# Functions Want To Be Free

Functions Want To Be Free
by David Stone
david@doublewise.net

# Composability

- Write code once, reuse many times
- Build something big from many small things
- Don't Repeat Yourself (DRY)

# Object-Oriented programming

- Focus on the goal, not the syntax
- The goal is to maximize encapsulation
  - Not to write member functions

# Prefer non-friend free functions

- Isolates changes in implementation
- Forces implementation via the public interface
  - Improves your interface
- http://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197

# Monoliths Unstrung

- http://www.gotw.ca/gotw/084.htm
- Herb Sutter looked at 103 member functions of std::basic_string
- He made 71 of them free functions
- Many of those duplicated the functionality of <algorithm>
  - find
  - copy

# Implement std::vector

- A top-down approach
- Implement functions in terms of other functions
- Everything left over cannot be a normal function

# Size

```
size() { return end() - begin(); }
empty() { return begin() == end(); }
max_size() {
    return min(
        allocator_traits<allocator_type>::max_size,
        numeric_limits<size_type>::max() / sizeof(value_type)
    );
}
```

# shrink_to_fit

```
shrink_to_fit() {
    if (capacity() > size()) {
        vector temp(
            move_iterator_if_noexcept(begin()),
            move_iterator_if_noexcept(end()
        );
        swap(temp);
    }
}
```

# Comparisons

- operator==(lhs, rhs) { lhs.size() == rhs.size() and std::equal(lhs.begin(), lhs.end(), rhs.begin()) }
- operator<(lhs, rhs)
  { std::lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end() }
- All other operators != > <= >= can be implemented in terms of these two

# Element access

- operator[](index) { begin()[index] }
- at(index) { if index < size() operator[] else throw }
- front() { *begin() }
- back() { *std::prev(end()) }

# Specialized iterators

- cbegin(), cend() = call on container const &
- rbegin(), rend() = construct reverse_iterator
- crbegin(), crend() = combine both of the above

# swap

- Member swap should be deprecated
- Free function swap should rarely be specialized

# swap implementation

```
template<typename T>
void swap(T & lhs, T & rhs) {
    rhs = std::exchange(lhs, std::move(rhs));
}
```

# alternate swap implementation

```
template<typename T>
void swap(T & lhs, T & rhs) {
    auto temp = std::move(lhs);
    lhs = std::move(rhs);
    rhs = std::move(lhs);
}
```

# unique_ptr swap

```
void swap(unique_ptr & lhs, unique_ptr & rhs) {
    auto temp = std::move(lhs);
    lhs = std::move(rhs);
    rhs = std::move(lhs);
}
```

# unique_ptr swap, manually inlined

```
void swap(unique_ptr & lhs, unique_ptr & rhs) {
    auto temp = lhs.ptr; lhs.ptr = nullptr;
    lhs.ptr = rhs.ptr; rhs.ptr = nullptr;
    rhs.ptr = temp; temp = nullptr;
    delete temp;
}
```

