

Day 9

Function Templates

Preston Seay & Emily Ai

Template Recap

What are the 3 quirks about templates that we should remember?

1. Write `template<...>` in both `.h` and `.cpp`.
2. Include `.cpp` at the bottom of the header file.
3. `typename` and `class` are equivalent.

Attendance - Grapes vs. Oranges



[https://www.106l.com/live/templates/warm-up?
key=ihMCT46n9x8Uj8ZSd3tNH377wndviWjQIIHBfby9IxA](https://www.106l.com/live/templates/warm-up?key=ihMCT46n9x8Uj8ZSd3tNH377wndviWjQIIHBfby9IxA)



Announcements!

1. Midterm Project releases Thursday
2. Apply to Section Lead!
3. Thursday's a packed lecture

Plan

1. Template Functions
2. Concepts
3. Variadic Templates
4. Bonus: Template Metaprogramming

Motivation

"I want a vector that operates on many types"

```
class IntVector {
```

```
};
```

```
class DoubleVector {
```

```
};
```

```
class StringVector {
```

```
    // Code to store
```

```
    // a list of
```

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

```
};
```



`std::vector`

Motivation

"I want to take the minimum of two **values**."

```
1 int min(int a, int b) {  
2     return a < b ? a : b;  
3 }
```

```
1 double min(double a, double b) {  
2     return a < b ? a : b;  
3 }
```

```
1 string min(string a, string b) {  
2     return a < b ? a : b;  
3 }
```

std::min

Reminder

```
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

You can add type qualifiers too!

```
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;  
}
```

Activity

Write a function template!

Function Template Instantiation

Explicit Instantiation

We explicitly say what type we want in angle brackets.

```
9
10 std::min<int>(10, 6); // Explicit
11 std::min<double>(4.3, 2.1); // Explicit
```

Function Template Instantiation

```
1 int min(int a, int b) {  
2     return a < b ? a : b;  
3 }  
4  
5 double min(double a, double b) {  
6     return a < b ? a : b;  
7 }  
8  
9  
10 std::min<int>(10, 6);  
11 std::min<double>(4.3, 2.1);
```

The compiler generates the code we requested.

Explicit Instantiation

We explicitly say what type we want in angle brackets.

Function Template Instantiation

Don't blink!

Function Template Instantiation

```
1 int min(int a, int b) {  
2     return a < b ? a : b;  
3 }  
4  
5 double min(double a, double b) {  
6     return a < b ? a : b;  
7 }  
8  
9  
10 std::min<int>(10, 6);  
11 std::min<double>(4.3, 2.1);
```

The compiler generates the code we requested.

Explicit Instantiation

We explicitly say what type we want in angle brackets.

Function Template Instantiation

```
1 int min(int a, int b) {  
2     return a < b ? a : b;  
3 }  
4  
5 double min(double a, double b) {  
6     return a < b ? a : b;  
7 }  
8  
9  
10 std::min(10, 6);  
11 std::min(4.3, 2.1);
```

The compiler generates the code we requested.

Implicit Instantiation

We let the compiler figure it out.

Warning: Implicit vs. Explicit

Q: What is the `min` here?

```
1 std::min("Preston", "Emily");
```

A: We don't know!

The compiler assumes we want to compare `const char*`

Solution: Explicit Instantiation

```
1 std::min<std::string>("Preston", "Emily");
```

Warning: Implicit vs. Explicit

Q: What is the `min` here?

```
1 std::min(106L, 107);
```

A: Doesn't compile!

The types aren't both `long` and aren't both `int`
so the compiler can't decide.

Solution: Explicit Instantiation

```
1 std::min<long>(106L, 107);
```

Function Templates with auto

This supports `int` and `long`...

```
1 template<class T, class U>
2 T min(T a, U b) {
3     return a < b ? a : b;
4 } // long min(long, int);
5
6 template<class T, class U>
7 U min(T a, U b) {
8     return a < b ? a : b;
9 } // long min(int, long);
```

But we're duplicating
code again!

Function Templates with auto

This supports `int` and `long`...

```
1 template<class T, class U>  
2 auto min(T a, U b) {  
3     return a < b ? a : b;  
4 }
```

[INSERT MEME HERE
ABOUT HOW COOL
COMPILERS ARE]

And the compiler will just
figure it out for us!

Activity 2

Find and fix the issue:

```
1 template <typename T>
2 T average(T a, T b) {
3     return (a + b) / 2;
4 }
5
6 int main() {
7     std::cout << average(2, 7) << std::endl;
8     return 0;
9 }
```

Plan

1. Template Functions
2. **Concepts**
3. Variadic Templates
4. Bonus: Template Metaprogramming

Our requirements are unclear.

Code Console

C++

Run

```
1 #include <string>
2 #include <complex>
3 template <typename T>
4 long minimum(T a, T b) {
5     return a < b ? a : b;
6 }
7 int main() {
8     int a = 5;
9     int b = 3;
10    return minimum(a, b);
11 }
```

Our requirements are unclear.

