

# CS241 Problem solving and its applications

## Lecture 2: Review on C++ II

Instructor: Yuye Ling, PhD

Shanghai Jiao Tong University  
John Hopcroft Center for Computer Science  
Department of Electronic Engineering

# The Essence of C++

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

Bjarne Stroustrup

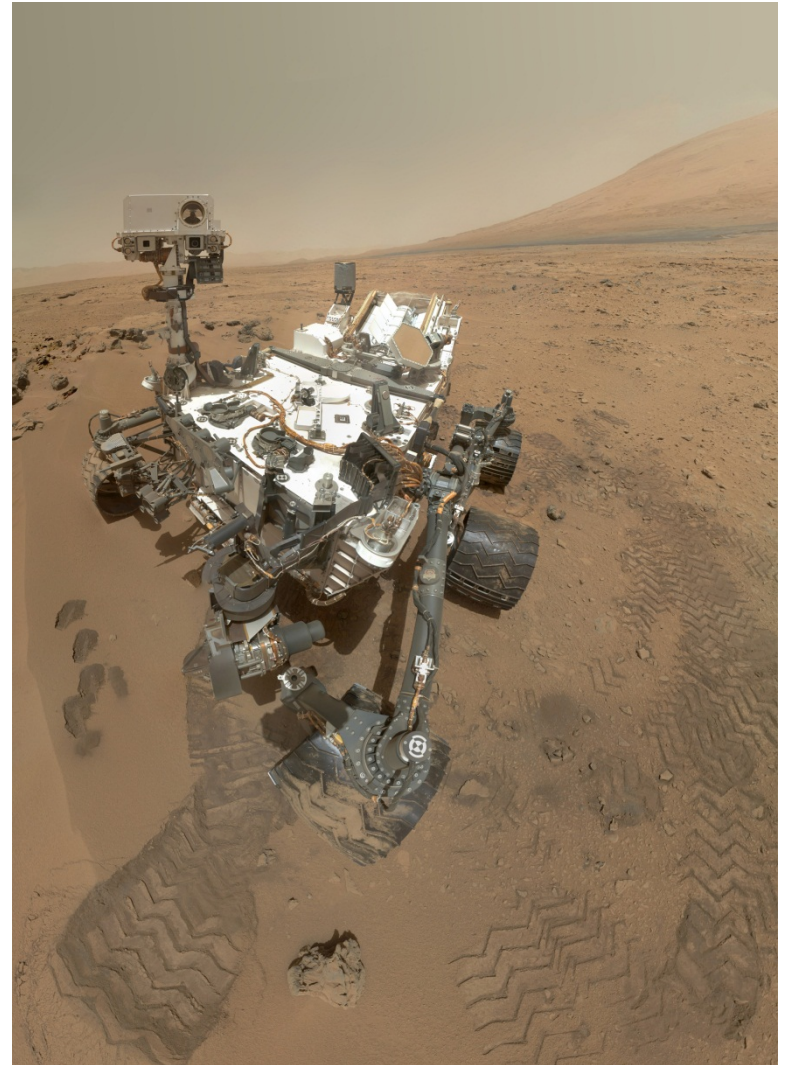
Texas A&M University

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



# Overview

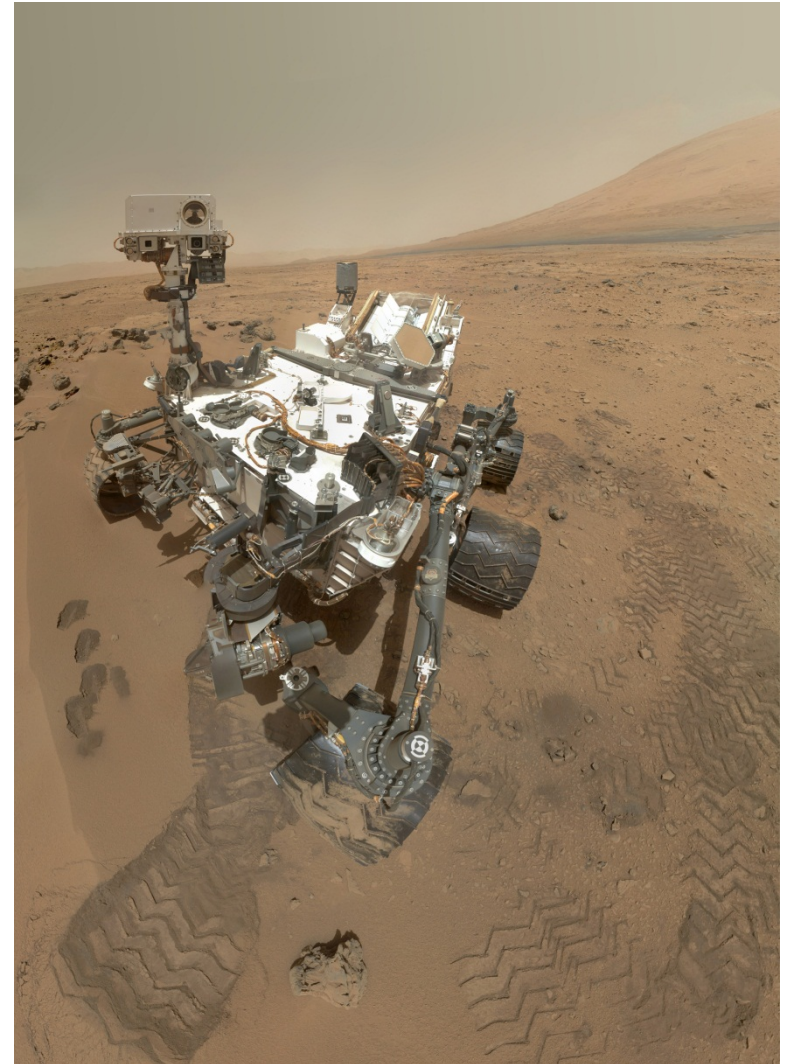
- Aims and constraints
- C++ in four slides
- Resource management
- ~~• OOP: Classes and Hierarchies~~