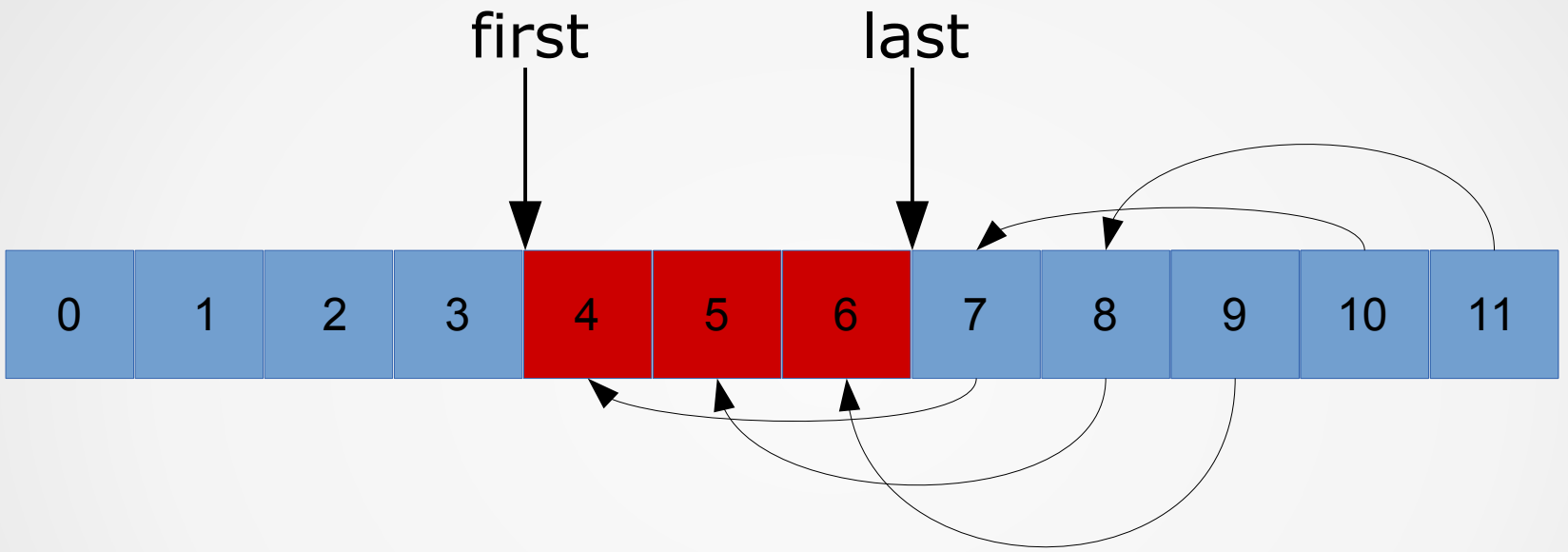
# unique_ptr swap, manually swapped

```
void swap(unique_ptr & lhs, unique_ptr & rhs) {
    auto temp = lhs.ptr;
    lhs.ptr = rhs.ptr;
    rhs.ptr = temp;
}
```

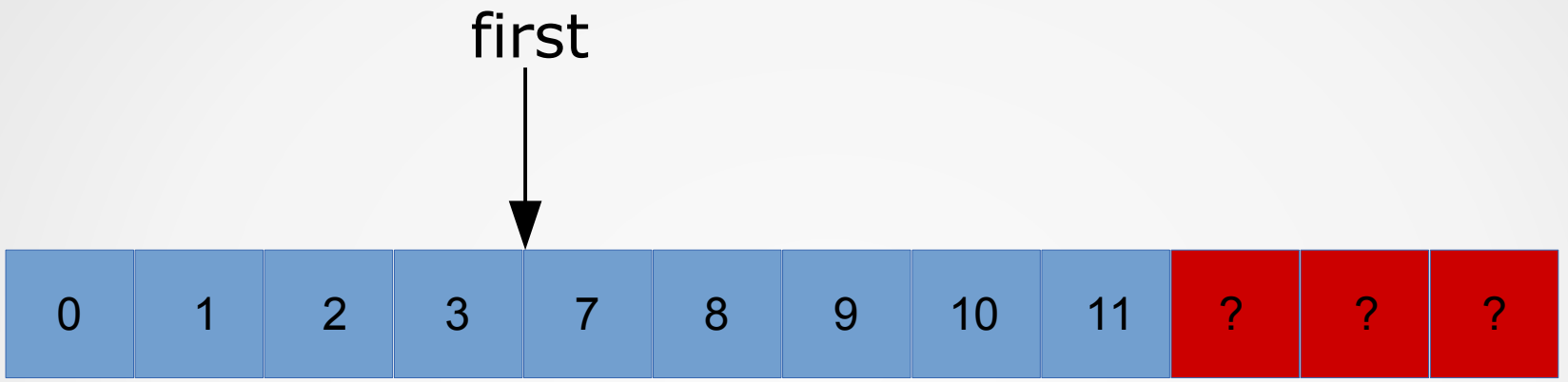
# unique_ptr swap, manually inlined

```
void swap(unique_ptr & lhs, unique_ptr & rhs) {
    auto temp = lhs.ptr; lhs.ptr = nullptr;
    lhs.ptr = rhs.ptr; rhs.ptr = nullptr;
    rhs.ptr = temp; temp = nullptr;
    delete temp;
}
```

