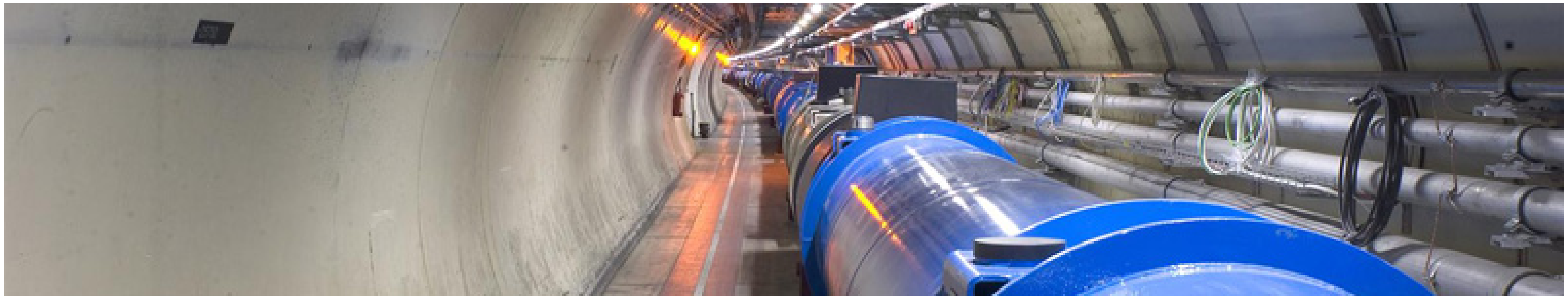
Foundations of C++

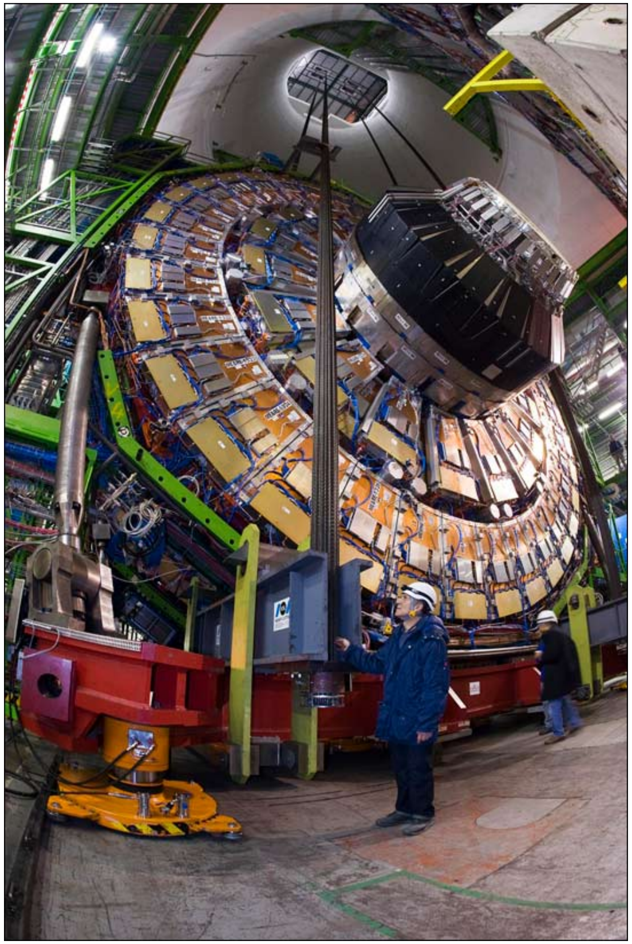
Bjarne Stroustrup

Texas A&M University



Overview

- Memory and objects
  - Construction and destruction
- Containers
  - Copy and Move
- Resources and RAII
- Class hierarchies
- Algorithms
- Compile-time computation
  - Type functions
- Concurrency
- Not: Casts, macros, pointer arithmetic, how to write bad code



Foundations of C++

- This is a talk about programming techniques
  - And language support for such techniques
- I present an industrial programmer's view of C++
  - No Γρεεκ Λεττερς
  - No Grammar
- I present fundamental examples
  - Not language details
  - Not legacy techniques/code
    - There are hundreds of millions of lines of code relying on the techniques I mention
  - An idealistic view: progress is necessary and possible
- I don't focus on the new C++ features
  - C++ is not (just)a series of historical strata

