

模板进阶

Advanced Template

现代C++基础

Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Supplementary**

 - Template parameter that's not a type

 - Type Deduction

 - Friend in class template

 - Laziness

- **Variadic Template**

- **SFINAE**

- **Common Techniques**

 - CRTP

 - Type Erasure

Advanced Template

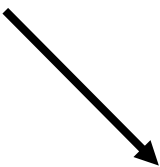
Supplementary Miscellaneous Knowledge

Supplementary Miscellaneous Knowledge

- Supplementary
 - Template parameter that's not a type
 - Type Deduction
 - Friend in class template
 - Laziness

Template parameter that's not a type

- First and foremost, very strangely...
 - Template parameter that's not a type \neq non-type template parameter (NTTP)
- There are three types of template parameters:
 - Type template parameter, i.e. `template<typename T>` or `template<class T>`;
 - Non-type template parameter, e.g. `template<int N>`;
 - Template template parameter.



Easier, so we talk about it first.

Template template parameter

- First observe how a template is defined:

```
template<typename T, int N>  
class A
```

- Similar to function parameter, name can be neglected if it's not actually used:

```
template<typename, int>  
class A
```

- And that's how a template template parameter should be defined:

```
template<template<typename, int> class SomeTemplate>  
class B
```

Here we can also use `typename` since C++17.

- And we can use template in this form to fill in the parameter!
- E.g. `B<A>`.

Template template parameter

- A more complex example:

```
template<template<typename T, T *> class SomeTemplate>
class B
```

- Require a template with a type parameter and an NTTP, and the NTTP depends on the type parameter.
- Notice that this T cannot be used outside:

```
template<template<typename T, T *> class SomeTemplate, typename U = T>
class B
{
    T a;
};
```

- This is very like scope of variable; outer scope cannot use names in inner scope.

Template template parameter

- Exercise: explain this template.

```
template<typename T, template<T> typename U>  
class B
```

- It accepts a type parameter **T**, and a template template parameter **U**...
 - Where **U** has a NTTP, whose type is **T**.
- For example:

```
template<typename T, template<T> typename U>  
class B  
{  
    U<T{}> a;  
};
```

```
template<int Size>  
class A  
{  
};
```

```
B<int, A> a;
```

Have **A<0> a;** as data member in **B<int, A>**.

Template template parameter

- A practical example: write a stack class!
 - Users may use any container to store elements as long as it supports push/pop/..., e.g. `std::vector/deque`, `std::inplace_vector` in C++26.

```
template<typename T, template<typename> class Container = std::deque>
class Stack
{
    Container<T> cont_;

public:
    void Push(const T &elem);
};

template<typename T, template<typename> class Container>
void Stack<T, Container>::Push(const T &elem)
{
    cont_.push_back(elem);
}
```

We notice that `std::stack` uses two type parameters, i.e. `template<typename T, typename Cont = std::deque<T>>`.

Template template parameter

- Example in standard library: `std::ranges::to`.

```
auto vec = r | std::ranges::to<std::vector>();
```

- Note 1: before C++17, template template parameter requires **exact match**.

- After C++17, default parameters are considered.

- E.g. `std::deque` actually has two parameters:

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

- You have to use:

```
template<typename T, template<typename, typename> class Container = std::deque>
class Stack
{
    Container<T, std::allocator<T>> cont_;
```

```
template<typename T, template<typename> class Container>
```

Cannot fill `std::deque`
before C++17.

NTTP

- Previously, our NTTPs are always integers...
 - E.g. `std::array<T, N>`, `IsPrime<N>`.
- While there are more possible types, including:
 - Enumerations (special kind of integer);
 - Pointers (and `nullptr_t`), pointer to member;
 - Lvalue reference;
 - Floating points;
 - Some “simple” classes;
 - `constexpr` Lambda.

} Since C++20
- These types are referred as “structural types”.

NTTP

- For example:

```
struct RGB
{
    int r, g, b;
};

template<decltype(&RGB::r) Channel>
void Test(RGB &color)
{
    color.*Channel = 0;
}
```

```
RGB color{ 1, 2, 3 };
Test<&RGB::g>(color);
std::cout << color.g;
```

```
void Func(int p)
{
    std::cout << p;
}

template<decltype(&Func) Pointer>
void Test2(int param)
{
    Func(param);
}
```

```
Test2<&Func>(1);
```

For a non-type *template-parameter* of reference or pointer type, or for each non-static data member of reference or pointer type in a non-type *template-parameter* of class type or subobject thereof, the reference or pointer value shall not refer or point to (respectively):

- a temporary object ([class.temporary]),
- a string literal object ([lex.string]),
- the result of a typeid expression ([expr.typeid]),
- a predefined __func__ variable ([dcl.fct.def.general]), or
- a subobject ([intro.object]) of one of the above.

There are actually many restrictions for passed pointer / reference, see [\[temp.arg.nontype\]](#).

NTTP

- For pointer and reference, passed argument has restrictions.
 1. Its address should be determined in compile time, so only those with static storage duration are allowed.
 2. Linkage requirement: C++98 external, C++11 includes internal, C++17 includes no linkage (i.e. static variable in function).

```
template<int *Ptr>
void Test3()
{
}
template<int &Ref>
void Test4()
{
}
static int m = 0;
```

```
Test3<&m>();
Test4<m>();
```

Whether two templates are the same instantiation depends on the address.

NTTP

```
MyClass<"hello"> x; //ERROR: string literal "hello" not allowed
```

- A special kind of “pointer” is string literals.
 - It’s not allowed to use string literals to initialize `const char*`.

• Solution:

```
extern char const s03[] = "hi"; // external linkage
char const s11[] = "hi"; // internal linkage
```

```
int main()
{
    Message<s03> m03; // OK (all versions)
    Message<s11> m11; // OK since C++11
    static char const s17[] = "hi"; // no linkage
    Message<s17> m17; // OK since C++17
}
```

- That’s inconvenient since we have to introduce additional named variable...
- And normally our understanding of equivalent template should be “have the same string content”, instead of same address.
 - Since C++20 you can use class-type NTTP!

NTTP

- For class NTTP, we first introduce literal types:

A literal type is any of the following:

Not class,
ignored.

- possibly cv-qualified `void` (so that `constexpr` functions can return void); (since C++14)

- scalar type;
- reference type;
- an array of literal type;
- possibly cv-qualified class type that has all of the following properties:
 - has a trivial(until C++20)constexpr(since C++20) destructor,
 - all of its non-static non-variant data members and base classes are of non-volatile literal types, and
 - is one of

- a lambda type, (since C++17)

- an aggregate union type that
 - has no variant members, or
 - has at least one variant member of non-volatile literal type,
- a non-union aggregate type, and each of its anonymous union members
 - has no variant members, or
 - has at least one variant member of non-volatile literal type,
- a type with at least one constexpr (possibly template) constructor that is not a copy or move constructor.

NTTP

For a non-type *template-parameter* of reference or pointer type, or for each non-static data member of reference or pointer type in a non-type *template-parameter* of class type or subobject thereof, the reference or pointer value shall not refer or point to (respectively):

- a temporary object ([class.temporary]),
- a string literal object ([lex.string]),
- the result of a `typeid` expression ([expr.typeid]),
- a predefined `__func__` variable ([dcl.fct.def.general]), or
- a subobject ([intro.object]) of one of the above.

- And class NTTP just requires:
 - Be a literal type;
 - All base classes and non-static data members are public, non-mutable and structural types (or array of structural types).
 - Particularly, for pointer and reference member, it has the same restrictions as NTTP.
- Finally, template class is also allowed to write at NTTP, and the concrete type will be deduced automatically.

```
template<std::array arr>
void f();

f<std::array<double, 8>{}>();
```

- So we can easily write a **FixedString** class...

NTPP

Cannot be `const char* str` as:

1. It's non-owning, cannot do complex operations like concatenation;
2. It's forbidden to point to string literal as the last page shows.

```
template<std::size_t N>
struct FixedString
{
    char str[N];
    constexpr FixedString(const char (&input)[N])
    {
        for (std::size_t i = 0; i < N; i++)
            str[i] = input[i];
    }
};

template<FixedString InputStr>
void Test() {}

int main()
{
    Test<"123">{};
}
```

We'll provide an exercise in homework to write a more reasonable class.

