System Software Group

Metaprogramming From Macro to Template

Yuchi, Chen

Metaprogramming

- Metaprogramming programs a program
  - Interpreter
  - Compiler
  - Assembler
  - Linker
  - Loader

- Move computations from run-time to compile-time

Running Example -- `max(a, b)`

- Implement a feature to return the max one of two variables
- Return the variable which is greater in the partial order
- Repo: https://github.com/yuchiche/template-metaprogramming
- Function Solution → Macro Solution → Template Solution

Function Solution

Function Solution

```
int ia = 10086, ib = 10001;
char stra[] = "10086”, strb[] = "10001";
int max_int(int a, int b)
{
    return a < b ? b : a;
}
char* max_str(char* a, char* b)
{
    return strcmp(a, b) > 0 ? a : b;
}
```

Commit 44391a00b1f7e3e463a21439e0ebfd8044af7f08

- Different function for different type
- Different name for different function

Macro Solution

Macro Solution

```
int ia = 10086, ib = 10001;
char stra[] = "10086", strb[] = "10001";

#define max(a, b) a < b ? b : a

max(ia, ib); // 10086
max(stra, strb); // 10001
max(ia == ib, ia); // 0
```

- Right for integer
- Wrong for C style string since the comparison is between address
- Wrong for expression (different type)
  - `ia == (ib < ia) ? ia : ia == ib`

Commit 95bb3c30fb1db67b478f6698c2a7724a1aeb6acd

Macro Solution

```
int ia = 10086, ib = 10001;
```

```
#define max(a, b) (a) < (b) ? (b) : (a)
10010 + max(ia, ib); // 10086
```

- Wrong for using as a whole expression
  - `10010 + (a) < (b) ? (b) : (a)`

Commit 425af4b898e0adb6699572ce526f7acca9f655ca

```
#define max(a, b) ((a) < (b) ? (b) : (a))
max(ia++, ib++); // 10087
ia, ib; // 10088 10002
```

- Wrong for post self incresing
  - `((ia++) < (ib++) ? (ib++) : (ia++))`

Commit eda92ca847496055142ba450e66426499c726685

Macro Solution

```c
int ia = 10086, ib = 10001;
char stra[] = "10086", strb[] = "10001";
#define max(a, b) ({    \
    typeof(a) _a = a;   \
    typeof(b) _b = b;   \
    _a < _b ? _b : _a;  \
})
max(ia++, ib++); // 10086
ia, ib; // 10087 10002
```

- Using **statement and declaration expression** for expressions having side effects (GNU ONLY)
- Still not working for C style string

Commit 3f5aac45c83d8a4266d3194754f87f0111ccfe48

Template Solution

A Combination of Function and Macro

Template Solution -- Single Template Parameter

```
int ia = 10086, ib = 10001;

char stra[] = "10086", strb[] = "10001";

template <typename T> T max(T a, T b)

{

    return a < b ? b : a;

}

template<> char* max(char* a, char* b)

{

    return strcmp(a, b) ? a : b;

}
```

Commit ae0d7f1ef6ab889d685b7c9957ca5d6fbd20fbfd

- Compiler will instance a new function for every calling type
  - `max(ia, ib)` and `max(stra, strb)` are different functions
- Compilation error for different types since there is only one template parameter T
  - `max(ia == ib, ia)`
  - `deduced conflicting types for parameter 'T' ('bool' and 'int')`

Template Solution -- Multi Template Parameters

```
template <typename T1, typename T2,
    typename R = CommonType<T1, T2>>
typename R::Type max(T1 a, T2 b)
{
    return a < b ? b : a;
}
```

- Add another 2 template parameters
- Return common type R of T1 and T2
- Deduce R in compilation time

Template Solution -- Common Type

- std::common_type<> since C++11

```
template <class _Ty1, class _Ty2>
using _Conditional_type =
    decltype(false ? std::declval<_Ty1>() : std::declval<_Ty2>());
template <class _Ty>
add_rvalue_reference_t<_Ty> declval() noexcept;
template <class _Ty>
using add_rvalue_reference_t = typename _Add_reference<_Ty>::_Rvalue;
```

- Ambiguous for bidirectional convertible types, e.g., int and float, etc.

