lecture10

what:template function
what more:
Template Functions
Concepts
Variadic Templates
Template Metaprogramming

Recall:

> 2025Fall-10-TemplateFunctions, p.5
>
> > Key Idea: Templates automate code generation

template function:

之前是template class,现在是template function
说白了就是把之前的int function写成template function
ex:

```
min(a,b)
min<int>(a,b);
```

> 2025Fall-10-TemplateFunctions, p.26
>
> > Option A: explicit instantiation(显式实例化)

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
```

2025Fall-10-TemplateFunctions, p.29

> Option B: implicit instantiation（隐式实例化）

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!

这种想法很像auto

auto的技巧

```
auto a=min(106,3.14)
```

但是这种做法有点风险
如果输入的数据不符合函数内部要求呢

Implicit instantiation can be finicky

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

min("Thomas", "Rachel");
```

const char*

What type is T? What are the types of the arguments?

Hint: you might know this if you've taken CS107!

针对这个字符串，我们可以用const char*（本质上是指针比大小） 也可以用 string

还有关于auto的问题

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);
```

What should the return type of this function be?

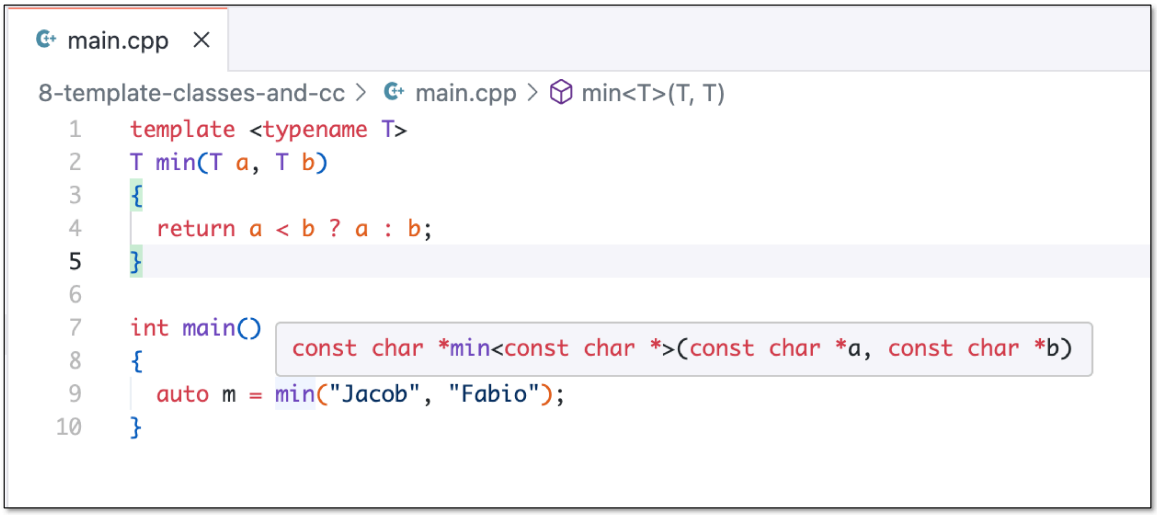
T = int

U = double

这种方法解决歧义

Pro tip: Use IDE to see instantiation types

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



2025Fall-10-TemplateFunctions, p.40

> Q: Where do we use template functions in practice?

尽量都用，举个例子

用vector写的find函数

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
}
```

且使用string类型的find，还得重写

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
```

使用template function重写

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);
```

find()大家族

`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 );
```

concepts 新概念

个人感觉这个东西就是保护
回到min()

Back to our `min` function

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

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

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

我们不能避免传入min()的参数一定是符合要求的

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 thomas { "Thomas", "tpoimen" };
StanfordID rachel { "Rachel", "rfern" };
min<StanfordID>(thomas, rachel); // ❌ Compiler error
```

这种传入个struct，这种情况不满足<>重载

局限性：

> 2025Fall-10-TemplateFunctions, p.61
>
> > Compiler only finds the error after instantiation

