Lecture 16: Template Metaprogramming

CS 106L, Fall '21

Agenda

- Motivating example
- Computations on Types
- Meta-functions and implementing identity.
- Template deduction and implementing is_same.
- Wrapping everything up with constexpr.

Disclaimer

- Goals for today
  - Introduce more advanced template concepts
  - Demonstrate these techniques can make the type system more flexible

- The code today will feel very "hacky"
  - This part of C++ was unintentional - it was "discovered" by accident
  - But ended up being very useful - you will see it in the STL!

Disclaimer

- Will I ever write TMP code?
  - Maybe. Definitely if you are implementing libraries.
  - But you will probably see them when debugging template error messages!

- Focus on high-level intuition over exact syntactical details!

Motivating Example

Live Code Demo:
distance, take i

Calculate the distance of two iterators of same type.

```cpp
vector<string> names{"Anna", "Ethan", "Nikhil", "Avery"};


auto iter_anna = find(names.begin(), names.end(), "Anna");

auto iter_avery = find(names.begin(), names.end(), "Avery");



cout << distance(iter_anna, iter_avery); // prints 3
```

Calculate the distance of two iterators of same type.

```cpp
deque<string> names{"Anna", "Ethan", "Nikhil", "Avery"};


auto iter_anna = find(names.begin(), names.end(), "Anna");

auto iter_avery = find(names.begin(), names.end(), "Avery");



cout << distance(iter_anna, iter_avery); // prints 3
```

Calculate the distance of two iterators of same type.

```cpp
set<string> names{"Anna", "Ethan", "Nikhil", "Avery"};

// names is sorted, iterator order: "Anna", "Avery", "Ethan", "Nikhil"

auto iter_anna = names.find("Anna");

auto iter_avery = names.find("Avery");



cout << distance(iter_anna, iter_avery); // prints 1
```

Implementation of distance, attempt 1.

```cpp
template <typename It>

size_t distance(It first, It last) {

    return last - first;

}
```

Implementation of distance, attempt 1.

```cpp
template <typename It>

size_t distance(It first, It last) {

    return last - first;

}
```

🚫 **No, this doesn't work!** This assumes **It** is a random access iterator.

Recall: Types of Iterators

- All iterators are **incrementable** (++)
- **Input** iterators can be on the right side of =:

  ```
  auto elem = *it;
  ```
- **Output** iterators can be on the left side of =:

  ```
  *elem = value;
  ```
- **Forward** iterators can be traversed multiple times:

  ```
  iterator a;
  b = a;
  a++; b++;
  assert (*a == *b)        // true
  ```

Recall: Types of Iterators

- **Random access** iterators support indexing by integers!

```
it += 3;                              // move forward 3
it -= 70;                             // move backwards by 70
auto elem = it[5];                    // offset by 5
```

- What iterators do each of the STL containers have?
  - Random access: vector, deque, array*
  - Bidirectional: map, set, list
  - Forward: unordered_map/set, forward_list

*Fun fact: in C++17 now array's have contiguous iterator, which are even stronger than random access iterators.

Implementation of distance, attempt 2.

```cpp
template <typename It>
size_t distance(It first, It last) {
    size_t result = 0;
    while (first != last) {
        ++first; ++result;
    }
    return result;
}
```

Implementation of distance, attempt 2.

```cpp
template <typename It>

size_t distance(It first, It last) {

    size_t result = 0;

    while (first != last) {

        ++first; ++result;

    }

    return result;

}
```

🚫 **This works!** But is not efficient if **It** is a random access iterator.

How does the STL achieve this?

std::**distance**

Defined in header `<iterator>`

```
template< class InputIt >
typename std::iterator_traits<InputIt>::difference_type      (constexpr since C++17)
    distance( InputIt first, InputIt last );
```

Returns the number of hops from `first` to `last`.

