

# Writing a `min` function

```
// Returns the smaller of a and b  
int min(int a, int b) {  
    return a < b ? a : b;  
}
```

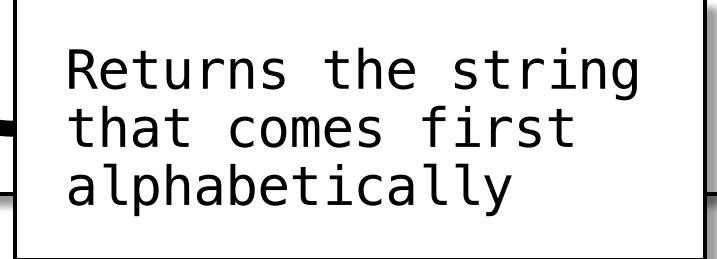
## Ternary Operator

Return `a` if `a < b` otherwise return `b`

# Writing a `min` function

`min` makes sense for more than just integers. How can we do this?

```
min(106, 107);          // int, returns 106  
min(1.2, 3.4);          // double, returns 1.2  
min("Jacob", "Fabio");  // string, returns "Fabio"
```



Returns the string  
that comes first  
alphabetically

# One solution: function overloading

```
int min(int a, int b) {  
    return a < b ? a : b;  
}  
  
double min(double a, double b) {  
    return a < b ? a : b;  
}  
  
std::string min(std::string a, std::string b) {  
    return a < b ? a : b;  
}
```





# Hmm... this looks familiar!

```
class IntVector {  
    class DoubleVector {  
        class StringVector {  
            } // Code to store  
            // a list of  
            // strings...  
        };
```

# Let's take this...

```
int min(int a, int b) {  
    return a < b ? a : b;  
}
```

```
double min(double a, double b) {  
    return a < b ? a : b;  
}
```

```
std::string min(std::string a, std::string b) {  
    return a < b ? a : b;  
}
```

This works, but it's missing the bigger idea!

# ...and turn it into this!

This is a **template**

T gets replaced with a specific type

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

# Remember: templates vs. functions

```
template <typename T>  
T min(T a, T b)
```

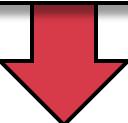
```
min<std::string>
```

This is a template.  
It's **not** a function

This is a function.  
A.K.A a template  
instantiation

# Template functions

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



We can also use  
references to avoid  
making a copy!

```
template <typename T>
T min(const T& a, const T& b) {
    return a < b ? a : b;
}
```

# Option A: explicit instantiation

Explicit instantiation passes the types directly, just like template classes

```
min<int>(106, 107);      // Returns 106  
min<double>(1.2, 3.4); // Returns 1.2
```

# Option A: explicit instantiation

Template functions cause the compiler to **generate code** for us

```
int min(int a, int b) {          // Compiler generated
    return a < b ? a : b;        // Compiler generated
}

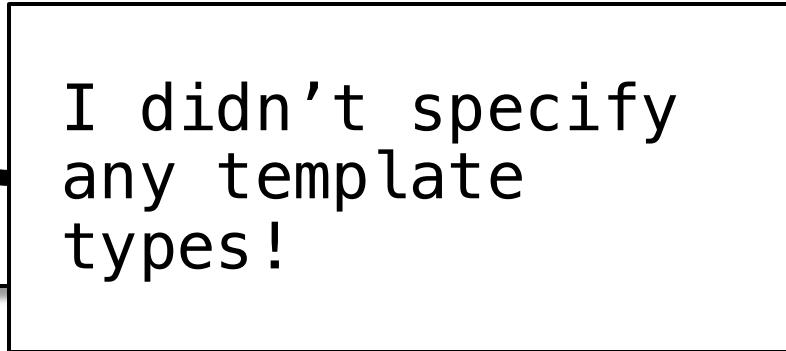
double min(double a, double b) {   // Compiler generated
    return a < b ? a : b;        // Compiler generated
}

min<int>(106, 107);           // Returns 106
min<double>(1.2, 3.4);        // Returns 1.2
```

# Option B: implicit instantiation

Implicit instantiation lets the compiler **infer** the types for us

```
min(106, 107);      // int, returns 106  
min(1.2, 3.4);     // double, returns 1.2
```



I didn't specify  
any template  
types!

# Implicit instantiation is kind of like **auto**

```
auto number = 106;
```

This still is an **int**,  
we just let the  
compiler figure it  
out

# Implicit instantiation is kind of like **auto**

```
int m = min(106, 107);
```

It's exactly as if we wrote

**min<int>(106, 107)**

# Implicit instantiation can be finicky

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

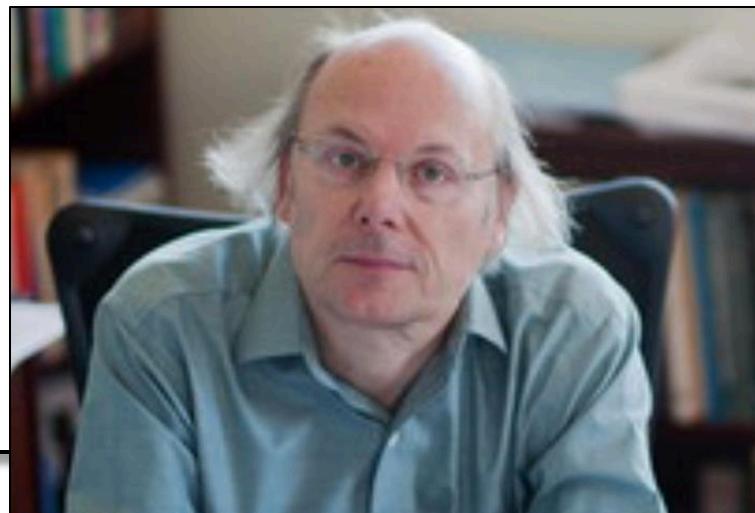
```
min("Jacob", "Fabio");
```

const char\*

What type is T? What are the types of the arguments?  
Hint: you might know this if you've taken CS107!

