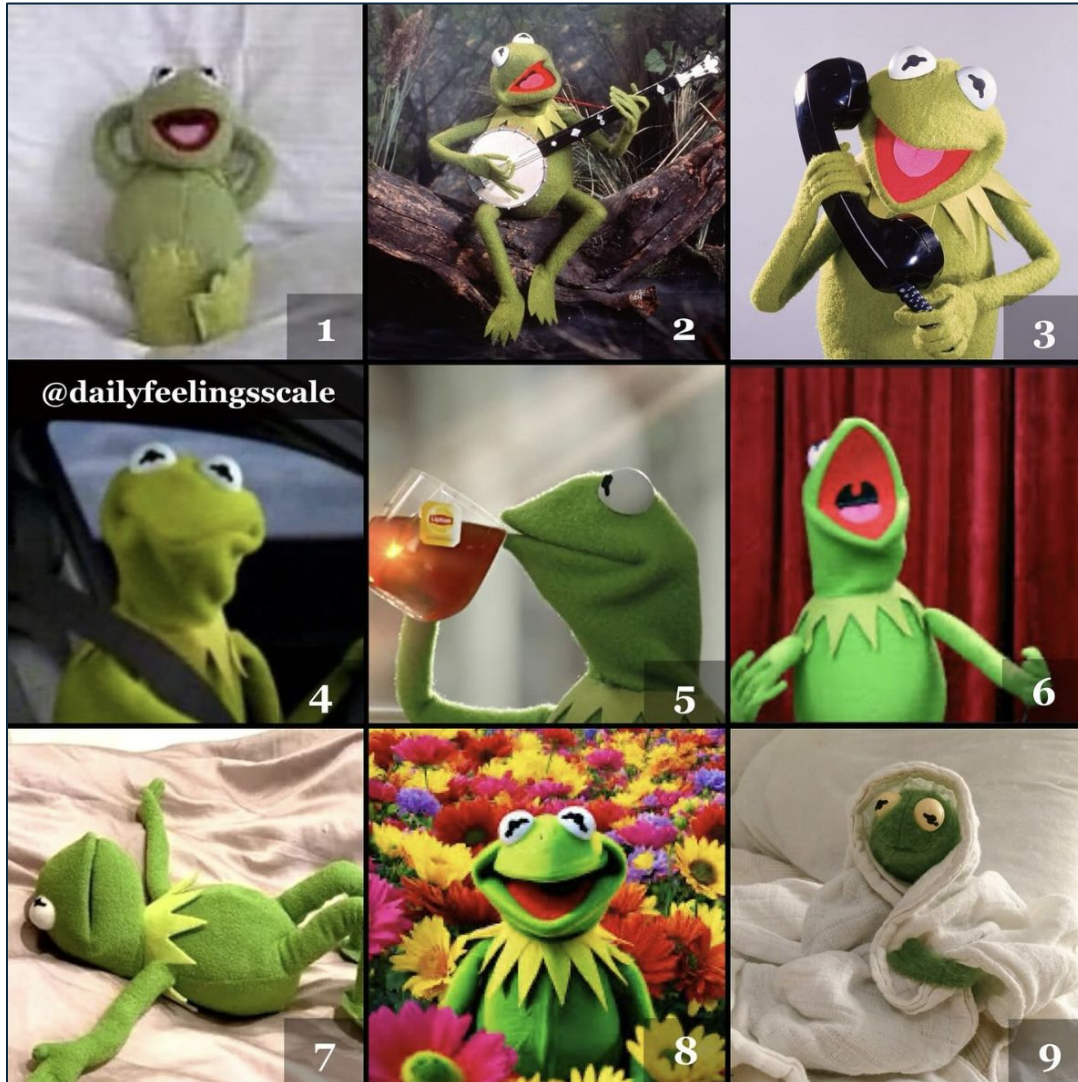


Welcome back! Link to Attendance Form ↓



GUYS HOW ARE WE DOING



tinyurl.com/cs106l-kermit



(forgive me i dont know how to use pollev LOL)

Recall: What are templates?

- Turn to a partner and discuss:
 - What is a template class?
 - Why would you use a template class?
- Introduce yourself and take 60s to talk!



Recall: What are templates?

- What is a template class?
 -  A blueprint for creating classes with generic types
- Why would you use a template class?
 -  Template classes eliminate code redundancy!