NTTP

- Floating points can be seen as instantiated by “underlying binary representation”.
 - E.g. if the platform uses IEEE 754, `Test<1.0f>` can be hypothetically viewed as `Test<3f800000>`.
 - But all `NaN` will be seen as equivalent.
- And we notice that floating point rounding error is still important.

```
template<double Val>
class MyClass {
};

int main()
{
    std::cout << std::boolalpha;
    std::cout << std::is_same_v<MyClass<42.0>, MyClass<17.7>>           // always false
              << '\n';
    std::cout << std::is_same_v<MyClass<42.0>, MyClass<126.0 / 3>>      // true or false
              << '\n';
    std::cout << std::is_same_v<MyClass<42.7>, MyClass<128.1/ 3>>      // true or false
              << "\n\n";

    std::cout << std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
                          MyClass<0.3 + 0.1 + 0.00001>>           // true or false
              << '\n';

    std::cout << std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
                          MyClass<0.00001 + 0.3 + 0.1>>           // true or false
              << "\n\n";

    constexpr double NaN = std::numeric_limits<double>::quiet_NaN();
    std::cout << std::is_same_v<MyClass<NaN>, MyClass<NaN>>           // always true
              << '\n';
}
```

NTTP

- Since C++17, it's also allowed to accept NTTP of any type.

```
template<auto Param> // or auto& / decltype(auto)
class A
```

```
A<1> a;
A<nullptr> b;
```

- Before that, you need to add an additional type parameter for assistance.
 - But you have to specify that type manually, very inconvenient...

```
template<typename T, T Param>
class A
```

```
A<int, 1> a;
A<std::nullptr_t, nullptr> b;
```

- It's then easy to accept lambda as NTTP:

NTTP

`std::invocable<F, Args...>` requires `F` to be callable with arguments `Args...`; here it means `Callable` should use 0 parameter.

```
template<std::invocable auto Callable>
class A
{
public:
    constexpr decltype(auto) operator()() { return Callable(); }
};

int main()
{
    A<[]() { return 0; }> a;
    A<[]() { std::cout << "Hello, world"; }> b;
}
```

Call `Callable`.

Lambda should be `constexpr` (thus no capture is allowed), but this is implicitly added since C++17 as long as all operations are allowed in `constexpr` function.

- Notice that each lambda has its unique type even if they have same closure body, so the instantiation is unique theoretically.

Supplementary Miscellaneous Knowledge

- Supplementary
 - Template parameter that's not a type
 - Type Deduction
 - Friend in class template
 - Laziness

Type Deduction

- We don't need to fill all template parameters due to type deduction...
- Each function parameter will deduce template parameter **independently**.
 - Assuming template parameter is P and passed argument type is A...
- For each deduction, there are generally two rules:
 - For non-reference parameter P, **decayed** A is deduced.
 - P will ignore its top-level cv-qualifier.
 - For reference parameter (e.g. P&), the original A is deduced.
 - ***auto** has basically the same rule.
- And finally, if deduction leads to conflict types, fail to match.

Type Deduction

- For example:

Top-level cv is ignored, equiv. to use 1 to deduce T.

```
template<typename T>
void Func3(const T)
{
}
```

```
Func3(1);
```

```
template<typename T>
void Func(T a, T b);

template<typename T>
void Func2(T &a, T &b);

int main()
{
    const int a = 1, b = 1;
    Func(a, b); // a & b: T -> int
    Func2(a, b); // a & b: T -> const int

    int c[8], d[10];
    Func(c, d); // a & b: T -> int*
    Func2(c, d); // a: T -> int[8]; b: int[10]
}
```

Not decayed,
so conflict.

Type Deduction

- In most cases, **conversion is forbidden** in deduction.

```
template<typename T>
void Func(T a, T b);

class A
{
public:
    operator int() { return 0; }
};

int main()
{
    Func(1, A{});
}
```

Deduction failure due to conflict, rather than $T = \text{int} + A \rightarrow \text{int}$ conversion.

Due to separate deduction, **this is inevitable.**

Deduction failure instead of $T = \text{int} + B<\text{int}> \rightarrow A<\text{int}>$ conversion.

This may be evitable?

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

template<typename T>
class B
{
public:
    operator A<T>() { return {}; }
};

template<typename T>
void Func(A<T> a);

int main()
{
    Func(B<int>());
}
```

Type Deduction

- There are three special cases to allow conversion:
 1. If P is reference, the deduced A can be more cv-qualified than A.
 2. If P is pointer, the deduced pointer can have qualification conversion.

```
template<typename T> void Func(const T &);  
template<typename T> void Func2(const T *);  
  
int main()  
{  
    int a = 1;  
    Func(a); // conversion: int -> const int, then T = int  
    Func2(&a); // conversion: int* -> const int*, then T = int  
}
```

1 and 2 are different,
since more cv-qualified
int* is **int* const**, not
const int*.

3. If P is base class (pointer), A is derived class (pointer), derived-to-base conversion is allowed.

```
template<typename T> class A {};  
  
template<typename T> class B : public A<T> {};  
  
template<typename T> void Func(A<T> a);  
  
int main()  
{  
    Func(B<int>{}); // conversion: B<int> -> A<int>  
}
```

Type Deduction

- Sometimes deduction needs further match...

```
template<typename T, typename U>
void Func(T (*funcPtr)(U &))
{
}

void Test(int &);

int main()
{
    Func(Test);
}
```

If we regard it as a whole, it's not reference type (i.e. V `funcPtr`).

Then we know $V = \text{void}(*)(\text{int}\&)$, the decayed function type.

And we match $T(*)(U\&)$, getting $T = \text{void}$ and $U = \text{int}$.

- Such match is exact, no conversion is allowed.

```
void Test(int);
```



Type Deduction

- Lots of pattern can be used to match, not described in detail.
 - Sometimes recursive match is needed.

- `cV(optional) T;`
- `T*;`
- `T&;`
- `T&&;` (since C++11)
- `T(optional) [I(optional)];`
- `T(optional) (U(optional));` (until C++17)
- `T(optional) (U(optional)) noexcept (I(optional));` (since C++17)
- `T(optional) U(optional)::*;`
- `TT(optional)<T>;`
- `TT(optional)<I>;`
- `TT(optional)<TU>;`
- `TT(optional)<>;`

In the above forms,

- `T(optional)` or `U(optional)` represents a type or *parameter-type-list* that either satisfies these rules recursively, is a non-deduced context in P or A, or is the same non-dependent type in P and A.
- `TT(optional)` or `TU(optional)` represents either a class template or a template template parameter.
- `I(optional)` represents an expression that either is an I, is value-dependent in P or A, or has the same constant value in P and A.

- `noexcept (I(optional))` represents an **exception specification** in which the possibly-implicit `noexcept` specifier's operand satisfies the rules for an `I(optional)` above. (since C++17)

```
template<typename T>
void f1(T*);
```

```
template<typename E, int N>
void f2(E(&)[N]);
```

```
template<typename T1, typename T2, typename T3>
void f3(T1 (T2::*)(T3*));
```

```
class S {
public:
    void f(double*);
};
```

```
void g (int*** ppp)
{
    bool b[42];
    f1(ppp); // deduces T to be int**
    f2(b); // deduces E to be bool and N to be 42
    f3(&S::f); // deduces T1 = void, T2 = S, and T3 = double
}
```

Type Deduction

- There also exists non-deduced context ([\[temp.deduct.type\]](#)), i.e. where parameters cannot be deduced.
 - `std::initializer_list` cannot be deduced, as we stated before.
 - `auto` in definition can deduce it, e.g. `auto a = {1,2,3}`.
 - The first/major array bound, if P is not reference / pointer type.
 - Reference deduction won't decay; pointer has already decayed the first bound.

```
template<int i>
void f1(int a[10][i]);

template<int i>
void f2(int a[i][20]);    // P = int[i][20], array type

template<int i>
void f3(int (&a)[i][20]); // P = int(&)[i][20], reference to array

void g()
{
    int a[10][20];
    f1(a);    // OK: deduced i = 20
    f1<20>(a); // OK
    f2(a);    // error: i is non-deduced context
    f2<10>(a); // OK
    f3(a);    // OK: deduced i = 10
    f3<10>(a); // OK
}
```

Type Deduction

- NTTP cannot be deduced from expression:
 - Cannot deduce $N = 5$ here.
- Qualified type names cannot be used for deduction:
 - Cannot deduce $T = \text{int}$ here.
- Default parameter cannot be used for deduction:
 - Though `Func()` should be equiv. to `Func(1)`, T cannot be deduced as `int`.
- `decltype` type.
 - Cannot deduce $T = \text{int}$ here.

```
template<std::size_t N>
void f(std::array<int, 2 * N> a);

std::array<int, 10> a;
f(a); // P = std::array<int, 2 * N
```

```
template<typename T>
void Func(T = 1);

int main()
{
    Func();
}
```

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

template<typename T>
void Func(typename A<T>::Type);

int main()
{
    Func(1);
}
```

```
template<typename T> T value = {};

template<typename T> void Func(decltype(value<T>));

int main()
{
    Func(1);
}
```

Type Deduction

- There are some other deduction contexts, not cover it in detail.
 - Address of an overload set, where return type will also be used for deduction.
 - If there are multiple best match after overload resolution, ambiguous.
 - Conversion function template.
 - ...
- Since C++17, class template argument deduction (CTAD) is introduced.
 - It deduces class template argument **from constructor**, so rules are similar to deducing from function call.

CTAD

- Just transform template parameter of class to ctors!

```
template<typename T>
struct UniquePtr
{
    UniquePtr(T *t);
};
```



```
struct X
{
    template<typename T>
    X(T *t);
};
```

- And then use rules before...
- Wait, where is copy/move ctor?
 - As deduction will strip reference, they're combined together as a single hypothetical function:

```
template<typename T> X(UniquePtr<T>);
```

- Besides, C++ allows **user-defined deduction guides** in CTAD.

