C++11 Smart Pointers and Algorithms

Howard Hinnant

May 16, 2012

Outline

- unique_ptr
- shared_ptr
- algorithms

Outline

- unique_ptr
- shared_ptr
- algorithms

unique_ptr

unique_ptr

- It is just like `auto_ptr`.

unique_ptr

- It is just like `auto_ptr`.
  - But better.

unique_ptr

- Almost anything you can do with `auto_ptr`, you can do with `unique_ptr` using the same syntax:

```
auto_ptr<int>
factory(int i)
{
    return auto_ptr<int>(new int(i));
}
```

unique_ptr

- Almost anything you can do with `auto_ptr`, you can do with `unique_ptr` using the same syntax:

```
unique_ptr<int>
factory(int i)
{
    return unique_ptr<int>(new int(i));
}
```

unique_ptr

- Almost anything you can do with `auto_ptr`, you can do with `unique_ptr` using the same syntax:

```
void client(auto_ptr<int> p)
{
    // Ownership transferred into client
}   // int* deleted here
```

unique_ptr

- Almost anything you can do with `auto_ptr`, you can do with `unique_ptr` using the same syntax:

```
void client(unique_ptr<int> p)
{
    // Ownership transferred into client
}   // int* deleted here
```

unique_ptr

- Almost anything you can do with `auto_ptr`, you can do with `unique_ptr` using the same syntax:

```
void test()
{
    auto_ptr<int> p = factory(2);
    // *p == 2
    p.reset(new int(3));
    // *p == 3
    client(factory(4));
}
```

unique_ptr

- Almost anything you can do with `auto_ptr`, you can do with `unique_ptr` using the same syntax:

```
void test()
{
    unique_ptr<int> p = factory(2);
    // *p == 2
    p.reset(new int(3));
    // *p == 3
    client(factory(4));
}
```

unique_ptr

- Ok, so what is the point of `unique_ptr`?

unique_ptr

- It is both easy and common to write generic code that looks like this:

unique_ptr

- It is both easy and common to write generic code that looks like this:

```
template <class T>
void foo(T t)
{
    T copy_of_t = t;
    assert(copy_of_t == t);
}
```

This assert is most often implicit in the code logic (not a literal explicit assert).

unique_ptr

- It is both easy and common to write generic code that looks like this:
- Early implementations of sort did just this by copying the pivot element from the sequence, with the subsequent logic assuming this was a copy.

```
template <class I>
void sort(I first, I last)
{
    // ...
    value_type pivot = *middle;
    // ...
}
```

unique_ptr

- It is both easy and common to write generic code that looks like this:
- Early implementations of sort did just this by copying the pivot element from the sequence, with the subsequent logic assuming this was a copy.
  - Sorting sequences of `auto_ptr` subsequently failed at run time because the expression that looked like a copy was really a move.

unique_ptr

unique_ptr

- In order to be safely usable in generic code, copying must have copy syntax, and moving must have some **other** syntax.

unique_ptr

- In order to be safely usable in generic code, copying must have copy syntax, and moving must have some **other** syntax.
- `auto_ptr` is unsafe because it moves with copy syntax. It is now deprecated.

unique_ptr

- In order to be safely usable in generic code, copying must have copy syntax, and moving must have some **other** syntax.
- `auto_ptr` is unsafe because it moves with copy syntax. It is now deprecated.
- `unique_ptr` will not compile if copy syntax is used. But it can be moved with syntax that can not be mistaken for a copy.

unique_ptr

- unique_ptr is a “move-only” type.
  - It can not be copied, but it can be moved.

unique_ptr

- unique_ptr is a “move-only” type.
  - It can not be copied, but it can be moved.

```
unique_ptr<int> p1(new int(3));

unique_ptr<int> p2 = p1;  // Does not compile!
```

unique_ptr

- unique_ptr is a “move-only” type.
  - It can not be copied, but it can be moved.

```
unique_ptr<int> p1(new int(3));
unique_ptr<int> p2 = std::move(p1);  // Ok!
```

unique_ptr

- unique_ptr is a “move-only” type.
  - It can not be copied, but it can be moved.

```
unique_ptr<int> p1(new int(3));

unique_ptr<int> p2 = source();  // Also Ok!
```

unique_ptr

- unique_ptr<Derived> can convert to unique_ptr<Base>.

unique_ptr

- unique_ptr<Derived> can convert to unique_ptr<Base>.

```
unique_ptr<Derived> source();  // function

unique_ptr<Base> p = source();
```

~Base() must be virtual

unique_ptr

- unique_ptr has a custom deleter.

unique_ptr

- unique_ptr has a custom deleter.

```
struct close_stream
{
    void operator()(std::ofstream* os) const
         {os->close();}
};

typedef unique_ptr<std::ofstream, close_stream>
        FilePtr;
```

which does this

point to this

and call this at destruction

unique_ptr

- unique_ptr has a custom deleter.

```
typedef unique_ptr<std::ofstream, close_stream>
        FilePtr;
```

unique_ptr

- unique_ptr has a custom deleter.

```
typedef unique_ptr<std::ofstream, close_stream>
        FilePtr;
```

- FilePtr owns the open state of an ofstream, not the object itself.

unique_ptr

- unique_ptr has a custom deleter.

```
typedef unique_ptr<std::ofstream, close_stream>
        FilePtr;
```

unique_ptr

- unique_ptr has a custom deleter.

```
typedef unique_ptr<std::ofstream, close_stream>
        FilePtr;

FilePtr get_log()
{
    static std::ofstream log_file;
    log_file.open("log file");
    return FilePtr(&log_file);
}

void foo()
{
    FilePtr fp = get_log();
    *fp << "some text\n";
}   // fp->close()
```

unique_ptr

- unique_ptr has a custom deleter.

unique_ptr

- unique_ptr has a custom deleter.

- If the deleter is “empty”, space for it will be optimized away.
  - The default deleters are empty.
  - sizeof(unique_ptr<T>) == sizeof(T*)

unique_ptr

- unique_ptr has a custom deleter.

unique_ptr

- unique_ptr has a custom deleter.

  - A deleter can be an lvalue reference type.
  - This can be useful if you want to keep the state for a deleter in one place.

```
unique_ptr<T, MyDeleter&> p(ptr, deleter);
```

unique_ptr

- unique_ptr has a custom deleter.
    - A deleter can be a function pointer.

unique_ptr

- unique_ptr has a custom deleter.
  - A deleter can be a function pointer.

```
template <class T>
unique_ptr<char, void(*)(void*)>
type_name(const T&)
{
  return unique_ptr<char, void(*)(void*)>
    (
      __cxa_demangle(typeid(T).name(), nullptr,
                             nullptr, nullptr),
      free
    );
}
```

This returns malloc'd memory memory.

