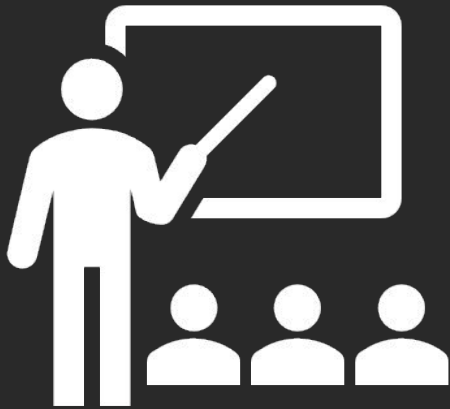


# Final Lecture

CS 106L Spring 2020 – Avery Wang and Anna Zeng  
Stanford University

# Game Plan



- special topic!
- announcements
- Zoom selfie
- C++'s greatest hits
- where to go next

# And our special topic is...

With 52% of respondents voting for this topic

# Lambda Implementation!

~short 20 minute intro

# Why were lambdas important again?

Hint: it was during the algorithms lecture...

# Concept Lifting

Looking at the assumptions you place on the parameters and questioning if they are really necessary.

Can you solve a more general problem by relaxing the constraints?

How many times does the element satisfy  
[predicate] in [a range of elements]?

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

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

```
// Unary Predicate (one argument)
bool is_equal_to_3(int val) {
    return val == 3;
}
```

```
// Binary Predicate (two arguments)
bool is_divisible_by(int dividend, int divisor) {
    return dividend % divisor == 0;
}
```



We can then call this function with a predicate.

```
bool is_less_than_5(int val) {  
    return val < 5;  
}  
  
int main() {  
    vector<int> vec{1, 3, 5, 7, 9};  
  
    count_occurrences(vec.begin(), vec.end(), is_less_than_5);  
    // prints 2  
}
```

# Old approach: function pointers

```
bool is_less_than_limit(int val) {  
    return val < limit; // compiler error!  
}
```

```
int main() {  
    vector<int> vec{1, 3, 5, 7, 9};  
    int limit = 5;  
  
    count_occurrences(vec.begin(), vec.end(),  
is_less_than_limit);  
    return 0;  
}
```

# New approach: lambda functions

```
bool is_less_than_limit(int val) {  
    return val < limit;  
}
```

```
int main() {  
    vector<int> vec{1, 3, 5, 7, 9};  
    int limit = 5;  
    auto is_less_than_limit = [limit](auto val) {  
        return val < limit;  
    };  
    count_occurrences(vec.begin(), vec.end(),  
is_less_than_limit);  
    return 0;  
}
```

# New approach: lambda functions

We don't know the type, ask compiler.

capture clause,  
gives access to outside  
variables

parameter list,  
can use auto!

return type,  
optional

```
auto is_less_than_limit = [limit](auto val) -> bool {  
    return val < limit;  
};
```

Accessible variables inside lambda limited to  
capture clause and parameter list.

# Doesn't this seem a little odd?

We don't know the  
type, ask compiler.



```
auto is_less_than_limit = [limit](auto val) -> bool {  
    return val < limit;  
};
```

# Doesn't this seem a little odd?

We don't know the  
type, ask compiler.



```
auto is_less_than_limit = [limit](auto val) -> bool {  
    return val < limit;  
};
```

Our previous explanation:  
compiler creates a class whose name you don't know.

# Doesn't this seem a little odd?

What exactly is this thing?



```
auto is_less_than_limit = [limit](auto val) -> bool {  
    return val < limit;  
};
```

- It is an object.
- You can use it as if it is a function.
  - You use the parentheses operator
  - Put the parameters in the parentheses, the object returns a return value

# Fun math example:

```
auto linear_function = [](int x) -> int {  
    return 4 * x + 2;  
};
```


Can you design a class whose objects perform the above function?



# Overloading the parenthesis operator

Can be anything!