# Implicit instantiation can be finicky

```
const char* min(const char* a, const char* b) {  
    return a < b ? a : b;  
}  
  
min<const char*>("Jacob", "Fabio");
```



Pointer comparison  
AHHHH!!! This is not  
what we wanted

← This is Bjarne  
judging you for using  
pointer comparison

# Implicit instantiation can be finicky

We can always use explicit instantiation in ambiguous cases like this

```
template <typename T>
T min(const T& a, const T& b) {
    return a < b ? a : b;
}

min<std::string>("Jacob", "Fabio");
```

const **char\*** gets  
converted to  
**std::string** here

↓ Here is Bjarne  
pleased with you for  
getting the compiler  
to understand you!



# Implicit instantiation can be finicky

Another example: the types of the parameters don't strictly match

```
template <typename T>
T min(const T& a, const T& b) {
    return a < b ? a : b;
}
```

```
min(106, 3.14); // ✗ Doesn't compile
```

int      double

**Explicit instantiation!**  
min<double>(106, 3.14)



# Implicit instantiation can be finicky

Another solution: make our template a little bit more flexible.

```
template <typename T, typename U>
????? min(const T& a, const U& b) {
    return a < b ? a : b;
}
```

```
min(106, 3.14);
```

T = int

U = double

What should the return type of this function be?

# Implicit instantiation can be finicky

Another solution: make our template a little bit more flexible.

```
template <typename T, typename U>
auto min(const T& a, const U& b) {
    return a < b ? a : b;
}
min(106, 3.14);
```

What should the return type of this function be?

It's complicated, let the compiler figure it out with **auto**

# Pro tip: Use IDE to see instantiation types

IDEs (e.g. VSCode, QtCreator) can show what types were actually used

The screenshot shows a code editor window for a file named "main.cpp". The code defines a template function "min<T>(T, T)" and uses it in the "main()" function. A tooltip or callout box highlights the instantiation of the template at line 9, showing that the template was instantiated with the type "const char \*".

```
main.cpp  ×

8-template-classes-and-cc > main.cpp > min<T>(T, T)

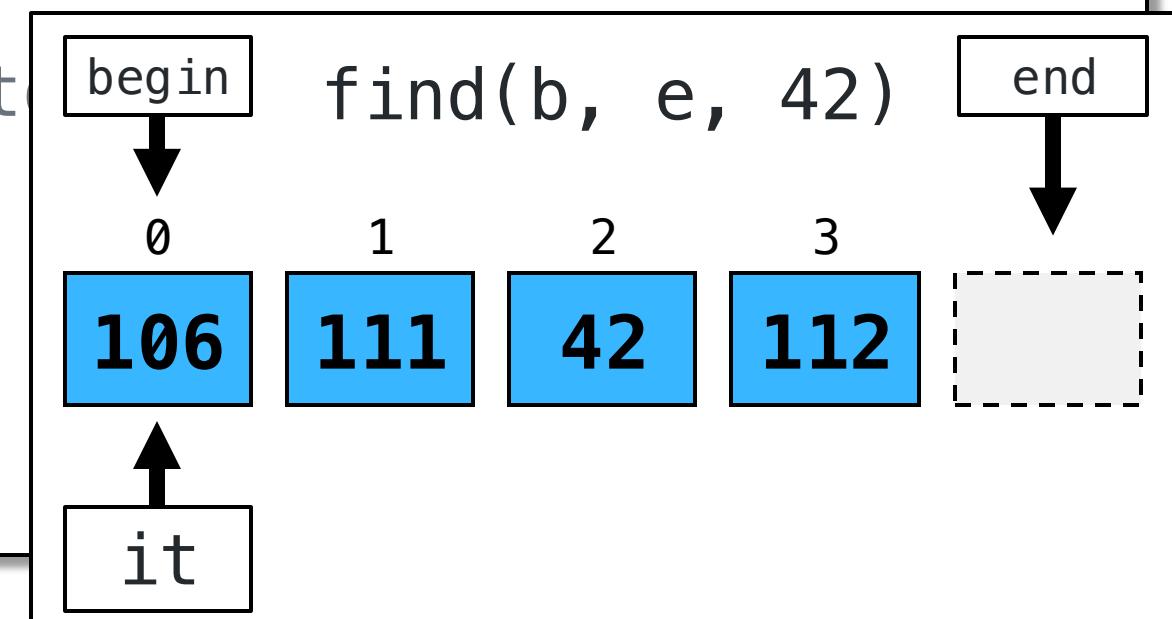
1 template <typename T>
2 T min(T a, T b)
3 {
4     return a < b ? a : b;
5 }
6
7 int main()
8 {
9     const char *min<const char *>(const char *a, const char *b)
10    auto m = min("Jacob", "Fabio");
11 }
```

