Template Meta Programming

> A learning note from reference [1].
>
> Author: Simon Lee

1 Template

1.1 Template

We can treat template as a function, who takes as input some values and returns a new type:

```
(T1, T2, T3, ...) => NT
```

Observe,

```
template <typename T>
class ContainerWrapper {
public:
  T first() { return container.front(); }
private:
  std::vector<T> container;
};
```

We can treat the template type `FunctionName<T>` as a function, namely *FunctionName*, taking as input the type `T` and returning `FunctionName<T>`.

We can now expand the template above, and makes the member `container` to be an parameter,

```
template <typename T, typename Container = std::vector<T>>
class ContainerWrapper {
public:
  T first() { return container.front(); }
private:
  Container container;
};
```

We can see from above, that the template the now similar to the generic programming in Java. The difference is that, in template meta programming (C++), we treat `Container` as a *duck type* (by now in C++ 11), the only two (implied) requirements of which is that (1) `Container` must have a default constructor, and (2) it must have a function of name *front* and of type `() => T`. Recall the same semantics in Java, things get quite different. In Java, we have to appoint that the generic type `Container<T>` implements an interface containing a function `front`, which obeys

the thought of *polymorphism*.

Or, even conciser:

```
template <typename T,
          template <typename E, typename A = std::allocator<E>> class Container
= std::vector>
class ContainerWrapper {
public:
  T first() { return container.front(); }
private:
  Container<T> container;
};
```

Therefore, what is a template ?

> Template is a function, who takes as input a type, a non-type[1], even **a template**, and returns a new type.

1.2 Operation

1.2.1 template specialization

When we've already defined a template, say, `SomeTemplate<A, B, C>`, where `A` / `B` / `C` can be a type, a non-type, and a template, we can do some operations on them to make them behave differently, similar to function overload. Specialization is only dependent on the concrete type, not the template parameters.

- **partial template specialization**

  We can partial specialize `SomeTemplate<A, B, C>` to `PartialSomeTemplate<A, B>` by designate a concrete value `CC` to `C` (a type/non-type/template) to it. And then redefined its behavior (the specialized template). The only requirements here is that, we must guarantee that, the designated value `CC` fits the definition of `C`.

  1. if `C` is a non-type, say a int, then `CC` must be a number which can be calculated while compiling.
  2. if `C` is a type, then `CC` must be a specific type.
  3. if `C` is a template, say `template <typename E, typename A = std::allocator<E>>`, then `CC` must be an instantiation of it, such as `std::vector`

  e.g. we can specialize `ContainerWrapper` defined above:

  ```
  template <typename T>
  class ContainerWrapper<T, std::array> {
    // here, we can stay unchanged, or change it to adapt std::array, e.g.
  delete front, modify free, and other functions, or anything you can do.
  }
  ```

- **full template specialization**

Different from above, a full template specialization is a specialization of a template with all parameters designated legally.

```
template <> // this must not be abondaned
class ContainerWrapper<int, std::vector> {
  // here, we can stay unchanged, or change it to adapt std::vector, e.g. delete front, modify free, and other functions, or anything you can do.
}
```

Once a specialization is done, the compiler will generate the corresponding codes. And when a template is instantiated, the compiler will look for the correct template (like a pattern matching). For instance, the compiler will use `ContainerWrapper<int, std::vector>` to instantiate `ContainerWrapper<int, std::vector>`, use `ContainerWrapper<T, std::array>` to instantiate `ContainerWrapper<double, std::array>`, use `ContainerWrapper<T, template <typename E, typename A = std::allocator<E>> class Container = std::vector>` to instantiate `ContainerWrapper<double, std::deque>`.

