

What's New With C++11 Containers?

Howard Hinnant
May 16, 2012

Outline

- Changes & additions to existing containers
- New containers.

Outline

- Changes & additions to existing containers
- New containers.

Move Semantics

Move Semantics

- All containers except `array<T, N>` are “allocator-aware”.

Move Semantics

- All containers except `array<T, N>` are “allocator-aware”.
- All allocator-aware containers now have a move constructor and move assignment operator.

Move Semantics

- All containers except `array<T, N>` are “allocator-aware”.
- All allocator-aware containers now have a move constructor and move assignment operator.
- It is very cheap to return these containers from factory functions.

Move Semantics

- All containers except `array<T, N>` are “allocator-aware”.
- All allocator-aware containers now have a move constructor and move assignment operator.
- It is very cheap to return these containers from factory functions.
- It is very cheap to move these containers into functions with by-value parameters.

Move Semantics

Move Semantics

- All allocator-aware containers will now move elements internally, instead of copy them internally.
- (e.g. vector reallocation / insert / erase).

Move Semantics

- All allocator-aware containers will now move elements internally, instead of copy them internally.
 - (e.g. vector reallocation / insert / erase).
- Therefore they can hold move-only types.
 - unique_ptr, stringstream, etc.

Move Semantics

```
typedef unique_ptr<Animal> Ptr;

vector<Ptr> make_barn() {
    vector<Ptr> v;
    v.push_back(Ptr(new Dog));
    v.push_back(Ptr(new Sheep));
    v.push_back(Ptr(new Cat));
    return v;
}
```

Move Semantics

```
typedef unique_ptr<Animal> Ptr;

vector<Ptr> make_barn() {
    vector<Ptr> v;
    v.push_back(Ptr(new Dog));
    v.push_back(Ptr(new Sheep));
    v.push_back(Ptr(new Cat));
    return v;
}

vector<Ptr> v = make_barn();
for (const auto& p : v)
    p->speak();
```

Move Semantics

```
typedef unique_ptr<Animal> Ptr;
```

```
vector<Ptr> make_barn() {
    vector<Ptr> v;
    v.push_back(Ptr(new Dog));
    v.push_back(Ptr(new Sheep));
    v.push_back(Ptr(new Cat));
    return v;
}
```

```
v.erase(
    remove_if(
        v.begin(), v.end(),
        [](const Ptr& p) {return p->is_sheep();}),
    v.end());
```

Move Semantics

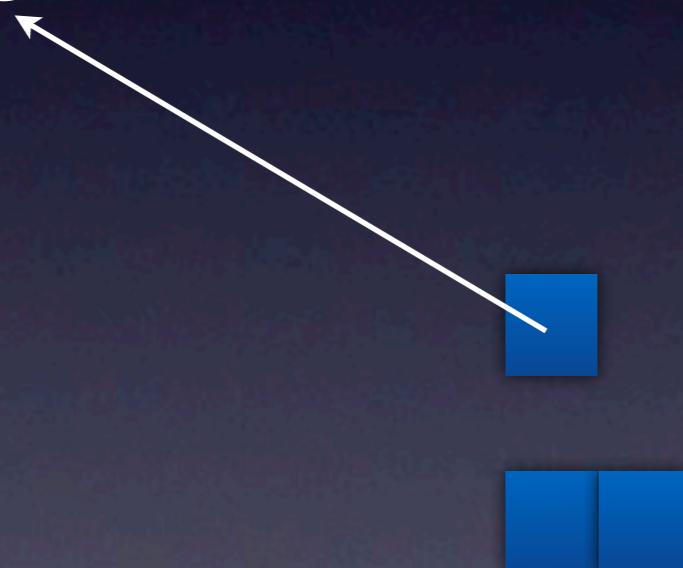
Dog



```
v.push_back(Ptr(new Dog));
```