# Writing a **find** function

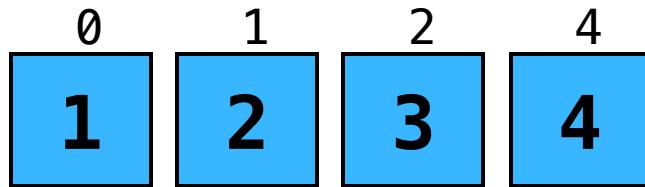
```
std::vector<int> v { 106, 111, 42, 112 };
auto it = find(v.begin(), v.end(), 42);
*it = 107;
// v = { 106, 111, 107, 112 }
```

# Writing a **find** function

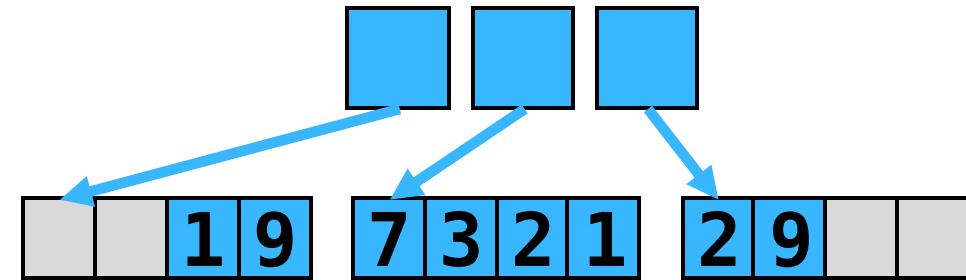
```
std::vector<int>::iterator find(  
    std::vector<int>::iterator begin,  
    std::vector<int>::iterator end,  
    int value  
) {  
    // Logic to find the iterator  
    // Should return end if no  
}
```



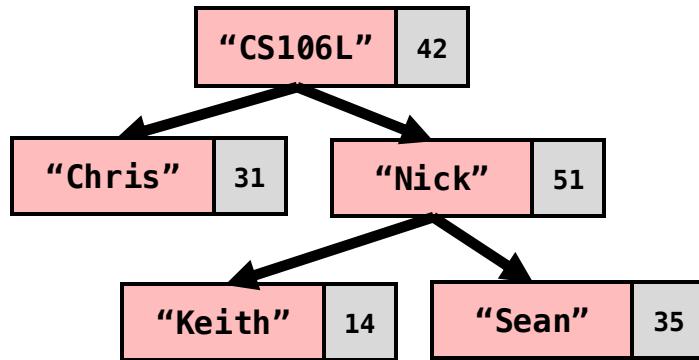
# Recall: we have many iterator types!



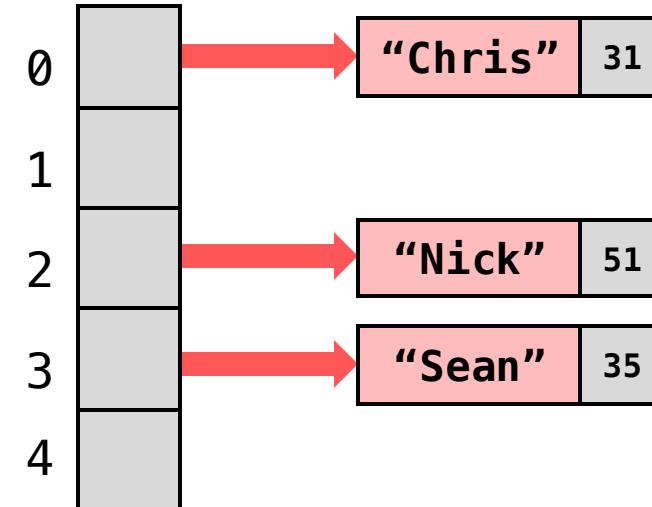
`vector<T>::iterator`



`deque<T>::iterator`



`map<K, V>::iterator`



`unordered_map<K, V>::iterator`

# But they all share the same interface!

```
// Copy construction  
auto it = c.begin();
```

```
// Increment iterator forward  
++it;
```

```
// Dereference iterator -- undefined if it == end()  
auto elem = *it;
```

```
// Equality: are we in the same spot?  
if (it == c.end()) ...
```

# This definition is too specific!

```
std::vector<int>::iterator find(  
    std::vector<int>::iterator begin,  
    std::vector<int>::iterator end,  
    int value  
) {  
    // Logic to find the iterator in this container  
    // Should return end if no such element is found  
}
```

# Writing a **find** function

Our **find** function won't work for other vectors, or other containers

```
std::vector<std::string> v { "seven", "kingdoms" };
auto it = find(v.begin(), v.end(), "kingdoms");
// Won't compile
```

```
std::set<std::string> s { "house", "targaryen" };
auto it = find(s.begin(), s.end(), "targaryen");
// Gods help us
```

# Writing a **find** function... but templated

Form a small group and let's implement this function! QR code on next slide.

```
template <????, ???>
???? find(???? begin, ??? end, ??? value) {
    // Logic to find and return the iterator
    // in this container whose element is value
    // Should return end if no such element is found
}
```

# find function in the STL

- Part of `<algorithm>` header (we'll talk more about this on Thursday)!
- You guys have all the tools now to read the C++ standard!