Whatever you want your function object to accept as parameter or return as output.



```
[return value] ClassName::operator() (parameters) {  
    // function body  
}
```

# Overloading the parenthesis operator

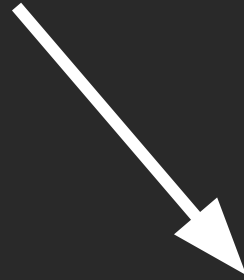
Function call operator must be overloaded as a member function within a class.



```
[return value] ClassName::operator() (parameters) {  
    // function body  
}
```

# Overloading the parenthesis operator

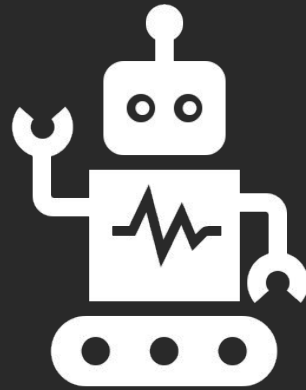
This first parenthesis specifies that we are overloading the parentheses operator



```
[return value] ClassName::operator() (parameters) {  
    // function body  
}
```



This second parenthesis is where we put the parameters.



# Example

Simple Example: linear equation objects

# Fun math example:

```
auto linear_function = [](int x) -> int {  
    return 4 * x + 2;  
};
```

```
class LinearFunction {  
    LinearFunction();  
    int operator()(int x);  
};
```

```
int LinearFunction::operator()(int x) {  
    return 4 * x + 2;  
}
```

# How can we use this class?

```
LinearFunction linear_function;  
cout << linear_function(3) << endl; // returns 14
```

# How can we use this class?

Declare an instance (object) of the class  
we just wrote.



```
LinearFunction linear_function;  
cout << linear_function(3) << endl; // returns 14
```



Call that object's parenthesis operator!

# How can we use this class?

```
// lambda approach
auto linear_function = [](int x) -> int {
    return 4 * x + 2;
};
cout << linear_function(3) << endl; // returns 14
```

```
// function object approach
LinearFunction linear_function;
cout << linear_function(3) << endl; // returns 14
```



# A little tougher math example...

```
int a = 4, b = 2;  
auto linear_function = [a, b](int x) -> int {  
    return a * x + b;  
};  
cout << linear_function(3) << endl; // returns 14
```

How would your class be able to store the variables that normally went into the capture clause?

# How do we capture variables?

```
class LinearFunction {  
    LinearFunction();  
    int operator() (int x);  
};
```

```
int LinearFunction::operator() (int x) {  
    return a * x + b;  
}
```



What variables are accessible in the member function?

# How do we capture variables?

```
class LinearFunction {  
    LinearFunction();  
    int operator() (int x);  
};
```

```
int LinearFunction::operator() (int x) {  
    return a * x + b;  
}
```



What variables are accessible in the member function?

- parameters (but can't change those)
- global variables (bad!)
- private member variables (!)

# How do we capture variables?

```
class LinearFunction {  
public:  
    LinearFunction();  
    int operator() (int x);  
private:  
    int a;  
    int b;  
};  
  
int LinearFunction::operator() (int x) {  
    return a * x + b;  
}
```

# How do we capture variables?

```
class LinearFunction {  
public:  
    LinearFunction();  
    int operator() (int x);  
private:  
    int a;  
    int b;  
};
```

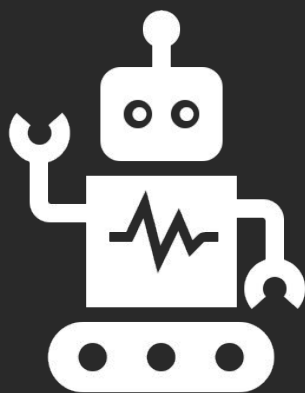


Where would you initialize these?

```
int LinearFunction::operator() (int x) {  
    return a * x + b;  
}
```

# How do we capture variables?

```
class LinearFunction {  
public:  
    LinearFunction(int a, int b);  
    int operator() (int x);  
private:  
    int a;  
    int b;  
};  
  
int LinearFunction::operator() (int x) {  
    return a * x + b;  
}
```