```
1 template <typename T>
2 long minimum(T a, T b) {
3     return a < b ? a : b;
4 }
5
6 int main() {
7     minimum(1, 2);
8     return 0;
9 }
```

1. $T < T$
2. `long(T)`

An example from std



<https://en.cppreference.com/cpp/algorithm/find> ↗

An example from std

```
1 template <typename It, typename Value>
2 It find(It begin, It end, Value val) {
3     It current = begin;
4     while (current != end) {
5         if (*current == val) break;
6         ++current;
7     }
8     return current;
9 }
```

When does checking happen?

```
1 template <typename It, typename Value>
2 It find(It begin, It end, Value val) {
3     It current = begin;
4     while (current != end) {
5         if (*current == val) break;
6         ++current;
7     }
8     return current;
9 }
10
11 int main() {
12     std::vector<int> nums {4, 2, 6, 7};
13     find(nums.begin(), nums.end(), 6);
14     return 0;
15 }
```

When does checking happen?

```
1 template <typename It, typename Value>
2 It find(It begin, It end, Value val) {
3     It current = begin;
4     while (current != end) {
5         if (*current == val) break;
6         ++current;
7     }
8     return current;
9 }
10
11 int main() {
12     find(1, 3, 6);
13     return 0;
14 }
```

When does checking happen?

```
1  /* Compiler generates this. */
2  int find(int begin, int end, int val) {
3      int current = begin;
4      while (current != end) {
5          if (*current == val) break;
6          ++current;
7      }
8      return current;
9  }
10 /* We wrote this. */
11 int main() {
12     find(1, 3, 6);
13     return 0;
14 }
```

Template Errors are Still Ugly

```
/main.c:5:9: error: indirection requires pointer operand ('int' invalid)
   5 |     if (*current == val) break;
      |         ^~~~~~
/main.c:12:3: note: in instantiation of function template specialization 'find<int, int>' requested here
   12 |     find(1, 3, 6);
      |         ^
1 error generated.
```

5-line function, 7 line error

```
67 |     __loc->~Tp();
    |     ^
include/c++/v1/__vector/vector.h:776:23: note: in instantiation of function template specialization 'std::allocator_traits<std::allocator<std::vector<std::complex<double>>>>::destroy<std::vector<std::complex<double>>>, 0>' requested here
   776 |     __alloc_traits::destroy(this->__alloc_, std::__to_address(--__soon_to_be_end));
    |     ^
include/c++/v1/__vector/vector.h:564:5: note: in instantiation of member function 'std::vector<std::vector<std::complex<double>>>>::__base_destruct_at_end' requested here
   564 |     __base_destruct_at_end(this->__begin_);
    |     ^
include/c++/v1/__vector/vector.h:249:16: note: in instantiation of member function 'std::vector<std::vector<std::complex<double>>>>::clear' requested here
   249 |     __vec_.clear();
    |     ^
include/c++/v1/__vector/vector.h:260:67: note: in instantiation of member function 'std::vector<std::vector<std::complex<double>>>>::__destroy_vector::operator()' requested here
   260 |     _LIBCPP_CONSTEXPR_SINCE_CXX20 _LIBCPP_HIDE_FROM_ABI ~vector() { __destroy_vector (*this)(); }
    |     ^
main.c:18:52: note: in instantiation of member function 'std::vector<std::vector<std::complex<double>>>>::~~vector' requested here
   18 |     std::vector<std::vector<std::complex<double>>>> bad;
    |     ^
include/c++/v1/__fwd/complex.h:22:7: note: template is declared here
   22 | class complex;
    | ^
n file included from /main.c:1:
n file included from /include/c++/v1/vector:314:
include/c++/v1/__vector/vector.h:733:69: error: implicit instantiation of undefined template 'std::complex<double>'
   733 |     std::__annotate_contiguous_container<Allocator>(data(), data() + capacity(), __old_mid, __new_mid);
    |     ^
include/c++/v1/__vector/vector.h:749:5: note: in instantiation of member function 'std::vector<std::complex<double>>>::__annotate_contiguous_container' requested here
   749 |     __annotate_contiguous_container(data() + __old_size, data() + size());
    |     ^
include/c++/v1/__vector/vector.h:565:5: note: in instantiation of member function 'std::vector<std::complex<double>>>::__annotate_shrink' requested here
   565 |     __annotate_shrink(__old_size);
    |     ^
include/c++/v1/__vector/vector.h:249:16: note: in instantiation of member function 'std::vector<std::complex<double>>>::clear' requested here
```

1-line function, 70 line error

Concepts Make Requirements Clear!

We support any type that may work with `typename` or `class`.

We can also specify **concepts**...
sets of constraints that are
checked at compile time.

Making a Comparable Concept

We name this set of constraints `Comparable`.

For all `a` and `b` of type `T`, we expect the following to hold:

"We must be able to say `T < T`" concept:

```
1 template <typename T>
2 concept Comparable = requires(T a, T b) {
3     a < b } -> std::convertible_to<bool>;
4 };
```

This must compile...

...and it must be convertible to a `bool`.

Using a 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 Make Errors Nicer

```
/main.c:20:5: error: no matching function for call to 'minimum'
 20 |     minimum(a, b);
    |           ~~~~~
/main.c:12:6: note: candidate template ignored: constraints not satisfied [with T = std::complex<int>]
 12 | auto minimum(T a, T b) {
    |      ^
/main.c:11:10: note: because 'std::complex<int>' does not satisfy 'Comparable'
 11 | requires Comparable<T>
    |      ^
/main.c:7:9: note: because 'a < b' would be invalid: invalid operands to binary expression ('std::complex<int>' and 'std::complex<int>')
  7 |     { a < b } -> std::convertible_to<bool>;
    |           ^
1 error generated.
```

std Has Many Concepts



<https://cppreference.com/cpp/concepts> ↗

std Has Many Concepts



https://en.cppreference.com/cpp/iterator#iterator_concepts_.28since_C.2B.2B20.29 ↗

Proper Concepts = Nice Errors

```
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

Plan

1. Template Functions
2. Concepts
3. **Variadic Templates**
4. Bonus: Template Metaprogramming

Goal: A min that supports N arguments.

```
1 min(5); // 5
```

```
2 min(6, 7); // 6
```

```
3 min(106, 107, 111, 112, 143, 144); // 106
```

Bad Solution: Many overloads...

```
1 template <Comparable T>
2 T min(T a, T b) {
3     return a < b ? a : b;
4 }
5
6 template <Comparable T>
7 T min(T a, T b, T c) {
8     return a < b ? a < c ? a : c : c < b ? c : b;
9 }
10
11 // and many more...
```

Improved Bad Solution: Many overloads...

```
1 template <Comparable T>
2 T min2(T a, T b) {
3     return a < b ? a : b;
4 }
5
6 template <Comparable T>
7 T min3(T a, T b, T c) {
8     return min2(a, min2(b, c));
9 }
10
11 // and many more...
```

Improved Bad Solution: Many overloads...

```
1 // many before ...
2
3 template <Comparable T>
4 T min26(T a, T b, /* c-y */, T z) {
5     return min2(a, min25(b, c, /* c-y */, z));
6 }
7
8 // and many more...
```

Activity

Which 106B concept does
this remind you of?

Base Case & Recursive Case

```
1 // Base Case:
2 template <Comparable T>
3 T min(const T& v) { return v; }
4
5 // Recursive Case:
6 template <Comparable T, Comparable... Args>
7 T min(const T& v, const Args&... args) {
8     auto m = min(args...);
9     return v < m ? v : m;
10 }
```

Base Case & Recursive Case

```
1 // Base Case:
2 template <Comparable T>
3 T min(const T& v) { return v; }
4
5 // Recursive Case:
6 template <Comparable T, Comparable... Args>
7 T min(const T& v, const Args&... args) {
8     auto m = min(args...);
9     return v < m ? v : m;
10 }
```

Variadic Template: matches 0 or more types

Parameter Pack: takes 0 or more parameters.

Pack Expansion: passes all of the args.

```
1 template <Comparable T>
2 T min(const T& v) { return v; }
3
4 template <Comparable T, Comparable... Args>
5 T min(const T& v, const Args&... args) {
6     auto m = min(args...);
7     return v < m ? v : m;
8 }
9
10 // What happens to: min(6, 7, 8, 9) ?
```

Let's follow along...

