

# The Essence of C++

with examples in C++84, C++98, C++11, and C++14

Bjarne Stroustrup

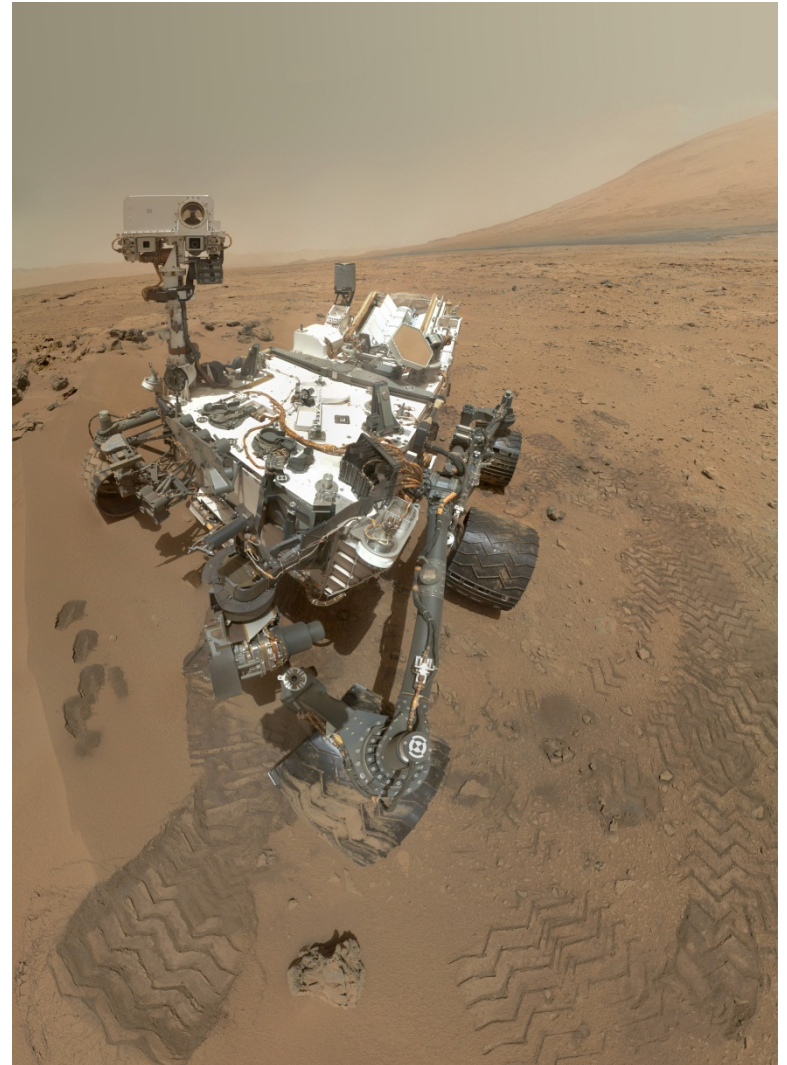
Morgan Stanley, Columbia University, Texas A&M University

[www.stroustrup.com](http://www.stroustrup.com)



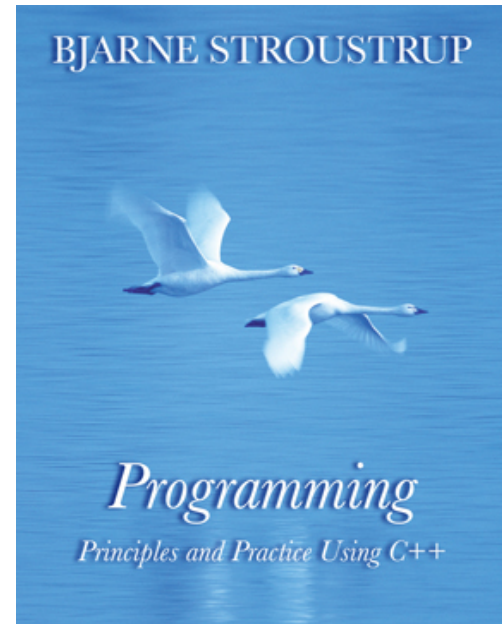
# Overview

- Aims and constraints
- C++ in four slides
- Resource management
  - RAI
  - Move semantics
- Generic Programming
  - Templates
  - Requirements checking