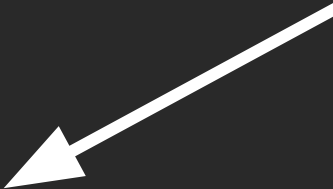


# Example

Advanced Example:  
linear equation objects with a capture

# How can we use this class?

Declare an instance (object) of the class we just wrote, **initializing the private variables.**



```
LinearFunction linear_function{4, 2};  
cout << linear_function(3) << endl; // returns 14
```



Call that object's parenthesis operator, which uses the stored information **in the private variables.**



# More member functions!

```
class LinearFunction {  
public:  
    LinearFunction();  
    int operator() (int x);  
    void verticalStretch (int scale);  
    friend LinearFunction operator+ (LinearFunction lhs,  
                                     LinearFunction rhs);  
  
private:  
    int a;  
    int b;  
};
```



You can add other member functions as well besides the operator().

# Can you...?

Make it into a template function	YES!
Overload with different parameters	YES!
Use default parameters	YES!
Make it into a variadic template	YES!

# Lambdas vs. Function Classes

Lambdas are really to write. Capturing variables is really easy, and auto hides a lot of the complexities.

Every time you create a lambda, the compiler generates a class just to create one single instance (object) of that class.

Available in C++11 only!

# Lambdas vs. Function Classes

Function classes are really flexible. Every technique from class design can be applied to function classes.

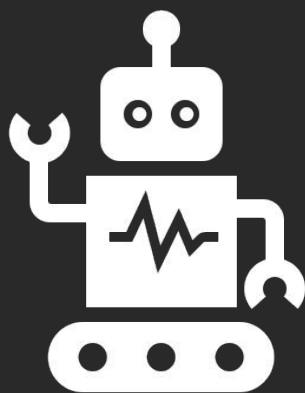
You can write one class, and use many instances of that same class.

Lots of code. You are declaring an entire class just to use a function object. Also not really beginner friendly.

# The Verdict

Use lambdas if you need a one-off function object to be passed into a function. Example: make a call to an STL algorithm.

Write a function class if you need many instances of the same function object, and need anything more complex (e.g. need to keep track of state).



# Example

A real example of functors: using the STL algorithms again!

# Announcements

# Announcements

- Assignment 2 is due this Sunday, but you may use late days up to Wednesday.
- Office hours + fairly fast responses on Piazza (you can type out your code and we'll debug it!)
- We will try to be as accommodating as possible.
  - If you are spending more than 10 hours on milestones 1 and 2 (purely on coding), shoot us an email. We'll work something out.
  - We've made many videos which get you pretty close to the correct answers. Watch them!



# Announcements

- We will release two more projects. They are open-ended design problems, and they come with an extensive testing harness + test suite.
- These are significantly more challenging than the HashMap assignment.
- They are completely optional, and there is no due date.
- Feel free to contact us over the summer if you want to talk about the project.

# Announcements

- Fill out course reviews!
- The future of CS 106L (whether or not it will be offered in the future), as well as our teaching careers, depend on your feedback and comments.

# Announcements

- This is our last class! We've loved having you all as students, and loved getting to know you throughout the quarter. :)
- Keep in touch with us after 106L - we'll gladly still answer questions about C++, CS, Stanford, life, etc.
  - Avery: [averywang@stanford.edu](mailto:averywang@stanford.edu)
  - Anna: [aszeng@stanford.edu](mailto:aszeng@stanford.edu)

# Announcements

- Connect with us on LinkedIn!
  - [Avery](#)
  - [Anna](#)
- Subscribe to our [YouTube channel](#)
  - See past recorded lectures of CS 106L in Fall 2019
  - In case we make more C++ videos (CS 106LX?)

# Zoom Selfie!

With 44% of respondents voting for this topic, the next-highest topic was...

With 44% of respondents voting for this topic, the next-highest topic was...

## *Modern C++'s Greatest Hits!*

Credits: Jason Turner, CppCon 2019, "The Best Parts of C++"

Link: <https://www.youtube.com/watch?v=iz5Qx18H6lg>

With 44% of respondents voting for this topic, the next-highest topic was...