  - (very briefly)
- GP: Templates
  - Requirements checking
- ~~• Challenges~~



# Overview

## ➤ Aims and constraints

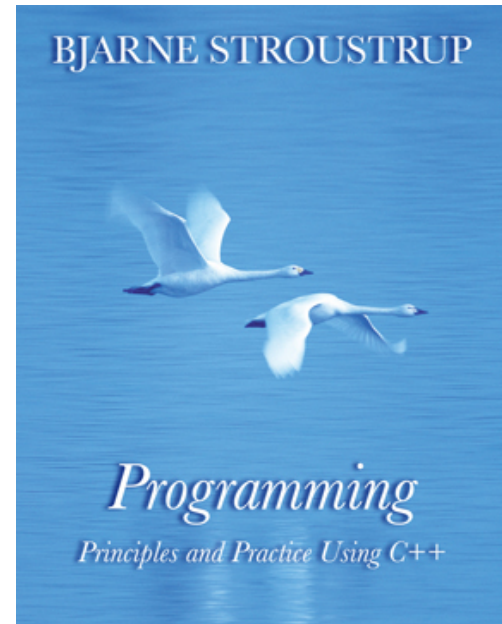
- C++ in four slides
- Resource management
- ~~• OOP: Classes and Hierarchies~~
  - (very briefly)
- GP: Templates
  - Requirements checking
- ~~• Challenges~~



# What did/do I want?

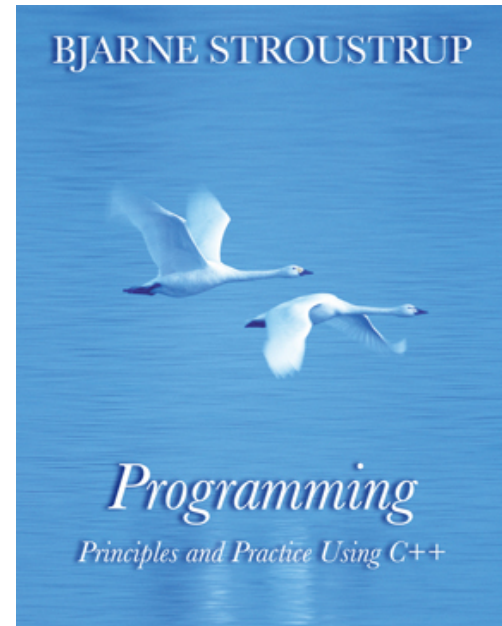
- Type safety
  - Encapsulate necessary unsafe operations
  - `printf`

```
1 int main()  
2 {  
3     printf("%s\n", 10);  
4     system("pause");  
5     return 0;  
6 }  
7
```