CTAD

- For example:

```
std::list l{ 1, 2, 3 };  
std::vector v{ l.begin(), l.end() };
```

 - CTAD of `std::list` is quite simple; it comes from `list(std::initializer_list<T>)`.
 - But vector from iterator is `template<typename Iter> vector(Iter, Iter)`.
 - There is no T at all...How is class template parameter deduced?
- User-defined deduction guide is very similar to ctor, just add trailing return type to specify “what should be deduced”.

```
template<class InputIt>  
vector(InputIt, InputIt)  
    -> vector<typename std::iterator_traits<InputIt>::value_type>;
```

Use `std::iter_value_t<InputIt>` since C++20.

explicit(optional) *template-name* (*parameter-list*) *requires-clause*(optional) -> *simple-template-id* ; (1)

template <*template-parameter-list*> *requires-clause*(optional)
explicit(optional) *template-name* (*parameter-list*) *requires-clause*(optional) -> *simple-template-id* ; (2)

CTAD

- **explicit**: if written, disable copy initialization deduction.

```
template<class InputIt>
explicit A(InputIt, InputIt)
    -> A<typename std::iterator_traits<InputIt>::value_type>;
```

```
A v{ l.begin(), l.end() };
// A v = { l.begin(), l.end() }; // deduction fails
```

- requires clause: specify concept constraints for when deduction happens.

```
template<class InputIt>
requires std::random_access_iterator<InputIt>
explicit A(InputIt, InputIt)
    -> A<typename std::iterator_traits<InputIt>::value_type>;
```

```
// A v{ l.begin(), l.end() }; // deduction fails
```

CTAD

- Note 1: CTAD cannot do partial deduction. All or nothing!

```
template<typename T1, typename T2> class A{ public: A(T1, T2); };  
A a{1,2}; // Okay;  
A<int> a{1,2}; // Nope, though 2 can be used to deduce int.
```

- Note 2: CTAD doesn't have to be same as some ctor; deduction and overload resolution are separate.

```
template<class InputIt>  
A(InputIt) -> A<typename std::iterator_traits<InputIt>::value_type>;
```

```
A v{ l.begin() }; // deduction succeeds, but A<int> doesn't have  
// proper ctor (no viable candidate).
```

CTAD

- Note 3: unlike method definition, the class name **cannot** use qualified name in deduction guide.
 - Assuming **A** is in namespace **Test**.

```
template<class InputIt>
Test::A(InputIt)->Test::A<typename std::iterator_traits<InputIt>::value_type>;
```

- So, usual practice is to write deduction guide immediately after class definition.

```
namespace Test
{
template<typename T>
class A
{
public:
    template<class InputIt>
    A(InputIt a, InputIt b)
    {
    }
};

template<class InputIt>
A(InputIt) -> A<typename std::iterator_traits<InputIt>::value_type>;

} // namespace Test
```

CTAD

- Note 4: in class context, injected template name is preferred over CTAD.
 - Injected class name means that `A` refers to current instantiation, no need to write `A<T>` explicitly.
 - But injected class name is disabled when using qualified name, then CTAD happens.

```
A &operator=(const A &another)
{
    ::A a{ another }; // CTAD
    swap(a);
    return *this;
}
```

```
template<typename T>
class A
{
public:
    A &operator=(const A &another)
    {
        A a{ another };
        swap(a);
        return *this;
    }
}
```

- In this example CTAD or not doesn't differ...

CTAD

- A better example:

```
template<class T>
struct X
{
    X(T) {}

    template<class Iter>
    X(Iter b, Iter e) {}

    template<class Iter>
    auto foo(Iter b, Iter e)
    {
        return X(b, e); // no deduction: X is the current X<T>
    }

    template<class Iter>
    auto bar(Iter b, Iter e)
    {
        return X<typename Iter::value_type>(b, e); // must specify what we want
    }

    auto baz()
    {
        return ::X(0); // not the injected-class-name; deduced to be X<int>
    }
};
```

Otherwise it's $X<T>(b, e)$, but we actually want $X<U>(b, e)$.

CTAD

- Note 5: in deduction guide, **T&&** is still universal reference, not rvalue reference.
 - Here due to reference collapsing, ctor is then transformed to **A(int&)**.
 - *So we say **A(T&&)** isn't universal reference because it cannot deduce lvalue reference without deduction guide.
- To prevent that astonishment, normally deduction guides use value type.
 - Here **A<int>** rather than **A<int&>** is deduced, compilation error as expected..




```
template<typename T>
class A
{
public:
    A(T &&) {} // rvalue reference
};

template<typename T>
A(T &&) -> A<T>; // universal reference

int main()
{
    int a = 1;
    A b{ a }; // A<int&> is deduced.
```

```
template<typename T> A(T) -> A<T>;
no instance of constructor "A<T>::A [with T=int]" matches the argument list
int ma
{
    in View Problem (Alt+F8) Quick Fix... (Ctrl+.)
    A b{ a }; // A<int> is deduced.
```

CTAD

class template argument deduction for alias templates (FTM)*	P1814R0 	10	19	19.27*
class template argument deduction for aggregates (FTM)*	P1816R0  P2082R1 	10* 11*	17	19.27*

- Note 6: only the class itself can be deduced; adding reference / pointer would lead to failure.
 - In most cases just use `auto`.
- Note 7: implicit deduction guide for aggregate is added since C++20.
 - Example: user-defined guide must be added in C++17.

```
A b{ a }; //
A &&c{ a };
A &d = b;
A *p = &b;
```

- Note 8: C++20 also introduces deduction for alias template.

```
template<typename T> struct A { T val; };
template<typename T> A(T) -> A<T>;
// 如果没有deduction guide, 则A a{1};不正确
```

- To put it simply, for every deduction guide, use alias to deduce parameters by trailing type as much as possible. Remove deduced ones, add alias constraints and use only non-deducible ones to form new guides.

Alias Template Deduction*

- This is very complex so **optional**.

- We use A as example here; #1 and #2 form two deduction guides.
- From #1, use `C<V*, V*>` to deduce `C<T, U>`, forming

```
template<class T, class U, class V>  
C(V *, V *) -> C<V *, V *>;
```

- From #2, use `C<V*, V*>` to deduce `C<T, std::identity_t<U>>`, forming

```
template<class T, class U, class V>  
C(V *, U) -> C<V *, std::identity_t<U>>;
```

- U is not deducible, as it's an alias of qualified type (non-deduced context).
- Strip useless parameters and add constraints (here A doesn't have constraints; if it's B, then add `std::integral` to W).

```
template <class T, class U> struct C {  
    C(T, U);  
}; // #1  
  
template<class T, class U>  
    C(T, U) -> C<T, std::type_identity_t<U>>; // #2  
  
template<class V> using A = C<V *, V *>;  
template<std::integral W> using B = A<W>;
```

Alias Template Deduction*

- Now we get two guides for alias:

```
template<class V>
C(V *, V *) -> C<V *, V *>;

template<class U, class V>
C(V *, U) -> C<V *, std::type_identity_t<U>>;
```

- Finally, when it's deduced, we need to check again whether it satisfies alias.

```
int i{};
double d{};
A a1(&i, &i); // deduces A<int> from #1
A a2(i, i); // error: cannot deduce V * from i both #1 and #2 have V*
A a3(&i, &d); // error: #1: cannot deduce (V*, V*) from (int *, double *)
// #2: cannot deduce A<V> from C<int *, double *> #2 deduces C<int*, double*>,
// but it doesn't match A<V>.
```

- See more practice in [C++ standard](#) and [cppreference](#).
template<class V> using A = C<V *, V *>;
 - gcc has a bug on this feature, see [c++ - How to write deduction guidelines for aliases of aggregate templates? - Stack Overflow](#) (and you can also do practice by this analysis).

CTAD

- Note 9: since C++23, CTAD can happen through inherited ctor.
 - For template parameters of derived class, if they're part of parameters of base class, they'll be deduced by guides of base class.
 - Let's see examples in the standard directly:

```
template <typename T> struct B {
    B(T);
};
template <typename T> struct C : public B<T> {
    using B<T>::B;
};
template <typename T> struct D : public B<T> {};

C c(42);           // OK, deduces C<int>
D d(42);           // error: deduction failed, no inherited deduction guides
B(int) -> B<char>;
C c2(42);          // OK, deduces C<char>
```

Inherited ctor.

```
template <typename T> struct E : public B<int> {
    using B<int>::B;
};
E e(42);           // error: deduction failed, arguments of E cannot be deduced from introduced guides

template <typename T, typename U, typename V> struct F {
    F(T, U, V);
};
template <typename T, typename U> struct G : F<U, T, int> {
    using G::F::F;
};

G g(true, 'a', 1); // OK, deduces G<char, bool>
```

T isn't part of base class.

T and U are part of base class.