**std::find, std::find\_if, std::find\_if\_not**

Defined in header `<algorithm>`

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

# **Concepts**

# Back to our `min` function

```
template <typename T>
T min(const T& a, const T& b) {
    return a < b ? a : b;
}
```

What **must be true** of a type `T` for us to be able to use `min`?

```
// For which T will the following compile successfully?
T a = /* an instance of T */;
T b = /* an instance of T */;
min<T>(a, b);
```

# Back to our `min` function

`T` must have an `operator<` to make sense in this context

```
struct StanfordID; // How do we compare two IDs?  
  
StanfordID jacob { "Jacob", "jtrb" };  
StanfordID fabio { "Fabio", "fabioi" };  
min<StanfordID>(jacob, fabio); // ✗ Compiler error
```

# What happened?

```
$ g++ main.cpp --std=c++20
main.cpp:9:12: error: invalid operands to binary expression
('const StanfordID' and 'const StanfordID')
return a < b ? a : b;
          ~ ^ ~
main.cpp:20:3: note: in instantiation of function template
specialization 'min<StanfordID>' requested here
min<StanfordID>(jacob, fabio);
^
1 error generated.
```

**Compiler only finds the error *after* instantiation**

# Recall: `std::set` also requires an operator<

Bad templates can produce really confusing compiler errors...

```
std::set<StanfordID> s { jacob, fabio };
```

```
jacobrobertsbaca@Jacobs-MacBook-Pro-3 8-template-classes-and-cc % clang++ main.cpp --std=c++20
In file included from main.cpp:1:
In file included from /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/string:520:
In file included from /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/_functional_base:16:
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/_functional/operations.h:487:21: error: invalid operands to binary expression ('const StanfordID' and 'const StanfordID')
    {return __x < __y;}
        ^ ~~~
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/_tree:2023:28: note: in instantiation of member function 'std::less<StanfordID>::operator<' requested here
    if (__hint == end() || value_comp()(&__v, *__hint)) // check before
        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/_tree:2114:36: note: in instantiation of function template specialization 'std::__tree<StanfordID, std::less<StanfordID>, std::allocator<StanfordID>>::__find_equal<StanfordID>' requested here
    __node_base_pointer& __child = __find_equal(__p, __parent, __dummy, __k);
        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/_tree:1257:16: note: in instantiation of function template specialization 'std::__tree<StanfordID, std::less<StanfordID>, std::allocator<StanfordID>>::__emplace_hint_unique_key_args<StanfordID, const StanfordID &>' requested here
    return __emplace_hint_unique_key_args(__p, _NodeTypes::__get_key(__v), __v).first;
        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/_set:697:25: note: in instantiation of member function 'std::__tree<StanfordID, std::less<StanfordID>, std::allocator<StanfordID>>::__insert_unique<StanfordID>' requested here
    __node_base_pointer = __insert_unique(__p, __key, __v);
```

Recall: `std::set` requires operator<

The error message continues to go on



# Also a problem for our **find** function

```
int main() {  
    int idx = find(1, 5, 3); // eh... 3 I guess? haelp  
}
```



```
main.cpp:16:9: error: indirection requires pointer operand ('int' invalid)  
    if (*it == value) {  
        ^~~  
main.cpp:29:3: note: in instantiation of function template specialization 'find<int, int>'  
    find(1, 5, 3); // eh... 3 I guess? haelp <U+1F972>  
    ^  
1 error generated.
```

**C++ beginner:** “Uh..  
Compiler, what the @!#\*  
do you mean?”

# Idea: How do we put **constraints** on templates?

- Templates are great, but the errors they produce when used incorrectly are unintuitive
- How can we be up-front about what we require of a template type?

C#

```
class EmployeeList<T>
where T : notnull,
Employee,
IComparable<T>, new()
```

Java

```
class ListObject<T
extends Comparable<T>>
```

# Idea: How do we put constraints on templates?

Compiler shouldn't instantiate a template unless all constraints are met

```
template <typename T>  
T min(const T& a, const T& b)
```

T must have operator<

```
template <typename T>  
struct set;
```

It must be an  
iterator type

```
template <typename It, typename T>  
It find(It begin, It end, const T& value)
```

# Creating a Comparable concept

```
template <typename T>
concept Comparable = requires(const T a, const T b) {
    { a < b } -> std::convertible_to<bool>;
};
```

# Creating a Comparable concept

**concept**: a named set of  
**constraints**

**requires**:

Given two **T**'s, I expect  
the following to hold

```
template <typename T>
concept Comparable = requires(const T a, const T b) {
    { a < b } -> std::convertible_to<bool>;
};
```

**constraint**: Anything  
inside the {} must  
compile without error

**constraint**: ...and the  
result must be bool-like  
*convertible\_to* is also a  
concept!

# Using our Comparable concept

```
template <typename T> requires Comparable<T>
T min(const T& a, const T& b);
```



```
// Super slick shorthand for the above
template <Comparable T>
T min(const T& a, const T& b);
```

