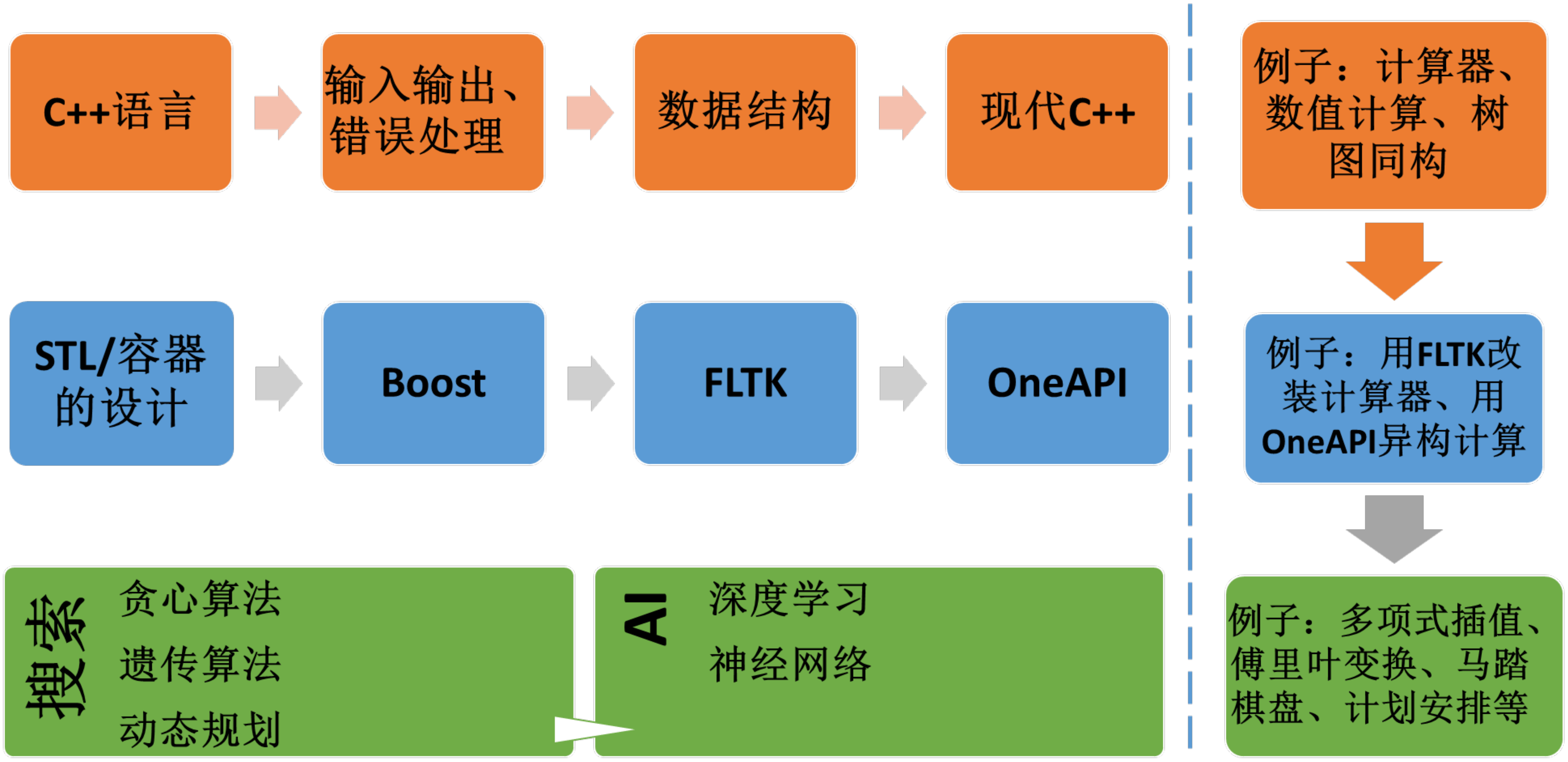

C++语言
输入输出、错误处理
数据结构
现代C++
例子：计算器、数值计算、树图同构
STL/容器的设计
Boost
FLTK
OneAPI
例子：用FLTK改装计算器、用OneAPI异构计算
搜索
贪心算法
遗传算法
动态规划
AI
深度学习
神经网络
例子：多项式插值、傅里叶变换、马踏棋盘、计划安排等

