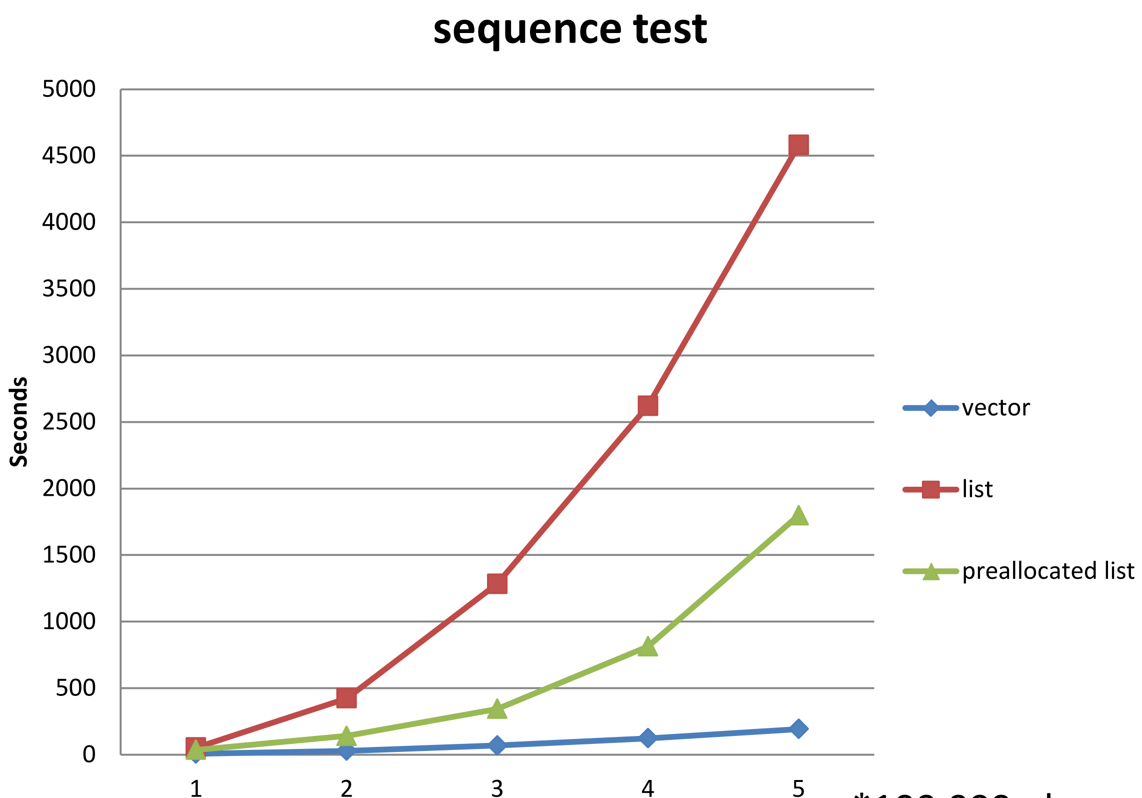
Use compact data

- 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?
- The sequence grows incrementally

Vector vs. List



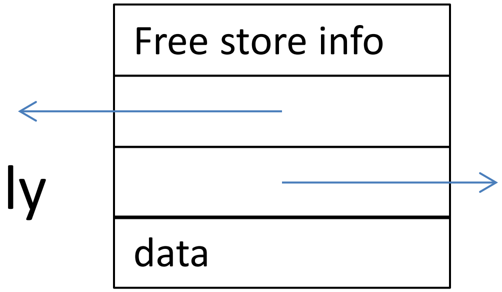
*100,000 elements

- 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

Implementation (performance-critical part)

```
template<class C> void do_insert(C& s, int N)
{
   for (int i=0; i<N; ++i) insert(s,randval[i]);
}

template<class C> void do_erase(C& s, int N)
{
   for (int i=0; i<N; ++i) {
      auto p = s.begin();
      int count = randval2[i];
      while (count--) ++p;      //advance(p,randval2[i]) would optimize vector version
      s.erase(p);               // remove element at position N
   }
}

template<class C> void insert(C& s, int n)
{
   auto p = find_if(s.begin(),s.end(),[=](int i){ return i>n;});   // find first larger or end
   s.insert(p,n);
}
```