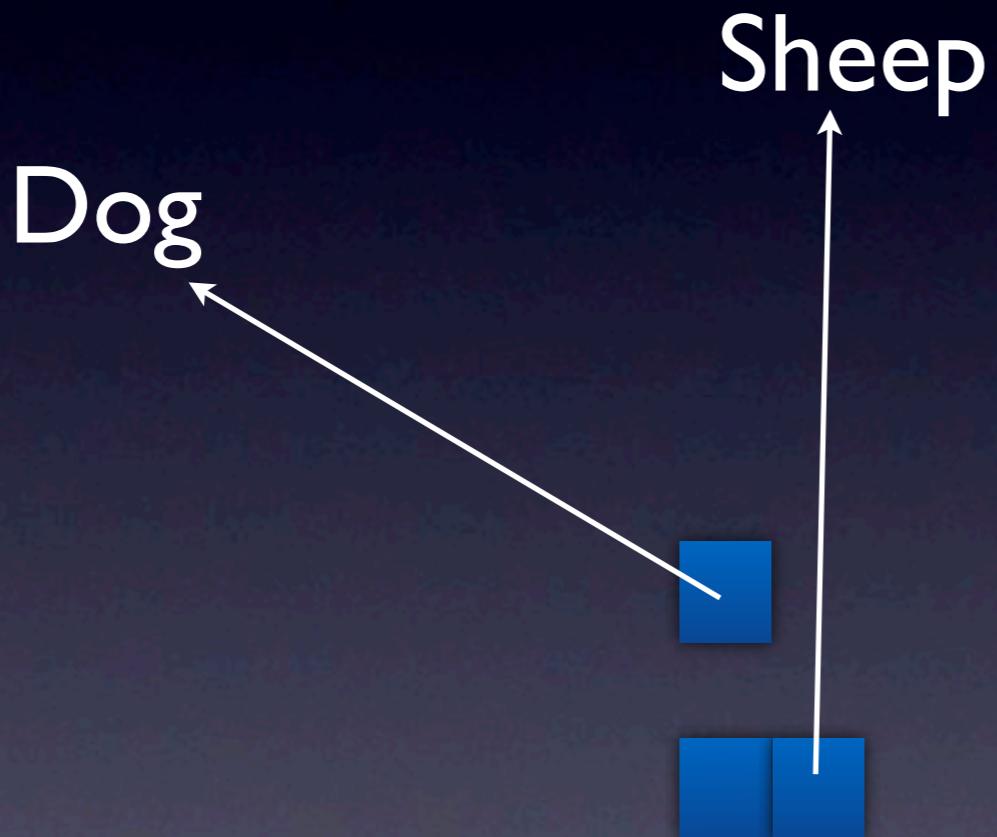
Move Semantics

Dog



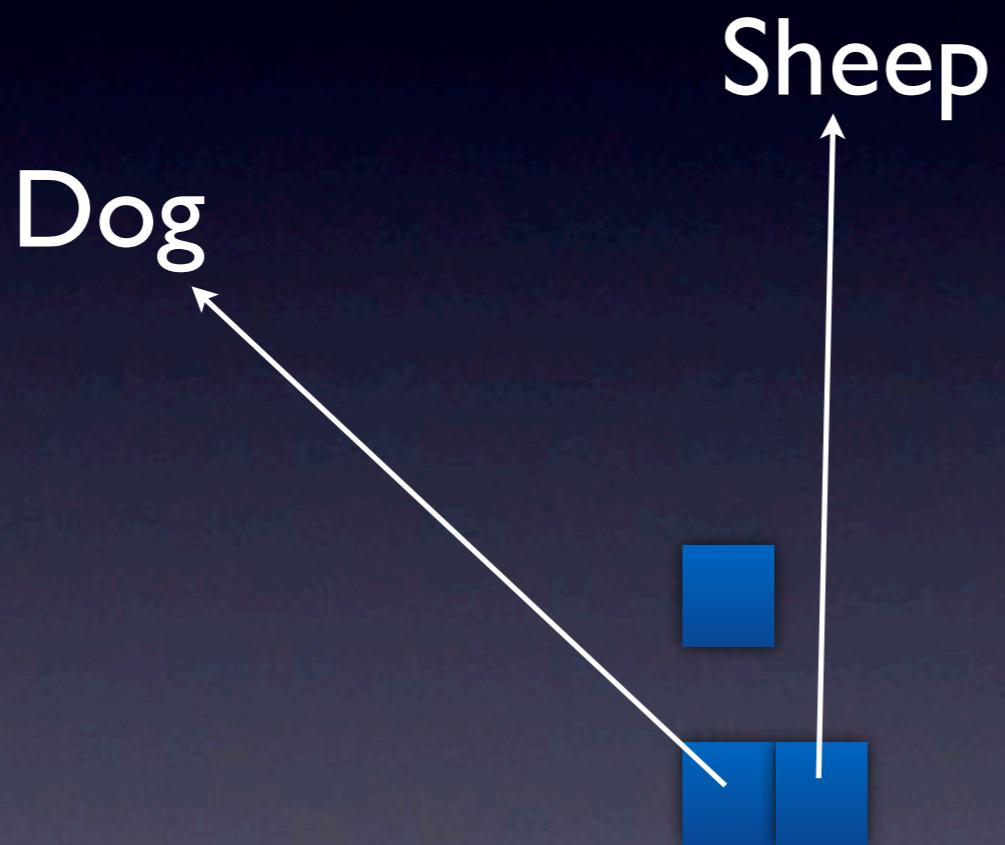
```
v.push_back(Ptr(new Sheep));
```

Move Semantics



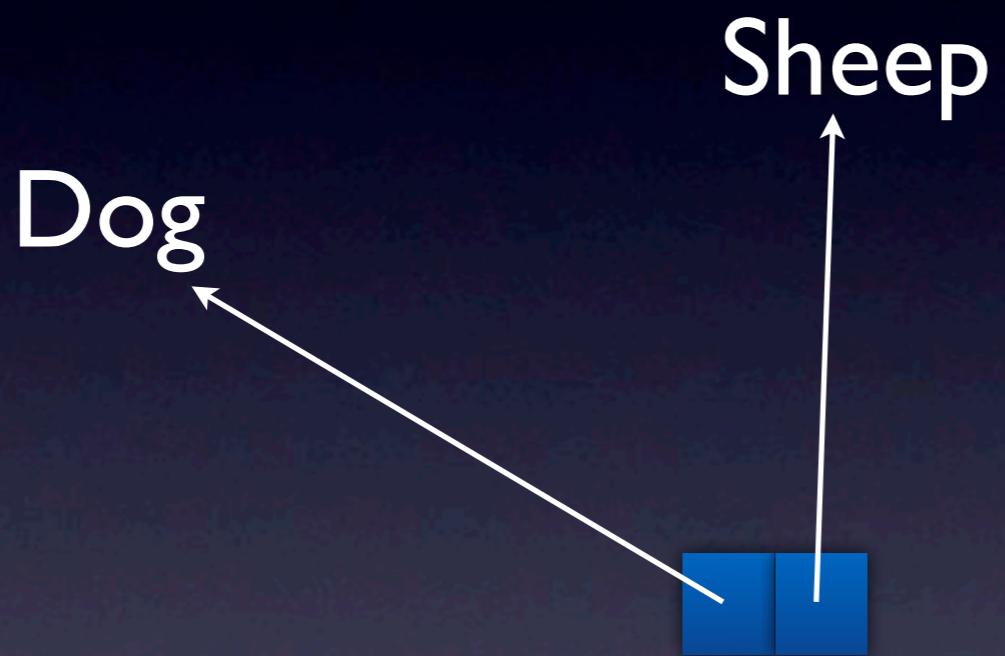
```
v.push_back(Ptr(new Sheep));
```

Move Semantics



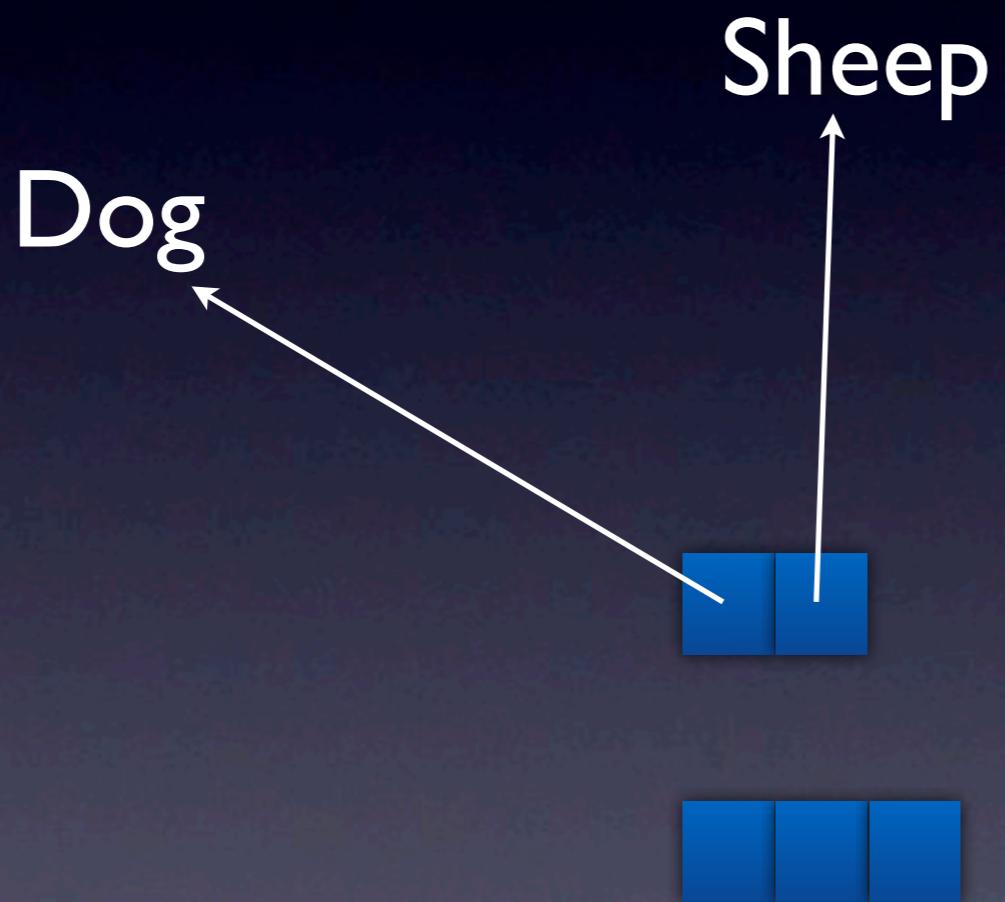
```
v.push_back(Ptr(new Sheep));
```

Move Semantics



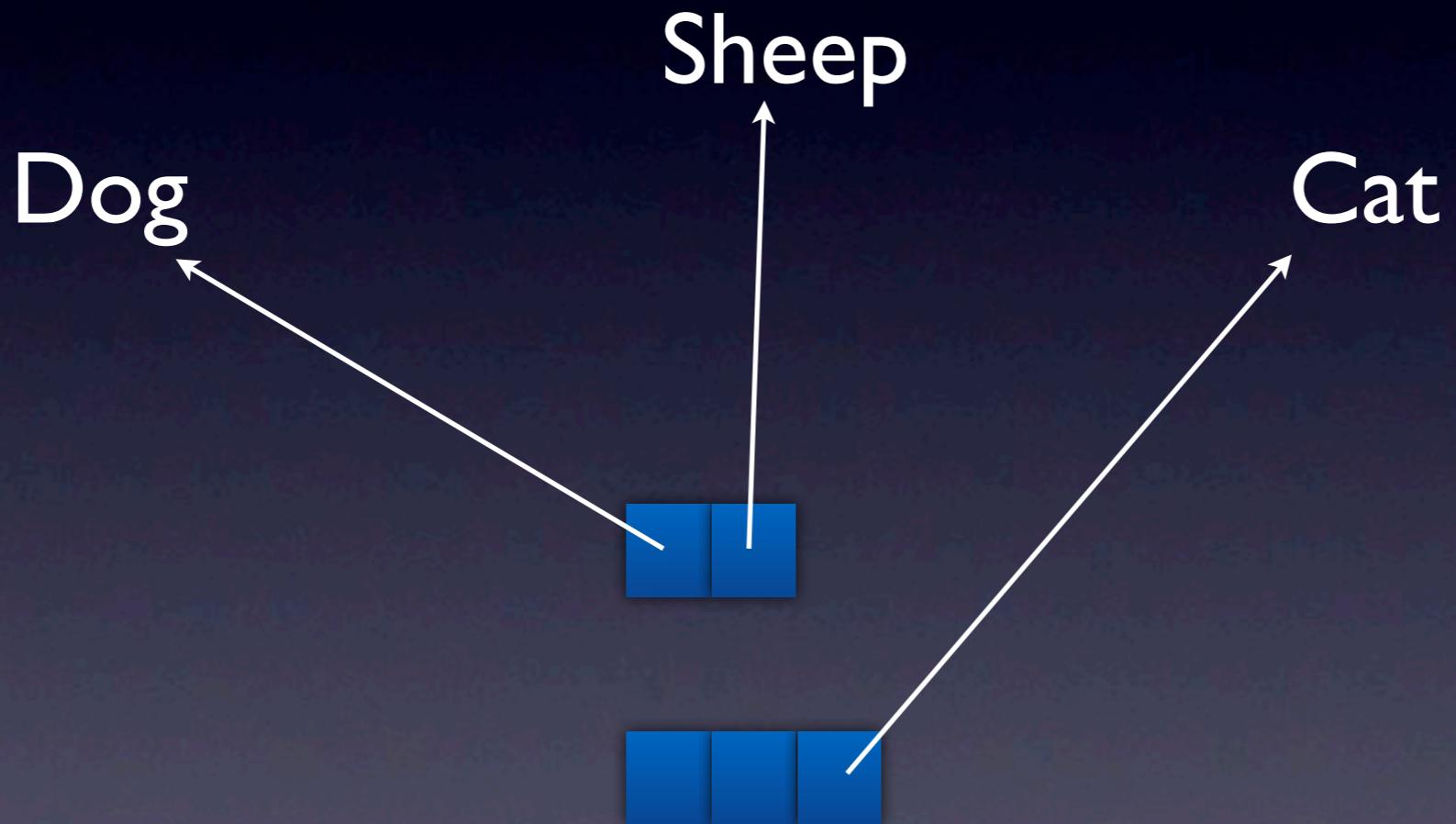
```
v.push_back(Ptr(new Sheep));
```

Move Semantics



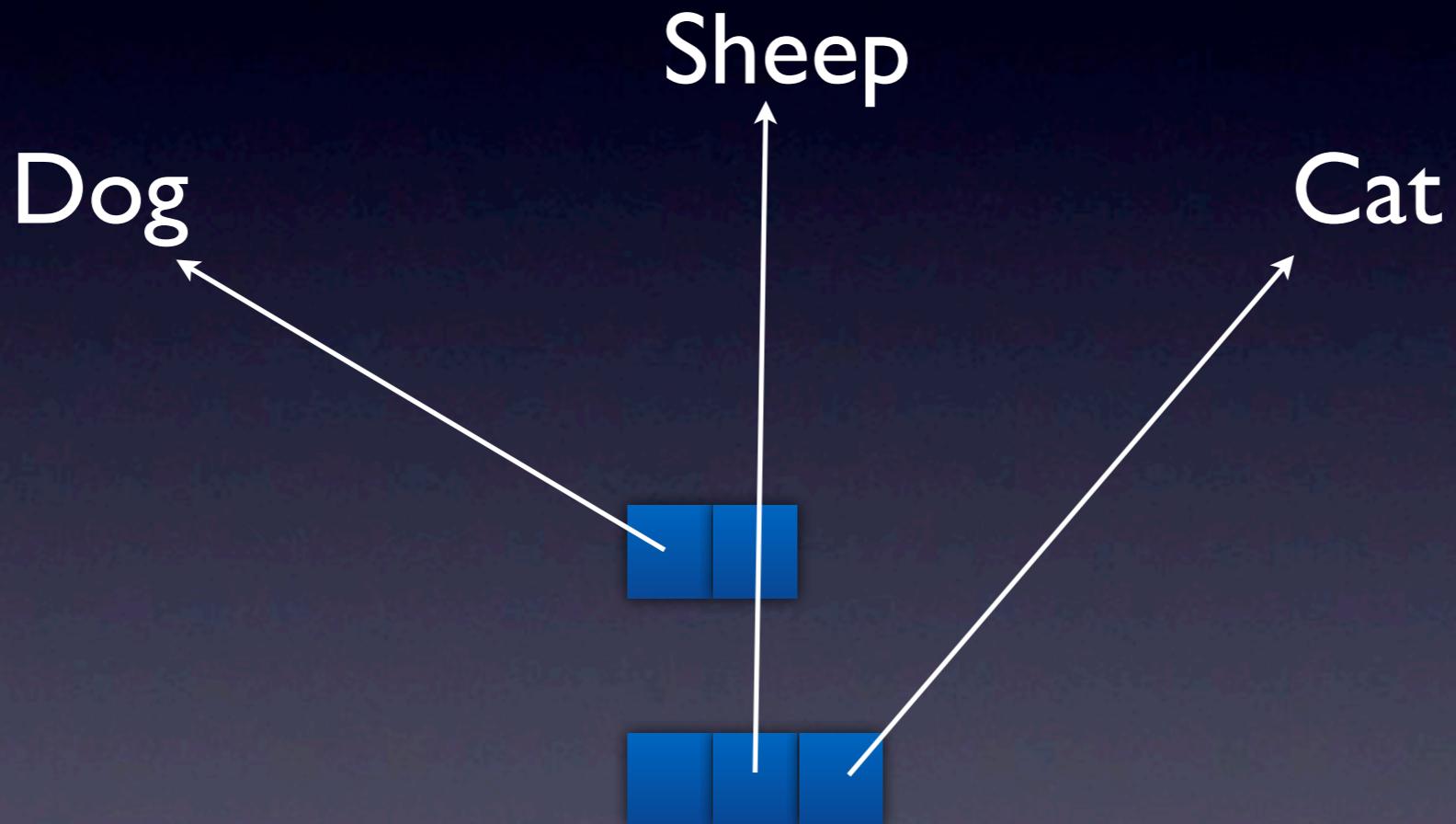
```
v.push_back(Ptr(new Cat));
```

Move Semantics



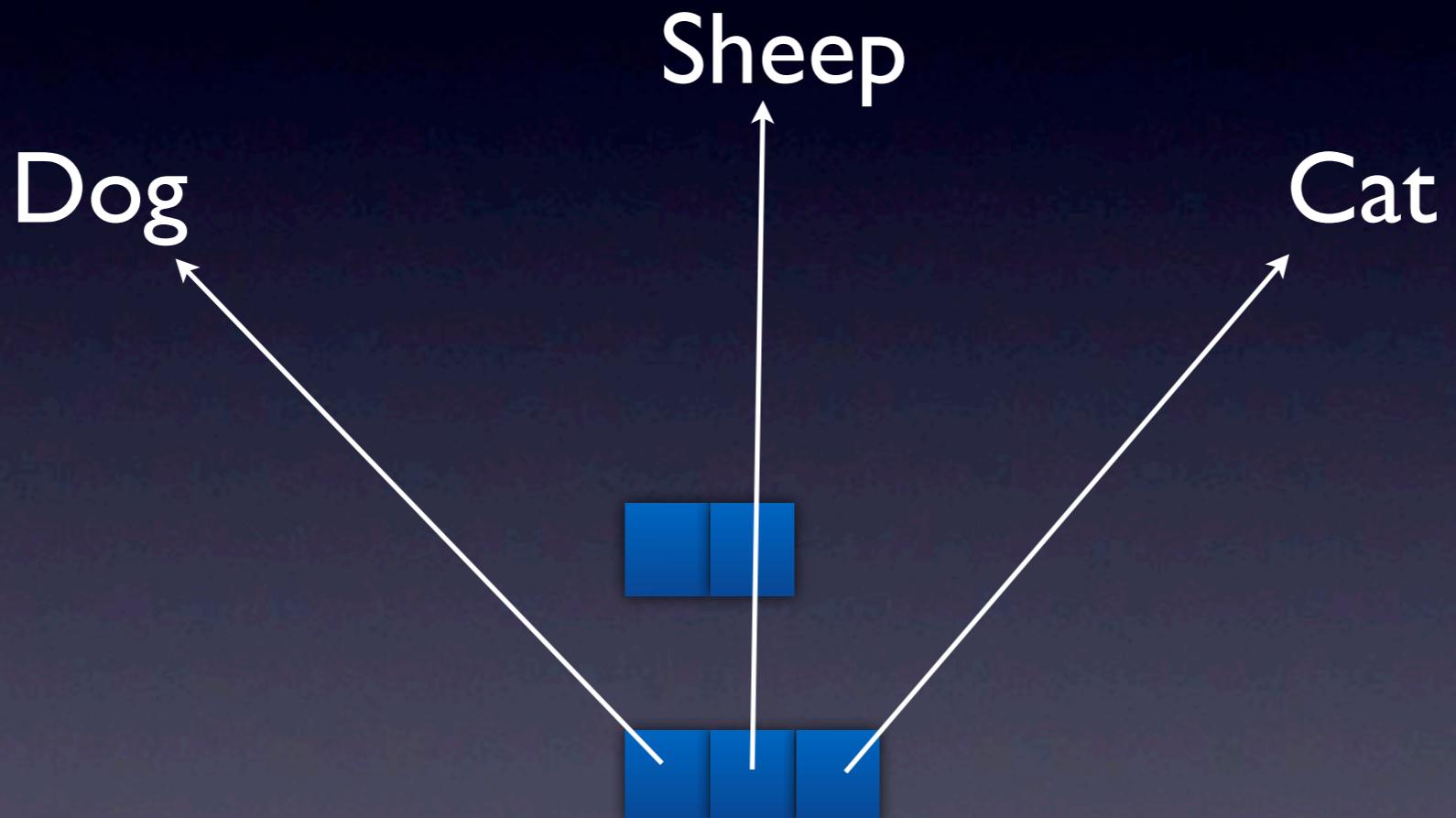
```
v.push_back(Ptr(new Cat));
```

Move Semantics



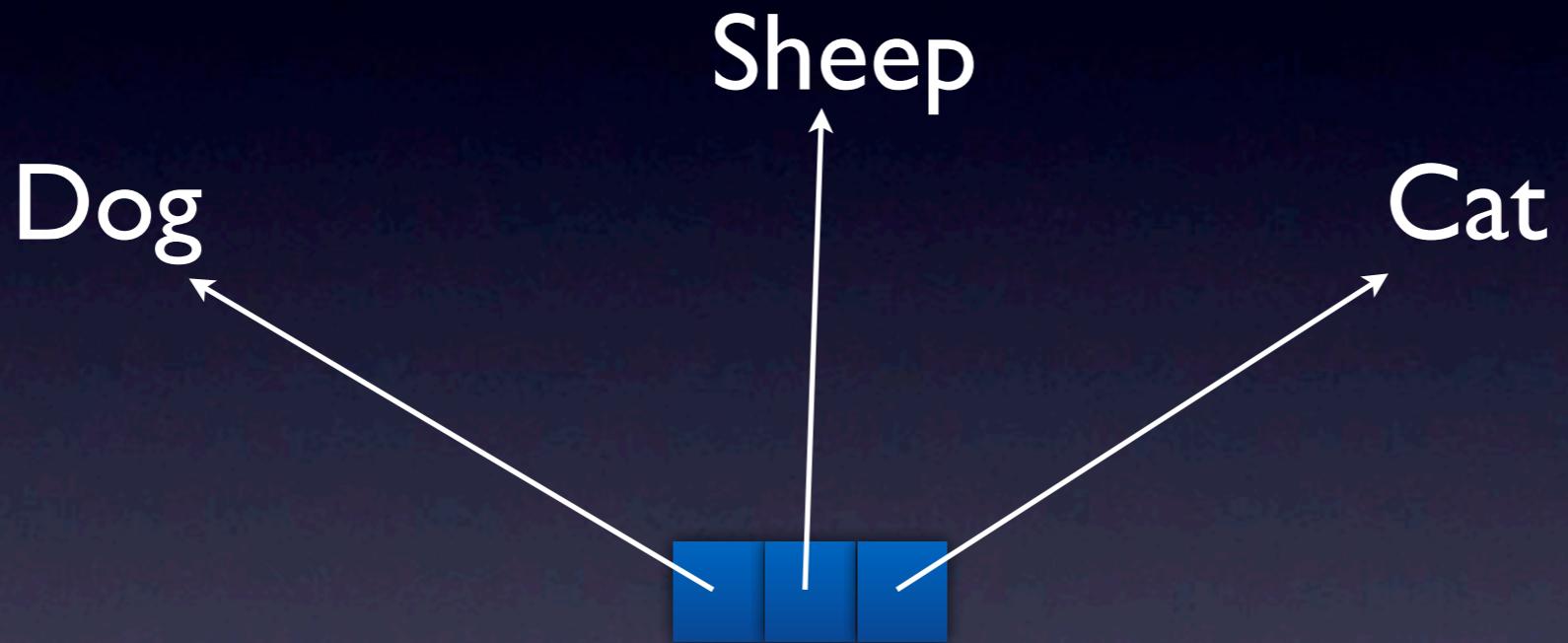
```
v.push_back(Ptr(new Cat));
```

Move Semantics



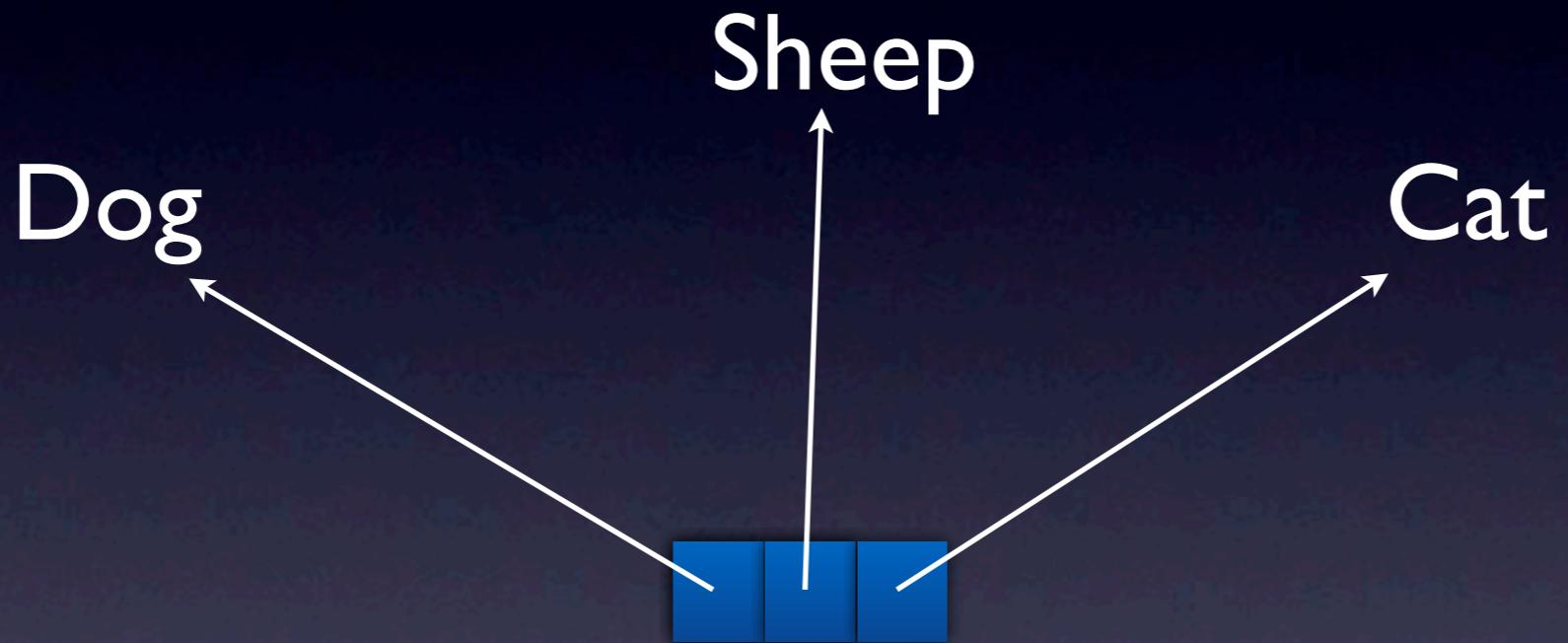
```
v.push_back(Ptr(new Cat));
```

Move Semantics



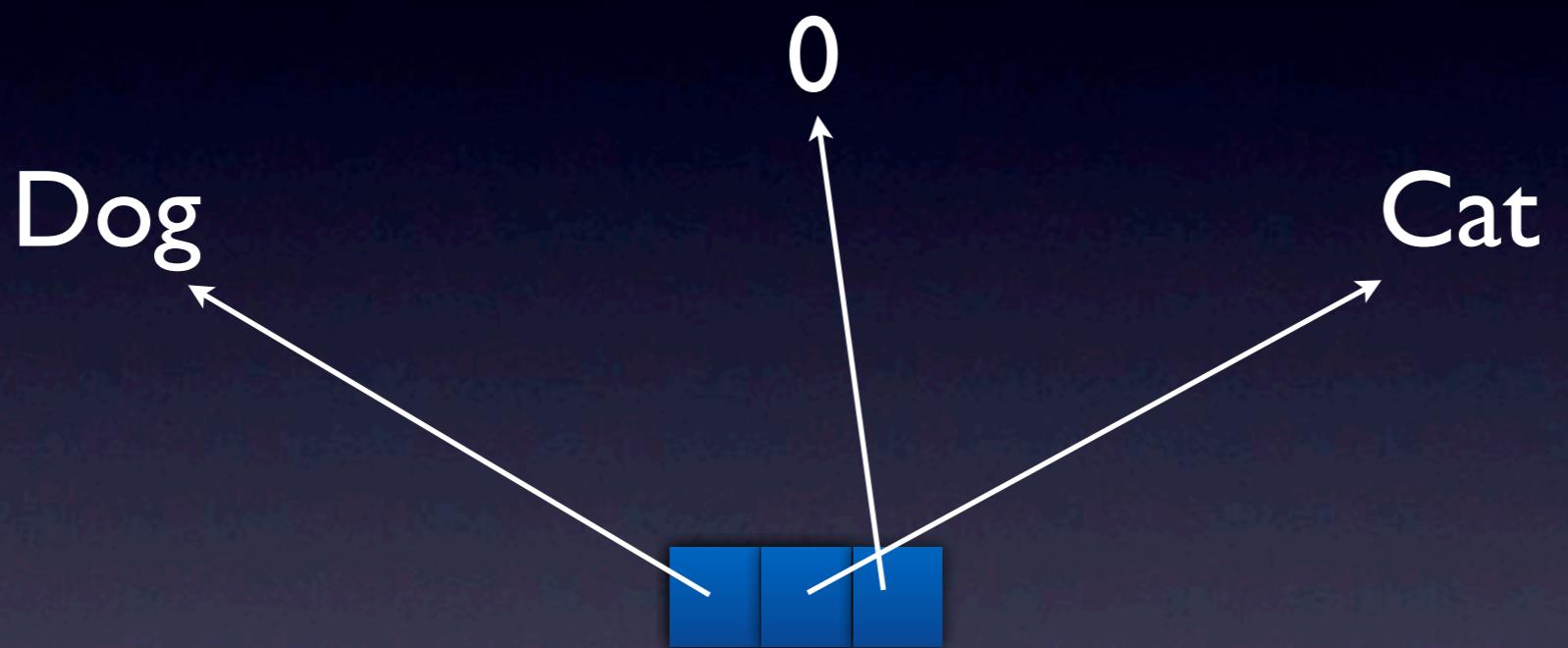
```
v.push_back(Ptr(new Cat));
```

Move Semantics



remove the sheep

Move Semantics



remove the sheep

Move Semantics



erase the sheep

Move Semantics

- Insertion members are now overloaded to move (not copy) rvalues into the allocator-aware containers:

```
void push_back(const value_type& v);  
void push_back(value_type&& v);
```

Move Semantics

- Insertion members are now overloaded to move (not copy) rvalues into the allocator-aware containers:

```
void push_front(const value_type& v);  
void push_front(value_type&& v);
```

Move Semantics

- Insertion members are now overloaded to move (not copy) rvalues into the allocator-aware containers:

```
iterator insert(const_iterator position,  
                const value_type& x);
```

```
iterator insert(const_iterator position,  
                value_type&& x);
```

Move Semantics

- Insertion members are now overloaded to move (not copy) rvalues into the allocator-aware containers:

```
mapped_type& operator[](const key_type& k);  
mapped_type& operator[](key_type&& k);
```

Move Semantics

- move-only `value_types` can also be moved out of *most* containers:

```
std::vector<A>::iterator i = v.begin();  
A a = std::move(*i);
```

Move Semantics

- move-only `value_types` can also be moved out of *most* containers:

```
std::vector<A>::iterator i = v.begin();
A a = std::move(*i);

A a = std::move(v.back());
```

Move Semantics

- move-only `value_types` can also be moved out of *most* containers:

```
std::vector<A>::iterator i = v.begin();
A a = std::move(*i);

A a = std::move(v.back());
A a = std::move(v[3]);
```

noexcept

- Many members of containers have been marked noexcept, including:
 - iterator factories

```
iterator      begin()      noexcept;
const_iterator begin() const noexcept;
iterator      end()        noexcept;
const_iterator end()      const noexcept;
```

noexcept

- Many members of containers have been marked noexcept, including:
 - Observers

```
size_type size() const noexcept;  
size_type max_size() const noexcept;  
bool empty() const noexcept;  
void clear() noexcept;
```

noexcept

- Many members of containers have not been marked noexcept, even though you might expect them to be:
- Special move members:
 - move constructor.
 - move assignment operator.

noexcept

- Many members of containers have not been marked noexcept, even though you might expect them to be:
- `erase`:

```
iterator erase(const_iterator p);  
iterator erase(const_iterator first,  
               const_iterator last);
```

noexcept

- Many members of containers have not been marked noexcept, even though you might expect them to be:
- swap:

```
void swap(C&);  
void swap(C& x, C& y);
```

noexcept

- Implementations are free to add noexcept (possibly conditional on template parameters) where appropriate.

```
void shrink_to_fit() noexcept;
```

noexcept

- Client code can inspect, at compile time, whether or not an expression is noexcept:

```
std::vector<int> c;
static_assert
(
    noexcept(c.shrink_to_fit()),
    "shrink_to_fit is not noexcept"
);
```

Emplace

- One can now “emplace” construct `value_types` into containers.

```
template <class ...Args>
iterator
emplace(const_iterator p,
        Args&& ...args);
```

Emplace

```
class A
{
public:
    A(const std::string& name, int n);
    A(const A&) = delete;
    A& operator=(const A&) = delete;
};

std::list<A> c;
c.emplace(c.begin(), "one", 1);
```

- Neither copy construction nor move construction required.

Emplace

- Sequences:

```
template <class ...Args>
iterator
emplace(const_iterator p, Args&& ...args);
```

```
template <class ...Args>
void
emplace_front(Args&& ...args);
```

```
template <class ...Args>
void
emplace_back(Args&& ...args);
```

Emplace

- forward_list:

```
template <class ...Args>
iterator
emplace_after(const_iterator p,
               Args&& ...args);
```

```
template <class ...Args>
void
emplace_front(Args&& ...args);
```

Emplace

- map / set / unordered_map / unordered_set

```
template <class... Args>
pair<iterator, bool>
emplace(Args&&... args);
```

```
template <class... Args>
iterator
emplace_hint(const_iterator p,
             Args&&... args);
```

Emplace

- `multimap / multiset / unordered_multimap / unordered_multiset`