# Concepts greatly improve compiler errors

Here's the error from before when instantiating a set **without** a concept

```
jacobrobertsbaca@Jacobs-MacBook-Pro-3 8-template-classes-and-cc % clang++ main.cpp --std=c++20
In file included from main.cpp:1:
In file included from /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Dev
OSX.sdk/usr/include/c++/v1/string:520:
In file included from /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Dev
OSX.sdk/usr/include/c++/v1/_functional_base:16:
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk
+v1/_functional/operations.h:487:21: error: invalid operands to binary expression ('const St
const StanfordID')
    {return __x < __y;}
        ~^ ~~~
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c+
+v1/_tree:2023:28: note: in instantiation of member function 'std::less<StanfordID>::operator<
here
    if (__hint == end() || value_comp()(&__v, *__hint)) // check before
        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk
+v1/_tree:2114:36: note: in instantiation of function template specialization 'std::__tree<S
:less<StanfordID>, std::allocator<StanfordID>>::__find_equal<StanfordID>' requested here
    __node_base_pointer& __child = __find_equal(__p, __parent, __dummy, __k);
        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk
+v1/_tree:1257:16: note: in instantiation of function template specialization 'std::__tree<S
:less<StanfordID>, std::allocator<StanfordID>>::__emplace_hint_unique_key_args<StanfordID, co
>' requested here
    return __emplace_hint_unique_key_args(__p, _NodeTypes::__get_key(__v), __v).first;
        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk
+v1/set:682:25: note: in instantiation of member function 'std::__tree<StanfordID, std::less<

```

template <typename T>  
struct std::set;



# Concepts greatly improve compiler errors

Here's the error when instantiating a set **with** a concept

```
main.cpp:32:3: error: constraints not satisfied for class template 'set' [with T = StanfordID]
  set<StanfordID> ids { jacob, fabio };
  ^~~~~~
main.cpp:13:11: note: because 'StanfordID' does not satisfy 'Comparable'
template <Comparable T>
  ^
main.cpp:10:7: note: because 'a < b' would be invalid: invalid operands to binary expression ('const StanfordID' and 'const StanfordID')
  { a < b } -> std::convertible_to<bool>;
  ^
4 errors generated.
```



```
template <Comparable T>
struct std::set;
```

# C++ comes with many built-in concepts

## Core language concepts

Defined in header `<concepts>`

`same_as` (C++20)

specifies that a type is the same as another type  
(concept)

`derived_from` (C++20)

specifies that a type is derived from another type  
(concept)

`convertible_to` (C++20)

specifies that a type is implicitly convertible to another type  
(concept)

`common_reference_with` (C++20)

specifies that two types share a common reference type  
(concept)

`common_with` (C++20)

specifies that two types share a common type  
(concept)

`integral` (C++20)

specifies that a type is an integral type  
(concept)

`signed_integral` (C++20)

specifies that a type is an integral type that is signed  
(concept)

`unsigned_integral` (C++20)

specifies that a type is an integral type that is unsigned  
(concept)

`floating_point` (C++20)

specifies that a type is a floating-point type  
(concept)

`assignable_from` (C++20)

specifies that a type is assignable from another type  
(concept)

`swappable`  
`swappable_with` (C++20)

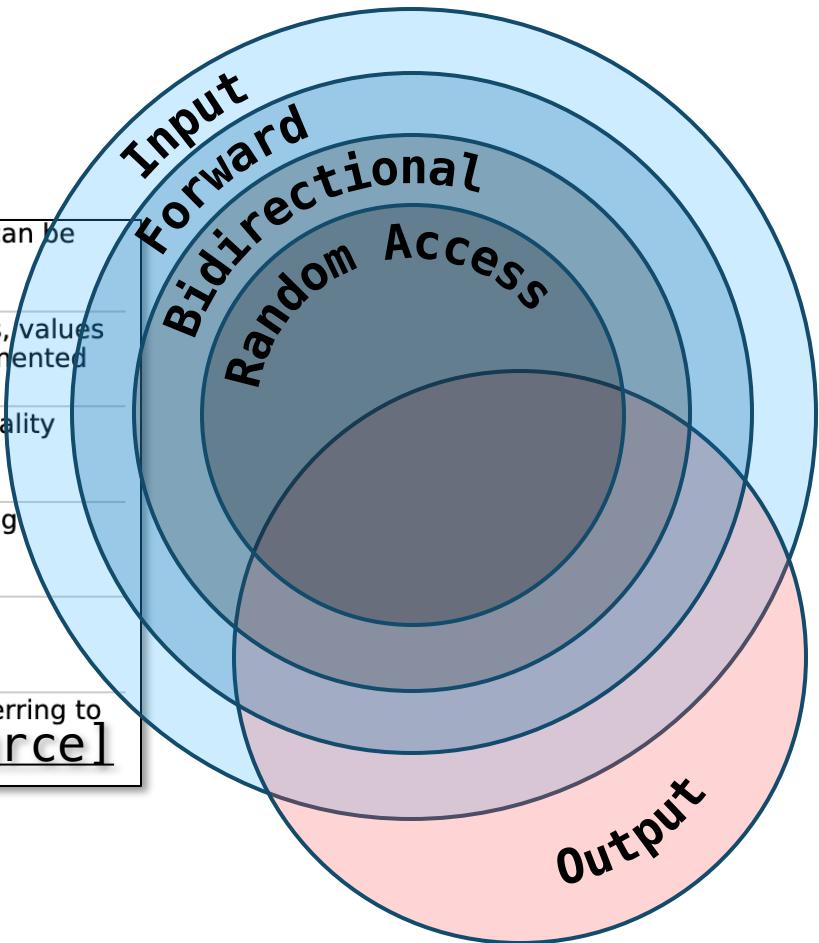
specifies that a type can be swapped or that two types can be swapped with  
each other  
(concept)