问题求解与实践
——C++回顾

主讲教师： 陈雨亭、沈艳艳

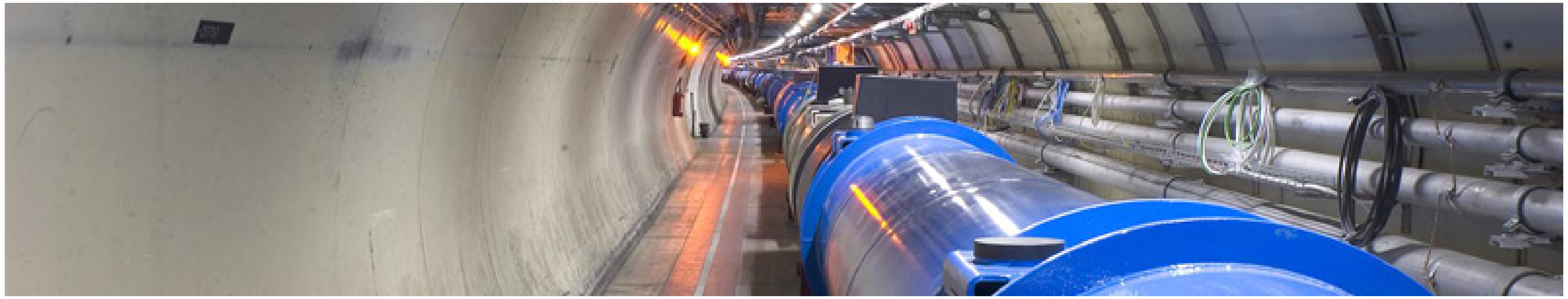
The Essence of C++

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

Bjarne Stroustrup

Texas A&M University

www.stroustrup.com



Overview

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

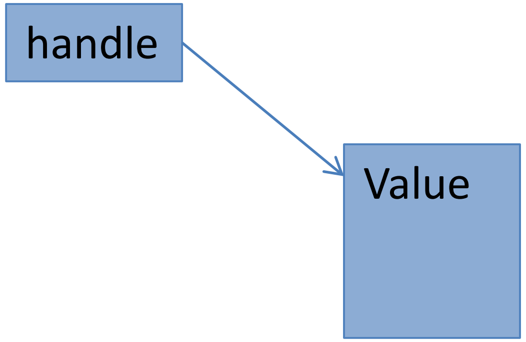
Resource Management

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?) 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

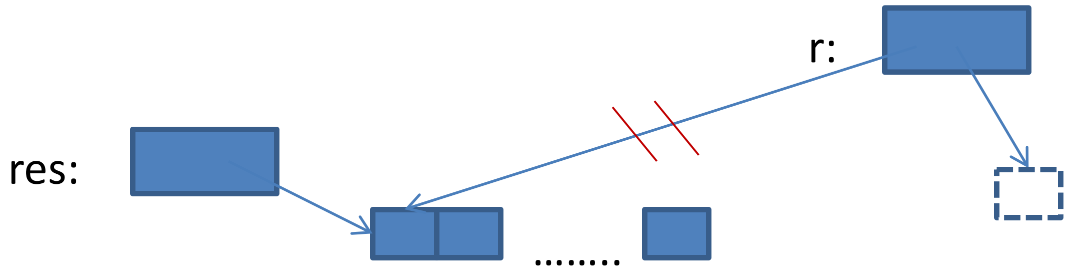
How to get a lot of data cheaply out of a function?