```
template <class... Args>
iterator  
emplace(Args&&... args);
```

```
template <class... Args>
iterator  
emplace_hint(const_iterator p,  
             Args&&... args);
```

cbegin/cend

- There is now a way to assert that you want `const_iterators`, even if you have a non-`const` container.

cbegin/cend

- There is now a way to assert that you want `const_iterators`, even if you have a non-const container.

```
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
reverse_const_iterator crbegin() const noexcept;
reverse_const_iterator crend() const noexcept;
```

cbegin/cend

- There is now a way to assert that you want `const_iterators`, even if you have a non-`const` container.

cbegin/cend

- There is now a way to assert that you want `const_iterators`, even if you have a non-const container.

```
deque<int> s;
for (auto i = s.cbegin(); i != s.cend(); ++i)
    cout << *i << '\n';
```

iterator to const_iterator

- insert and erase members used to specify position using iterator.
- They now specify position using const_iterator.

iterator to const_iterator

- insert and erase members used to specify position using iterator.
- They now specify position using const_iterator.

```
iterator  
insert(iterator p,  
       const value_type& v);
```

```
iterator  
erase(iterator p);
```

```
iterator  
erase(iterator first,  
      iterator last);
```

iterator to const_iterator

- insert and erase members used to specify position using iterator.
- They now specify position using const_iterator.

iterator to const_iterator

- insert and erase members used to specify position using iterator.
- They now specify position using const_iterator.

iterator

```
insert(const_iterator p,  
       const value_type& v);
```

iterator

```
erase(const_iterator p);
```

iterator

```
erase(const_iterator first,  
      const_iterator last);
```

iterator to const_iterator

- This can cause problems for multimap, map, unordered_map and unordered_multimap:

```
iterator  
erase(const_iterator p);
```

iterator to const_iterator

- This can cause problems for multimap, map, unordered_map and unordered_multimap:

```
iterator  
erase(const_iterator p);
```

```
iterator  
erase(const key_type& key);
```

iterator to const_iterator

- This can cause problems for multimap, map, unordered_map and unordered_multimap:

```
struct A
{
    template <class T> A(T);
};
```

```
map<A, int> m;
map<A, int>::iterator i = ...
m.erase(i); // ambiguous: which erase?
```

iterator to const_iterator

- This can cause problems for multimap, map, unordered_map and unordered_multimap:
- Must explicitly turn *i* into a const_iterator.

```
map<A, int> m;  
map<A, int>::iterator i = ...  
m.erase(map<A, int>::const_iterator(i));
```

iterator to const_iterator

- Given a non-const container, you can now convert a `const_iterator` to iterator in constant time:

iterator to const_iterator

- Given a non-const container, you can now convert a `const_iterator` to iterator in constant time:

```
list<int> l;  
list<int>::const_iterator ci = ...  
  
// convert const_iterator ci to iterator
```

iterator to const_iterator

- Given a non-const container, you can now convert a `const_iterator` to iterator in constant time:

```
list<int> l;
list<int>::const_iterator ci = ...  
  
// convert const_iterator ci to iterator
list<int>::iterator i = l.erase(ci, ci);
```

initializer_list

initializer_list

- One can now construct a container with an initializer_list:

```
vector<int> v = {1, 2, 3};
```

initializer_list

initializer_list

- One can now assign to a container with an initializer_list:

```
vector<int> v;  
v = {1, 2, 3};
```

initializer_list

- One can now assign to a container with an initializer_list:

```
vector<int> v;  
v = {1, 2, 3};  
  
v.assign({1, 2, 3});
```

initializer_list

initializer_list

- One can now insert into a container with an initializer_list:

```
vector<int> v;  
v.insert(v.cbegin(), {1, 2, 3});
```

Construct N values

```
vector<A> v(10);
```

Construct N values

```
vector<A> v(10);
```

- In C++03 this default constructs A once, and then inserts 10 copies of the default constructed A.
- Requires A to be copy constructible.

Construct N values

```
vector<A> v(10);
```

- In C++03 this default constructs A once, and then inserts 10 copies of the default constructed A.
 - Requires A to be copy constructible.
- In C++11 this default constructs A 10 times.
 - A need not be copy constructible or even move constructible.

resize N values

```
vector<A> v;  
v.resize(10);
```

resize N values

```
vector<A> v;  
v.resize(10);
```

- In C++03 this default constructs A once, and then appends copies of the default constructed A as necessary to make `size() == 10`.
- Requires A to be copy constructible.

resize N values

```
vector<A> v;  
v.resize(10);
```

- In C++03 this default constructs A once, and then appends copies of the default constructed A as necessary to make `size() == 10`.
- Requires A to be copy constructible.
- In C++11 this default constructs A as necessary to make `size() == 10`.
- A need not be copy constructible.