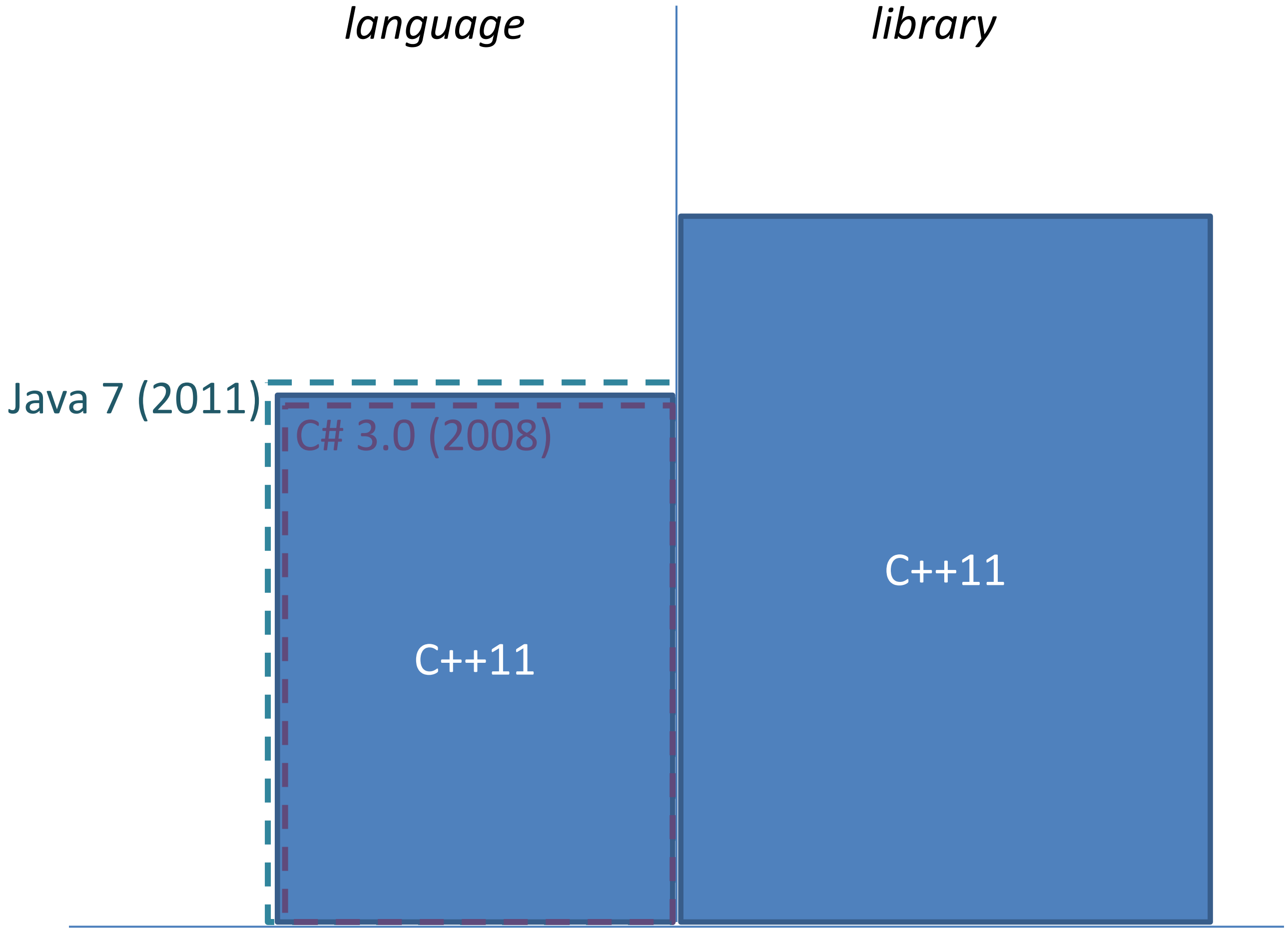
Complexity

- C++ is huge
  - But so are other language used for production code
- Complexity goes somewhere
  - Language, library, application, infrastructure
- The very notion of programming is changing/fracturing
  - Library users
  - Scripters
  - System builders
  - Infrastructure builders
  - Embedded systems builders
  - …

Portable C++

proxies for size comparisons:
spec #words
library #types (non-'plumbing')

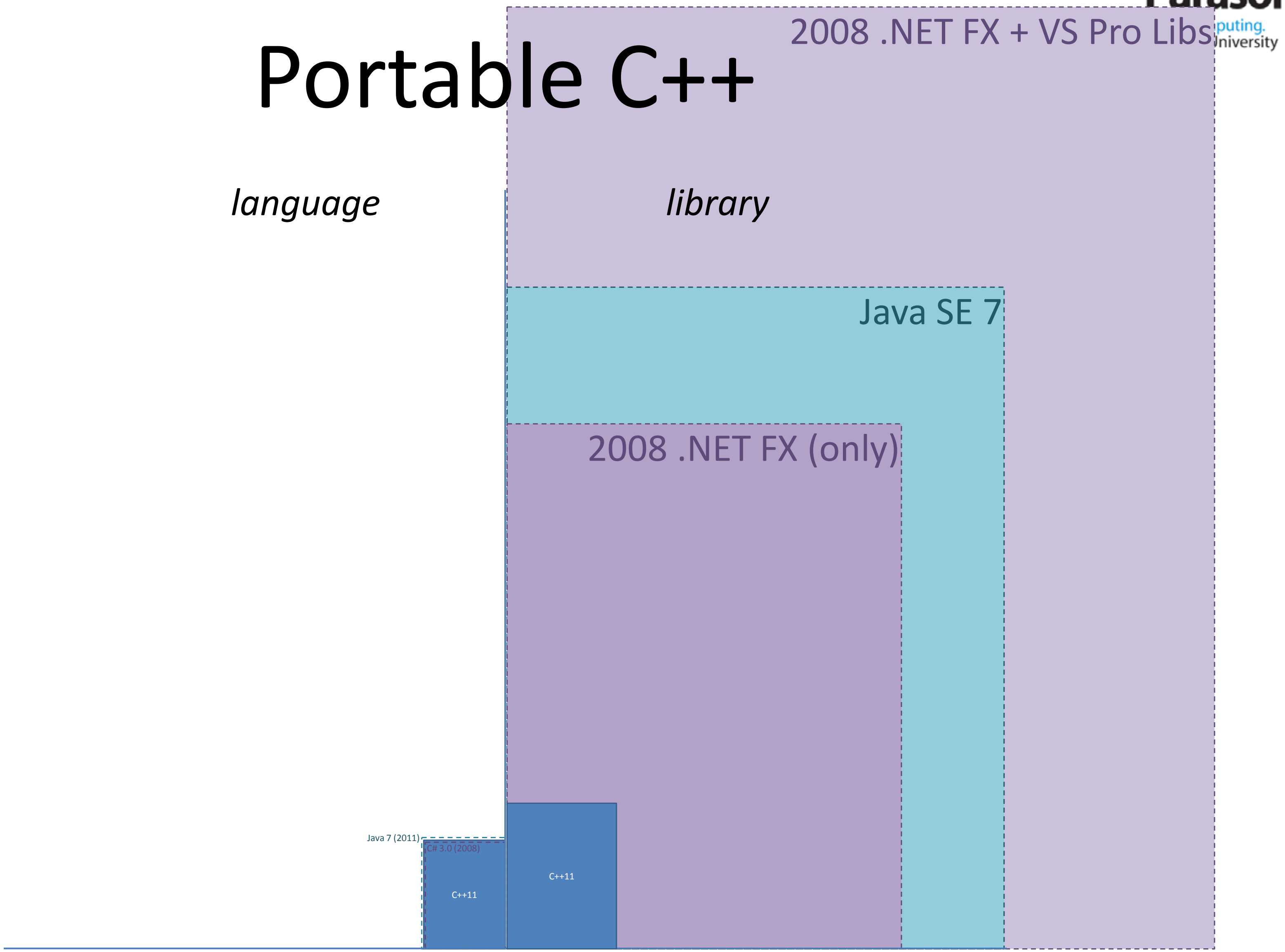
Parasol
Smarter computing.
Texas A&M University