The behavior is undefined if `last` is not reachable from `first` by (possibly repeatedly) incrementing `first`.	(until C++11)
If InputIt is not *LegacyRandomAccessIterator*, the behavior is undefined if `last` is not reachable from `first` by (possibly repeatedly) incrementing `first`. If InputIt is *LegacyRandomAccessIterator*, the behavior is undefined if `last` is not reachable from `first` and `first` is not reachable from `last`.	(since C++11)

Parameters

`first` - iterator pointing to the first element

`last` - iterator pointing to the end of the range

Type requirements

- InputIt must meet the requirements of *LegacyInputIterator*. The operation is more efficient if InputIt additionally meets the requirements of *LegacyRandomAccessIterator*

Return value

The number of increments needed to go from `first` to `last`. The value may be negative if random-access iterators are used and `first` is reachable from `last` (since C++11)

Complexity

Linear.

However, if InputIt additionally meets the requirements of *LegacyRandomAccessIterator*, complexity is constant.

Implementation of distance, attempt 3, sort of.

```
template <typename It>

size_t distance(It first, It last) {

    category = what iterator category It is

    if (category is random_access) {

        // fast O(1) code (last - first)

    } else {

        // slow O(N) code (count how many times need to increment first to get to last)

    }

}
```

This is what we will be working towards!

We need to perform some computations on types!

```
template <typename It>

size_t distance(It first, It last) {

    category = what iterator category It is

    if (category is random_access) {

        // fast O(1) code (last - first)

    } else {

        // slow O(N) code (count how many times need to increment first to get to last)

    }

}
```

How do we take a template type, and operate on it?

🤔 **Questions?** 🤔

Computations on Types

Comparison of values vs. type computations

Computation on Values

Computations on Types

```
int s = 3;
```

```
using S = int;
```

We store values with...variables!

We store types using type aliases (C++11).
(previously, you had to use typedefs)

Comparison of values vs. type computations

Computation on Values

```
int s = 3;
int triple = 3 * s;
```

We can create new values with previous values.

Computations on Types

```
using S = int;
using cl_ref = const S&;
```

We can create new types with previous types.

Comparison of values vs. type computations

Computation on Values

```
int s = 3;
int triple = 3 * s;
int result = foo(triple);
```

You can pass variables into functions.

Computations on Types

```
using S = int;
using cl_ref = const S&;
```

???

Comparison of values vs. type computations

Computation on Values

```
int s = 3;
int triple = 3 * s;
int result = foo(triple);
```

You can pass variables into functions.

Computations on Types

```
using S = int;
using cl_ref = const S&;
using result =
    std::remove_reference<cl_ref>::type;
```

You can pass types into **meta**-functions.

Comparison of values vs. type computations

Computation on Values

```cpp
int s = 3;
int triple = 3 * s;
int result = foo(triple);


bool equals = (result == 0);
```

You can evaluate boolean expressions.

Computations on Types

```cpp
using S = int;
using cl_ref = const S&;
using result =
    std::remove_reference<cl_ref>::type;

constexpr bool equals =
    std::is_same<result, const int>::value;
```

You can evaluate boolean expressions - notice that we got a **value** by passing a **type** into a **meta**-function.

Comparison of values vs. type computations

Computation on Values

```cpp
int s = 3;
int triple = 3 * s;
int result = foo(triple);


bool equals = (result == 0);


if (equals)
    exit(1);
```

You can alter control flow using boolean expressions.

Computations on Types

```cpp
using S = int;
using cl_ref = const S&;
using result =
    std::remove_reference<cl_ref>::type;

constexpr bool equals =
    std::is_same<result, const int>::value;

if constexpr (equals)
    exit(1);
```

You can alter **compiler-generated code** using constant boolean expressions!

Write a "function" that determines if two types are equal.

- Old stuff
  - Type aliases
  - Member types
- New stuff
  - Meta-functions
  - constexpr
  - Static member values

```cpp
using S = int;
using cl_ref = const S&;
using result =
    std::remove_reference<cl_ref>::type;

constexpr bool equals =
    std::is_same<result, const int>::value;

if constexpr (equals)
    exit(1);
```

Write a "function" that determines if two types are equal.

