Lecture 8: Functions

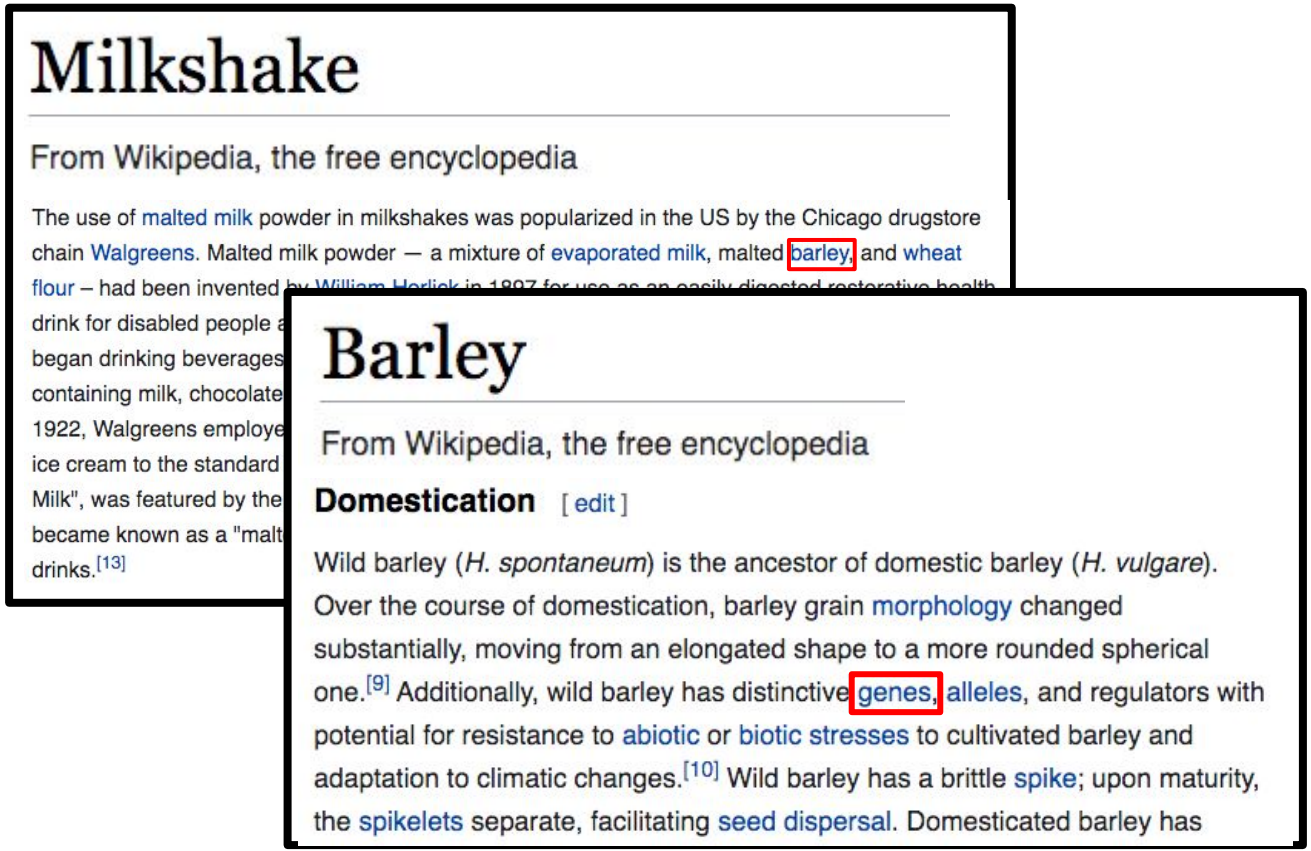
CS 106L, Fall ‘20

Today's Agenda

- Assignment 1
- Recap: Concept lifting
- New stuff with concept lifting
- Lambda functions
- Intro to STL

Assignment 1: WikiRacer

Milkshake → Gene



Milkshake

From Wikipedia, the free encyclopedia

The use of malted milk powder in milkshakes was popularized in the US by the Chicago drugstore chain Walgreens. Malted milk powder — a mixture of evaporated milk, malted barley, and wheat flour – had been invented by William Horlick in 1897 for use as an easily digested restorative health drink for disabled people a
began drinking beverages
containing milk, chocolate
1922, Walgreens employe
ice cream to the standard
Milk", was featured by the
became known as a "malt
drinks.[13]