[\[source\]](#)

# ...including iterator concepts!

<b>input_iterator</b> (C++20)	specifies that a type is an input iterator, that is, its referenced values can be read and it can be both pre- and post-incremented (concept)
<b>output_iterator</b> (C++20)	specifies that a type is an output iterator for a given value type, that is, values of that type can be written to it and it can be both pre- and post-incremented (concept)
<b>forward_iterator</b> (C++20)	specifies that an <b>input_iterator</b> is a forward iterator, supporting equality comparison and multi-pass (concept)
<b>bidirectional_iterator</b> (C++20)	specifies that a <b>forward_iterator</b> is a bidirectional iterator, supporting movement backwards (concept)
<b>random_access_iterator</b> (C++20)	specifies that a <b>bidirectional_iterator</b> is a random-access iterator, supporting advancement in constant time and subscripting (concept)
<b>contiguous_iterator</b> (C++20)	specifies that a <b>random_access_iterator</b> is a contiguous iterator, referring to elements that are contiguous in memory (concept)

[source]



Remember our iterator types?

# Fixing up our **find** function

```
template <std::input_iterator It, typename T>
It find(It begin, It end, const T& value);
```

```
int idx = find(1, 5, 3); // WHY DOES THIS NOT WORK?
```

```
main.cpp:10:11: note: because 'int' does not satisfy 'input_iterator'  
template <std::input_iterator It, typename T>  
      ^
```



AWWW THANKS

**How do we create a function that accepts a  
variable number of parameters?**

# Back to our `min` function

```
template <Comparable T>
T min(const T& a, const T& b) {
    return a < b ? a : b;
}

min(2.4, 7.5);           // This works
min(2.4, 7.5, 5.3);     // What about this?
min(2.4, 7.5, 5.3, 1.2); // or this?
```

# One solution: function overloading

```
template <Comparable T>
T min(const T& a, const T& b) { return a < b ? a : b; }
```

```
template <Comparable T>
T min(const T& a, const T& b, const T& c) {
    auto m = min(b, c);
    return a < m ? a : m;
}
```

```
template <Comparable T>
T min(const T& a, const T& b, const T& c, const T& d) {
    auto m = min(b, c, d);
    return a < m ? a : m;
}
```

3 element overload  
calls 2 element

4 element overload  
calls 3 element

Seems almost recursive!

# One solution: function overloading

```
min(2.4, 7.5);           // This works  
min(2.4, 7.5, 5.3);     // This works now  
min(2.4, 7.5, 5.3, 1.2); // and this works too!
```

```
min(2.4, 7.5, 5.3, 1.2, 3.4, 6.7, 8.9, 9.1); 
```

// Time to write 7 overloads I guess...

**Yes! Templates + recursion** 🎉 🎊

# But first... a (slightly) different solution

Can't we solve this recursively using `std::vector`!?

```
template <Comparable T>
T min(const std::vector<T>& values);

// Passing a vector<double> here!
// Note the { } braces (list initialized vector)
min({ 2.4, 7.5 });
min({ 2.4, 7.5, 5.3 });
min({ 2.4, 7.5, 5.3, 1.2 });
```

# But first... a (slightly) different solution

Can't we solve this recursively using `std::vector`!?

```
template <Comparable T>
T min(const std::vector<T>& values) {
    if (values.size() == 1) return values[0];
    const auto& first = values[0];
    std::vector<T> rest(++values.begin(), values.end());
    auto m = min(rest);
    return first < m ? first : m;
}
```

Talk to a partner for 60s. What should the last two lines be?

# But first... a (slightly) different solution

Can't we solve this recursively using `std::vector`!?

```
template <Comparable T>
T min(const std::vector<T>& values) {
    if (values.size() == 1) return values[0];
    const auto& first = values[0];
    std::vector<T> rest(++values.begin(), values.end());
    auto m = min(rest);
    return first < m ? first : m;
}
```

**Base Case:** if we only have one element, return that element!

# But first... a (slightly) different solution

Can't we solve this recursively using `std::vector`!?

```
template <Comparable T>
T min(const std::vector<T>& values) {
    if (values.size() == 1) return values[0];
    const auto& first = values[0];
    std::vector<T> rest(++values.begin(), values.end());
    auto m = min(rest);
    return first < m ? first : m;
}
```

**Recursive Case:** compare first element to min of remaining elements!

# But first... a (slightly) different solution

Can't we solve this recursively using `std::vector`!?

```
template <Comparable T>
T min(const std::vector<T>& values) {
    if (values.size() == 1) return values[0];
    const auto& first = values[0];
    std::vector<T> rest(++values.begin(), values.end());
    auto m = min(rest);
    return first < m ? first : m;
}
```

This solution is correct. But does anyone see any **inefficiencies**?