- Old stuff
  - Type aliases
  - Member types
- New stuff
  - Meta-functions
  - constexpr
  - Static member values

```cpp
using S = int;
using cl_ref = const S&;
using result =
    std::remove_reference<cl_ref>::type;

constexpr bool equals =
    std::is_same<result, const int>::value;

if constexpr (equals)
    exit(1);
```

Meta-functions

Recap: template types and values

```cpp
// vector.h and array.h
template <typename T>
class vector {
    // stuff
};


template <typename T, size_t N>
class array {
    // more stuff
};
```

```cpp
// main.cpp
int main() {

    vector<int> vec{1, 2, 3};
    array<int, 3> arr{1, 2, 3};

}
```

Recap: template instantiation

```cpp
// vector.h
template <typename T>
class vector {
  void push_back(const T& val);
};


// main.cpp
int main() {
  vector<int> vec1{1, 2, 3};
  vector<double> vec2{2.3};
}
```

```cpp
// compiler generated
class vector_int {
    void push_back(const int& val);
};




class vector_double {
    void push_back(const double& val);
};
```

A meta-function, abstractly.

A meta-function is a "function"

that operates on some **types**/values ("parameters")

and outputs some **types**/values ("return values").

A meta-function, concretely.

A meta-function is a **struct**

that has **public member types/fields** which depend on

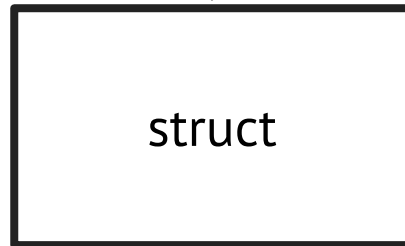what the **template types/values** are instantiated with.

Regular vs. meta-functions

parameters values

⬇

```
function
```

⬇
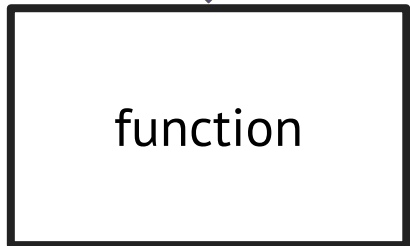
return value

template types or values

⬇

```
struct
```

↙ ↘

types: a member type called "type"

values: a static member called "value"

We'll write an identity function that "outputs" its "input".
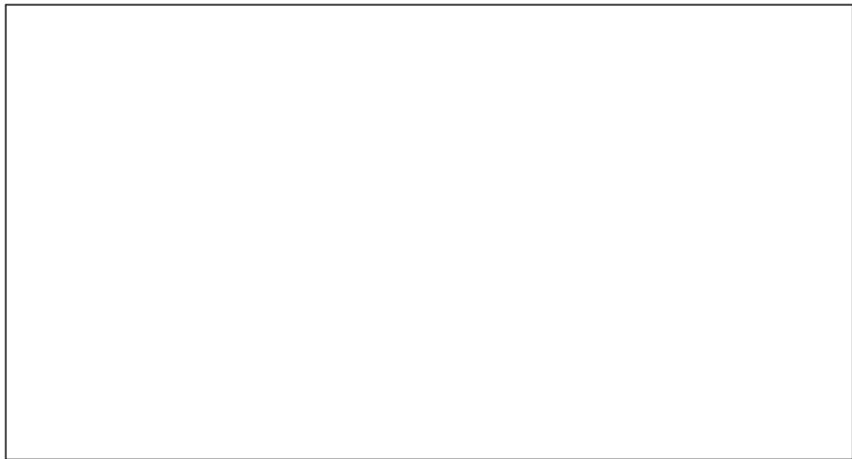
Input type, output type
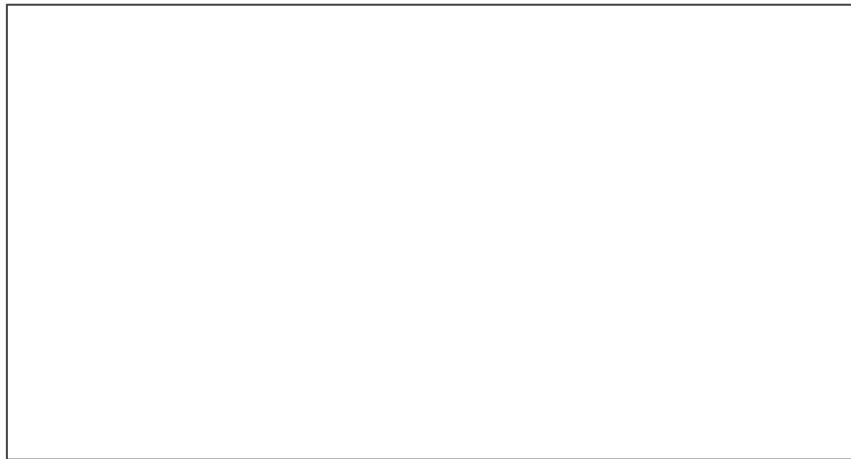
Input value, output value

