Templates

Swap Function

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

https://godbolt.org/z/vszT69xz1

Here we have a function that swaps the values of two ints.

But a swap function is useful for many different types…

Swap Function

```cpp
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

```cpp
void swap(double& x, double& y) {
    double temp = x;
    x = y;
    y = temp;
}
```

```cpp
void swap(std::string& x, std::string& y) {
    std::string temp = x;
    x = y;
    y = temp;
}
```

The only thing changing in this code is the type.

Is there a way to save us having to write this function over and over with different types?

Yes, templates!

Templates

```
template <typename T>
void my_swap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}
```

https://godbolt.org/z/c35fcKxvr

A template gives us a parameter to represent a type. This parameter can be instantiated with different types.

```
int a {3};
int b {5};
my_swap<int>(a,b);
```

```
std::string hello {"Hello"};
std::string world {"World"};
my_swap<std::string>(hello, world);
```

We can also omit the type inside the angle brackets: the compiler can deduce it.

Templates

```
template <typename T>
void my_swap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}
```

https://godbolt.org/z/c35fcKxvr

A template moves the work of instantiating the code with different types from the programmer to the compiler.

The compiler will explicitly write a version of the function for every type needed.

Print Function

```cpp
template <typename T>
void print(const std::vector<T>& vec) {
    for (const T& x : vec) {
        std::cout << x << '\n';
    }
    std::cout << '\n';
}
```

https://godbolt.org/z/fW914PEb7

We can write a function to print a vector containing any type of element.

Here T stands for the type of elements in the vector.

Print Function

```
template<typename T>
void print(const std::vector<T>& vec) {
    for (const auto& x : vec) {
        std::cout << x << '\n';
    }
    std::cout << '\n';
}
```

The compiler can deduce the type of element in the container. We can use auto in the for loop instead.

Print Function

```
void print(const auto& container) {
    for (const auto& x : container) {
        std::cout << x << '\n';
    }
    std::cout << '\n';
}
```

https://godbolt.org/z/6q8n6eM4f

As of C++20 you can also use auto in the parameter list too!

This prints out the contents of any container that allows range-based for loops.

The auto syntax is easier than using a template type for the container and for the type of element in the container.

Back to Swap

```
void my_swap(auto& x, auto& y) {
    auto temp = x;
    x = y;
    y = temp;
}
```

https://godbolt.org/z/z469b6d1z

A difference between auto and explicitly using a template type is that with a template we can express that the type of x and y is the same.

auto here allows x and y to be of different types, which may not compile.

Templated Class

We can also use templates in defining a class.

We can upgrade our MyInteger class to be a wrapper around an arbitrary type, rather than just an integer.

We follow the exposition of Stepanov to create a templated "Singleton" class.

Efficient Programming with Components, Lecture 2 Part 1

Templated Class

We can also use templates in defining a class.

We can upgrade our MyInteger class to be a wrapper around an arbitrary type, rather than just an integer.

We follow the exposition of Stepanov to create a templated “Singleton” class.

Efficient Programming with Components, Lecture 2 Part 1

```
class MyInteger {
 private:
  int value {};

 public:
  // constructor
  explicit MyInteger(int input = 0) : value {input} {}

  // copy constructor
  MyInteger(const MyInteger& x) : value {x.value} {}

  // assignment operator
  MyInteger& operator=(const MyInteger& x) {
    value = x.value;
    return *this;
  }

  // destructor
  ~MyInteger() {}

  // determine if two MyIntegers are equal
  friend bool operator==(const MyInteger& x, const MyInteger& y) {
    return x.value == y.value;
  }

  // determine if one MyInteger is less than another
  friend bool operator<(const MyInteger& x, const MyInteger& y) {
    return x.value < y.value;
  }
};
```

We can make this a templated class to allow not just an int but any type*.

*what operations must a type allow for this to work?