This will be used to deallocate the memory.

unique_ptr

- unique_ptr has a custom deleter.

```
template <class T>
unique_ptr<char, void(*)(void*)>
type_name(const T&)
{
  return unique_ptr<char, void(*)(void*)>
    (
      __cxa_demangle(typeid(T).name(), nullptr,
                                  nullptr, nullptr),
      free
    );
}
```

unique_ptr

- unique_ptr has a custom deleter.

```
template <class T>
unique_ptr<char, void(*)(void*)>
type_name(const T&)
{
  return unique_ptr<char, void(*)(void*)>
    (
      __cxa_demangle(typeid(T).name(), nullptr,
                     nullptr, nullptr),
      free
    );
}
```

```
cout << type_name(x).get() << '\n';
```

unique_ptr

```
cout << type_name(x).get() << ‘\n’;
```

- compare to:

```
char* name = nullptr;
try
{
  name = __cxa_demangle(typeid(x).name(), nullptr,
                                        nullptr, nullptr);
  cout << name << '\n';
  free(name);
}
catch (...)
{
  free(name);
  throw;
}
```

unique_ptr

- unique_ptr has an array form:

unique_ptr

- unique_ptr has an array form:

```
unique_ptr<T[], D>
```

unique_ptr

- unique_ptr has an array form:

```
unique_ptr<T[], D>
```

- This `unique_ptr` does not have `operator*()`.

unique_ptr

- unique_ptr has an array form:

```
unique_ptr<T[], D>
```

- This `unique_ptr` does not have `operator*()`.
- Nor will it convert from derived to base.

unique_ptr

- unique_ptr has an array form:

```
unique_ptr<T[], D>
```

- This `unique_ptr` does not have `operator*()`.
- Nor will it convert from derived to base.
- But it does have `T& operator[](size_t) const`.

unique_ptr

- unique_ptr has an array form:

```
unique_ptr<T[], D>
```

- This `unique_ptr` does not have `operator*()`.
- Nor will it convert from derived to base.
- But it does have `T& operator[](size_t) const`.
- The default deleter uses delete[].

```
unique_ptr<char[]> p(new char[10]);
p[0] = ‘a’;
```

unique_ptr

- unique_ptr has an array form:

```
template <class T>
unique_ptr<char[], void(*)(void*)>
type_name(const T&)
{
    return unique_ptr<char[], void(*)(void*)>
      (
        __cxa_demangle(typeid(T).name(), nullptr,
                              nullptr, nullptr),
        free
      );
}
```

unique_ptr

- unique_ptr has an array form:

```
template <class T>
unique_ptr<char[], void(*)(void*)>
type_name(const T&)
{
  return unique_ptr<char[], void(*)(void*)>
    (
      __cxa_demangle(typeid(T).name(), nullptr,
                     nullptr, nullptr),
      free
    );
}

auto nm = type_name(x);
for (int i = 0; nm[i]; ++i)
  cout << nm[i];
```

braces added

deallocation still uses

indexing can now be used

unique_ptr

- unique_ptr has support for incomplete types (useful for pimpl).

unique_ptr

- unique_ptr has support for incomplete types (useful for pimpl).

```
class A
{
    class impl;
    unique_ptr<impl> ptr_;
public:
    A();
    A(A&&);
    A& operator=(A&&);
    ~A();
};
```

unique_ptr

- unique_ptr has support for incomplete types (useful for pimpl).

```
class A::impl {};

A::A() = default;
A::A(A&&) = default;
A& A::operator=(A&&) = default;
A::~A() = default;
```

unique_ptr

- unique_ptr has support for incomplete types (useful for pimpl).

```
class A::impl {};

A::A() = default;
A::A(A&&) = default;
A& A::operator=(A&&) = default;
A::~A() = default;
```

- Each special member can be defaulted, but must be outlined into a source once `A::impl` is complete.

unique_ptr

- unique_ptr has support for incomplete types (useful for pimpl).
- If you accidentally attempt to do anything with the incomplete `A::impl` that is not allowed, a compile-time error is guaranteed.

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).
- Use this to put unique_ptr into process-shared memory, using an “offset_ptr” as the storage.

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

```
template <class T>
struct MyDeleter
{
  struct pointer
  {
    // emulate a pointer ...
  };

  void operator()(pointer p);
};

unique_ptr<int, MyDeleter<int>> p;
```

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

```
unique_ptr<int, MyDeleter<int>> p;
```

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

```
unique_ptr<int, MyDeleter<int>> p;
```

```
unique_ptr<int, MyDeleter<int>>::pointer
```

is the same type as

```
MyDeleter<int>::pointer
```

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

```
unique_ptr<int, MyDeleter<int>> p;
```

```
MyDeleter<int>::pointer
```

```
unique_ptr<int, MyDeleter<int>>::pointer
```

unique_ptr

- unique_ptr has support for custom storage (internal pointer type).

```
unique_ptr<int, MyDeleter<int>> p;
```

If

`MyDeleter<int>::pointer`

does not exist, then

`unique_ptr<int, MyDeleter<int>>::pointer`

is

`int*`

unique_ptr

- unique_ptr can be put into containers and manipulated with algorithms.

unique_ptr

- unique_ptr can be put into containers and manipulated with algorithms.

```
int main() {
  typedef unique_ptr<int> Ptr;
  Ptr p[] = {Ptr(new int(2)), Ptr(new int(3)),
             Ptr(new int(1))};
```

unique_ptr

- unique_ptr can be put into containers and manipulated with algorithms.

```
int main() {
  typedef unique_ptr<int> Ptr;
  Ptr p[] = {Ptr(new int(2)), Ptr(new int(3)),
             Ptr(new int(1))};
  vector<Ptr> v(make_move_iterator(begin(p)),
                make_move_iterator(end(p)));
```

unique_ptr

- unique_ptr can be put into containers and manipulated with algorithms.

```
int main() {
  typedef unique_ptr<int> Ptr;
  Ptr p[] = {Ptr(new int(2)), Ptr(new int(3)),
             Ptr(new int(1))};
  vector<Ptr> v(make_move_iterator(begin(p)),
                make_move_iterator(end(p)));
  sort(v.begin(), v.end(), [](const Ptr& x,
                              const Ptr& y)
                             {return *x < *y;});
```

unique_ptr

- unique_ptr can be put into containers and manipulated with algorithms.

```
int main() {
  typedef unique_ptr<int> Ptr;
  Ptr p[] = {Ptr(new int(2)), Ptr(new int(3)),
             Ptr(new int(1))};
  vector<Ptr> v(make_move_iterator(begin(p)),
                make_move_iterator(end(p)));
  sort(v.begin(), v.end(), [](const Ptr& x,
                              const Ptr& y)
                             {return *x < *y;});
  for (const auto& p : v)
    cout << *p << ' ';
  cout << '\n';
}
```

unique_ptr

- Use `const unique_ptr<T>` when you want to guarantee that ownership is not transferred out of scope.

unique_ptr

- Use `const unique_ptr<T>` when you want to guarantee that ownership is not transferred out of scope.

```
const unique_ptr<Base> p(new Derived);
// ...
return p;  // Compile-time error
```

unique_ptr

- Use `const unique_ptr<T>` when you want to guarantee that ownership is not transferred out of scope.

```
const unique_ptr<Base> p(new Derived);
// ...
swap(p, p2);  // Compile-time error
```

unique_ptr

- Use `const unique_ptr<T>` when you want to guarantee that ownership is not transferred out of scope.