Using this, we can do some interesting things, such as implement an operator to check the equivalence of two type,

```
template <typename T1, typename T2>
struct is_type_equal {
  enum { ret = false };
};

template <typename T>
struct is_type_equal<T, T> {
  enum { ret = true };
};

std::cout << is_type_equal<int, float>::ret << std::endl; // prints flase
```

1.2.2 compile-time computation

- **numeric computation**

  For non-type parameters, the compiler can do numeric computation while compiling. Observe,

```
template <int x, int y>
struct calculate {
  enum {
    sum = x + y,
    sub = x - y,
    mul = x * y,
    div = x / y
  };
};
```

```
10
11 int main() {
12   std::cout << calculate<1,2>::sum << std::endl; // prints 3
13   std::cout << calculate<1,2>::sub << std::endl; // prints -1
14   std::cout << calculate<1,2>::mul << std::endl; // prints 2
15   std::cout << calculate<1,2>::div << std::endl; // prints 0
16 }
```

The compiler will compute all the four `enum`s in compile time. And then you can just use `calculate` as an operator like `calculate<1, 2>::sum`.

- **type computation**

  For type parameters, the compiler can do type computation while compiling. Observe,

```
1  template <typename T>
2  struct extract_type {
3    using lref_t          = T &;
4    using rref_t          = T &&;
5    using const_lref_t    = const T &;
6    using const_rref_t    = const T &&;
7    using pointer_t       = T *;
8    using const_pointer_t = const T *;
9  };
10
11 int x = 9;
12 typename extract_type<int>::lref_t lrefx = x;
```

  When compiling, the compiler can compute all the type in `extract_type<T>`.

- **recursive computation**

  The compiler can do recursive computation, too. Observe,

```
1  // numeric computation
2  #define __f(...) fib<__VA_ARGS__>::ret
3
4  template <int N>
5  struct fib {
6    enum { ret = __f(N-1) * __f(N-2) };
7  };
8
9  template <>
10 struct fib<0> {
11   enum { ret = 1 };
12 };
13
14 template <>
15 struct fib<1> {
```

```
  enum { ret = 1 };
};

std::cout << fib<5>::ret << std::endl; // prints 5

// type computation
#define __p(...) typename pointer_of<__VA_ARGS__>::type

template <typename T, int N>
struct pointer_of {
  using type = __p(__p(T, N-1), 1);
};

template <typename T>
struct pointer_of<T, 1> {
  using type = T *;
};

int *px = nullptr;
typename pointer_of<int, 2>::type ppx = &px;
```

Code shown above gives an operator `fib<N>` which can compute the fibonacci number in compilation, and an operator `pointer_of<T, N>` to compute the pointer^N of type T. And we use template specialization to define their stop condition (at N = 0 or/and 1). This is a powerful capability !

As shown above, any computation results are stored inside the template (either `fib<N>::ret` or `extract_type<T>::lref_t`), and we can use them whenever needed.

2 Meta Programming using template

2.1 Meta function

We've seen above that, a template is just a function, of whom the arguments are passed in the angle bracket `<>`, and the return value is stored inside the template. And we name these functions (`calculate` / `extract_type` / `fib` / `pointer_of` /...) *meta functions*, and it is the foundation of C++ meta programming.

2.2 High order meta function

In functional programming, the function itself is a first class member, an ordinary data type like string, int, etc... And a function is a high order function if, it takes as input a function, or returns a function. Similarly, in C++ meta programming, we have high order functions. Observe,

```
template <int X, int Y, template <int> class F>
struct max_if_f {
  enum { ret = F<X>::ret < F<Y>::ret ? Y : X };
};
```

```
5
6  template <int X>
7  struct square {
8    enum { ret = X * X };
9  };
10
11 template <int X, int Y>
12 using max_if_square = max_if_f<X, Y, square>;
13
14 std::cout << max_if_square<-5, 2>::ret << std::endl; // prints -5
```

As shown, we create a *high order meta function* `max_if_t`, who takes as input two ints, and a *meta function* `F`, then returns the max value between `F` of them. And we pass `square` to it then alias it as `max_if_square`, and this is called *Meta Function Forwarding*.

> Attention here: C++ (11) does not allow the template specialization inside a template or class, and if needed, define them outside, and use `using` to refer to them.

