Functions and Algorithms

CS 106L, Fall '21

How can we make use of template functions and containers?

Today's agenda

- Review of template functions
- Function pointers and lambdas
- Break for announcements!
- STL algorithms

How can we build functions that work regardless of input type?

Let’s write a function to count the number of times an element appears in a container.

Counting Occurrences: Attempt 1

```
int count_occurrences(std::string str, char target){
    int count = 0;
    for (size_t i = 0; i < str.size(); ++i){
        if (str[i] == target) count++;
    }
    return count;
}
```

Usage: `count_occurrences("Xadia", 'a');`

Counting Occurrences: Attempt 1

```
int count_occurrences(std::string str, char target){
    int count = 0;
    for (size_t i = 0; i < str.size(); ++i){
        if (str[i] == target) count++;
    }
    return count;
}

Usage: count_occurrences("Xadia", 'a');
```

We can’t reuse this for other datatypes, like when finding how many 3's are in a std::vector<int>.

Counting Occurrences: Attempt 2

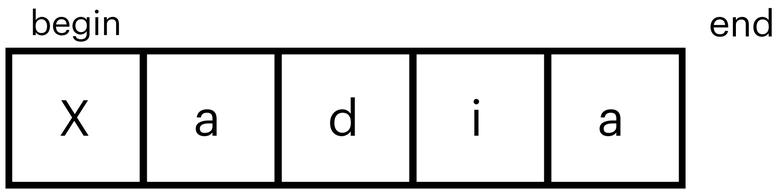
```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```

Counting Occurrences: Attempt 2

```
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;
}

Usage: std::string str = "Xadia";
       count_occurrences(str.begin(), str.end(), 'a');
```

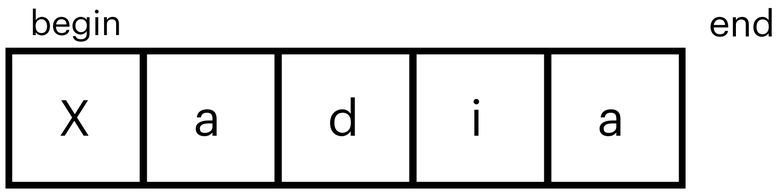


Counting Occurrences: Attempt 2

```
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;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```

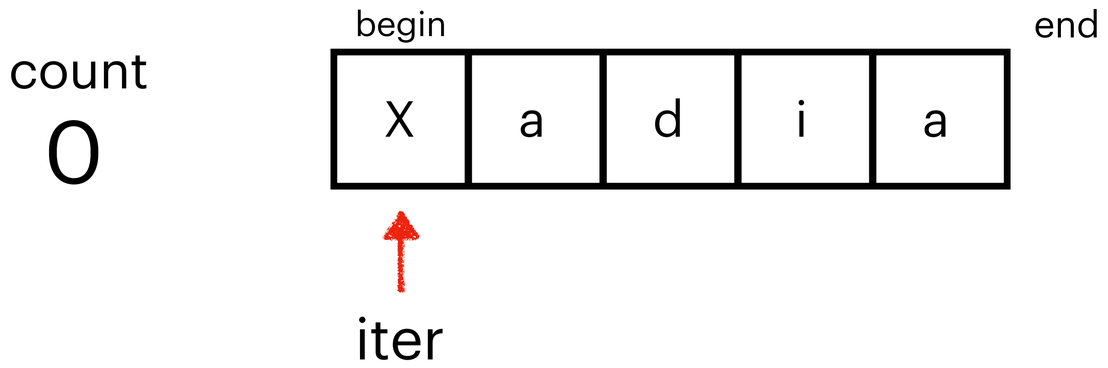
count
0



Counting Occurrences: Attempt 2

```
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;
}
```

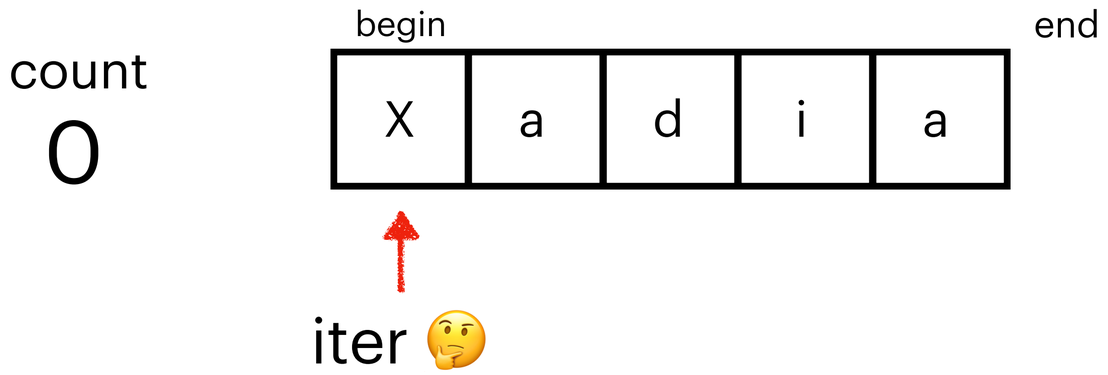
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