- `const std::unique_ptr<T>` is arguably a better type than boost::scoped_ptr<T> for guaranteeing that ownership is not transferred out of scope.

Migrating from auto_ptr to unique_ptr

Migrating from auto_ptr to unique_ptr

- It is safe to do a global search for "`auto_ptr`" and replace with "`unique_ptr`".
- If the result compiles, you are good to go.

Migrating from auto_ptr to unique_ptr

- It is safe to do a global search for "`auto_ptr`" and replace with "`unique_ptr`".
- If the result compiles, you are good to go.
- If the result does not compile, it probably looks something like this:

```
p1 = p2;
```

Migrating from auto_ptr to unique_ptr

- If the result does not compile, it probably looks something like this:

Migrating from auto_ptr to unique_ptr

- If the result does not compile, it probably looks something like this:
- Try this instead:

```
p1 = std::move(p2);
```

- But inspect the code to see if `p2` is inappropriately used afterwords:

```
f(p2);
```

- Why is `f()` being called with a moved-from smart pointer?

Migrating from auto_ptr to unique_ptr

- People have confirmed to me that this exercise has found bugs in large projects.

Outline

- unique_ptr
- shared_ptr
- algorithms

Outline

- unique_ptr
- shared_ptr
- algorithms

shared_ptr

- The most bullet-proof, most functional reference counted pointer.
  - This is the same shared_ptr you know and love from boost.

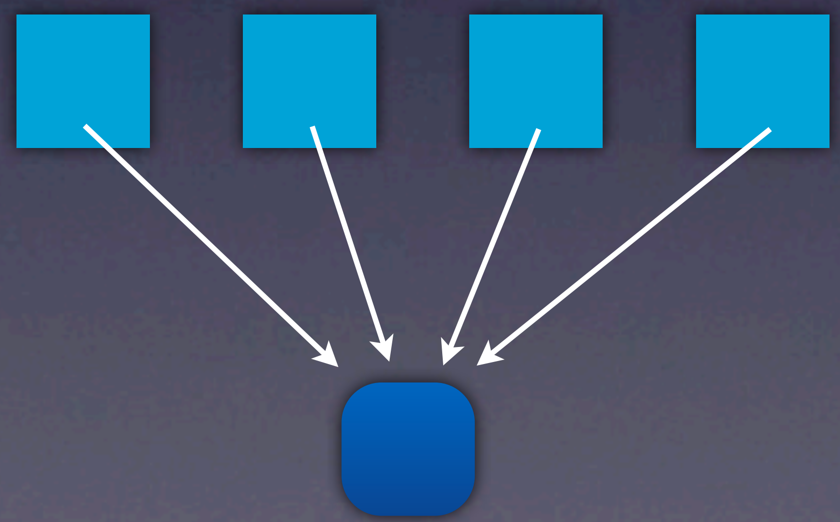
shared_ptr

A contrast with unique_ptr

shared_ptr

A contrast with unique_ptr

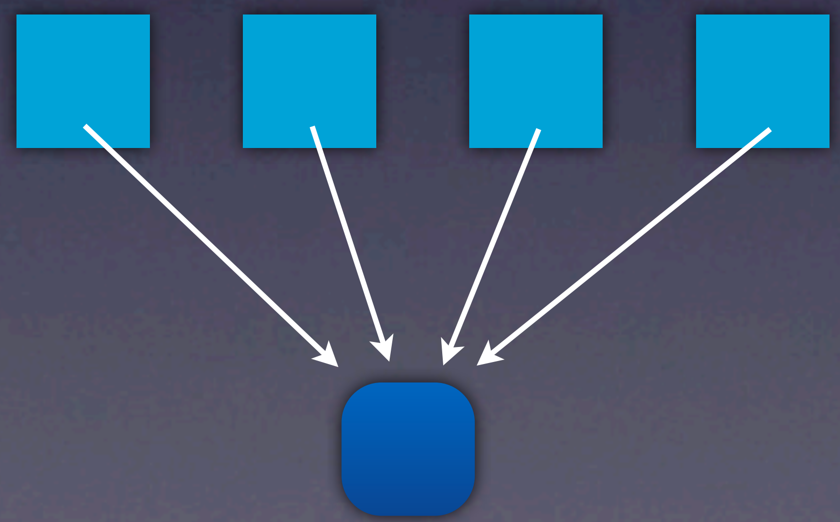
- shared_ptr models shared ownership.



shared_ptr

A contrast with unique_ptr

- shared_ptr models shared ownership.
- unique_ptr models unique ownership.



shared_ptr

A contrast with unique_ptr

shared_ptr

A contrast with unique_ptr

- A unique_ptr can be converted to a shared_ptr.
- But not vice-versa.

shared_ptr

A contrast with unique_ptr

shared_ptr

A contrast with unique_ptr

- sizeof(shared_ptr<T>) == 2 words
- sizeof(unique_ptr<T>) == 1 word

shared_ptr

A contrast with unique_ptr

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - shared_ptr's deleter is not part of its type. It is only specified in a constructor.
  - unique_ptr's deleter is part of the type. One can optionally specify a deleter in the constructor.

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - Clients of a shared_ptr factory function do not need to know about the type of the deleter the shared_ptr was constructed with.
    - There is only one type of shared_ptr.

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - Clients of a shared_ptr factory function do not need to know about the type of the deleter the shared_ptr was constructed with.
    - There is only one type of shared_ptr.
  - Clients of a unique_ptr factory function must know the type of the unique_ptr's deleter.

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - Some of the “hidden deleter” abstraction can be gained for unique_ptr by using a function pointer for the deleter, and hiding what function it points to:

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - Some of the "hidden deleter" abstraction can be gained for unique_ptr by using a function pointer for the deleter, and hiding what function it points to:

```
unique_ptr<T, void(*)(void*)>
source();
```

Deallocated by std::free, or maybe something else?

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - The shared_ptr “type-erased” deleter design requires that the deleter be stored on the heap.
    - shared_ptr<T>(new T) requires two allocations: one for the T and one for the control block holding the deleter.