2.3 Anything is a type

As codes above shown, we use `enum { ret = … }` to represent the return value, and `using type = …` to represent a return type. This is not that acceptable in our subsequent processing. For unification, we make every thing a type. Observe,

```
1  template <int X>
2  struct aint {
3    enum { value = X };
4    using type = aint<X>;
5  };
```

Then we can use `aint<4>::value` to represent the value, and `aint<4>::type` to represent its type. Similarly,

```
1  struct anull {}; // anull indicates the end
2
3  template <bool> struct abool;
4
5  template <>
6  struct abool<true> {
7    enum { value = true };
8    using type = abool<true>;
9  };
10
11 template <>
12 struct abool<false> {
13   enum { value = false };
14   using type = abool<false>;
15 };
```

```
16
17 using atrue  = abool<true>;
18 using afalse = abool<false>;
```

Then whenever we want to pass a non-type, say `4`, we can use `aint<4>` and inside use `aint<4>::value`. As an example, we modify `max_if_f<X, Y, F>`,

```
1  template <typename X, typename Y, template <typename> class F>
2  struct max_if_f {
3    enum { value = F<X>::value < F<Y>::value ? Y::value : X::value };
4  };
5
6  template <typename X>
7  struct square {
8    enum { value = X::value * X::value };
9  };
10
11 template <typename X, typename Y>
12 using max_if_square = max_if_f<X, Y, square>;
13
14 std::cout << max_if_square<aint<-5>, aint<1>>::value << std::endl; // prints -5
```

Well, beautiful !

2.4 Anything is a function

In functional programming, the usage of `aint<5>::type` is not acceptable. Therefore we encapsulate some functions for them,

```
1  template <typename T1, typename T2>
2  struct is_eq {
3    enum { value = false };
4    using type = afalse;
5  };
6
7  template <typename T>
8  struct is_eq<T, T> {
9    enum { value = true };
10   using type = atrue;
11 };
12
13 // __value(T)
14 template <typename T>
15 struct the_value {
16   enum { value = 0 };
17 };
18
19 template <int X>
```

```
struct the_value<aint<X>> {
  enum { value = X };
};

template <bool B>
struct the_value<abool<B>> {
  enum { value = B };
};

template <>
struct the_value<anull> {
  enum { value = -1 };
};

// do not even try `t::value` and maybe, `((t)::value)`! you will regret it, trust me! :-). want to know the reason? use them and preprocess the file to see the preprocessed result using opetion -E in g++/clang. :-)
#define __value(t)    the_value<t>::value

#define __int(x)      typename aint<(x)>::type
#define __bool(v)     typename abool<(v)>::type
#define __true()      typename atrue::type
#define __false()     typename afalse::type

#define __is_eq(x, y) typename is_eq<x, y>::type
```

Then, we can use everything in a function,

```
std::cout << __value(__is_eq(__int(4), __int(4))) << std::endl;     // 1
std::cout << __value(__is_eq(__int(4), __bool(true))) << std::endl; // 0
std::cout << __value(__is_eq(__true(), __true())) << std::endl;     // 1
std::cout << __value(__is_eq(__true(), __false())) << std::endl;    // 0
std::cout << __value(__is_eq(__false(), __false())) << std::endl;   // 1
```

They are all computed in compile time. However, we could do more. Let's define some operations on `aint` and `abool`.

```
// __add(x, y)
template <typename X, typename Y> struct f_add;

template <int X, int Y>
struct f_add<aint<X>, aint<Y>> {
  using type = aint<X+Y>;
};

#define __add(x, y) typename f_add<x, y>::type

// __and(x, y)
```

```
template <typename X, typename Y> struct f_and;

template <bool X, bool Y>
struct f_and<abool<X>, abool<Y>> {
  using type = abool<X && Y>;
};

#define __and(x, y) typename f_and<x, y>::type

std::cout << __value(__add(__int(1), __int(2))) << std::endl;  // prints 3
std::cout << __value(__and(__true(), __false())) << std::endl; // prints false
```

Moreover, `__sub(x, y)` / `__mul(x, y)` / `__div(x, y)` / `__mod(x, y)` / `__or(x, y)` / `__not(x, y)` /... can be defined using the similar way.