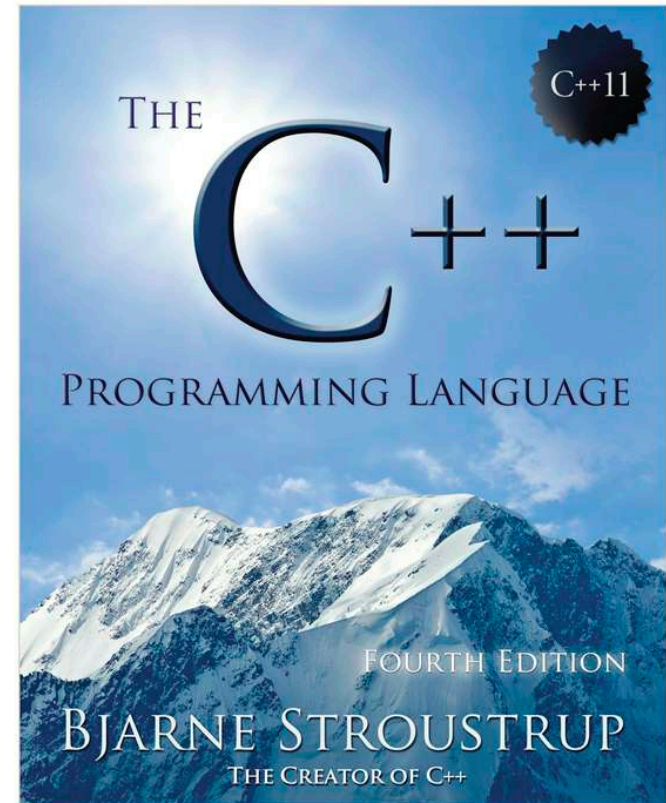
# What did/do I want?

- Type safety
  - Encapsulate necessary unsafe operations
- Resource safety
  - It's not all memory
- Performance
  - For some parts of almost all systems, it's important
- Predictability
  - For hard and soft real time
- Teachability
  - Complexity of code should be proportional to the complexity of the task
- Readability
  - People and machines (“analyzability”)



# Who did/do I want it for?

- Primary concerns
  - Systems programming
  - Embedded systems
  - Resource constrained systems
  - Large systems
- Experts
  - “C++ is expert friendly”
- Novices
  - C++ is not *just* expert friendly





# What is C++?

Template  
meta-programming!

Class hierarchies

A hybrid language

A multi-paradigm  
programming language

It's C!

Embedded systems  
programming language

Low level!

A random collection  
of features

Generic programming

An object-oriented  
programming language

Too big!