# *Modern C++'s Greatest Hits!*

(A Speed-Run)

Credits: Jason Turner, CppCon 2019, "The Best Parts of C++"

Link: <https://www.youtube.com/watch?v=iz5Qx18H6lg>



# There are many things we take for granted...

1. Having a C++ **standard**
2. **Const** - enforcing that certain objects aren't modified
3. **Object destruction** at end of scope
4. **Templates** whose types are generated at compile-time
5. The **Standard Template Library**: the ultimate generic set of composable tools (algorithms, containers)

(all C++98)

# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

Phew!

# ...and so much more!

(Year Added: C++11, C++14, C++17, C++20)

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

Phew!

# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

Phew!



# ...and so much more!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)

# Let's go!

6. `std::array` (fixed-size stack-based container)
7. List/uniform initialization (ex. `std::vector<int> v{1, 2, 3};`)
8. Variadic templates (allowing a variable number of parameters!)
9. `constexpr` (compile-time generation of code and data)
10. `auto` 😊
11. Return type deduction for functions
12. Lambdas! (unnamed function objects)
13. Generic and variadic lambdas
14. range-based for loop (aka: for-each loop)
15. Structured bindings (ex. `const auto& [key, val] = *map.begin();`)
16. Concepts (specify type requirements)
17. `std::string_view` (use C strings without allocating more memory)
18. Text formatting! (similar to Python formatting options)
19. Ranges! (similar to Python comprehensions)
20. Class template argument deduction! (ex. `std::vector vec{1, 2, 3};`)
21. rvalue references and move semantics (using knowledge of object lifetimes)
22. Guaranteed copy elision
23. Defaulted and deleted functions (prevent misuse of your class)
24. `std::unique_ptr` and `std::make_unique`
25. Fold expressions (for common operators in variadic expansions)


Phew!

# #8: Variadic Templates

Allows a variable number of parameters in templated functions!

```
template<typename ... Types> void f(Types ... args);  
f();           // OK: args contains no arguments  
f(1);          // OK: args contains one argument: int  
f(2, 1.0);     // OK: args contains two arguments: int and double
```

Parameter pack:  
0 or more types.




# #8: Variadic Templates

Allows a variable number of parameters in templated functions!

```
template<typename ... Types> void f(Types ... args);  
f();           // OK: args contains no arguments  
f(1);          // OK: args contains one argument: int  
f(2, 1.0);     // OK: args contains two arguments: int and double
```

Parameter pack:  
0 or more types.



When is this useful? A lot of places!

- Implementing printf() (i.e. printing different types)
- Tuples
- Inheritance
- Perfect forwarding

```
template<class ... Ts> struct Tuple {};  
Tuple<> t0;           // Ts = no args  
Tuple<int> t1;         // Ts = int  
Tuple<int, float> t2;  // Ts = int, float  
Tuple<0> error;        // error: 0 isn't a type
```

See Lecture 7 (Templates) Supplemental Material for a longer version of this!

# #9: constexpr

Compile-time generation of code and data.

```
const double PI = 3.141593;           // Is PI known at compile-time?
int main() {
    const double radius = 1.5;
    const double area = PI * radius * radius;
    std::cout << area;
}
```

# #9: constexpr

Compile-time generation of code and data.

```
constexpr double PI = 3.141593;      // Compiler will replace PI with its value!*
int main() {
    const double radius = 1.5;
    const double area = PI * radius * radius;
    std::cout << area;
}
```

No! But we can ask the compiler to generate the value at compile-time, using constexpr.

\* Different than preprocessor macros!

# #9: constexpr

Compile-time generation of code and data.

```
constexpr double PI = 3.141593;      // Compiler will replace PI with its value!*
int main() {
    const double radius = 1.5;
    const double area = PI * radius * radius;
    std::cout << area;
}
```

No! But we can ask the compiler to generate the value at compile-time, using constexpr.  
...And in fact, we can even generate the value of PI on the fly, all at compile-time!

```
constexpr double calculate_pi() {
    return 22.0/7; // or something more clever
}
constexpr double pi = calculate_pi();
```

\* Different than preprocessor macros!