2.5 Condition - Pattern Matching

There are two ways to do pattern matching: (1) template specialization (we've introduced above), and (2) function overloading.

- **template specialization**

  Using knowledge we've known above, we can define a `__if(c, t, f)` statement that receives a condition type `c` and then determine to `t` when `__true()`, and `f` when `__false()`.

```
template <typename C, typename T, typename F> struct s_if;

template <typename T, typename F>
struct s_if<atrue, T, F> {
  using type = T;
};

template <typename T, typename F>
struct s_if<afalse, T, F> {
  using type = F;
};

#define __if(c, t, f) typename s_if<c, t, f>::type
```

  Then, have a try! We define a operator `larger_type<T1, T2>` to choose a larger type,

```
template <typename T1, typename T2>
struct larger_type {
  using type = __if(__bool(sizeof(T1) > sizeof(T2)), T1, T2);
};

struct LargerOne {
```

```
  static const char *s;
  char paddig[2];
};

struct SmallerOne {
  static const char *s;
  char paddig;
};

const char *LargerOne::s = "larger_one";
const char *SmallerOne::s = "smaller_one";

// have a try
std::cout << larger_type<LargerOne, SmallerOne>::type::s << std::endl; //
wow! prints "larger_one" :-)
```

- **function overloading**

  One of the powerful function in programming language is the function overloading, via which, the compiler will help us to choose the suitable function according to our data type. Hence, we do not have to differentiate them by their names. Therefore, we can use it to do something.

  Let's implement an operator `__is_convertible(T, U)` checking whether a type `T` can be converted to type `U` by compiler.

```
template <typename D, typename B>
struct is_convertible {
private:
  using yes = char;
  using no  = int;

  static yes test(B);   // if D is a B, then this will be invoked
  static no  test(...); // we don't care about the parameters
  static D   a_D();

public:
  using type = abool<sizeof(test(a_D())) == sizeof(yes)>; // we use a_D()
instead of D() to avoid the overhead of constructing a new object
};

#define __is_convertible(D, B) typename is_convertible<D, B>::type
```

  Code above uses the overloaded function `test` to check the convertibility. One more step, we can implements `__is_a(D, B)` to check if `D` is a subtype of `B`.

```
#define __is_a(D, B) __and(__is_convertible(const D *, const B *), \
                           __and(__not(__is_eq(const B*, const void*)), \
                                 __not(__is_eq(const D, const B))))
```

Check it!

```
std::cout << __value(__is_a(int, char)) << std::endl;     // false
std::cout << __value(__is_a(char, int)) << std::endl;     // false
std::cout << __value(__is_a(Derived, Base)) << std::endl; // true
std::cout << __value(__is_a(Base, Derived)) << std::endl; // false
```

2.6 Loop - Recursion

As the condition statement relies on pattern matching, the loop statement relies on recursion. C++ provides variadic templates, which makes the loop available.

Let's firstly define an operator `reduce` which is of much power in functional programming.

```
// __reduce(f, x, ...), x is the initial value
template <template <typename, typename> class F, typename X, typename ...N>
struct f_reduce;

// assume N is the first, <4, 3, 2, 1, ...>, then N is 4, the last to be dealed with
template <template <typename, typename> class F, typename X, typename N, typename ...R>
struct f_reduce<F, X, N, R...> {
  using type = typename f_reduce<F, typename f_reduce<F, X, R...>::type, N>::type;
};

template <template <typename, typename> class F, typename X, typename N>
struct f_reduce<F, X, N> {
  using type = typename F<X, N>::type;
};

#define __reduce(f, x, ...) typename f_reduce<f, x, __VA_ARGS__>::type
```

Fine now, how to implement a `sum` so that `sum` could receive variadic ?

```
#define __sum(...) __reduce(f_add, __VA_ARGS__)
```

Wow, so easy! Let's test it!

```
std::cout << __value(__sum(__int(1), __int(2))) << std::endl;            // 3
std::cout << __value(__sum(__int(1), __int(2), __int(3))) << std::endl;  // 6
std::cout << __value(__sum(__int(1), __int(2), __int(-4))) << std::endl; // -1
std::cout << __value(__sum(__int(1))) << std::endl;                      //
compile error
```