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - The unique_ptr deleter is stored in the pointer itself.
    - unique_ptr<T>(new T) requires only one allocation.

shared_ptr

A contrast with unique_ptr

- Both shared_ptr and unique_ptr support custom deleters, but...
  - Use of make_shared<T>(Args...) can reduce the number of allocations required to construct a shared_ptr down to one.
  - There is no make_unique<T>(Args...) at this time...

shared_ptr

A contrast with unique_ptr

- shared_ptr<T[]> is not supported at this time.
  - But you can use a custom deleter to get the right deallocation:

shared_ptr

A contrast with unique_ptr

- shared_ptr<T[]> is not supported at this time.
  - But you can use a custom deleter to get the right deallocation:

```
shared_ptr<T> p(new T[3],
                default_delete<T[]>());
```

shared_ptr

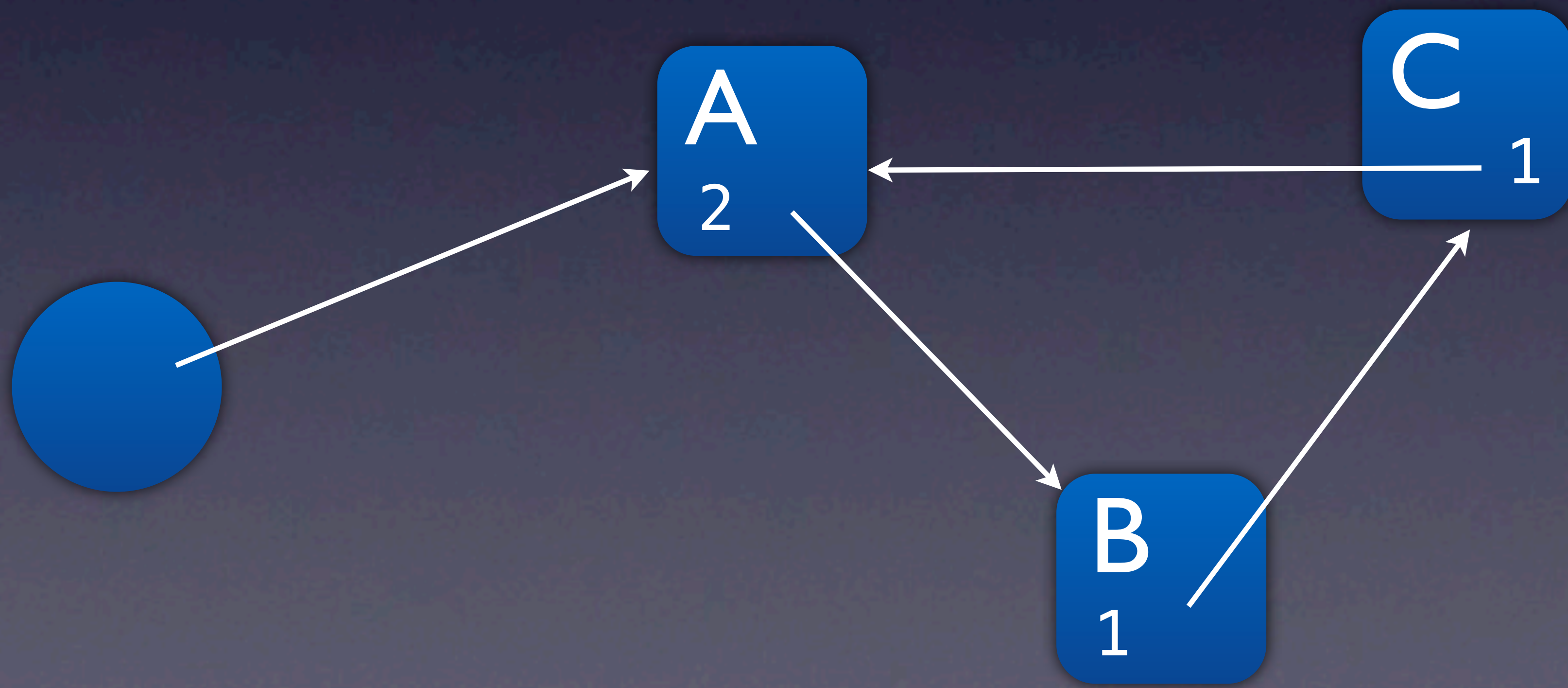
Ownership cycles

- Cyclic ownership is a constant danger when reference counted pointers are used indiscriminately.

shared_ptr

Ownership cycles

- Cyclic ownership is a constant danger when reference counted pointers are used indiscriminately.

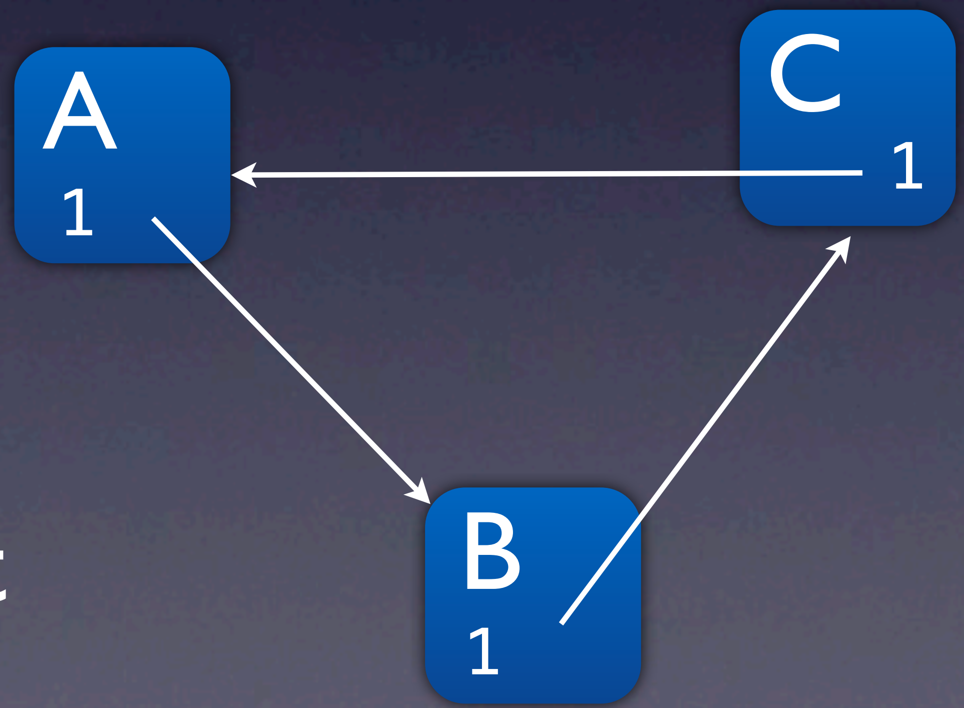


shared_ptr

Ownership cycles

- Cyclic ownership is a constant danger when reference counted pointers are used indiscriminately.