Classes

Buffer  
overflows



# C++

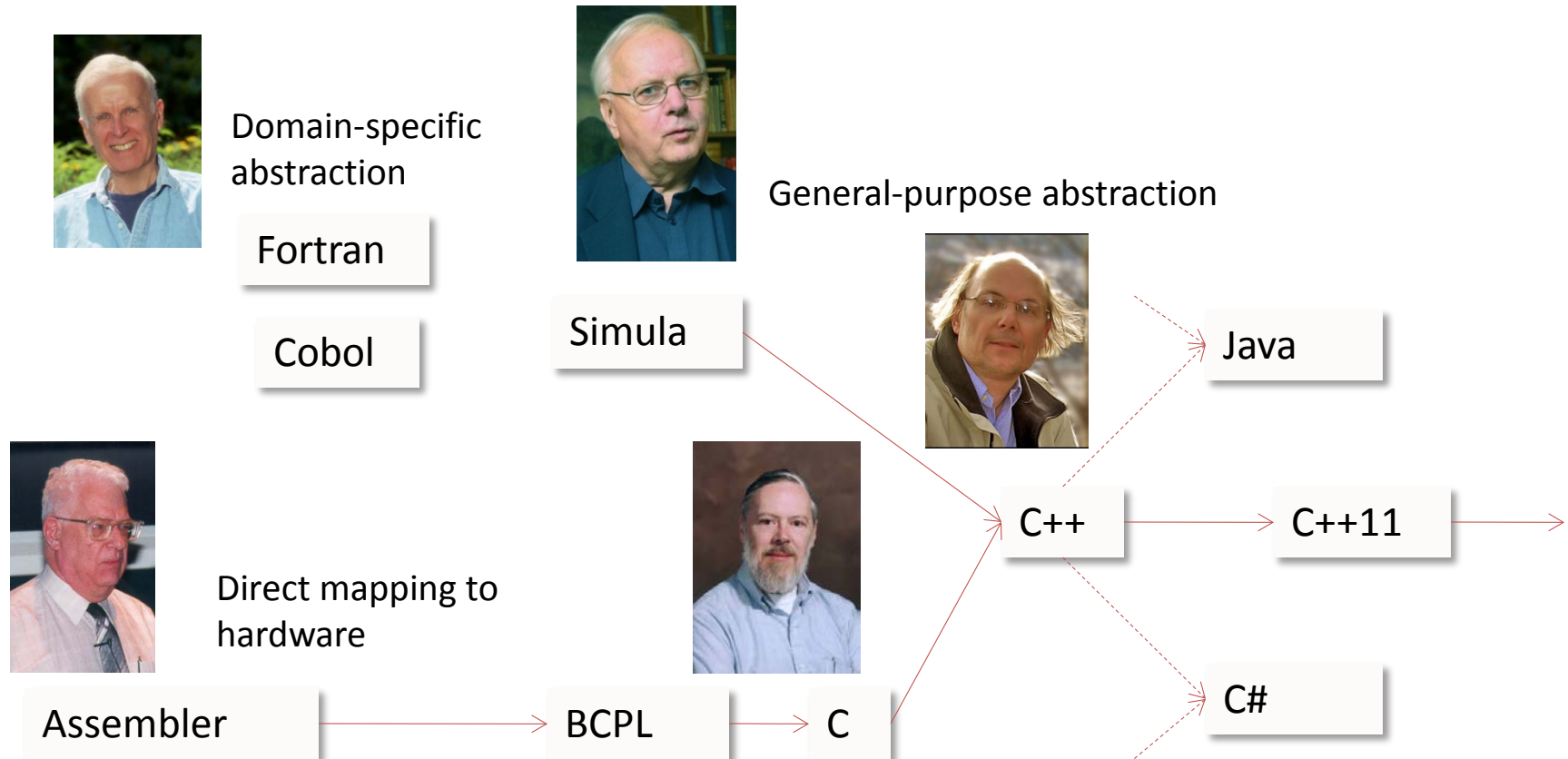
A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications

# Programming Languages



# What does C++ offer?

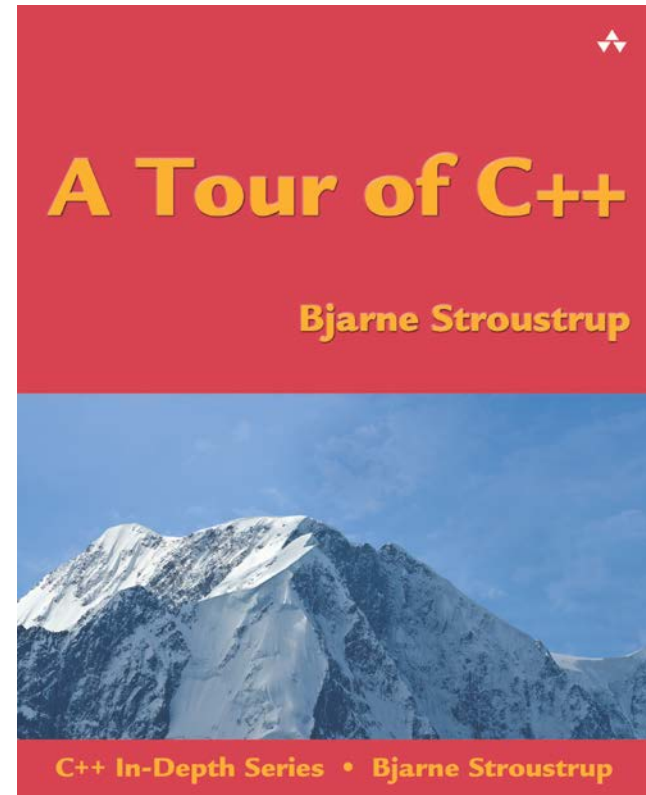
- Not perfection
  - Of course
- Not everything for everybody
  - Of course
- A solid fundamental model
  - Yes, really
- 30+ years of real-world “refinement”
  - It works
- Performance
  - A match for anything
- The best is buried in “compatibility stuff”
  - long-term stability is a feature





