http://web.stanford.edu/class/cs106l/

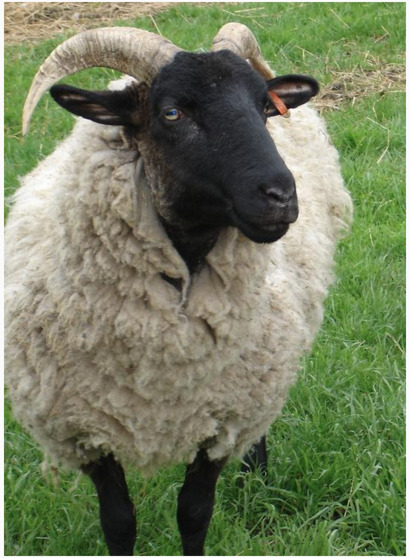
Functions and Lambdas

How can we make template functions even more general?

CS106L - Fall 22

Attendance!

https://bit.ly/3svpQPX



Announcements!

- No class **next Tuesday** – midquarter break!
  - Office hours during class time (3-4:30pm)
  - Review material from the lectures so far!
- No class **Nov 8** – Democracy Day!
- Assignment 2 is due Nov. 27th
- Grades for assignment 1 released soon!

Functions and Lambdas

http://web.stanford.edu/class/cs106l/

CONTENTS

Functions and Lambdas

http://web.stanford.edu/class/cs106l/

CONTENTS

Template functions are completely generic functions!

Just like classes, they work regardless of type!

Let's break it down:

Indicating this function is a template

Specifies that Type is generic

List of your template variables

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

Calling template functions

We can **explicitly** define what type we will pass, like this:

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}

// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin<int>(3, 4) << endl; // 3
```

Just like in template classes!

Calling template functions

We can also **implicitly** leave it for the compiler to deduce!

```
template <typename T, typename U>
auto smarterMyMin(T a, U b) {
  return a < b ? a : b;
}

// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin(3.2, 4) << endl; // 3.2
```

Review: Template Functions

- Template functions allow you to parametrize the type of a function to be anything without changing functionality
- Generic programming can solve a complicated conceptual problem for any specifics – powerful and flexible!
- Template code is instantiated at compile time; template metaprogramming takes advantage of this to run code at compile time

Functions and Lambdas

http://web.stanford.edu/class/cs106l/

CONTENTS

Let’s review that count_occurrences function!

This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

```
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');
```

This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
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');
```

This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
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 if we wanted to find all the vowels in "Xadia"?

This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

```
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');
```

This is a successfully templated function!

This code will work for any containers with any types, for a single specific target.

Will this work for a more general category of targets than one specific value?

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

isVowel(*iter) ?

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

Predicate Functions

Any function that returns a boolean value is a predicate!

Predicate Functions

Any function that returns a boolean value is a predicate!

- isVowel() is an example of a predicate, but there are tons of others we might want!

Predicate Functions

Any function that returns a boolean value is a predicate!

- isVowel() is an example of a predicate, but there are tons of others we might want!

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

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

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

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

Predicate Functions

Any function that returns a boolean value is a predicate!

- isVowel() is an example of a predicate, but there are tons of others we might want!
- A predicate can have any amount of parameters…

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

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

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

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

Predicate Functions

Any function that returns a boolean value is a predicate!

- isVowel() is an example of a predicate, but there are tons of others we might want!
- A predicate can have any amount of parameters…

Unary

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

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

Binary

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

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

Let’s use that!

```
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);
```

Let’s use that!

```
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);
```

Functions and Lambdas

http://web.stanford.edu/class/cs106l/

Let’s use that!

```
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);
```

Let's use that!

```
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);
```

What type is UniPred???

Let's use that!

```
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) {
        i
    }
    retur
}

bool isVow
    std::s
    retur
}
```

Usage: `std`
`cou` … `vel);`

What type is UniPred???

Function Pointers

UniPred is what's called a **function pointer!**

Function Pointers

UniPred is what's called a **function pointer!**

- Function pointers can be treated just like other pointers

Function Pointers

UniPred is what's called a **function pointer!**

- Function pointers can be treated just like other pointers
- They can be passed around like variables as parameters or in template functions!

Function Pointers

UniPred is what's called a **function pointer!**

- Function pointers can be treated just like other pointers
- They can be passed around like variables as parameters or in template functions!
- They can be called like functions!

Is this good enough?

Are there any ways this could be an issue?

```
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);
```

Poor Generalization

Unary predicates are pretty limited and don't generalize well.

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

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

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

Poor Generalization

Unary predicates are pretty limited and don't generalize well.

Ideally, we'd like something like this!

```
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;
}
```

Can we use binary predicates?

If we could, it would be nice to use a binary predicate to handle this!

```
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;
}
```

Can we use binary predicates?

How do we know what value to use? What about unary (or any other number of arguments) 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;
}
```

We can't pass this in from the predicate!

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

The Catch-22

We want our function to know more information about our predicate.

The Catch-22

We want our function to know more information about our predicate.

However, we can't pass in more than one parameter.

The Catch-22

We want our function to know more information about our predicate.

However, we can't pass in more than one parameter.

How can we pass along information without needing another parameter?

Let's use lambdas!

Lambdas are inline, anonymous functions that can know about functions declared in their same scope!

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

Let's use lambdas!

Lambdas are **inline**, anonymous functions that can know about variables declared in their same scope!

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

Let's use lambdas!

Lambdas are inline, **anonymous** functions that can know about variables declared in their same scope!

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

Let’s use lambdas!

Lambdas are inline, **anonymous** functions that can know about variables declared in their same scope!

Specifies that Type is generic

Outside parameters go here

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

Function body goes here!

Let’s use lambdas!

It might look something like this!

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

Let’s use lambdas!

It might look something like this!

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

Capture Clauses

You can capture any outside variable, both by reference and by value.