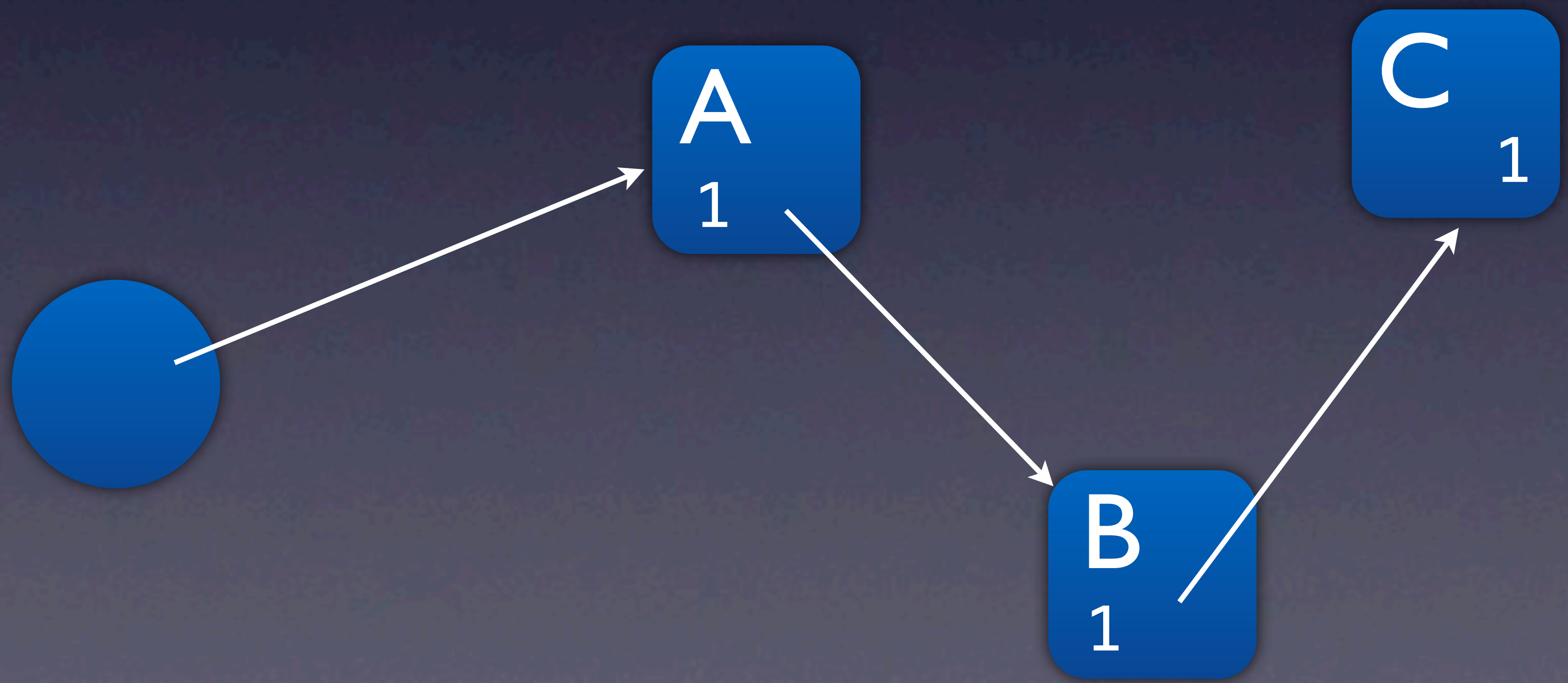
A cycle can not be deleted because there is always a reference count keeping everything alive.



shared_ptr

Ownership cycles

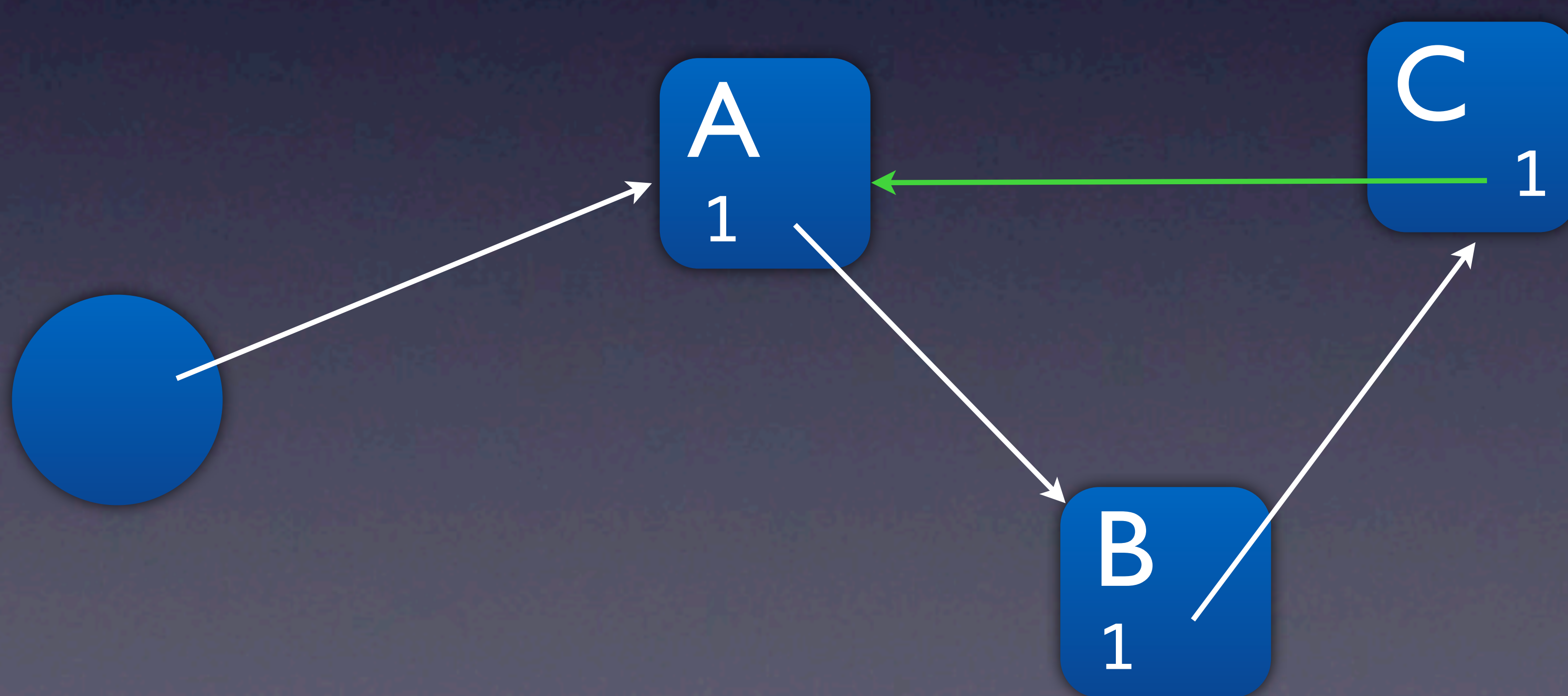
- The solution is to use weak_ptr to break the cycle.
- weak_ptr does not own what it points to.
- weak_ptr knows when its pointee gets deleted.



shared_ptr

Ownership cycles

- The solution is to use weak_ptr to break the cycle.
- weak_ptr does not own what it points to.
- weak_ptr knows when its pointee gets deleted.



shared_ptr

Ownership cycles

- The solution is to use weak_ptr to break the cycle.
- weak_ptr does not own what it points to.
- weak_ptr knows when its pointee gets deleted.