```cpp
template <typename T>
class Singleton {
 private:
  T value {};

 public:
  // constructor
  explicit Singleton(T input = T {}) : value {input} {}

  // copy constructor
  Singleton(const Singleton& x) : value {x.value} {}

  // assignment operator
  Singleton& operator=(const Singleton& x) {
    value = x.value;
    return *this;
  }

  // destructor
  ~Singleton() {}

  // determine if two Singletons are equal
  friend bool operator==(const Singleton& x, const Singleton& y) {
    return x.value == y.value;
  }

  // determine if one Singleton is less than another
  friend bool operator<(const Singleton& x, const Singleton& y) {
    return x.value < y.value;
  }
};
```

Doing this is as easy as replacing int everywhere with T.

We can instantiate this class as

```cpp
Singleton<int> x {3};
Singleton<std::string> z {"hello"};
```

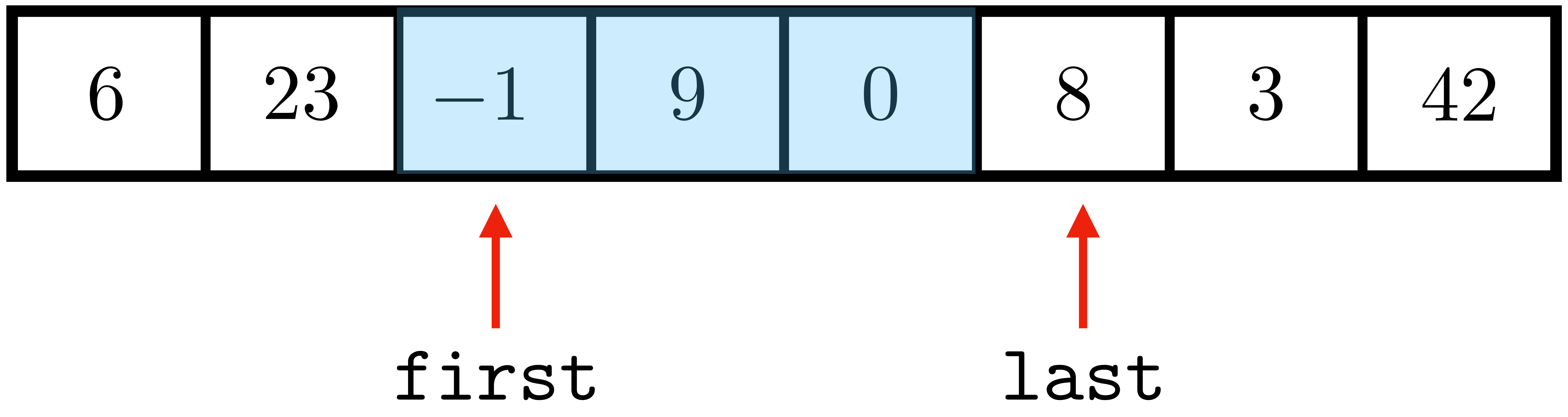
The type in the angle brackets can also be omitted.

Iterators

Find

Let's say we have an array and we want to determine if it contains a given element.

Let's consider a general version of the problem where we search in a range of elements in the array specified by two pointers.



We will use a half-open range: from `first` up to but not including `last`.

Find

```
int* find(int* first, int* last, int value) {
    int* ptr = first;
    for (; ptr != last; ++ptr) {
        if (*ptr == value) {
            return ptr;
        }
    }
    return ptr;
}
```

https://godbolt.org/z/4TxzGKWY6

If value is not found in the range we return `last`.

This serves as a sentinel value as it is not part of the range.

Find

```
int* find(int* first, int* last, int value) {
    int* ptr = first;
    for (; ptr != last; ++ptr) {
        if (*ptr == value) {
            return ptr;
        }
    }
    return ptr;
}
```

https://godbolt.org/z/4TxzGKWY6

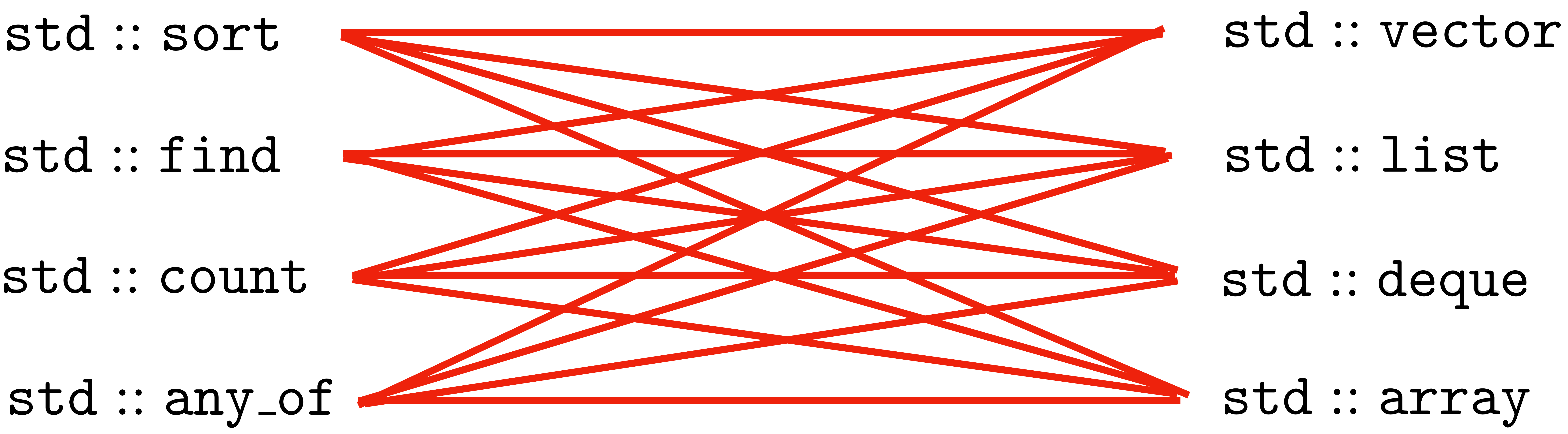
Find is a natural operation that we might want to implement for any container.

Do we have to write a separate function for each container?

Algorithms

Containers

$< \texttt{algorithm} >$



If we have m algorithms and n containers, we would have to write $m * n$ functions.

Generic Programming

Start with a concrete algorithm.