Our `__sum()` works fine for parameters >= 2, but when it comes to 1 parameter, a compile error happened! Why? Recall that reduce works rely on a function `F` who takes as input 2 very parameters! Oops! The requirement of `__reduce` is that the number of parameters (except the function `F`) is 2 (with one initial value and an array with least length of 1). Let's modify the reduce to make the reduce operation more flexible[2]! Add a specialization, and modify the macro,

```
template <template <typename, typename> class F, typename X>
struct f_reduce<F, X> {
  using type = X;
};

#define __sum(...) __reduce(f_add, __VA_ARGS__)
```

Okay, we get it using template partial specialization! The `__sum(...)` works! Similarly, we can define `__max(…)` / `__min(…)/…`, a lot of them.

> Once you want to make reduce match the original semantics, i.e. takes at least 2 input values, you can define `__sum(…)` like,
>
> ```
> // __sum(...)
> template <typename ...N>
> struct f_sum {
>   using type = typename f_reduce<f_add, N...>::type;
> };
>
> template <typename N>
> struct f_sum<N> {
>   using type = N;
> };
>
> #define __sum(...) typename f_sum<__VA_ARGS__>::type
> ```

2.7 Immutability

Variables we use during compilation are all immutable. Can we write such code ?

```
using A = int;
A = char;      // illegal
```

Absolutely no! All variables are bounded to their initial value, and can never be modified. In other words, they are all const.

The immutability in programming provides many advantages.

2.8 Laziness

The compiler is lazy. Observe the following codes,

```
using Zero = __int(0);
```

If we bound `__int(0)` to Zero, the value `Zero::value` won't be calculated, even be generated until we explicitly use it. In other words, if we does not use `Zero::value` in our code, the compiler won't calculate it. The laziness of the compiler improves the performance of the compiler, and also reduce the space needed.

Moreover, if we define a function for a class(or struct, or union), but never invoke it, the compiler won't generate it!

2.9 Duck type

We mentioned previously that, C++ template meta programming leverages *duck type*. We say that because we only need to provide concrete types that have some functions.

And, when you treat C++ template meta programming as a separate programming language, it is more like a interpretive language with strong type checking, within (1) type, (2) non-type(int, char, bool, pointers, etc..) , and (3) template. When you pass a non-type (e.g. `1`) to a parameter who needs a type (e.g. `typename T`), the compiler complains errors.

2.10 TypeList

In programming, list is a fundamental element to most data types. And here we can implement a `__typelist(…)` that can contain a list of types in compile time.

Let's first consider the question: what is a list?

In function programming, a list is a recursive data type,

```
[1, [2, [3, [4, []]]]]
```

Ok, have this knowledge is enough for us. Because we have introduced the `recursion` previously.

Firstly, let's define a easy data type called `apair<F, S>` of whom the first element is `F`, and the second `S`.

```cpp
// __pair(F, S)
template <typename F, typename S>
struct apair {
  using first  = F;
  using second = S;
};

#define __pair(F, S) apair<F, S>
```

Using `apair<F, S>`, we can define a recursive operator `typelist<…>` to create a typelist. Watch out here, we will clearly claim that, the `__typelist(…)` we'll define is merely a *meta function*, not a type, with no differences with other meta functions like `__value()`, but with difference with `aint`. `__typelist` will accept a bunch of types and returns a `apair<F, S>`.

```cpp
// __empty()
struct aempty {};
#define __empty() aempty

// __typelist(...)
template <typename... T>
struct typelist {
  using type = aempty;
};

template <typename H, typename... T>
struct typelist<H, T...> {
  using type = apair<H, typename typelist<T...>::type>;
};

template <typename H>
struct typelist<H> {
  using type = apair<H, aempty>;
};

#define __typelist(...) typename typelist<__VA_ARGS__>::type
```

The `aempty` above is a empty type means containing nothing. It is of great importance.