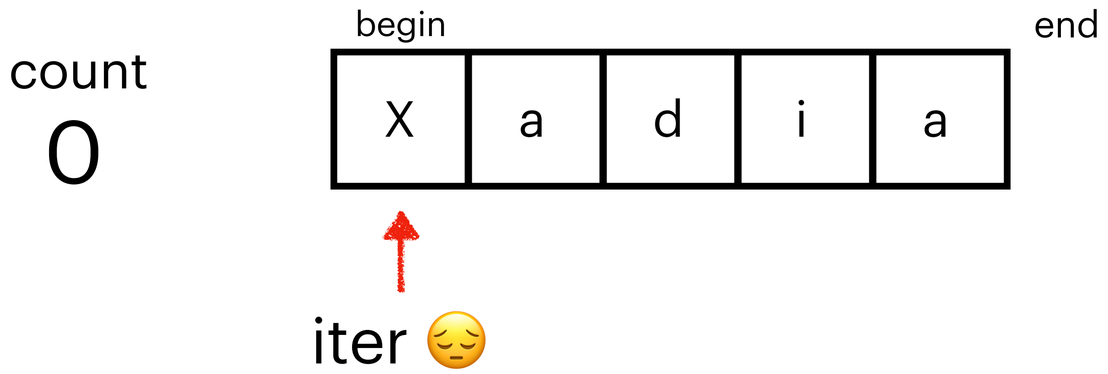
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

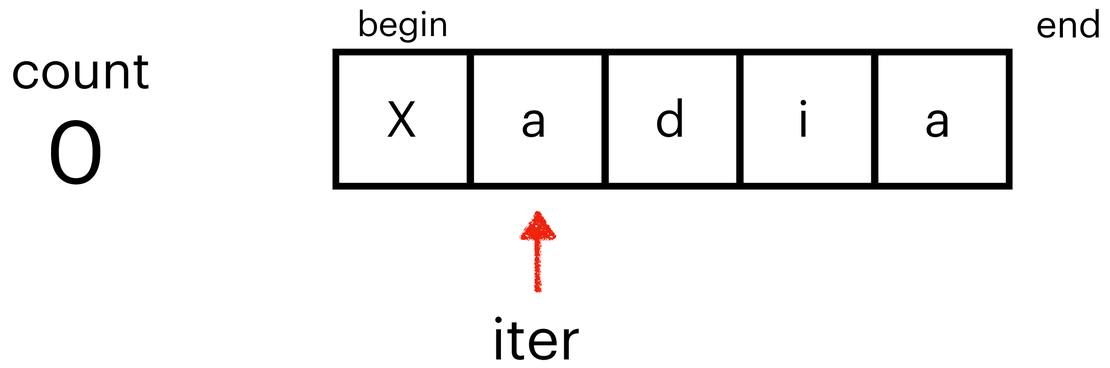
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

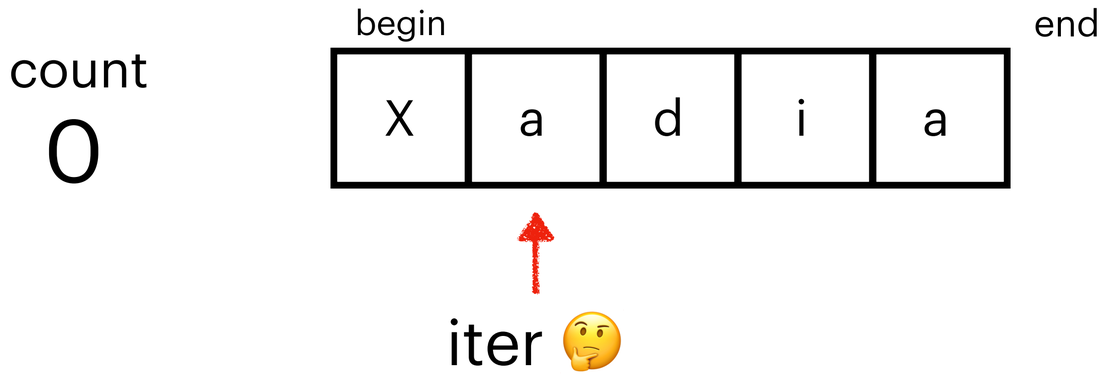
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