Allocators

- Allocators with state are now fully supported.
- Allocators may or may not be “equal” to one another.
- Allocator::pointer no longer needs to be a built-in pointer.

A Minimal Allocator

```
template <class T>
class MyAlloc {
public:
    typedef T value_type;

    MyAlloc();
    template <class U>
        MyAlloc(const MyAlloc<U>&);

    T* allocate(std::size_t);
    void deallocate(T*, std::size_t);
};
```

- Much of the C++03 boilerplate is now defaulted

A Minimal Allocator

```
template <class T, class U>
bool
operator==(const MyAlloc<T>&,
            const MyAlloc<U>&);
```

```
template <class T, class U>
bool
operator!=(const MyAlloc<T>&,
            const MyAlloc<U>&);
```

- Much of the C++03 boilerplate is now defaulted

A Minimal Allocator

- If two allocators compare equal, they must be able to deallocate each other's allocated memory.
- A copy constructed allocator must compare equal to its original.
- A move constructed allocator must compare equal to the prior value of the original.

A Minimal Allocator

- You do not need to supply other nested types:
 - `pointer`
 - `const_pointer`
 - `size_type`
 - `difference_type`
 - `rebind<U>::other`
- Much of the C++03 boilerplate is now defaulted

A Minimal Allocator

- You do not need to supply other member functions:
 - `construct`
 - `destroy`
- But you can if you want to override the defaults.
- Much of the C++03 boilerplate is now defaulted

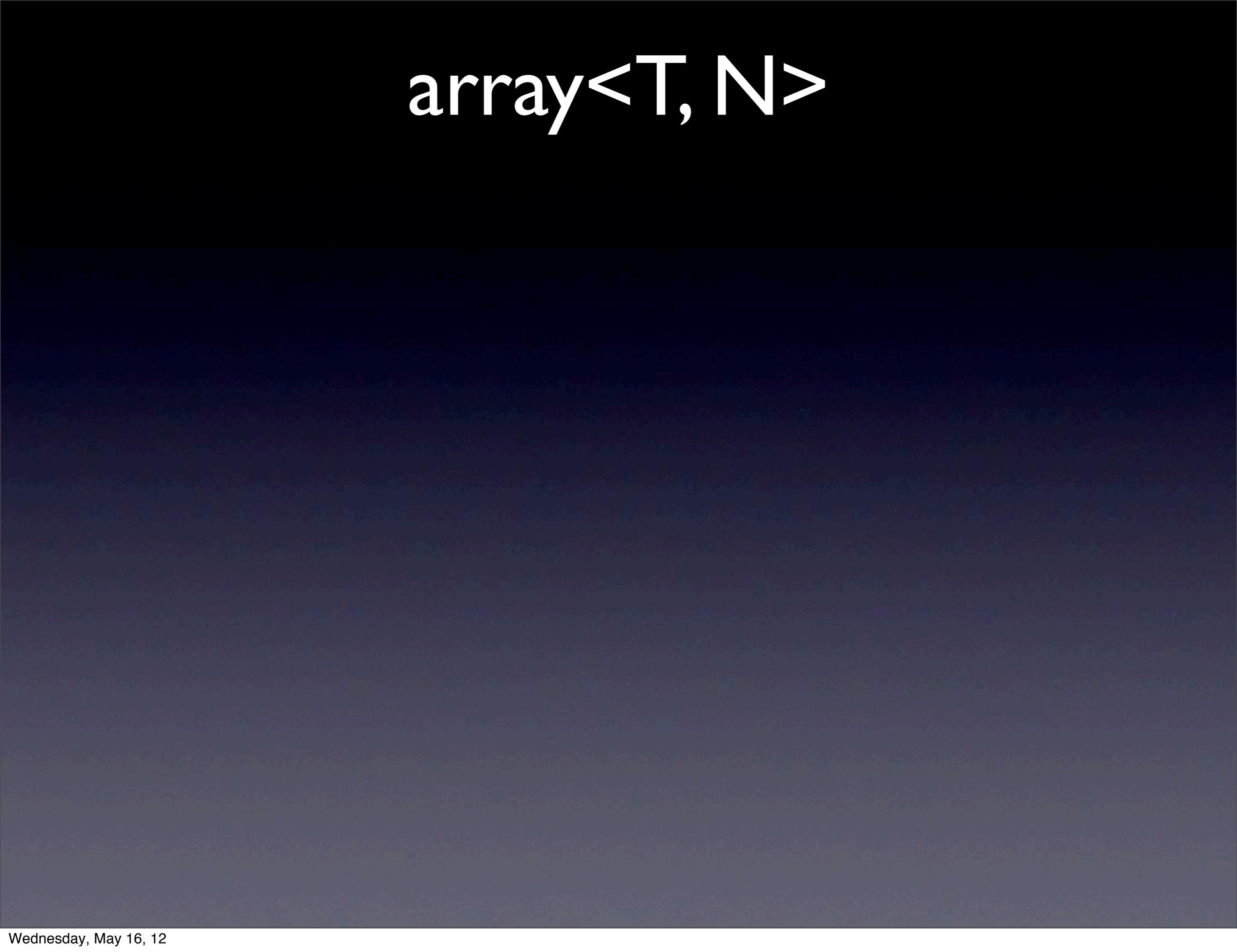
Outline

- Changes & additions to existing containers
- New containers.

Outline

- Changes & additions to existing containers
- New containers.

array<T, N>



array<T, N>

- <array>

array<T, N>

- <array>
- The alternative to a built-in array.

array<T, N>

- <array>
- The alternative to a built-in array.
- Compile time size.

array<T, N>

- <array>
- The alternative to a built-in array.
- Compile time size.
- Not heap allocated.

array<T, N>

- Advantages over a built-in array:

array<T, N>

- Advantages over a built-in array:
 - You can return it from a function.

```
int [3]
make( )
{
    int a[3] = {1, 2, 3};
    return a;
}
```

array<T, N>

- Advantages over a built-in array:
 - You can return it from a function.

```
std::array<int, 3>
make()
{
    std::array<int, 3> a = {1, 2, 3};
    return a;
}
```

array<T, N>

- Advantages over a built-in array:

array<T, N>

- Advantages over a built-in array:
 - It will not implicitly decay to a pointer.

```
template <class T>
void process(T t)
{
    // T is int*
}

int a[3] = {1, 2, 3};
process(a);
```

array<T, N>

- Advantages over a built-in array:
 - It will not implicitly decay to a pointer.

```
template <class T>
void process(T t)
{
    // T is array<int, 3>
}

array<int, 3> a = {1, 2, 3};
process(a);
```

array<T, N>

- array<T, N> is:
 - Swappable
 - Equality Comparable
 - Less-than Comparable

array<T, N>

- array<T, N> is an aggregate
 - (no user-declared constructors)

array<T, N>

- array<T, N> is an aggregate
 - (no user-declared constructors)

```
array<int, 3> a1 = {1, 2, 3};
```

array<T, N>

- Has iterator support.

array<T, N>

- Has iterator support.

```
iterator      begin()      noexcept;
const_iterator begin() const noexcept;
iterator      end()        noexcept;
const_iterator end()  const noexcept;
```

array<T, N>

- Has iterator support.

```
reverse_iterator           rbegin()           noexcept;
reverse_const_iterator    rbegin() const    noexcept;
reverse_iterator           rend()           noexcept;
reverse_const_iterator    rend() const    noexcept;
```

array<T, N>

- Has iterator support.

```
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
reverse_const_iterator crbegin() const noexcept;
reverse_const_iterator crend() const noexcept;
```

array<T, N>

- Has access support.

array<T, N>

- Has access support.

```
reference operator[](size_type n);  
const_reference operator[](size_type n) const;
```

```
reference at(size_type n);  
const_reference at(size_type n) const;
```

```
reference front();  
const_reference front() const;
```

```
reference back();  
const_reference back() const;
```

array<T, N>

- Misc:

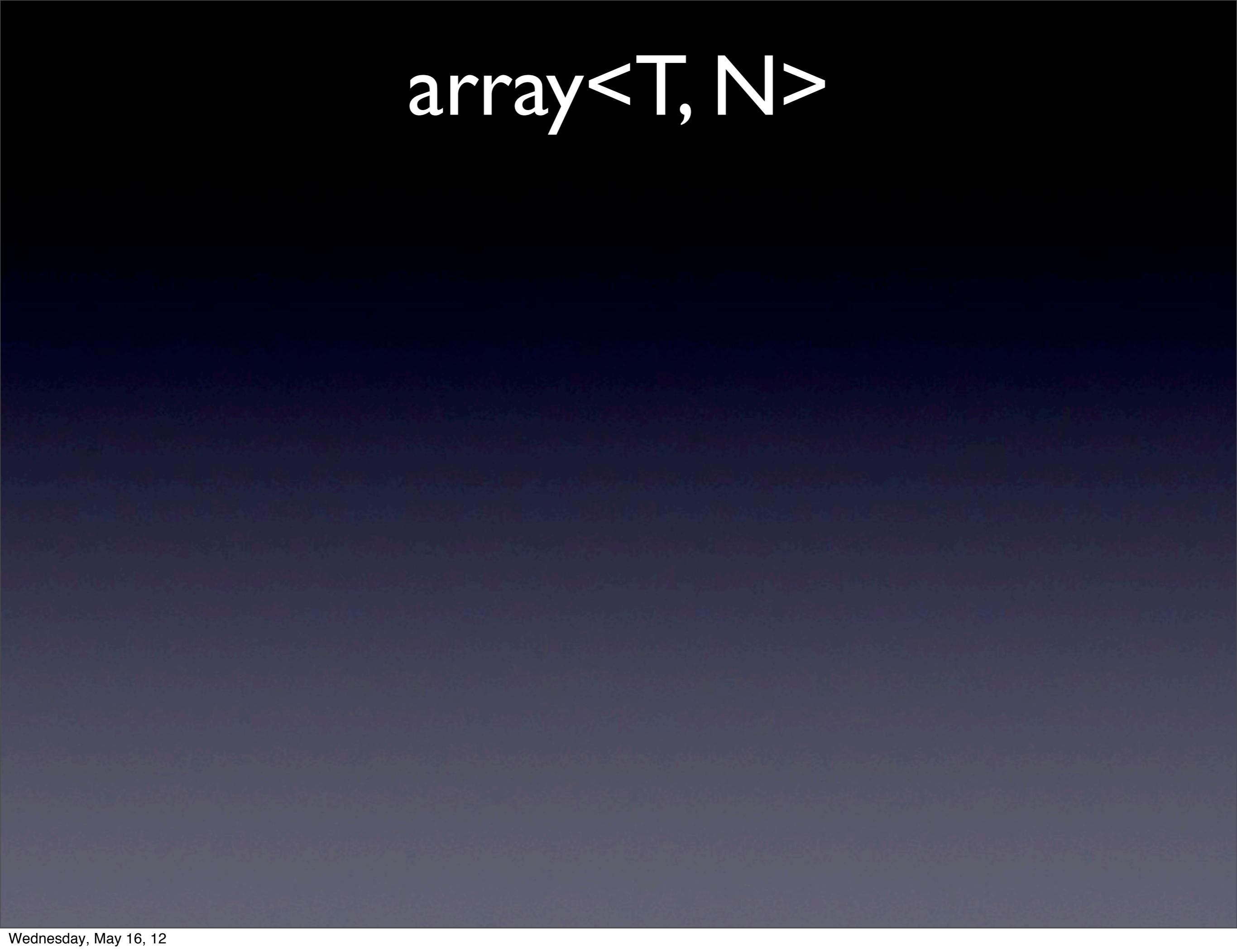
array<T, N>

- Misc:

```
void fill(const T& t);
constexpr size_type size() noexcept;
constexpr bool empty() noexcept;

    T* data() noexcept;
const T* data() const noexcept;
```

array<T, N>



array<T, N>

- `array<T, 0>` is not a compile-time error.
 - But don't try to access any elements.

```
template <class T, size_t N>
void display(const array<T, N>& a)
{
    cout << "{";
    if (!a.empty()) // Ok for N == 0
    {
        cout << a[0];
        for (size_t i = 1; i < N; ++i)
            cout << ", " << a[i];
    }
    cout << "}\n";
}
```

array<T, N>

- array<T, N> has a tuple interface:

array<T, N>

- `array<T, N>` has a tuple interface:

```
array<int, 3> a = {4, 5, 6};
```

array<T, N>

- `array<T, N>` has a tuple interface:

```
array<int, 3> a = {4, 5, 6};
```

```
get<1>(a) == a[1]
```

array<T, N>

- `array<T, N>` has a tuple interface:

```
array<int, 3> a = {4, 5, 6};
```

```
get<1>(a) == a[1]
```

```
tuple_size<array<int, 3>>::value == 3
```

array<T, N>

- `array<T, N>` has a tuple interface:

```
array<int, 3> a = {4, 5, 6};
```

```
get<1>(a) == a[1]
```

```
tuple_size<array<int, 3>>::value == 3
```

```
tuple_element<1, array<int, 3>>::type is int
```

forward_list<T,A>

Wednesday, May 16, 12

`forward_list<T,A>`

- `<forward_list>`

`forward_list<T,A>`

- `<forward_list>`
- A singly linked list.

`forward_list<T,A>`

- `<forward_list>`
- A singly linked list.
- Evolved from the SGI slist.