With no operations on the list, we can do nothing. Therefore, let's define some operations on it! The common ones are length, get, set, insert and remove.

We've claimed that, recursion is of great importance. Algorithms following are all in recursion manner apparently from the definition of typelist. Let's show a relatively complex one, and the rest ones you can define them by yourself!

```cpp
// __tl_insert
template <typename TL, int N, typename X>
struct tl_insert {
```

```
  using type = TL;
};

template <typename F, typename S, int N, typename X>
struct tl_insert<apair<F, S>, N, X> {
  using type = apair<F, typename tl_insert<S, N-1, X>::type>;
};

template <typename F, typename S, typename X>
struct tl_insert<apair<F, S>, 0, X> {
  using type = apair<X, apair<F, S>>;
};

template <int N, typename X>
struct tl_insert<aempty, N, X> {
  using type = apair<X, aempty>;
};

#define __tl_insert(TL, n, S) typename tl_insert<TL, (n), S>::type
#define __tl_append(TL, S)    __tl_insert(TL, __value(__tl_length(TL)), S)
#define __tl_prepend(TL, S)   __tl_insert(TL, 0, S)
```

Now have a try!

```
using L = __typelist(int, float, void *, Base, Derived, const LargerOne &);
ASSERT_EQ(1, __value(__is_eq(aint<6>, __tl_length(L))));
ASSERT_EQ(1, __value(__is_eq(int, __tl_get(L, 0))));
ASSERT_EQ(1, __value(__is_eq(float, __tl_get(L, 1))));
ASSERT_EQ(1, __value(__is_eq(void *, __tl_get(L, 2))));
ASSERT_EQ(1, __value(__is_eq(Base, __tl_get(L, 3))));
ASSERT_EQ(1, __value(__is_eq(Derived, __tl_get(L, 4))));
ASSERT_EQ(1, __value(__is_eq(const LargerOne &, __tl_get(L, 5))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(L, 6))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(L, 7))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(int, 7))));

using LI1 = __tl_insert(L, 3, double);
ASSERT_EQ(1, __value(__is_eq(aint<7>, __tl_length(LI1))));
ASSERT_EQ(1, __value(__is_eq(int, __tl_get(LI1, 0))));
ASSERT_EQ(1, __value(__is_eq(float, __tl_get(LI1, 1))));
ASSERT_EQ(1, __value(__is_eq(void *, __tl_get(LI1, 2))));
ASSERT_EQ(1, __value(__is_eq(double, __tl_get(LI1, 3))));
ASSERT_EQ(1, __value(__is_eq(Base, __tl_get(LI1, 4))));
ASSERT_EQ(1, __value(__is_eq(Derived, __tl_get(LI1, 5))));
ASSERT_EQ(1, __value(__is_eq(const LargerOne &, __tl_get(LI1,6))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(LI1, 7))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(LI1, 8))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(int, 7))));

```

```
using LI2 = __tl_append(L, double);
ASSERT_EQ(1, __value(__is_eq(aint<7>, __tl_length(LI2))));
ASSERT_EQ(1, __value(__is_eq(int, __tl_get(LI2, 0))));
ASSERT_EQ(1, __value(__is_eq(float, __tl_get(LI2, 1))));
ASSERT_EQ(1, __value(__is_eq(void *, __tl_get(LI2, 2))));
ASSERT_EQ(1, __value(__is_eq(Base, __tl_get(LI2, 3))));
ASSERT_EQ(1, __value(__is_eq(Derived, __tl_get(LI2, 4))));
ASSERT_EQ(1, __value(__is_eq(const LargerOne &, __tl_get(LI2,5))));
ASSERT_EQ(1, __value(__is_eq(double, __tl_get(LI2,6))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(LI2, 7))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(LI2, 8))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(int, 7))));

using LI3 = __tl_prepend(L, double);
ASSERT_EQ(1, __value(__is_eq(aint<7>, __tl_length(LI3))));
ASSERT_EQ(1, __value(__is_eq(double, __tl_get(LI3, 0))));
ASSERT_EQ(1, __value(__is_eq(int, __tl_get(LI3, 1))));
ASSERT_EQ(1, __value(__is_eq(float, __tl_get(LI3, 2))));
ASSERT_EQ(1, __value(__is_eq(void *, __tl_get(LI3, 3))));
ASSERT_EQ(1, __value(__is_eq(Base, __tl_get(LI3, 4))));
ASSERT_EQ(1, __value(__is_eq(Derived, __tl_get(LI3, 5))));
ASSERT_EQ(1, __value(__is_eq(const LargerOne &, __tl_get(LI3,6))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(LI3, 7))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(LI3, 8))));
ASSERT_EQ(1, __value(__is_eq(anull, __tl_get(int, 7))));
```