# What did/do I want?

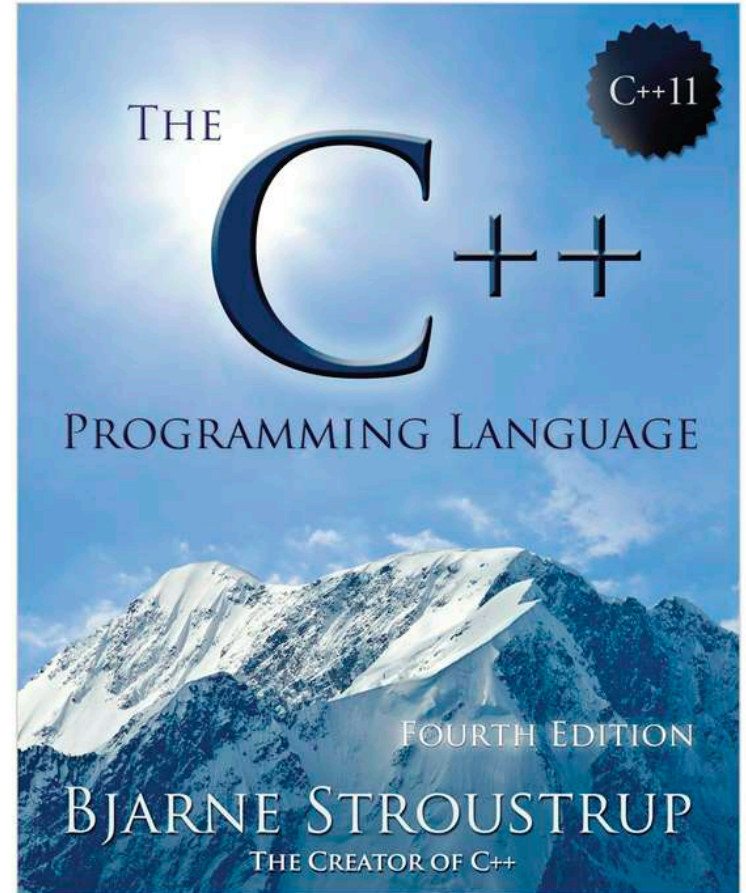
- Type safety
  - Encapsulate necessary unsafe operations
- Resource safety
  - It's not all memory
- Performance
  - For some parts of almost all systems, it's important
- Predictability
  - For hard and soft real time
- Teachability
  - Complexity of code should be proportional to the complexity of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

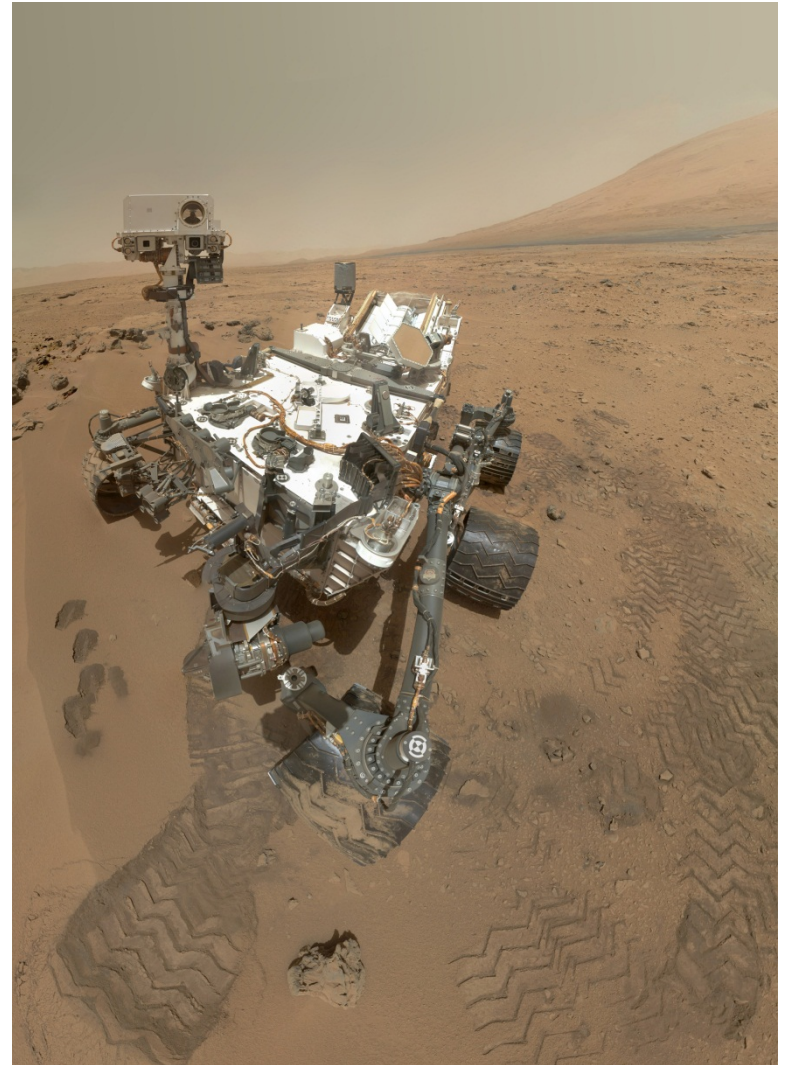
# What does C++ offer?