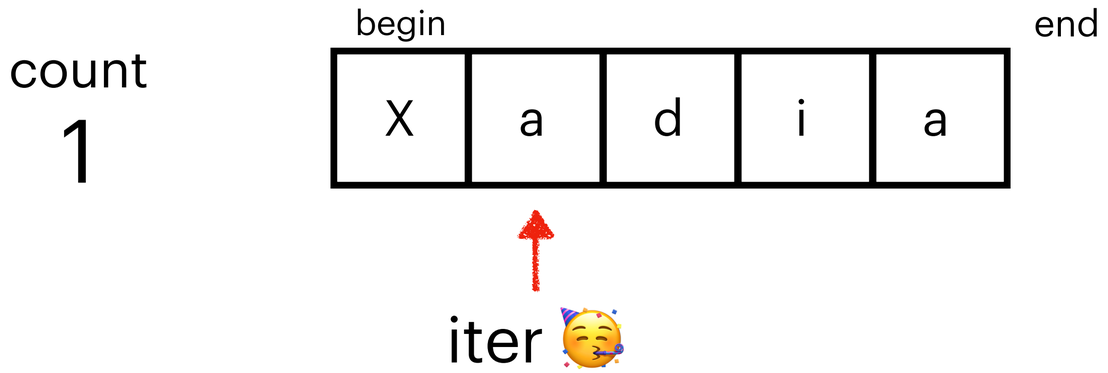
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

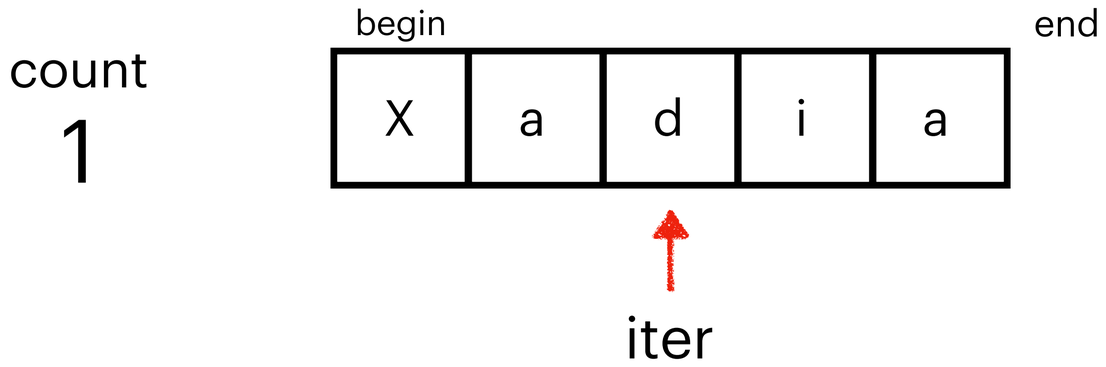
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

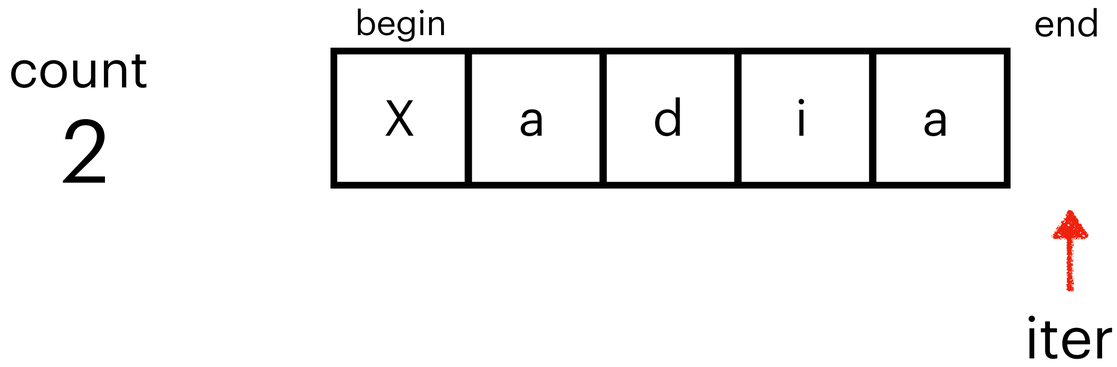
Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Counting Occurrences: Attempt 2

```
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;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Are we done?

