PY410 / 505
Computational Physics 1

Salvatore Rappoccio

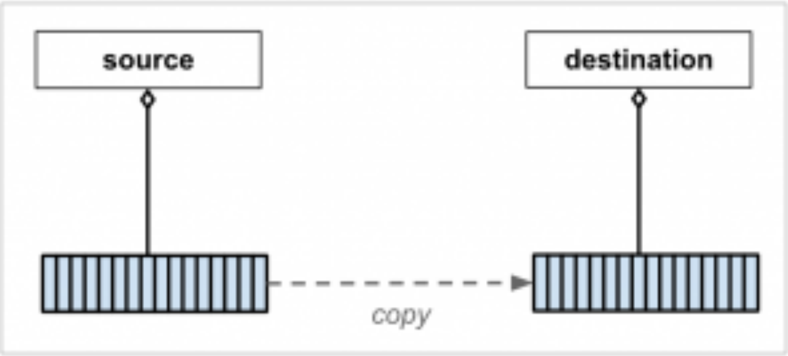
Advanced C++

- C++ underwent major revision in mid-00’s
- C++0x (x was supposed to be 4, but..) turned into C++11
- There is now C++17, other updates

- Major changes in C++11

https://en.wikipedia.org/wiki/C%2B%2B11

Copy : member data is cloned



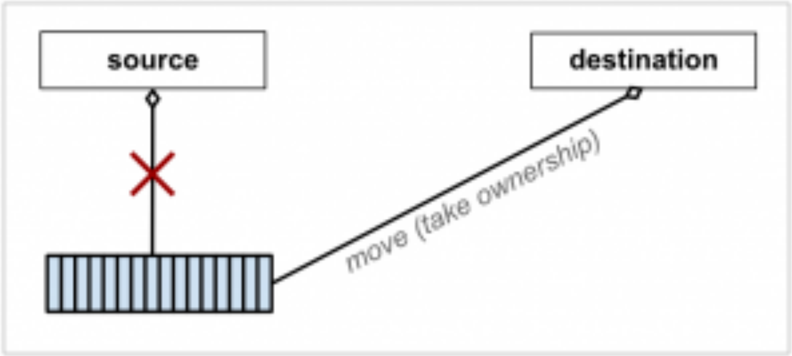
```
template <class T> swap(T& a, T& b)
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

Expensive!

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
http://avidinsight.uk/2013/05/understanding-cpp11-move-semantics/

Advanced C++

Move : member data is reassigned

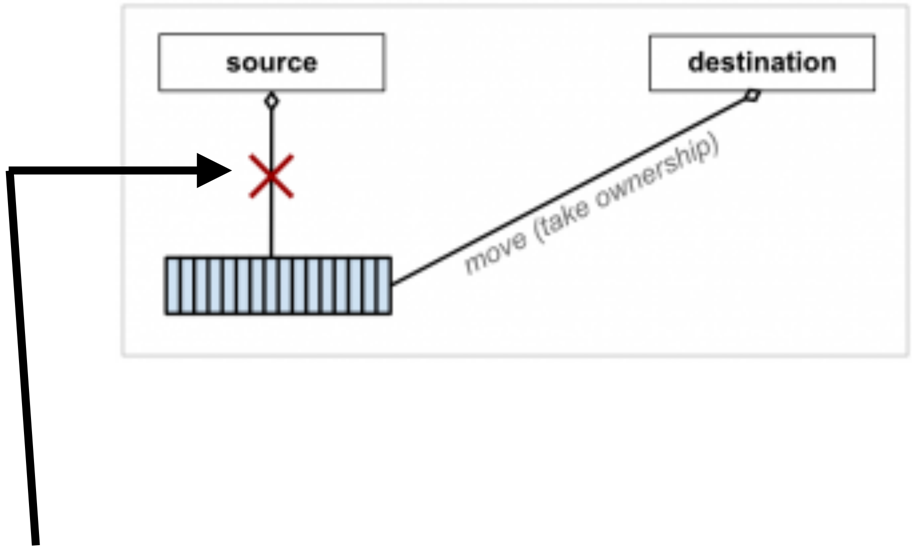


Cheap!

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
http://avidinsight.uk/2013/05/understanding-cpp11-move-semantics/

Advanced C++

Move : member data is reassigned



Setting this to “null” is not allowed in C++03!

Cheap!

But not supported in old C++

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
http://avidinsight.uk/2013/05/understanding-cpp11-move-semantics/

Advanced C++

lvalue reference

```
A a;
A& a_ref1 = a;   // an lvalue reference
```

rvalue reference

```
A a;
A&& a_ref2 = a;   // an rvalue reference
```

rvalue reference can bind to a TEMPORARY variable!

```
A&  a_ref3 = A();   // Error!
A&& a_ref4 = A();   // Ok
```

After function A()’s temporary return value goes out of scope, does not delete the memory used for it

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html

Advanced C++

- Move semantics

```
template <class T> swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Moves a’s member data to tmp, state of a is undefined
Moves b’s member data to a, state of b is undefined
Moves tmp’s member data to b, state of tmp is undefined

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html

Advanced C++

- How does this help?

