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`!

constinit

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

```
static constinit int a = 10;

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

- `constinit` 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:

```cpp
// 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.

```cpp
// 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.

```cpp
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.

```cpp
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:

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

- What if we want to output "There!" when T is int?
  - By specialization!

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

```cpp
Func<char>('a');
Func<int>(1);
```

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

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

```cpp
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.

```cpp
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.

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

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

❌

```cpp
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.

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

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

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

```
Func2<int, double>(0); // T isn' 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!

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

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

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

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

```cpp
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; it 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 hypothetic #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 hypothetic #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:

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

```cpp
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.
  - You may just see it as a normal class.

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

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

```cpp
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 hypothetic 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:

```cpp
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:

```cpp
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:

```cpp
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.

```cpp
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);
```

consteval 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 consteval if since C++23.
    - No parentheses!
  - Negate: if !consteval {…}.

```cpp
template<std::size_t N = 1024>
constexpr float Sin(float x)
{
    if consteval
    {
        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);
    }
}
```

```cpp
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.

is_constant_evaluated

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

```
constexpr bool is_constant_evaluated() {
    if consteval {
        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 consteval.

```
consteval int f(int i) { return i; }

constexpr int g(int i) {
    if consteval {
        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.

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

```cpp
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:

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

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

Then define

```cpp
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:

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

Dependent name

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

- Note: You cannot have an identity 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                          // typename optional
foo(const T& cont, typename T::value_type arg)  // 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*.

Rarely used.

```
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
  ...
}
```

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.

```cpp
namespace std {
    int function = 1;
}

int f = 0;

int main()
{
    std::function<int()> f;
}
```

   - 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.

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.

```cpp
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>
class Shell {
  public:
    template<int N>
    class In {
      public:
        template<int M>
        class Deep {
            public:
            virtual void f();
        };
    };
};
```

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

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

Credit: C++ Templates – The Complete Guide 2nd 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;`

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

int main()
{
    constexpr int a = 1, b = 0;
    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

- 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.

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

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

Nested Specialization*

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

```cpp
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.

```cpp
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
```