

模板基础与移动语义

---

Template Basics and  
Move Semantics

# 现代C++基础 Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

---

- **Template Basics**
  - **Compile-time Evaluation**
  - **Compile-time Branch Selection**
    - **Specialization**
    - **Overload resolution**
  - **Tricky Details**
  - **Concepts**
- **Final part of Move Semantics**
  - **Universal Reference and Perfect Forwarding**

# Supplementary

- Before starting, we'd supplement some basic knowledge.

## 1. Since C++14, it's allowed to define variable template.

- E.g.

```
template<class T>  
const T pi = T(3.1415926535897932385L);
```

- Though it's **const**, it has external linkage (since it's a template).
- Non-static variable template cannot be defined inside a class.

## 2. Member function template cannot be virtual.

- Intuitive reason: compiler need to determine the size of vtable; so it needs to know number of virtual functions.
  - But function template can be instantiated freely.

# Template Basics

Compile-time Evaluation

# Motivation

- **const** has two functionalities in C++:
  - Not changeable;
  - Possibly can be determined in compile time.

- For example: `void Test(int a)`

```
{  
    const int b = a;  
    const int size = 10;  
    int arr[size]{}; // legal  
    // int arr[b]{}; // illegal, b isn't compile-time value.  
}
```

- Sometimes we hope to force the variable to be determined in compile time.

# constexpr variable

- So we need **constexpr**!

```
void Test(int a)
{
    // constexpr int b = a; // compile error
    constexpr int size = 10;
    int arr[size]{}; // legal
}
```

- **constexpr** implies **const**, so it has internal linkage in global field.
  - Normally in header file/class it will also be decorated with **inline**.
- So what if we want the initial value determined in compile time, while make it changeable afterwards?
  - That is, **constexpr** without **const**!

# constexpr

- That's **constexpr** (since C++20).
  - You can only use **constexpr** for global / static / thread-local variables.

```
static constexpr int a = 10;

void Test(int a)
{
    // static constexpr int b = a;  // compile error
    static constexpr int size = 10;
    // int arr[size]{}; // illegal, since size isn't const.
}
```

- **constexpr** can help to solve *static initialization order fiasco*.
  - That is, the initialization order of global variables in different TUs is not determined; so you cannot let one initialization rely on the other.

# constinit

- Example:

```
// a.h
extern int a;
```

```
// a.cpp
int a = 1;
```

```
// b.cpp, #include "a.h"
static int b = a; // a may have uninitialized value
```

- But **constinit** ensures by compilers that when it's used, it's definitely initialized.
  - For non-compile-time initialization, you still need singleton pattern.

```
// a.h
extern constinit int a;
```

```
// a.cpp
constinit int a = 1;
```

```
// b.cpp, #include "a.h"
static int b = a; // a == 1
```



# constexpr function

- We may want complex compile-time computation, so we need functions executed in compile time...
- And that's `constexpr` function.
- The history:
  - C++11 – only one line; `constexpr` function can only contain a return statement & type aliases & `static_assert`.
  - C++14 – allow multiple lines, e.g. branches like loop & condition;
  - C++20 – allow try – catch block, virtual function.
    - throwing an exception will lead to compile error.
  - C++23 – allow goto, use non-constexpr variables, static & thread-local variable.

Before C++23, `constexpr` ctor has less requirements than other `constexpr` functions; but C++23 makes them unified.

# constexpr function

- So C++23 requirements are quite simple:

A **constexpr function** must satisfy the following requirements:

- it must not be a **coroutine**
  - for constructor and destructor, the class must have no virtual base classes
- For example, we want to know whether a number is prime at compile time.
    - In C++11, you have to use recursion to do it in a single return:

```
constexpr bool DoIsPrime(unsigned int a, unsigned int b)
{
    return b == 1 ? true : (a % b != 0 && DoIsPrime(a, b - 1));
}

constexpr bool IsPrime(unsigned int a)
{
    return a <= 1 ? false : DoIsPrime(a, a / 2);
}
```

# constexpr function

- And in C++14, you can use loop to do it:

```
constexpr bool IsPrime(unsigned int a)
{
    if (a <= 1) return false;

    for (unsigned int i = 2; i <= a / 2; i++)
    {
        if (a % i == 0)
            return false;
    }
    return true;
}
```

# constexpr & consteval function

- Unlike `constexpr` variables, `constexpr` functions are allowed to not get the value at the compile time.

```
constexpr bool p1 = IsPrime(2); // must be evaluated at compile time
bool p2 = IsPrime(3); // may or may not be evaluated at compile time
bool p3 = IsPrime(argc); // Okay, runtime
// constexpr bool p4 = IsPrime(argc); // compile error
```

But normally it is, due to compiler optimization.

- If you want to **force** the function to be evaluated at compile time, you need `consteval`.

```
consteval bool IsPrime(unsigned int a)

constexpr bool p1 = IsPrime(2); // must be evaluated at compile time
bool p2 = IsPrime(3); // must be evaluated at compile time
// bool p3 = IsPrime(argc); // compile error
// constexpr bool p4 = IsPrime(argc); // compile error
```

# constexpr & consteval lambda

- These two specifiers can also be added in lambda.

- `constexpr` since C++17, `consteval` since C++20.

- E.g. 

```
auto isPrime = [](unsigned int a) consteval -> bool {  
    return true;  
};
```

- Notice that if all operations in lambda is `constexpr`, `constexpr` is implied (so explicit specification can be omitted).

- E.g. 

```
auto isPrime = [](unsigned int a) -> bool {  
    return true;  
};  
constexpr bool b = isPrime(1);
```

# Template Basics

Compile-time Branch Selection

# Compile-time Branch Selection

- We've known branch since we're novices...
  - However, code path selection only happens at runtime, depending on the value of the condition.
- But we've already learnt compile-time evaluation...
  - Correspondingly, we could choose to execute some code or not at compile time. Non-taken branch can be completely eliminated!
- There are several ways to do so:
  - By specialization, so when some conditions are met, only one of the specializations will be chosen.
    - And for functions, there is no partial specialization so overload may be needed.
  - By control statement in a code block, e.g. `constexpr if`.

# Template Basics

- Compile-time branch selection
  - Function overload resolution and specialization
  - Class specialization
  - Selection in code block



# Template specialization

- A simple template function:

```
template<typename T>
void Func(T arg)
{
    std::println("Here {}. ", arg);
}
```

- What if we want to output “There!” when T is int?

- By specialization!

```
template<>
void Func<int>(int arg)
{
    std::println("There {}!", arg);
}
```

```
Func<char>('a'); Here a.
Func<int>(1);   There 1!
```

- From `int arg`, compilers can deduce the specialized type and we can just write:

```
template<>
void Func(int arg)
{
    std::println("There {}!", arg);
}
```

# Template specialization

- Note1: don't mistake it from explicit template instantiation.
  - Instantiation: no `<>` after `template`. `template void Func<int>(int arg);`
    - Instantiation can also eliminate `<int>` here since it could be deduced.
- Note2: a specialization must be declared before it's used, otherwise the behavior is implementation-defined.
  - For example, the compiler could generate according to the primary template since it doesn't see specialization here.
  - Or, it could search for specialization globally and use it directly.
- Note3: a full specialization isn't a template anymore; thus you cannot define it in header file (re-definition).
  - You can either use `inline`, or only write the specialization prototype.

```
template<>
inline void Func(int arg)
{
    std::println("There {}!", arg);
}
```

```
template<> void Func(int arg);
```

# Template specialization

- You can add new specifiers (e.g. `inline`, `constexpr`) since it can be viewed as a new function.
- For class specialization, since class can be written directly in header file, it's Okay.
- Note4: default template parameter can be omitted when writing specialization.

```
template<typename T = int>
void Func();
// Equiv to Func<int>
template<> void Func()
{
    std::cout << "TestIt!";
}
```

# Template specialization

- Note5: you can also specialize template member function, but it's not allowed to be defined inside class.
  - \*But msvc and clang allow to do so.

```
class A
{
    template<typename T>
    void Func() {}

    template<> void Func<int>() { }
```



```
class A
{
    template<typename T>
    void Func() {}
};

template<> void A::Func<int>() { }
```



- These notes also apply on class template specialization.

# Type Deduction

- Similarly, you don't need to write `<...>` when calling it if they could be deduced from the parameters.

- Non-deducible template type parameters may be written first to minimize explicit ones.

```
Func('a');  
Func(1);
```

```
template<typename U, typename V, typename T>  
V Func2(T a)  
{  
    std::vector<U> vec;  
    return V{};  
}
```

Func2<int, double>(0); // T isn't needed explicitly

Notice that the returned type **cannot be deduced from the caller**, e.g. `double b = Func2<...>(...)` cannot deduce `V` as `double`.

- By contrast: here `T` must be specified explicitly.

```
template<typename T, typename U, typename V>  
V Func2(T a)  
{  
    std::vector<U> vec;  
    return V{};  
}
```

Func2<int, int, double>(0);

# Overload and specialization

- But there are some special conditions...
  - For example, we want to use some code when “**T** is pointer”.
- For class, you can use *partial specialization*; but functions don't provide it.
- Reason & Solution: functions can use overloads!

```
template<typename T>
void Func(T* arg)
{
    std::println("Nobody.");
}
```

```
int a = 1;
Func(&a);
```

```
Here.
There 1!
Nobody.
```

- What if we use **nullptr** as parameter?
  - Oops!

```
Func(nullptr);
```

```
Here.
There 1!
Here.
```

# Overload resolution

- So in fact we have two candidates:
  - `template<typename T> void Func(T);` (named as F1)
  - `template<typename T> void Func(T*);` (named as F2)
- When we use `int*`, F2 is preferred over F1; when we use `nullptr`, F1 is preferred over F2.
- So there is an inner matching order; that's **overload resolution**.