# Erase (conceptual)



# Erase (conceptual)



# erase

```
erase(iterator b, iterator e) {
    auto to_clear = std::move(e, end(), b);
    while (to_clear != end()) {
        pop_back();
    }
}
```

# erase

```
erase(iterator target) {
    erase(target, std::next(target));
}
```

# clear

```
clear() {
    erase(begin(), end());
}
```

# resize

```
resize(size_type count) {
    while(size() > count) {
        pop_back();
    }
    while(size() < count) {
        emplace_back();
    }
}
// Plus copying overload
```

# push_back

```
push_back(value_type const & x) {
    emplace_back(x);
}
push_back(value_type && x) {
    emplace_back(std::move(x));
}
```

# emplace_back

```
emplace_back(Ts && ... args) {
    emplace(end(), std::forward<Ts>(args)...);
}
```

# assign

- Three overloads
  - n copies of some value
  - Range (iterator pair)
  - std::initializer_list<value_type>
    - Trivial to implement as iterator pair

# repeat_n

- Eric Niebler's range library has a repeat_n function
- Generates a range that returns n copies of the value
- Using an idea like this, we can reduce duplication

# assign

```
assign(size_type count, value_type const & value) {
    auto range = repeat_n(value, count);
    assign(range.begin(), range.end());
}
```

# assign

```
assign(ForwardIterator first, ForwardIterator last) {
    auto count = std::distance(first, last);
    if (count <= size()) {
        auto new_end = std::copy(first, last, begin());
        erase(new_end, end());
    } else if (count <= capacity()) {
        auto middle = my_special_copy_n(first, size(), begin());
        insert(end(), middle, last);
    } else {
        *this = { first, last };
    }
}
```

# insert

```
insert(iterator position, value_type const & value) {
    emplace(position, value);
}
insert(iterator position, value_type && value) {
    emplace(position, std::move(value));
}
```

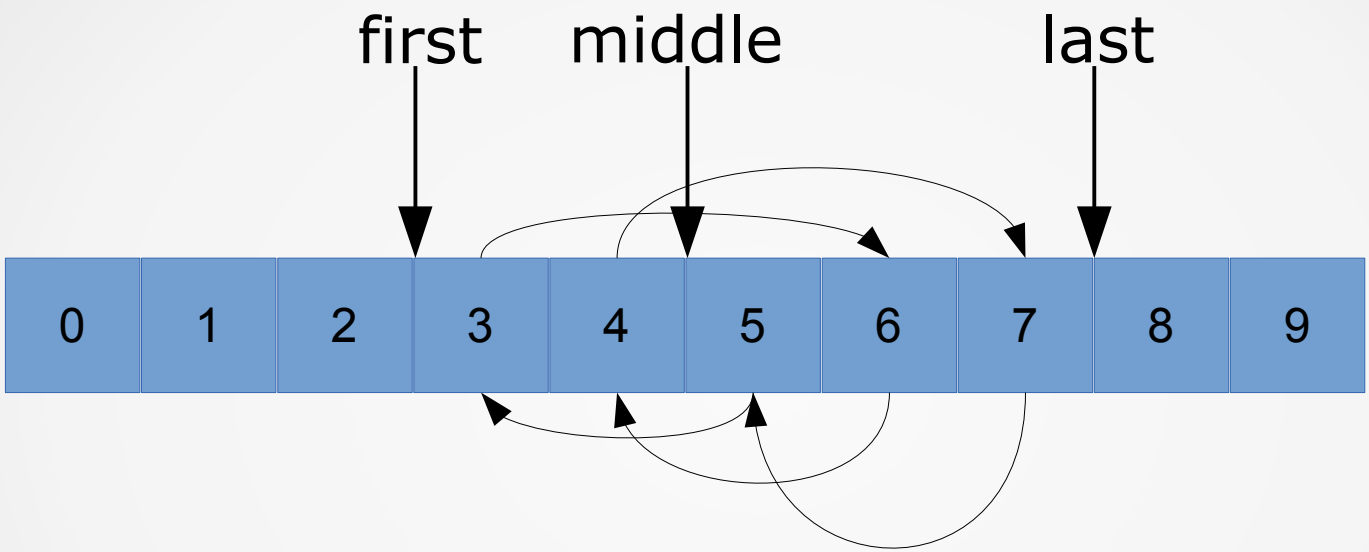
# insert

```
insert(iterator position, size_type count, value_type const
& value) {
    auto range = repeat_n(value, count);
    insert(position, range.begin(), range.end());
}
insert(iterator position, initializer_list<value_type> init) {
    insert(position, init.begin(), init.end());
}
```