```
int* find(int* first, int* last, int value) {
    int* ptr = first;
    for (; ptr != last; ++ptr) {
        if (*ptr == value) {
            return ptr;
        }
    }
    return ptr;
}
```

Identify the primitive operations that make this algorithm work.

Generalize the algorithm to any type that supports those primitive operations.

Generic Programming

```
int* find(int* first, int* last, int value) {
    int* ptr = first;
    for (; ptr != last; ++ptr) {
        if (*ptr == value) {
            return ptr;
        }
    }
    return ptr;
}
```

Let us abstract out the functionality provided by pointers here:

We can increment a pointer (go to next element).

We can check if two pointers are equal.

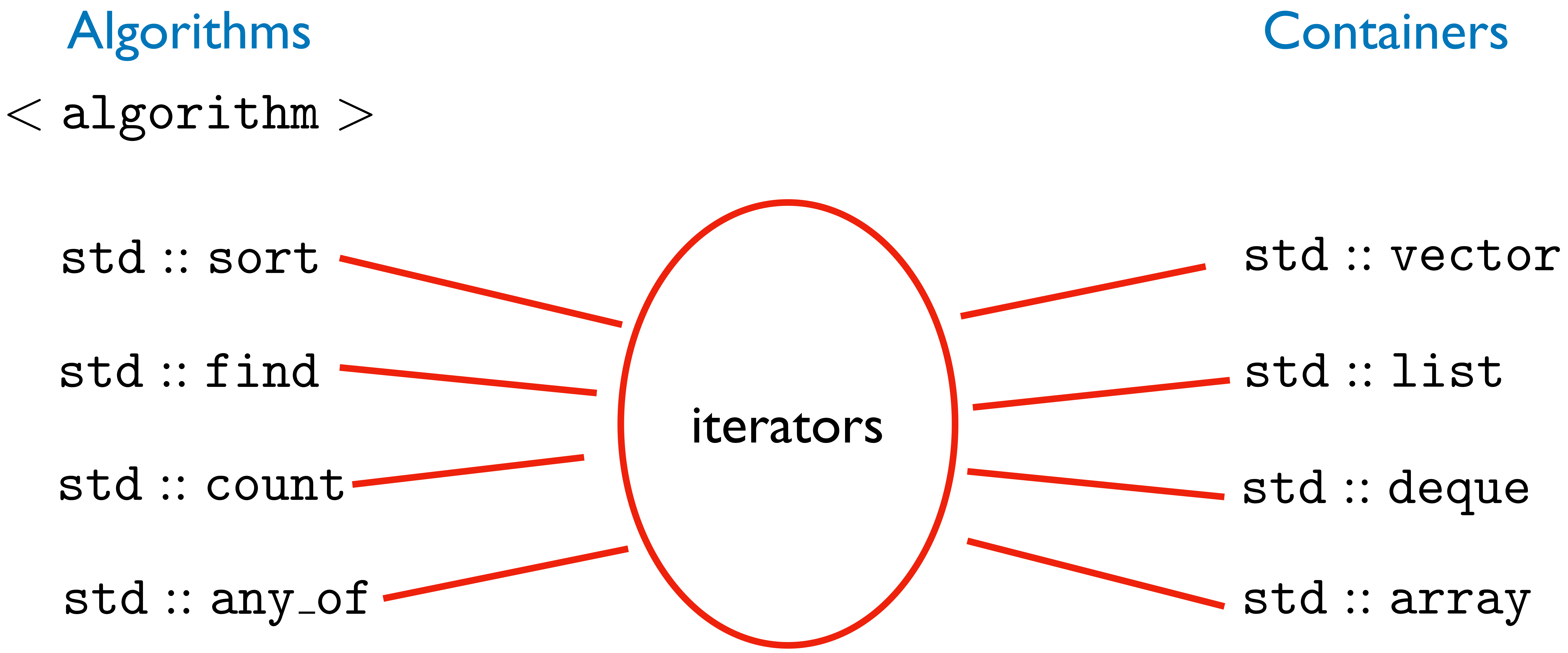
We can dereference a pointer.

Iterators

An iterator is like a generalized pointer, that supports these operations (and sometimes more).

Every C++ sequence container defines an iterator.

We can then write algorithms generically in terms of iterators.



Iterators are the link between algorithms and containers in C++.

Each sequence container defines an iterator.

Algorithms are then generically written in terms of iterators*.

Types of Iterators