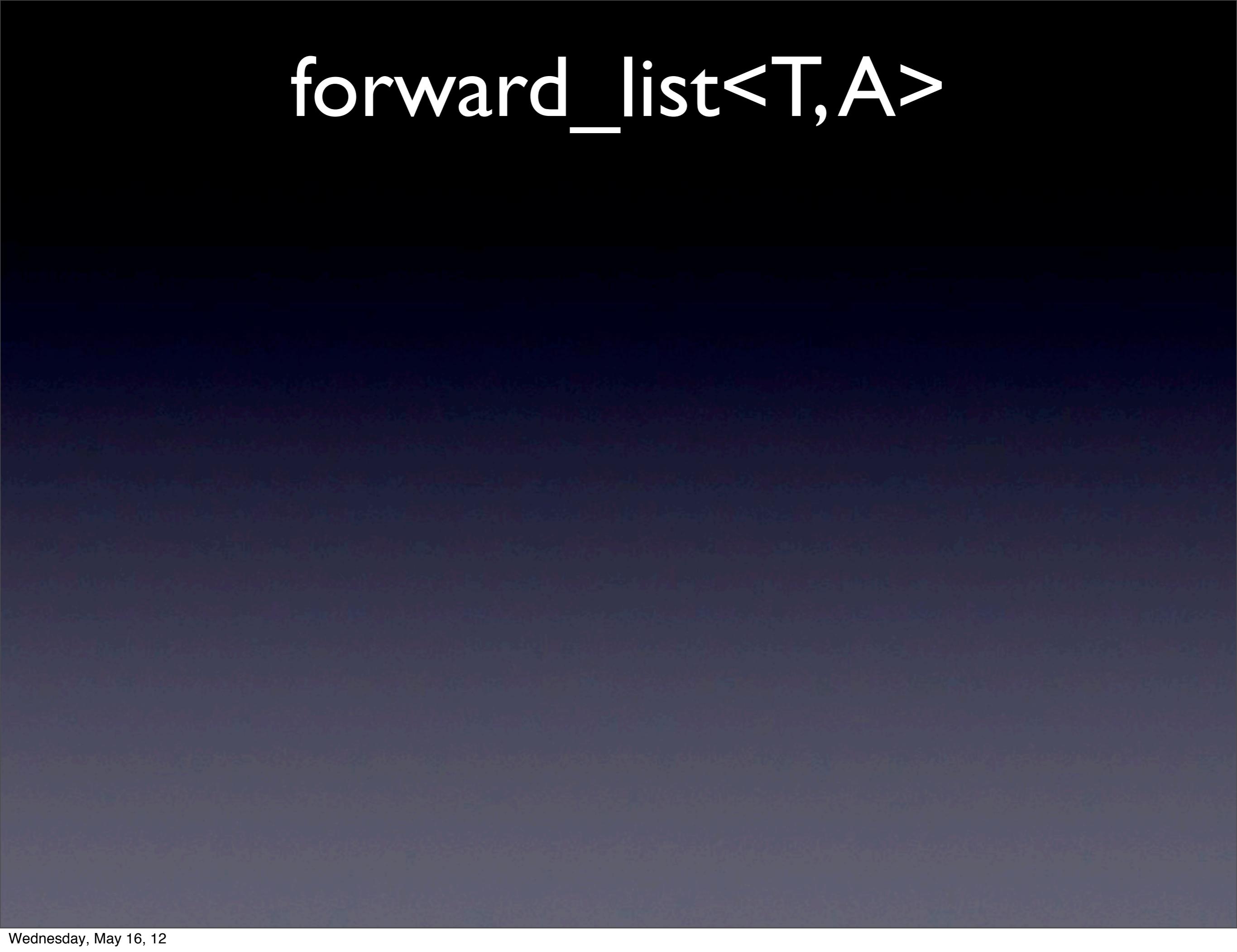
`forward_list<T,A>`

- `<forward_list>`
- A singly linked list.
- Evolved from the SGI slist.
- Reason to exist: A space optimization of (doubly linked) `std::list`.

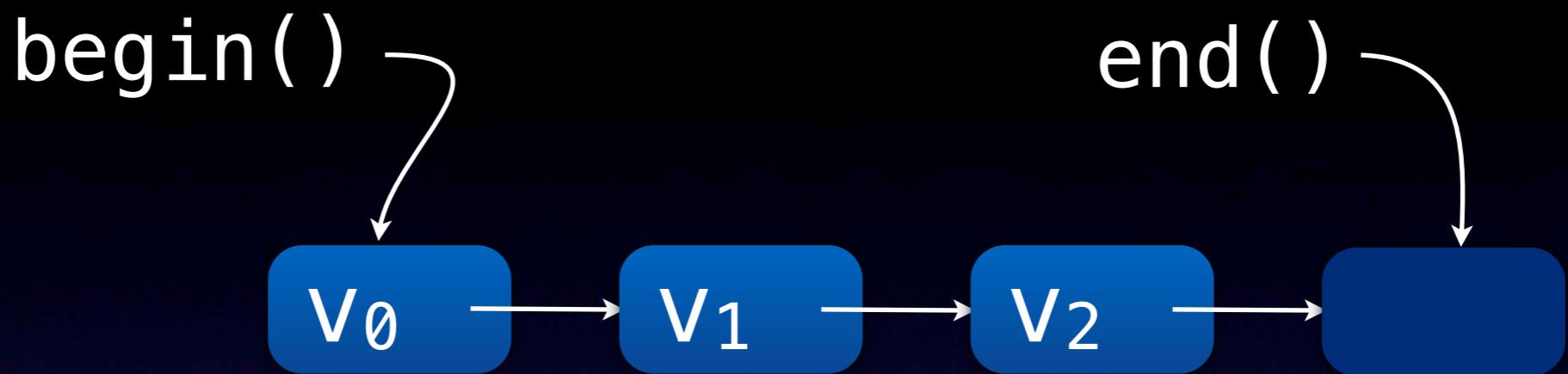
`forward_list<T,A>`

- `<forward_list>`
- A singly linked list.
- Evolved from the SGI slist.
- Reason to exist: A space optimization of (doubly linked) `std::list`.
- Just like list, except ...

forward_list<T,A>

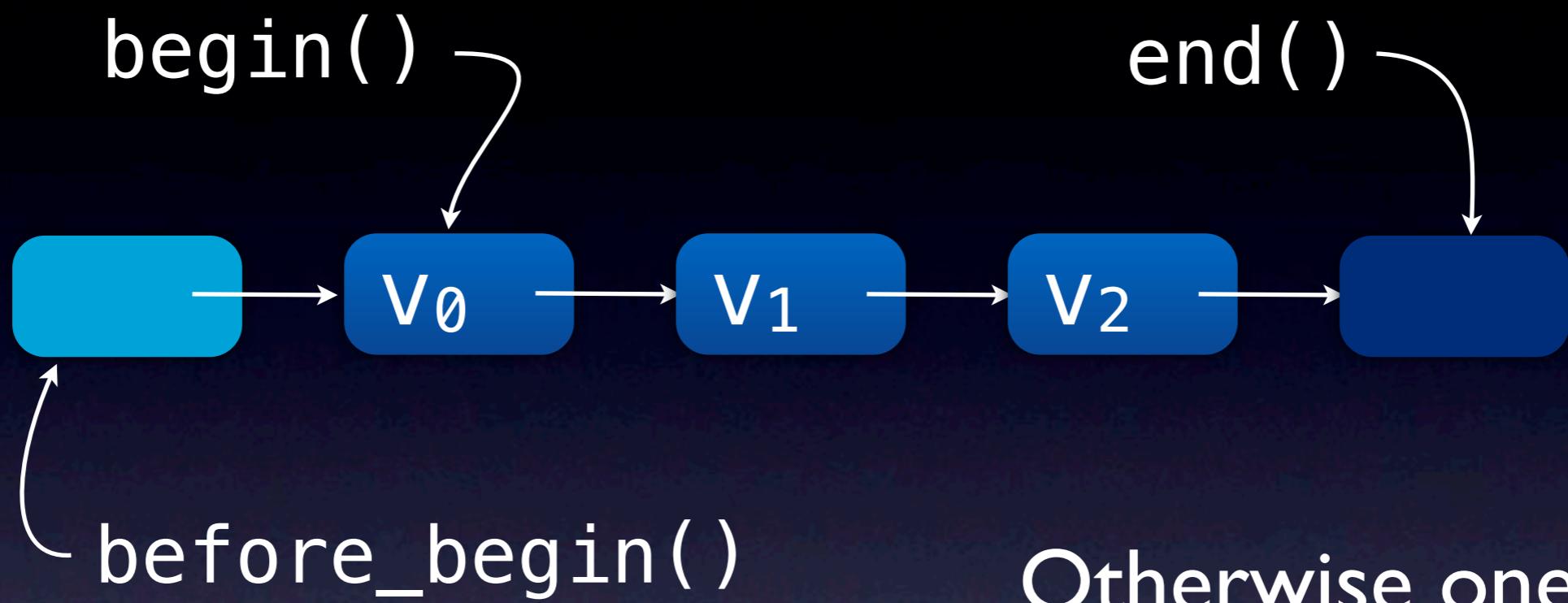


forward_list<T,A>



- Not like other std:: sequences.
- To insert or erase you have to refer to the prior position instead of the position after.

forward_list<T,A>



Otherwise one could
not “`push_front()`”

- Not like other `std::` sequences.
- To insert or erase you have to refer to the prior position instead of the position after.
- A “pseudo-node” exists prior to the first element.

forward_list<T,A>



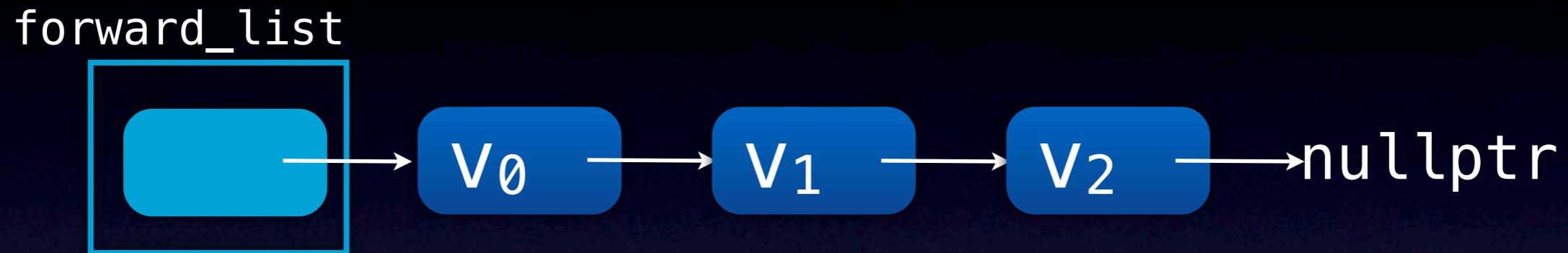
forward_list<T,A>

forward_list



- Typically:
 - `sizeof(forward_list<T>) == sizeof(T*)`

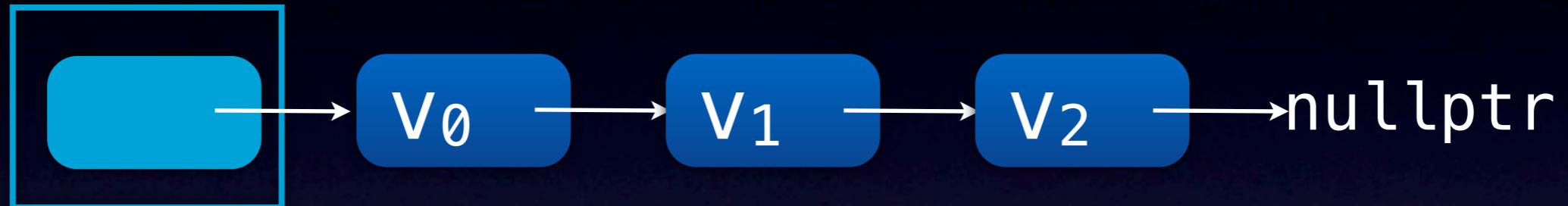
forward_list<T,A>



- Typically:
 - `sizeof(forward_list<T>) == sizeof(T*)`
 - `end() == nullptr`

forward_list<T,A>

forward_list



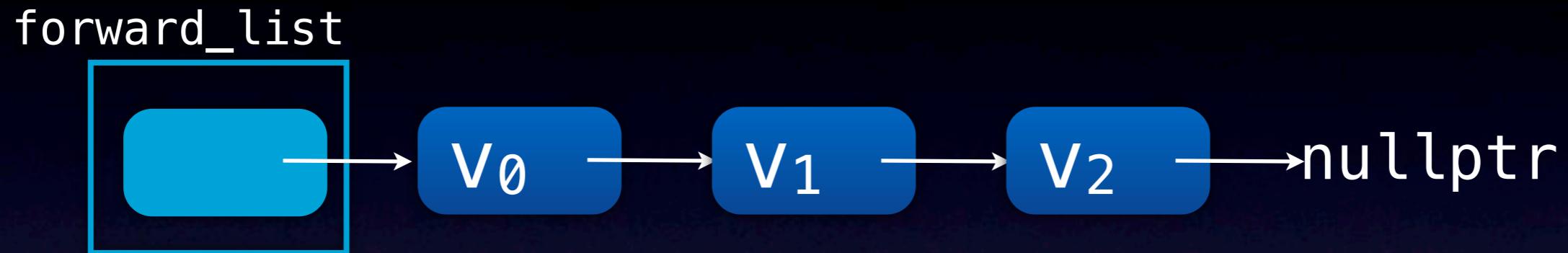
forward_list<T,A>

forward_list



```
iterator begin() noexcept;  
const_iterator begin() const noexcept;  
iterator end() noexcept;  
const_iterator end() const noexcept;
```

forward_list<T,A>



```
const_iterator cbegin() const noexcept;  
const_iterator cend() const noexcept;
```

forward_list<T,A>

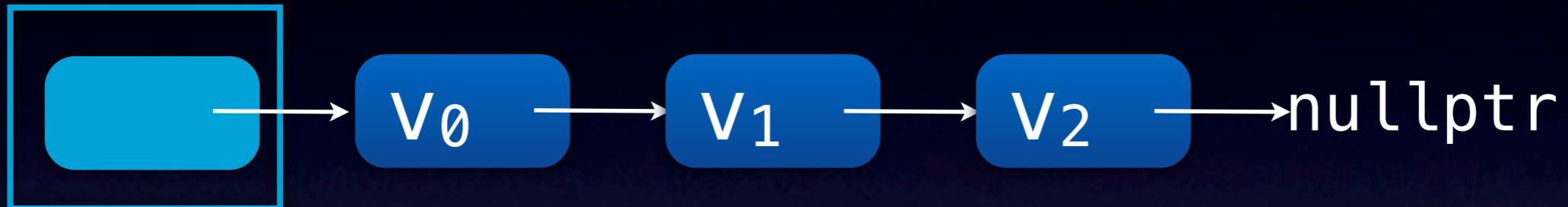
forward_list



```
iterator before_begin()           noexcept;  
const_iterator before_begin()    const noexcept;  
const_iterator cbefore_begin()   const noexcept;
```