Supplementary Miscellaneous Knowledge

- Supplementary
 - Template parameter that's not a type
 - Type Deduction
 - Friend in class template
 - Laziness

Friend in class template

- To define a friend method for a class:
 - We can also define friend function in the class, and ADL will help us to find it.
 - Like defining `operator<<` as friend.

```
class A
{
    friend void Func(A &a);
};

void Func(A &) {}
```

- For a class template, splitting definition is harder...
 - Is it correct?

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

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

No!

Friend of `A<T>` is a normal function `func(A<T>)`, not a template function.

*But unlike normal functions, their definition is instantiated only when used.

Friend in class template

- If we call & compile:

```
A<int> a;  
Func(a);
```

```
error: main.cpp.obj : error LNK2019: 无法解析的外部符号 "void __cdecl Func(class A<int> &)" (?Func@@YAXAEAV? $A@H@@@Z), 函数 main 中引用了该符号  
build\windows\x64\release\main.exe : fatal error LNK1120: 1 个无法解析的外部命令
```

- For `A<int>`, the friend method is `Func(A<int>)`, so we may try:
 - Succeed to compile!
 - But we can never define `Func` for **any** T.
- Solution 1: make template method as friend instead of normal method.

```
template<typename T>  
class A  
{  
    int member;  
  
    template<typename U> friend void Func(A<U> &a);  
};  
  
template<typename U>  
void Func(A<U> &a)
```

```
template<typename T>  
class A  
{  
    int member;  
    friend void Func(A &a);  
};  
  
void Func(A<int> &a)  
{  
    a.member;  
}
```

Friend in class template*

- Limitation: not exactly same as our previous intention. **Optional** as the difference is subtle.
 - Since all instantiation regard template Func as friend, instead of its own normal Func.

```
template<typename T>
class A
{
    int member;

    template<typename U, typename V>
    friend void Func(A<U> &, A<V> &);

    template<typename U>
    friend void Func2(A<T> &, A<U> &);
};

template<typename U, typename V>
void Func(A<U> &a, A<V> &b)
{
    a.member;
    b.member;
}
```

Both `A<int>`
and `A<float>`
regard it as
friend, so okay.

```
template<typename U>
void Func2(A<int> &a, A<U> &b)
{
    a.member;
    b.member;
}

int main()
{
    A<int> a;
    A<float> b;
    Func(a, b); Okay
    Func2(a, b); Error: cannot
                access b.member.
}
```

`A<int>` friend:
`void Func2(A<int>&, A<U>&);`

`A<float>` friend:
`void Func2(A<float>&, A<U>&);`

Different friends, so `A<int>`
friend cannot access `A<float>`
member.

Friend in class template

- Solution 2: specify template instantiation as friend.
 - Limitation: hard to code; we need two additional forward declarations.

```
template<typename T> class A; // 声明A
template<typename T> void Func(A<T>&); // 声明Func
template<typename T> // 定义A
class A
{
    friend void Func<T>(A<T>& a); // 声明友元函数是一个模板实例
};
template<typename T> void Func(A<T>&){} // 定义Func
```

We notice that no **template** keyword is needed here, unlike explicit instantiation.

- Solution 3: define friend methods inside the class directly, and use ADL to call it.
 - Limitation: ADL is quite obscure.

Friend in class template

- For friend class:

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

template<typename T>
class A
{
    template<typename U> friend class B; // All instantiations of B are friend.
    friend class B<T>; // Only one instantiation, B<T>, is friend.
    friend T; // Template parameter as friend, never add "class" keyword.
```

- For **friend T**, if **T** isn't a class, it will be ignored.

Supplementary Miscellaneous Knowledge

- Supplementary
 - Template parameter that's not a type
 - Type Deduction
 - Friend in class template
 - Laziness

Lazy Instantiation

- When a class template is instantiated, not all of its members are immediately fully instantiated.
 - Some of them will only be fully instantiated **when they're actually used.**

³ The implicit instantiation of a class template specialization causes

- (3.1) — the implicit instantiation of the declarations, **but not of the definitions,** of the non-deleted class member functions, member classes, scoped member enumerations, static data members, member templates, and friends; and
- (3.2) — the implicit instantiation of the definitions of deleted member functions, unscoped member enumerations, and member anonymous unions.

The implicit instantiation of a class template specialization does not cause the implicit instantiation of default arguments or *noexcept-specifiers* of the class member functions.

C++ standard, section [\[temp.inst\]](#).

Lazy Instantiation

- Example:

```
template<int N> // This should be std::size_t at best; we just want to
               // illustrate lazy instantiation, so here we need N <= 0
               // to make Array<N> ill-formed.
class Array
{
public:
    int arr[N];
};

template<typename T, int N>
class A
{
public:
    // Successful ones
    void Test() { Array<N> arr; }
    struct Test2
    {
        Array<N> arr;
    };

    // Failed ones
    void Test3(Array<N> arr) {}
    union {
        Array<N> arr;
        int m;
    };
};
```

E.g. we define
`A<int, -1> a`.

As long as we don't call / use them
to force their full instantiation.

Declaration is not valid.
Only definition can be
lazy instantiated.

Lazy Instantiation

- Similarly, there exists “laziness” in some other cases.
 - Value template initialization:

```
template<typename T>  
T v = T::default_value();
```

 - When value of `v` is not used, `default_value()` is allowed to not exist in `T`.
 - For example, `decltype(v<int>)` will compile successfully.
 - Default value:

```
void Func(T a = T{}) {}
```

 - When default value isn't used, this assignment is allowed to make no sense.
 - E.g. when `Func` is called without filling default value (i.e. only calling `Func(xxx)`, never calling `Func()`), `T` is allowed to be not default constructible.
 - Pointer definition:

```
void Test4(Array<N> *arr) {}
```

 - Definition of `Array` isn't checked so `N = -1` won't make compilation fail.
 - And concept (check our homework in the last lecture!).

5. 补充一个知识，模板类成员函数可以对其模板参数施加约束，当约束不满足时不会导致编译错误，只会去掉这个成员函数。判断以下代码中哪些语句不能够编译通过：

Lazy Instantiation

- Some laziness depends on compiler.
 - Virtual function: some compilers always instantiate virtual function to construct full vtable.
 - Semantic error: instantiation check is after syntactic and semantic check theoretically, while some compilers will drop semantic check if it's not instantiated.

Phase 7: Compiling

Compilation takes place: each preprocessing token is converted to a `token`. The tokens are syntactically and semantically analyzed and translated as a `translation unit`.

Phase 8: Instantiating templates

Each translation unit is examined to produce a list of required template instantiations, including the ones requested by `explicit instantiations`. The definitions of the templates are located, and the required instantiations are performed to produce *instantiation units*.

```
void error() { Array<-1> boom; }
```

Defined in `class A`;
Not all compilers detect this
as error if it's not called.

Advanced Template

Variadic Template

Variadic Template

- Variadic Template
 - Basics (Pack expansion & ...)
 - Fold expression

Variadic Template

- Many functions can accept unbounded number of parameters...
 - `std::vector::emplace_back`, `std::format`, `std::invoke`, etc..
- This is enabled by variadic template!
 - Let's use printing a bunch of parameters as example.

```
#include <iostream>
```

```
void print ()  
{  
}
```

```
template<typename T, typename... Types>
```

```
void print (T firstArg, Types... args)
```

```
{
```

```
    std::cout << firstArg << '\n'; // print first argument
```

```
    print(args...);
```

```
    // call print() for remaining arguments
```

```
}
```

```
std::string s("world");
```

```
print (7.5, "hello", s);
```

firstArg is 7.5
and T is double.

args are "hello", s
and Types are const
char*, std::string.

Called *template parameter pack*,

meaning "any number of template parameter".

Called *function parameter pack*,

meaning "any number of parameter".

Called *pack expansion*,
meaning unpack `args`
to `arg1`, `arg2`,

```
print(7.5, "hello", s) -> print("hello", s) -> print(s) -> print()
```

Variadic Template

- Every function call passes value type, which incurs overhead.
 - They're read-only, make them `const&!`

```
template<typename T, typename... Args>  
void print(T firstArg, const Args &...args)
```

- And we may want to eliminate empty function...
 - That is, when number of args is 0, do nothing.
 - You can use operator `sizeof...(pack)`.

```
template<typename T, typename... Args>  
void print(T firstArg, const Args &...args)  
{  
    std::cout << firstArg << "\n";  
    if constexpr (sizeof...(args) != 0)  
    {  
        print(args...);  
    }  
}
```

Here you can also use `sizeof...(Args)`; any pack is ok.

Variadic Template

- Exercise 1: write declaration of `std::invoke` (use universal ref, no need to add concept).

```
template<typename F, typename... Args>  
decltype(auto) Invoke(F func, Args &&...args)
```

- And we need to forward `args` to `func`.
 - Assuming that `func` is called by `operator()`. `return func(std::forward<Args>(args)...);`
- So generally, pack expansion is:
 - Write a pattern, as if operating on a single normal parameter.
 - E.g. here the pattern is `std::forward<Args>(args)`.
 - And add `...` (sometimes need additional space before `...`), meaning that applying pattern on every element and concatenating with `,`.
 - So it's like `std::forward<Arg1>(arg1), std::forward<Arg2>(arg2)...`
 - Notice that it's semantic substitution, **not a comma expression**.

Variadic Template

- Exercise 2: write a function that accepts a container and many indices, and passing corresponding elements to `print`.