Recall: What are templates?

```
class IntVector {
```

```
};
```

```
class DoubleVector {
```

```
};
```

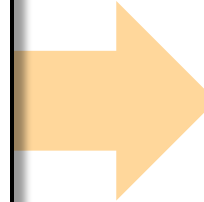
```
class StringVector {
```

```
    // Code to store
```

```
    // a list of
```

```
    // strings...
```

```
};
```



```
template <typename T>
```

```
class vector {
```

```
    // So satisfying.
```

```
};
```

```
vector<int> v1;
```

```
vector<double> v2;
```

```
vector<string> v3;
```

Key Idea: Templates automate code generation

Recall: Template Instantiation

When you write code like this...

```
template <typename T>
class Vector {
    T& at(size_t index);
    // More methods...
};

Vector<int> v;
```

Compiler produces code like this...

```
class IntVector {
    int& at(size_t index);
    // More methods...
};

IntVector v;
```

Is there more that templates can do?

Lecture 10: Template Functions

CS106L, Spring 2026
Preston Seay & Rachel Fernandez

Today's Agenda

- Template Functions
 - How can we extend template classes to functions? Code demo!
- Concepts
 - How can we make C++ templates sane?
- Variadic Templates
 - How do we build functions that accept a variable number of arguments?
- Template Metaprogramming
 - How do we run code at compile time?

We may not get through everything today!

(Slides are posted if you want to review after)

What questions do you have?



bjarne_about_to_raise_hand

Template Functions

Bjarne has a problem...

Hey I really need a way to get the smallest of two values in C++!



Writing a **min** function

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



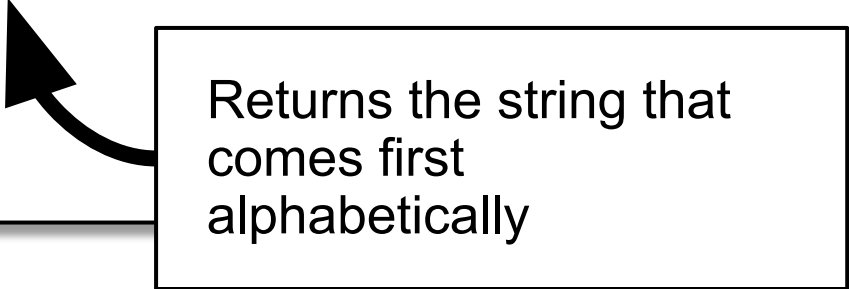
Ternary Operator

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("Preston", "Rachel"); // string, returns "Preston"
```



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...
```

```
};
```

```
};
```

```
};
```

We can use templates!

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**



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

T gets replaced with a specific type

A template is like a factory

int

string



min<int>

min<string>

```
template <typename T>
```

```
T min(T a, T b)
```

Remember: templates vs. functions

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

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

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

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;  
}
```

What questions do you have?



bjarne_about_to_raise_hand

How do we call template functions?

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
}                                  // Compiler generated

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

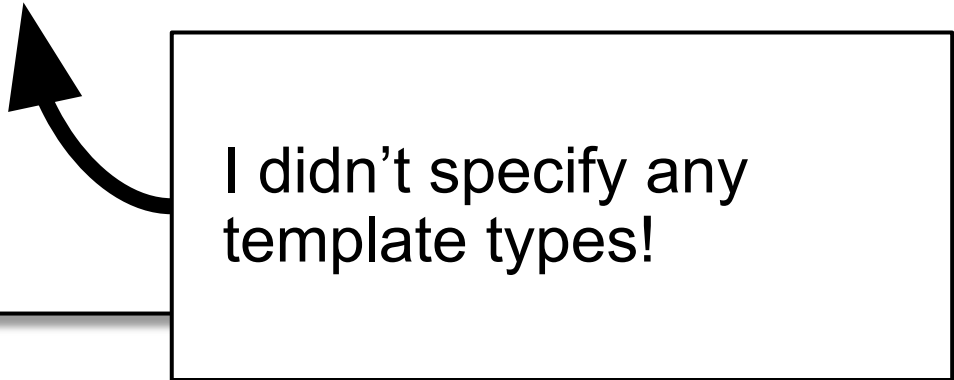
min<int>(106, 107);               // Returns 106
min<double>(1.2, 3.4);           // Returns 1.2
```

Key Idea: Templates automate code generation

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("Preston", "Rachel");
```

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*>("Preston", "Rachel");
```



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>("Preston", "Rachel");
```

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