# Overload resolution

- So which function is called is determined in these procedures:
  1. Names are looked up to find all possible functions to form an *overload set*.
    - ADL helps in this step but we don't cover it.
  2. Discard illegal functions by judging from their prototypes to form *viable function candidates*.
    - E.g. non-deducible templates;
    - E.g. `Func(int, double)` cannot be called by `Func(1)` due to wrong parameter number.
  3. Perform *overload resolution* to find the best candidate. If there isn't the best one, compile error.
  4. Check whether the candidate compiles.
    - E.g. if it's `=delete`, then compile error (yes, it's not excluded in step2);
    - E.g. there is `static_assert` in function body that's not satisfied.



# Overload resolution

- To put it simply, overload resolution just tries to find “the most precise one” determined by parameters. The order is:
  1. Perfect match or match with minimal adjustments (i.e. decay, add cv-qualifier).
  2. Match with promotion, e.g. `short->int`, `float->double`.
  3. Match with standard conversions (pre-defined ones), e.g. `int->short`.
    - For a conversion sequence (at most s-u-s), match the shorter.
  4. Match with user-defined conversions.

# Overload resolution

- If there are still more than one candidates, more rules will apply:
  1. More specialized ones are preferred, including considering value category;
  2. Non-template ones are preferred than template ones;
  3. For pointers, conversion order is: `Derived-to-base` > `void*` > `bool`.
  4. For `initializer_list`, when using universal initialization, it's preferred over other ones.
    - And that's why `std::vector<int>(5, 1) ≠ std::vector<int>{5, 1}`.
  5. Functors are preferred over surrogate functions (i.e. need conversion to become callable functor).
- ...
- It's very complicated and we don't cover it more; if you're interested, see [\[over.match\]](#).

# Overload resolution

- So now we know why:

So in fact we have two candidates:

- `template<typename T> void Func(T);` (named as F1)
- `template<typename T> void Func(T*);` (named as F2)

When we use `int*`, F2 is preferred over F1; when we use `nullptr`, there is a conversion to match `T*`, thus F1 is preferred over F2.

- `int*`: both exact match, but F2 is more specialized than F1.
- `nullptr`: F1 exact match (`T` is `nullptr_t`), while F2 doesn't.
- Exercise: what if we add `void Func(int*);` as F3?
  - `int*`: F3 > F2 > F1, since non-template is preferred when all exact match.
  - `nullptr`: F1 exact match, F3 needs conversion, so F1 > F3.

To use only template, you need to explicitly write `Func<...>()` (so non-template won't be a candidate due to syntax error).

# “More specialized”

- One more concern: what is “more specialized”?
- Formally, we say template A is more specialized than B if:
  - Hypothesize that there exist concrete types U1, U2, ... to substitute all template parameters in A, if it couldn't be deduced by B, then we say A isn't more specialized by B.
- “More specialized” is a partial ordering, so maybe neither template is more specialized than the other, which causes **ambiguous call**.
  - Notice that this will only be judged when calling, not when functions are defined.

# “More specialized”

- Example:

```
template<class T>
void f(T, T*);    // #1
template<class T>
void f(T, int*);  // #2
```

- It seems that #2 is more specialized than #1, but:
- #1 from hypothetical #2: for `f(U1, int*)`, #1 will:
  - For first parameter, deduce `T` as `U1`;
  - For second parameter, deduce `T` as `int`.
  - `U1 ≠ int`, thus deduction fails -> #2 is not more specialized than #1.
- #2 from hypothetical #1: for `f(U1, U1*)`, #2 will:
  - For first parameter, deduce `T` as `U1`;
  - For second parameter, fail to call -> #1 is not more specialized than #2.
- Thus, ambiguous call:

```
void m(int* p)
{
    f(0, p); //
```

Notice that `f(0.0, double*)` isn't ambiguous since #1 is exact match while #2 isn't.

# Template Basics

- Compile-time branch selection
  - Function overload resolution and specialization
  - Class specialization
  - Selection in code block

# Class template specialization

- Similarly, you can define full specialization for class:

```
template<typename T>
class A
{
    int m;
public:
    int GetM() const { return m; }
};
```

```
template<>
class A<int>
{
    double c;
public:
    int GetC() const { return c; }
};
```

- Typical example in standard library: `std::vector` and `std::vector<bool>`.
- Specialized class is a separate class, which can have completely different data member and member functions.

```
template<typename T> class B {};
```

  - You may just see it as a normal class. 

```
template<> class B<int> { public: void f(); };
```

No `template<>` when split member function definition, just like a normal class.

```
void B<int>::f() { }
```

# Class template specialization

- And, you can also define partial specialization!

```
template<typename T>
class A<T*>
{
    int k;
};
```

- Unlike functions, you cannot “overload” a class, like define non-template `class A;` `template<typename T, typename U> class A;` etc.
- Matching order is just choosing the most specialized one among all candidates.
  - E.g. `A<int*>` can match both `A<T*>` and `A<T>`, but the former is more specialized (formally, you can use a hypothetical type to deduce it).
  - `A<int>` can only match `A<T>`, so it's `A<T>`.




# Partial specialization

- Note1: partial specialization is not allowed to have default template parameter.
  - Reason: specialization only determines “whether a type matches it”; it doesn’t determine “what a type is”.

- Example: 

```
template<typename T, typename U = int>
class A { };
```

```
template<typename T = int>
class A<T, T> { };
```



- `A<int>` is determined to be `A<int, int>` by the primary template; then `A<int, int>` is judged to match the specialized one.

# Partial specialization

- Note2: NTTP partial specialization cannot depend on other template parameters.

- Forbidden cases: 

```
template<int N, int M>
class A {};
```

```
template<typename T>
class A<T::a, T::b> { };
```

```
template<class T, T t> struct C {}; // primary template
template<class T> struct C<T, 1>;  // error: type of the argument 1 is T,
                                   // which depends on the parameter T

template<int X, int (*array_ptr)[X]> class B {}; // primary template
int array[5];
template<int X> class B<X, &array> {}; // error: type of the argument &array is
                                       // int(*)[X], which depends on the parameter X
```

# Partial specialization

- Note3: variable template can also be specialized.
  - Partial specialization of variable template isn't regulated in the standard but all compilers implement it.
  - Particularly, the type of specialized variable can be different from the primary template.
- Note4: partial specialization is allowed to be defined inside the class.

- Example:

```
class A
{
    template<typename T> class B {};
    template<typename T> class B<T*> { int a; };
    // template<> class B<int> {}; // wrong
};

template<> class A::B<int> {}; // right
```

# Template Basics

- Compile-time branch selection
  - Function overload resolution and specialization
  - Class specialization
  - Selection in code block

# constexpr if

- Sometimes it's too troublesome to define all special cases by specialization...

- For example:

```
template<int N> struct M
{ static inline constexpr int value2 = N + 1; };
template<> struct M<0>
{ static inline constexpr int value = 100; };
```

```
template<int N>
int Func() { return M<N>::value2; }
```

```
template<> int Func<0>() { return M<0>::value; }
```

- It can't be better if we can code them together:

```
template<int N>
int Func() {
    if (N == 0)
        return M<0>::value;
    else
        return M<N>::value2;
}
```

# constexpr if

- However, it won't compile when instantiation.
  - E.g. `N == 0`, then `M<0>::value2` is invalid.
  - Though this branch is always not taken, but that's runtime thing!
- What we want: when some compile-time condition isn't met, the code segment isn't checked and generated at all.
  - That's constexpr if!
    - Since C++17.

```
if constexpr (N == 0)
    return M<0>::value;
else
    return M<N>::value2;
```

- Notice that `else if` should use `constexpr` too; only `else` can omit it.