shared_ptr

Ownership cycles

```
class C
{
  shared_ptr<A> ptr_;
public:
  C() = default;
  ~C() {std::cout << "~C()\n";}
  void set(shared_ptr<A> p) {ptr_ = p;}
};
```

shared_ptr

Ownership cycles

To fix, simply replace shared_ptr with weak_ptr.

```
class C
{
  weak_ptr<A> ptr_;
public:
  C() = default;
  ~C() {std::cout << "~C()\n";}
  void set(shared_ptr<A> p) {ptr_ = p;}
};
```

shared_ptr

Ownership cycles

To fix, simply replace shared_ptr with weak_ptr.

```
class C
{

public:
  C() = default;
  ~C() {std::cout << "~C()\n";}
  void set(shared_ptr<A> p) {ptr_ = p;}
};
```

shared_ptr

weak_ptr

- weak_ptr must be converted to shared_ptr in order to be dereferenced.
- Explicit construction will throw `bad_weak_ptr` is expired.

```
class C
{
  weak_ptr<A> ptr_;
public:
  shared_ptr<A> get() const
    {return shared_ptr<A>(ptr_);}
};
```

shared_ptr

weak_ptr

- weak_ptr must be converted to shared_ptr in order to be dereferenced.
- Use `lock()` to instead return a null `shared_ptr` when `expired`.

```
class C
{
  weak_ptr<A> ptr_;
public:
  shared_ptr<A> get() const

};
```

shared_ptr

weak_ptr

- weak_ptr must be converted to shared_ptr in order to be dereferenced.
- Use `lock()` to instead return a null `shared_ptr` when `expired`.

```
class C
{
  weak_ptr<A> ptr_;
public:
  shared_ptr<A> get() const
    {return ptr_.lock();}
};
```

shared_ptr

weak_ptr

- Not being able to directly dereference a `weak_ptr` is a critical safety feature in multithreaded code.

shared_ptr

weak_ptr

- Not being able to directly dereference a `weak_ptr` is a critical safety feature in multithreaded code.

Thread A

```
weak_ptr<A> wp = ...
if (!wp.expired())
  wp->do_something();
```

Thread B

```
sp.reset();
```

Won't even compile!

What wp refers to could get destructed during do_something()!

shared_ptr

weak_ptr

- Not being able to directly dereference a `weak_ptr` is a critical safety feature in multithreaded code.

Thread A | Thread B

```
weak_ptr<A> wp = ...
shared_ptr<A> sp(wp);
sp->do_something();              sp.reset();
```

shared_ptr

weak_ptr

- Not being able to directly dereference a `weak_ptr` is a critical safety feature in multithreaded code.

Thread A

```
weak_ptr<A> wp = ...
shared_ptr<A> sp(wp);
sp->do_something();
```

Thread B

```
sp.reset();
```

A successful conversion to `shared_ptr`, atomically extends the life time long enough to complete `do_something()`!

Which Smart Pointer is Preferred?

- So which smart pointer should I reach for first?

Which Smart Pointer is Preferred?

- Neither!

Which Smart Pointer is Preferred?

- Prefer holding data members directly.

```
class A
{
  B b_;
};
```

- Use pointers, even smart ones, sparingly.

Which Smart Pointer is Preferred?

- Use a smart pointer when you need to point to a base class.

```
class A
{
  unique_ptr<Base> ptr_;
};
```

- Prefer `unique_ptr` to model a single owner.
- Unique ownership is simpler to reason about than shared ownership.

Which Smart Pointer is Preferred?

```
class A
{
  unique_ptr<Base> ptr_;
public:
   A(const A& a)
      : ptr_(a.ptr_ ?
                 a.ptr_->clone() :
                 nullptr) {}
  A(A&&) noexcept = default;
  // ...
};
```

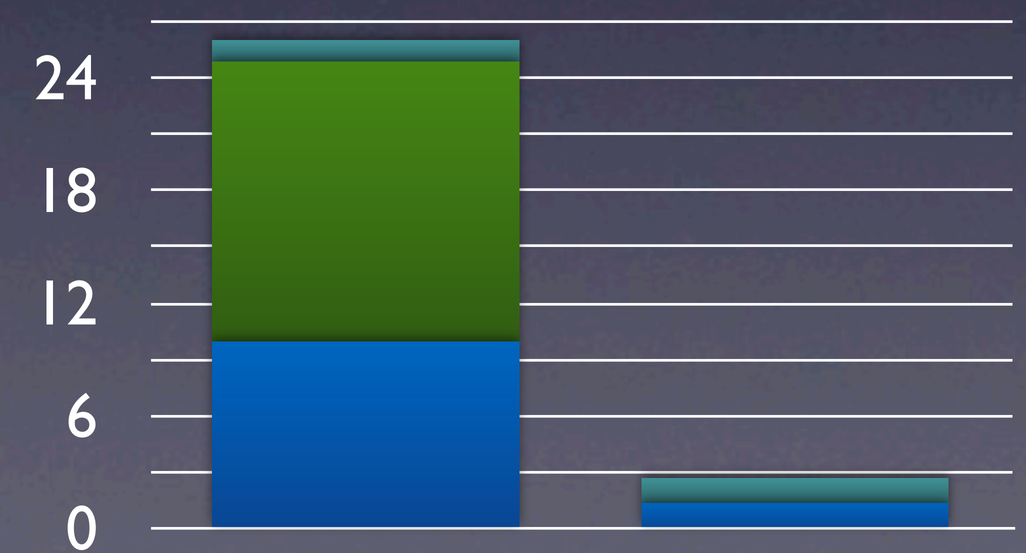
- Do not choose `shared_ptr` just to enable cheap copies. Let move semantics take care of that.

Which Smart Pointer is Preferred?

- Do not choose `shared_ptr` just to enable cheap copies. Let move semantics take care of that.
  - Most copies turn into moves in C++11.
  - Moving a unique_ptr is twice as fast as moving a shared_ptr.

Which Smart Pointer is Preferred?

- Do not choose `shared_ptr` just to enable cheap copies. Let move semantics take care of that.
    - Most copies turn into moves in C++11.
    - Moving a unique_ptr is twice as fast as moving a shared_ptr.



Which Smart Pointer is Preferred?

```
class A
{
  shared_ptr<Base> ptr_;
public:
  // ...
};
```

- Choose `shared_ptr` only when you actually need shared ownership semantics, or when you know you will need copies (not moves) and the pointee is *always* immutable.

Outline

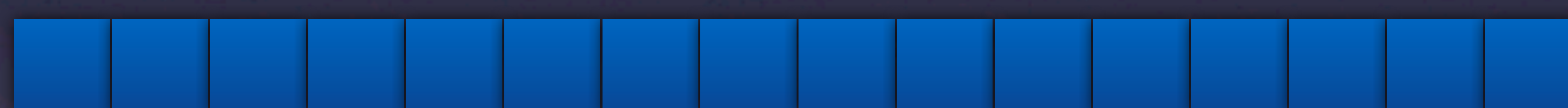
- unique_ptr
- shared_ptr
- algorithms

Outline

- unique_ptr
- shared_ptr
- algorithms

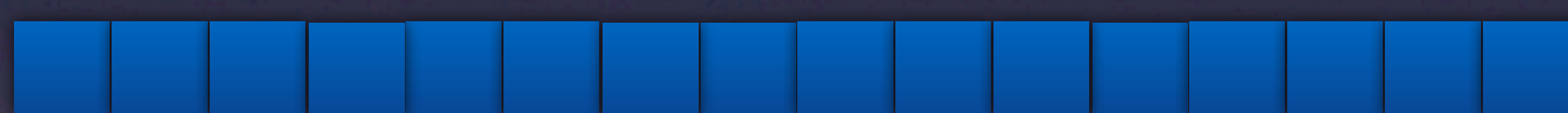
<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.