# #13: Generic and Variadic Lambdas

Lambdas can be variadic as well - both in the capture clause and in the parameters!

Parameter pack may appear in the capture clause:

```
template<class ...Args>
void f(Args... args) {
    auto lambda = [&, args...]() { return g(args...); };
    // use the lambda
}
```

Or in the parameters list:

```
auto lambda = [/*captures*/](auto ... params){
    return std::vector<int>{params...};
};
```



# #17: std::string\_view

Allows you to use C strings without converting to a std::string (which allocates memory)!

```
std::string_view sv{some_string_like_thing}; // no copy
```

Can be much faster, especially with large strings! See [here](#) for the full example:

```
int main() {  
    std::string large = "0123456789-123456789-123456789-123456789";  
    std::string substr = large.substr(10);  
  
    std::string_view largeStringView{large.c_str(), large.size()};  
    largeStringView.remove_prefix(10);  
  
    assert(substr == largeStringView);  
}
```

# #18: Text Formatting

Allows for smarter formatting of output! Closer to languages like Python.

Currently exists in the C++ `{fmt}` library, but soon to be added officially in C++20.

```
std::string s = fmt::format("I'd rather be {1} than {0}.", "right", "happy");  
// s == "I'd rather be happy than right."
```

If you can't wait to start outputting with text formatting, you can download the fmt library [here!](#)

# #19: Ranges

This is crazy... Let's simplify our algorithms even further! Similar to Python list comprehensions.

```
int main()
{
    std::vector<int> ints{0,1,2,3,4,5};
    auto even = [](int& i){ return 0 == i % 2; };
    auto square = [](int& i) { return i * i; };

    std::vector<int> res;
    // Filter out only the even values
    std::copy_if(ints.begin(), ints.end(), std::back_inserter(res), even);
    // Square each of the remaining values
    std::for_each(res.begin(), res.end(), square);
    // Print out our new values
    std::copy(res.begin(), res.end(), std::ostream_iterator<int>(std::cout, " "));
}
```

# #19: Ranges

This is crazy... Let's simplify our algorithms even further! Similar to Python list comprehensions.

```
int main()
{
    std::vector<int> ints{0,1,2,3,4,5};
    auto even = [](int& i){ return 0 == i % 2; };
    auto square = [](int& i) { return i * i; };

    std::vector<int> res;
    // Filter out only the even values
    std::copy_if(ints.begin(), ints.end(), std::back_inserter(res), even);
    // Square each of the remaining values
    std::for_each(res.begin(), res.end(), square);
    // Print out our new values
    std::copy(res.begin(), res.end(), std::ostream_iterator<int>(std::cout, " "));
}
```

We can replace this entirely using ranges (after C++20)!

# #19: Ranges

This is crazy... Let's simplify our algorithms even further! Similar to Python list comprehensions.

```
int main()
{
    std::vector<int> ints{0,1,2,3,4,5};
    auto even = [](int& i){ return 0 == i % 2; };
    auto square = [](int& i) { return i * i; };

    // Using ranges!
    for(int i : ints | std::views::filter(even) | std::views::transform(square)){
        std::cout << i << ' ';
    }
}
```

# #20: Class Template Argument Deduction

A new acronym: CTAD!

Just like how function template arguments can be deduced, class templates can be deduced as well!

```
std::vector vec{1,2,3};    // automatically deduces that this is vector<int>!
```

# #20: Class Template Argument Deduction

A new acronym: CTAD!

Just like how function template arguments can be deduced, class templates can be deduced as well!

```
std::vector vec{1,2,3};    // automatically deduces that this is vector<int>!
```

This applies to everything we've seen so far!

```
std::pair p(2, 4.5);       // deduces to std::pair<int, double>
std::tuple t(4, 3, 2.5);   // same as auto t = std::make_tuple(4, 3, 2.5);
auto lck = std::lock_guard(mtx); // deduces to std::lock_guard<std::mutex>
```

How does the compiler determine the type? That's complicated - read [the docs](#) if interested.