Vector vs. List



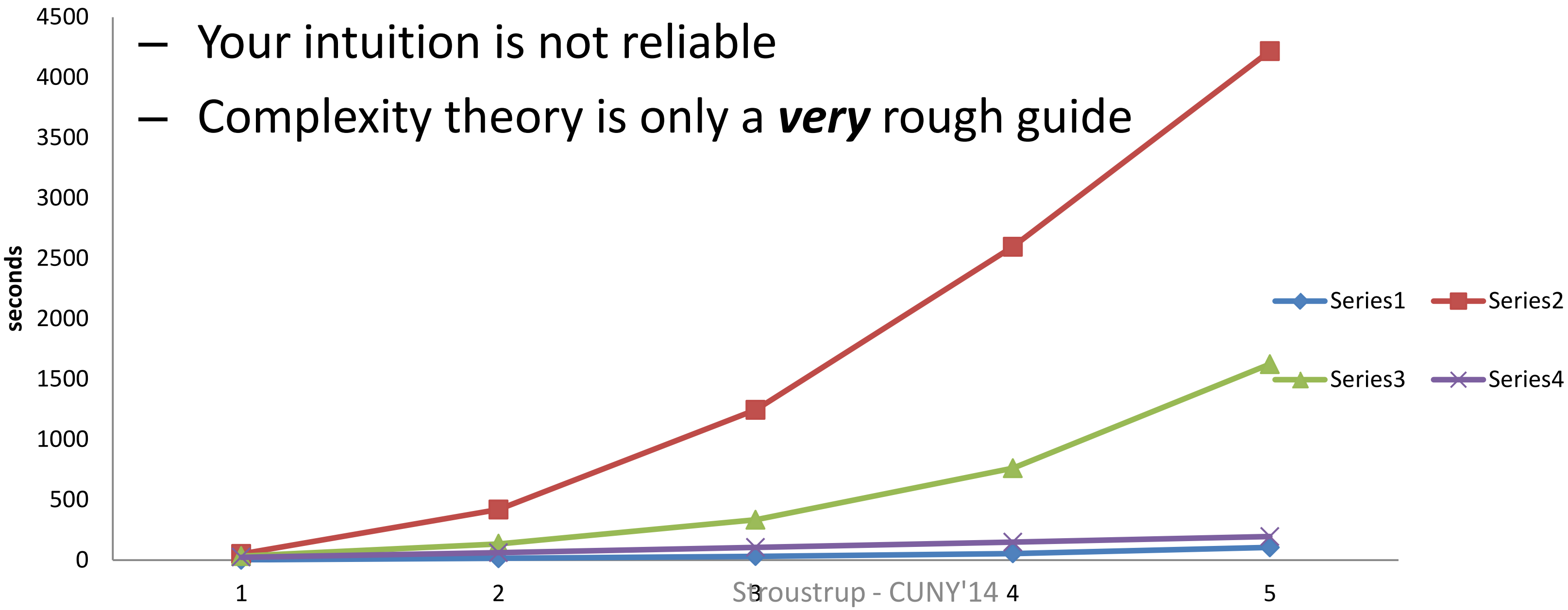
- The amount of memory used differ dramatically
  - List uses 4+ words per element
  - Vector uses 1 word per element
- An unoptimized list does more allocations
  - One per element
    - Can be eliminated by pre-allocation
- Memory access is relatively slow
  - Unpredictable memory access gives many more cache misses
- Find the insertion and deletion points
  - Linear search

This completely dominates

- Use a map?
  - But then we could use binary search and indexed access for a vector
  - Also: this misses the point. Don't just optimize for the exercise

Vector vs. List

- Implications:
  - Don’t store data unnecessarily.
  - Keep data compact.
  - Access memory in a predictable manner.
- Measure!
  - Your intuition is not reliable
  - Complexity theory is only a ***very*** rough guide