Recall: the syntax to access a member type.

Input type, output type

Input value, output value

typename

```
using K = identity<int>::type;
```

```
int val = identity<3>::value;
```

The input is the template type or value.

Input type, output type

```
template <typename T>

struct identity {



};
```

Input value, output value

```
template <int V>

struct identity {



};
```

```
using K = identity<int>::type;
```

```
int val = identity<3>::value;
```

The output is a public member that depends on input.

Input type, output type

```cpp
template <typename T>

struct identity {

    using type = T;

};
```

Input value, output value

```cpp
template <int V>

struct identity {

    static const int value = V;

};
```

```cpp
using K = identity<int>::type;
```

```cpp
int val = identity<3>::value;
```

The output is a public member that depends on input.

Input type, output type

```
template <typename T>

struct identity {

    using type = T;

};
```

Input value, output value

```
template <int V>

struct identity {

    static const int value = V;

};
```

We don't need to actually instantiate a struct to use this meta-function.

```
using K = identity<int>::type;
```
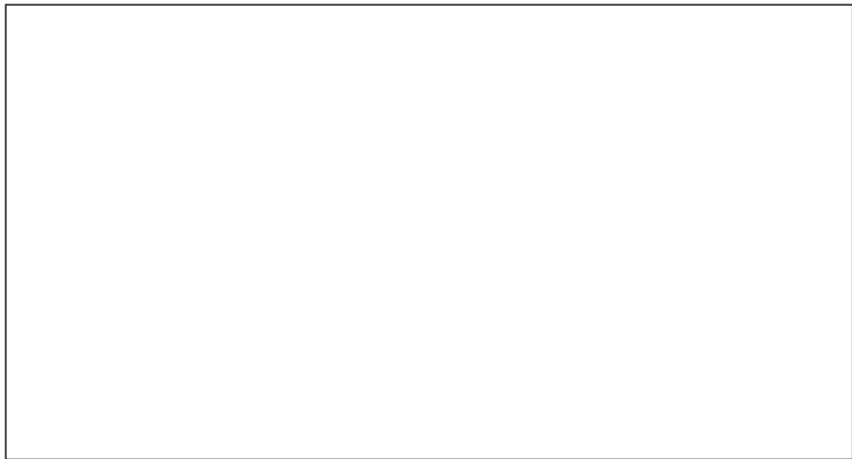
```
int val = identity<3>::value;
```

Meta-functions are structs that treats its template types/values as the parameters, and places the return values as public members.

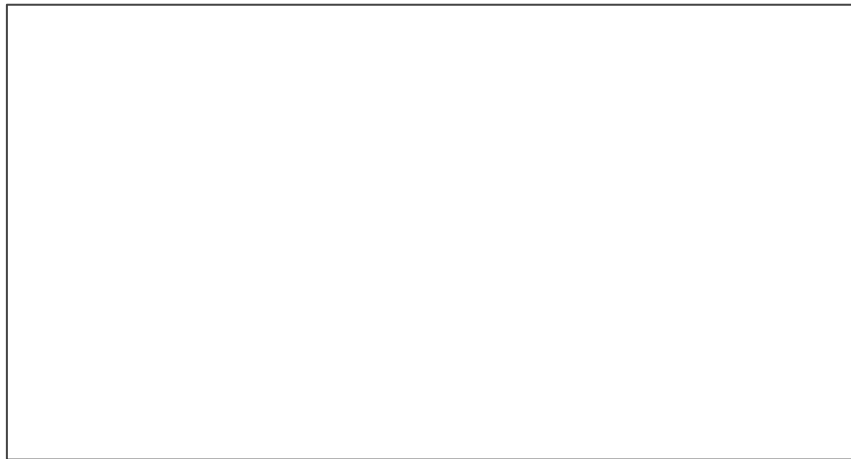We **never** need to create an instance of the struct.