# What does C++ offer?

- C++ in Four slides
  - Map to hardware
  - Classes
  - Inheritance
  - Parameterized types



- If you understand **int** and **vector**, you understand C++
  - The rest is “details” (1,300+ pages of details)

# Map to Hardware

- Primitive operations => instructions

- +, %, ->, [], (), ...

value

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

handle

- **vector**, string, thread, Matrix, ...

value

- Objects can be composed by simple concatenation:

- Arrays

- Classes/structs

value

value

handle

handle

value

value

# Classes: Construction/Destruction

- From the first week of “C with Classes” (1979)

```
class X {                                // user-defined type
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.”

# Abstract 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;  
};
```

- 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)
- Is the root of a hierarchy of derived classes

# Parameterized Types and Classes

- Templates
  - Essential: Support for generic programming
  - Secondary: Support for compile-time computation

```
template<typename T>
```

```
class vector { /* ... */ };           // a generic type
```

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

```
template<typename C>
```

```
void sort (C& c) { /* ... */ }       // a generic function
```

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



# Not C++ (fundamental)

- No crucial dependence on a garbage collector
  - GC is a last and imperfect resort
- No guaranteed type safety
  - Not for all constructs, but most code can be type safe
  - C compatibility, history, pointers/arrays, unions, casts, ...
- No virtual machine required
  - For many reasons, we often want to run on the real machine
  - You can run on a virtual machine (or in a sandbox) if you want to



# Not C++ (market realities)

- Lots and lots of libraries
- No huge “standard” library
  - No owner
    - To produce “free” libraries to ensure market share
  - No central authority
    - To approve, reject, and help integration of libraries
- No standard
  - Graphics/GUI
    - Competing frameworks
  - XML support
  - Web support
  - ...



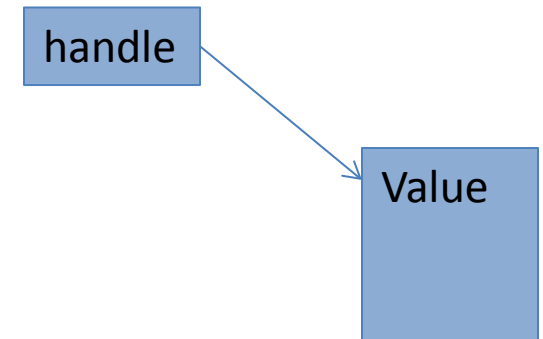
[illegible]

# Resource management

- A resource should be owned by a “handle”
  - A “handle” should present a well-defined and useful abstraction
    - E.g. a vector, string, file, thread
- Use constructors and a destructor

```
class Vector {      // vector of doubles
    Vector(initializer_list<double>); // acquire memory; initialize elements
    ~Vector();       // destroy elements; release memory
    // ...
private:
    double* elem;    // pointer to elements
    int sz;          // number of elements
};

void fct()
{
    Vector v {1, 1.618, 3.14, 2.99e8}; // vector of doubles
    // ...
}
```



# Resource management

- A handle usually is scoped
  - Handles lifetime (initialization, cleanup), and more

```
Vector::Vector(initializer_list<double> lst)
    :elem {new double[lst.size()]}, sz{lst.size()};    // acquire memory
{
    uninitialized_copy(lst.begin(),lst.end(),elem);    // initialize elements
}
```

```
Vector::~~Vector()
{
    delete[] elem;    // destroy elements; release memory
};
```



# Resource management

- What about errors?
  - A resource is something you acquire and release
  - A resource should have an owner
  - Ultimately “root” a resource in a (scoped) handle
  - “Resource Acquisition Is Initialization” (RAII)
    - Acquire during construction
    - Release in destructor
  - Throw exception in case of failure
    - Can be simulated, but not conveniently
  - Never throw while holding a resource **not** owned by a handle
- In general
  - Leave established invariants intact when leaving a scope