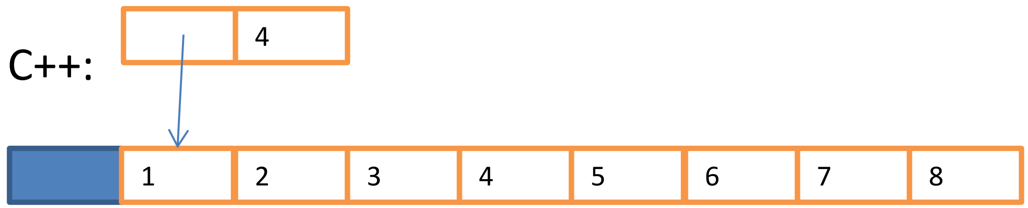
Experiment!

- Measure
- Control
- Simplify
- Reason

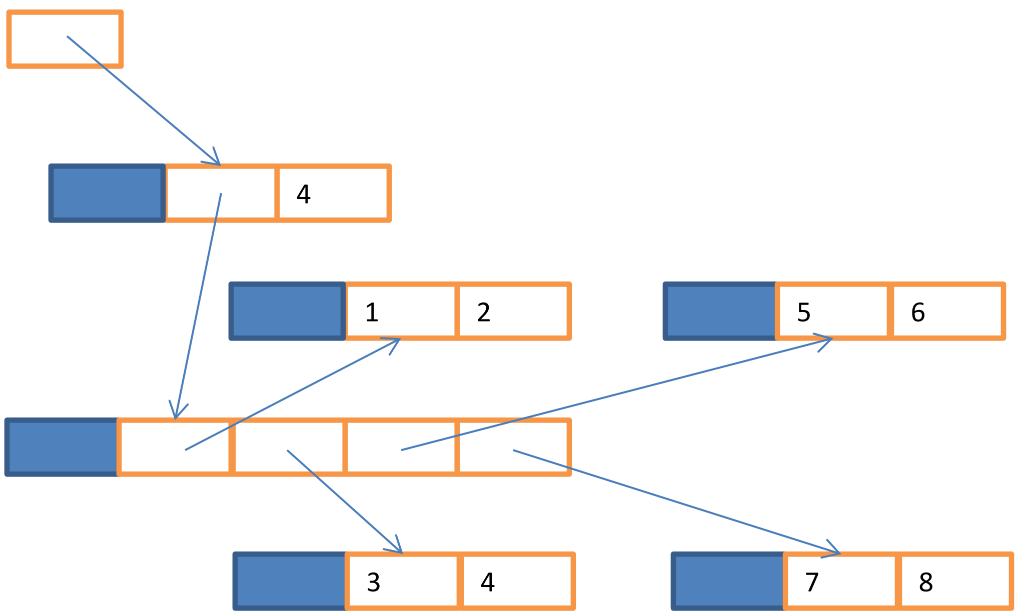
Use compact layout

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

C++:



“True OO” style:



C++98 to C++14

C++98:

```
circle* p = new circle(pnt,42);
vector<shape*> v = load_shapes();
for (vector<shape*>::iterator i = v.begin(); i != v.end(); ++i) {
    if (*i && **i == *p)
        cout << **i << " is a match\n";
}
```

not exception-safe

missing try/catch,

```
// ... later, possibly elsewhere ...
for( vector<shape*>::iterator i = v.begin();  i != v.end(); ++i
) {
    delete *i;
}
delete p;
```

Easy to forget;

easy to get wrong

C++14:

```
auto p = make_unique<circle>(pnt,42);
auto v = load_shapes();
for (s : v) {
    if (s && *s == *p)
        cout << *s << " is a match\n";
}
```

C++11/14

- It feels like a new language
  - I find it easier to express my ideas
  - My code is clearer and more compact
  - My programs are faster
- You get few benefits if you insist
  - Writing in 1970s C style
  - Writing in 1980s “Pure OO” style