Herb Sutter

Scale: C++ language: 400 pages; C++ standard library: 750 pages

Portable C++



Herb Sutter

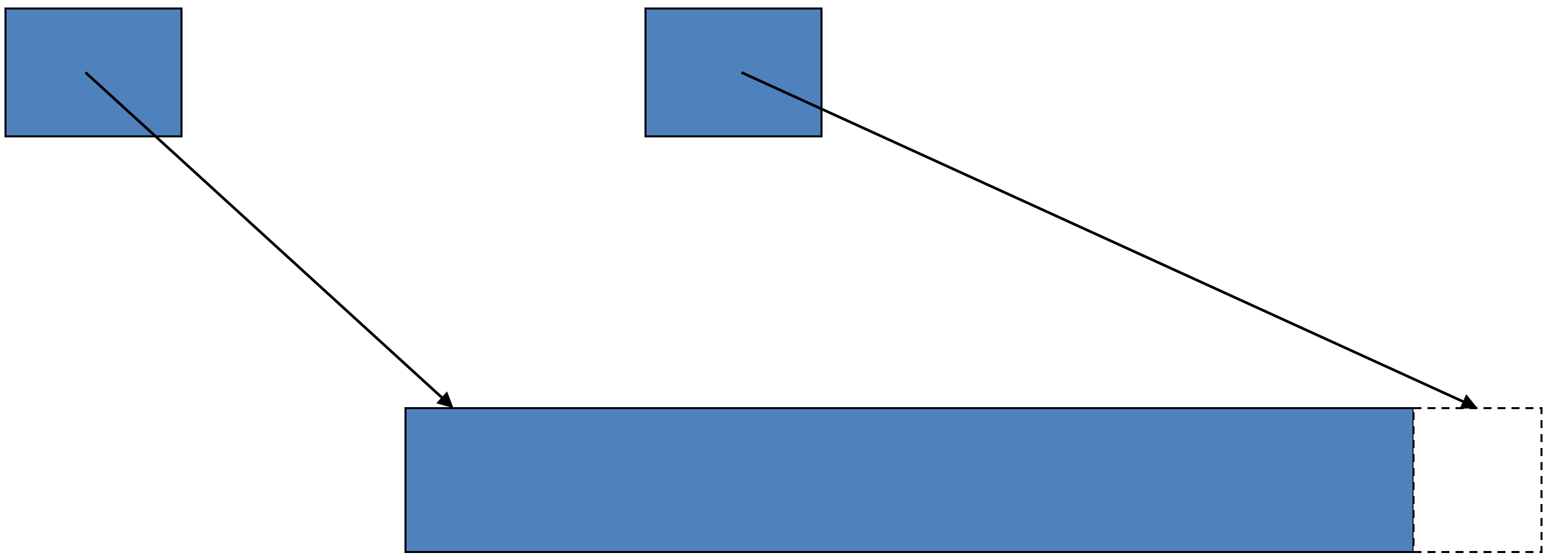
C++

- ISO/IEC 14882-2011 aka C++11, formerly “C++0x”
- Basics:
  - A simple and direct mapping to hardware
  - Zero-overhead abstraction mechanisms
- Supports
  - Classical systems programming
  - Infrastructure applications
    - resource-constrained and mission-critical
  - Light-weight abstraction
  - A type-rich style of programming
    - C++ supports type-safe programming with a non-trivial set of types.
  - And more

Other concerns

- Most of what is important to software development organizations don’t show in code fragments
  - Tool chains
  - Stability ***and*** progress
  - Interoperability with other languages
  - Availability of libraries
  - Availability of trained developers

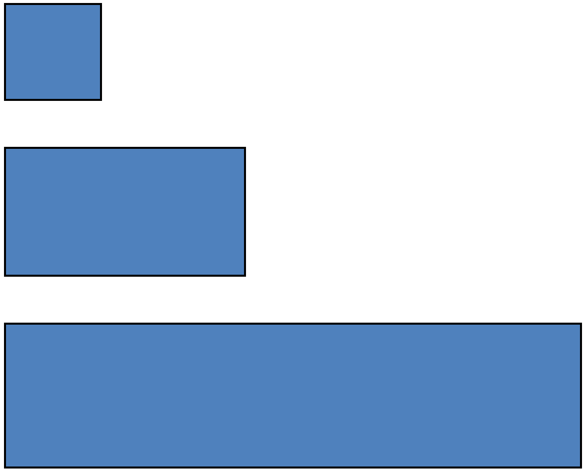
Memory model



Memory is sequences of objects addressed by pointers

Memory model (built-in type)

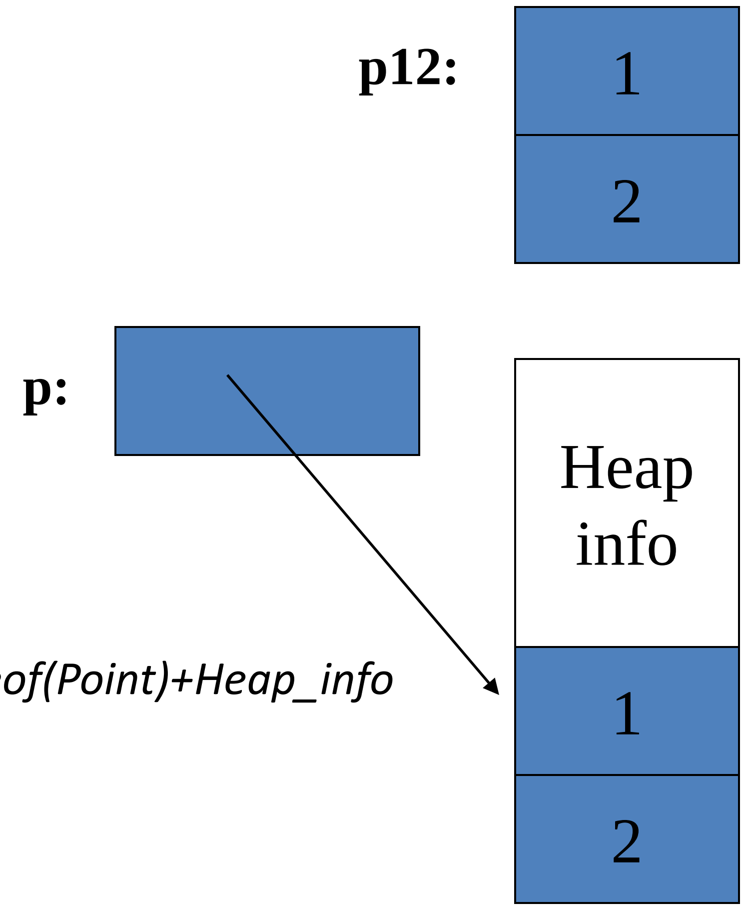
- char
- short
- int
- long
- (long long)
- float
- double
- long double
- T* (pointer)
- T& (implemented as pointer)



Memory model ("ordinary" class)

```
class Point {
    int x, y;
    // …
};

// sizeof(Point)==2*sizeof(int)

Point p12 {1,2};

Point* p = new Point{1,2};

// memory used for "p":sizeof(Point*)+sizeof(Point)+Heap_info
```

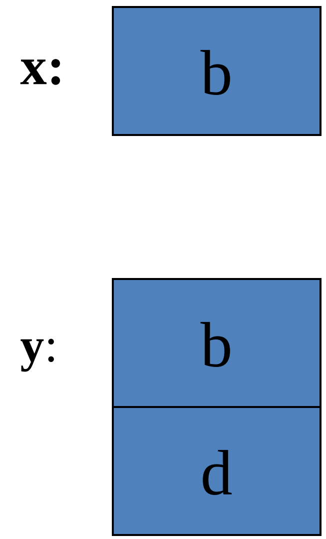
- Simple Composition



Memory model – class hierarchy

```
class B {
    int b;
};

class D : public B {
    int d;
};
```

```
B x;
D y;
```