```
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      auto m = min("Jacob", "Fabio");
10 }
```

const char *min<const char *>(const char *a, const char *b)

What questions do you have?



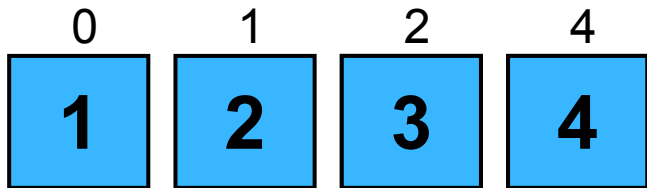
bjarne_about_to_raise_hand

Q: Where do we use template functions in practice?

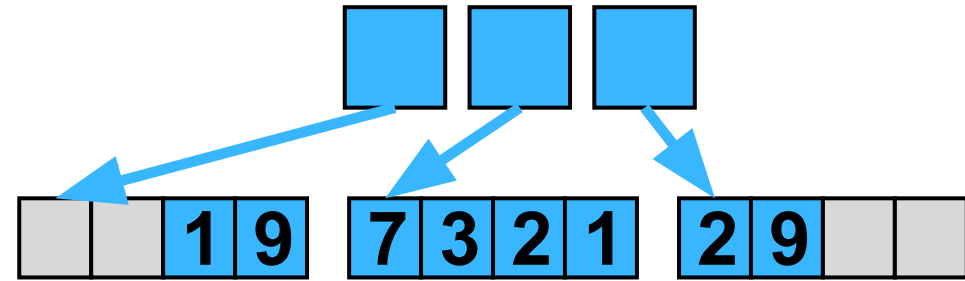
A: All over the place!

One prominent example: iterators

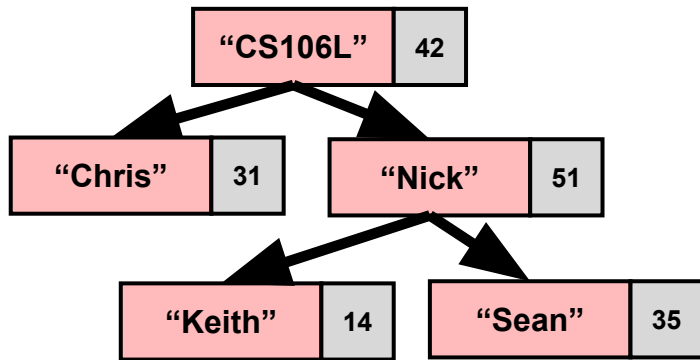
Recall: we have many iterator types!



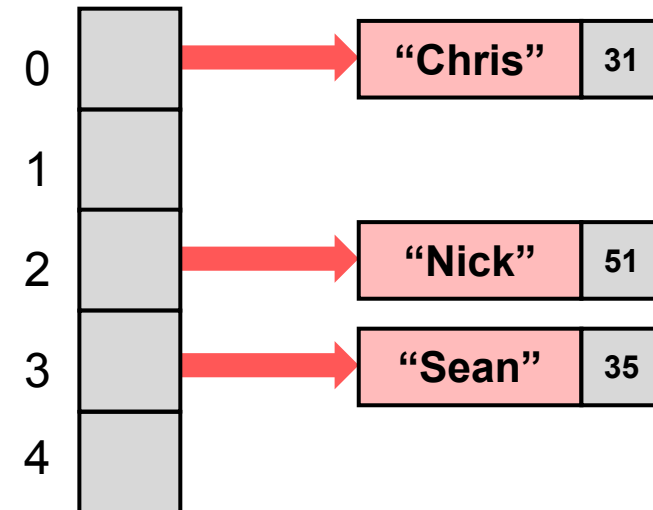
`vector<T>::iterator`



`deque<T>::iterator`



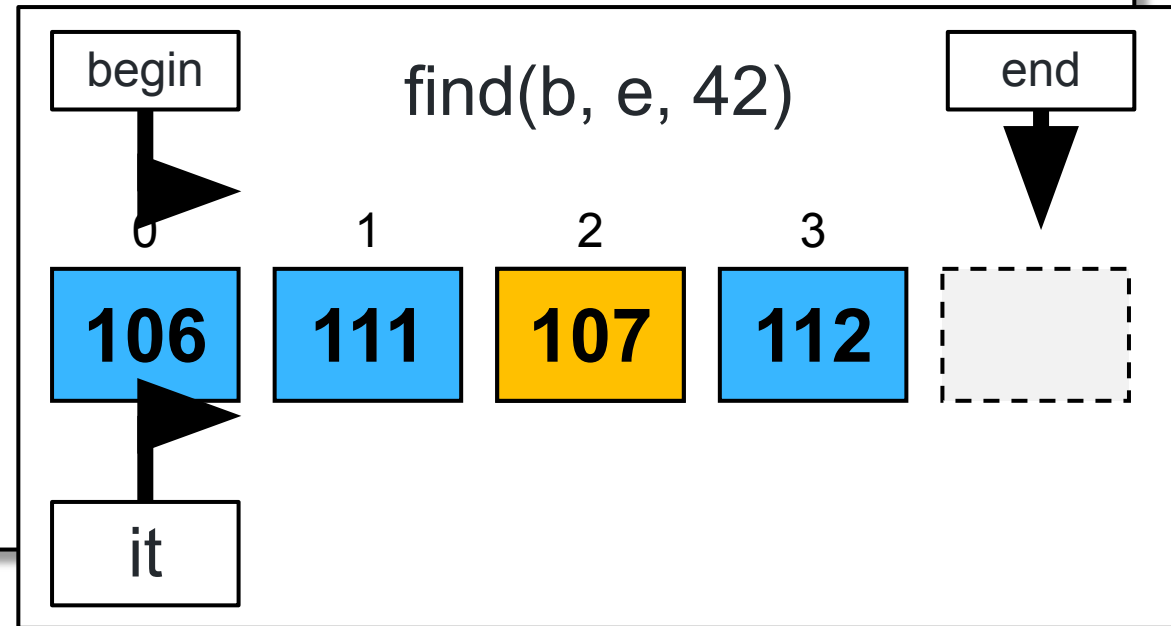
`map<K, V>::iterator`



`unordered_map<K, V>::iterator`

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 in this container  
    // Should return end if no such element is found  
}
```