Old bad way

```
A modify( A & a){
   return A(a);
}
A a;
A retval = modify( a );
```

Easy to write.
Lots of copies.
Really dumb.

Old annoying way

```
void modify( A & a){
   ...
}
A a;
A retval;
modify( retval );
```

Performant.
Annoying to write.

New way, “explicitly”:

```
A modify( A & a){
   return A(a);
}
A a;
A && retval = modify( retval );
```

Performant.
Confusing.

New way, “implicitly”:

“A” must have a move constructor!

```
A modify( A & a){
   return A(a);
}
A a;
A retval = modify( retval );
```

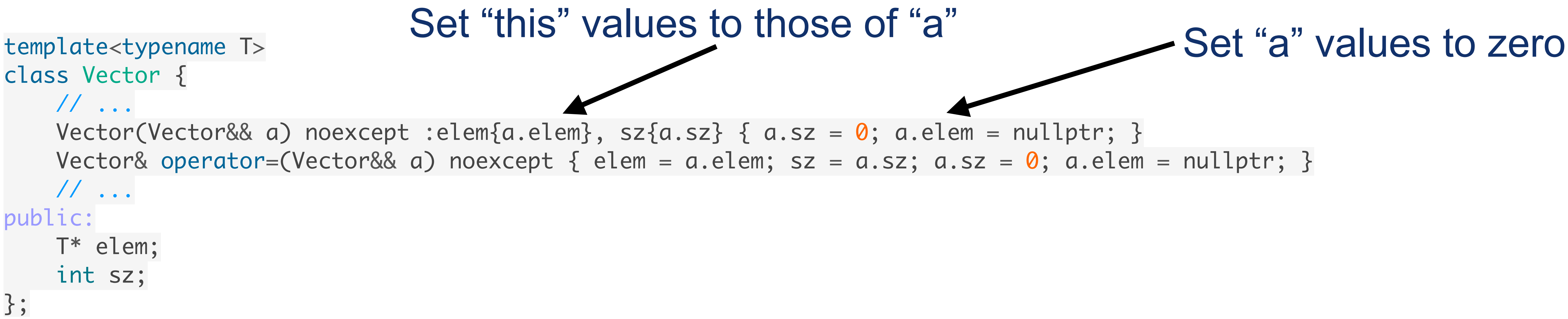
Easy to write.
Performant.

wins!

Advanced C++

- The “new way” with C++11 looks just like the “old way” how you wanted all along, but requires a “move constructor” to be guaranteed to be implemented correctly

- Move constructor example (std::vector):



Set “this” values to those of “a”

Set “a” values to zero

```
template<typename T>
class Vector {
    // ...
    Vector(Vector&& a) noexcept :elem{a.elem}, sz{a.sz} { a.sz = 0; a.elem = nullptr; }
    Vector& operator=(Vector&& a) noexcept { elem = a.elem; sz = a.sz; a.sz = 0; a.elem = nullptr; }
    // ...
public:
    T* elem;
    int sz;
};
```

(“noexcept” means it cannot throw exception… it’s complicated)

http://www.modernescpp.com/index.php/c-core-guidelines-copy-and-move-rules

Advanced C++

- So now, to make your code performant, implement the “Rule of 5”:
  - Copy constructor
  - Move constructor
  - Copy operator=
  - Move operator=
  - Destructor

- See “AdvCpp”!

Advanced C++

- “Old school” C++ (03 and earlier) : Initializing data was annoying

Old way

```
int aa[] = {1,2,3,4};
std::vector<int> a(aa);
```

New way

```
std::vector<int> a = {1,2,3,4};
```

Better way to initialize lists in new standard

https://en.wikipedia.org/wiki/C%2B%2B11

Advanced C++

- Type inference
  - Previously: had to explicitly state type
  - Now : compiler can deduce the type

Old way

```
std::vector< std::map<int,float>::const_iterator >::const_iterator i = v.begin();
```

New way

```
auto i = v.begin();
```

Can also use “decltype” (declare type) to make other variables of that type!

```
decltype(i) j = i+2;
```

https://en.wikipedia.org/wiki/C%2B%2B11

Advanced C++

- Range-based for loop
  - Looked this before, can be combined with “auto” to make things very compact

```
vector<int> aa= {1,2,3,4};
for ( auto x : aa )
   cout << x << endl;
```

Advanced C++

- Anonymous (lambda) functions

```
[](int x, int y) -> int { return x + y; }
```

  - Imagine you want to sort:

Previously:

```
// sort using a custom function object
    struct {
        bool operator()(int a, int b) const
        {
            return a < b;
        }
    } customLess;
    std::sort(s.begin(), s.end(), customLess);
```

C++11:

```
// sort using a lambda function
    std::sort(s.begin(), s.end(),
    [](int a, int b){return a < b;});
```

Lots less typing

https://en.cppreference.com/w/cpp/algorithm/sort

Advanced C++

- Can allocate lists of whatever types you want (tuples)

```
typedef std::tuple <int, double, long &, const char *> test_tuple;
long lengthy = 12;
test_tuple proof (18, 6.5, lengthy, "Ciao!");

lengthy = std::get<0>(proof);  // Assign to 'lengthy' the value 18.
std::get<3>(proof) = " Beautiful!";  // Modify the tuple’s fourth element.
```