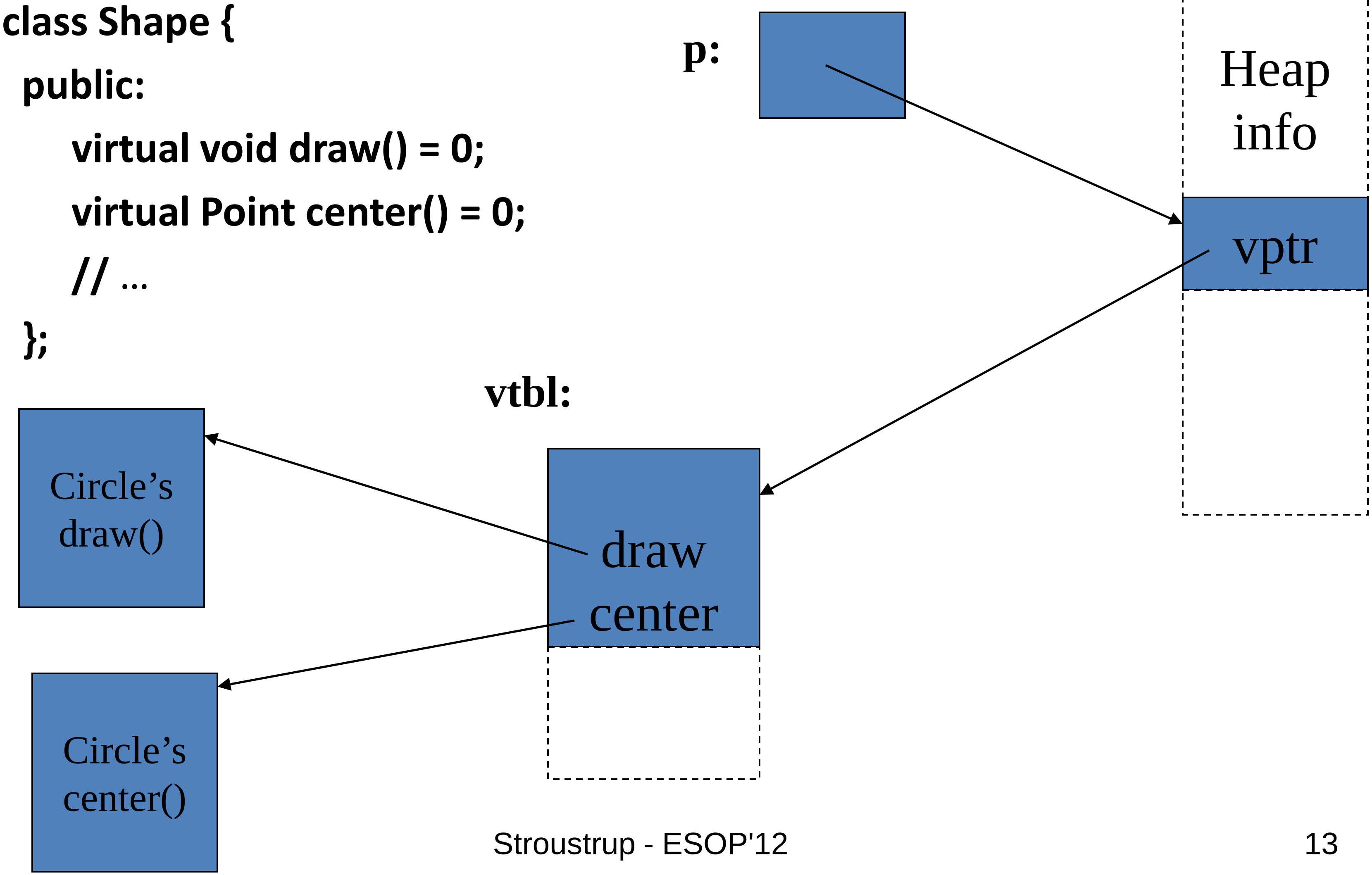


- Simple composition

Memory model (polymorphic type)

```
class Shape {
 public:
     virtual void draw() = 0;
     virtual Point center() = 0;
     // …
};
```

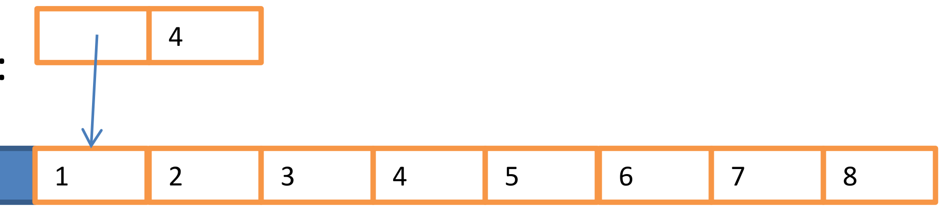
`Shape* p = new Circle{{x,y},r};`



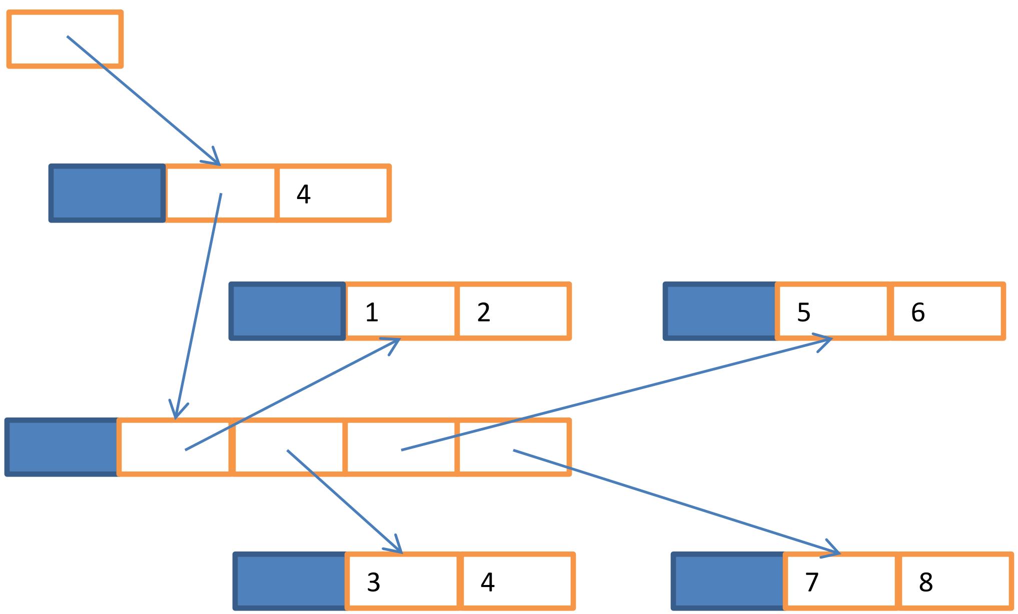
Use compact layout

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

C++:



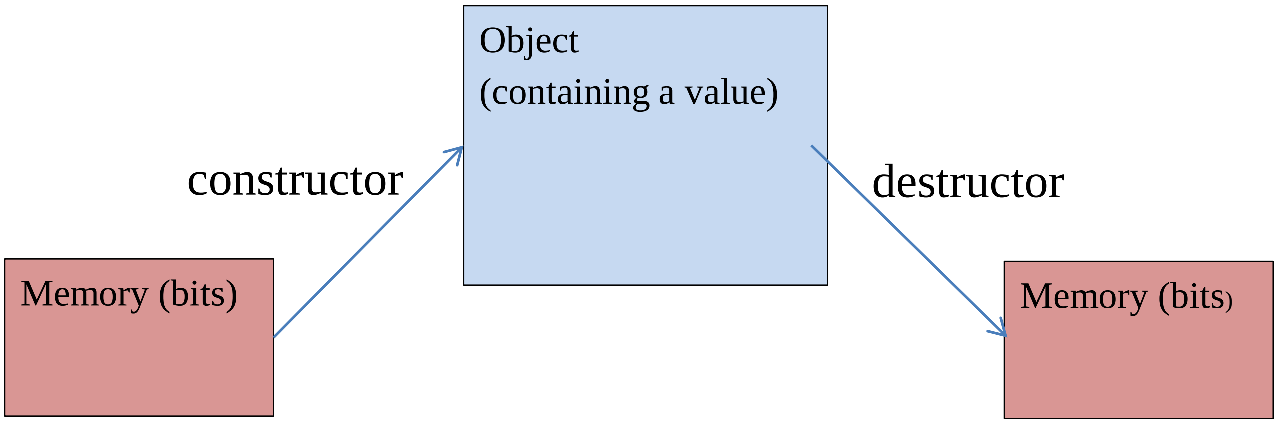
"True OO" style:



A loss

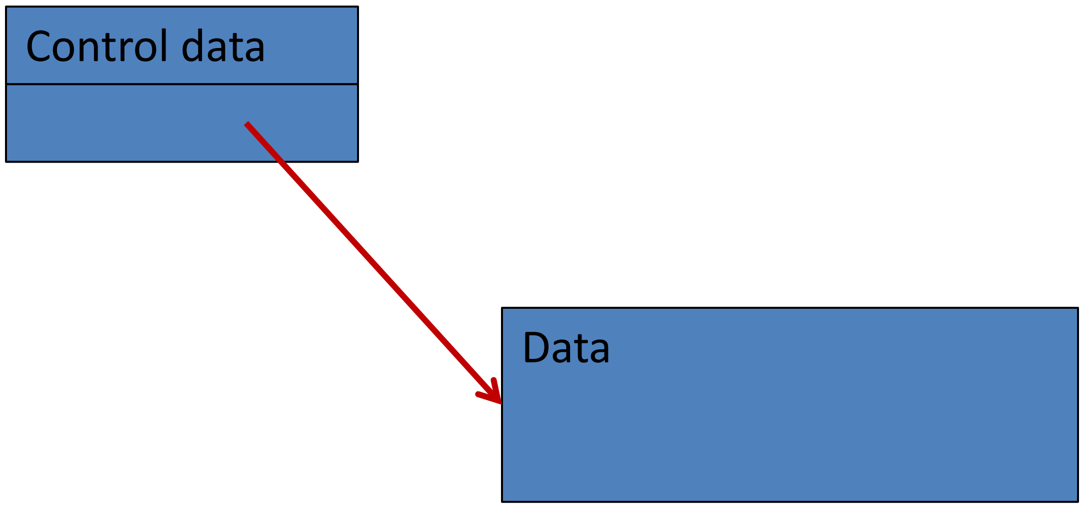
- Many students and developers don’t understand the language-to-machine mapping
  - To them, it’s “magic”
  - In the context of infrastructure projects, that’s a significant problem

Constructors and destructors



- Constructor: make an object from memory
  - An object holds a value
  - Has a type
  - Has an interface
  - Has meaning
- Destructor: make an object (back) into memory
  - Memory is just interpreted bits

A resource handle



- Examples
  - Containers: vector, list, map, …
  - Smart pointers: unique_ptr, shared_ptr, delayed_value, remote_object, …
  - Locks, thread handles, sockets, iostreams
  - File handle
  - …

Vector: the archetypical resource handle

- Slight simplification of **std::vector**

```
template<typename T>                // T is the element type
class Vector {
public:
    Vector();                       // default constructor: make empty vector
    Vector(int  n);                 // constructor: initialize to n  elements
    Vector(initializer_list<T>) ;   // constructor: initialize with element list
    ~Vector();                      // destructor: deallocate elements
    int size() ;                    // number of elements
    T& operator[](int i);           // access the ith element
    void push_back(const T& x);     // add x as a new element at the end
    T* begin();                     // fist element
    T* end();                       // one-beyond-last element
private:
    int sz;                         // number of elements
    T* elem;                        // pointer to sz elements of type T
};
```

Vector use

- Simple use of vectors

```
void f(Vector<string>& vs)
{
    Vector<int> sizes;                  // empty vector: sizes.size()==0
    for (auto  x : vs)                  // loop through all elements of vs
        sizes.push_back(x.size());      // add element to vector (grow)

    if (0<vs.size())                    // check size
        vs[0] = "Whatever!";            // subscripting

    // …
}
```

Vector use

- Simple use of vectors

```
int main()
{
    f({"Wheeler", "Wilkes", "Radcliffe", "Appleton", "Rutherford"});
    Vector<string> places(10);   // 10 empty strings
    places[2] = "Cambridge";
    // …
    f(places);
}
```

Constructor

- N default elements

```
template<typename T>
Vector<T>::Vector(int n) // make a vector with n elements of default value
      :sz{n},
       elem{allocate<T>(sz)}         // allocate space for sz elements of type T
{
      if (sz<0) throw std::runtime_error{"negative Vector size"};
      std::uninitialized_fill(elem,elem+sz,T{});         // initialize to default
}
```

- The **allocate<T>()** function is a simplification of the standard-library **allocator** mechanism to make the examples fit on slides

Destructor

- Essential:
  - release resource (in this case free memory)

```
template<typename T>
Vector<T>::~Vector()            // destructor releases resources acquired
{
    destroy<T>(elem,n);   // invoke member destructors, then deallocate elem[]
}
```

- Note: the elements typically have destructors
  - E.g. **Vector<Vector<string>>**
  - Explicit invocation of destructors is extremely rare
    - basically only in sophisticated container implementations

Initializer-list constructor

- A class can have many constructors

```
template<typename T>
Vector<T>::Vector(std::initializer_list<T> lst)      // elements from the list
    :sz{lst.size()},
     elem{allocate<T>(sz)}
{
    std::uninitialized_copy(lst.begin(), lst.end(), elem);
}
```

A matched blend of techniques

- The standard library containers (e.g., **vector**, **map**, **set**, and **list)** :
  - classes for separating interfaces from implementations
  - constructors for establishing invariants, including acquiring resources
  - destructors for releasing resources
  - templates for parameterizing types and algorithms with types
  - mapping of source language features to user-defined code
    - e.g. **[]** for subscripting, the **for**-loop, **new**/**delete** for construction/destruction on the free store, and the **{}** lists.
  - use of half-open sequences, e.g. [**begin()**:**end()**), to define for-loops and general algorithms.
  - Use of standard-library facilities to simplify specification and implementation
- This abstraction from "memory" to "containers of objects" carries no overheads
  - beyond the code necessarily executed for memory management, initialization, and error checking.