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");  
// oh man D:
```

What questions do you have?



bjarne_about_to_raise_hand

Let's write a template function!

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

Form a small group and discuss how to implement this function!

```
template <typename Iterator, typename TElem>
???? 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<std::vector<int>::iterator, int>(b, e, 42);
```

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

Let's use the template types!

```
template <typename Iterator, typename TElem>
Iterator find(Iterator begin, Iterator end, TElem 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<std::vector<int>::iterator, int>(b, e, 42);
```

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

Let's implement a simple find function!

```
template <typename Iterator, typename TElem>
Iterator find(Iterator begin, Iterator end, TElem value) {
    Iterator it = begin;
    while (it != end) {
        if (*it == value) break;
        ++it;
    }
    return it;
}
```

```
find<std::vector<int>::iterator, int>(b, e, 42);
```

find function in the STL

- Part of `<algorithm>` header (we'll talk more about this on Thursday)!
- You now have all the tools 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 );
```



BUT WAIT

THERES MORE

makeameme.org

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 preston { "Preston", "pseay" };
```

```
StanfordID rachel { "Rachel", "rfern" };
```

```
min<StanfordID>(preston, rachel); // ❌ 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>(preston, rachel);
```

```
^
```

```
1 error generated.
```

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>(thomas, rachel);
```

```
^
```

```
1 error generated.
```

What happened?

Compiler instantiated our template, and only then did it spot the error

```
StanfordID preston { "Preston", "pseay" };  
StanfordID rachel { "Rachel", "rfern" };  
min<StanfordID>(preston, rachel);
```

Compiler: "min for StanfordIDs, coming right up!"

```
StanfordID min(const StanfordID& a, const StanfordID& b)  
{  
    return a < b ? a : b;  
}
```

Compiler: "AHHH what do I do here! I don't know how to compare two StanfordIDs"

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 { preston, rachel };
```

```
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::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::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:682:25: note: in instantiation of member function 'std::__tree<StanfordID, std::allocator<StanfordID>>::set' requested here
    set(const __key& __k, const __value& __v, const __comp& __comp = less<__key>(), const __allocator& __allocator = allocator<__key, __value>(), const __traits& __traits = traits<__key, __value>())
    ^
```

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: “Uhh..
Compiler, what the @!#* do you
mean?”

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

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)
```

What questions do you have?



bjarne_about_to_raise_hand

Introducing C++ concepts!

Creating a **Comparable** concept

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

Creating a **Comparable** concept

concept: a named set of *constraints*

```
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>;
};
```

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

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!