- 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



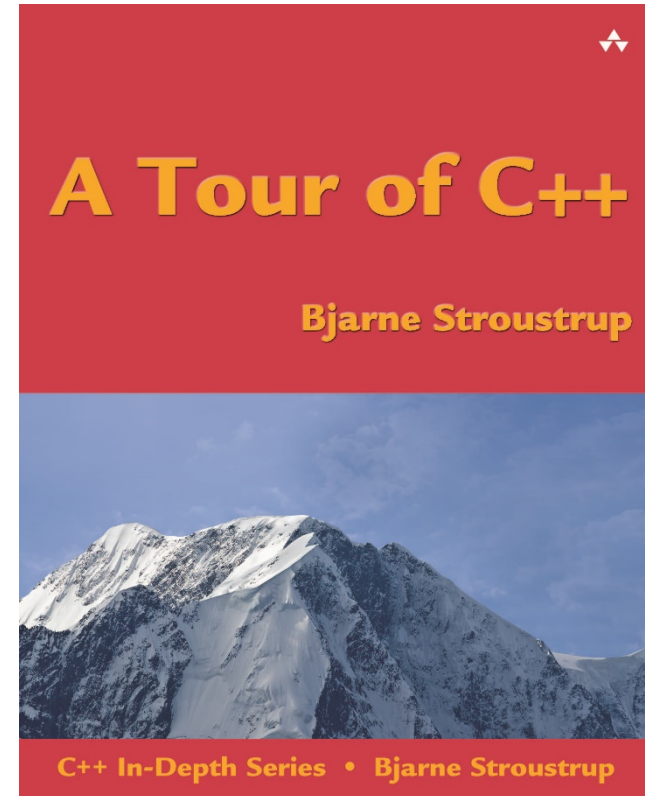
# Overview

- Aims and constraints
- C++ in four slides
- Resource management
- ~~• OOP: Classes and Hierarchies~~
  - (very briefly)
- GP: Templates
  - Requirements checking
- ~~• Challenges~~



# What does C++ offer?

- C++ in Four slides
  - Map to hardware **Low level**
  - Classes **OOP**
  - Inheritance **Polymorphism**
  - Parameterized types **GP**



# Map to Hardware

- Primitive operations => instructions

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

value

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

handle

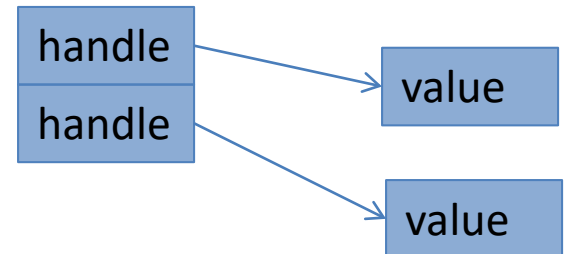
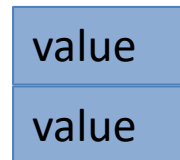
- **vector**, string, thread, Matrix, ...

value

- Objects can be composed by simple concatenation:

- Arrays

- Classes/structs





# 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

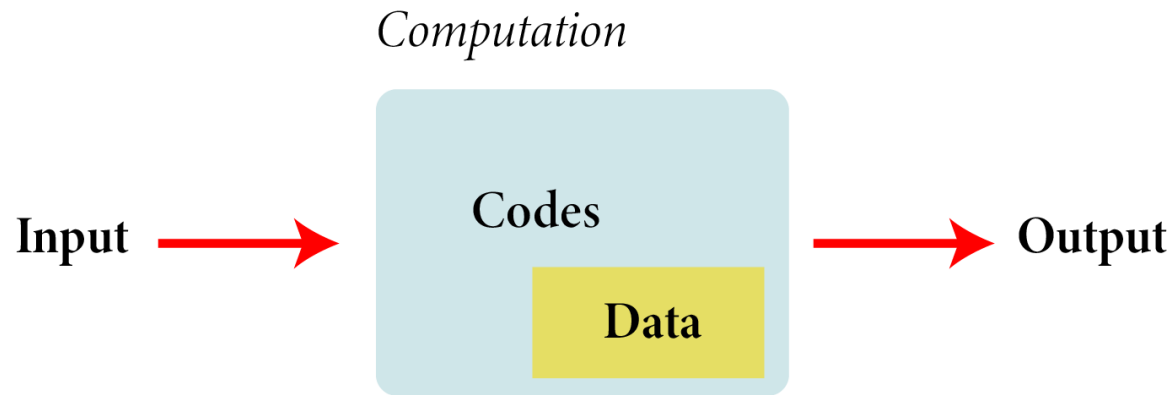
- Insulate the user from the implementation

```
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
- 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
- **Why we need such a hierarchy?**
- Support run-time polymorphism (through virtual function)

# Parameterized Types and Classes

- Templates
  - Essential: Support for generic programming



- Separate the algorithms from the data structure

# Parameterized Types and Classes

- Templates

- Essential: Support for generic programming

```
template<typename T>
```

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

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

```
template<typename C>
```

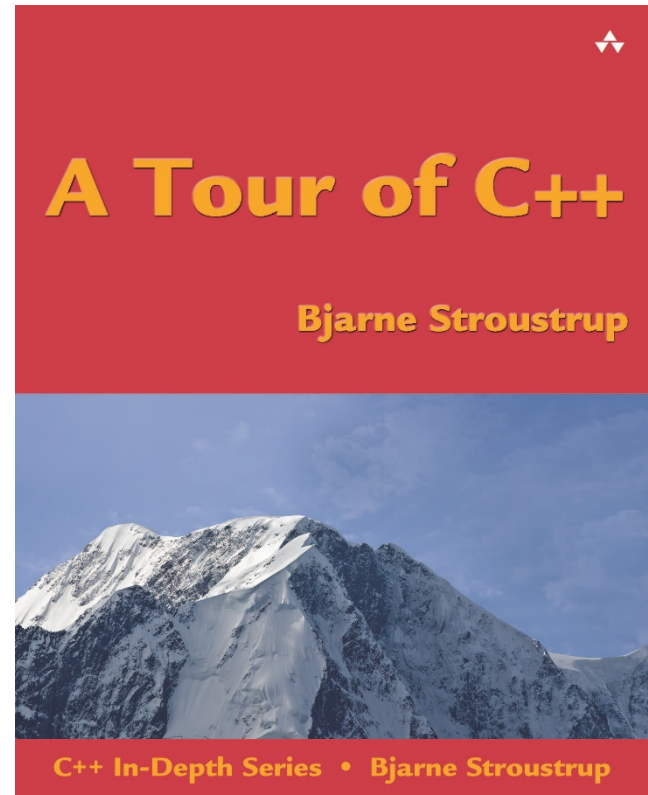
```
void sort (Const& c) { /* ... */ }    // a generic function
```

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

- Secondary: Support for compile-time computation

# 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)



# 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
  - C compatibility, history, pointers/arrays, unions, casts, ...
- No virtual machine
  - 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)

- 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
  - ...



# Overview

- Aims and constraints
- C++ in four slides
- Resource management
- ~~• OOP: Classes and Hierarchies~~
  - (very briefly)
- GP: Templates
  - Requirements checking
- ~~• Challenges~~





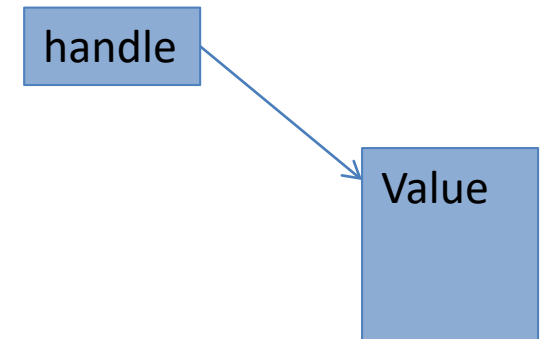
[illegible]

# 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
  - 一言以蔽之： use class to encapsulate resources
- 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 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`), ...
- The handle’s constructor acquires the resource and the handle’s destructor release it
- What will be the benefits?