Barley

From Wikipedia, the free encyclopedia

Domestication [edit]

Wild barley (*H. spontaneum*) is the ancestor of domestic barley (*H. vulgare*). Over the course of domestication, barley grain morphology changed substantially, moving from an elongated shape to a more rounded spherical one.[9] Additionally, wild barley has distinctive genes, alleles, and regulators with potential for resistance to abiotic or biotic stresses to cultivated barley and adaptation to climatic changes.[10] Wild barley has a brittle spike; upon maturity, the spikelets separate, facilitating seed dispersal. Domesticated barley has

Emu → Stanford University

Emu

From Wikipedia, the free encyclopedia

Encyclopædia Britannica Eleventh Edition

From Wikipedia, the free encyclopedia

Cornell University

From Wikipedia, the free encyclopedia

Stanford University

From Wikipedia, the free encyclopedia

```
To find a ladder from startPage to endPage:
  Make startPage the currentPage being processed.
  Get set of links on currentPage.
  If endPage is one of the links on currentPage:
    We are done! Return path of links followed to get here.
  Otherwise visit each link on currentPage in an intelligent way and
  search each of those pages in a similar manner.
```

Screenshots

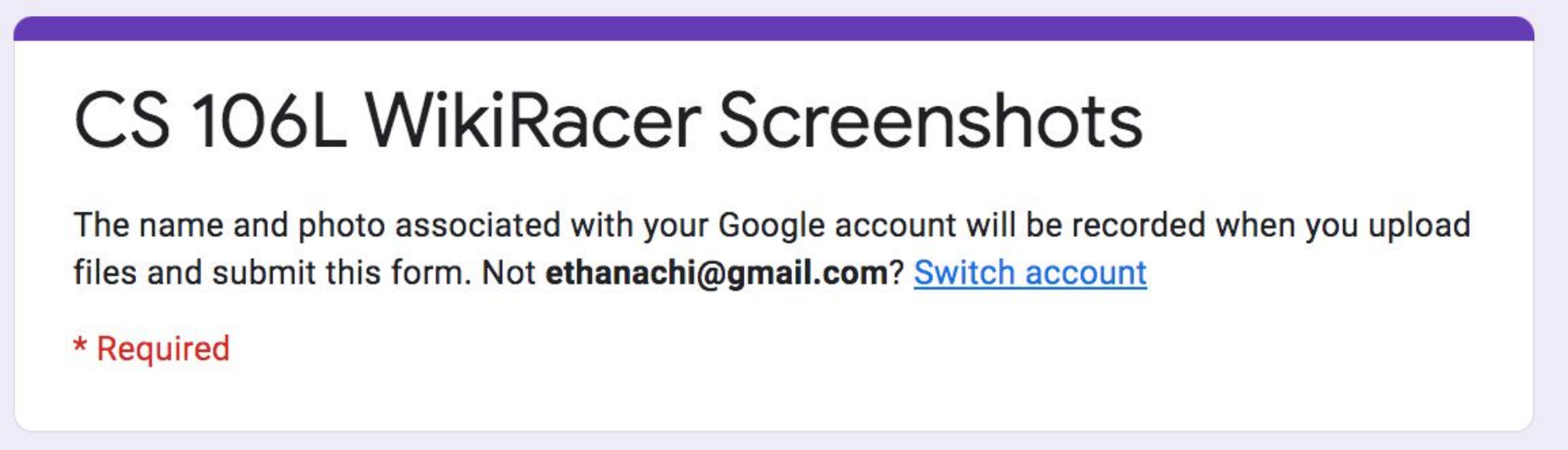
Screenshots

In order to verify that your computer works correctly with the Qt libraries, please take some screenshots and submit them! Open the **InternetTest** project in Qt Creator and run it. This should prompt you with a console with a bunch of text; Every time the console asks you to "take screenshot and press enter to continue", take a screenshot, then press enter.

Once you have **4** screenshots in total, please submit here. If you get any compiler errors, or anything strange, please screenshot those too.