# constexpr if

- Example: previous homework on variant
  - For `std::variant<int, double, std::string>`, convert to a string.

```
std::visit(  
    [](const auto &value) {  
        using InnerType = std::decay_t<decltype(value)>;  
        if constexpr (std::is_same_v<InnerType, std::string>)  
            return value;  
        else  
            return std::to_string(value);  
    }, currVar);
```

# constexpr if

- There exists a special condition – when it's evaluated at compile time, do something.
  - For example, we want to write a `constexpr` `sin x`.
    - At runtime, it's better to use `std::sin` directly, which may utilize hardware utility to accelerate.
    - At compile time, we may use Taylor expansion  $\sin x = \sum_{i=0}^{+\infty} (-1)^i x^i / (2i + 1)!$  to evaluate; it is slow but can at least be evaluated at compile time.
  - That's `constexpr if` since C++23.
    - No parentheses!
  - Negate: `if !constexpr {...}`.

```
float x = Sin(1); // Just same as std::sin(1);
constexpr float y = Sin(1); // Just same as Taylor expansion with 1024 terms.
```

For more compile-time math function implementations, see [C++: constexpr的数学库 - 知乎](#); C++23 will also make e.g. `std::sin` to be `constexpr` directly.

```
template<std::size_t N = 1024>
constexpr float Sin(float x)
{
    if constexpr
    {
        float sin = 0.0, temp = x;
        for (std::size_t i = 0; i < N; ++i)
        {
            sin += temp;
            temp *= -x * x / ((2 * i + 2) * (2 * i + 3));
        }
        return sin;
    }
    else {
        return std::sin(x);
    }
}
```



# is\_constant\_evaluated

- C++20 introduces `std::is_constant_evaluated`, which is same as:

```
constexpr bool is_constant_evaluated() {  
    if constexpr {  
        return true;  
    } else {  
        return false;  
    }  
}
```

```
template<std::size_t N = 1024>  
constexpr float Sin(float x)  
{  
    if (std::is_constant_evaluated())  
    {  
        float sin = 0.0, temp = x;  
    }  
}
```

- Notice that you cannot use `if constexpr (std::is_constant_evaluated())`, since the condition of `if constexpr` is always evaluated at compile time, which means **it's always true** here.
- Since it can only be used in runtime branch, it's less powerful than `if constexpr`.

```
constexpr int f(int i) { return i; }  
constexpr int g(int i) {  
    if constexpr {  
        return f(i) + 1; // ok: immediate function context  
    } else {  
        return 42;  
    }  
}
```

Cannot be substituted with `if (std::is_constant_evaluated());`

# Template Basics

Tricky Details

*Most tricky details in templates  
are caused by ambiguity!*

# Name lookup

- Names could be divided into two parts:
  - Dependent / Non-dependent name: if a name depends on template parameter, then it's dependent name.
  - Qualified / Non-qualified name: if a name is specified by `::`, `..`, `->`, then it's qualified. A fully qualified name is like `::a.b`.
    - Compilers need to automatically determine the identity of non-qualified name. ADL helps here (but we don't cover it).
    - Since C++ allows name reuse (in different blocks), so it will be looked up upwards.
- In templates, two-phase lookup is performed.
  - Non-dependent names are looked up when template is defined.
  - Dependent names are looked up when template is instantiated.

# this->

1. When the base class is dependent name, then `this->` is needed to access members of base class.

- E.g.

```
int a = 0;
template<typename T>
class A
{
public:
    int a;
};
```

```
template<typename T>
class B : public A<T>
{
public:
    int func() { return this->a; }
};
```

- Reason: `a` is non-dependent name, so it's looked up when the template is defined, not when instantiated. Thus `return a;` will then be returning the global one.
  - If the global `a` is not defined, `return a;` will cause compilation error.

# Two-phase lookup

- Why don't we lookup all names when templates are defined?
  - Reason: template may be specialized afterwards, which needs lookup in the second phase to know whether the identity exists.

- For example:

```
template<typename T>
class A {};

template<typename T>
class B : public A<T>
{
public:
    int func() { return a; }
};
```

Then define

```
template<>
class A<int>
{
public:
    int a;
};
```

You cannot  
know whether  
**a** exists when **B**  
is defined!

- Thus C++ requires to access data member by **this->** in this case; **this** is dependent so its name lookup is performed in the second stage.

# Two-phase lookup

- Why don't we lookup all names when templates are instantiated?
  - We can, but C++ hopes to expose syntax error as early as possible. Thus statements with only non-dependent names can be checked directly without instantiation.
- Notice that some compilers may check all things only at instantiation, e.g. msvc.
  - Since VS2017 (msvc 15.3), msvc adds `/permissive-` to enable two-phase lookup.

# typename

## 2. `A::B` could be either a type or a variable.

- For non-dependent names, it could be easily determined; but for dependent names, it's still ambiguous...

```
template <typename T> struct X {  
    using MemberType = int;  
};  
  
template<> struct X<float> {  
    static inline const int MemberType = 1;  
};
```

```
template<typename T> void Test()  
{  
    int b = 1;  
    {  
        T::MemberType * b;  
    }  
    return;  
}  
  
int main()  
{  
    Test<X<int>>(); int* b in Test.  
    Test<X<float>>(); 1 * b in Test.  
    return 0;  
}
```

# typename

- C++ chooses to always regard it as variable!
  - When it's actually type, you need to add a **typename**.
- For example:

```
template<typename T>
class A
{
public:
    void func(std::vector<T> &vec)
    {
        typename std::vector<T>::iterator it = vec.begin();
    }
};
```

Dependent name

```
template<typename T>
void func()
{
    typename T::type a{}
}
```

- Note: You cannot have an identifier that's possibly data or types; it **must** be determined before instantiation.



# typename

- To be specific, you **can** use **typename** when:
  - The type is a qualified name;
  - It's not after keywords **class/struct/union/enum**;
  - It's not Base class appears at inheritance specification and ctor.
    - i.e. **class A: typename B<T>** is wrong;
    - **A(int a) : typename B<T>{a} {}** is wrong.
- And you **must** use **typename** when:
  - Rules above;
  - The type is a dependent name;
  - It's not the current instantiation.
    - i.e. just the current type itself.

```
template<typename T>
class A
{
public:
    using Type = int;

    template<typename U>
    void func()
    {
        A<T>::Type p; // current instantiation, can neglect typename
        typename A<U>::Type p2; // unknown specialization
    }
};
```

# typename

- Exercise: which **typename** is wrong / correct but unnecessary / correct & must?

```
template<typename1 T>
struct S : typename2 X<T>::Base {
    S() : typename3 X<T>::Base(typename4 X<T>::Base(0)) {
    }
    typename5 X<T> f() {
        typename6 X<T>::C * p;    // declaration of pointer p
        X<T>::D * q;               // multiplication!
    }
    typename7 X<int>::C * s;

    using Type = T;
    using OtherType = typename8 S<T>::Type;
};
```

# typename

- Since C++20, many rules are relaxed.
  - To be short, when a dependent name appears where only type is possible, `typename` can be omitted.
- To be specific:
  - Return type of functions and lambda;
  - Aliasing declarations, e.g. `using Type = A<U>::Type;`
  - Target type of C++-style cast (e.g. `static_cast<A<U>::Type>(...)`);
  - Type of new expression (e.g. `new A<U>::Type{1}`);
  - Parameter type in *requires expression*, covered later.
  - Data member type, NTTP type;
  - Parameter types of member function & lambda;
  - Default value of template type parameter (e.g. `template <typename T, typename U = A<T>::Type>`)

# typename

- Notice that member function parameter and global function parameter differ here.

```
template<typename T>
TYPENAME T::value_type
foo(const T& cont, typename T::value_type arg) // typename optional
// typename required
```

```
template<typename T,
        auto ValT = typename T::value_type{}> // typename required
class MyClass {
    void print(TYPENAME T::iterator) const; // typename optional
};
```

```
template<typename T>
class MyClass {
public:
    template<typename U>
    void print(T arg, U::v arg2);
};
```

Okay too.

# typename

- A full example adopted from *C++20 – the Complete Guide*.

```
template<typename T,
        auto ValT = typename T::value_type{}> // typename required
class MyClass {
    TYPENAME T::value_type val; // typename optional
public:
    using iterator = TYPENAME T::iterator; // typename optional

    TYPENAME T::iterator begin() const; // typename optional
    TYPENAME T::iterator end() const; // typename optional
    void print(TYPENAME T::iterator) const; // typename optional
    template<typename T2 = TYPENAME T::value_type> // second typename optional
        void assign(T2);
};

template<typename T>
TYPENAME T::value_type // typename optional
foo(const T& cont, typename T::value_type arg) // typename required
{
    typedef typename T::value_type ValT2; // typename required
    using ValT1 = TYPENAME T::value_type; // typename optional
    typename T::value_type val; // typename required

    typename T::value_type other1(void); // typename required
    auto other2(void) -> TYPENAME T::value_type; // typename optional

    auto l1 = [] (TYPENAME T::value_type) { // typename optional
        };

    auto p = new TYPENAME T::value_type; // typename optional
    val = static_cast<TYPENAME T::value_type>(0); // typename optional
    ...
}
```

Rarely used.

# template

## 3. Template parameter specification is also ambiguous...

- E.g. `std::function<int()> f;`
  - It could be parsed as `(std::function < int()) > f`, where `std::function` and `f` are interpreted as variables.
- So how to parse is determined by the identity again (whether it's a template or not).
- C++ regulates that if the name is a template, `<` is always interpreted as the beginning of parameter specification; otherwise less-than operator.

```
namespace std {  
    int function = 1;  
}  
  
int f = 0;  
  
int main()  
{  
    std::function<int()> f;  
}
```

# template

- Again, dependent name cannot determine its identity...
  - So it will always be interpreted as less-than operator; when it's actually a template, you need to use `template` keyword explicitly.

```
template<std::size_t N>
void Test(std::bitset<N>& n)
{
    n.template to_string<char>();
}
```

- Here `n` is dependent and thus `to_string` cannot know whether it's a template. When `<` follows, it would be interpreted as less-than.

# template

- Another horrible example:

```
template<typename T>          template<typename T, int N>
class Shell {                 class Weird {
public:                         public:
    template<int N>            void case1 (
    class In {                 typename Shell<T>::template In<N>::template Deep<N>* p) {
    public:                     p->template Deep<N>::f(); // inhibit virtual call
        template<int M>        }
        class Deep {
        public:
            virtual void f();
    };
};
};
```

This **typename** can be omitted since C++20,  
as it's parameter of member function.