- 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;
```

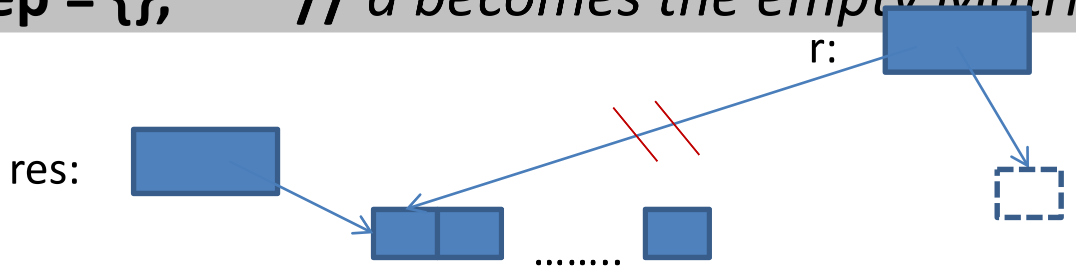


- Define move a constructor for **Matrix**
  - don’t copy; “steal the representation”

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**
     - **RAII**
     - **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";
```

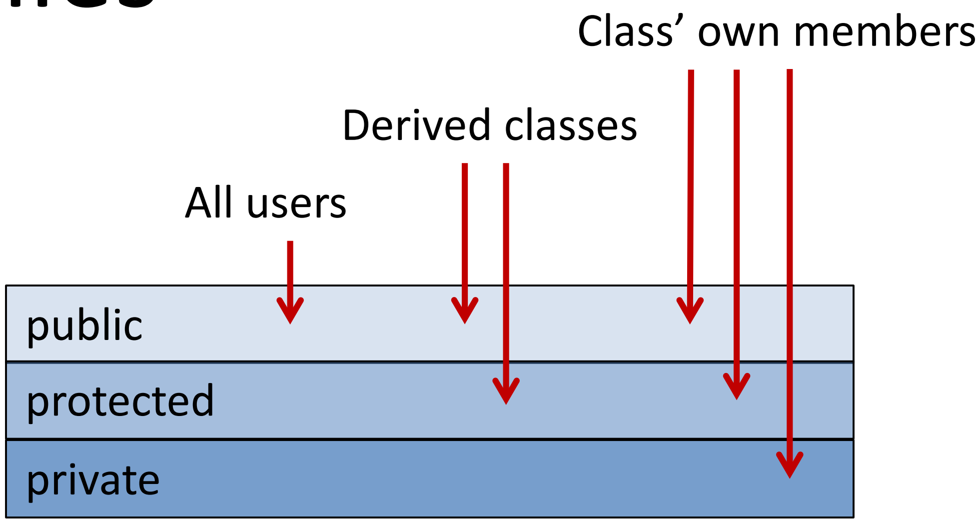
RAII and Move Semantics

- 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**
    - …

OOP

Class hierarchies

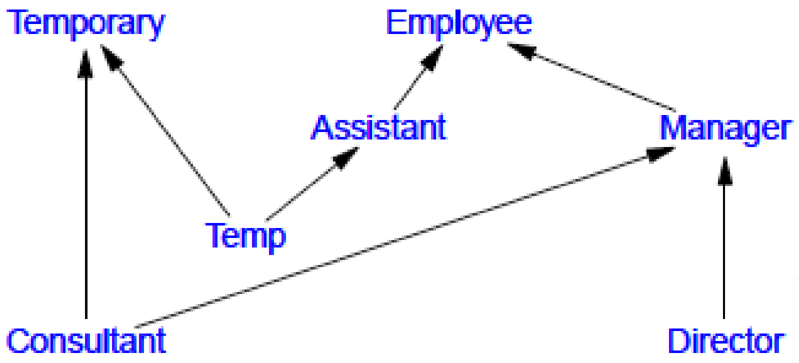
- Protection model



- No universal base class
  - an unnecessary implementation-oriented artifact
  - imposes avoidable space and time overheads.
  - encourages underspecified (overly general) interfaces
- Multiple inheritance
  - Separately consider interface and implementation
  - Abstract classes provide the most stable interfaces
- Minimal run-time type identification
  - **dynamic_cast<D*>(pb)**
  - **typeid(p)**