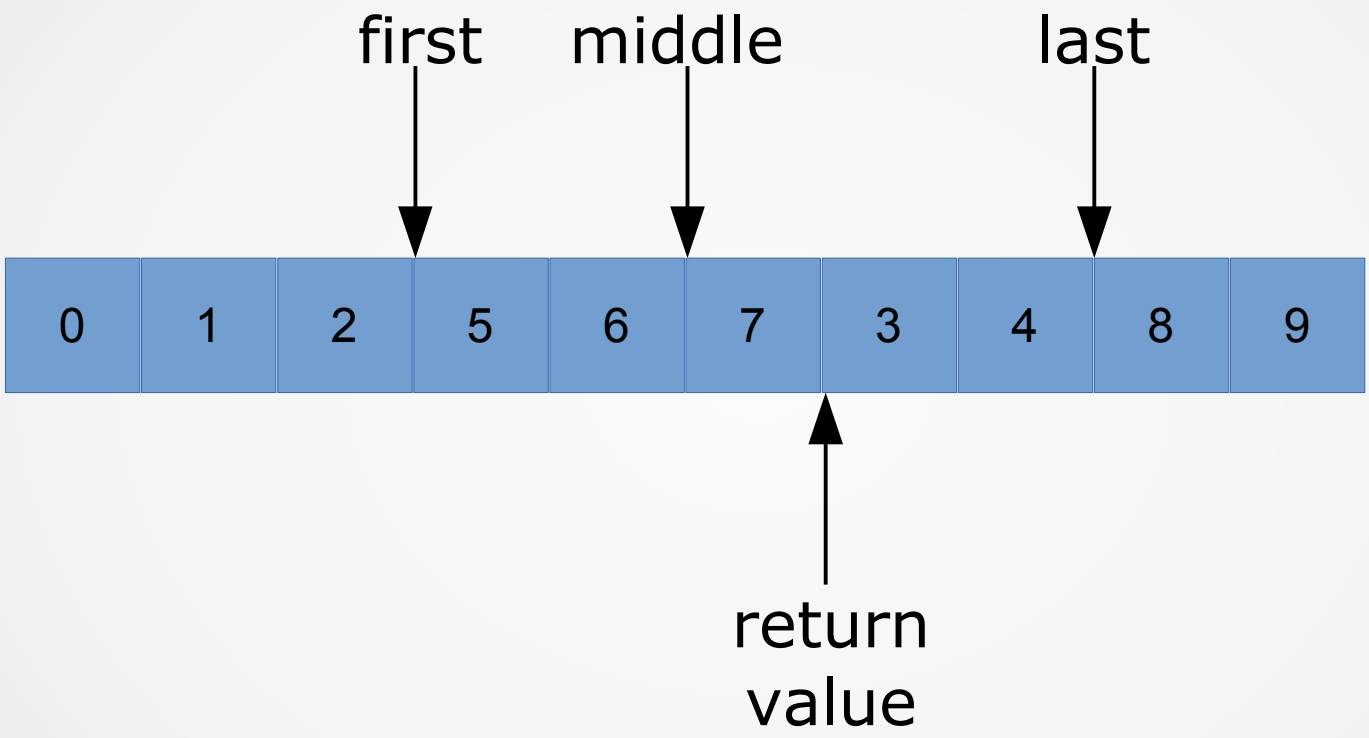
# A bad insert

```
insert(iterator position, iterator first, iterator last) {
    for (; first != last; ++first) {
        insert(position, *first);
    }
}
```