Credit: C++ Templates – The Complete Guide 2<sup>nd</sup> ed. by  
David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor



# template

- Note1: in template parameter specification, the first closing `>` will always be interpreted as the ending.
  - It's parsed as `M<a> b > m`.
  - You need additional parentheses to make it right: `M<(a > b)> m;`
- Note2: nested template (e.g. `vector<vector<int>>`) needs additional space (i.e. `int>` ) before C++11, to prevent ambiguity with operator `>>`.
  - Since C++11, it's also specially regulated.

```
template<bool S>
struct M {};

int main()
{
    constexpr int a = 1, b = 0;
    M<a > b> m;
}
```


# template

- Note3: C++ exists digraph and trigraph (e.g. <: equiv. to [), which makes e.g. S<::i> ambiguous.
  - Since C++11, it's specially regulated that <:: is never treated as <: + : (which makes it [:], but as a whole.
  - And since C++17, trigraph is removed.

# Nested Specialization\*

- Sometimes we may specialize a template in a template class...
  - It has many restrictions and is not always possible.
  - Very complicated and not commonly used, so we make it **optional**.
- 1. A fully specialized template cannot be defined in a not-fully specialized enclosing template.

```
template<typename T> class A {  
public:  
    template<typename U> class B {};  
};  
  
template<> template<> class A<int>::B<int> { void func(); };  
// template<typename T> template<> class A<T>::B<int> {};  
// template<typename T> template<> class A<T*>::B<int> {};
```



Primary template &  
partially specialized  
template aren't allowed.

# Nested Specialization\*

- But when a fully specialized enclosing class is explicitly defined, this `template<>` isn't needed anymore...

- As we've said, fully specialized class is just a normal class.

```
template<> class A<int> {  
public:  
    template<typename U> class C {};  
};
```

```
template<> class A<int>::C<int> { public: int a; };
```

- `template<>` is only needed when explicit specialization isn't specified.

- Similarly, to define `func` here: `template<> template<> class A<int>::B<int> { void func(); };`

- Since `B<int>` is already a normal class.

```
void A<int>::B<int>::func() { }
```

# Nested Specialization\*

2. Partial specialization can be normally defined regardless of the specialization status of the enclosing class.

```
template<typename T>
class C {
    template<typename U> class D {};

    template<typename U> class D<U**> {}; // Okay
};

template<typename T>
    template<typename U>
class C<T>::D<U*> { }; // Okay
```

- So sometimes to bypass the full-specialization restriction, you could add a dummy type parameter.

```
template<typename T>
class C {
    // The second param is never used.
    template<typename U, typename=void> class D {};
};

template<typename T>
    template<typename Dummy>
class C<T>::D<int, Dummy> { }; // Okay
```

# Template Basics

Concept

# Template Basics

- Concept
  - require clause and concept
  - Subsumption
  - Some exercises on concept

# Motivation

- Templates are good at reducing code replication.
  - What if we instantiate it by some unintended types?

- For example:

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

```
Max<const char*>("12", "34");
```

- E.g. We don't want users to compare pointers.
  - It will also report error when **T** isn't comparable, and users have to read the full source code / the error message to know all illegal operations...
- So why not add explicit constraints on types? Then users just need to read these constraints directly!



# Concept

- That's what concept for.
  - Before C++20, users can add constraint in an obscure way called "SFINAE", which is miserable for both users to read and programmers to write.
- Let's show an example for concept directly:

The parentheses are necessary when it's an expression instead of a pure value.

```
template<typename T>  
requires (!std::is_pointer_v<T>)  
T Max(const T& a, const T& b)  
{  
    ...  
    return a < b ? b : a;  
}
```

*requires clause*

```
1>main.cpp  
1>D:\Work\CS\C++\Test\Project2\main.cpp(16,5): error C2672: "Max" : 未找到匹配的重载函数  
1>   D:\Work\CS\C++\Test\Project2\main.cpp(9,3):  
1>   可能是 "T Max(const T &, const T &)"  
1>   D:\Work\CS\C++\Test\Project2\main.cpp(16,5):  
1>   未满足关联约束  
1>   D:\Work\CS\C++\Test\Project2\main.cpp(8,11):  
1>   未满足约束
```

# Concept

- Any template parameter can be constrained by requires clause:

```
template<typename T>  
requires (!std::is_pointer_v<T>)  
class A
```

```
template<typename T>  
requires (!std::is_pointer_v<T>)  
static T var{};
```

```
template<typename T>  
requires (!std::is_pointer_v<T>)  
using Type = T::value_type;
```

- You can also connect multiple constraints by **&&** and **||**.

```
template<typename T>  
requires (!std::is_pointer_v<T>) && (sizeof(T) >= 8)  
T Max(const T& a, const T& b)
```

- But we may hope to reuse these constraints, instead of writing it over and over again...

# Concept

- So we can declare a concept!

```
template<typename T>
concept MyNotPointer = !std::is_pointer_v<T> && sizeof(T) >= 8;

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

- That's still naïve; we need more constraints...
  - For example, what kind of expressions should be legal?
- Then you need *requires expression*.

# Concept

- For example:

```
template<typename T>
concept MyLessThan = requires(T x) { x < x; };

template<typename T>
requires MyNotPointer<T> && MyLessThan<T>
T Max(const T& a, const T& b)
```

- It checks whether  $x < x$  is legal, i.e. whether  $T$  can be compared by  $<$ . It never uses its runtime result.
- $T\ x$  is just a hypothetical parameter; it doesn't matter whether it has an accessible ctor. If parameters aren't needed, `requires { ... }` is enough.
- You can also combine a requires expression with logical operators.

```
template<typename T>
concept MyMaxConstraint = !std::is_pointer_v<T> && sizeof(T) >= 8
&& requires(T x) { x < x; };
```

# Concept

- Sometimes we just use a concept once, so we can insert it into the requires clause anonymously.

```
template<typename T>  
requires requires(T x) { x < x; }  
T Max(const T& a, const T& b)
```

*requires expression*  
*requires clause*

- Such constraint is same as `requires MyLessThan<T>`.
- Small exercise: can we combine two concepts like

```
template<typename T>  
concept MyMaxConstraint = requires(T x) {  
    !std::is_pointer_v<T>;  
    sizeof(T) >= 8;  
    x < x;  
};
```

# Concept

- Nope.
  - As we've said, it just checks whether the expression is legal, instead of whether it's true.
    - E.g. `sizeof(char) >= 8` results in `false`, but it's legal to write this expression. So it doesn't check anything.
- Similarly, this doesn't check anything either:

```
template<typename T>
concept MyMaxConstraint = requires(T x) {
    MyNotPointer<T>;
    x < x;
};
```

- Remember: it doesn't check something **valid but not true!**



# Requires expression

- So what we write before in requires expression is called *simple requirement*.
- Beyond that, there exist other three types of requirements.

- Type requirement: **typename** xxx;

- E.g.

```
template<typename T>
concept MyMaxConstraint = requires(T x) {
    x < x;
    typename T::value_type;
};
```

satisfy

```
class A
{
public:
    using value_type = int;
    auto operator<(const A&) const { return true; }
};
```

- Notice that if **typename** isn't added, then it's simple requirement that checks whether there exists a static data member.

```
template<typename T>
concept MyMaxConstraint = requires(T x) {
    x < x;
    T::value;
};
```

satisfy

```
class A
{
public:
    static int value;
    auto operator<(const A&) const { return true; }
};
```

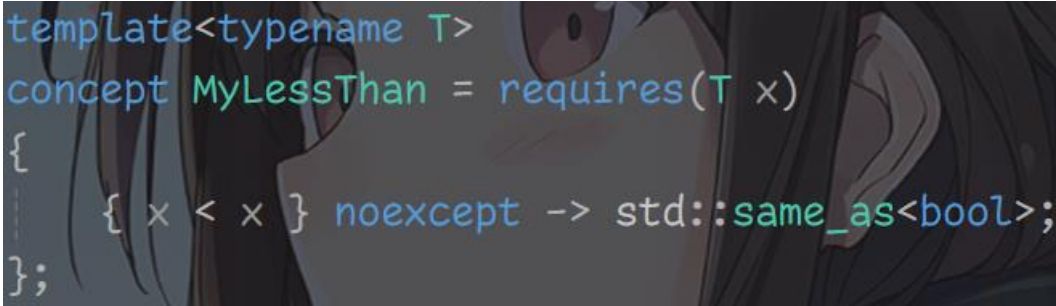
# Requires expression

- Compound requirement: check whether the expression is legal **and** whether the type of result satisfies some constraints.

```
{ expression } noexcept(optional) return-type-requirement(optional) ;
```

---

*return-type-requirement* - -> *type-constraint*



```
template<typename T>  
concept MyLessThan = requires(T x)  
{  
    { x < x } noexcept -> std::same_as<bool>;  
};
```

- The **noexcept** requires the operation to be **noexcept**.
- And **std::same\_as<bool>** is equivalent to **std::same\_as<decltype(x < x), bool>**; the result type will be fed as the first type parameter directly.



# Requires expression

- Nested requirement: `requires BooleanExpression;`
  - This checks both whether the expression is legal and whether it's true.