Absolutely minimal overheads

- There is no data stored in a **Vector** object
  - beyond the two named members (three in **std::vector**)
- The element type need not be part of a hierarchy
  - the only requirements on a template argument are imposed by its use
  - “duck typing.”
- **Vector** operations are not dynamically resolved
  - Not **virtual**
  - Simple operations, such as **size()** and **[]**, are typically inlined
- A Vector is allocated where needed
  - On stack, in objects
- A vector is accessed directly
  - Not through a handle (it *is* a handle)

Copy and move

- Copy constructor

```
Vector capitals {"Helsinki", "København", "Riga", "Tallinn"};
Vector c2 = capitals;    // error: no copy defined for Vector
 // By default, you can copy only objects with "simple representations.

// So we define a suitable copy:

template<typename T>
Vector<T>::Vector(const Vector& v)      // copy constructor
    : sz{v.sz}, elem{allocate<T>(v.sz)}
{
    std::uninitialized_copy(v.begin(),v.end(), elem);
}
```

Copy and move

- Copy assignment

```
template<typename T>
Vector<T>& Vector<T>::operator=(Vector<T>& v)   // copy assignment
{
    Vector<T> tmp {v};          // copy v
    destroy<T>(elem,sz);        // delete old elements
    elem = tmp.elem;            // "steal" tmp's representation
    sz = tmp.sz;
    tmp.elem = nullptr;         // tmp will be empty when destroyed
    tmp.sz = 0;
    return *this;
}
```

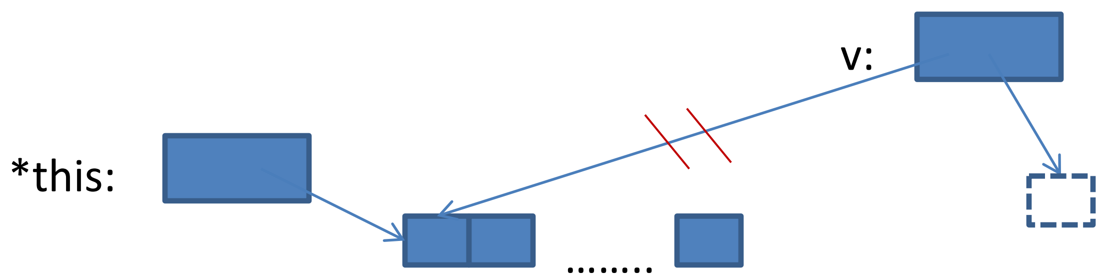
Copy and move

```
Vector<int*> find_all(Vector<int>& v, int val) // find all occurrences of val in v
{
    Vector<int*> res;
    for (int& x : v)
        if (x==val)
            res.push_back(&x);        // add the address of the element to res
    return res;
}

void  test()
{
    Vector<int> lst {1,2,3,1,2,3,4,1,2,3,4,5};
    for (int* p :  find_all(lst,3))
        cout << "address: " << p << ", value: " << *p << "\n";
    // …
}
```

Copy and move

- Move constructor

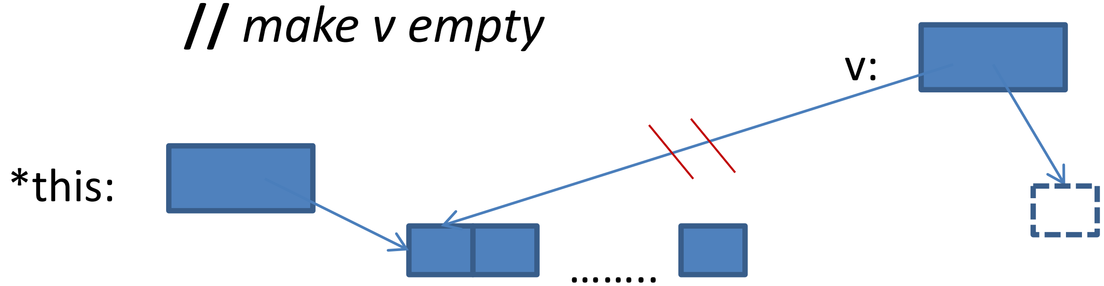
```
template<typename T>
Vector<T>::Vector(const Vector&& v)            // move constructor
    : sz{v.sz}, elem{v.elem}       // grab v's elements
{
    v.elem = nullptr;              // make v empty
    v.sz = 0;
}
```



Copy and move

- Move assignment

```
template<typename T>
Vector<T>& Vector<T>::operator=(Vector<T>&& v)    // move assignment
{
    destroy<T>(elem,sz);          // delete old elements
    elem = v.elem;                // grab v's elements
    sz = v.sz;
    v.elem = nullptr;             // make v empty
    v.sz = 0;
    return *this;
}
```



Vector

Parasol
Smarter computing.
Texas A&M University

- Copy and move declarations added to Vector

```
template<typename T>              // T is the element type
class Vector {
public:
    // …
    Vector(const Vector&);                   // copy constructor
    Vector(Vector&&);                        // move constructor
    Vector& operator=(const Vector&);        // copy assignment
    Vector& operator=(Vector&&);             // move assignment
    // …
};
```

Vector

- And of course, a user doesn't have to implement **Vector**

  #include<vector>

  will get an even more flexible and efficient version

- BUT: all the techniques and language facilities are available for all to use for their own abstractions

Resources and errors

- Exceptions
- Resources
- RAII

Error Handling: Exceptions

```
void do_task(int i)
{
	if (i==0) throw std::runtime_error{"do_task() of zero"};
	if (i<0) throw Bad_arg{i};
	// do the task and return normally
}

void task_master(int i)
{
	try {
			do_task(i);
			// ...
	}
	catch (Bad_arg a) {
			cout << "do_task() of negative" << a.val << "\n";
	}
}
```

Real-world constraints

- Sometimes, you can’t use exceptions
  - Hard real time
  - Messy old code
- Implications
  - Duplication of styles and components
  - Complexity
  - Confusion
- This happens again and again
  - for different programming techniques
  - For different language features
  - A major source of complexity

Resources and Errors

- Many (most?) resources are not just memory
  - A non-memory resource requires a release operation
  - Not just freeing of memory

```
//     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<mutex> lck {m};    // grab (acquire) the mutex
    sh+=1;     // manipulate shared data
}                                        // implicitly release the mutex
```

Simplify control structure

- Prefer algorithms to unstructured code

Algorithms

- Messy code it a major source of errors and inefficiencies
- We must use more “standard” 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(),"Aarhus");   // find “Aarhus” in lst:

    if (p!=lst.end()) {     // found: *p==“Aarhus”
      // …
    else                    // not found *p!=“Aarhus”
      // …
}
```

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

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

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

    if (p!=vs.end()) {        // found: *p<"Griffin"
      // …
    }
    else {                    // not found *p>="Griffin"
      // …
    }
    // …
}
```