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);
```



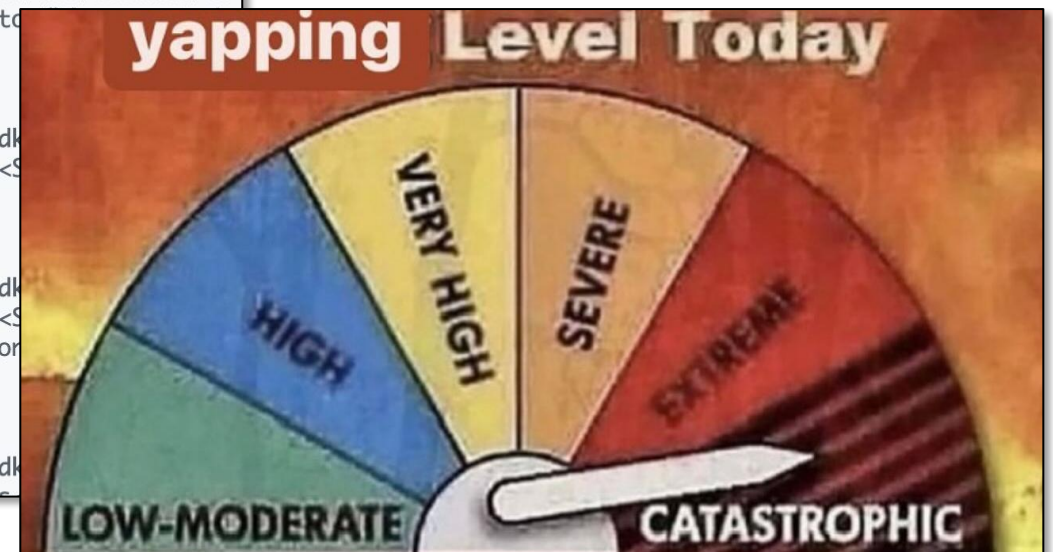
```
// Super slick shorthand for the above  
template <Comparable T>  
T min(const T& a, const T& b);
```

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>::operato
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, cor
>' 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>`

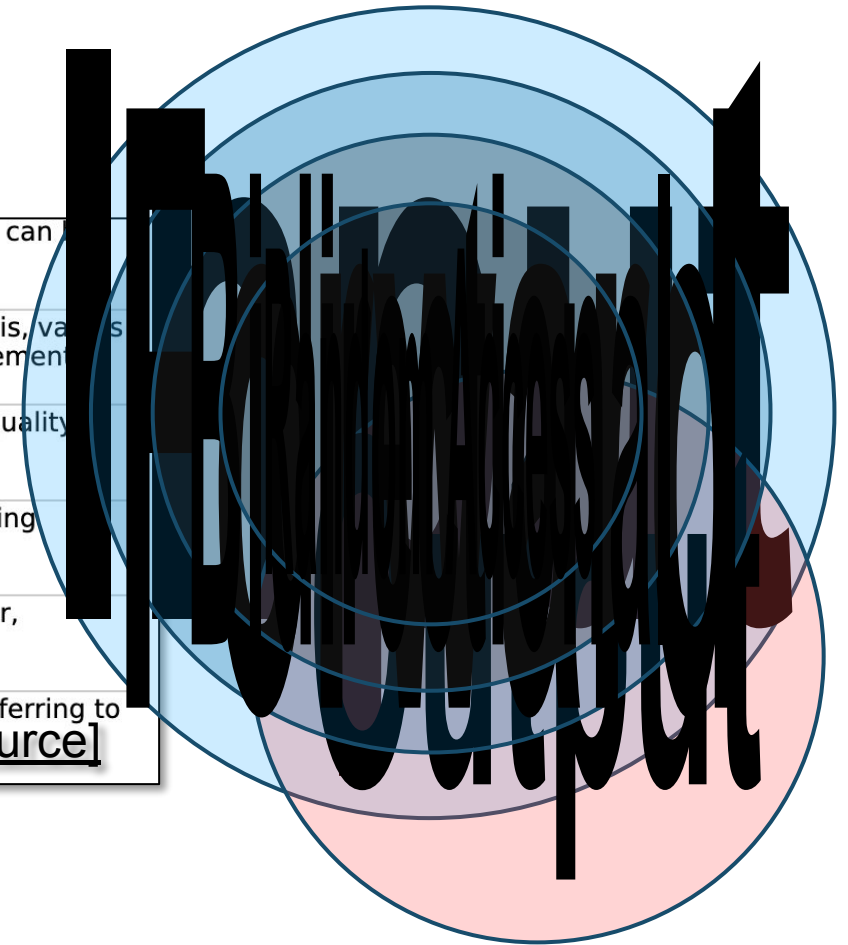
<code>same_as</code> (C++20)	specifies that a type is the same as another type (concept)
<code>derived_from</code> (C++20)	specifies that a type is derived from another type (concept)
<code>convertible_to</code> (C++20)	specifies that a type is implicitly convertible to another type (concept)
<code>common_reference_with</code> (C++20)	specifies that two types share a common reference type (concept)
<code>common_with</code> (C++20)	specifies that two types share a common type (concept)
<code>integral</code> (C++20)	specifies that a type is an integral type (concept)
<code>signed_integral</code> (C++20)	specifies that a type is an integral type that is signed (concept)
<code>unsigned_integral</code> (C++20)	specifies that a type is an integral type that is unsigned (concept)
<code>floating_point</code> (C++20)	specifies that a type is a floating-point type (concept)
<code>assignable_from</code> (C++20)	specifies that a type is assignable from another type (concept)
<code>swappable</code> <code>swappable_with</code> (C++20)	specifies that a type can be swapped or that two types can be swapped with each other (concept)

[source](#)

...including iterator concepts!

<code>input_iterator</code> (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)
<code>output_iterator</code> (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)
<code>forward_iterator</code> (C++20)	specifies that an <code>input_iterator</code> is a forward iterator, supporting equality comparison and multi-pass (concept)
<code>bidirectional_iterator</code> (C++20)	specifies that a <code>forward_iterator</code> is a bidirectional iterator, supporting movement backwards (concept)
<code>random_access_iterator</code> (C++20)	specifies that a <code>bidirectional_iterator</code> is a random-access iterator, supporting advancement in constant time and subscripting (concept)
<code>contiguous_iterator</code> (C++20)	specifies that a <code>random_access_iterator</code> 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>
```