🤔 **Questions?** 🤔

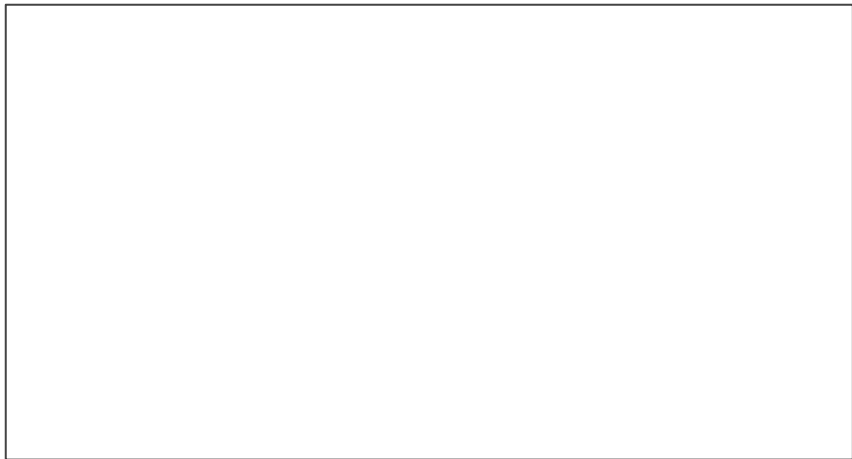# We'll write the is_same meta-function!

Input type, output value

Input value, output value

Recall: the syntax to access a member type.

Input type, output value

Input value, output value

```
bool diff = is_same<int, double>::value;
bool same = is_same<int, int>::value;
```
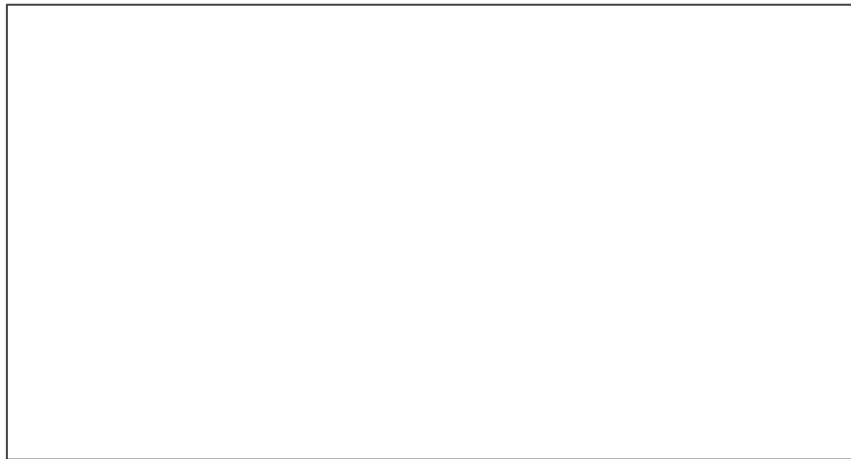
```
bool diff = is_same<3, 4>::value;
bool same = is_same<3, 3>::value;
```

Now the input is two template types or values.

Input type, output type

```cpp
template <typename T, typename U>

struct is_same {




};
```

Input value, output value

```cpp
template <int V, int W>

struct is_same {




};
```

```cpp
bool diff = is_same<int, double>::value;
bool same = is_same<int, int>::value;
```

```cpp
bool diff = is_same<3, 4>::value;
bool same = is_same<3, 3>::value;
```

How do we check if two types are the same?