# “Resource Acquisition is Initialization” (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 mtx;      // a resource  
int sh;              // shared data
```

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

# Pointer Misuse

- Many (most?) direct 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::runtime_error{"Weird!"}; // leak
    if (x<200) return;                     // leak
    // ...
    delete p;                             // and I want my garbage collector! ☹️
}
```

- But, garbage collection would not release non-memory resources anyway
- But, why use a “naked” 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::runtime_error{"Weird!"};    // no leak
    if (x<200) return;                                // no leak
    // ...
}
```

- `shared_ptr` provides a form of garbage collection
- But I'm not sharing anything
  - use a `unique_ptr`

# 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::runtime_error{"Weird!"};    // no leak
    if (x<200) return;                                // no leak
    // ...
}
```



# Why do we use pointers?

- And references, iterators, etc.
- To represent ownership
  - **Don't!** Instead, use handles
- To reference resources
  - from within a handle
- To represent positions
  - Be careful
- To pass large amounts of data (into a function)
  - E.g. pass by **const** reference
- To return large amount of data (out of a function)
  - **Don't!** Instead use move operations

# How to get a lot of data cheaply out of a function?

- Ideas
  - Return a pointer to a **new**'d object
    - Who does the **delete**?
  - Return a reference to a **new**'d object
    - Who does the **delete**?
    - Delete what?
  - Pass a target object
    - We are regressing towards assembly code
  - Return an object
    - Copies are expensive
    - Tricks to avoid copying are brittle
    - Tricks to avoid copying are not general
  - Return a handle
    - Simple and cheap

# Move semantics

- Return a **Matrix**

```
Matrix operator+(const Matrix& a, const Matrix& b)
```

```
{
```

```
    Matrix r;
```

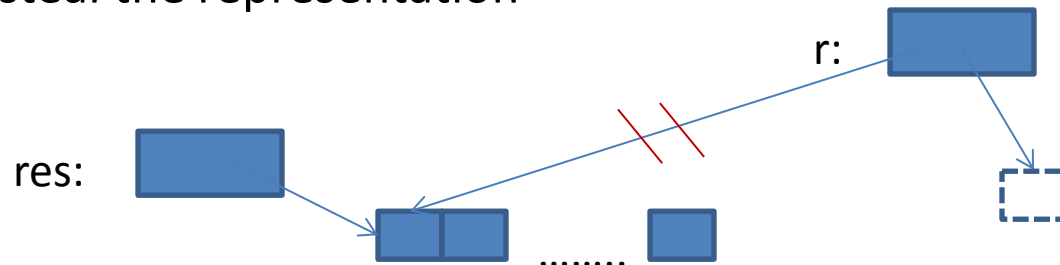
```
    // copy a[i]+b[i] into r[i] for each i
```

```
    return r;
```

```
}
```

```
Matrix res = a+b;
```

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

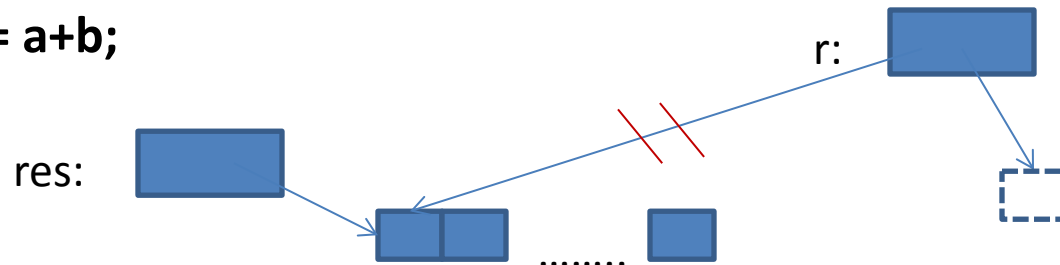


# Move semantics

- 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;



# No garbage collection needed

- For general, simple, implicit, and efficient resource management
- Apply these techniques in order:
  1. Store data in containers
    - The semantics of the fundamental abstraction is reflected in the interface
    - Including lifetime
  2. Manage **all** resources with resource handles
    - RAI
    - Not just memory: **all** resources
  3. Use “smart pointers”
    - They are still pointers
  4. Plug in a garbage collector
    - For “litter collection”
    - C++11 specifies an interface
    - Can still leak non-memory resources