```
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?) uses of pointers in local scope are not **exception** safe

```
void f(int n, int x)
{
    Gadget* p = new Gadget{n};
    // ...
    if (x<100) throw std::runtime_error{"Weird!"};           // leak
    if (x<200) return;                                       // leak
    // ...
    delete p;
}
```

- But, garbage collection would not release non-memory resources anyway
- But, why use a plain 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?
- **What is the essence of using pointer?**
- Do we really need to dynamically allocate the memory?
- 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

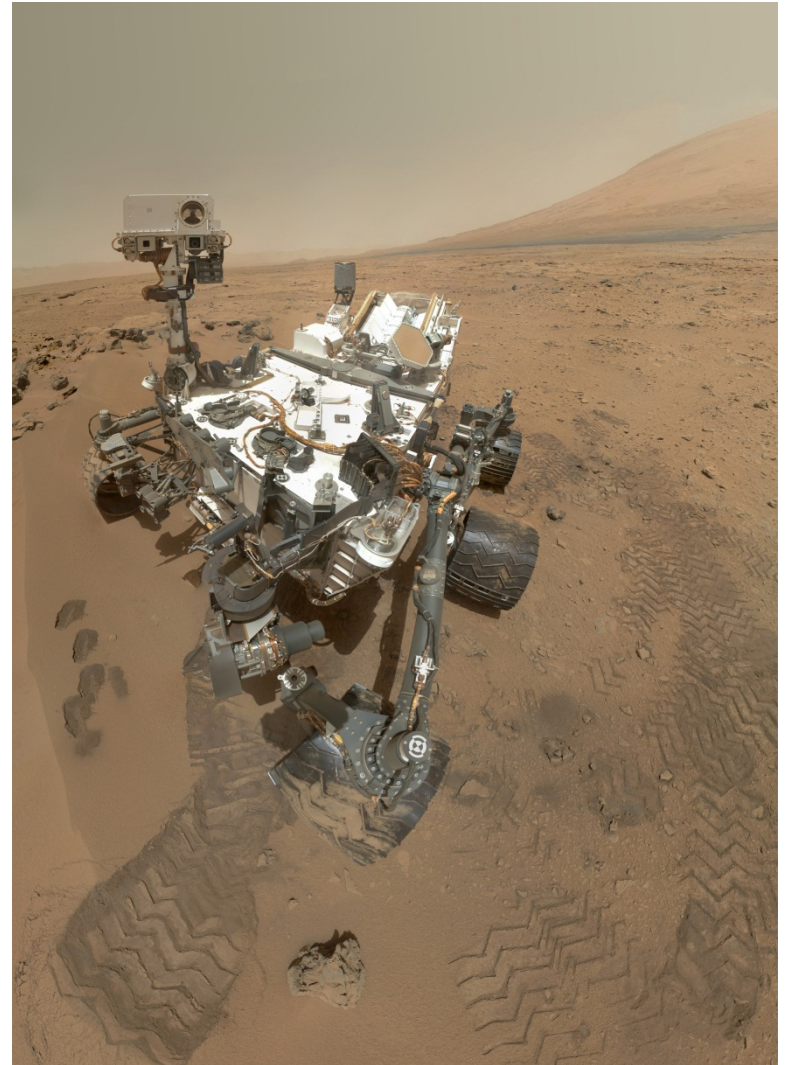
# No garbage collection needed

- 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



# Overview

- Aims and constraints
- C++ in four slides
- Resource management
- ~~• OOP: Classes and Hierarchies~~
  - (very briefly)
- GP: Templates
  - Requirements checking
- ~~• Challenges~~





# GP



# Generic Programming: Templates

- 1980: Use macros to express generic types and functions
  - Again, what is macro?
  - We could not only define constants but also **functions** (available in C)

```
1 #include <stdio.h>
1 #include <stdio.h>
2 #define MULTIPLY(a, b) a*b
3 int main()
4 {
5     printf("%d", MULTIPLY(2+3, 3+5));
6     return 0;
7 }
8
```

- What if we want to invoke  $5*8$ ?