同时，有时候c++的报错涉及template就很复杂，需要工具让我们定位才行
为了防止这种情况，设计了concept

以c#和java举例

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>>
```

我们希望c++有类似功能

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)

template <typename T>
struct set;

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

T must have operator<

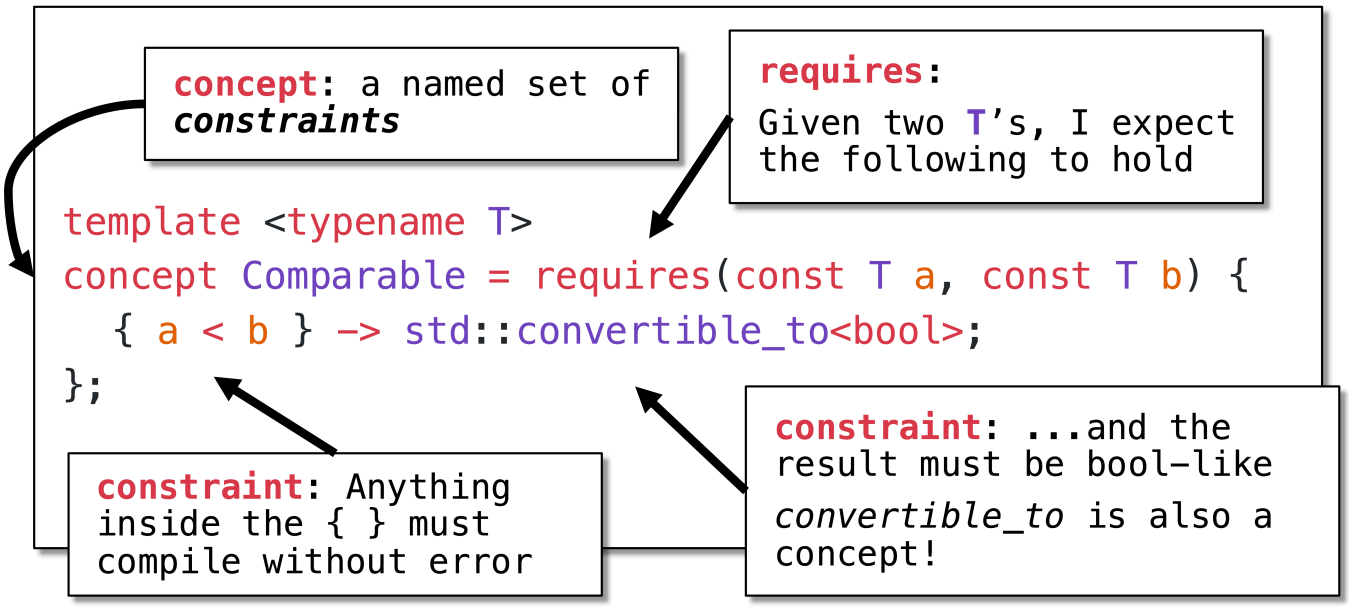
It must be an iterator type

concept语法

Creating a Comparable concept

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

Creating a Comparable concept



不同的写法

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);
```

这种很好的看清了报错

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;
```

升级后的find()

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>
          ^
```

variadic templates

总结：就是为了解决传入参数不定的问题

min(a,b)

min(a,b,c)

min(a,b,c,d)

一种解决办法（原始）

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!

使用template解决（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) {
  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;
}
```

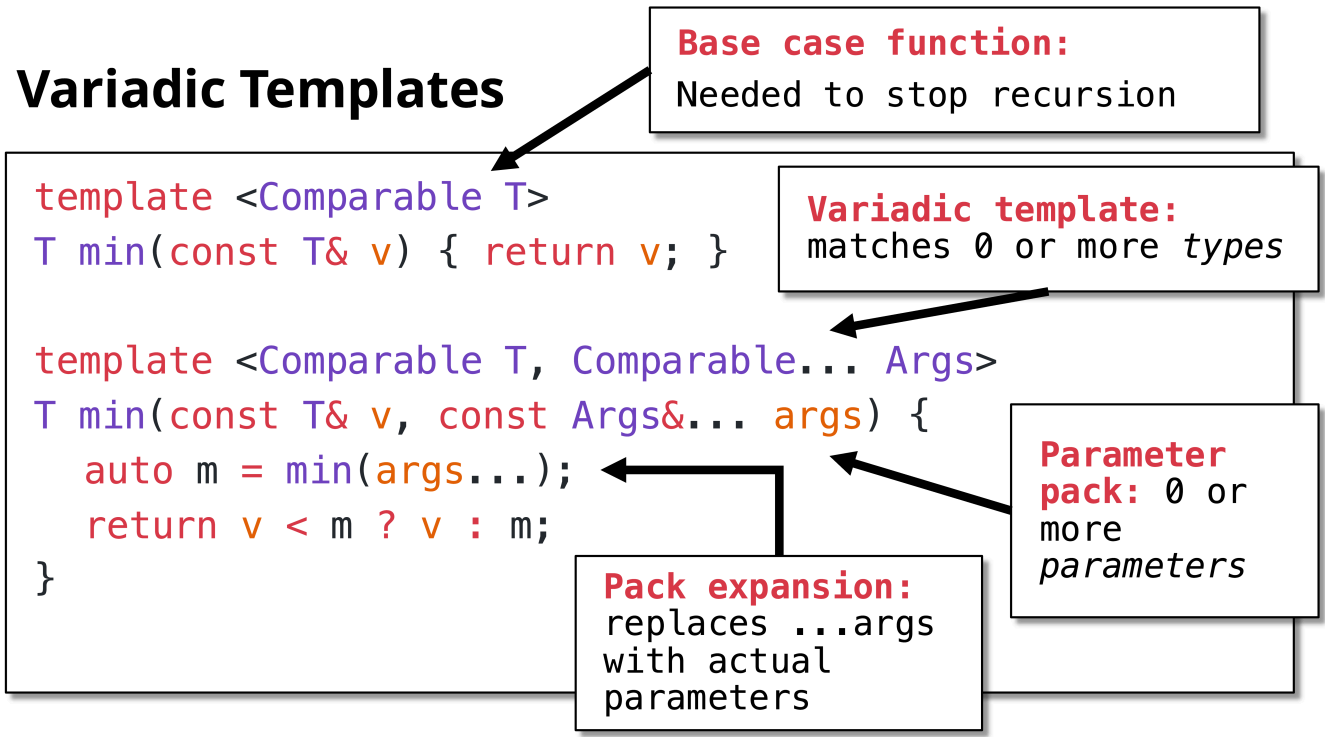
Talk to a partner for 60s. What should the last two lines be?

variadic templates做法

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



这种写法的好处是可以那个...里面是其他类型的T

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

因此引出了format

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...);
}
```