```
template<typename T, typename... IdxTypes>
void print_elems(const T &container, IdxTypes... idx)
{
    print(container[idx]...);
}
```

- Exercise 3: what's the type of `result`?

```
template<typename... T1>
class A
{
public:
    template<typename... T2>
    static auto func()
    {
        return std::tuple<std::pair<T1, T2>...>();
    }
};
```

```
auto result = A<char, int>::func<float, double>();
```

Answer: `std::tuple<std::pair<char, float>, std::pair<int, double>>`

Variadic Template

- [Hard] Exercise 4: explain what `g` accepts.

```
template<typename... OuterTypes>
class Nested {
    template<typename... InnerTypes>
    void f(InnerTypes const&... innerValues) {
        g(OuterTypes(InnerTypes(innerValues)...)...);
    }
};
```

Assuming
`sizeof...(OuterTypes) == N,`
`sizeof...(InnerTypes) == M.`

- `InnerTypes(innerValues)...` is expanded to `I1{iv1}, I2{iv2}, ...`
 - And these `M` parameters are used to construct some type `OuterTypes`, forming a new pattern.
- Expand this pattern, we get `O1{ I1{iv1}, I2{iv2}, ... }, O2{ I1{iv1}, I2{iv2}, ... }, ...`
 - And these `N` parameters are passed to `g`.

Variadic Template

- Exercise 5: write a class that **public** inherits all of its template parameters.

```
template<typename... T1>  
class A : public T1...
```

- Note: since C++17, it's allowed to inherit all methods by expansion:

```
template<typename... T1>  
class A : public T1...  
{  
public:  
    using T1::T1...; // inherit all ctor  
};
```

Variadic Template

- Exercise 6: write a ctor to accept params to initialize each base class.

```
template<typename... T1>
class A : public T1...
{
public:
    // 注意不能用T1&&, 因为不是universal reference.
    template<typename... T2>
    A(T2 &&...args) : T1{ std::forward<T2>(args) }...
    {
    }
};
```

Variadic Template

- Exercise 7: explain code below.

```
template<typename... Funcs>
struct Overloaded : Funcs...
{
public:
    using Funcs::operator()...;
};

int main()
{
    auto overloads = Overloaded{ [](int i) { return std::to_string(i); },
                                [](const std::string &s) { return s; },
                                [](auto &&val) { return std::string{}; } };
    std::variant<int, std::string> v{ 1 };
    std::visit(overloads, v);
}
```

Variadic Template

- This is basically equivalent to generate:

```
struct Overloaded0 : A, B, C
{
public:
    std::string operator()(int i) const { return std::to_string(i); }
    std::string operator()(const std::string &s) const { return s; }
    std::string operator()(auto &&val) const { return std::string{}; }
};
```

- Here A, B, C are compiler-generated types for lambda, and each of them generates an overload for `operator()`.
- Notice that aggregates can have base class and implicit CTAD guides only since C++20; in C++17 we need additional guides.

Variadic Template

- Exercise 8: write deduction guide for **Overloaded** in C++17.

```
template<typename... Args>  
Overloaded(Args...) -> Overloaded<Args...>;
```

- Exercise 9: what does **g(1, 2)** print?

```
template<class... Args>  
int h(Args... args)  
{  
    return sizeof...(Args);  
}  
  
template<class... Args>  
void g(Args... args)  
{  
    print(h(args) + args...);  
    print(h(args...) + args...);  
}
```

Expansion pattern will examine all previous code as a whole!

Answer: 2 3 3 4

$h(\text{args}) + \text{args}...$ \Leftrightarrow
 $h(\text{arg1}) + \text{arg1},$
 $h(\text{arg2}) + \text{arg2}, \dots$

$h(\text{args}...) + \text{args}...$ \Leftrightarrow
 $h(\text{arg1}, \text{arg2}, \dots) + \text{arg1},$
 $h(\text{arg1}, \text{arg2}, \dots) + \text{arg2}, \dots$

Variadic Template

- Note 1: semantic substitution is different from plain text substitution. It has two steps:
 - Syntactic parse: examine what meaning the code wants to explain; this is unrelated to size of pack.
 - Substitution: insert types in AST.

- This explains why it works when `sizeof...(T1) == 0`:

```
template<typename... T1>
class A : public T1...
```

 - If it's just text substitution, then it's `class A: public {}` (wrong syntax!).
 - If it's parsed to AST and then types are substituted, then it means "A inherits 0 type", which is correct.

- Note 2: variadic pack can only be the last parameters.

```
template<typename... Ts, typename T>
void Error(Ts... args, T arg);
```

```
Error(1, 2);
```

Intuitive reason: pack will match eagerly, so `args` matches `1, 2` and `arg` matches nothing.

Variadic Template

- But this is Okay:

```
template<typename... Ts> class Y {};  
  
template<typename... Ts, typename U>  
class Y<U, Ts...> {};
```
- Reason: `Ts...` is written at last so it matches last. This is unrelated to position of template parameter declaration (as it can be deduced)!
- Note 3: variadic template is less preferred than normal template in overload resolution.

```
template<typename T> void print(T); // preferred.  
template<typename... Ts> void print(Ts...);
```
- But when non-variadic one has more parameters filled with default, it's still ambiguous.*

```
// ambiguous when calling print(1), #1 preferred when print(1,2)  
template<typename T> void print(T, T = T());  
template<typename T, typename... Ts> void print(T, Ts... args);
```

*: Though this is regulated in C++ standard, only clang implements it.

Variadic Template

- Note 4: NTTP can also use variadic template.

```
template<std::size_t... Idx, typename T>
void Access(const T &container)
{
    print(container[Idx]...);
}
```

```
std::vector<int> v{ 1, 2, 3 };
Access<1, 2>(v);
```

- But there is no way to hybrid two variadic templates.
 - i.e. impossible to define “any number of template parameter, either type or non-type”.
- Note 5: pack can be indexed at compile time since C++26.
 - But only id pack; expression pack cannot be indexed.

```
template<typename... Ts>
void Test(Ts... args)
{
    auto &arg0 = args...[0];
    using LastType = Ts...[sizeof...(args) - 1];
}
```

Variadic Template

- And template template parameter pack isn't allowed to index.
- Note 6: friend can also be expanded pack since C++26.

```
struct C {};  
struct E { struct Nested; };  
  
template<class... Ts>  
class R  
{  
    friend Ts...;  
};  
  
template<class... Ts, class... Us>  
class R<R<Ts...>, R<Us...>>  
{  
    friend Ts::Nested..., Us...;  
};  
  
R<C, E> rce;           // classes C and E are friends of R<C, E>  
  
R<R<E>, R<C, int>> rr; // E::Nested and C are friends of R<R<E>, R<C, int>>
```

Variadic Template

- Note 7: to capture variadic arguments in lambda, you need to:

```
void func(auto... args) {  
    int a = 0;  
    [&, args...]() { }; // 对a进行引用, args按值拷贝  
}
```

- However, you cannot transform it to a named parameter like normal capture:

```
void func(std::unique_ptr<int> ptr0) {  
    [ptr = std::move(ptr0)]() { };  
}
```

- Since C++20, this is improved so that you can write:

```
[...args_ = std::move(args)]() {  
    return foo(args_...);  
};
```

Variadic Argument

- We notice that C also has variadic argument (e.g. `printf`), but it's very obscure to use.

- We're not going to talk about how to use it, but several notes:

1. It still uses ellipsis, but not on a pack expansion:

```
int printx(const char* fmt, ...);
```

- Comma before ... can be omitted (deprecated since C++26), so you can write something like:

```
template<typename T> template<typename... Ts>  
void Test(T...);      void Test(Ts.....);
```

Variadic
template

Variadic
argument

2. It has lowest overload resolution precedence, so `void Test(...)`; means "the least preferred overload".

- We'll utilize this property in SFINAE.

Variadic Template

- Variadic Template
 - Basics (Pack expansion & ...)
 - Fold expression

Fold expression

e.g. $f1() +_1 f2() +_2 f3()$, it's $\text{root}(+_2) \rightarrow \text{lChild}(f1() + f2()) \rightarrow \text{rChild}(f3())$, while lChild is $\text{root}(+_1) \rightarrow \text{lChild}(f1()) \rightarrow \text{rChild}(f2())$;

- We can know before $+_1$ is evaluated, lChild and rChild is first evaluated.
- However, you can evaluate in the sequence of:
 - lChild evaluates $f1()$
 - rChild evaluates $f3()$, gets the value.
 - lChild evaluates $f2()$, gets the value.
 - This still obeys our rules, e.g. $f1()$ and $f2()$ evaluated before lChild .
- So if we output a in $f1()$, b in $f2()$, c in $f3()$, any permutation of abc is possible!

- With only simple pack expansion, print is still very strange...
 - Why do we have to do recursion? Normally it should just be a loop...
- To reduce recursive operations in pack, fold expression is introduced since C++17.
 - For example, to add all elements in a pack:

```
template<typename... Args>
auto AddAll(Args... args)
{
    auto result = (args + ... + 0);
    return result;
}
```

- $(\text{Pack OP } \dots \text{ OP Init})$, meaning $(\text{Pack}_1 \text{ op } (\dots \text{ op } (\text{Pack}_{N-1} \text{ op } (\text{Pack}_N \text{ op } \text{Init}))))$
- We notice that it doesn't necessarily mean $\text{Pack}_N \text{ op } \text{Init}$ evaluates first; this depends on **evaluation order**, as we reviewed in Lecture 1.

Fold expression

- Exercise: write print by fold expression.
 - Hint: comma expression.

`()` is necessary to raise precedence; in fold expression, pack should be with precedence not lower than cast.