Input type, output type

```cpp
template <typename T, typename U>

struct is_same {

    static const bool value =
            ????;

};
```

Input value, output value

```cpp
template <int V, int W>

struct is_same {

    static const bool value =
            (V == W);

};
```

```cpp
bool diff = is_same<int, double>::value;
bool same = is_same<int, int>::value;
```

```cpp
bool diff = is_same<3, 4>::value;
bool same = is_same<3, 3>::value;
```

Template Deduction

Template Specialization

You can have a "generic" template, as well as "specialized" templates for particular types.

Templates can be fully specialized.

```
template <typename T>              // generic
class vector {
    // implementation using an array
    // that is resized as necessary
};


template <>                        // specialized
class vector<bool> {
    // implementation using a bit array
    // so each element takes one bit
};
```

Templates can be partially specialized.

```
template <typename K, typename V>              // generic
struct HashMap { /* stuff */  };


template <typename K, typename V>              // partially specialized
struct HashMap<K*, V> { /* stuff */  };        // first type is a pointer


template <typename V>                          // partially specialized
struct HashMap<int, V> { /* stuff */  };       // first type is int


template <>                                    // fully specialized
struct HashMap<int, int> { /* stuff */  };     // both types are int
```

CTAD Rules ("Class Template Argument Deduction")

If there are two template classes declared, there is a ranking system to determine which one to use.

More specialized templates have higher priority.

SFINAE ("Substitution Failure is Not An Error")

If the compiler finds that something went wrong when substituting template parameters, it will simply ignore that template. No error will be issued.

CTAD + SFINAE

1. Find all matching templates.
2. Rank the templates. ??? 依靠
3. Try to substitute the first. If success, done!
   If fail, don't panic (SFINAE), try the next one.
4. Error only if all fail, or if there is a tie.

Template Deduction Rules

All the template rules, summarized.

Compiler will rank all the templates,
try each until it finds one that works.

Error only if none work or if there is a tie.

What happens in this code snippet?

```cpp
template <typename T, typename U>

struct is_same {

  static const bool value = false;

};



template <typename T>

struct is_same<T, T> {

  static const bool value = true;

};
```

```cpp
bool diff = is_same<int, double>::value;
bool same = is_same<int, int>::value;
```

What happens in this code snippet? (for reference)

```cpp
template <typename T, typename U>
struct is_same {    // generic
  static const bool value = false;
};


template <typename T>
struct is_same<T, T> { // specialized
  static const bool value = true;
};
```

- Compiler will try the specialized template before the generic template.
- If two template types are the same, substitution succeeds and value is true.
- Otherwise, compiler tries the second one which will succeed, and value is false.

Same technique to check other qualities of a type.

```cpp
template <typename T>

struct is_pointer {

  static const bool value = false;

};


template <typename T>

struct is_pointer<T*> {

  static const bool value = true;

};
```

```cpp
bool no = is_pointer<int>::value;
bool yes = is_pointer<int*>::value;
```

Same technique to change and return a type.

```cpp
template <typename T>

struct remove_const {

  using type = T;

};



template <typename T>

struct remove_const <const T> {

  using type = T;

};
```

```cpp
using K = remove_const<int>::type;
using M = remove_const<const int>::type;
```

Template Deduction Summarized

This is a "hack".

We are exploiting the compiler's template matching rules to implement an if/else statement for types.

🤔 **Questions?** 🤔

We've built up a collection of predicate meta-functions (they take in a type and return a bool).

What can we do with them?

Wrapping it up with constexpr

Live Code Demo:
distance, take ii

A little background about iterators.

Each iterator has a member type that represents what iterator category it is in.

Every iterator category (e.g. random access) has a dummy "type" object associated with it.

Implementation of distance, attempt 3, fully complete.

```cpp
template <typename It>

size_t distance(It first, It last) {

    using category = typename std::iterator_traits<It>::iterator_category;

    if (std::is_same<std::random_access_iterator_tag, category>::value) {

        return last - first;

    } else {

        // slow O(N) code (count how many times need to increment first to get to last)

    }

}
```

🚫 **No, this doesn't work!** last - first doesn't compile if **It** is not random access.