Uniform initialization

- You can use **{}**-initialization for all types in all contexts

```
int a[] = { 1,2,3 };
vector<int> v { 1,2,3 };

vector<string> geek_heros = {
   "Dahl", "Kernighan", "McIlroy", "Nygaard ", "Ritchie", "Stepanov"
};

thread t {};    // default initialization
                // remember "thread t();" is a function declaration

complex<double> z {1,2};           // invokes constructor
struct S { double x, y; } s {1,2};  // no constructor (just initialize members)
```

Uniform initialization

- **{}**-initialization **X{v}** yields the same value of **X** in every context

```
X x{a};
X* p = new X{a};
z = X{a};                    // use as cast

void f(X);
f({a});                      // function argument (of type X)

X g() {
    // …
    return {a};              // function return value (function returning X)
}

Y::Y(a) : X{a} { /* … */ };          // base class initializer
```

auto

- Deduce a type of an object from its initializer

```
auto x = 1;            // x is an int
auto y = 1.2;          // y is a double
```

- Most useful when types gets hard to type or hard to know

```
template<class C>
void use(C& c)
{
	for (auto p = c.begin(); p!=c.end(); ++p)	// p is a ???
		cout << *p << '\n';
}
```

- Curio: The oldest C++11 feature
  - I implemented it in 1983/84

range-for

- Make the simplest loops simpler

```
template<class C>
void use(C& c)
{
	for (auto x : c)
			cout << x << ‘\n’;
}

for(auto x : { 1, 2, 5, 8, 13})
	test(x);
```

User-Defined Literals

- Examples
  - **"Hello! "** **// const char***
  - **"Howdy! "s** **// std::string**
  - **2.3*5.7i** **//** “**i**” for “imaginary”: a **complex** number
  - **4h+6min+3s** **//** 4 hours, 6 minutes, and 3 seconds
- Can be used for type-rich programming
  - **Speed s = 100m/9s;** **//** very fast for a human
  - **Acceleration a1 = s/9s;** **//** OK
  - **Acceleration a2 = s;** **//** error: unit mismatch
- Definition
  - **complex<double> operator "" i(long double d) { return {0,d}; }**

General constant expressions

- Think
  - ROM
  - concurrency
  - Compile-time computation (performance, compactness)
  - Type safety (reliability, maintainability)

```
constexpr int abs(int i) { return (0<=i) ? i : -i; } // can be constant expression

struct Point {
    int x, y;
    constexpr Point(int xx, int yy) : x{xx}, y{yy} { } // "literal type"
};

constexpr Point p1{1,2};             // must be evaluated at compile time: ok
constexpr Point p2{p1.y,abs(x)};     // ok?: is x is a constant expression?
```

Lambda expressions

- A lambda expression ("a lambda") is a use-once function object

```
template<class C, class Oper>
void for_all(C& c, Oper op)          // assume that C is a container of pointers
{
    for (auto& x : c)
        op(*x);   // pass op() a reference to each element pointed to
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));          // read shape from input

    for_all(v, [](Shape& s){ s.draw(); });          // draw_all()
    for_all(v, [](Shape& s){ s.rotate(45); });      // rotate_all(45)
}
```

Variadic templates

- Any number of arguments of any types

```
template <class F, class ...Args>             // thread constructor
      explicit thread(F&& f, Args&&... args); // argument types must
                                               // match the operation's
                                               // argument types

void f0();                    // no arguments
void f1(int);                 // one int argument

thread t1 {f0};
thread t2 {f0,1};                          // error: too many arguments
thread t3 {f1};                            // error: too few arguments
thread t4 {f1,1};
thread t5 {f1,1,2};                        // error: too many arguments
thread t3 {f1,"I'm being silly"};          // error: wrong type of argument
```

Template aliases

- Notation matters
- C++98 exposes all details when we use templates

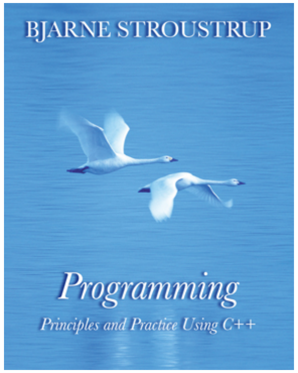
```
typename iterator_traits<For>::value_type x;
```