```
template<typename... Args>
void print(const Args &...args)
{
    ((std::cout << args << " "), ..., 0);
}
```

Comma expression will evaluate one by one from left to right, so the output is determined.

- Besides binary right fold, there also exists binary left fold:
 - $(\text{Init OP } \dots \text{ OP Pack})$, meaning $(((((\text{Init op Pack}_1) \text{ op Pack}_2) \text{ op } \dots) \text{ op Pack}_N)$
 - They differ when operation is not commutative:

```
template<typename... Args>
void Test(const Args &...args)
{
    std::cout << (args - ... - 0) << " " << (0 - ... - args) << '\n';
}
```

$\text{args} == 1, 2, 3,$
 $\text{args} - \dots - 0 \Leftrightarrow$
 $(1 - (2 - (3 - 0))) = 1 - (-1) = 2,$
 $0 - \dots - \text{args} ==$
 $((0 - 1) - 2) - 3 = -6,$

Fold expression

- And besides binary fold, there also exist unary fold:
 - Unary right fold: $(Pack\ OP\ \dots)$, meaning $(Pack_1\ op\ (\dots\ op\ (Pack_{N-1}\ op\ Pack_N)))$
 - Unary left fold: $(\dots\ OP\ Pack)$, meaning $((Pack_1\ op\ Pack_2)\ op\ \dots)\ op\ Pack_N$
- Without `Init`, this expression is only valid when `sizeof...(Pack) > 0!`

```
template<typename... Args>
auto AddAll(Args... args)
{
    return (args + ...);
}
```

`AddAll()` will lead to compilation error.

- Exception 1: comma expression with 0 size is right, which means nothing.

```
template<typename... Args>
void print(const Args &...args)
{
    ((std::cout << args << " "), ...);
}
```

`print()` is correct, nothing is done.
(Formally result is `void`).

Fold expression

Notice that it may be astonishing when `operator&&/||` is overloaded to return types other than `bool`. Then:
`sizeof...(values) == 0` → return type is `bool`;
`sizeof...(values) > 0` → return type is overloaded type.

- Exception 2: logical expression with 0 size is correct.
 - Return a value that doesn't interfere its operation result (i.e. not short-circuit).
 - For `&&`, return true; for `||`, return false.

```
template<typename... Values>
int allTrue(const Values &...values)
{
    return (... && values);
}
template<typename... Values>
int anyTrue(const Values &...values)
{
    return (... || values);
}
int main()
{
    std::cout << allTrue(1, 1, 0) << allTrue(1, 1) << allTrue() << '\n'; // 011
    std::cout << anyTrue(1, 1, 0) << anyTrue(0, 0) << anyTrue() << '\n'; // 100
    return 0;
}
```

Fold expression

- Since C++26, it's also allowed to reasonably use fold expression in constraints.
 - Before that, it's valid but **doesn't subsume**.

```
template <class T> concept A = std::is_move_constructible_v<T>;  
template <class T> concept B = std::is_copy_constructible_v<T>;  
template <class T> concept C = A<T> && B<T>;
```

```
// in C++23, these two overloads of g() have distinct atomic constraints  
// that are not identical and so do not subsume each other: calls to g() are ambiguous  
// in C++26, the folds are expanded and constraint on overload #2 (both move and copy  
// required), subsumes constraint on overload #1 (just the move is required)
```

```
template <class... T>  
requires (A<T> && ...) void g(T...); // #1
```

```
template <class... T>  
requires (C<T> && ...) void g(T...); // #2
```

Advanced Template

SFINAE

SFINAE

- We've already learnt *concept* to exclude unintended types.
 - But how is it implemented before C++20?
- By SFINAE (Substitution Failure Is Not An Error)...
 - Pronounced as si-fee-nay.
- As its name, for a template, when substituting corresponding template parameters leads to an invalid result...
 - It's just excluded from viable candidates during overload resolution, not causing compilation error.

• For example:

```
template<typename T>
T Func(int)
{
    std::println("AAA");
    return {};
}
```

```
template<typename T>
T Func(typename T::value_type param)
{
    std::println("BBB");
    return {};
}
```

`Func<int>(1)` is valid though `int::value_type` doesn't exist.

It's just excluded from overload resolution.

SFINAE

- But overload resolution only checks declaration (i.e. prototype), so invalid function body will still lead to compilation error.
- Normally, SFINAE cooperates with two structs in `<type_traits>`:
 - `std::enable_if<Cond, T = void>` to check a Boolean;
 - When `Cond` is `true`, it defines `type = T`;
 - When `Cond` is `false`, it doesn't define `type`;
 - `std::void_t<Args...>` to check expressions;
 - E.g. `std::void_t<decltype(Exp1), decltype(Exp2)>`, if `Exp1/Exp2` is invalid, `decltype(Exp)` is invalid.
 - If all types are valid, then it's alias of `void`.
 - These two can already convey most of require expressions.

SFINAE


```
Defined in header <iterator>
struct input_iterator_tag {}; (1)
struct output_iterator_tag {}; (2)
struct forward_iterator_tag : public input_iterator_tag {}; (3)
struct bidirectional_iterator_tag : public forward_iterator_tag {}; (4)
struct random_access_iterator_tag : public bidirectional_iterator_tag {}; (5)
struct contiguous_iterator_tag : public random_access_iterator_tag {}; (6) (since C++20)
```

- Example 1: implement `std::advance(it, n)`.
 - For random access iterator, it should be `it += n`;
 - Otherwise increment in a loop.
- First implement iterator tag check:

```
template<typename Iterator>
constexpr bool IsRandomAccess = std::is_convertible_v<
    typename std::iterator_traits<Iterator>::iterator_category,
    std::random_access_iterator_tag>;

template<typename Iterator>
constexpr bool IsInput = std::is_convertible_v<
    typename std::iterator_traits<Iterator>::iterator_category,
    std::input_iterator_tag>;
```

`is_convertible`
is due to tag
inheritance.



SFINAE

- And then define different overloads:

```
template<typename Iterator, typename Distance>
std::enable_if_t<IsRandomAccess<Iterator>> advanceIterImpl(Iterator &x,
                                                           Distance n)
{
    x += n;
}

template<typename Iterator, typename Distance>
std::enable_if_t<(IsInput<Iterator> && !IsRandomAccess<Iterator>)>
advanceIterImpl(Iterator &x, Distance n)
{
    while (n > 0)
        ++x, --n;
}
```

Necessary,
otherwise
ambiguous call
for non-random-
access iterator.

SFINAE

We notice that `void` in `test(void*)` is necessary to make it has higher precedence than `(...)` when calling (i.e. `...` should match at least one param). See [stackoverflow](#).

- Example 2: Implement `std::is_default_constructible`.
 - We might as well use a method that differs in return type.

```
template<typename T>
struct IsDefaultConstructible
{
private:
    template<typename U, typename = std::void_t<decltype(U())>>
    static char test(void *); When U() is invalid, this is excluded.
    template<typename>
    static long test(...); Least preferred in overload resolution; if test<T>
    doesn't match method above, it matches this one.

public:
    static constexpr bool value =
        std::is_same_v<decltype(test<T>(nullptr)), char>;
}; true ⇔ same ⇔ first overload is valid ⇔ T() is valid ⇔ default constructible
```

We can also choose to not use `std::void_t` here; but when there are more types, you need more `typename = ...` if no `std::void_t`.

SFINAE

- Class template can also use SFINAE by its template declaration.
- Solution2:

```
template<typename, typename = void>
struct IsDefaultConstructible2 : std::false_type {};
template<typename T>
struct IsDefaultConstructible2<T, std::void_t<decltype(T())>> : std::true_type {};
```

Equiv. to define static constexpr bool value = false.

- We notice that `void` in `typename = void` is necessary.
- Reason:
 - When `T()` is valid, the specialization is equiv. to specialize `<T, void>`.
 - Users need to call `IsDefaultConstructible<T>`, and filling `void` can make it match specialization (if it's not SFINAE'ed out).

SFINAE

- Example 3: Implement **Dictionary**.
 - When it supports **operator<**, use **std::map**;
 - Otherwise when it supports **std::hash**, use **std::unordered_map**;
 - Otherwise manually implement **std::vector**-based linear search.
- Is it correct to code like this?