# #25: Fold Expressions

A super handy shorthand for common operators in variadic functions!

Example: Let's make a quick sum function!

```
template<typename ... T>  
auto sum(const T & ... t)  
{  
    // how do we sum this?  
}
```



# #25: Fold Expressions

A super handy shorthand for common operators in variadic functions!

Example: Let's make a quick sum function!

```
template<typename T>
auto sum(const T &t)
{
    return t;
}
```

```
template<typename First, typename Second, typename ... T>
auto sum(const First &first, const Second &second, const T & ... t)
{
    return first + sum(second, t...);
}
```

# #25: Fold Expressions

A super handy shorthand for common operators in variadic functions!

Example: Let's make a quick sum function!

```
template<typename T>
auto sum(const T &t)
{
    return t;
}

template<typename First, typename Second, typename ... T>
auto sum(const First &first, const Second &second, const T & ... t)
{
    return first + sum(second, t...);
}
```

If only there was some way to ask the compiler to sum up the parameter set for us...

# #25: Fold Expressions

A super handy shorthand for common operators in variadic functions!

Example: Let's make a quick sum function!

```
template<typename ... T>  
auto sum(const T & ... t)  
{  
    return (t + ...);  
}
```

# #25: Fold Expressions

A super handy shorthand for common operators in variadic functions!

Example: Let's make a quick sum function!

```
template<typename ... T>
auto sum(const T & ... t)
{
    return (t + ...);
}
```

Works with 32 operators!

*op* - any of the following 32 *binary* operators: `+` `-` `*` `/` `%` `^` `&` `|` `=` `<` `>` `<<` `>>` `+=` `-=` `*=` `/=`  
`%=` `^=` `&=` `|=` `<<=` `>>=` `==` `!=` `<=` `>=` `&&` `||` `,` `.*` `->*`. In a binary fold, both *ops* must be the same.



Where to go from here?

# Where to go?

Use C++!

# Where to go?

Use C++!

Talk to / encourage each other!



# Where to go?

Use C++!

Talk to / encourage each other!

Let us know what kind of awesome projects you make in the future!

# Where to go?

Use C++!

Talk to / encourage each other!

Let us know what kind of awesome projects you make in the future!

Further C++ reading:

**Accelerated C++**

*Andrew Koenig*

**Effective C++**

*Scott Meyers*

**Effective Modern C++**

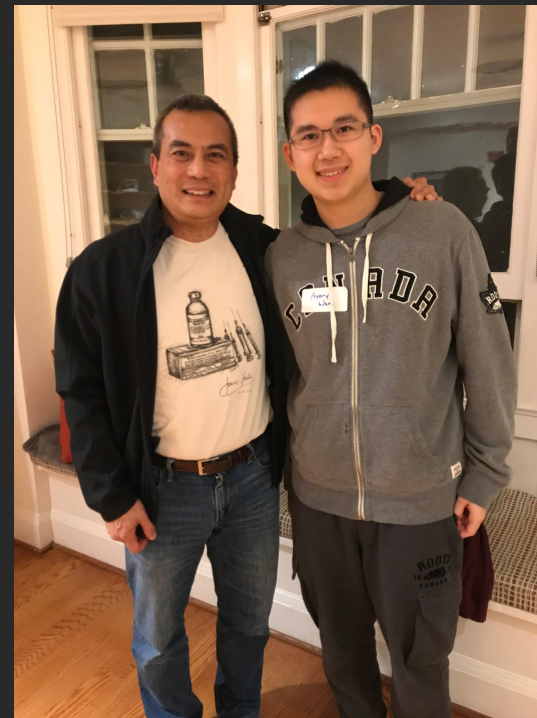
*Scott Meyers*

**Exceptional C++**

*Herb Sutter* + [blog!](#)

**Modern C++ Design**

*Andrei Alexandrescu*



# Thanks Avery!

- Anna

(I cheated because I got to make the final slides ;) )



Thank you!