```
template<typename T>
concept MyMaxConstraint = requires(T x) {
    x < x;
    requires !std::is_pointer_v<T> && sizeof(T) >= 8;
};
```

# Concept

- Note1: for any type deduction & template parameter, you can add a single constraint by abbr.:

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

```
MyMaxConstraint auto a = A{};
```

```
MyMaxConstraint decltype(auto) Test()
{
    return 1.0;
}
```

When the deduced type doesn't match the constraint, compile error.

```
auto Max(const MyMaxConstraint auto& a, const MyMaxConstraint auto& b)
{
    return a < b ? b : a;
}
```

The deduced parameter will be the first parameter of the concept, same as compound requirements.

Notice that this is equivalent to

```
template<MyMaxConstraint T, MyMaxConstraint U> auto Max(const T& a, const U& b)
```

# Concept

- You can also combine it with requires clause:
  - Equivalent to combine with **&&**.

```
template<MyMaxConstraint T>  
requires AnotherConcept<T>  
T Max(const T& a, const T& b)
```

- Note2: concept can be used as boolean expression.
  - The result is whether the concept is fulfilled.

```
constexpr bool m = MyMaxConstraint<A>;
```

- Note3: NTTP can also use concept:
  - It cannot use abbr.

```
template<std::size_t N>  
concept LessThan8 = N <= 8;  
  
template<std::size_t N>  
requires LessThan8<N>  
struct MyArray { int a[N]; };
```

# Concept

- Note4: concept only adds syntactic constraints, not semantic constraints.
  - E.g. “.size() should be  $O(1)$ ”; this cannot be checked by compilers (halting problem).
  - Semantic constraints should be documented explicitly.
  - E.g. in the standard library, `std::invocable<F, Args...>` and `std::regular_invocable<F, Args...>`.
    - They have the same syntactic constraints, i.e. check whether `F` is callable with `Args`.
    - However, the latter has semantic constraints equality preserving; that is, it shouldn't change the status of function objects and parameters.
    - Concept cannot check whether a callable satisfies it, so they're equivalent from the compiler's view; the semantic difference is only documented.

# Concept

- Sometimes, the semantic constraint is added by manual tags.
  - The most typical one is iterator categories; you cannot distinguish contiguous iterators with random iterators syntactically.
  - That is, you cannot know whether continuous memory is occupied by iterators.
- So, standard library just adds a type alias `iterator_concept` since C++20.
  - For example, for iterator of vector, `using iterator_concept = std::contiguous_iterator_tag`.
  - Before C++20, it's `iterator_category = std::random_iterator_tag`; that's not accurate so it's recommended to:
    - Use `std::iter_value_t` instead of `iterator_traits<>::value_type`.
    - Use `std::iter_reference_t` instead of `iterator_traits<>::reference`.
    - Use `std::iter_difference_t` instead of `iterator_traits<>::difference_type`.

Notice that requirements of iterators are also slightly changed in C++20, e.g. input iterators is allowed to not provide copying.

# Concept

- Note5: There exist lots of concepts in the standard library.
  - As we've seen before, `std::same_as<T, U>`.
  - We're not going to talk about them in details; basically the functionality can be deduced from the name.

Credit: C++20 – The Complete Guide by Nicolai M. Josuttis. For more details, see Chapter 5.

You can borrow it from PKU library.

Concept	Constraint
<code>integral</code>	Integral type
<code>signed_integral</code>	Signed integral type
<code>unsigned_integral</code>	Unsigned integral type
<code>floating_point</code>	Floating-point type
<code>movable</code>	Supports move initialization/assignment and swaps
<code>copyable</code>	Supports move and copy initialization/assignment and swaps
<code>semiregular</code>	Supports default initialization, copies, moves, and swaps
<code>regular</code>	Supports default initialization, copies, moves, swaps, and equality comparisons
<code>same_as</code>	Same types
<code>convertible_to</code>	Type convertible to another type
<code>derived_from</code>	Type derived from another type
<code>constructible_from</code>	Type constructible from others types
<code>assignable_from</code>	Type assignable from another type
<code>swappable_with</code>	Type swappable with another type
<code>common_with</code>	Two types have a common type
<code>common_reference_with</code>	Two types have a common reference type
<code>equality_comparable</code>	Type supports checks for equality
<code>equality_comparable_with</code>	Can check two types for equality
<code>totally_ordered</code>	Types support a strict weak ordering
<code>totally_ordered_with</code>	Can check two types for strict weak ordering
<code>three_way_comparable</code>	Can apply all comparison operators (including the operator <code>&lt;=&gt;</code> )
<code>three_way_comparable_with</code>	Can compare two types with all comparison operators (including <code>&lt;=&gt;</code> )
<code>invocable</code>	Type is a callable for specified arguments
<code>regular_invocable</code>	Type is a callable for specified arguments (no modifications)
<code>predicate</code>	Type is a predicate (callable that returns a Boolean value)
<code>relation</code>	A callable type defines a relationship between two types
<code>equivalence_relation</code>	A callable type defines an equality relationship between two types
<code>strict_weak_order</code>	A callable type defines an ordering relationship between two types
<code>uniform_random_bit_generator</code>	A callable type can be used as a random number generator

Table 5.1. Basic concepts for types and objects

Concept	Constraint
<code>default_initializable</code>	Type is default initializable
<code>move_constructible</code>	Type supports move initializations
<code>copy_constructible</code>	Type supports copy initializations
<code>destructible</code>	Type is destructible
<code>swappable</code>	Type is swappable
<code>weakly_incrementable</code>	Type supports the increment operators
<code>incrementable</code>	Type supports equality-preserving increment operators

Table 5.2. Auxiliary concepts



Concept	Constraint
<code>range</code>	Type is a range
<code>output_range</code>	Type is a range to write to
<code>input_range</code>	Type is a range to read from
<code>forward_range</code>	Type is a range to read from multiple times
<code>bidirectional_range</code>	Type is a range to read forward and backward from
<code>random_access_range</code>	Type is a range that supports jumping around over elements
<code>contiguous_range</code>	Type is a range with elements in contiguous memory
<code>sized_range</code>	Type is a range with cheap size support
<code>common_range</code>	Type is a range with iterators and sentinels that have the same type
<code>borrowed_range</code>	Type is an lvalue or a <b>borrowed range</b>
<code>view</code>	Type is a <b>view</b>
<code>viewable_range</code>	Type is or can be converted to a view
<code>indirectly_writable</code>	Type can be used to write to where it refers
<code>indirectly_readable</code>	Type can be used to read from where it refers
<code>indirectly_movable</code>	Type refers to movable objects
<code>indirectly_movable_storable</code>	Type refers to movable objects with support for temporaries
<code>indirectly_copyable</code>	Type refers to copyable objects
<code>indirectly_copyable_storable</code>	Type refers to copyable objects with support for temporaries
<code>indirectly_swappable</code>	Type refers to swappable objects
<code>indirectly_comparable</code>	Type refers to comparable objects
<code>input_output_iterator</code>	Type is an iterator
<code>output_iterator</code>	Type is an output iterator
<code>input_iterator</code>	Type is (at least) an input iterator
<code>forward_iterator</code>	Type is (at least) a forward iterator
<code>bidirectional_iterator</code>	Type is (at least) a bidirectional iterator
<code>random_access_iterator</code>	Type is (at least) a random-access iterator
<code>contiguous_iterator</code>	Type is an iterator to elements in contiguous memory
<code>sentinel_for</code>	Type can be used as a sentinel for an iterator type
<code>sized_sentinel_for</code>	Type can be used as a sentinel for an iterator type with cheap computation of distances
<code>permutable</code>	Type is (at least) a forward iterator that can reorder elements
<code>mergeable</code>	Two types can be used to merge sorted elements into a third type
<code>sortable</code>	A type is sortable (according to a comparison and projection)
<code>indirectly_unary_invocable</code>	Operation can be called with the value type of an iterator
<code>indirectly_regular_unary_invocable</code>	Stateless operation can be called with the value type of an iterator
<code>indirect_unary_predicate</code>	Unary predicate can be called with the value type of an iterator
<code>indirect_binary_predicate</code>	Binary predicate can be called with the value types of two iterators
<code>indirect_equivalence_relation</code>	Predicate can be used to check two values of the passed iterator(s) for equality
<code>indirect_strict_weak_order</code>	Predicate can be used to order two values of the passed iterator(s)

Table 5.3. Concepts for ranges, iterators, and algorithms

# Template Basics

- Concept
  - require clause and concept
  - Subsumption
  - Some exercises on concept



# Concept subsumption

- We know that specialization just means “implement a different version for special cases”.
  - While concepts just describe “some special cases”.
  - So of course we can use concept to do specialization!
- For example: 

```
template<typename T> class SomeClass {};
```

 Is specialization of 

```
template<IsPointer T> class SomeClass<T> {};
```
- But specialization has matching order; the “most specialized” one will be the selected one among all templates.
  - So if we add multiple specializations by concept, we need to know their order.
  - For example:

```
template<IsPointer T> class SomeClass<T> {};
```

 More specialized than 

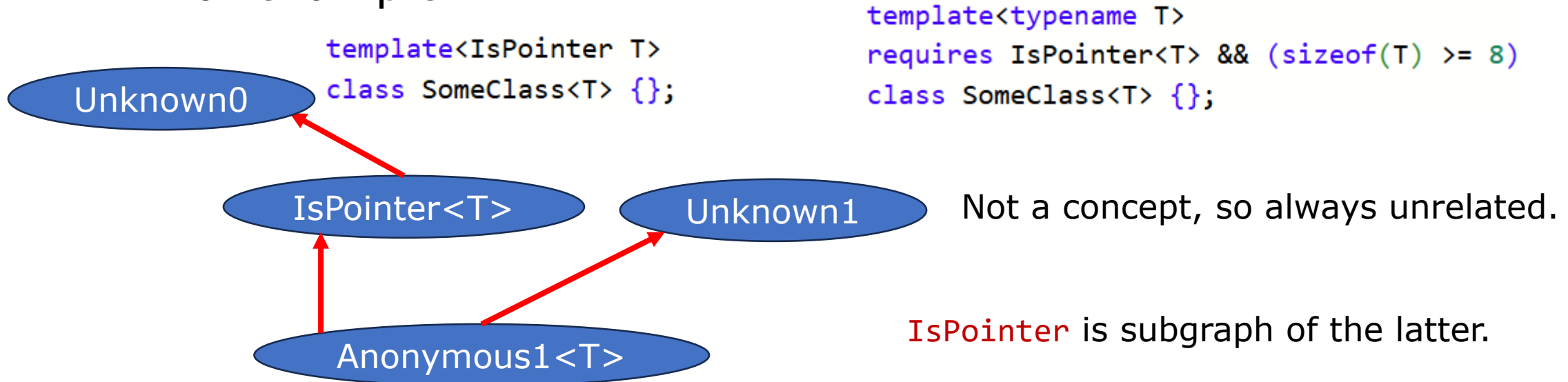
```
template<typename T> requires IsPointer<T> && (sizeof(T) >= 8) class SomeClass<T> {};
```

# Concept subsumption

- So how is the concept order determined?
  - We can notice that concept is just logical expressions.
    - Some atomic constraints, with  $\&$ ,  $\|$ ,  $!$ .
  - And logical expressions can imply (蕴含) !
    - E.g.  $a \& b \rightarrow a$ .
  - So generally, concept A is more specialized than / subsumes concept B  $\Leftrightarrow$  logical expressions A imply expressions B.
- But, implication is not easy to deduce.
  - From mathematical logic, we know  $a \rightarrow b$  can be transformed as  $a \wedge \neg b$ .
  - Assuming that for any type  $T$ ,  $a_i(T), b_j(T)$  may be true or false freely, then it will be SAT problem to judge whether it could be satisfied. That's NP-complete!

# Concept subsumption

- So to solve that, compilers don't do full logical judgement; instead, it only considers equivalence by concept name.
  - Non-concepts (including **!Concept**) will be always considered not related.
  - So we could build a concept subsumption graph;
  - And A subsumes B  $\Leftrightarrow$  graph B is subgraph of graph A.
- For example:



# Concept subsumption

- What if:

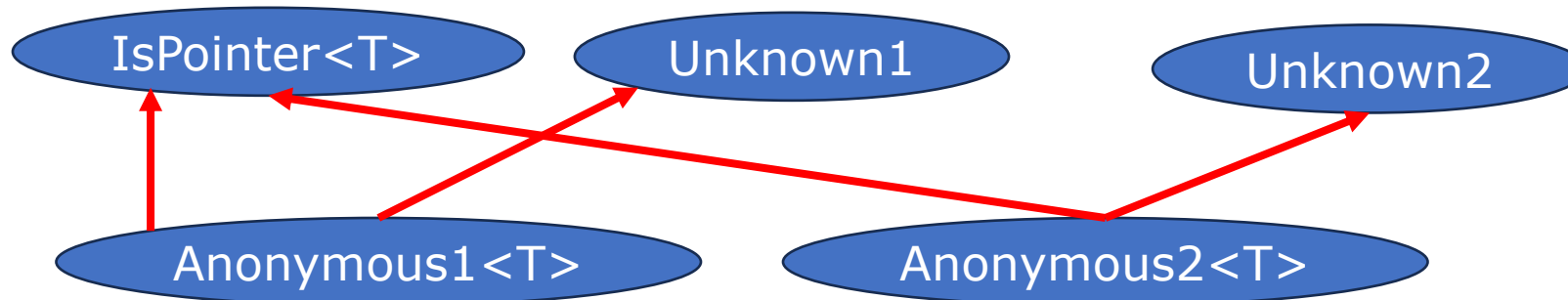
```
template<typename T>  
requires IsPointer<T> && (sizeof(T) >= 8)  
class SomeClass<T> {};
```

```
template<typename T>  
requires IsPointer<T> && (sizeof(T) >= 4)  
class SomeClass<T> {};
```

```
error: ambiguous template instantiation for 'class  
SomeClass<int*>' x86-64 gcc 14.2 #1
```

- We can deduce the first one is more specialized than the second one, but compilers don't know.
- Graph:

Unknown1 and Unknown2 are always seen as unrelated, so they don't subsume.



# Concept subsumption

- Part of subsumption graph of concepts in the standard library:

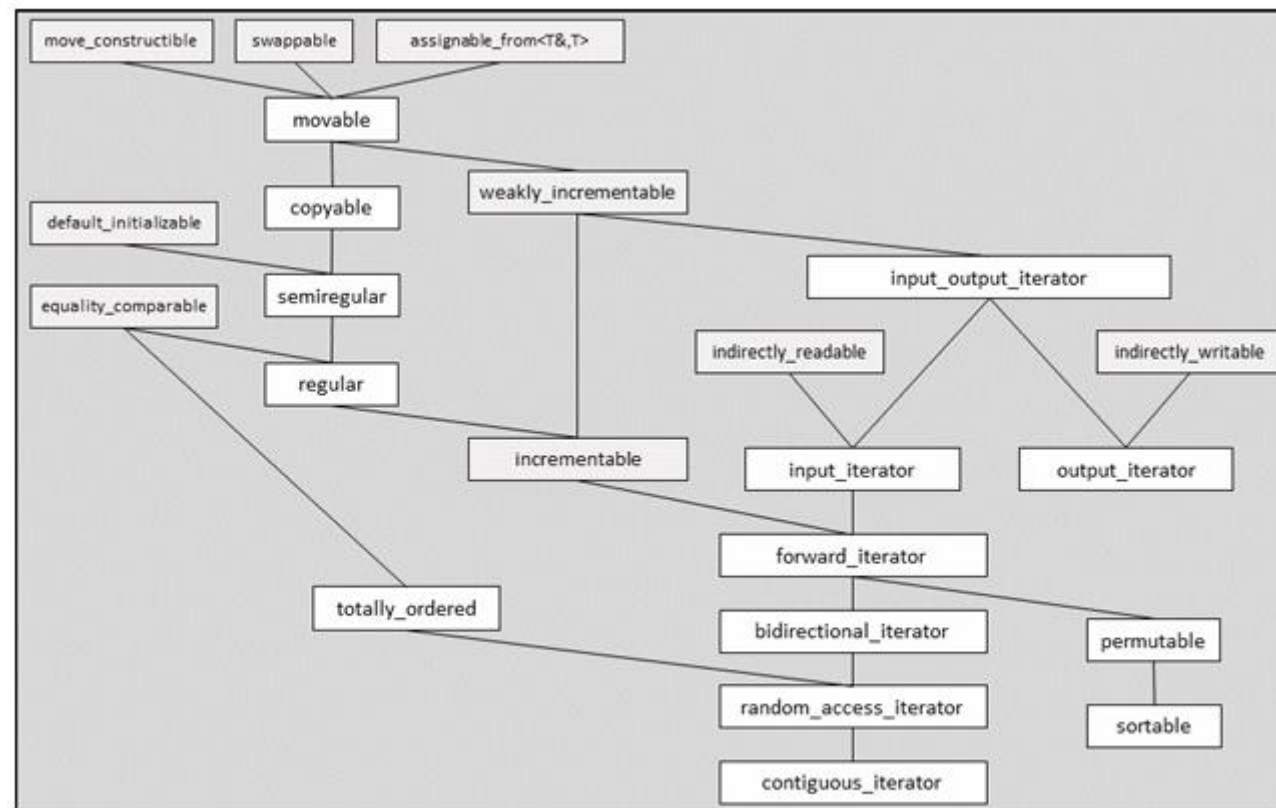


Figure 5.1. Subsumption graph of C++ standard concepts (extract)

Credit: C++20 – The Complete Guide by Nicolai M. Josuttis.

# Concept subsumption

- Exercise: implement communicative concept `SameAs` by type traits `std::is_same_v`.
  - Is it correct to use:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>;
```
  - No, since it's not communicative, `SameAs<T, U>` isn't considered equivalent as `SameAs<U, T>`.
  - Now is it correct? 