```
[]                  // 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
```

Capture Clauses

You can capture any outside variable, both by reference and by value.

- Use just the = symbol to capture everything by value, and just the & symbol to capture everything by reference

```
[]                  // 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
```

We've solved our problem!

```
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);
```

We've solved our problem!

```
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);
```

Using Lambdas

Lambdas are pretty computationally cheap and a great tool!

- Use a lambda when you need a short function or to access local variables in your function.
- If you need more logic or overloading, use function pointers.

Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

```
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);
```

Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

- They can create **closures** of "customized" functions!

```
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);
```

Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

- They can create **closures** of "customized" functions!

```
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);
```

Closure: a single instantiation of a functor object

Aside: What the Functor?

A **functor** is any class that provides an implementation of operator().

- They can create **closures** of "customized" functions!
- Lambdas are just a reskin of functors!

```
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);
```

Closure: a single instantiation of a functor object

Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

The STL has an overarching, standard function object!

```
std::function<return_type(param_types)> func;
```

Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

The STL has an overarching, standard function object!

```
std::function<return_type(param_types)> func;
```

Everything (lambdas, functors, function pointers) can be cast to a standard function!

Tying it all together

So far, we've talked about lambdas, functors, and function pointers.

The STL has an overarching, standard function object!

```
std::function<return_type(param_types)> func;
```

Everything (lambdas, functors, function pointers) can be cast to a standard function!

Much bigger and more expensive than a function pointer or lambda!

Functions and Lambdas

http://web.stanford.edu/class/cs106l/

CONTENTS

Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

1. Look both ways before crossing the street.

Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

1. Look both ways before crossing the street.
2. Never tell a pre-med you're stressed.

Coding Philosophy 101

There are few universal, scientifically proven pieces of wisdom that will lead to a happier life:

1. Look both ways before crossing the street.
2. Never tell a pre-med you're stressed.
3. When coding, never reinvent the wheel.

New toys!

The STL implements an entire library of algorithms written by C++ developers!

New toys!

The STL implements an entire library of algorithms written by C++ developers!

- To utilize, `#include <algorithm>` in your file!

New toys!

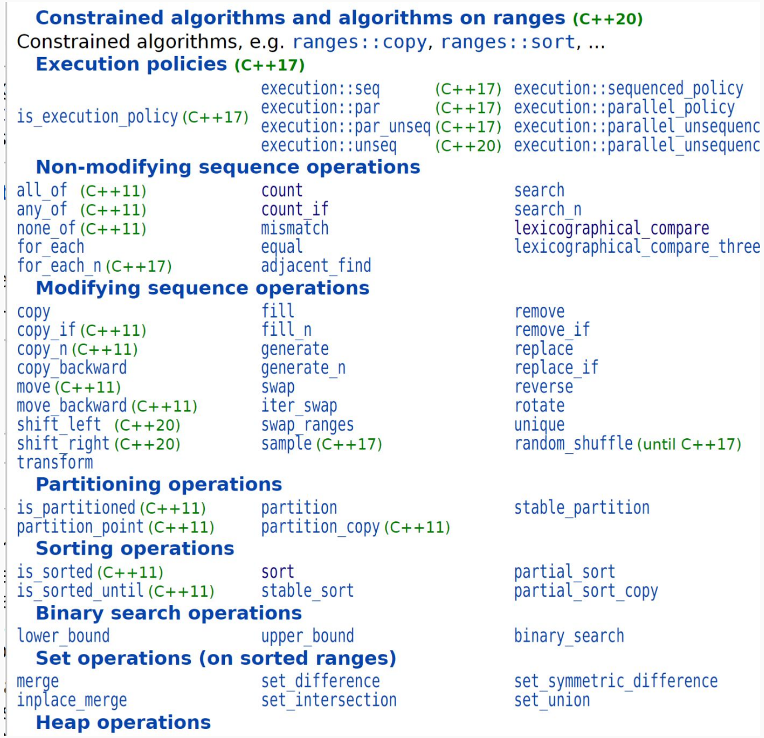
The STL implements an entire library of algorithms written by C++ developers!

- To utilize, `#include <algorithm>` in your file!
- All algorithms are **fully generic**, **templated** functions!

New toys!

The STL implements an entire library of algorithms written by C++ developers!

- To utilize, `#include <algorithm>` in your file!
- All algorithms are **fully generic**, **templated** functions!



Look familiar?

count_occurrences

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

std::count_if

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
    count( InputIt first, InputIt last, const T& value );
```

Algorithms

All standard algorithms work on iterators.

- Efficient searching, sorting, complex data structure operations, smart pointers, and more are all there for you to use!
- Check out the documentation to get more information!

Template Functions

http://web.stanford.edu/class/cs106l/

Let’s check it out!

Summary

- Lambda functions are inline functions that let you pass outside variables in using capture clauses!
- Lambdas can be used to pass predicate function pointers to template functions for more generalizability.
- The STL implements tons of cool algorithms that we can use without rewriting them!

Thanks!

Next up: Operators!