```
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;
}

Usage: std::string str = "Xadia";
       count_occurrences(str.begin(), str.end(), 'a');
```

Could we reuse this to find how many vowels are in “Xadia”, or how many odd numbers were in a std::vector<int>?

What can we generalize to solve our problem?

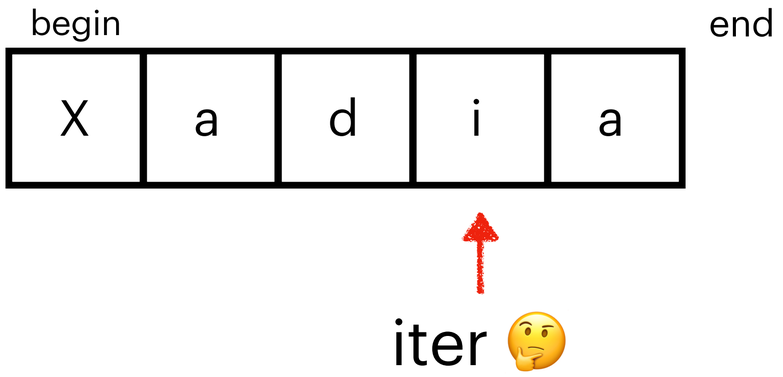
```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) count++;
    }
    return count;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```

What can we generalize to solve our problem?

```
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;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), 'a');
```



Function Pointers and Lambdas

Predicate Functions

Any function that returns a boolean is a predicate!

Unary Predicate

```
bool isLowercaseA(char c) {
    return c == 'a';
}

bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Binary Predicate

```
bool isMoreThan(int num, int limit) {
    return num > limit;
}

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

Function Pointers for generalization

```
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 (*iter == val pred(*iter)) count++;
    }
    return count;
}

bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), isVowel);
```

Function Pointers for generalization

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

bool isVowel(char c) {
    std::string vowels = "aeiou";
    return vowels.find(c) != std::string::npos;
}
```

Usage:
```
std::string str = "Xadia";
count_occurrences(str.begin(), str.end(), isVowel);
```

`isVowel` is a pointer, just like **`Node *`** or **`char *`**! It's called a "function pointer", and can be treated like a variable.

What's wrong with function pointers?

```
bool isMoreThan3(int num) {
    return num > 3;
}

bool isMoreThan4(int num) {
    return num > 4;
}

bool isMoreThan5(int num) {
    return num > 5;
}
```

Function pointers don't generalize well.

What's wrong with function pointers?

```
bool isMoreThan3(int num) {
    return num > 3;
}

bool isMoreThan4(int num) {
    return num > 4;
}

bool isMoreThan5(int num) {
    return num > 5;
}

// a generalized version of the above
bool isMoreThan(int num, int limit) {
    return num > limit;
}
```

Function pointers don't generalize well.

Can we use binary predicates?

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

Let's say we used binary predicates

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

What do we pass as the predicate's second argument? Can we pass in unary predicates anymore? How about 4-argument predicates?

We've run into a fundamental problem.

```
bool isMoreThan3(int num) {
    return num > 3;
}

bool isMoreThan4(int num) {
    return num > 4;
}

bool isMoreThan5(int num) {
    return num > 5;
}

bool isMoreThan(int num, int limit) {
    return num > limit;
}

bool isBetween(int num, int lower, int higher) {
    return lower < num && num < higher;
}
```

We need to give our unary predicate more information without adding another parameter.

Somehow, our function pointer has to know about "limit" without it being passed in as a parameter.

Can we create functions with more ways to input information than just parameters?

Introducing Lambdas!

Inline functions that can *know* about other variables.

what's really going on here?

Capture Clause
Outside variables your function uses

Parameters
You can use auto in lambda parameters!

```
auto var = [capture-clause] (auto param) -> bool
{
    ...
};
```

Body
Your function code goes here.

What exactly is the capture clause?