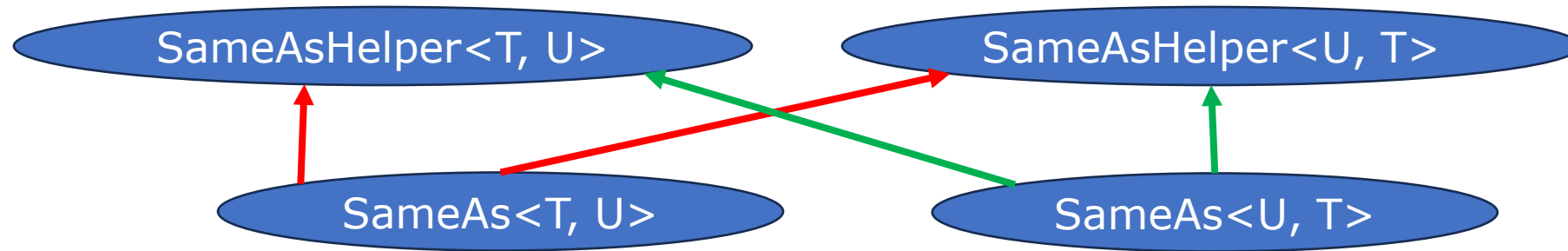
```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U> && std::is_same_v<U, T>;
```
  - No, since non-concepts will always be considered unrelated; the atomic constraint is just `SameAs<T, U>` and `SameAs<U, T>`, which is considered not equivalent.
  - Solution:

```
template<typename T, typename U>
concept SameAsHelper = std::is_same_v<T, U>;

template<typename T, typename U>
concept SameAs = SameAsHelper<T, U> && SameAsHelper<U, T>;
```

# Concept subsumption

- Now, the atomic constraint is `SameAsHelper`, so the graph is like:



- So `SameAs<T, U>` and `Same<U, T>` have the same graph, and they're considered equivalent!

# Concept subsumption

- Concept also matters in overloading resolution (“more specialized”).
  - When a concept subsumes another, then it’s preferred in resolution.

- For example: 

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

```
template<IsPointer T>
auto Max(const T& a, const T& b) { return *a < *b ? *b : *a; }
```

```
int main()
{
    int a = 1;
    Max(&a, &a);
}
```



# “More specialized”

- Notice that the second one cannot be changed as `Max(T a, T b)` since it breaks “more specialized”. If you want it, you need:

```
template<typename T>  
requires (!IsPointer<T>)  
T Max(const T& a, const T& b) { return a < b ? b : a; }
```

```
template<typename T>  
requires IsPointer<T>  
T Max(T a, T b) { }
```

- Reason: when both of two candidates are not more specialized than the other with rules before, more rules kick in:
  - If their template parameters or function parameters differ in length, ambiguous;
  - Otherwise, if template parameters are not equivalent or function parameters are not of same type, ambiguous;
  - Otherwise, if one template is more constrained (determined by concept subsumption) than the other, it’s regarded as more specialized.

Notice that [the actual rules](#) are slightly more complex since some function call can be reordered (e.g. `a == b` can be rewritten as `b == a` to make it compile since C++20), don’t cover it here.

# “More specialized”

- Now you can explain why we need additional constraint before!

- We know that `template<typename T> T Max(const T& a, const T& b)` and `template<typename T> T Max(T a, T b);` are not more specialized than the other when calling...
  - So they're first checked by parameter length, which is same.
  - Then their parameter forms are checked; though template parameters are equivalent, but functions parameter are not, so ambiguous.
  - Concepts aren't checked yet, so `IsPointer<T>` doesn't help to make the function “more specialized”.

# Template Basics

- Concept
  - require clause and concept
  - Subsumption
  - Some exercises on concept

# Concept

- By concepts, we could do more interesting things.
  - E.g. more special compile-time computation; more type traits.
- Exercise1: implement `std::is_nothrow_move_constructible` and `std::is_nothrow_move_assignable` by concept.

```
template<typename T>
concept NoexceptMoveConstructible = requires(T x) {
    { T{ std::move(x) } } noexcept;
};
```

```
template<typename T>
concept NoexceptMoveAssignable = requires(T x) {
    { x = std::move(x) } noexcept;
};
```

# Concept

- Exercise2: utilize concept & class specialization to write **IsPrime**.
  - Consider: Isn't it enough to implement it like this? Where is concept and specialization?

```
template<unsigned int N, unsigned int M>
struct DoIsPrime
{
    static constexpr inline bool value = M == 1 ? true : (N % M != 0 && DoIsPrime<N, M-1>::value);
};

template<unsigned int N>
struct IsPrime
{
    static constexpr inline bool value = N <= 1 ? false : DoIsPrime<N, N / 2>::value;
};
```

# Concept

- Let's try it: `bool p = IsPrime<4>::value;`

```
<source>:4:92: fatal error: template instantiation depth exceeds maximum of 900 (use '-ftemplate-depth=' to increase the maximum)
4 | static constexpr inline bool value = M == 1 ? true : (N % M != 0 && DoIsPrime<N, M-1>::value);
  |
```

- Reason: expressions need to be checked whether it's valid.
  - `IsPrime<4>` → `DoIsPrime<4,2>` → `DoIsPrime<4,1>`
  - It should stop now, but there is no short circuit for expression validity check, so it will continue to instantiate...
    - `DoIsPrime<4,1>` → `DoIsPrime<4,0>` → `DoIsPrime<4,(unsigned)-1>` → ...
- So the recursion never stops.
- To make it stop early, we need specialization; only the most specialized one will be instantiated.

# Concept

- Solution:

IsPrime<4> →  
DoIsPrime<4,2> →  
DoIsPrime<4,1>

And then only the most specialized one is instantiated, so only the second one is used. No infinite recursion!

```
template<unsigned int N, unsigned int M>
struct DoIsPrime
{
    static constexpr inline bool value = N % M != 0 && DoIsPrime<N, M-1>::value;
};
```

```
template<unsigned int N, unsigned int M>
requires (M == 1)
struct DoIsPrime<N, M>
{
    static constexpr inline bool value = true;
};
```

```
template<unsigned int N>
struct IsPrime
{
    static constexpr inline bool value = DoIsPrime<N, N / 2>::value;
};
```

```
template<unsigned int N>
requires (N <= 1)
struct IsPrime<N>
{
    static constexpr inline bool value = false;
};
```

# Move Semantics

Universal Reference and Perfect Forwarding



# Move semantics

- Universal Reference and Perfect Forwarding
  - Basics
  - Reference Collapsing Rule

# Motivation of Universal Reference

- Consider a function that doesn't care about the reference type, but just wants to transfer as is to a new function.
  - E.g. `std::vector::emplace_back(args...)`; it should just transfer `args` to the ctor and maintain its reference type.
  - Each parameter makes it need to overload for 3 or 4 times, so it needs  $4^N$  overloads, what??
    - And template is an important way to reduce replicate code.
  - So, can we use template to keep the type of reference?
  - That is universal reference!

Note: universal reference is called "forwarding reference" in C++ standard.  
Universal reference is named by Scott Meyers before entering the draft formally and widely used.

# Universal Reference

- Example:

```
template<typename T>
void func(T&& arg);
```

- Yes, when && cooperates with template type parameter directly, **it's never rvalue reference**, but universal reference.
- When accepting lvalue of type `MyType`, `T&&` is deduced as `MyType&`; `const` lvalue as `const MyType&`; rvalue as `MyType&&`.
  - For example:

```
template<typename T>
void f(T&& arg)
{
    std::println("{} {} {} {}",
        std::is_same_v<decltype(arg), std::string&>,
        std::is_same_v<decltype(arg), std::string&&>,
        std::is_same_v<decltype(arg), const std::string&>,
        std::is_same_v<decltype(arg), const std::string&&>);
}
```

```
std::string v;
const std::string v0;
f(v);
f(std::move(v));
f(std::string{});

f(v0);
f(std::move(v0));
```

```
true false false false
false true false false
false true false false
false false true false
false false false true
```

# Universal Reference

- Note: **T&&** is universal reference only when **T** is **exactly** a direct template type parameter.

- Let's see some misleading cases...

## 1. The type is not the parameter itself.

- Example1:

```
template<typename T>
void func(typename T::value_type&& arg)
```

- **value\_type** is a normal type under **T**, so this is just rvalue reference of it.
- Example2: try to specialize universal reference?

- This specialization is just rvalue reference of **std::string**, not universal reference of **std::string**!
- i.e. it only accepts rvalue.

```
template<typename T>
void func(T&& arg);
```

```
template<>
void func<std::string>(std::string&& arg);
```

# Universal Reference

2. The template type isn't provided by the function directly.

- Example: template class
- When instantiating, `T` will be assigned to a specific type, so `T&&` will be rvalue reference of `T`.

```
template<typename T>
class C
{
    void func(T&& arg);
};
```

3. The reference specifier isn't `&&`, e.g. `T&` and `T`.

4. It contains `const` and `volatile` (cv-qualifiers).

- `const T&&` only automatically deduces const rvalue reference of `T`.

# Universal Reference

- A special form of universal reference is **auto&&**.
  - For template: `void Func(auto&& arg);` `auto l = [](auto&& arg){};`
    - We've said **auto** is short for a new template type parameter.
  - For type deduction, it's still universal reference!

The range-based `for` statement

```
for ( init-statementopt for-range-declaration : for-range-initializer ) statement
```

is equivalent to

```
{  
    init-statementopt  
    auto &&range = for-range-initializer ;  
    auto begin = begin-expr ;  
    auto end = end-expr ;  
    for ( ; begin != end; ++begin ) {  
        for-range-declaration = * begin ;  
        statement  
    }  
}
```

# Universal Reference

- Exercise: write a template function that accepts `vector<T>&` and assigns every element to the default constructed one.