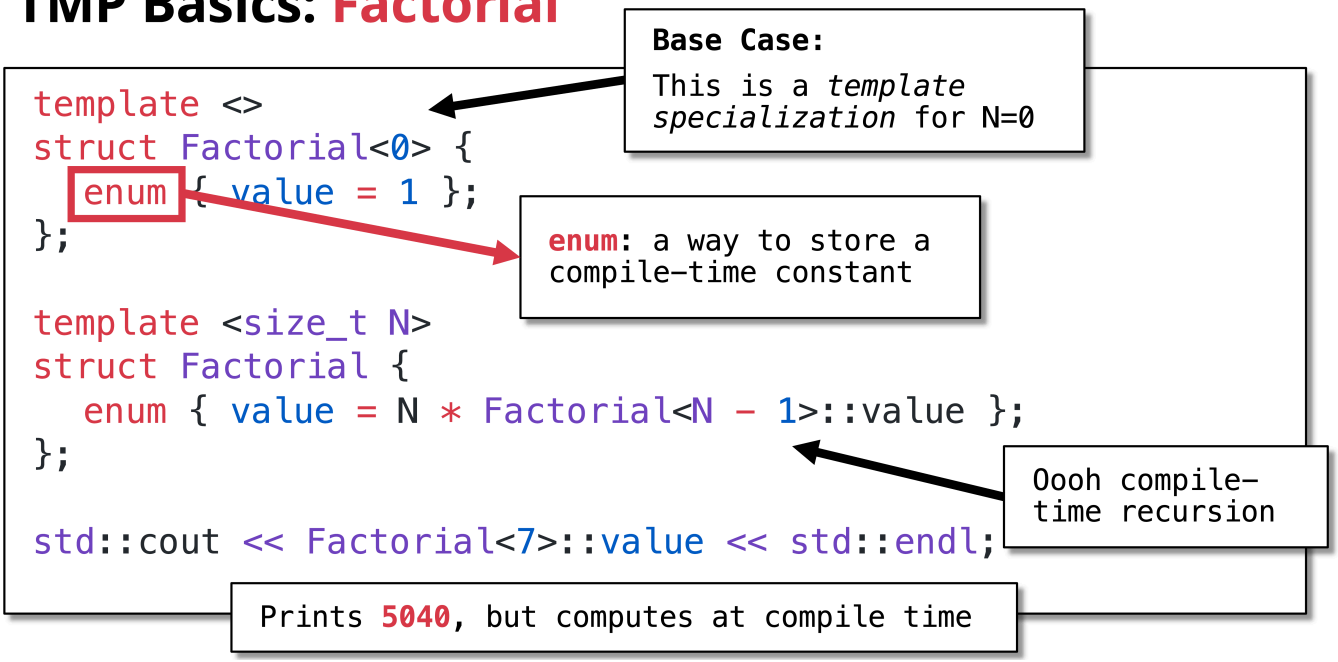
What happens when we instantiate format?

```
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
```

Template Metaprogramming（模板元编程）

用c++模板在编译期间做计算（在运行钱算好了）

TMP Basics: Factorial




```
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;
```

汇编

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

但是这种TMP的写法很晦涩，比如

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
         BOOST_PP_COMMA_IF(BOOST_PP_DE
         T
         > type;
   };
};
```

引入关键字：constexpr/consteval

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 😡😡"

constepr:尽量用
consteval:必须用

Racap

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