⚠ Screenshots are **due Sunday, October 11 at 11:59PM!** This is so that if any issues come up, we will have enough time to patch them up.

Fill in the form for credit



CS 106L WikiRacer Screenshots

The name and photo associated with your Google account will be recorded when you upload files and submit this form. Not **ethanachi@gmail.com**? Switch account

* Required

Screenshots due Sunday, Oct 11

Step 1: extract links

Here's an example of what our function should do. Given the code:

```
<p>
  In <a href="/wiki/Topology">topology</a>, the <b>long line</b> (or <b>Alexan
  <a href="/wiki/Topological_space">topological space</a> somewhat similar to
  <a href="/wiki/Lindel%C3%B6f_space">Lindelöf</a> nor
  <a href="/wiki/Separable_space">separable</a>). Therefore, it serves as one
  <a href="http://www.ams.org/mathscinet-getitem?mr=507446">[1]</a>. Intuitive
  <a href="/wiki/Special:BookSources/978-1-55608-010-4">this</a> book for more
  </p>
```

In this case, our function would return an unordered_set containing the following strings:

```
{"Topology", "Topological_space", "Real_line", "Lindel%C3%B6f_space", "Separab
```

Step 2: exploration

```
vector<string> findWikiLadder(const string& start_page,
const string& end_page) {
  // creates WikiScraper object
  WikiScraper scraper;

  // Comparison function for priority_queue
  auto cmpFn = /* declare lambda comparator function */;

  // creates a priority_queue names ladderQueue
  std::priority_queue<vector<string>, vector<vector<string>>>
  decltype(cmpFn)> ladderQueue(cmpFn);

  // ... rest of implementation

}
```

Things required for credit

- lambdas
- STL functions
- iterators

we'll learn more about #1 and #2 **today**!

Due: Friday, Oct. 23 at **11:59pm PST**

*if you filled out survey #1 you have **1** late day*

we will send out survey #2 soon!

Recap: concept lifting

What assumptions are we making about the parameters?

Can we solve a more general problem by relaxing some of the constraints?

Why write generic functions?

Count the number of times **3** appears in a **std::list<int>**.
Count the number of times **“X”** appears in a **std::istream**.
Count the number of times **a vowel** appears in a **std::string**.
Count the number of times **a college student** appears in a **census**.

How many times does a <T> appear in an iterator<T>?

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (*iter == val) count++;
  }
  return count;
}

vector<string> v; count_occurrences(v.begin(), v.end(), “test”);
```

🤔 Great! This is a very general way to solve the problems

We can now solve these questions…

Count the number of times **3** appears in a **list<int>**.
Count the number of times **'X'** appears in a **std::deque<char>**.
Count the number of times **'Y'** appears in a **string**.
Count the number of times **5** appears in the **second half of a vector<int>**.

But how about this?

Count the number of times **an odd number** appears in a **vector<int>**.
Count the number of times **a vowel** appears in a **string**.

Concept lifting cont.

Generalization: A predicate is a function which takes in some number of arguments and returns a boolean.

Unary Predicates

```
bool isEqualTo3(int a) {
     return (a == 3);
}


bool isVowel(char c) {
     return std::find(“aeiou”, c) != -1;
}
```

Binary Predicate

```
bool isDivisibleBy(int a, int b) {
     return (a % b == 0);
}


bool isLessThan(int a, int b) {
     return (a < b);
}
```

Calling this function with a predicate

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (pred(*iter) == val) count++;
  }
  return count;
}
```

```
bool is_even(int i) { return (i % 2) == 0; }
vector<int> v; count_occurrences(v.begin(), v.end(), is_even);
// this is a function pointer
```

Function pointers generalize poorly

```
bool is_greater_than_5(int i) {
    return (i > 5);
}

bool is_greater_than_6(int i) {
    return (i > 6);
}

bool is_greater_than_7(int i) {
    return (i > 7);
}
// We can't add the limit as a parameter to the function (why?)
```

This is fundamentally a scope problem

We need to pass the **limit** in without adding another parameter...