# Range-for, auto, and move

- As ever, what matters is how features work in combination

```
template<typename C, typename V>  
vector<Value_type<C>*> find_all(C& c, V v) // find all occurrences of v in c  
{  
    vector<Value_type<C>*> res;  
    for (auto& x : c)  
        if (x==v)  
            res.push_back(&x);  
    return res;  
}
```

```
string m {"Mary had a little lamb"};  
for (const auto p : find_all(m,'a')) // p is a char*  
    if (*p!='a')  
        cerr << "string bug!\n";
```

# RAII and Move Semantics

- All the standard-library containers provide it
  - **vector**
  - **list, forward\_list** (singly-linked list), ...
  - **map, unordered\_map** (hash table),...
  - **set, multi\_set, ...**
  - ...
  - **string**
- So do other standard resources
  - **thread, lock\_guard, ...**
  - **istream, fstream, ...**
  - **unique\_ptr, shared\_ptr**
  - ...



# GP



Stroustrup - Essence, short - Columbia'14

# Generic Programming: Templates

- 1980: Use macros to express generic types and functions
- 1987 (and current) aims:
  - Extremely general/flexible
    - “must be able to do much more than I can imagine”
  - Zero-overhead
    - vector/Matrix/... to compete with C arrays
  - Well-specified interfaces
    - Implying overloading, good error messages, and maybe separate compilation
- “two out of three ain’t bad”
  - But it isn’t great either
  - it has kept me concerned/working for 20+ years

# Templates

- Compile-time duck typing
  - Leading to template metaprogramming
- A massive success in C++98, better in C++11, better still in C++14
  - STL containers
    - **template<typename T> class vector { /\* ... \*/ };**
  - STL algorithms
    - **sort(v.begin(),v.end());**
  - And much more
- Better support for compile-time programming
  - C++11: **constexpr** (improved in C++14)

# Containers and Algorithms

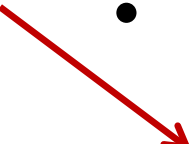
- The C++ standard-library algorithms are expressed in terms of half-open sequences [first:last)
  - For generality and efficiency
  - If you find that verbose, define container algorithms

```
namespace Extended_STL {  
    // ...  
    template<typename C, typename Predicate>  
    Iterator<C> find_if(C& c, Predicate pred)  
    {  
        return std::find_if(c.begin(),c.end(),pred);  
    }  
    // ...  
}
```

```
auto p = find_if(v, [](int x) { return x%2; } );    // assuming v is a vector<int>
```

# Duck Typing is Insufficient



- There are no proper interfaces
  - Leaves error detection far too late
    - Compile- and link-time in C++
  - Encourages a focus on implementation details
    - Entangles users with implementation
  - Leads to over-general interfaces and data structures
    - As programmers rely on exposed implementation “details”
  - Does not integrate well with other parts of the language
    - Teaching and maintenance problems
  - We must think of generic code in ways similar to other code
    - Relying on well-specified interfaces (like OO, etc.)
- 

# Using Constraints: Concept

- We must specify what we expect of template arguments
- Concept:
  - A set of requirements on one or more template arguments
  - A compile-time predicate on a set of types and values
  - For example
    - **Sequence<T>** is T a sequence type
    - **Container<T>** is T a container type
    - **Forward\_iterator<T>** is T a forward iterator
    - **Integer<T>** is T and integer type
    - **Function<T,A>** can a T be called with an argument of type A
  - Use
    - **template<typename C> requires Container<C>() void sort(C& c);**
    - **template<Container C> void sort(C& c);** *// shorthand notation*
    - **void sort(Container& c);** *// terse notation*





# Generic Programming is just Programming

- *Traditional code*

```
double sqrt(double d);    // C++84: accept any d that is a double  
double d = 7;  
double d2 = sqrt(d);     // fine: d is a double  
double d3 = sqrt(&d);    // error: &d is not a double
```

- *Generic code*

```
void sort(Container& c);  // C++14: accept any c that is a Container  
vector<string> vs { "Hello", "new", "World" };  
sort(vs);               // fine: vs is a Container  
sort(&vs);              // error: &vs is not a Container
```