# Some problems with this approach...

- It recursively copies the vector (can avoid with wrapper function!)
- Must allocate a vector for every call (unavoidable overhead)

```
template <Comparable T>
T min(const std::vector<T>& values);

// Passing a vector<double> here!
// Note the { } braces (list initialized vector)
min({ 2.4, 7.5 });
min({ 2.4, 7.5, 5.3 });
min({ 2.4, 7.5, 5.3, 1.2 });
```

# Recall: function overloading

```
template <Comparable T>
T min(const T& a, const T& b) { return a < b ? a : b; }
```

```
template <Comparable T>
T min(const T& a, const T& b, const T& c) {
    auto m = min(b, c);
    return a < m ? a : m;
}
```

```
template <Comparable T>
T min(const T& a, const T& b, const T& c, const T& d) {
    auto m = min(b, c, d);
    return a < m ? a : m;
}
```

3 element overload  
calls 2 element

4 element overload  
calls 3 element

Seems almost recursive!

# Variadic Templates

**Base case function:**

Needed to stop recursion

```
template <Comparable T>
T min(const T& v) { return v; }
```

**Variadic template:**

matches 0 or more types

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

**Parameter pack:** 0 or  
more  
*parameters*

**Pack expansion:**  
replaces ...args  
with actual  
parameters

# What's going on?

Recursive  
Case:

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

Base Case:

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min(2, 7, 5, 1)
```

**Implicit  
instantiation!**

What happens when  
the compiler sees  
a function call  
like this?

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
template <Comparable... Args>
int min(const int& v, const Args&... args) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
template <Comparable... Args>
int min(const int& v, const Args&... args) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

Pack expansion: Args  
is expanded

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
template <Comparable... Args>
int min(const int& v, const int& a0, const int& a1, const int& a2) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
template <Comparable... Args>
int min(const int& v, const int& a0, const int& a1, const int& a2) {
    auto m = min(args...);
    return v < m ? v : m;
}
```

Pack expansion: args  
is expanded

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
int min(const int& v, const int& a0, const int& a1, const int& a2) {
    auto m = min(a0, a1, a2);
    return v < m ? v : m;
}
```

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
int min(const int& v, const int& a0, const int& a1, const int& a2) {
    auto m = min(a0, a1, a2);
    return v < m ? v : m;
}
```

What did we just generate?

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
int min(const int& v, const int& a0, const int& a1, const int& a2) {
    auto m = min(a0, a1, a2);
    return v < m ? v : m;
}
```

Voila! The compiler  
**generated an overload** for  
us!!!

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int, int>(2, 7, 5, 1)
```

T = int  
Args = [int, int, int]

```
int min(const int& v, const int& a0, const int& a1, const int& a2) {
    auto m = min(a0, a1, a2);
    return v < m ? v : m;
}
```

Wait... what is this?  
It's another template instantiation!

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int, int>(a0, a1, a2);
```

T = int

Args = [int, int]

```
int min(const int& v, const int& a0, const int& a1) {
    auto m = min(a0, a1);
    return v < m ? v : m;
}
```

Hey look! Another template instantiation

# What's going on?

```
template <Comparable T, Comparable... Args>
T min(const T& v, const Args&... args)
```

```
template <Comparable T>
T min(const T& v) { return v; }
```

```
min<int, int>(a0, a1);
```

T = int

Args = [int]

```
int min(const int& v, const int& a0) {
    auto m = min(a0);
    return v < m ? v : m;
}
```

Hey look! Another template instantiation

# What's going on?

✓ Compiler always tries to choose most specific template

```
min<int>(a0);
```

```
int min(const int& v) {  
    return v;  
}
```

```
template <Comparable T, Comparable... Args>  
T min(const T& v, const Args&... args)
```

```
template <Comparable T>  
T min(const T& v) { return v; }
```

T = int  
Args = []

T = int

# What just happened?

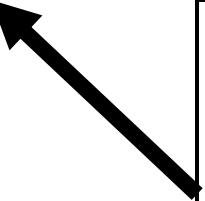
A single call to `min(2, 7, 5, 1)` generated the following functions

```
min(2, 7, 5, 1);

min<int, int, int, int> // T = int, Args = [int, int, int]
min<int, int, int>      // T = int, Args = [int, int]
min<int, int>           // T = int, Args = [int]
min<int>                 // T = int
```

# Variadic templates recap

- Compiler generates any number of overloads using recursion
  - This allows us to support any number of function parameters
- For more advanced variadic templates, check the hidden slides!
- Instantiation happens **at compile time**



Templates do work at compile time. Can we use this to our advantage?

# **Template Metaprogramming**

**How can we do work at compile time?**

# TMP Basics: Factorial

```
template <>
struct Factorial<0> {
    enum { value = 1 };
};
```

**Base Case:**

This is a *template specialization* for N=0