Lambda Functions

Lambda functions let you make a new function on the fly

```
int main() {
    auto print_int = [](int x) {
                        cout << x << endl;
                    };

    // print_int is a function now!
    print_int(5); // “5”
    print_int(7); // “7”

    // what type is print_int? who cares!
}
```

Lambda capture allows you to pass information in

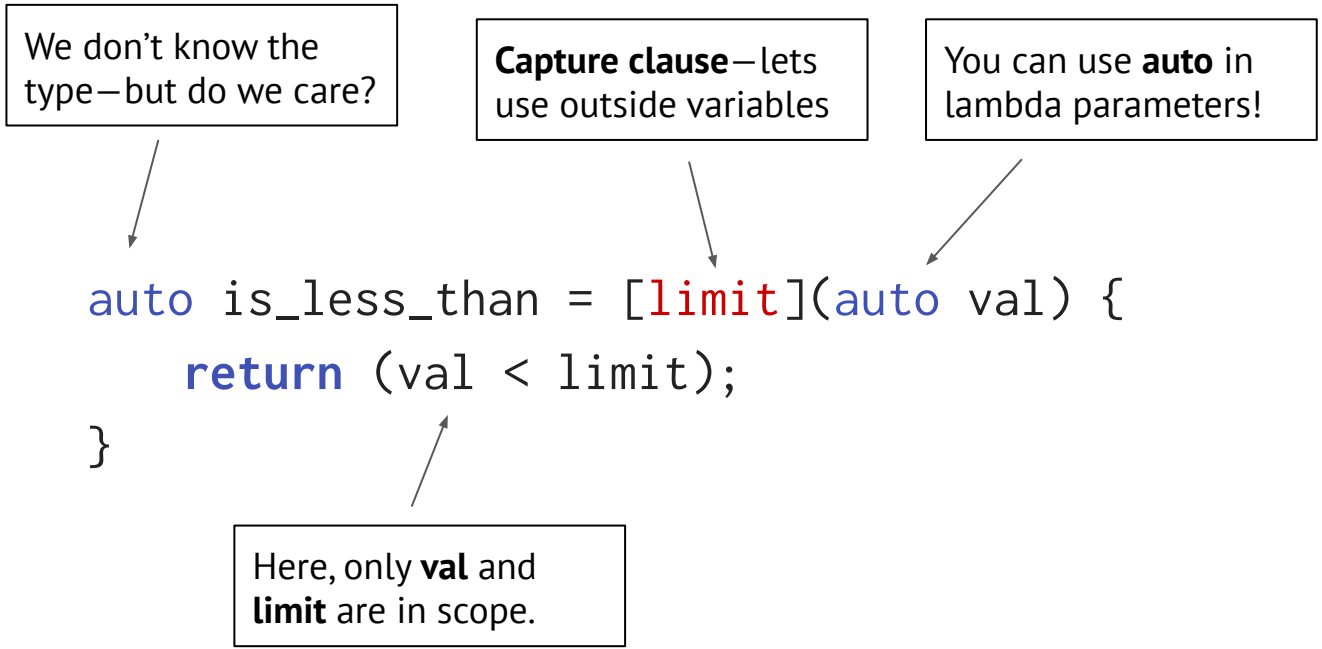
```
int main() {
    int limit;
    std::cin >> limit;
    auto is_less_than = [limit](int val) { return (val < limit) };

    // this solves our earlier problem!
}
```

Counting how many numbers are less than a value

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (pred(*iter) == val) count++;
  }
  return count;
}
```

```
auto is_less_than = [limit](int val) { return (val < limit) };
vector<int> v; count_occurrences(v.begin(), v.end(), is_less_than);
// counts the number of times a number under limit appears
```

Lambda syntax



Capture values

```
auto lambda = [capture-values](arguments) {
    return expression;
}
```

```
[x](arguments)      // captures x from surrounding scope by value
[x&](arguments)     // captures x from surrounding scope by reference
[x, y](arguments)   // captures x, y by value
[&](arguments)      // captures everything by reference
[&, x](arguments)   // captures everything except x by reference
[=](arguments)      // captures everything by copy
```

Algorithms & STL

Last time...