There is a hiccup in this nice picture. We don't want to pay any price in performance for a more generic algorithm.

The way we can access elements varies depending on the container.

In a singly linked list we can only move forward, not backward.

→ forward iterator

In a doubly linked list we can move forward or backward.

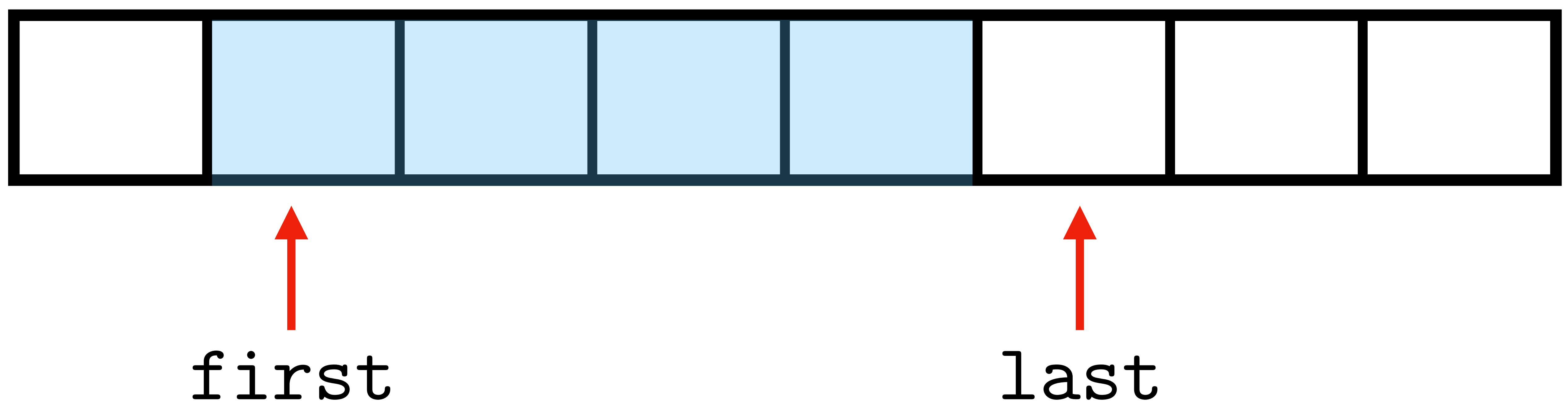
→ bidirectional iterator

In a vector we can quickly jump to any element.

→ random access iterator

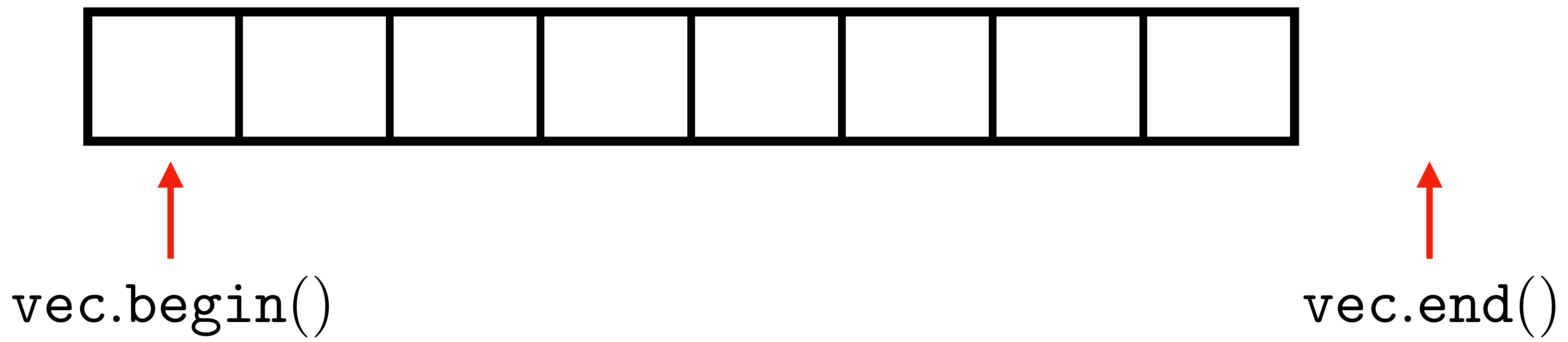
Half-Open Intervals

As in our find example, many standard library algorithms work on a half-open range specified by two iterators.



Begin and End

Every sequence container provides two member functions that return an iterator, `begin` and `end`.



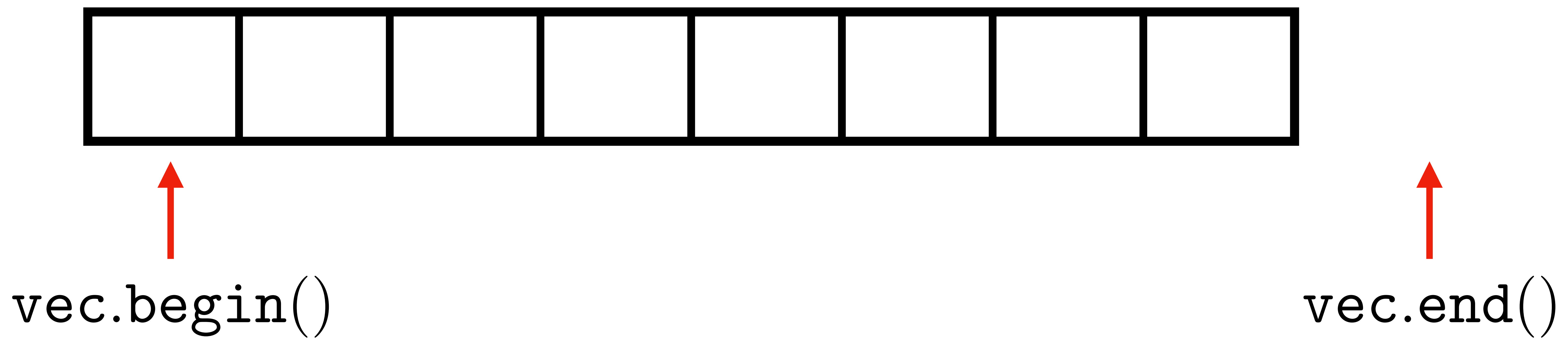
`begin` points to the first element in the container.

`end` is a sentinel value that is not part of the container.

`vec.end()` should not be dereferenced.

Begin and End

Every sequence container provides two member functions that return an iterator, `begin` and `end`.



The half-open range from `vec.begin()` up to but not including `vec.end()` is the entire vector.

Find Example

Here is how we can use the find function in the standard library.

```
std::list<int> li {1,2,3,4};
auto it = std::find(li.begin(), li.end(), 3);
```

This returns an iterator to the first occurrence of 3 in the list.

Iterating over a container

Iterating through a list:

```
std::list<int> li {1,2,3,4};
for (std::list<int>::iterator it = li.begin(); it != li.end(); ++it) {
    std::cout << *it << '\n';
}
```

Iterators adopt the dereferencing syntax from pointers:

`*it` is the value pointed to by the iterator `it`.

Use of Iterators

Iterating through a list:

```
std::list<int> li {1,2,3,4};
for (std::list<int>::iterator it = li.begin(); it != li.end(); ++it) {
    std::cout << *it << '\n';
}
```

* the same idiom can be used for any other sequence container.

* iterators for `list` do not support comparison.

* this is a good time for `auto`.

Iterating backwards

There is a nice syntax for iterating backwards over a container

```
std::list<int> li {1,2,3,4};
for (auto it = li.rbegin(); it != li.rend(); ++it) {
    std::cout << *it << '\n';
}
```

This uses `rbegin` and `rend` which return a reverse iterator.

Other than that the syntax is the same—we increment the reverse iterator.