```
template <size_t N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

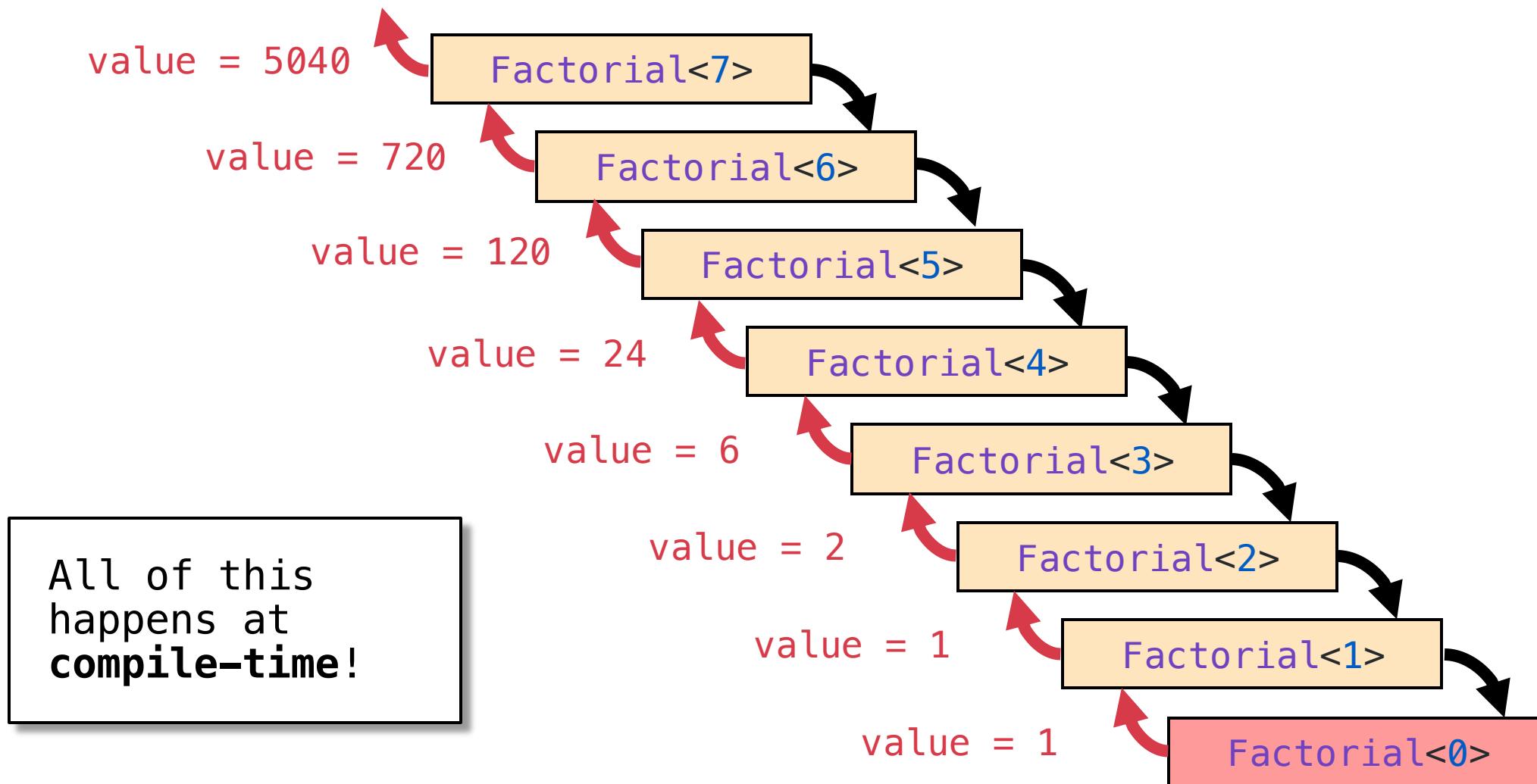
std::cout << Factorial<7>::value << std::endl;
```

**enum:** a way to store a compile-time constant

Oooh compile-time recursion

Prints **5040**, but computes at compile time

# Template instantiations for Factorial<7>



**TMP** is Turing complete

**We can execute arbitrary code at compile time**

# But the syntax is not always pretty...

```
template<>
struct push_back_impl< aux::vector_tag<BOOST_PP_DEC(i_)> >
{
    template< typename Vector, typename T > struct apply
    {
        typedef BOOST_PP_CAT(vector,i_)<
            BOOST_PP_ENUM_PARAMS(BOOST_PP_DE,
            BOOST_PP_COMMA_IF(BOOST_PP_DE
                T
            > type;
    };
};
```



## **How can we have**

- 1) Compile-time execution**
- 2) Readable code**

# Instead of this...

```
template <>
struct Factorial<0> {
    enum { value = 1 };
};

template <size_t N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

std::cout << Factorial<7>::value << std::endl;
```

# Use `constexpr`/`consteval`

An institutionalization of template metaprogramming (new in C++20)

```
constexpr size_t factorial(size_t n) {  
    if (n == 0) return 1;  
    return n * factorial(n - 1);  
}
```

**constexpr**  
“Dear compiler,  
please try to run me  
at compile time 😊”

```
consteval size_t factorial(size_t n) {  
    if (n == 0) return 1;  
    return n * factorial(n - 1);  
}
```

**consteval**  
“Dear compiler, YOU  
MUST RUN ME AT  
COMPILE TIME 😡 😡”

# When should I use **templates**?

- I want the compiler to automate a repetitive coding task
  - Template functions, variadic templates
- I want better error messages
  - Concepts
- I don't want to wait until runtime
  - Template metaprogramming, `constexpr/consteval`