Template Solution -- Common Type

```
template <typename T1, typename T2>
struct CommonType {
    using Convert2T1 =
        Convertible<T2, T1>;
    using Convert2T2 =
        Convertible<T1, T2>;
    static_assert(...);
    using Type = typename IfElse<
        Convert2T1::value, T1, T2>::Type;
};
```

Commit 83fc663a74e237a8882a45e8b52ae2716d5e6114

- Detect if there exists conversion between T1 and T2
- static_assert() for both or none
- Select the common type based on the detection

- Since C++11:
  - std::is_convertible<>
  - std::conditional<>

SFINAE Based Convertible

SFINAE Based Convertible

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror (**SFINAE**, pronounced like *sfee-nay*)
  - Template substitution process may lead to errors and compiler will simply ignore these errors
- Compiler would do implicit conversions during compilation process
  - Promotion, e.g., `short` to `int`, `int` to `long long`
  - Standard conversion, e.g., `int` to `float`
  - User defined conversion, e.g., constructor
  - ……
- Let the compiler perform conversion during substitution process and SFINAE out failed candidate(s)

SFINAE Based Convertible

```
template <typename From, typename To>
struct Convertible {
    static void auxiliary(To);
    template <typename U, typename =
        decltype(auxiliary(std::declval<U>()))>
        static std::true_type test(void*);
    template <typename U>
        static std::false_type test(...);
    static constexpr bool value =
        decltype(test<From>(nullptr))::value;
};
```

- Define 2 overloaded function `test()` and the former is a better match than the latter when calling with `nullptr`
- Deduce the name-omitted template parameter by passing a right value of type `From` to `auxiliary(To)`
- If the parameter of type `To` could be constructed from the right value, then the deduction succeeds and the former returning `std::true_type` is matched, otherwise the latter

Partial Specialization Based If Else

Partial Specialization Based If Else

- Primary Template
  - A definition with all template parameters are parameterized

- Partial Specialization
  - An alternative definition of template with certain parameters substituted

- Specialization
  - All template parameters are substituted by template arguments

Partial Specialization Based If Else

```
template<bool condition,
    typename IfType, typename ElseType>
struct IfElse {
    using Type = IfType;
};
template<typename IfType,
    typename ElseType>
struct IfElse<false, IfType, ElseType> {
    using Type = ElseType;
};
IfElse<Convertible<T2, T1>, T1, T2>::Type
```

- Define a primary class template with:
  - a non type template parameter (`bool` type)
  - 2 template parameter
  - yield the `IfType` by default
- Define a partial specializaiton with:
  - the non type template parameter to `false`
  - yield the `ElseType`
- Use `Convertible<>::value` as the non type template argument to instance a specialized `IfElse<>`

Template Solution -- Less Than Comparable

```
class MyClass
{
public:
    int n;
    bool operator>(const MyClass &lhs)
    {
        return n < lhs.n;
    }
};
MyClass mca{10086}, mcb{10001};
```

- Compilation error for classes that are not less than comparable
  - `max(mca, mcb)`
  - `no match for 'operator<' (operand types are 'MyClass' and 'MyClass’)`
- Detect whether template argument is less than comparable

Template Solution -- Less Than Comparable

```
template<typename T>
struct LessThanComparable
{
    template <typename U, typename = decltype(std::declval<U>() < std::declval<U>())>
    static std::true_type test(void*);
    template <typename U>
    static std::false_type test(...);

    static constexpr bool value =
        decltype(test<T>(nullptr))::value;
};
```

- If type U has overloaded operator<, then the deduction in template declaration succeeds and the former is matched, otherwise the latter

Template Solution -- Final

```
template <typename T1, typename T2, typename R = CommonType<T1, T2>,
    typename LTCR = LessThanComparable<R>>
typename R::Type max(T1 a, T2 b)
{
    static_assert(LTCR::value, "Common type is not less than comparable.");
    return a < b ? b : a;
}
```

Commit 133308dedd2263a935f247caf1ddf3ac1822314a intel®