```
template<typename Key>
using SupportLessThan = decltype(Key{} < Key{});

template<typename Key>
using SupportHash = decltype(std::hash<Key>{}(Key{}));
```

```
template<typename Key, typename Value, typename = void>
class Dictionary
{
};

template<typename Key, typename Value>
class Dictionary<Key, Value, std::void_t<SupportLessThan<Key>>>
{
};

template<typename Key, typename Value>
class Dictionary<Key, Value, std::void_t<SupportHash<Key>>>
{
};
```

SFINAE

- Not really, since if a type supports both less-than and hash, two specializations will conflict.
 - But unlike concept, we cannot negate the expression to “check it doesn’t work” in `void_t`.
 - Solution: convert this check to bool value and use `enable_if...`
- So first write two struct that check this (try it yourself)!

It’s just like defining a concept, but in a very strange way.

```
template<typename Key, typename = void>
struct SupportLessThan : std::false_type {};

template<typename Key>
struct SupportLessThan<Key, std::void_t<decltype(Key{} < Key{})>>
    : std::true_type {};

template<typename Key, typename = void>
struct SupportHash : std::false_type {};

template<typename Key>
struct SupportHash<Key, std::void_t<decltype(std::hash<Key>{}(Key{}))>>
    : std::true_type {};
```

SFINAE

- Then specialize like this:

```
template<typename Key, typename Value, typename = void>
class Dictionary
{
};

template<typename Key, typename Value>
class Dictionary<Key, Value, std::enable_if_t<SupportLessThan<Key>::value>>
{
};

template<typename Key, typename Value>
class Dictionary<
    Key, Value,
    std::enable_if_t<SupportHash<Key>::value && !SupportLessThan<Key>::value>>
{
};
```

SFINAE

```
std::void_t<decltype(Key{} < Key{})>>
```

- A small problem: what if Key isn't default constructible?
 - This is still excluded, which is not as expected.
- In concept, we introduce "hypothetic parameters" in requires expression to avoid that.
- Before that, we can use `std::declval<T>()`.
 - It returns `T&&`, but **has no implementation**.
 - Yes, this function cannot be really called; it's just used to pretend to return something, which can be used for expression.
 - So in our example, it should be:

```
struct SupportLessThan<  
    Key, std::void_t<decltype(std::declval<Key>() < std::declval<Key>())>>  
    : std::true_type
```

```
struct SupportHash<Key,  
    | | | | std::void_t<decltype(std::hash<Key>{}(std::declval<Key>()))>>  
    : std::true_type
```

Of course, you can `std::declval<std::hash<Key>>()` if you want.

SFINAE

- But notice that since the function theoretically adds rvalue reference, it cannot e.g. call lvalue-only methods.
 - You need e.g. `std::declval<Key&>` to make lvalue also work (due to reference collapsing).

```
class B {
public:
    void Test() &;
};

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

template<typename T>
class A<T, std::void_t<decltype(std::declval<T>().Test())>> {
public:
    inline constexpr static bool value = true;
};
static_assert(A<B>::value);
```

SFINAE-friendly Declaration

- Notice again that if error doesn't happen in head, SFINAE won't kick in.
 - Thus, a SFINAE-friendly traits should expose all requirements there.
 - Specifically:

```
template<typename T> class A{ /* NO SFINAE */ };  
template<typename T> T a = /* NO SFINAE */;  
template<typename T> void f(T x = /* NO SFINAE */xx);  
template<typename T> void f() noexcept(/* NO SFINAE */);
```

- Example 1: if default argument is invalid, it will cause compilation error directly.

SFINAE-friendly Declaration

- Example 2: `addA` will be SFINAE'ed out if `t + u` is invalid, but `addB` won't.

```
template<typename T, typename U>
auto addA(T t, U u) -> decltype(t + u)
{
    return t + u;
}
template<typename T, typename U>
decltype(auto) addB(T t, U u)
{
    return t + u;
}
```

- Example 3: if we want to write a generic `operator+` for our `Array<T>`.

*Actually the standard regulates that it should cause an error, but msvc doesn't implement it.

SFINAE-friendly Declaration

- So if we define an overload for e.g. `Array<A> + Array`, when `A+B` is invalid, whether it compiles depends on whether `PlusResult` is instantiated...
 - It's compiler-dependent*.
- So if you really want to use struct, you'd better to expose it to head (e.g. with partial specialization):

```
template<typename T1, typename T2, typename = void>
struct PlusResult {};

template<typename T1, typename T2>
struct PlusResult<
    T1, T2, std::void_t<decltype(std::declval<T1>() + std::declval<T2>())>>
{
    using Type = decltype(std::declval<T1>() + std::declval<T2>());
};
```

std::conditional

- Finally supplement some knowledge on `std::conditional`.
- Sometimes we may find it inconvenient to define specialization for if-then-else case.
 - For example, in Dictionary example, the specializations are like (in pseudo code):

```
if (SupportLessThan<Key>)  
    using ProperDictionary = std::map<Key, Value>;  
else  
    using ProperDictionary = std::unordered_map<Key, Value>;
```

- Instead, we can write with `std::conditional<Cond, TrueType, FalseType>`:

```
template<typename Key, typename Value>  
using ProperDictionary =  
    std::conditional<SupportLessThan<Key>::value, std::map<Key, Value>,  
                    std::unordered_map<Key, Value>>::type;
```

std::conditional

- And then:

```
template<typename Key, typename Value>
class Dictionary<
    Key, Value,
    std::enable_if_t<SupportLessThan<Key>::value || SupportHash<Key>::value>>
    : public ProperDictionary<Key, Value>
{
};
```

- You can also use `std::conditional_t`:

```
template<typename Key, typename Value>
using ProperDictionary =
    std::conditional_t<SupportLessThan<Key>::value, std::map<Key, Value>,
                    std::unordered_map<Key, Value>>;
```

- But not in all cases `std::conditional` can be used to directly substitute specialization, when `TrueType` or `FalseType` may be invalid.

std::conditional

- For example, if we want to define a `TryUnsignedType<T>...`
 - For integer (but not `bool`) `T`, make it unsigned;
 - Otherwise original `T`.

- We may write it like:

```
error C2338: static_assert failed: 'make_unsigned<T> requires that T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type.'
```

```
template<typename T>
struct TryUnsignedT
{
    using type =
        std::conditional_t<std::is_integral_v<T> && !std::is_same_v<T, bool>,
                          std::make_unsigned_t<T>, T>;
};
```

- Problem: `std::make_unsigned_t<T>` is valid only when condition is `true`; but to check whether the expression is valid, both branches will be checked in any condition.

std::conditional

- We've seen similar problem before, where we have to use specialization:

- 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.

std::conditional

- Instead, we can also defer type evaluation!

```
template<typename T>
struct TryUnsignedT
{
    using type = typename std::conditional_t<
        std::is_integral_v<T> && !std::is_same_v<T, bool>,
        std::make_unsigned<T>, std::type_identity<T>>::type;
};
```

- Though `make_unsigned<T>::type` may not exist, `make_unsigned<T>` always exist! After selecting correct branch, we use `::type` to get the actual type.
 - `std::type_identity<T>` is just using `type = T`, as a wrapper in this case.
- *Our dictionary example doesn't need to care about this problem since less-than operator is only needed when methods are called (due to lazy instantiation).

Advanced Template

Commonplace Techniques

Commonplace Techniques

- Commonplace Techniques
 - CRTP
 - Type Erasure

C RTP

- *Curiously Recurring Template Pattern* (CRTP) is a technique for **static polymorphism**.
 - For dynamic polymorphism, we use inheritance with virtual functions and usually need indirect jump by vtable.