# Algorithm: rotate



# Algorithm: rotate



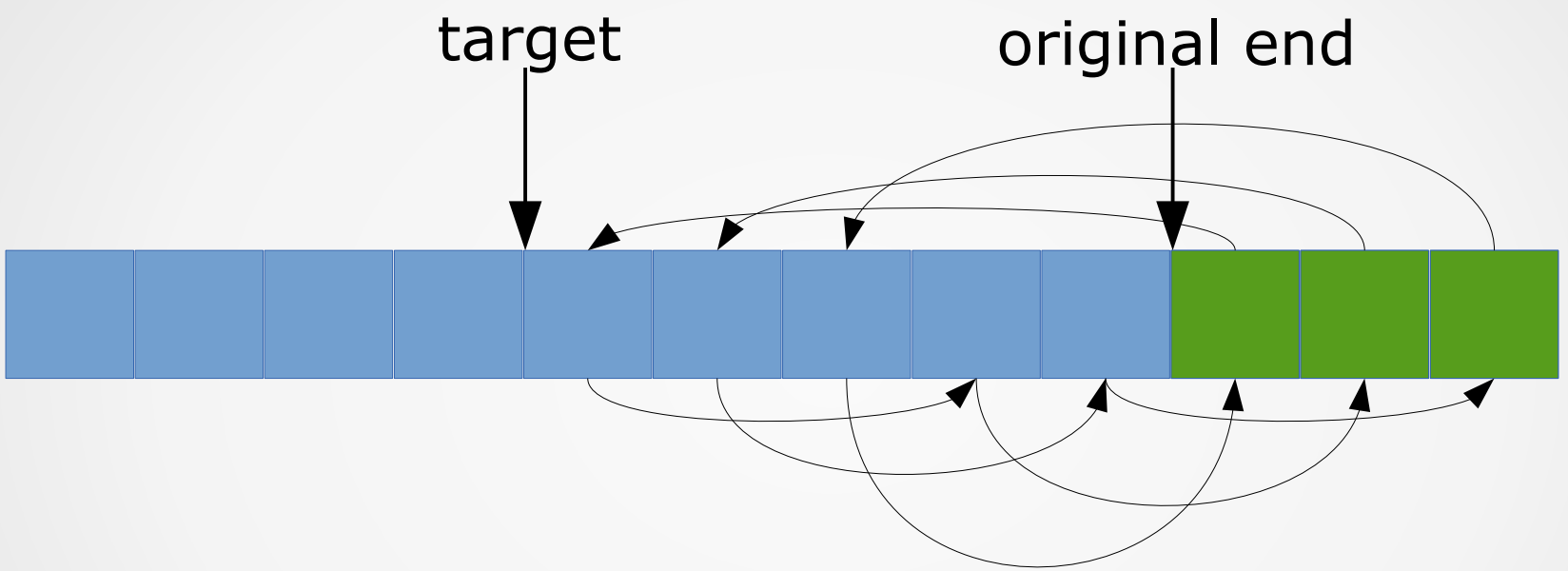
# Insert (a trap)

```
insert(iterator target, iterator first, iterator last) {
    auto original_end = end();
    for (; first != last; ++first) {
        push_back(*first);
    }
    std::rotate(target, original_end, end());
}
```

# Insert (still a trap)

```
insert(iterator target, iterator first, iterator last) {
    auto original_end = append(first, last);
    std::rotate(target, original_end, end());
}
```

# Our insertion algorithm



# What is left?

- begin
- end
- data
- get_allocator
- capacity
- reserve
- insert (range-based overload)
- emplace
- pop_back

# Why?

- Nice benefits to the implementation of vector
- What about other containers?