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”

GP

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());                          // sort the vector using <

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

     // …
}
```

- We parameterize over element type and container type

Algorithms

- Simple, efficient, and general implementation
  - For any forward iterator
  - 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;
}
```

Function Objects and Lambdas

- General function object
  - Can carry state
  - Easily inlined (i.e., close to optimally efficient)

```
struct Less_than {
	String s;
	Less_than(const string& ss) :s{ss} {}		// store the value to compare against
	bool operator()(const string& v) const { return v<s; } // the comparison
};
```

Lambda notation
  - We can let the compiler write the function object for us

```
auto p = std::find_if(vs.begin(),vs.end(),
			[](const string& v) { return v<"Griffin"; } );
```

Container 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>
```

```cpp
class CountEven
{
    int& count_;
public:
    CountEven(int& count)
    void operator()(int val)
    {
        if (!(val & 1))       // va
        {
            ++ count_;
        }
    }
};
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each(v.begin(), v.end(), CountEven(even_count));
std::cout << "The number of even is " << even_count << std::endl;
```

```cpp
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each( v.begin(), v.end(), [&even_count](int val)
        {
            if (!(val & 1))  // val % 2 == 0
            {
                ++ even_count;
            }
        });
std::cout << "The number of even is " << even_count << std::endl;
```

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*

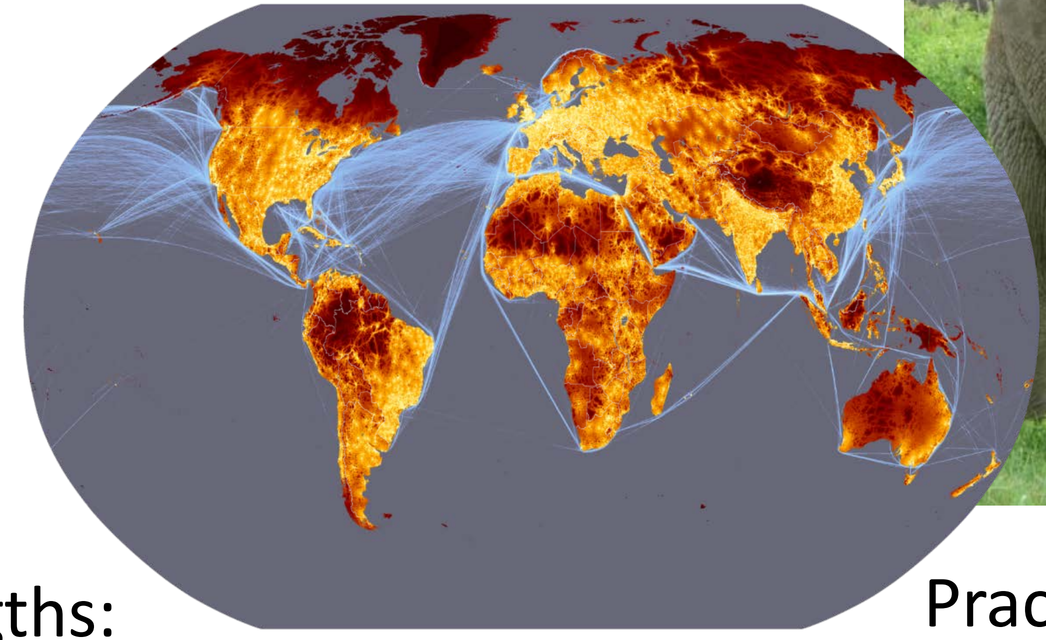
```
double sqrt(double d);          // C++84: accept any d that is a double
double d = 7;
double d2 = sqrt(d);    // fine: d is a double
double d3 = sqrt(&d); // error: &d is not a double
```

- 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

问题求解与实践
——C++回顾

THANKS

FOR YOUR WATCHING