# C++14 Concepts: Error handling

- Error handling is simple (and fast)

```
template<Sortable Cont>  
    void sort(Cont& container);
```

```
vector<double> vec {1.2, 4.5, 0.5, -1.2};  
list<int> lst {1, 3, 5, 4, 6, 8, 2};
```

```
sort(vec);      // OK: a vector is Sortable  
sort(lst);      // Error at (this) point of use: Sortable requires random access
```

- **Actual** error message  
error: 'list<int>' does not satisfy the constraint 'Sortable'

# C++14 Concepts: Overloading

- Overloading is easy

```
template <Sequence S, Equality_comparable<Value_type<S>> T>  
    Iterator_of<S> find(S& seq, const T& value);
```

```
template<Associative_container C>  
    Iterator_type<C> find(C& assoc, const Key_type<C>& key);
```

```
vector<int> v { /* ... */ };
```

```
multiset<int> s { /* ... */ };
```

```
auto vi = find(v, 42);
```

*// calls 1st overload:*

*// a vector is a Sequence*

```
auto si = find(s, 12-12-12);
```

*// calls 2nd overload:*

*// a multiset is an Associative\_container*

# C++14 Concepts

- We have reached the conventional notation
  - with the conventional meaning

- *Traditional code*

```
double sqrt(double d);    // C++84: accept any d that is a double  
double d = 7;  
double d2 = sqrt(d);     // fine: d is a double  
double d3 = sqrt(&d);    // error: &d is not a double
```

- *Generic code*

```
void sort(Container& c);  // C++14: accept any c that is a Container  
vector<string> vs { "Hello", "new", "World" };  
sort(vs);                // fine: vs is a Container  
sort(&vs);               // error: &vs is not a Container
```

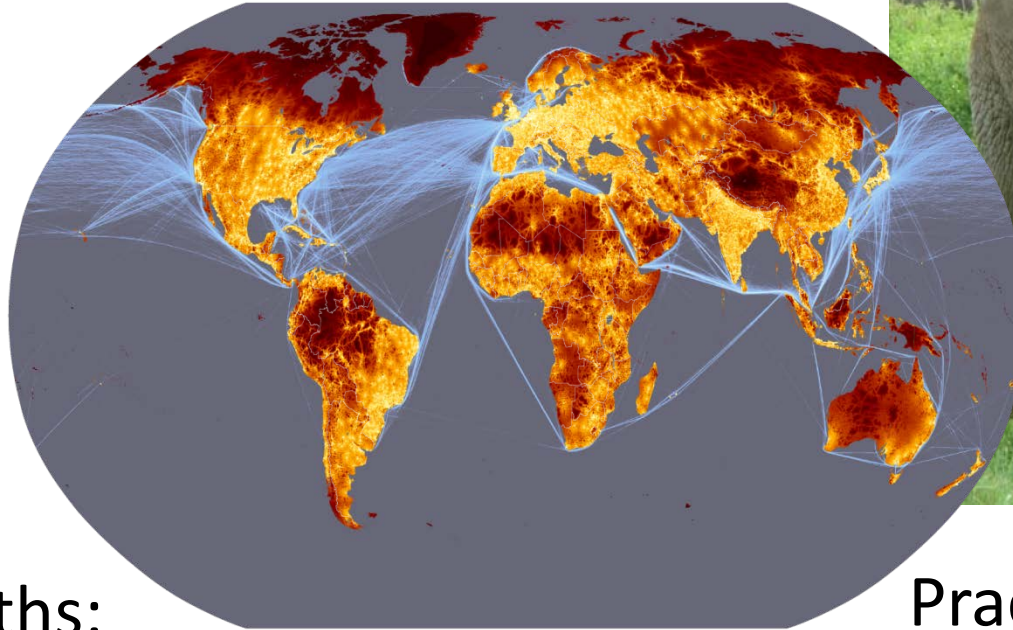
# “Paradigms”

- Much of the distinction between object-oriented programming, generic programming, and “conventional programming” is an illusion
  - based on a focus on language features
  - incomplete support for a synthesis of techniques
  - The distinction does harm
    - by limiting programmers, forcing workarounds

```
void draw_all(Container& c)    // is this OOP, GP, or conventional?  
    requires Same_type<Value_type<Container>,Shape*>  
{  
    for_each(c, [](Shape* p) { p->draw(); } );  
}
```

# Questions?

C++: A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications

Practice type-rich programming