- Is it correct?

```
template<typename T>
void ClearToDefault(std::vector<T>& vec)
{
    for (auto& elem : vec)
    {
        elem = {};
    }
}
```

- There exists a special case: `std::vector<bool>...`
    - We've said iterating over it will get a **proxy** that represents a bit.
      - And that proxy is of value type, which cannot be referred by `auto&`.
  - Solution: use `auto&&/decltype(auto)`. `for (auto&& elem : vec) for (decltype(auto) elem : vec)`
    - Then it will either be deduced as rvalue reference or value.

# Perfect forwarding

```
template<typename T>
void func(T&& arg)
{
    func_(arg); ❌
}
```

- Now we can accept any reference, but we still need to transfer it as is.
  - However, `arg` is always lvalue, so `func_` cannot know its original type.
  - Of course, we can use `if constexpr` to do it manually.
  - But it can't be better if we can process uniformly...
- Target: make `arg` to have type `decltype(arg)` again.
  - That is what perfect forwarding for!

```
template<typename T>
void func(T&& arg)
{
    func_(std::forward<T>(arg));
}
```



# Perfect forwarding

- `std::forward<T>` is essentially `std::move` for rvalue reference, and no effects for lvalue reference.
  - Usually we use `std::forward` when you don't distinguish lvalue reference from rvalue reference (so you cannot tell whether it's a `std::move`).
- Note1: universal reference is always a reference; it cannot be a value type.
- Note2: `<T>` in `std::forward<T>` is necessary; it cannot be deduced automatically.
  - The reason will be covered later.

# Overload resolution on references

- How is overload resolution changed when universal reference is considered?
  - Universal reference is always second-best choice.
- A thorough summary of overload resolution (v is lvalue while c is const lvalue):

Call	f(X&)	f(const X&)	f(X&&)	f(const X&&)	template<typename T> f(T&&)
f(v)	1	3	no	no	2
f(c)	no	1	no	no	2
f(X{ })	no	4	1	3	2
f(move(v))	no	4	1	3	2
f(move(c))	no	3	no	1	2

Table 9.1. Rules for binding all references

# Overload resolution on references

- **ATTENTION**: fallback will be disabled if universal reference is used!
  - That is, non-const lvalue and const rvalue cannot fallback on **const&** in copy ctor.
- Example: implement **std::any**.
  - You can use any type for its ctor, so it's like:

```
class Any
{
public:
    template<typename T> Any(T&& arg); // ctor
    Any(const Any& another); // copy ctor
    Any(Any&& another); // move ctor
};
```

- Problem: non-const lvalue & const rvalue won't call copy & move ctor...

f(const X&)
3
1
4
4
3

# Overload resolution on references

- Solution: still by concept!

```
template<typename T>  
requires (!std::same_as<std::remove_cvref_t<T>, Any>)  
Any(T&& arg); // ctor
```

- But in some cases it's still problematic; e.g. if some class inherits **Any**, then calling copy/move ctor in inheritance (e.g. **Base{ std::move(another) }**) is still wrong...
- Anyway, **pay special attention** to universal reference ctor!
  - Similarly, for any function with universal reference parameter, it's usually not a good idea to overload.
    - That will disable many implicit conversions since universal reference is an exact match.
      - e.g. define a **std::string** overload, but **const char[]** doesn't use it.

# Move semantics

- Universal Reference and Perfect Forwarding
  - Basics
  - Reference Collapsing Rule

# Reference Collapsing

- We've known what **T&&** is when matching different expressions, but what is **T**?
- The standard regulates the deduction should be:
  - For rvalues, **T** is the value type (keep cv-qualifiers).
  - For lvalues, **T** is lvalue reference (keep cv-qualifiers).
    - Wait, what's **T& &&**?
- **Reference Collapsing Rule:** when multiple reference appears at the same time,
  - If all references are rvalue reference, the collapsed reference is rvalue reference.
  - If there is at least one lvalue reference, the collapsed reference is lvalue reference.

# Reference Collapsing

- Example: how is `std::move` implemented?
  - all references -> rvalue reference, keep cv-qualifier
    - `static_cast<RightReference>(exp);`

- So is it like this?

```
template<typename T>
T&& std::move(T&& obj)
{
    return static_cast<T&&>(obj);
};
```

- For lvalue, `T` is deduced as lvalue reference, and so `T&&` is still lvalue reference, that's wrong.
- So, we need to strip out the reference and add `&&`, then it's always rvalue reference.

# Reference Collapsing

- Solution:

- You can also use `decltype(auto)` in return type.

```
template<typename T>
std::remove_reference_t<T>&& std::move(T&& obj)
{
    return static_cast<std::remove_reference_t<T>&&>(obj);
};
```

- Exercise: implement `std::forward<T>`.

- First analyze why `<T>` is always needed?
  - What if we just call `std::forward(arg)`?
    - `arg` is lvalue, thus caller cannot know its original reference type...
- While `T` keeps that!
  - lvalue -> `T` is lvalue reference -> `std::forward<T>` should do nothing.
  - rvalue -> `T` is value type -> `std::forward<T>` should convert `arg` to rvalue ref.



# Reference Collapsing

- Answer: 

```
template<typename T>
decltype(auto) Forward(std::remove_reference_t<T>& arg)
{
    if constexpr (std::is_lvalue_reference_v<T>)
    {
        return arg;
    }
    return std::move(arg);
}
```

- And there is a simpler solution!

```
template<typename T>
decltype(auto) Forward(std::remove_reference_t<T>& arg)
{
    return static_cast<T&&>(arg);
}
```

- BTW, users basically just call `std::forward<T>(arg)`, but to prevent users from calling e.g. `std::forward<T>(std::move(arg))`, it provides an overload on rvalue reference, which is same as `std::move`.

# Reference Collapsing

- Note1: pay attention to deduction conflicts.
  - Example: Is it legal to `insert(v, s);`?

```
template<typename T>
void insert(std::vector<T>& vec, T&& elem)
{
    vec.push_back(std::forward<T>(elem));
}
std::vector<std::string> v;
std::string s = "test";
```

- No. From `v`, `T` is deduced as `std::string`; but from `s`, `T` is deduced as `std::string&`, which causes conflicts.
- Solution: use `std::vector<std::remove_reference_t<T>>&` or two template type parameters, e.g. `T1` and `T2`.

# Reference Collapsing

- Note2: how should `auto&& arg` be perfectly forwarded?
  - We've said `std::forward<T>` is same as `static_cast<T&&>...`

```
void func(auto&& arg)
{
    func_(std::forward<decltype(arg)>(arg));
}
```

- When `arg` is rvalue reference `T0&&`, `decltype(arg)` is `T0&&`.
  - So reference collapsing tells us that `static_cast<T&&> = static_cast<T0&& &&>`  
 $\Leftrightarrow$  `static_cast<T0&&>`, which converts it to rvalue reference (and thus normally move it).
- When `arg` is lvalue reference `T0&`, `decltype(arg)` is `T0&`.
  - `static_cast<T&&> = static_cast<T0& &&>`  $\Leftrightarrow$  `static_cast<T0&>`, which converts it to lvalue reference (and thus have no effect).

# Deducing this

- We can also use universal reference on deducing this.
  - For wrapper classes, e.g. `std::optional`, when we call `.value()`, it will return reference to the underlying value.
    - For rvalue `std::optional`, return rvalue reference;
    - For lvalue `std::optional`, return lvalue reference.
  - Before C++23, we need to write four overloads by reference qualifier.

```
constexpr T& value() &;  
constexpr const T& value() const &;  
  
constexpr T&& value() &&;  
constexpr const T&& value() const &&;
```

- By deducing this, it could be simplified as:

```
auto&& value(this auto&& self)  
{  
    if (!self) { throw std::bad_optional_access{}; }  
    return std::forward<decltype(self)>(self).value_;  
}
```

If `self` is rvalue reference, then `std::forward<T>(self)` is same as `std::move(self)`, which creates xvalue; and data member of xvalue is still xvalue, so it's right.

# Summary

- In this lecture, we've talked about:
  - For basic templates part:
    - constexpr function / variable, consteval function, constexpr variable;
    - Full specialization, partial specialization, overload resolution;
      - "More specialized"
    - constexpr if, consteval if;
    - Tricky details, including `this->`, `typename`, `template` and nested specialization.
    - Concepts
      - Requires clause;
      - Requires expression (simple, type, compound, nested);
      - Concept subsumption
      - Concept-based specialization & overloading

# Summary

- For move semantics part:
  - Universal reference, its usage and some misleading non-universal reference examples.
    - Pay attention to ctor with universal reference; concept is needed to add constraint. (Functions with universal reference should avoid function overloading or at least use with extreme care).
  - Perfect forwarding.
  - Overload resolution of references.
  - How `std::move` and `std::forward` are implemented.
  - Reference collapsing rules.
    - Deducing this

# Next lecture...

- We'll go into more advanced topics about template.
  - More about template parameter that's not a type;
  - More about type deduction;
  - Friend in class template;
  - Variadic templates and unpacking;
  - SFINAE
  - Type erasure
  - EBCO
  - CRTP
- That's a lot...