The problem right now.

The other branch doesn't compile, even though we know that branch won't ever be run.

Need a way to remove the offending code when the if statement knows that part won't be run.

Pre-C++17: std::enable_if

Create a meta-function which purposefully will generate a substitution failure.

Use it to "turn on and off" different functions.

If curious, ask me after class. This is like the weirdest, most "hacky" thing in C++.

What happens in this code snippet? (skipped)

```
template <typename B, typename T = void>

struct enable_if {    };


template <typename T>

struct enable_if<true, T> {

  using type = T;

};
```

(handwritten annotation: "bool" written above `typename` which is struck out)

```
using K = enable_if<B, int>::type;
```

- If B is true, then K would be type int.
- If B is false, this won't compile!

(handwritten annotation: no "type")

A super cool keyword: constexpr.

constexpr (C++11): you can calculate this expression at compile-time. Stronger form of const.

if constexpr (C++17): you can calculate the boolean at compile-time. Replace the entire if/else with the code that will actually be run.

Implementation of distance, attempt 3, fully complete.

```cpp
template <typename It>

size_t distance(It first, It last) {

    using category = typename std::iterator_traits<It>::iterator_category;

    if constexpr (std::is_same<std::random_access_iterator_tag, category>::value) {

        return last - first;

    } else {

        // slow O(N) code (count how many times need to increment first to get to last)

    }

}
```

✅ **This is correct!** Offending code is removed if iterator not random access.

Live Code Demo:
exploring the assembly code!

What is the template code generated?

```
vector<string> names{"Anna", "Ethan", "Nikhil", "Avery"};


auto iter_anna = find(names.begin(), names.end(), "Anna");

auto iter_avery = find(names.begin(), names.end(), "Avery");



cout << distance(iter_anna, iter_avery) << endl;

// prints 3
```

What is the template code generated?

```
set<string> names{"Anna", "Ethan", "Nikhil", "Avery"};


auto iter_anna = names.find("Anna");

auto iter_avery = names.find("Avery");



cout << distance(iter_anna, iter_avery) << endl;

// prints 1
```

Template code generated for previous two slides.

```cpp
size_t distance_vector(vector<int>::iterator first, vector<int>::iterator last) {

    return last - first;

}


size_t distance_set(set<int>::iterator first, set<int>::iterator last) {

    size_t result = 0;

    while (first != last) { ++first; ++result; }

    return result;

}
```

✅ The compiler determines the correct code to generate for each type.

Template Metaprogramming Summarized

Use template techniques to modify the actual source code of the program.

constexpr if can be used to turn off pieces of the code which wouldn't compile for certain types.

Key Takeaways

- Templates allow you to treat types as "variables".
- Meta-functions lets you modify these types, or to query information about these template types.
- constexpr (if) gives you the flexibility to modify the source code based on the template type.
- This allows you to optimize code based on type!

Where does TMP pop up in the real world?

std::move

Defined in header `<utility>`

```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t ) noexcept;
```
(since C++11)
(until C++14)

```
template< class T >
constexpr std::remove_reference_t<T>&& move( T&& t ) noexcept;
```
(since C++14)

```
82   template<typename data_t, typename index_t>
83   typename std::enable_if<std::is_same<data_t, float>::value, void>::type
84   index_select_add(const Tensor &select_indices,
85                            const Tensor &add_indices,
86                            const Tensor &src,
87                            Tensor &output,
88                            const Tensor& offsets,
89                            bool include_last_offset) {
90     int64_t ddim = src.size(1);
91     auto* select_indices_data = select_indices.data_ptr<index_t>();
92     auto* output_data = output.data_ptr<float>();
```

Pytorch C++ implementation of index_select.

🤔 **Questions?** 🤔

Where to Go Next

- This talk was heavily inspired Jody Hagins' talk at cppcon, 2020. It's available online!
- The C++20 hype is the introduction of **concepts**, developed to make some of the stuff today less complicated and more natural!

Next time

Final lecture