# Size for all sequence containers

- size() { end() - begin() }
  - For random-access iterators
    - std::distance can give linear-time size
  - std::list would return m_size
- empty() { begin() == end() }
  - Not size() == 0
  - Works for std::forward_list
- max_size() just works

# Comparisons

- operator==(lhs, rhs) { lhs.size() == rhs.size() and std::equal(lhs.begin(), lhs.end(), rhs.begin()) }
- operator==(lhs, rhs) { std::equal(lhs.begin(), lhs.end(), rhs.begin(), rhs.end()) }
- operator<(lhs, rhs)
  { std::lexicographical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end() }
- All other operators != > <= >= can be implemented in terms of these two

# Element access

- operator[](index) { begin()[index] }
- at(index) { if index < size() operator[] else throw }
- front() { *begin() }
- back() { *std::prev(end()) }

# Specialized iterators

- cbegin(), cend() = call on container const &
- rbegin(), rend() = construct reverse_iterator
- crbegin(), crend() = combine both of the above

# swap

- Use the default

# erase

```
erase(iterator b, iterator e) {
    auto to_clear = std::move(e, end(), b);
    while (to_clear != end()) {
        pop_back();
    }
}
```

# erase

```
erase(iterator target) {
    magic_internal_unlink_call();
}

erase(iterator target) {
    erase(target, std::next(target));
}
```

# clear

```
clear() {
    erase(begin(), end());
}
```

# resize

```
resize(size_type count) {
    while(size() > count) {
        pop_back();
    }
    while(size() < count) {
        emplace_back();
    }
}
```

# push_back

```
push_back(value_type const & x) {
    emplace_back(x);
}
push_back(value_type && x) {
    emplace_back(std::move(x));
}
```

# emplace_back

```
emplace_back(Ts && ... args) {
    emplace(end(), std::forward<Ts>(args)...);
}
```

# assign

```
assign(size_type count, value_type const & value) {
    auto range = repeat_n(value, count);
    assign(range.begin(), range.end());
}
```

# assign

```
assign(ForwardIterator first, ForwardIterator last) {
    auto count = std::distance(first, last);
    if (count <= size()) {
        auto new_end = std::copy(first, last, begin());
        erase(new_end, end());
    } else if (count <= capacity()) {
        auto middle = my_special_copy_n(first, size(), begin());
        insert(end(), middle, last);
    } else {
        *this = { first, last };
    }
}
```

# Friends and members

# operators that must be members

- operator=
- operator[]
- operator()
- operator->

# Guidelines

- If it must be a member, make it a member
  - Virtual functions
  - Member operators
  - Constructors
- If it can be a non-friend function, make it free
  - Only if no loss of efficiency
    - Remember insert
- Otherwise, maximize consistency

# Consistency

- size()
  - Random-access iterators
  - Access member variable for std::list
  - Make it a free function
  - Friend of std::list

# Consistency

- insert
  - Only one overload needs private access
  - Make overloads non-member, non-friends
  - friend the one overload that needs it

# Friend functions violate encapsulation

- So do member functions

# Uniform Function Call Syntax

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4174.pdf
  - Bjarne Stroustrup
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4165.pdf
  - Herb Sutter

# What they both do

- x.f()
  - First looks for member function f
  - Then looks for free function f that accepts an x
- Would allow backwards compatible changes

# N4174 (Stroustrup)

- f(x)
  - First looks for member function f
  - Then looks for free function f that accepts an x
- The goal is consistency
  - Each call syntax would be identical
- Range-based for loop rules for begin and end
- Member functions hide free functions
- Considers removing inaccessible members from overload resolution

# N4165 (Sutter)

- Does not propose changing f(x)
- x.f(a)
  - Can call f(x, a) under either proposal
  - Can call f(a, x) under N4165
- The goal is backward compatibility

# Current version

- Backward compatible
- Intersection of both proposals

# Functions Want To Be Free

- Maximize code reuse
- Maximize encapsulation