^



Concepts recap

- Two reasons to use concepts
 - Better compiler error messages
 - Better IDE support (Intellisense/autocomplete, etc.)
- Concepts are still a new feature
 - STL does not yet support them fully
 - We'll talk more about this on Thursday!

What questions do you have?



bjarne_about_to_raise_hand

Variadic Templates

**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...
```

Wait... Templates are all about code generation

Can the compiler write the overloads for us?

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 (uniform 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. How does this code work?

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!

What questions do you have?



bjarne_about_to_raise_hand

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 });
```

What we would like to have

```
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!

// This just works!
min(2.4, 7.5, 5.3, 1.2, 3.4, 6.7, 8.9, 9.1);
```

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!

Introducing... variadic templates

Variadic Templates

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

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

Variadic Templates

Base case function:

Needed to stop recursion

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

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

Variadic Templates

Base case function:

Needed to stop recursion

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

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

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;  
}
```

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*

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

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

Phew... this is a lot to unpack

(pun intended)

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; }
```

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



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


```
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; }
```

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



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


```
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; }
```

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



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

```
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; }
```

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

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

```
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; }
```

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



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

```
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; }
```

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

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

```
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; }
```

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



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

```
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; }
```

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

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

```
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; }
```

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



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

```
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; }
```

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

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

```
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; }
```

T = int
Args = [int, int]

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

```
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; }
```

T = int
Args = [int]

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

```
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?

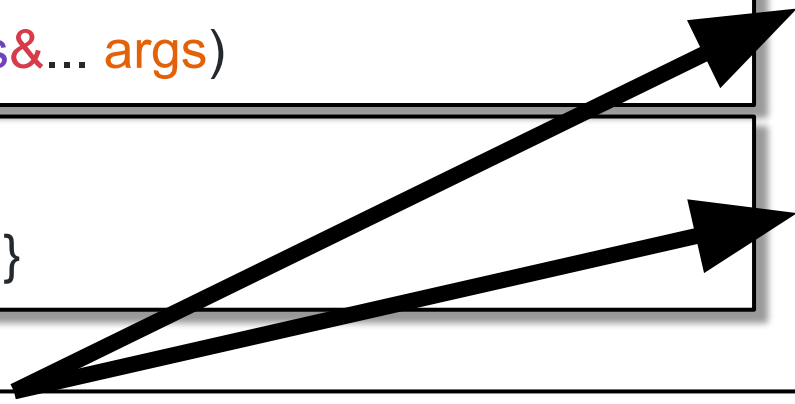
✓ Compiler always tries to choose most specific template

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

```
T = int  
Args = []
```

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

```
T = int
```



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

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

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
```

What questions do you have?



bjarne_about_to_raise_hand

Variadic types don't have to be the same

- In this example, all the **T**'s were the same
- In practice, they don't have to be
- For example, imagine a **printf**-style function like so:
 - `format("Queen {}, Protector of the {} Kingdoms", "Rhaenyra", 7);`
 - The `{}`'s get filled in with arbitrary number/type of arguments

Variadic types don't have to be homogeneous

Imagine we wanted to implement a f-string printer, a la Python

```
format("Queen {}, Protector of the {} Kingdoms", "Rhaenyra", 7);
```

```
// Prints: Queen Rhaenyra, Protector of the 7 Kingdoms
```