Algorithms

- The implementation is still trivial

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

Algorithms: function objects

- General function object
  - Can carry state
  - Easily inlined

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

Lambda notation

- We can let the language write the function object

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

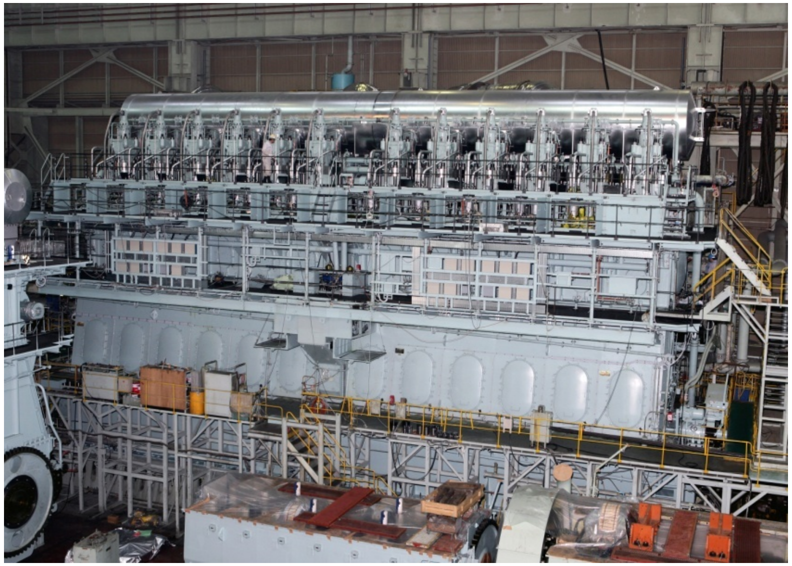
Container algorithms

- 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>
        void sort(C& c) { std::sort(c.begin(),c.end(); }
        // …
}
```

Compile-time Computation

- Type-rich computation at compile time.
  - *Efficiency*: To pre-calculate a value (often a size).
    - Simple cases (only) done by an optimizer.
  - *Type-safety*: To compute a type at compile time.
  - *Simplify concurrency*: you can't have a race condition on a constant.
- No just error-prone macro hacking



Type-rich compile-time computation

- Just like other code
  - Except it is executed by the compiler
  - To do anything interesting we need a type system

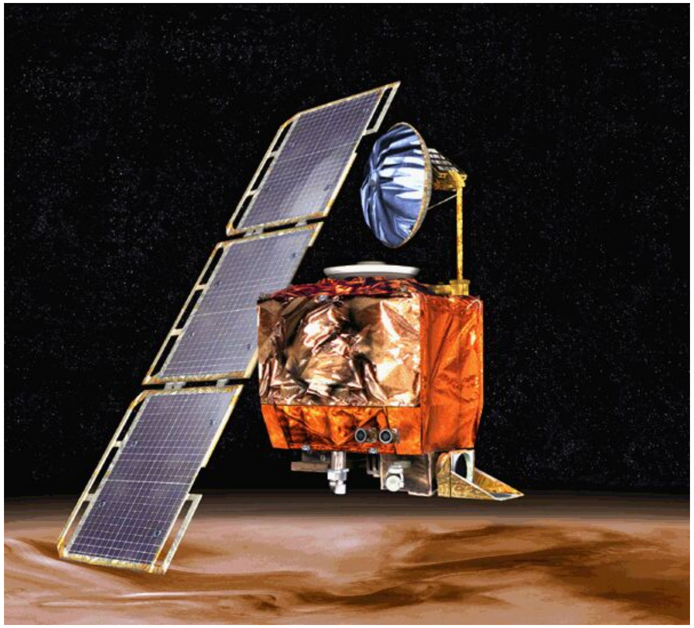
```
struct City { double  x, y };

constexpr double csqrt(double) { /* calculate square root */ }
constexpr double square(double d) { return d*d; }
constexpr double dist(City c1, City c2)
 {
      return csqrt(square(abs(c1.x-c2.x))+square(abs(c1.y-c2.y)));
}

constexpr double d = dist(NewYork,Boston);      // a simple use
```

Unit checking: 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



Type Functions

- The standard library depends on
  - Compile-time selection of types
  - Compile-time calculation of values
  - Compile-time selection of algorithms
- So does much other code

- Sometimes, referred to as “Template Meta-programming”
  - Keep it simple
- The key notion is a “Type function”
  - takes at least one type argument or returns at least one type
    - **sizeof(T)**
    - **SameType<T,U>**
    - **Value_type<Iter>**

Type Functions

- Functions that answer questions about types or return types

```
template<typename Cont>
void sort(Cont& c)                    // sort container of type Cont
{
    if (Has_random_access<Cont>())      // ask about Cont's properties
      sort(c.begin(),c.end());
    else  {
      vector<Value_type<Cont>> v {c.size()};  // get an associated type from Cont
      copy(c.begin(),c.end(),v);              // copy, sort, and copy back
      sort(v);
      copy(v.begin(),v.end(),c);
    }
}
```

Type Functions

- I really wanted to overload on “concepts” (compiler-supported predicates on sets of types and values):

  template<Random_access_container> …

  template<Bidirectional_access_container> …

  template<Forward_access_container> …

- But for now I will show how to use type functions
  - As widely used in current C++
  - How to non-intrusively add properties to types
  - Using “traits”

Type Functions

- Traits classes (an important technique/workaround)
  - A general mechanism for adding non-intrusively properties to types

```
template<typename Cont>
struct container_traits {          // general/default traits
    using value_type = typename Cont::value_type;        // member type
    using access_category = typename Cont::access_tag;   // member type
    // …
};
```

- Why not always use **Cont::value_type**?
  - Because **T[N]::value_type** is invalid syntax

Type Functions

- We use a template alias to return a type:

```
template<typename Cont>
using Value_type = typename container_traits<Cont>::value_type;

template<typename Cont>
using Access_category =
                    typename container_traits<Cont>::access_category;
```

- So, **Value_type<Vector<double>>** is **double**

Parasol
Smarter computing.
Texas A&M University

Type Functions

- We use **constexpr** function templates to return values

```
template <typename T, typename U>
constexpr bool Is_same()
{
    return is_same<T, U>::value;    // is_same is a std type trait (an intrinsic)
}

template<typename Cont>
constexpr bool Has_random_access<Cont>()
{
    return Is_same<Access_category<Cont>, random_access_tag>();
}
```

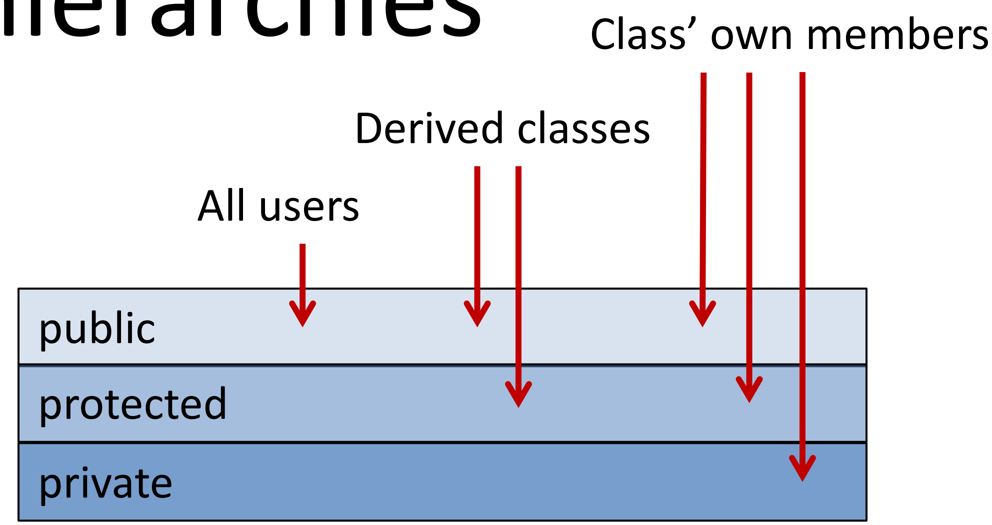
- **Has_random_access<Vector<double>>** is **true**

Type Functions

- Functions that answer questions about types or return types