```
int count_occurrences(const vector<int>& vec, int val) {
  int count = 0;
  for (size_t i = 0; i < vec.size(); i++) {
    if (vec[i] == val) count++;
  }
  return count;
}

vector<int> v; count_occurrences(v, 5);
```

🤔 Making too many assumptions made our code non-portable.

With lambdas

```
template <typename InputIt, typename DataType, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred) {
  int count = 0;
  for (auto iter = begin; iter != end; ++iter) {
    if (pred(*iter) == val) count++;
  }
  return count;
}
```

Now the function is **maximally generic.**
(Question: Why do we use **`InputIt`** rather than the collection itself?)

Hey, this looks familiar...

std::count, std::count_if

Defined in header <algorithm>

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
    count( InputIt first, InputIt last, const T &value );                    (1)  (until C++20)
template< class InputIt, class T >
constexpr typename iterator_traits<InputIt>::difference_type
              count( InputIt first, InputIt last, const T &value );          (1)  (since C++20)

template< class ExecutionPolicy, class ForwardIt, class T >
typename iterator_traits<ForwardIt>::difference_type
    count( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, const T &value );   (2)  (since C++17)

template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type
    count_if( InputIt first, InputIt last, UnaryPredicate p );               (3)  (until C++20)
template< class InputIt, class UnaryPredicate >
constexpr typename iterator_traits<InputIt>::difference_type
              count_if( InputIt first, InputIt last, UnaryPredicate p );     (3)  (since C++20)

template< class ExecutionPolicy, class ForwardIt, class UnaryPredicate >
typename iterator_traits<ForwardIt>::difference_type
    count_if( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, UnaryPredicate p );   (4)  (since C++17)
```

Algorithms & STL

STL is a collection of generic template functions.

```
std::count_if(InputIt first, InputIt last, UnaryPredicate p);
```
*Counts the number of elements between **first** and **last** satisfying **p**.*

```
std::find(InputIt first, InputIt last, UnaryPredicate p);
```
*Finds the first element between **first** and **last** satisfying **p**.*

```
std::sort(RandomIt first, RandomIt last);
```
*Sorts the elements between **first** and **last**.*

STL functions operate on iterators.

```
std::minmax_element(InputIt first, InputIt last);
```

*Returns a **tuple** [min, max] over the elements between **first** and **last**.*

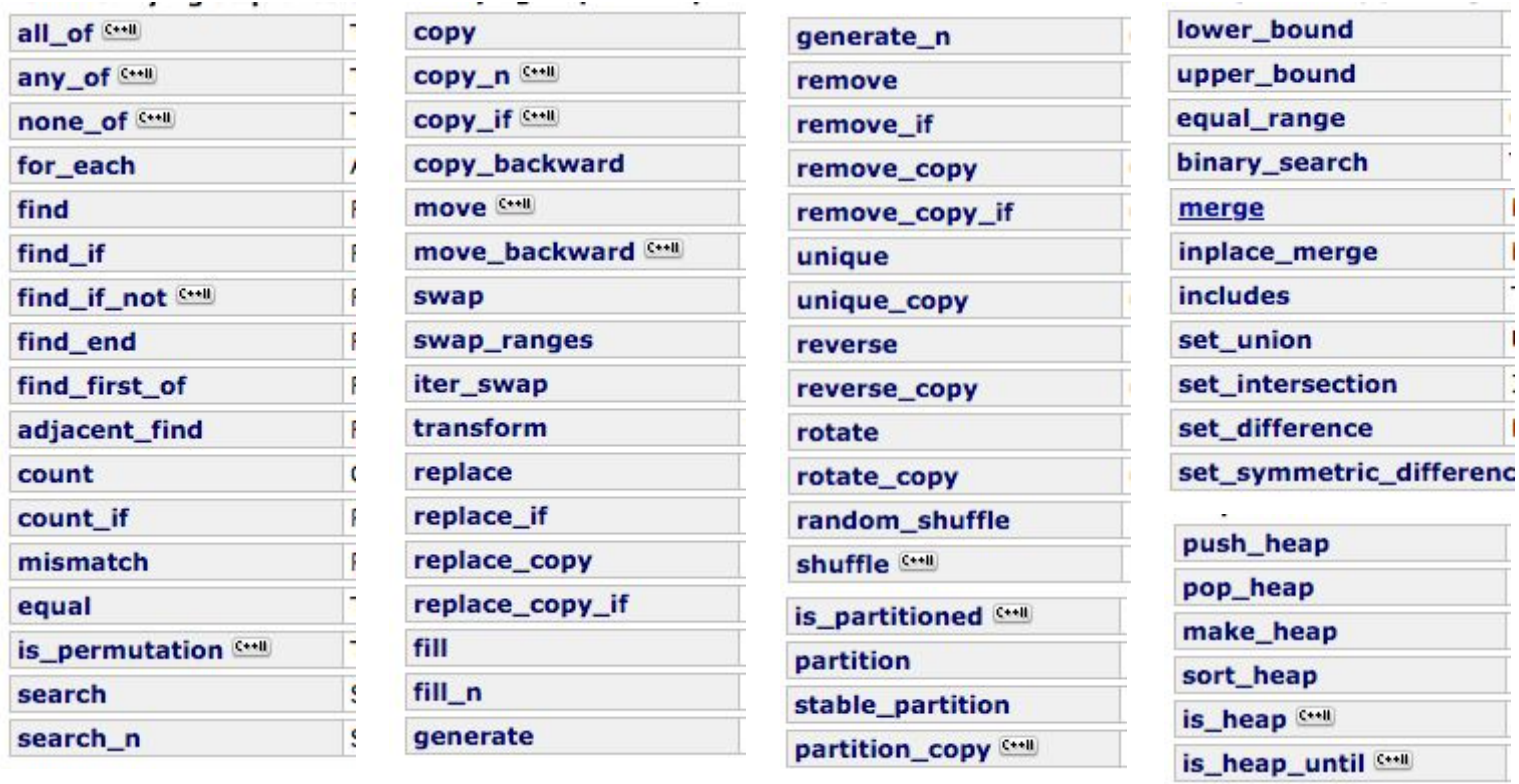
```
std::stable_partition(InputIt first, InputIt last, UnaryPredicate p);
```