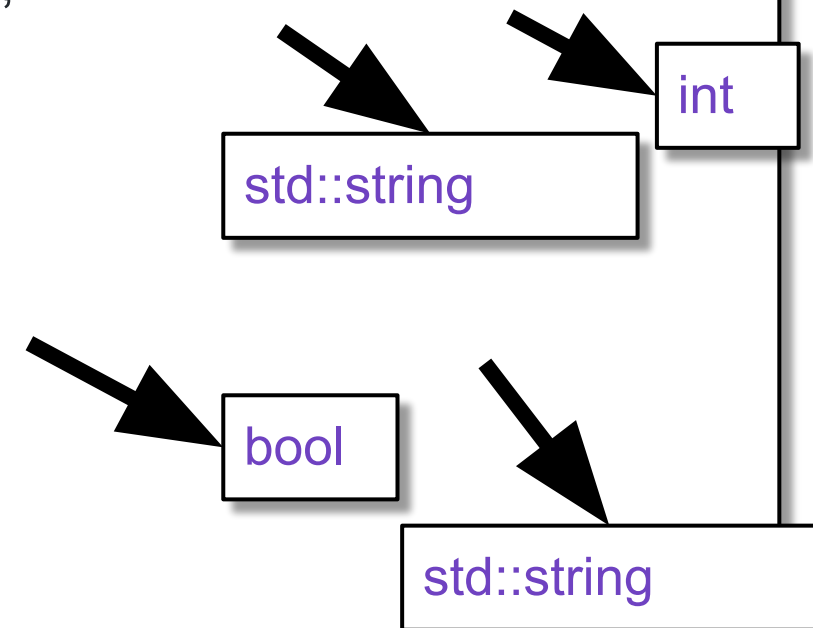
```
std::cout << std::boolalpha;
```

```
format("The {} enemy won't {} out the {}", true, "wait", "storm");
```

```
// Prints: The true enemy won't wait out the storm
```

```
format("Winter is coming");
```

```
// Prints: Winter is coming
```



We can't just use a vector...

What would the underlying type of the vector be?

```
template <typename T>
void format(const std::string& fmt, std::vector<T> args) {
    // ...
}

format("{} {}", { true, "facts" });
// ❌ No common type for vector
```

Sure, this works if all the arguments share the same type, but not if we want *different* types.

Implementing `format`

```
void format(const std::string& fmt) {
    std::cout << fmt << std::endl;
}

template <typename T, typename... Args>
void format(const std::string& fmt, T value, Args... args) {
    auto pos = fmt.find("{}");
    if (pos == std::string::npos) throw std::runtime_error("Extra arg");
    std::cout << fmt.substr(0, pos);
    std::cout << value;
    format(fmt.substr(pos + 2), args...);
}
```

What happens when we instantiate **format**?

```
format("Lecture {}: {} (Week {})", 9, "Templates", 5);
```

```
format<int, std::string, int>()
```

```
// T = int, Args = [std::string, int]
```

```
format<std::string, int>()
```

```
// T = std::string, Args = [int]
```

```
format<int>()
```

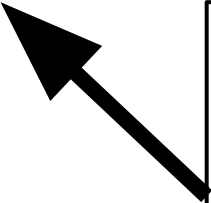
```
// T = int, Args = []
```

```
format()
```

```
// Base case! Not a template, no type arguments
```

Variadic templates recap

- Compiler generates any number of overloads using recursion
 - This allows us to support any number of function parameters
- 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

Base Case:

This is a *template specialization* for N=0

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

enum: a way to store a compile-time constant

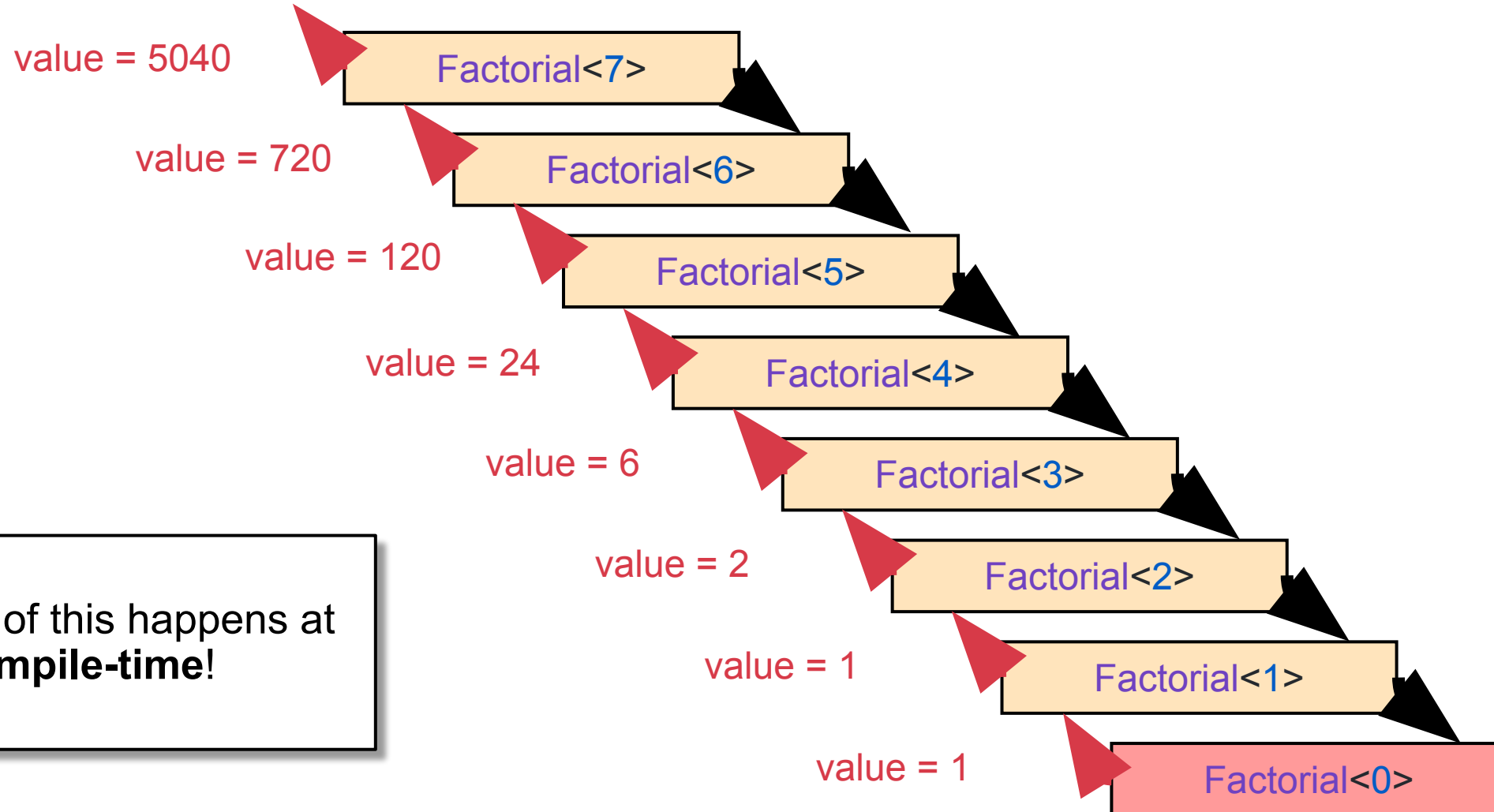
```
template <size_t N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};
```

Oooh compile-time recursion

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

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

Template instantiations for Factorial<7>



All of this happens at
compile-time!

Output assembly of Factorial<7>

```
int main() {  
    std::cout << Factorial<7>::value;  
    return 0;  
}
```

main:

```
push rax  
mov edi, offset cout  
mov esi, 5040  
call ostream::operator<<(int)  
xor eax, eax  
pop rcx  
ret
```

Result is **baked in** to the executable



What questions do you have?



bjarne_about_to_raise_hand

Another example: Fibonacci

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

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

```
template <size_t N>
struct Fibonacci {
    enum { value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value };
};
```

What is TMP?

How is **TMP** used in the real world?

- Baking results into an executable at compile time (e.g. factorial)
- Optimizing matrices/trees/other mathematical structures
- Policy-based design: passing around behaviour through templates
- Boost MPL library

TMP allows programming for types

The `boost::mpl` library is a popular library for metaprogramming

```
using namespace boost;
```

```
using Move = mpl::vector<MoveUp, MoveRight>;
```

```
using MoveRotate = mpl::push_back<Move, Rotate45>::type;
```

```
template <typename Transformations>
```

```
void apply(Object&);
```

```
apply<Move>(object); // move object up and right
```

```
apply<MoveRotate>(object); // move object up/right, rotate 45deg
```

This is a vector of **TYPES!!!**



Compiler generates code specific to the transformations



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_DEC(i_))
            BOOST_PP_COMMA_IF(BOOST_PP_DEC(i_))
            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);  
}
```

```
consteval 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

“Dear compiler, YOU MUST RUN ME AT COMPILE TIME 🤔🤔”

What questions do you have?



bjarne_about_to_raise_hand

Recap

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/constexpr

Next Time: Functions and Algorithms

Writing smarter, more flexible algorithms