<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.



- These algorithms no longer even require copyability: they will work with move-only types such as unique_ptr<T>.

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

random_shuffle

swap_ranges

sort

push_heap

iter_swap

inplace_merge

remove

pop_heap

remove_if

partition

stable_partition

stable_sort

make_heap

next_permutation

prev_permutation

reverse

partial_sort

unique

sort_heap

rotate

nth_element

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

```
vector<unique_ptr<T>> v = ...
sort(v.begin(), v.end(),
                   [](const unique_ptr<T>& x,
                      const unique_ptr<T>& y)
                         {return *x < *y;});
```

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

```
vector<unique_ptr<T>> v = ...
rotate(v.begin(), v.begin()+5, v.end());
```

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

```
vector<unique_ptr<T>> v = ...
remove(v.begin(), v.end(), nullptr);
```

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

```
vector<unique_ptr<T>> v = ...
remove(v.begin(), v.end(), nullptr);
```

Remove all of the nulls in the sequence.

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

```
vector<unique_ptr<T>> v = ...
```

Consider using:

```
std::vector<std::unique_ptr<T>>
```

over:

```
boost::ptr_vector<T>.
```

<algorithm>

- Many sequence-permuting standard algorithms will now use move or swap, not copy.

```
vector<unique_ptr<T>> v = ...
```

Consider using:

```
std::vector<std::unique_ptr<T>>
```

over:

```
boost::ptr_vector<T>.
```

You get a wider range of available algorithms.

<algorithm>

New algorithms

- There are about 20 new algorithms in C++11

<algorithm>

New algorithms

- There are about 20 new algorithms in C++11
  - Here are a few of my favorites...

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element
  - Find both the minimum and maximum.

```
template<class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
  minmax_element(ForwardIterator first,
                 ForwardIterator last);

template<class ForwardIterator,
         class Compare>
  pair<ForwardIterator, ForwardIterator>
  minmax_element(ForwardIterator first,
                 ForwardIterator last,
                 Compare comp);
```

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element

```
struct A
{
  static int comp_count;

  int data;
  A(int d) : data(d) {}

  friend std::ostream&
  operator<<(std::ostream& os, const A& x)
    {return os << x.data;}
};
```

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element

```
int A::comp_count = 0;

bool
operator <(const A& x, const A& y)
{
    ++A::comp_count;
    return x.data < y.data;
}
```

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element

```
int main()
{
   mt19937_64 eng;
   uniform_int_distribution<> d(0, 1000000);
   vector<A> v;
   for (int i = 0; i < 1000; ++i)
     v.push_back(d(eng));
   auto p = minmax_element(v.begin(), v.end());
   cout << "v.size() = " << v.size() << '\n';
   cout << "min = " << *p.first << '\n';
   cout << "max = " << *p.second << '\n';
   cout << "# of comparisons = "
        << A::comp_count << '\n';
}
```

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element

```
v.size() = 1000
min = 629
max = 999101
# of comparisons = 1498
```

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element

```
auto p =
    std::make_pair
    (
      std::min_element(v.begin(), v.end()),
      std::max_element(v.begin(), v.end())
    );
```

<algorithm>

New algorithms

- minmax_element

<algorithm>

New algorithms

- minmax_element

minmax

```
v.size() = 1000
min = 629
max = 999101
# of comparisons = 1498
```

min + max

```
v.size() = 1000
min = 629
max = 999101
# of comparisons = 1998
```

minmax is at least 33% faster than using min and max!

And it is easier to use.

<algorithm>

New algorithms

- is_sorted_until

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

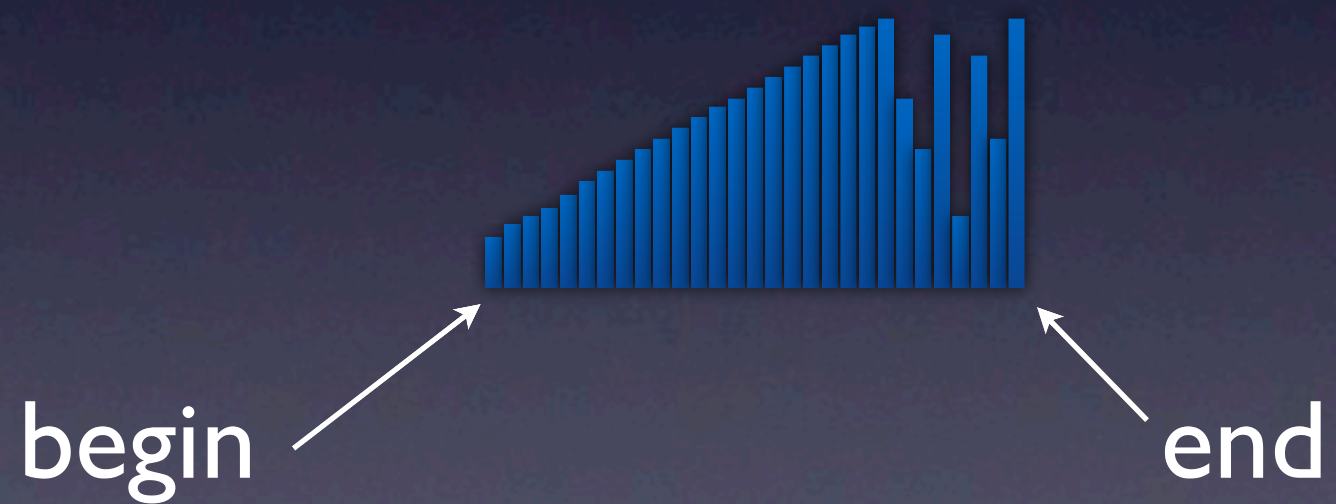
```
template<class ForwardIterator>
ForwardIterator
is_sorted_until(ForwardIterator first,
                ForwardIterator last);

template <class ForwardIterator,
          class Compare>
ForwardIterator
is_sorted_until(ForwardIterator first,
                ForwardIterator last,
                Compare comp);
```