*Reorders the elements between **first** and **last** such that all elements for which **p** returns true come before those for which it returns false.*

```
std::copy(InputIt first, InputIt last, OutputIt target);
```

*Copies the elements between **first** and **last** into **target.** (There's also a std::copy_if).*

There are a lot of algorithms...



Things you can do with the STL

binary search • heap building • min/max
lexicographical comparisons • merge • set union
set difference • set intersection • partition • sort
*n*th sorted element • shuffle • selective removal •
selective copy • for-each • move backwards

all in their most general form

Algorithms for Assn1

Element search with std::find

`std::find` finds the 1st instance of an element
or finds the 1st instance of an element satisfying a predicate **p**.
Returns an iterator to the element, or **.end()** if not.

```
std::vector<int> pi = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
auto it1 = std::find(pi.begin(), pi.end(), 9);
std::distance(pi.begin(), it1);                    // answer in chat
auto it2 = std::find(pi.begin(), pi.end(), 7);
std::distance(pi.begin(), it2);                    // answer in chat
auto it3 = std::find(pi.begin(), pi.end(), [](i) {
                                          return i % 3 == 2; });
std::distance(pi.begin(), it3);                    // answer in chat
cout << *it3 << endl;                              // answer in chat
```

Subsequence search with std::search

`std::search` finds the 1st instance of subsequence [**s_first, s_last**] in [**first, last**]. It returns an **iterator** to the occurrence in the main sequence, or **.end()** if it is not found.

```
std::vector<int> pi = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
std::vector<int> subseq = {1, 5, 9};

auto it = std::search(pi.begin(), pi.end(),
                      subseq.begin(), subseq.end());

std::distance(pi.begin(), it); // 3
```

Elegant evaluation with std::all_of

std::all_of(InputIt first, InputIt last, Pred p) returns a bool representing whether all of the elements between first and last satisfy **p.**

```
std::vector<int> values = {1, 3, 5, 7, 9, 11, 12};

bool val = std::all_of(values.begin(), values.end(),
                       [](i) { return i % 2 == 1;});
bool val2 = std::all_of(values.begin(), values.end() - 1,
                       [](i) { return i % 2 == 1;});

// what are val1 and val2? answer in chat
```


Questions?