```
[]                  // captures nothing
[limit]             // captures lower by value
[&limit]            // captures lower by reference
[&limit, upper]     // captures lower by reference, higher by value
[&, limit]          // captures everything except lower by reference
[&]                 // captures everything by reference
[=]                 // captures everything by value


auto printNum = [] (int n) { std::cout << n << std::endl; };
printNum(5); // 5

int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
isMoreThan(6); // true
limit = 7;
isMoreThan(6);

int upper = 10;
auto setUpper = [&upper] () { upper = 6; };
```

Lambdas solve our earlier issue!

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

Usage:

```
int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
std::vector<int> nums = {3, 5, 6, 7, 9 ,13};

count_occurrences(nums.begin(), nums.end(), isMoreThan);
```

Are they expensive? And what *really* are they?

```
auto var = [capture-clause] (auto param) {
    ...
};
```

- Lambdas are cheap, but copying them may not be.
- Use lambdas when you need a short function, or one with read/write access to local variables!
- Use function pointers for longer logic and for overloading.
- We use “auto” because type is figured out in compile time.

Functors and Closures

```
class functor {
public:
    int operator() (int arg) const { // parameters and function body
        return num + arg;
    }
private:
    int num; // capture clause

};

int num = 0;
auto lambda = [&num] (int arg) { num += arg; };
lambda(5);
```

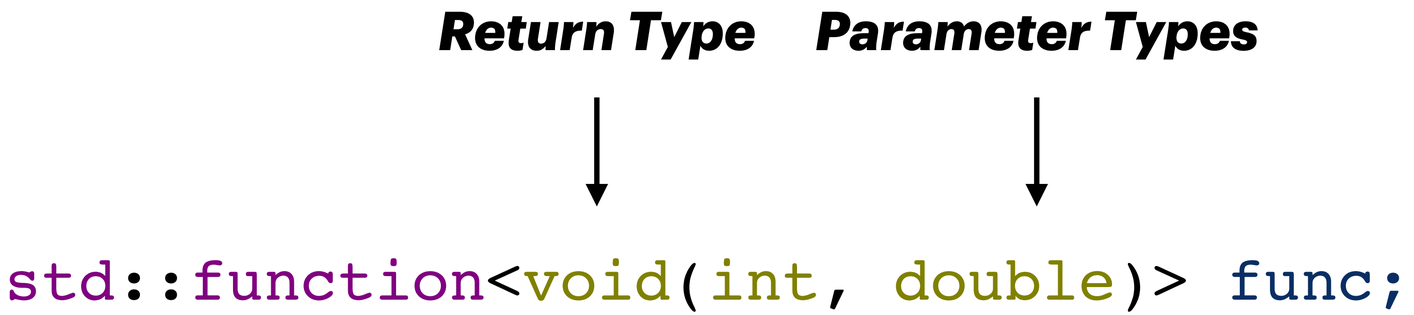
- A functor is any class that provides an implementation of `operator()`.
- Lambdas are essentially syntactic sugar for creating a functor.
- If lambdas are functor classes, then “closures” are instances of those classes.
- At runtime, closures are generated as instances of lambda classes.

How do functors, lambdas, and function pointers relate?

A standard function, `std::function<…>`, is the one to rule them all — it's the overarching type for anything callable in C++. Functors, lambdas, and function pointers can all be casted to standard functions.

How do functors, lambdas, and function pointers relate?

A standard function, `std::function<…>`, is the one to rule them all — it's the overarching type for anything callable in C++. Functors, lambdas, and function pointers can all be casted to standard functions.



How do functors, lambdas, and function pointers relate?

std::function is a complex,
heavy, expensive, magical type
that can hold any callable entity.

a wee function pointer

How do functors, lambdas, and function pointers relate?

```
void functionPointer (int arg) {
    int num = 0;
    num += arg;
}

// or

int num = 0;
auto lambda = [&num] (int arg) { num += arg; };
lambda(5); // num = 5;

std::function<void(int)> func = lambda;
```

We could cast either `functionPointer` or `lambda` to `func`, as both of them have a void return signature and take in one integer parameter.

Let’s take a pause for announcements!

Announcements

- Next lecture will be a recap of the class so far! If you haven't come to lecture, we encourage you to!
- This the last lecture of material that'll be on assignment 2.
- Assignment 2 has been out for a week, and will be due at the end of week 6 (next Friday)!
  - If you need any help, stop by our office hours!

Building `count_occurrence` was fun, but took a bit of time!

Is there a library where people have implemented commonly used, generic functions?

Introducing STL Algorithms!

A collection of ***completely generic*** functions written by C++ devs!