<algorithm>

New algorithms

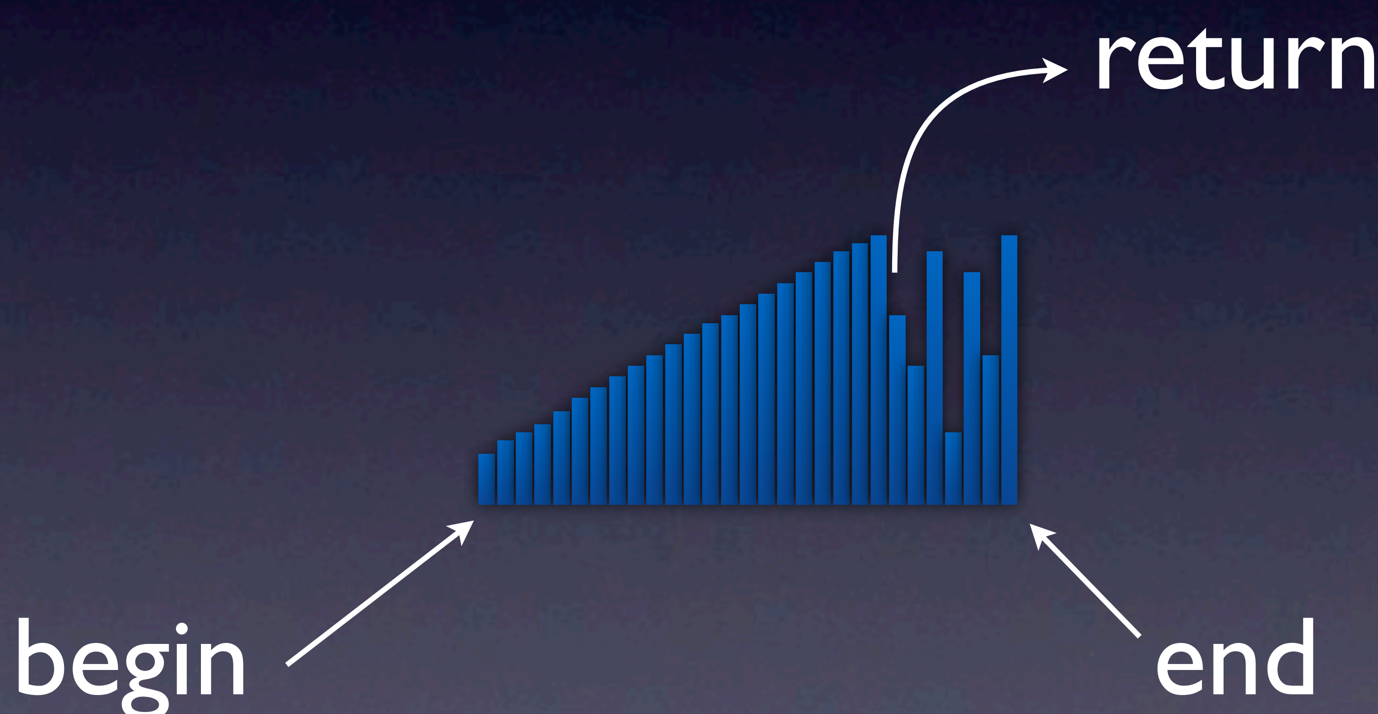
- is_sorted_until
  - Find sorted prefix of sequence.



<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.



<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.
  - Can be used for faster sorts when it is known that there is a large sorted prefix.

```
std::vector<int> v;
for (int i = 0; i < 100000; ++i)
  v.push_back(i);
for (int i = 0; i < 10000; ++i)
  v.push_back(d(eng));
```

↑ `uniform_int_distribution`

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
typedef std::chrono::high_resolution_clock Clock;
typedef std::chrono::duration<float, std::micro>
                                         microsec;
auto t0 = Clock::now();



auto t1 = Clock::now();
std::cout << microsec(t1-t0).count() << " ms\n";
```

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
typedef std::chrono::high_resolution_clock Clock;
typedef std::chrono::duration<float, std::micro>
                                         microsec;
auto t0 = Clock::now();

auto t1 = Clock::now();
std::cout << microsec(t1-t0).count() << " ms\n";
```

High resolution timer boiler plate

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
typedef std::chrono::high_resolution_clock Clock;
typedef std::chrono::duration<float, std::micro>
                                        microsec;
auto t0 = Clock::now();

std::stable_sort(v.begin(), v.end());

auto t1 = Clock::now();
std::cout << microsec(t1-t0).count() << " ms\n";
```

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
typedef std::chrono::high_resolution_clock Clock;
typedef std::chrono::duration<float, std::micro>
                                        microsec;
auto t0 = Clock::now();
auto i = std::is_sorted_until(v.begin(), v.end());
std::stable_sort(i, v.end());
std::inplace_merge(v.begin(), i, v.end());
auto t1 = Clock::now();
std::cout << microsec(t1-t0).count() << " ms\n";
```

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
auto i = std::is_sorted_until(v.begin(), v.end());
std::stable_sort(i, v.end());
std::inplace_merge(v.begin(), i, v.end());
```

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
auto i = std::is_sorted_until(v.begin(), v.end());
std::stable_sort(i, v.end());
std::inplace_merge(v.begin(), i, v.end());
```

is 2.6* times faster than:

```
std::stable_sort(v.begin(), v.end());
```

<algorithm>

New algorithms

- is_sorted_until
  - Find sorted prefix of sequence.

```
auto i = std::is_sorted_until(v.begin(), v.end());
std::stable_sort(i, v.end());
std::inplace_merge(v.begin(), i, v.end());
```

is 2.6* times faster than:

```
std::stable_sort(v.begin(), v.end());
```

(**your mileage may vary*)

<algorithm>

New algorithms

- is_permuation
  - Determine if one sequence is a permutation of another.

```
template<class FwdItr1, class FwdItr2>
  bool
  is_permutation(FwdItr1 first1, FwdItr1 last1,
                 FwdItr2 first2);

template<class FwdItr1, class FwdItr2,
         class BinaryPred>
  bool
  is_permutation(FwdItr1 first1, FwdItr1 last1,
                 FwdItr2 first2, BinaryPred pred);
```

<algorithm>

New algorithms

- is_permuation
  - Determine if one sequence is a permutation of another.

```
template<class FwdItr1, class FwdItr2,
         class BinaryPred>
  bool
  is_permutation(FwdItr1 first1, FwdItr1 last1,
                 FwdItr2 first2, BinaryPred pred);
```

- Reasonably efficient: Linear if `equal(first1, last1, first2)`.

<algorithm>

New algorithms

- is_permuation
  - Determine if one sequence is a permutation of another.

```
template<class FwdItr1, class FwdItr2,
         class BinaryPred>
  bool
  is_permutation(FwdItr1 first1, FwdItr1 last1,
                 FwdItr2 first2, BinaryPred pred);
```

- Reasonably efficient: Finds “false” cases quickly.

<algorithm>

New algorithms

- is_permuation
  - Determine if one sequence is a permutation of another.

```
int x[] = {1, 2, 3, 4, 5};
const unsigned N = sizeof(x) / sizeof(int);
int y[N] = {2, 1, 3, 5, 4};
int count = 0;
bool b = std::is_permutation(x, x+N, y,
                [&](int a, int b) -> bool
                   {++count; return a == b;});
```

Summary

Summary

- Use unique_ptr for unique ownership.
  - You can use it in containers and with algorithms.

Summary

- Use unique_ptr for unique ownership.
  - You can use it in containers and with algorithms.
- Use shared_ptr for shared ownership.
  - Don't use it just because you need to put it in a container or use it with algorithms.

Summary

- Use unique_ptr for unique ownership.
  - You can use it in containers and with algorithms.
- Use shared_ptr for shared ownership.
  - Don't use it just because you need to put it in a container or use it with algorithms.
- Learn about and take advantage of new algorithms to make your code faster.