- C++11 allows us to hide details

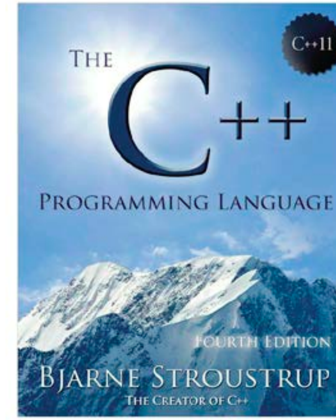
```
template<typename Iter>
using Value_type<T> = typename std::iterator_traits<For>::value_type;
// …
Value_type<For> x;
```

- Had I had an initializer, I could have used **auto**

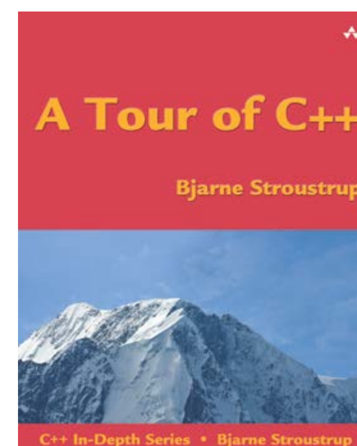
```
auto x = *p;
```



C++ Information



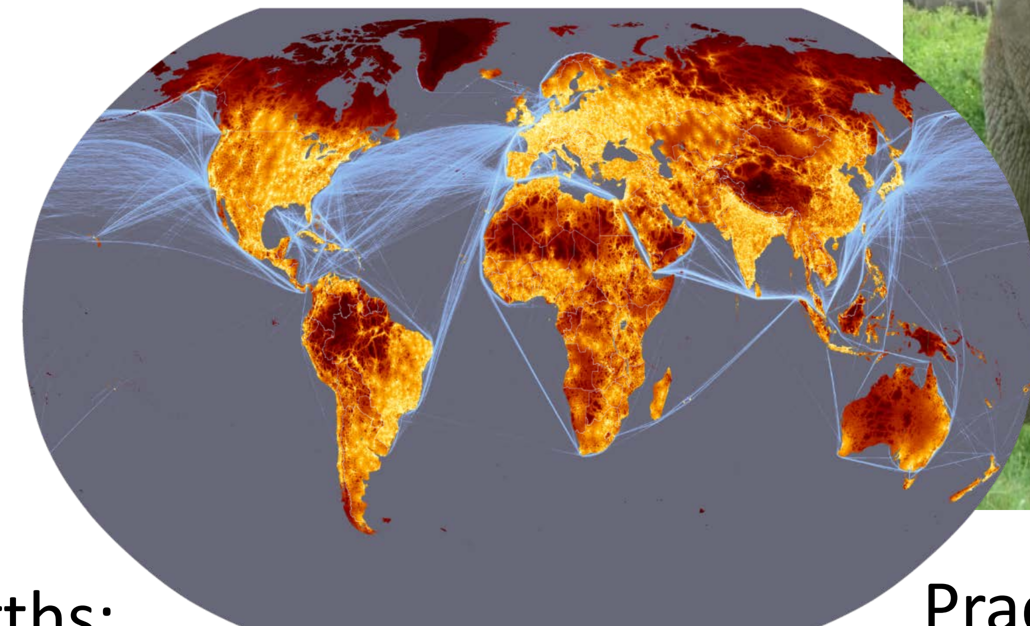
- www.isocpp.org
  - The C++ Foundation's website
  - Standards information, articles, user-group information
- Bjarne Stroustrup
  - *A Tour of C++*: All of C++ in 180 pages
  - *The C++ Programming Language (4th* edition): All of C++ in 1,300 pages
  - *Programming: Principles and Practice using C++* (2nd edition)
  - www.stroustrup.com: Publication list, C++ libraries, FAQs, etc.



- The ISO Standards Committee site
  - Search for "WG21"
  - *The ISO standard*: All of C++ in 1,300 pages of "standardese"
  - All committee documents (incl. proposals)

Questions?

C++: A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications

Practice type-rich programming