forward_list<T,A>

forward_list



forward_list<T,A>

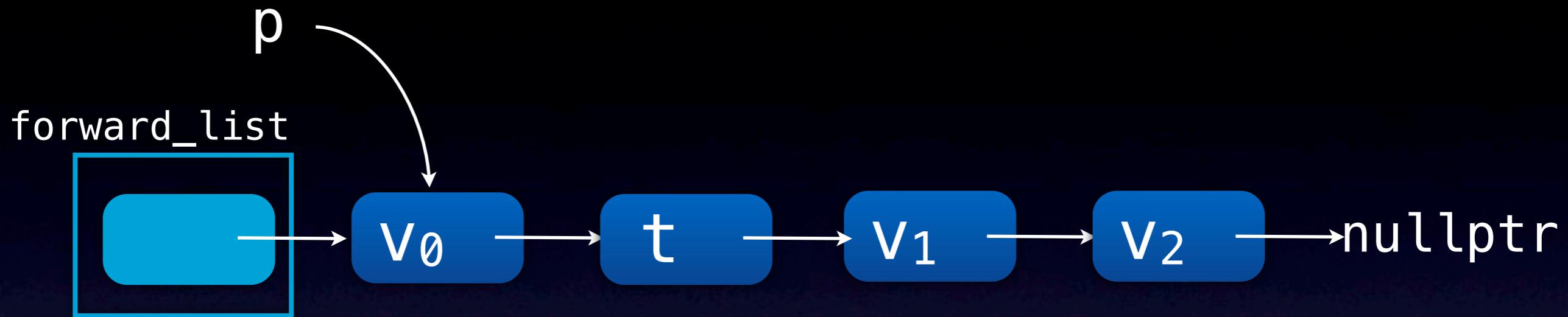
forward_list



iterator

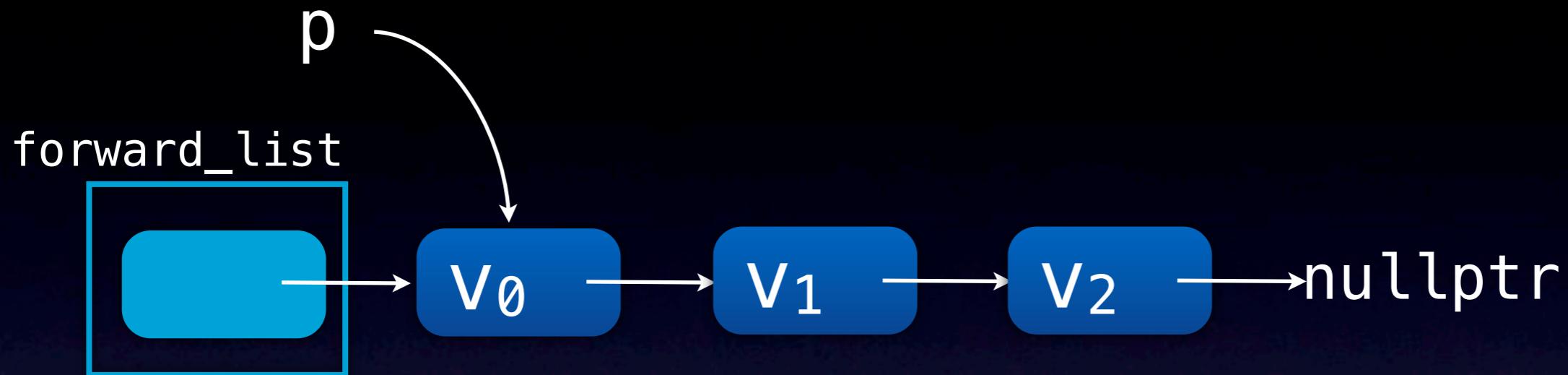
```
insert_after(const_iterator p,  
            const T& t);
```

forward_list<T,A>



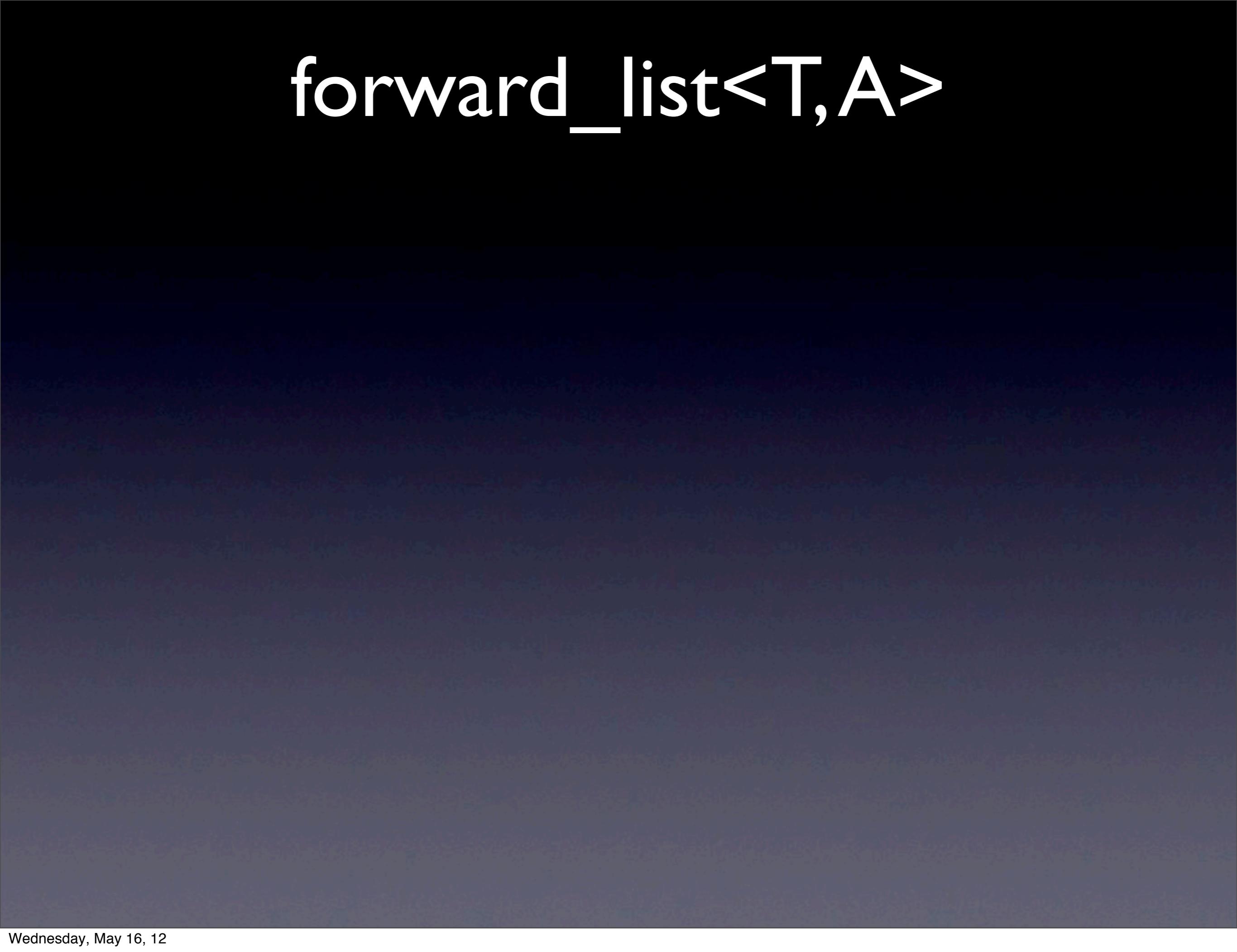
iterator
insert_after(const_iterator p,
const T& t);

forward_list<T,A>



iterator
`erase_after(const_iterator p);`

forward_list<T,A>



`forward_list<T,A>`

- There is no insert or erase.
- These would have to be $O(N)$.

`forward_list<T,A>`

- There is no `insert` or `erase`.
- These would have to be $O(N)$.
- Only `insert_after` and `erase_after`.

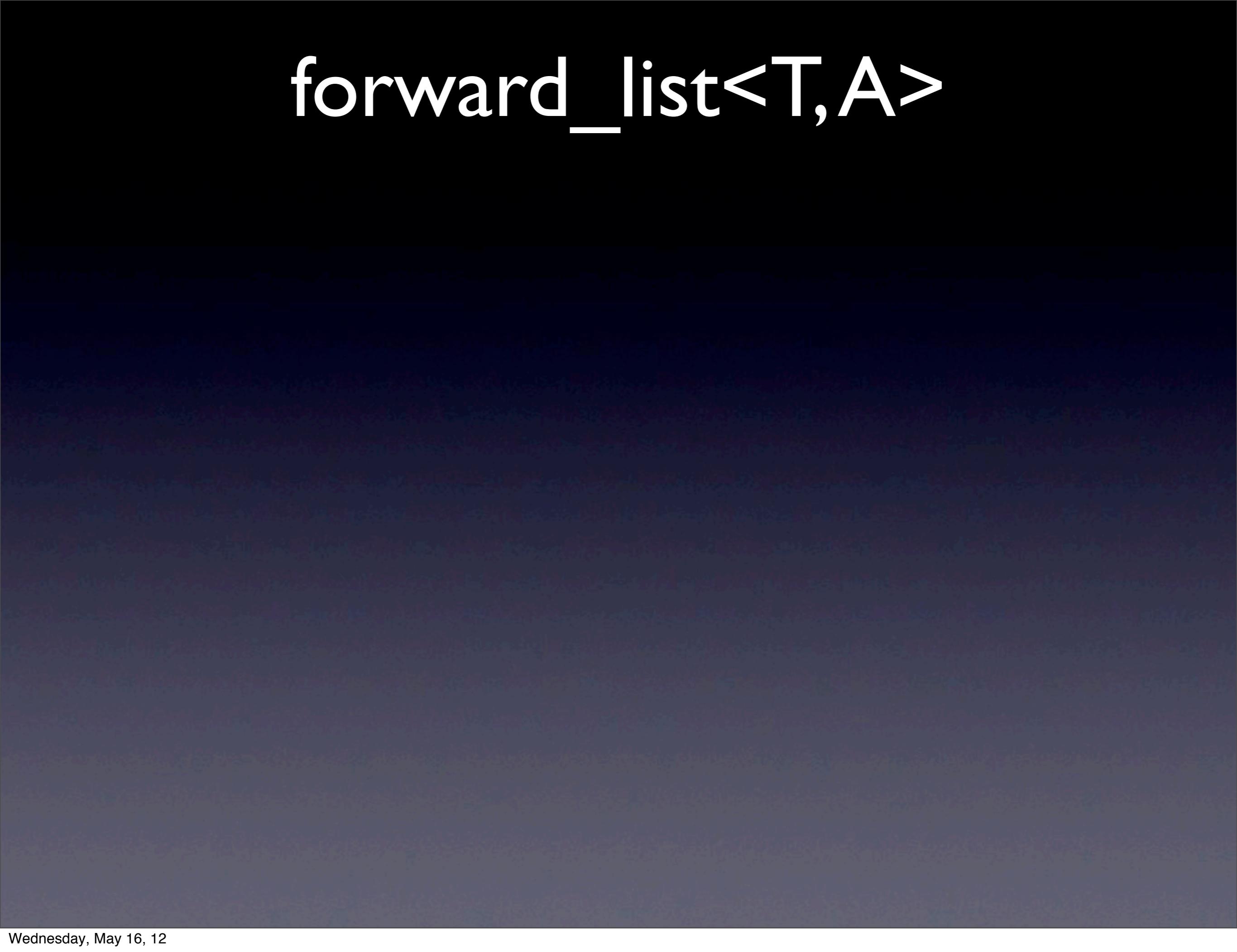
`forward_list<T,A>`

- There is no `insert` or `erase`.
- These would have to be $O(N)$.
- Only `insert_after` and `erase_after`.
- You can `splice_after`, not `splice`.

`forward_list<T,A>`

- There is no `insert` or `erase`.
- These would have to be $O(N)$.
- Only `insert_after` and `erase_after`.
- You can `splice_after`, not `splice`.
- You can `emplace_after`, not `emplace`.

forward_list<T,A>



`forward_list<T,A>`

- There is no `size()`.

`forward_list<T,A>`

- There is no `size()`.
- Use `distance(begin(), end())` if desired.

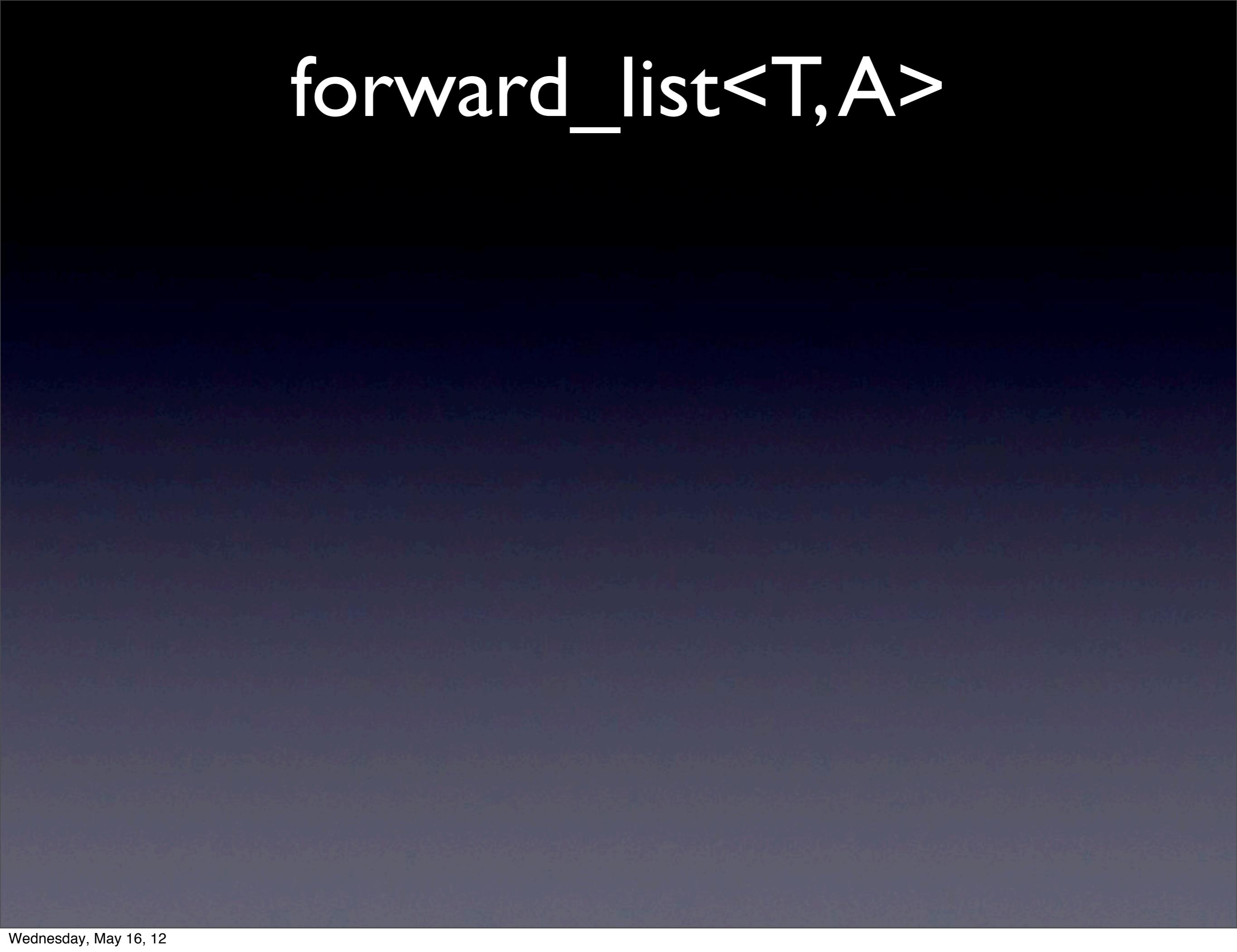
`forward_list<T,A>`

- There is no `size()`.
 - Use `distance(begin(), end())` if desired.
 - Recall: this is a space optimization!