# 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 really good 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)

# Algorithms

- Messy code is a major source of errors and inefficiencies
- We must use more explicit, well-designed, and tested algorithms
- The C++ standard-library algorithms are expressed in terms of half-open sequences [**first:last**)
  - For generality and efficiency

```
void f(vector<int>& v, list<string>& lst)
```

```
{
```

```
    sort(v.begin(),v.end());
```

```
    auto p = find(lst.begin(),lst.end(),"Shanghai"); // find "Shanghai" in the list
```

```
    // ...
```

```
}
```

- We parameterize over element type and container type

# Algorithms

- Simple, efficient, and general implementation
  - For any forward iterator
    - An iterator is something that points to an element of a sequence.
  - For any (matching) value type

```
template<typename Iter, typename Value>
```

```
Iter find(Iter first, Iter last, Value val) // find first p in [first:last) so that *p==val  
{  
    while (first!=last && *first!=val)  
        ++first;  
    return first;  
}
```

# Algorithms and Function Objects

- Parameterization with criteria, actions, and algorithms
  - Essential for flexibility and performance

```
void g(vector< string>& vs)
{
    auto p = find_if(vs.begin(), vs.end(), Less_than{"Griffin"});

    // ...
}
```

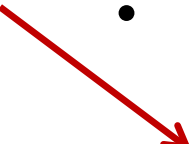
# Algorithms and Function Objects

- The implementation is still trivial

```
template<typename Iter, typename Predicate>
Iter find_if(Iter first, Iter last, Predicate pred) // find first p in [first:last) so that pred(*p)
{
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

# 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.)
- 

# 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
```

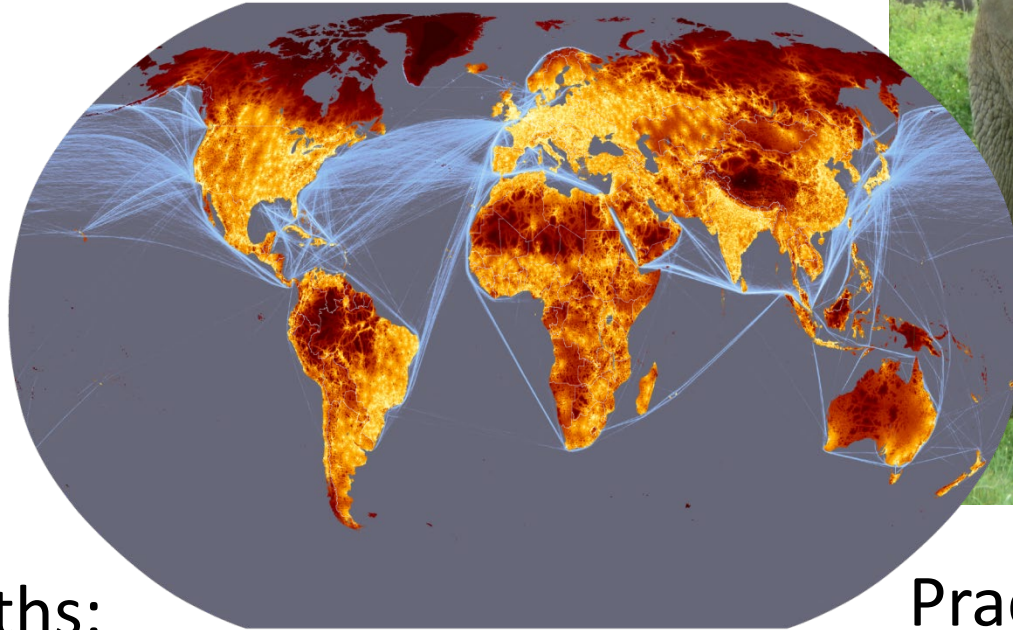
- *Generic code*

```
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
```



# Questions?

C++: A light-weight abstraction programming language



Key strengths:

- software infrastructure
- resource-constrained applications

Practice type-rich programming