

模板基础与移动语义

---

Template Basics and  
Move Semantics

# 现代C++基础 Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

---

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

# Supplementary

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

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

- E.g.

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

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

## 2. Member function template cannot be virtual.

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

# Template Basics

Compile-time Evaluation

# Motivation

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

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

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

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

# constexpr variable

- So we need **constexpr**!

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

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

# constexpr

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

```
static constexpr int a = 10;

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

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

# constinit

- Example:

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

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

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

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

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

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

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



# constexpr function

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

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

# constexpr function

- So C++23 requirements are quite simple:

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

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

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

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

# constexpr function

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

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

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

# constexpr & consteval function

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

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

But normally it is, due to compiler optimization.

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

```
consteval bool IsPrime(unsigned int a)

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

# constexpr & consteval lambda

- These two specifiers can also be added in lambda.

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

- E.g. 

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

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

- E.g. 

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

# Template Basics

Compile-time Branch Selection

# Compile-time Branch Selection

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

# Template Basics

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



# Template specialization

- A simple template function:

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

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

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

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

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

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

# Template specialization

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

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

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

# Template specialization

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

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

# Template specialization

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

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

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



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

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



- These notes also apply on class template specialization.

# Type Deduction

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

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

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

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

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

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

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

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

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

# Overload and specialization

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

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

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

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

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

```
Func(nullptr);
```

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

# Overload resolution

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

# Overload resolution

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



# Overload resolution

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

# Overload resolution

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

# Overload resolution

- So now we know why:

So in fact we have two candidates:

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

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

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

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

# “More specialized”

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

# “More specialized”

- Example:

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

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

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

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

# Template Basics

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

# Class template specialization

- Similarly, you can define full specialization for class:

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

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

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

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

  - You may just see it as a normal class. 

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

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

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

# Class template specialization

- And, you can also define partial specialization!

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

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




# Partial specialization

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

- Example: 

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

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



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

# Partial specialization

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

- Forbidden cases: `template<int N, int M>`  
`class A {};`

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

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

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

# Partial specialization

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

- Example:

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

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

# Template Basics

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

# constexpr if

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

- For example:

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

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

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

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

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

# constexpr if

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

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

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

# constexpr if

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

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

# constexpr if

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

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

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

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



# is\_constant\_evaluated

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

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

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

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

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

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