`forward_list<T,A>`

- There is no `size()`.
 - Use `distance(begin(), end())` if desired.
 - Recall: this is a space optimization!
 - $O(N)$ `size()` is a leading cause of performance bugs.

forward_list<T,A>



`forward_list<T,A>`

- There is `push_front`, `pop_front` and `emplace_front`.

`forward_list<T,A>`

- There is `push_front`, `pop_front` and `emplace_front`.
- No `push_back`, `pop_back` or `emplace_back`.

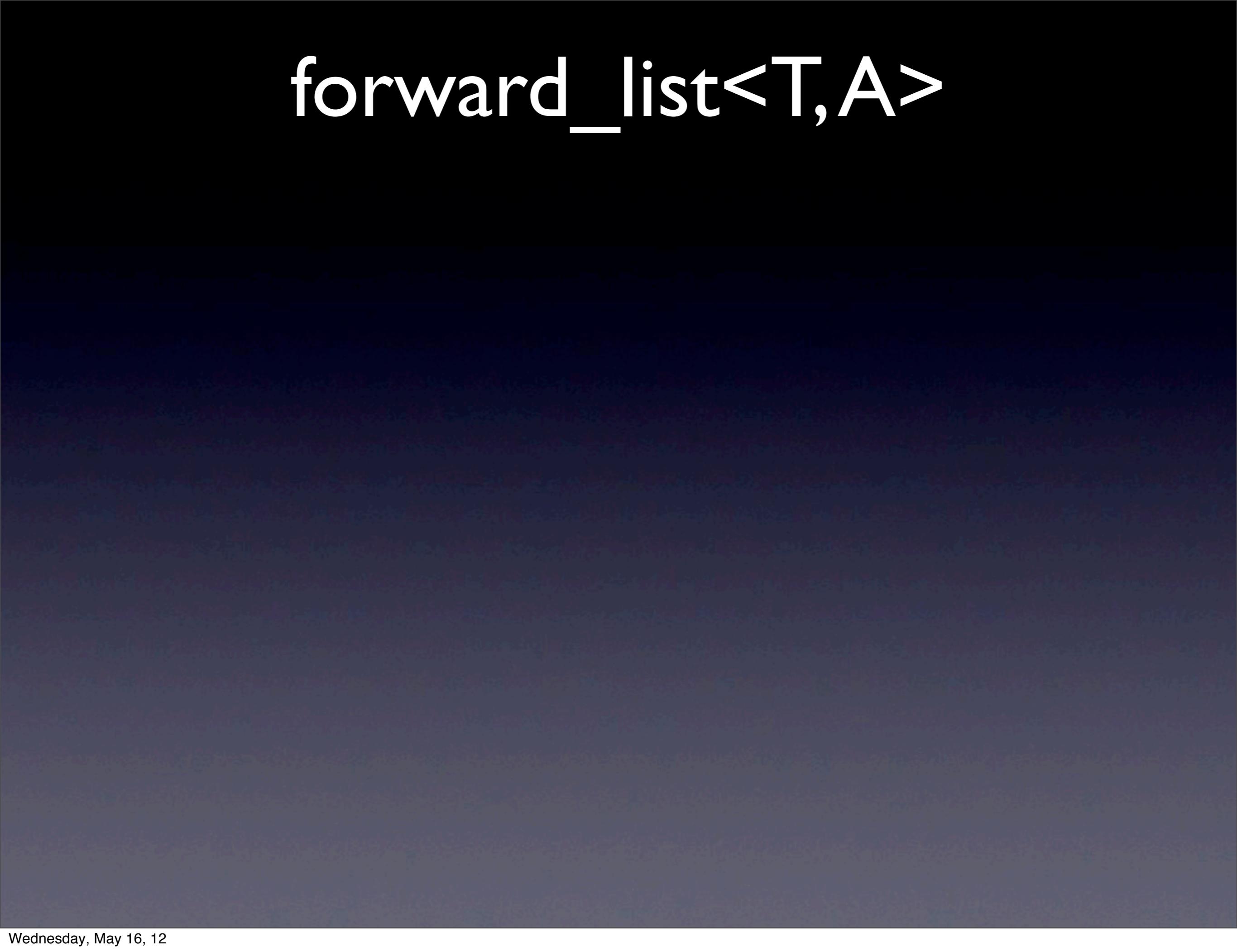
`forward_list<T,A>`

- There is `push_front`, `pop_front` and `emplace_front`.
- No `push_back`, `pop_back` or `emplace_back`.
- There is `front()`.

`forward_list<T,A>`

- There is `push_front`, `pop_front` and `emplace_front`.
- No `push_back`, `pop_back` or `emplace_back`.
- There is `front()`.
- No `back()`.

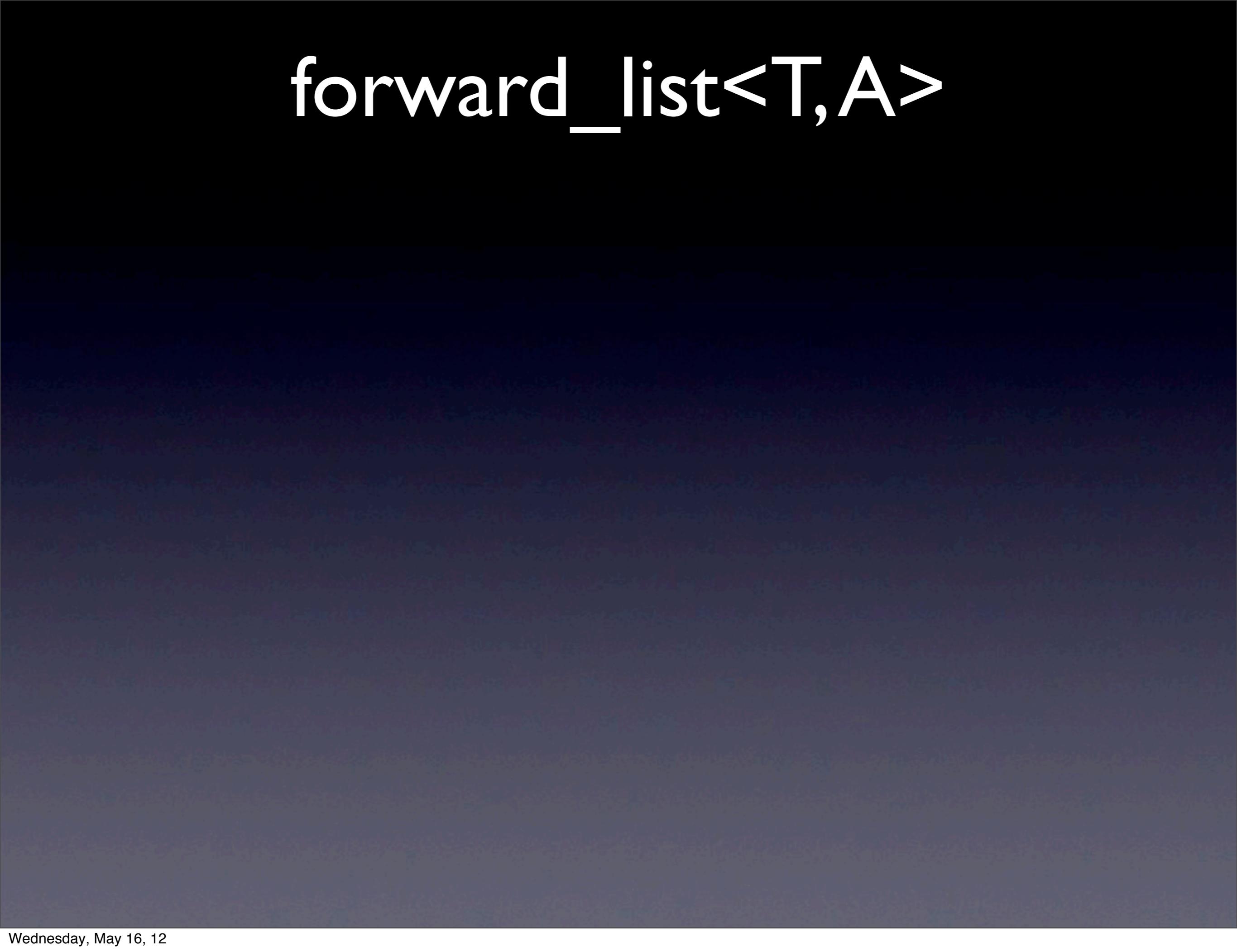
forward_list<T,A>



`forward_list<T,A>`

- Iterators are forward only, not bidirectional.

forward_list<T,A>



`forward_list<T,A>`

- But otherwise `forward_list` is just like `list`.

unordered containers

unordered containers

- <unordered_map>
 - `unordered_map<Key, T, Hash, Pred>`,
`unordered_multimap<Key, T, Hash, Pred>`
- <unordered_set>
 - `unordered_set<Key, Hash, Pred>`,
`unordered_multiset<Key, Hash, Pred>`
- These are similar to (multi)map/set, but are hash containers instead of binary tree containers.

unordered containers

unordered containers

- Reason to exist:
 - Performance optimization over (multi)map/set.

unordered containers

- Reason to exist:
 - Performance optimization over (multi)map/set.
 - But higher performance is not guaranteed.

unordered containers

unordered containers

```
#include <string>
#include <map>

int main()
{
    typedef std::map<int, std::string> Map;
    Map m;
    m[1] = "one";
    m[2] = "two";
    m[3] = "three";
    //...
    Map::iterator i = m.find(2);
    if (i != m.end())
        i->second[0] = 'T';
    else
        i = m.insert(i, std::make_pair(2, "Two"));
}
```

unordered containers

```
#include <string>
#include <unordered_map> ← Change header

int main()
{
    typedef std::unordered_map<int, std::string> Map;
    Map m;
    m[1] = "one";
    m[2] = "two";
    m[3] = "three";
    //...
    Map::iterator i = m.find(2);
    if (i != m.end())
        i->second[0] = 'T';
    else
        i = m.insert(i, std::make_pair(2, "Two"));
}
```

Change header

Change typedef

unordered containers

- API is largely compatible with map/set, except:

unordered containers

- API is largely compatible with map/set, except:
 - Iterators are forward, not bidirectional.

This won't work:

```
for (auto e = m.end(); e != m.begin();)
{
    --e;
    if (e->second == "two")
        e = m.erase(e);
}
```

unordered containers

- API is largely compatible with map/set, except:
 - Iterators are forward, not bidirectional.

unordered containers

- API is largely compatible with map/set, except:
 - Iterators are forward, not bidirectional.

But this will:

```
for (auto i = m.begin(); i != m.end(); )  
{  
    if (i->second == "two")  
        i = m.erase(i);  
    else  
        ++i;  
}
```

unordered containers

- API is largely compatible with map/set, except:

unordered containers

- API is largely compatible with map/set, except:
 - Template comparator is “equal_to”, not “less”.
 - Includes a template “hash” function.

```
template <class Key,  
         class Hash = hash<Key>,  
         class Pred = equal_to<Key>,  
         class Alloc = allocator<Key>>  
class unordered_set;
```

unordered containers

- API is largely compatible with map/set, except:
 - Template comparator is “equal_to”, not “less”.
 - Includes a template “hash” function.

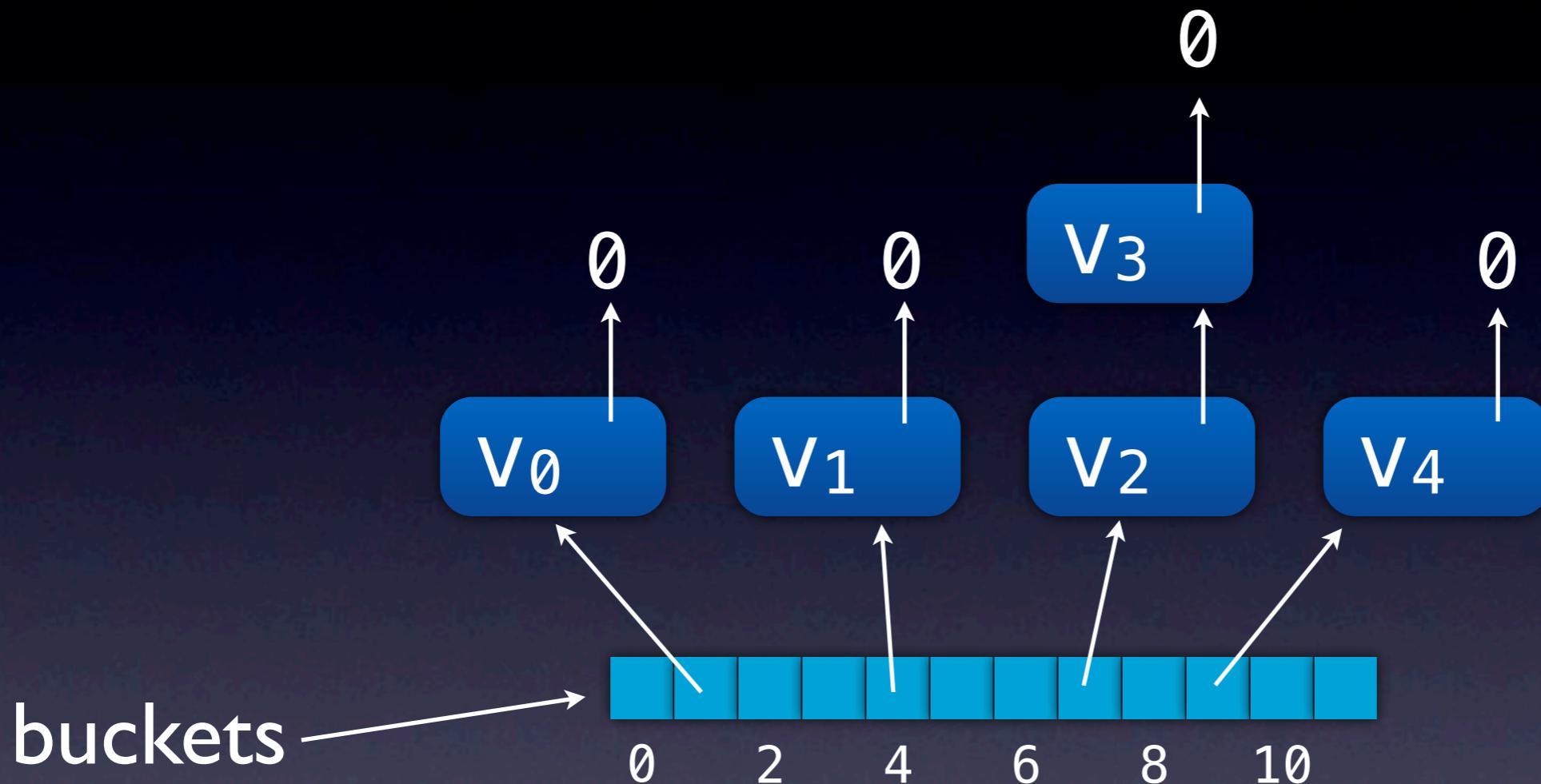
```
template <class Key, class T,  
         class Hash = hash<Key>,  
         class Pred = equal_to<Key>,  
         class Alloc =  
             allocator<pair<const Key, T>>>  
class unordered_map;
```