```
template<typename Cont>
void sort(Cont& c)                    // sort container of type Cont
{
    if (Has_random_access<Cont>())      // ask about Cont's properties
        sort(c.begin(),c.end());
    else  {
        vector<Value_type<Cont>> v {c.size()};  // get an associated type from Cont
        copy(c.begin(),c.end(),v);              // copy, sort, and copy back
        sort(v);
        copy(v.begin(),v.end(),c);
    }
}
```

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
  - Interface and implementation
  - Abstract classes provide the most stable interfaces
- Minimal run-time type identification
  - **dynamic_cast<D*>(pb)**
  - **typeid(p)**

“Paradigms”

- Much of the distinction between object-oriented programming and generic 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

```
template<typename Cont>
void draw_all(Cont& c)
{
    for_each(c.begin(),c.end(), [](Shape* p) { p->draw(); }
}
```

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

No garbage collection needed

- 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
     - Note: non-memory 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

Type safety

- C++ is not guaranteed to be statically type safe
  - "C is a strongly typed; weakly checked, language" – DMR
- A language designed for general and performance critical systems programming with the ability to manipulate hardware cannot be.
- Problems
  - untagged unions
  - explicit type conversions (casts)
  - arrays without (guaranteed) range checks
  - ability to deallocate a free store (heap) object while holding on to a pointer allowing for post-allocation access.
  - ability to deallocate an object not allocated on the free store

Challenges

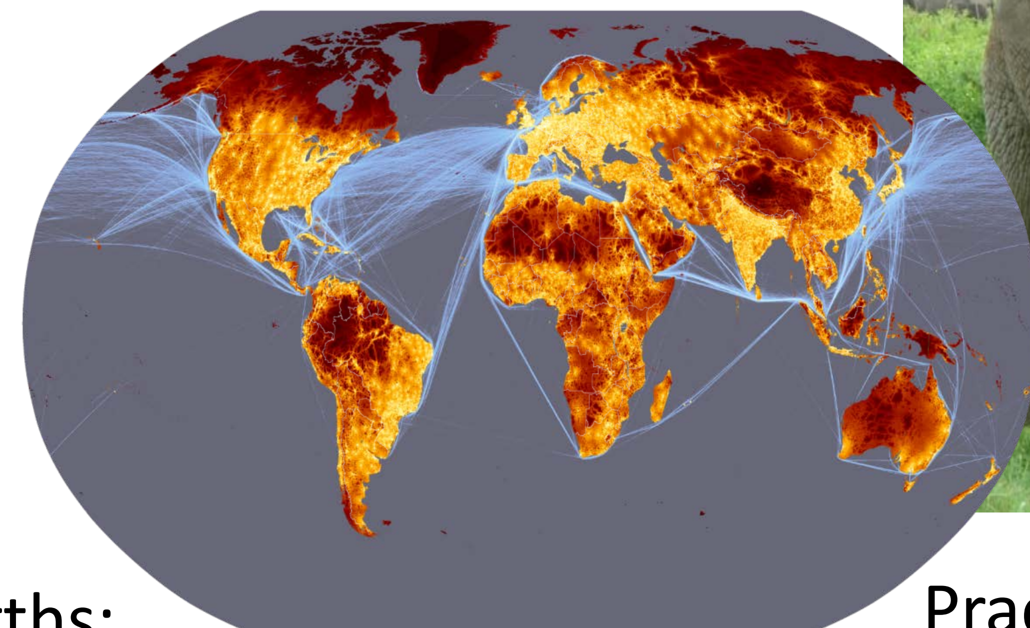
- Obviously, C++ is not perfect
  - How can we make programmers prefer modern C++ styles over low-level (C-style) code, which is far more error-prone and harder to maintain, yet no more efficient?
  - How can we make C++ a better language given the Draconian constraints of C and C++ compatibility?
  - How can we improve and complete the techniques and models (incompletely and imperfectly) embodied in C++?
- In the context of C++, solutions that eliminate major C++ strengths are not acceptable
  - Compatibility (link, source code)
  - Performance
  - Portability
  - Range of application areas

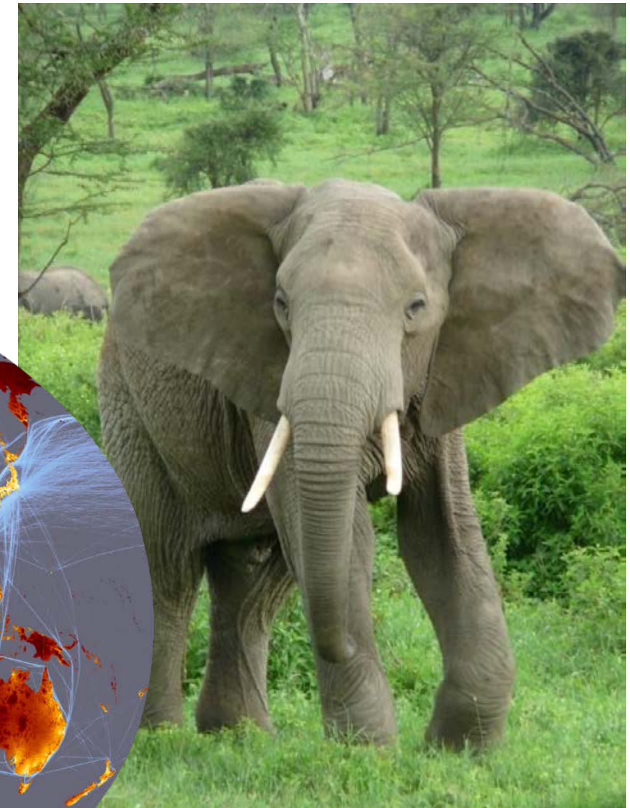
Challenges

- Close more type loopholes
  - in particular, find a way to prevent misuses of **delete** without spoiling RAII
- Simplify concurrent programming
  - in particular, provide some higher-level concurrency models as libraries
- Simplify generic programming
  - in particular, introduce simple and effective concepts
- Simplify programming using class hierarchies
  - in particular, eliminate use of the visitor pattern
- Better support for combinations of object-oriented and generic programming
- Make exceptions usable for hard-real-time projects
  - that will most likely be a tool rather than a language change
- Find a good way of using multiple address spaces
  - as needed for distributed computing
  - would probably involve defining a more general module mechanism that would also address dynamic linking, and more.
- Provide many more domain-specific libraries
- Develop a more precise and formal specification of C++

Questions?

C++: A light-weight abstraction programming language





Key strengths:

- software infrastructure
- resource-constrained applications

Practice type-rich programming