Advanced C++

- Better pointers
  - std::shared_ptr is like a regular pointer, but calls “delete” when it goes out of scope automatically:

```
shared_ptr<A> factory_for_A(){
  return shared_ptr<A> ( new A() );
}
shared_ptr<A> a = factory_for_A();
```

  - Can also now hold vector<shared_ptr> (in previous C++, had auto_ptr, but this was not supported)

```
std::vector< std::shared_ptr<A> > v_stuff;
```

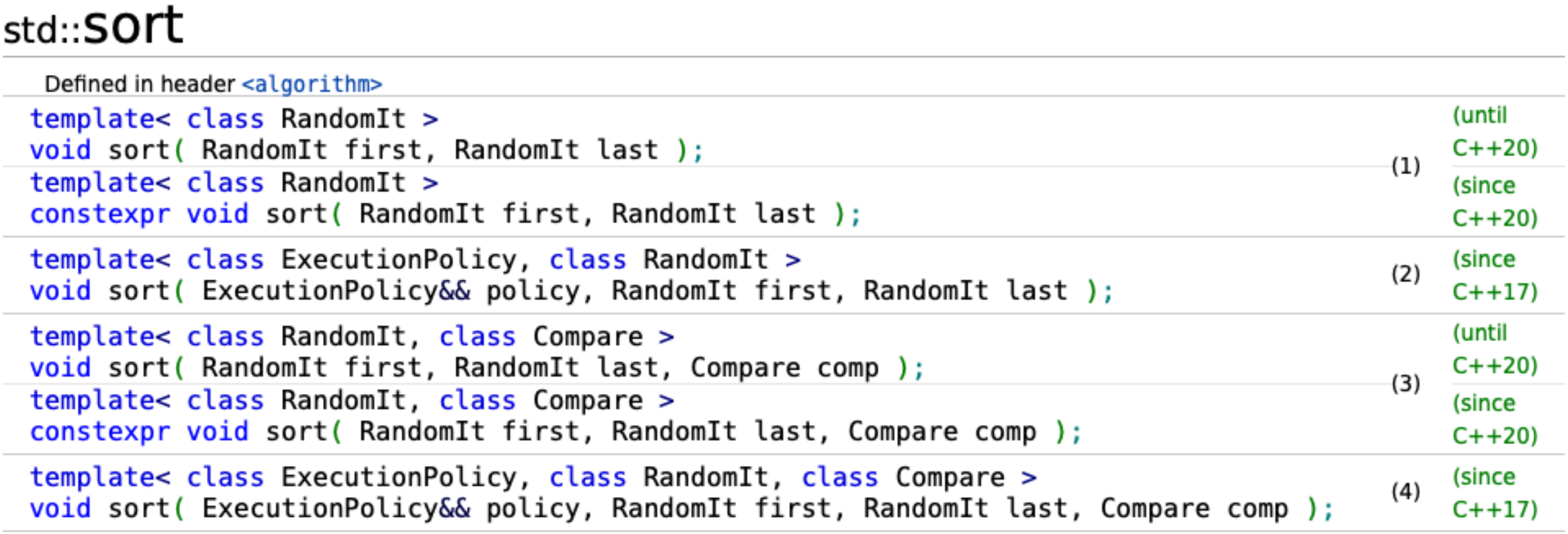
v_stuff can hold a list of A *,
or ANYTHING derived from A!

Advanced C++

- We’ve seen some examples of objects from the Standard Template Library (STL).
  - std::vector, std::map, std::string, etc
  - http://www.cplusplus.com/reference/stl/
- There are many algorithms that can operate on them!
  - std::sort, std::find, etc
  - https://en.cppreference.com/w/cpp/algorithm
- This brings the full power of C++ and templates to bear
  - Fast. Performant. Not terrible syntax.
- The STL documentation should become your absolute best friend when coding C++

Advanced C++

- Example : Sorting:
  - https://en.cppreference.com/w/cpp/algorithm/sort
  - Example: AdvCpp/sorting.cpp



std::sort

Defined in header `<algorithm>`

`template< class RandomIt >`
`void sort( RandomIt first, RandomIt last );`
	(1)	(until C++20)

`template< class RandomIt >`
`constexpr void sort( RandomIt first, RandomIt last );`
		(since C++20)

`template< class ExecutionPolicy, class RandomIt >`
`void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt last );`
	(2)	(since C++17)

`template< class RandomIt, class Compare >`
`void sort( RandomIt first, RandomIt last, Compare comp );`
	(3)	(until C++20)

`template< class RandomIt, class Compare >`
`constexpr void sort( RandomIt first, RandomIt last, Compare comp );`
		(since C++20)

`template< class ExecutionPolicy, class RandomIt, class Compare >`
`void sort( ExecutionPolicy&& policy, RandomIt first, RandomIt last, Compare comp );`
	(4)	(since C++17)