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

```
hasher    hash_function() const;  
key_equal key_eq()           const;
```

unordered containers



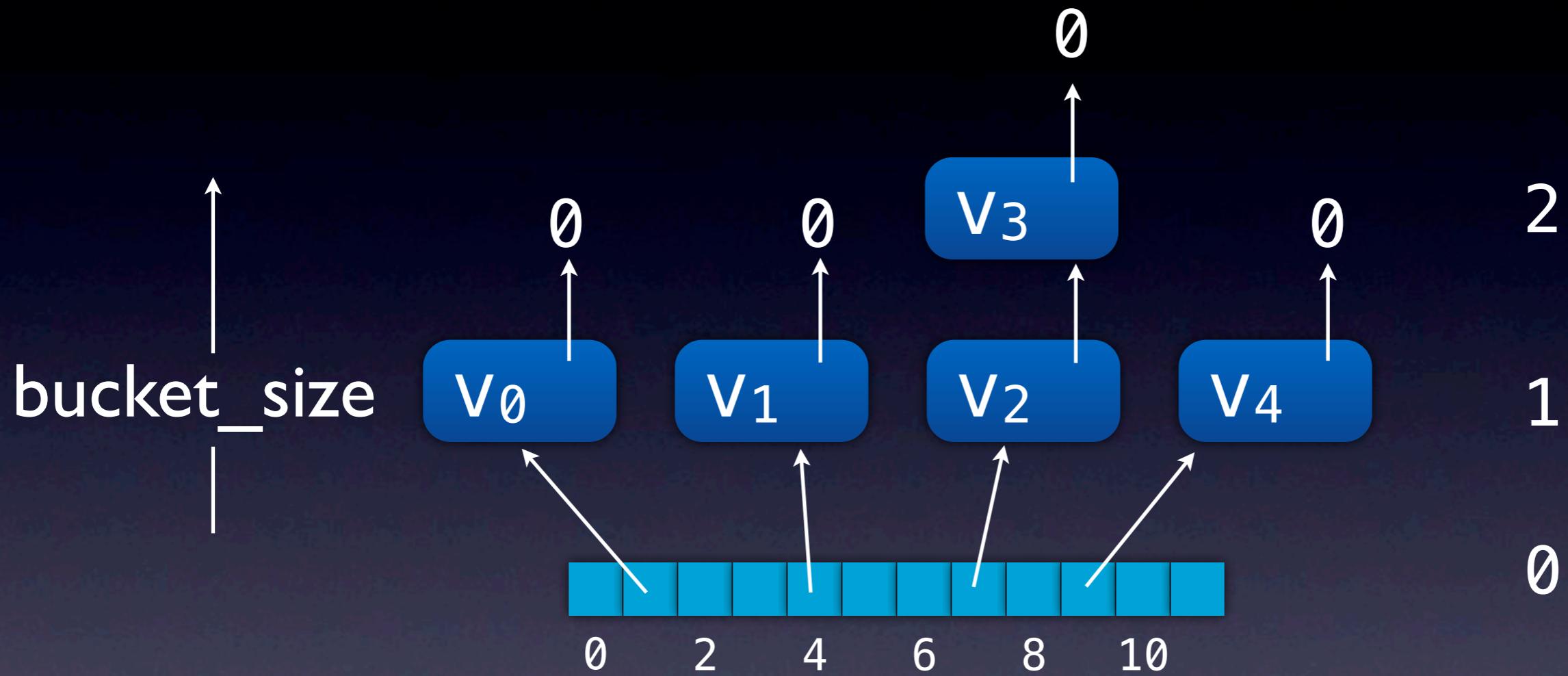
- Classic hash table structure.
- An array of singly-linked lists.

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

```
size_type bucket_count() const noexcept;  
size_type max_bucket_count() const noexcept;
```

unordered containers



- Classic hash table structure.
- An array of singly-linked lists.

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

```
size_type bucket_size(size_type n) const;  
size_type bucket(const key_type& k) const;
```

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

iterate just over a single bucket:

```
local_iterator begin (size_type n);
local_iterator end   (size_type n);
const_local_iterator begin (size_type n) const;
const_local_iterator end   (size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend  (size_type n) const;
```

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

`load_factor = size() / bucket_count()`

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

`load_factor = size() / bucket_count()`

```
float load_factor() const noexcept;
```

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

`load_factor = size() / bucket_count()`

`float load_factor() const noexcept;`

`load_factor_max ≥ size() / bucket_count()`

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

`load_factor = size() / bucket_count()`

`float load_factor() const noexcept;`

`load_factor_max ≥ size() / bucket_count()`

`float max_load_factor() const noexcept;`

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

`load_factor = size() / bucket_count()`

`float load_factor() const noexcept;`

`load_factor_max ≥ size() / bucket_count()`

`float max_load_factor() const noexcept;`

`void max_load_factor(float z);`

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

set `bucket_count()` $\geq n$:

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

set bucket_count() $\geq n$:

```
void rehash(size_type n);
```

unordered containers

- API is largely compatible with map/set, except:
 - Adds API to manage and inspect hash container structure.

set bucket_count() $\geq n$:

```
void rehash(size_type n);
```

Create enough buckets for n values:

```
void reserve(size_type n);
```

unordered containers

- API is largely compatible with map/set, except:
 - No lower/upper_bound but does have equal_range.
 - No operator<() but does have operator==().

unordered containers

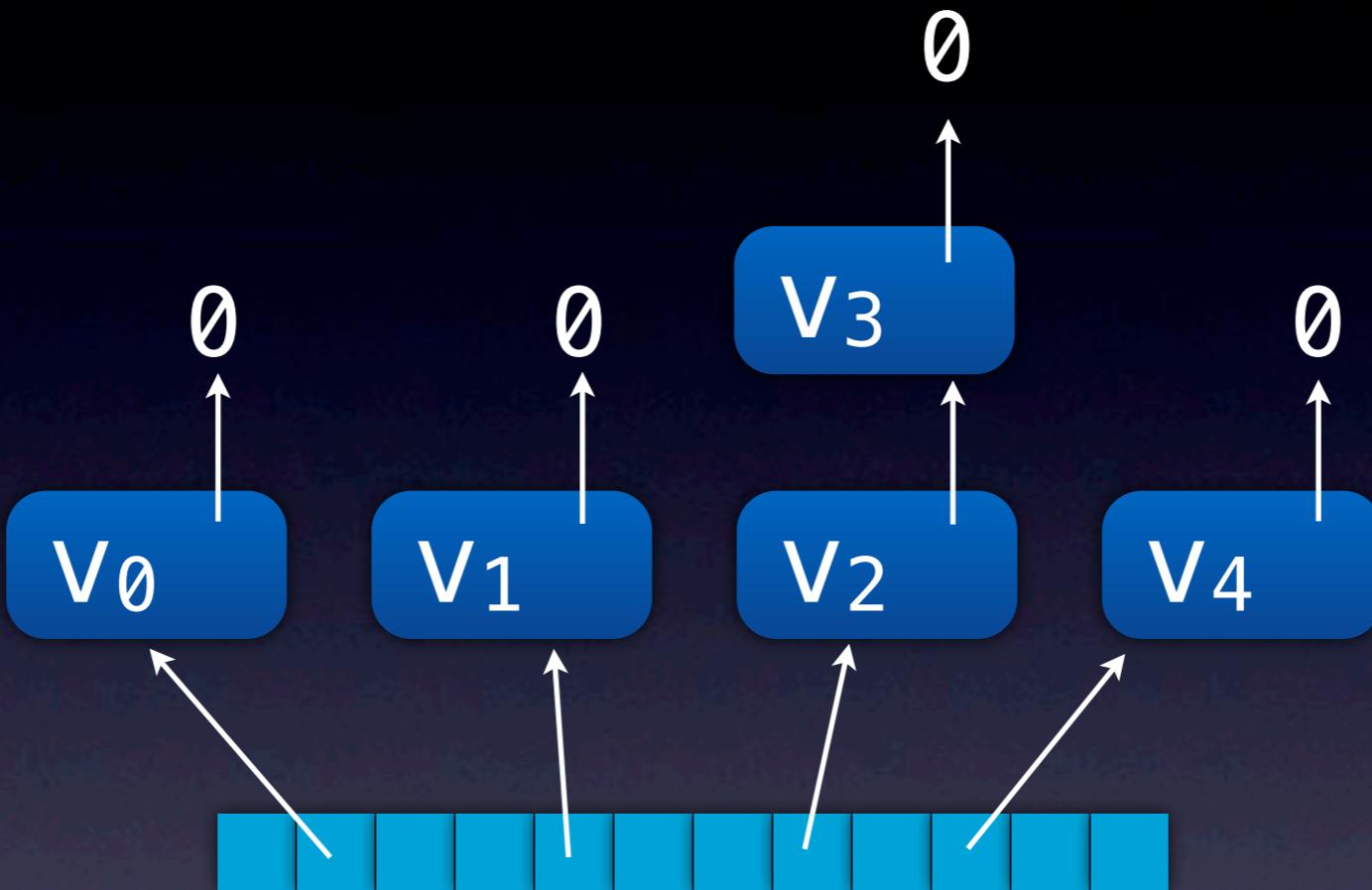
- API is largely compatible with map/set, except:
 - No operator`<()` but does have operator`==()`.
 - `x == y` implies that `x` and `y` have all of the same contained values, but not necessarily in the same order.

```
unordered_set<int> c1 = {1, 2, 3};  
unordered_set<int> c2 = {2, 3, 1};  
  
assert(c1 == c2); ✓
```

unordered containers

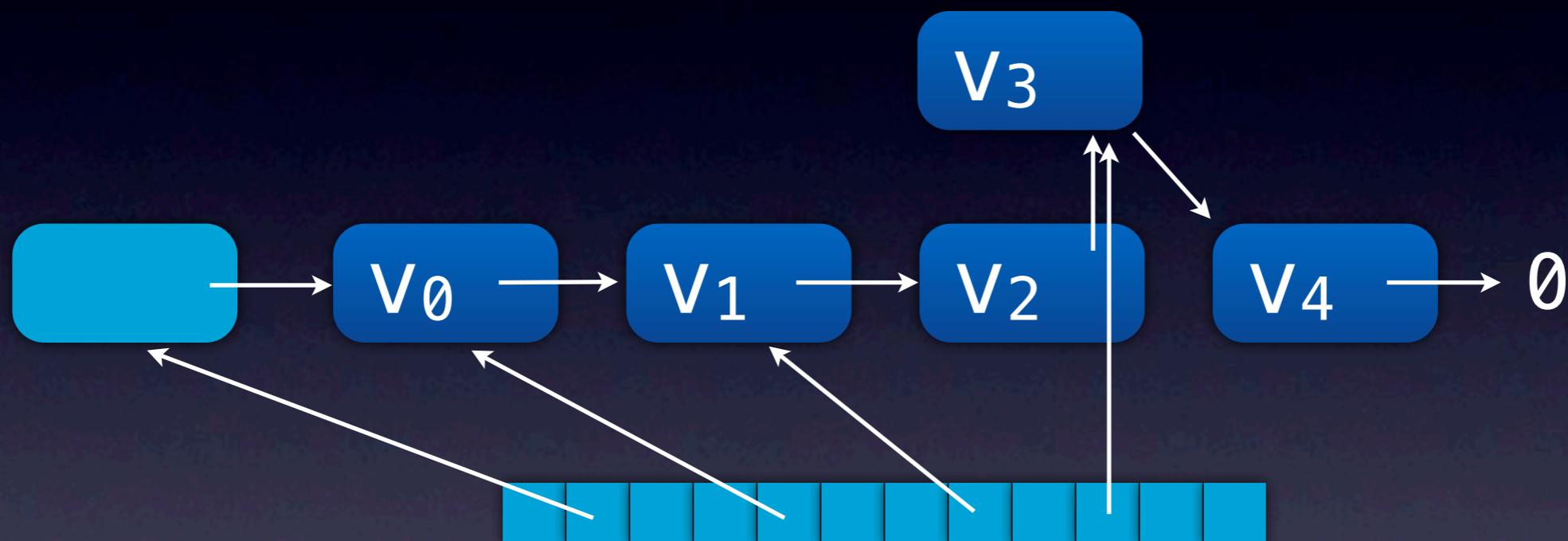
- API is largely compatible with map/set, except:
 - No operator`<()` but does have operator`==()`.
 - `x == y` implies that `x` and `y` have all of the same contained values, but not necessarily in the same order.
 - This operation is linear for `unordered_map` and `unordered_set` (assuming good hash).
 - For `unordered_multimap` and `unordered_multiset` it can get quadratic in the size of the largest equal range.

unordered containers



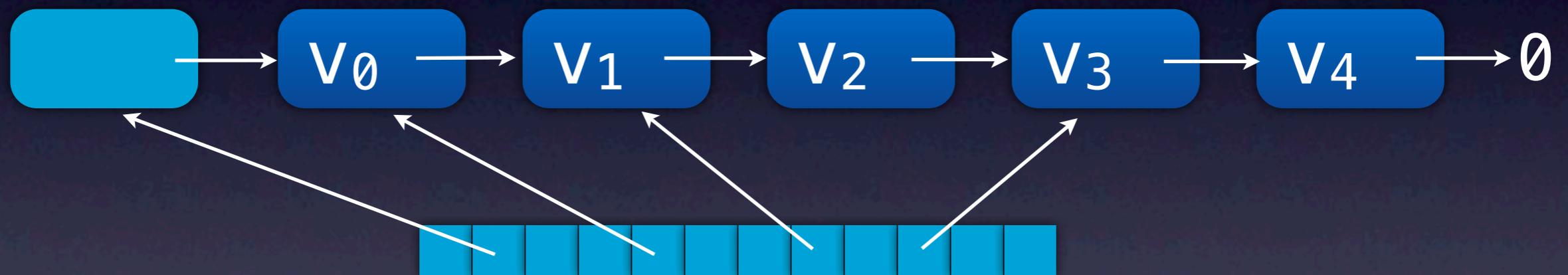
- Classic hash table structure.
 - An array of singly-linked lists.

unordered containers



- Revised for faster iteration.
- One singly linked list with an array of pointers into it.

unordered containers



- Revised for faster iteration.
- One singly linked list with an array of pointers into it.

Summary

- Containers are updated with:
 - Move semantics.
 - `emplace`
 - `initializer_list`
- New Containers
 - `array`
 - `forward_list`
 - `unordered containers`