```
#include <algorithm>
```

sort • reverse • min_element • max_element • binary_search • stable_partition • find • find_if • count_if • copy • transform • insert • for_each • etc.!

STL Algorithms operate on iterators

As an example, let's say we had a vector, nums.

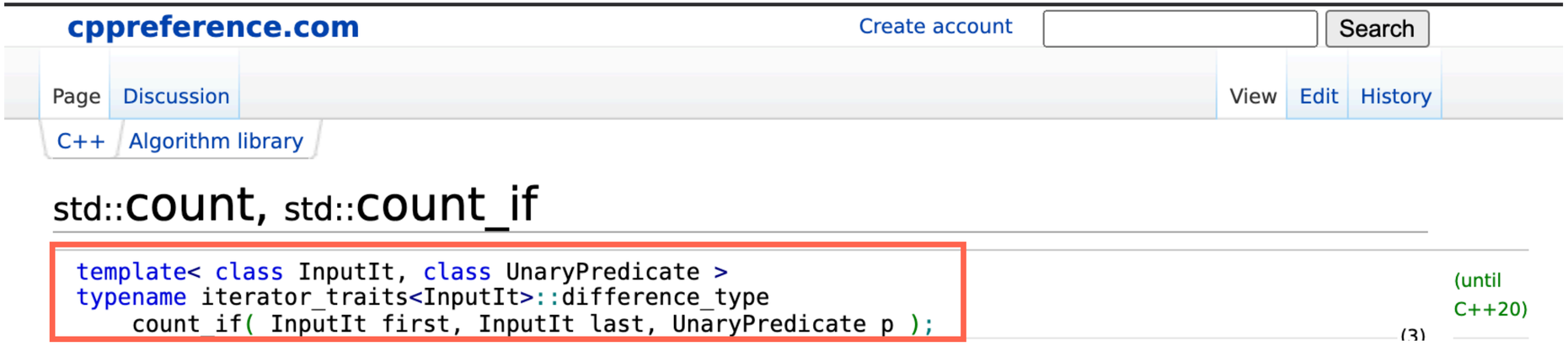
first iterator
Points to the first element of data structure

last iterator
Points to the end of the data structure

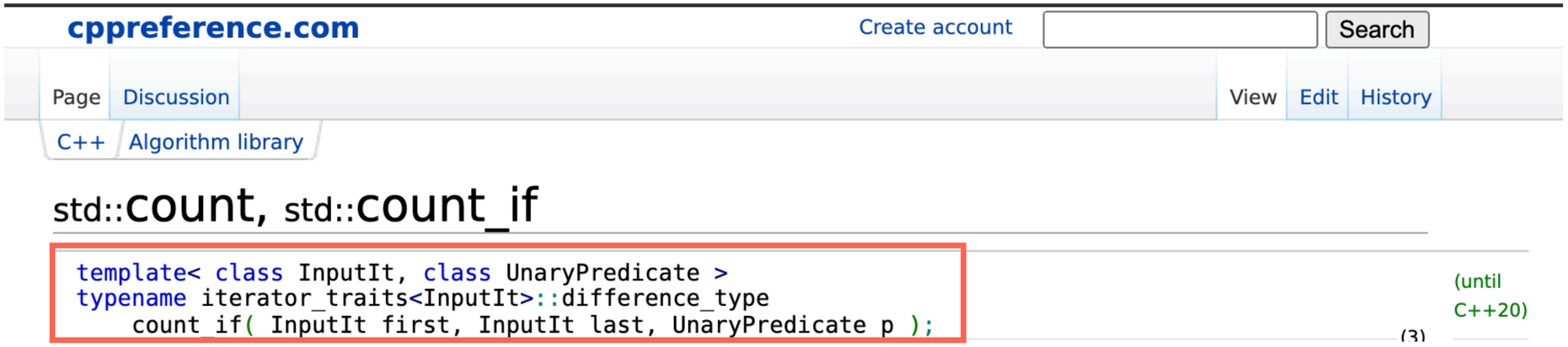
```
std::count_if(nums.begin(), nums.end(), isBetween);
```


Function Pointer, Lambda, etc.
Like the one we've written!

Google STL Functions to read their documentation



Google STL Functions to read their documentation



This looks familiar, doesn't it?

```
template <typename InputIt, typename UniPred>
int count_occurrences(InputIt begin, InputIt end, UniPred pred);
```

Demo! Let's say you wanted to find an algorithm that would let you separate a vector based on some condition.

For instance, let's try to separate a **`std::vector<int>`** into odd and even portions.

STL Functions allow you to accomplish all of these and more!

effective searching • efficient sorting • complex and concise data structure usage • parallelized big data manipulation • arrays with abilities • smart pointers that destruct memory for you!

all are provided to you in their most generic form!