Variadic Functions Are Flexible

Code Console

C++

Run

```
1 #include <print> // From C++23
2
3 int main() {
4     // Variadic function with 0 extra args.
5     std::print("Hello, world!");
6
7     // Variadic function with 2 extra args.
8     std::println("And hello {}{}!", 106, 'L');
9     return 0;
10 }
```

Variadic Functions Activity

Plan

1. Template Functions
2. Concepts
3. Variadic Templates
4. **Bonus: Template Metaprogramming**

Template Metaprogramming

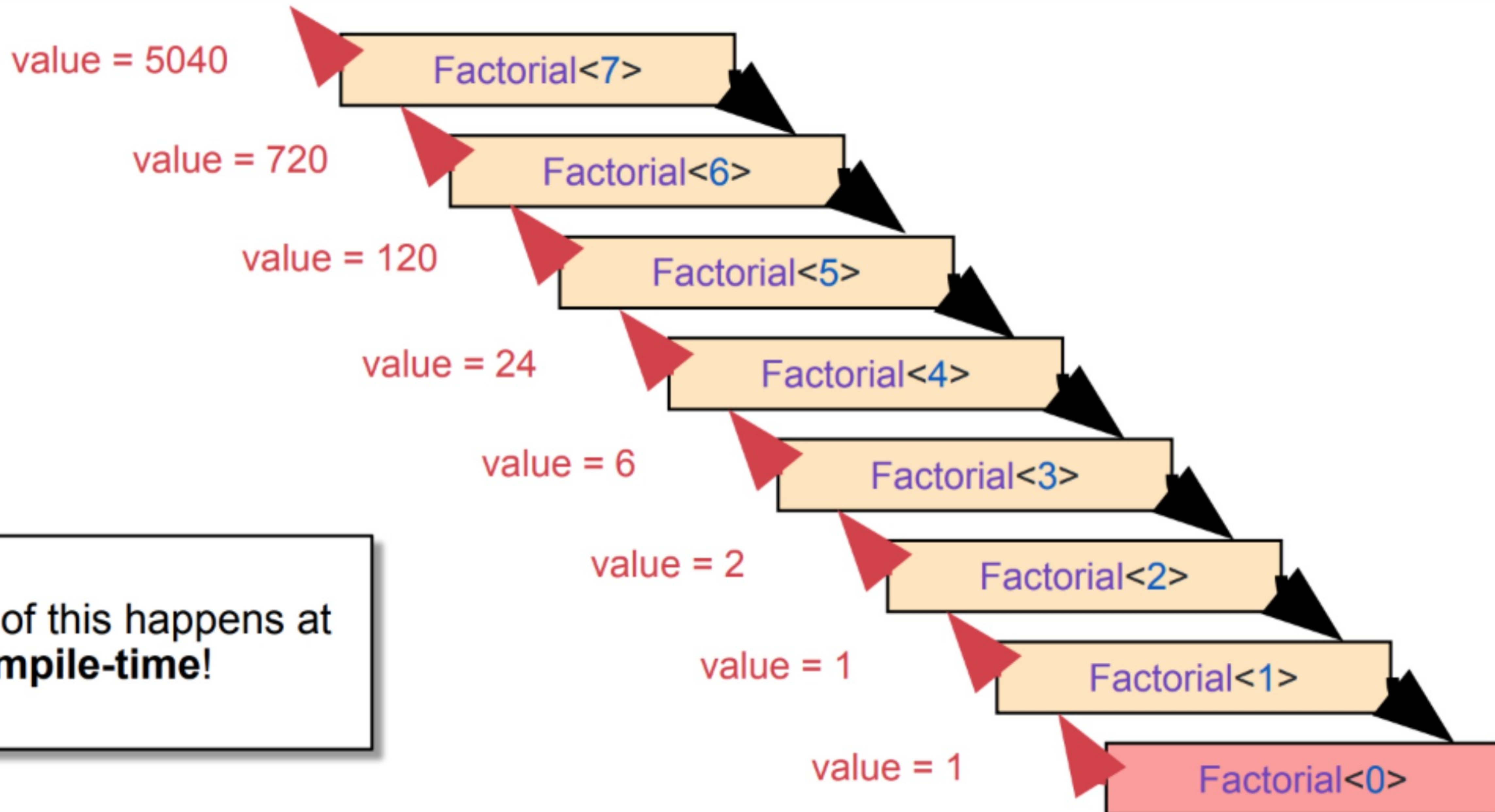
Code Console

C++

Run

```
1 #include <iostream>
2
3 // Template for the recursive case:
4 template<int N>
5 struct Factorial {
6     enum { value = N * Factorial<N - 1>::value };
7 };
8 // "Template Specialization" for the base case.
9 template<>
10 struct Factorial<0> {
11     enum { value = 1 };
12 };
13
14 int main() {
15     // Calculated at compile time!
16     std::cout << Factorial<5>::value << std::endl;
17 }
```

Template Metaprogramming



All of this happens at
compile-time!

Template Metaprogramming IRL

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

Neat Compile-Time Execution with `constexpr` and `constexpr`

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 🙏”

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

constexpr

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

When to 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