```
class Base
{
public:
    virtual void Func() = 0;
};
```

```
class Derived1 : public Base
{
public:
    void Func() override {}
};
```

```
class Derived2 : public Base
{
public:
    void Func() override {}
};
```

```
void Test(Base &obj)
{
    obj.Func();
}
```

- But in template, we can achieve it by defining the same set of members & template function:

```
class Derived1
{
public:
    void Func() {}
};
```

```
class Derived2
{
public:
    void Func() {}
};
```

```
template<typename Base>
void Test(Base &obj)
{
    obj.Func();
}
```

CRTP

- In dynamic polymorphism, we can use template method pattern to reduce implementation code in derived class.
 - As we reviewed in Lecture 2...
- In static polymorphism, we can achieve it in this way:

```
template<typename Derived>
class StudentBase
{
public:
    float GetGPA()
    {
        return static_cast<Derived*>(this)->GetGPACoeff() * 4.0f;
    }
};
```

```
class Student : public StudentBase<Student>
{
public:
    float GetGPACoeff() { return 0.8f; }
};
```

```
class JuanWang : public StudentBase<JuanWang>
{
public:
    float GetGPACoeff() { return 1.0f; }
};
```

```
class Student
{
public:
    float GetGPA() { return GetGPACoeff() * 4.0; }
private:
    virtual float GetGPACoeff(){ return 0.8; };
};

class JuanWang: public Student
{
    virtual float GetGPACoeff(){ return 1.0; };
};
```

```
template<typename T>
void Func(StudentBase<T> &student)
{
    student.GetGPA();
}
```

CRTP

- Of course, you can hide implementation by making base class friend:

```
template<typename Derived>
class StudentBase
{
public:
    using Super = StudentBase<Derived>;
```

```
class Student : public StudentBase<Student>
{
    friend Super;           Type alias as friend
                            doesn't need "class".
    float GetGPACoeff() { return 0.8f; }
};
```

- What makes it “curious” is that a class can be used as template argument before it has complete definition, and inherits from that template.
 - You can also return different types in methods of derived class, which is more powerful than covariance.
- Since C++23, you can use deducing this to simplify CRTP in some cases.

CRTP

```
class StudentBase
{
public:
    using Super = StudentBase;
    float GetGPA(this auto &&self) { return self.GetGPACoeff() * 4.0f; }
};
```

```
class Student : public StudentBase
{
    friend Super;
    float GetGPACoeff() { return 0.8f; }
};
```

```
class JuanWang : public StudentBase
{
    friend Super;
    float GetGPACoeff() { return 1.0f; }
};
```

```
Student student;
student.GetGPA();
```

Intuitively, `student.GetGPA()` \Leftrightarrow `GetGPA(student)`, so the template parameter can be automatically deduced (i.e. equiv. to `student.GetGPA<Student>()`).

CRTP

- But you cannot write something like:

- Since it has dropped the concrete type, such call can only deduce `T = StudentBase`.

```
void Func(StudentBase &student)
{
    student.GetGPA();
}
```

- Solution: Just use `template<typename T> void Func(T&)`.
 - And optionally add `requires std::derived_from<T, StudentBase>`.

Commonplace Techniques

- Commonplace Techniques
 - CRTP
 - Type Erasure

Type Erasure

- We can notice that dynamic & static polymorphism exhibit different pros & cons.

Dynamic polymorphism in C++ exhibits the following strengths:

- Heterogeneous collections are handled elegantly. E.g. you can use `std::vector<Base*>`.
- The executable code size is potentially smaller (because only one polymorphic function is needed, whereas distinct template instances must be generated to handle different types).
- Code can be entirely compiled; hence no implementation source must be published (distributing template libraries usually requires distribution of the source code of the template implementations).

In contrast, the following can be said about static polymorphism in C++:

- Collections of built-in types are easily implemented. More generally, the interface commonality need not be expressed through a common base class.
- Generated code is potentially faster (because no indirection through pointers is needed a priori and nonvirtual functions can be inlined much more often).
- Concrete types that provide only partial interfaces can still be used if only that part ends up being exercised by the application. i.e. due to lazy instantiation.

Credit: C++
Templates the
Complete Guide,
2nd ed..

Type Erasure

- To bridge the gap between two polymorphism, type erasure is introduced.
 - For example, for `std::function`, obviously e.g. function pointer cannot be derived from it, so direct dynamic polymorphism fails.
 - On the other hand:
 - It only records return type & param types instead of fully recording the original type;
 - And it can be used as a normal object instead of always requiring template method to call the correct methods.which makes static polymorphism impossible.
- Type erasure is to keep dynamic polymorphism in a non-invasive way (i.e. not requiring direct inheritance).

These names (i.e. façade and proxy) are not official, just my personal naming.

Type Erasure

- We still use `Student` example to give an overview, assuming that we just want the class has a `GetGPACoeff()`, and expose `GetGPA()`.
 - There are three key factors in type erasure:
 - #1: A façade that exposes what you want, e.g. `GetGPA()` here.
 - #2: A proxy abstract base that implements as if it's a base in dynamic polymorphism; it has virtual functions that you want, e.g. `GetGPACoeff()` here.
 - #3: A template proxy that inherits from the proxy base, which stores the concrete objects (e.g. `Student` or `JuanWang` here) and implements virtual functions.
 - And façade needs to have:
 - #4: A storage (e.g. pointer + `new`) that stores the template proxy while referred by the pointer of proxy base type.
 - #5: A template constructor to accept any possible type to construct the template proxy.

Type Erasure

- To be specific:
 - #2, the proxy base:
 - #3, the template proxy:

```
class StudentProxyBase
{
public:
    virtual float GetGPACoeff() = 0;
};
```

```
template<typename ConcreteStudentType>
class StudentProxy : public StudentProxyBase
{
    ConcreteStudentType student;

public:
    // You can also add another overload for ConcreteStudentType&&.
    StudentProxy(const ConcreteStudentType &init_student)
        : student{ init_student }
    {
    }
    float GetGPACoeff() override { return student.GetGPACoeff(); }
};
```

Of course, if you don't want two overloads, you can use

```
template<typename T>
requires std::same_as<ConcreteStudentType,
std::remove_cvref_t<T>>
StudentProxy(T&&) and forward.
```

Type Erasure

- #1, the façade:

```
class Student
{
public:
    float GetGPACoeff() const { return 0.8f; }
};

class JuanWang
{
public:
    float GetGPACoeff() const { return 1.0f; }
};

int main()
{
    Student s;
    JuanWang j;

    StudentBase stu{ s };
    std::println("{} ", stu.GetGPA());
    StudentBase stu2{ j };
    std::println("{} ", stu2.GetGPA());

    return 0;
}
```

```
class StudentBase
{
    StudentProxyBase *studentProxy = nullptr;

public:
    StudentBase() = default;
    template<typename T> #5
    StudentBase(T &&student)
    {
        studentProxy = new StudentProxy{ std::forward<T>(student) };
    }

    float GetGPA()
    {
        if (studentProxy == nullptr)
        {
            throw std::runtime_error{
                "Cannot deference without underlying student"
            };
        }
        return studentProxy->GetGPACoeff() * 4.0f;
    }
};
```

#4

Type Erasure

- There are still obvious problems in façade, but it's easy to fix:

1. No dtor to deallocate `studentProxy`;

```
virtual ~StudentProxyBase() = default; ~StudentBase() { delete studentProxy; }
```

2. No proper function to move or copy.

- You need to define `clone` in the proxy, and call it in façade's copy assignment & ctor.
- Move assignment & ctor just operates on the pointer, no need to modify proxy.

```
virtual StudentProxyBase *clone() const = 0;
```

```
StudentBase(const StudentBase &another)  
    : studentProxy{ another.studentProxy->clone() }  
{  
}  
  
StudentBase(StudentBase &&another)  
    : studentProxy{ std::exchange(another.studentProxy, nullptr) }  
{  
}
```

```
StudentProxy *clone() const override  
{  
    return new StudentProxy{ student };  
}
```

Type Erasure

- Of course, we can have lots of variation here:
 1. Do we have to use heap memory for storage? We may utilize stack for small objects!
 - That's just what `std::function` and `std::any` do, called SBO and SOO respectively.
 2. Do we have to use inheritance as "base"? We may drop vtable!
 - For example, we may store the function pointer manually, as we done in homework of Lecture 5. Do it if you haven't!
 - We may even split some pointers to be shared among the same type while cache some metadata. That's what the library [proxy](#) does and is faster!
 3. etc....
- We'll use `std::function` as homework for you.

```
std::any hiddenObject_;  
void (*walkProxy_)(std::any &);  
std::string (*talkProxy_)(std::any &, int);
```

Type Erasure

- To conclude, type erasure is a compromise between two ways:

E.g. `std::vector`
`<std::any>`.



Dynamic polymorphism in C++ exhibits the following strengths:

- Heterogeneous collections are handled elegantly.
- The executable code size is potentially smaller (because only one polymorphic function is needed, whereas distinct template instances must be generated to handle different types).
- Code can be entirely compiled; hence no implementation source must be published (distributing template libraries usually requires distribution of the source code of the template implementations).

Between two
polymorphism



In contrast, the following can be said about static polymorphism in C++:



- Collections of built-in types are easily implemented. More generally, the interface commonality need not be expressed through a common base class.
- Generated code is potentially faster (because no indirection through pointers is needed a priori and nonvirtual functions can be inlined much more often).
- Concrete types that provide only partial interfaces can still be used if only that part ends up being exercised by the application.

Depends on impl.,
like [proxy](#). But never
faster than pure static
polymorphism.



Depends on impl.,
but generally it's impossible
(as we said, virtual function
may force instantiation,
depending on compiler).

Summary

- NTTP and template template parameter
- Type Deduction
 - CTAD and user-defined deduction guide
- Friend
- Lazy instantiation
- Variadic Template
 - Fold expression
- SFINAE
 - `std::conditional`
- CRTP & Type Erasure
- Now you can implement most of classes and functions in the standard library yourself!

Next lecture...

- We'll talk about basics on parallelism.
 - Threads;
 - Locking mechanism, semaphores...
 - Synchronization mechanism, condition variables...
 - High-level abstraction, future-promise model...