Wow! Worked!

2.10 2.5-phase C++

As we mentioned previously, we can treat C++ as a 2.5-phase programming languages after template is introduced.

1. *0.5* the first half phase is the macro. Macro is a powerful tools (but not turing-complete because it just replaces text) with the help of the preprocessor.
2. *1* the next full phase is the template meta programming. In this phase, the compiler acts as an interpreter. It will lazily generates the result of our meta functions for us.
3. *1* the last full phase is the run-time C++ as we already familiar with.

3 Mysterious Type Traits

3.1 type traits support

Now let's open this. Right here, you will see a great number of compile-time functions, such as `std::enable_if`. And open one of them, you will see the **Possible implementation**, and now, as a programmer of template meta programming, how easy they are! And you have the ability to implement them. Such as,

```cpp
template <typename B, typename T = void>
struct enable_if;

template <typename T>
struct enable_if<atrue, T> {
  using type = T;
};

template <typename T>
struct enable_if<afalse, T> {};
```

3.2 iterator traits

Observe the following code,

```cpp
template <typename Iter>
?? val(Iter it) {
  return *it;
}
```

We put `??` where the return type should be placed. The function `val` we defined would like to get all the inner value of an iterator, then how to handle it? Recall the `::type`? Yes, we can use it!

```cpp
template <typename Iter>
typename Iter::type val(Iter it) {
  return *it;
}
```

Okay, as long as the type `Iter` has a field type then we can get it. Therefore, we need to write an interface for all iterators who really want to use the function above.

```cpp
template <typename T>
struct IteratorInterface {
  using type = T;
};
```

Fine by now, all iterators can use it,

```cpp
class MyIntIterator : public IteratorInterface<int> {
  // blabla
};

// use it
val<MyIntIterator>(my_int_iterator); // or just val(my_iterator)
```

But you may notice, the raw pointer, e.g. `char*`, is also a type of iterator, and we also want to use the `val` function. Question here is that, a raw pointer e.g. `char*` does not have an inner type called `type`. How to handle it?

Okay, you may find the way to it. We can write a meta function who returns (or extracts, or *traits*) all the inner type defined, and for the raw type, we specialize it!

```
template <typename Iter>
struct iterator_traits {
  using type = Iter::type;
};

template <typename T>
struct iterator_traits<T *> {
  using type = T;
};

template <typename T>
struct iterator_traits<const T *> {
  using type = T;
};
```

Then the `val` get turned to,

```
template <typename Iter>
typename iterator_traits<Iter>::type val(Iter it) {
  return *it;
}
```

And when comes across a raw type, e.g. `char *x = &p; char v = val(x);` the compiler will find the specialized `iterator_traits<T *>` or `iterator_traits<const T *>` and returns `T`. Perfect :-) !

Yes, as you see, this is called *iterator traits* ~

In STL, there are more inner types to be designated, such as `value_type` (`tppe` in our case), `difference_type`, `pointer`, `reference` and `iterator_category`. See here, they are easy to you now! :-)

4 End

Thanks for [1] greatly. It helps organize my knowledge on template of C++, so that I can treat it from a higher lever. And this, helps me greatly to learn more on the design details of the traits skills I've already known but not that know before.

Reference

1. Series: C++11 模板元编程，作者：MagicBowen，Email：e.bowen.wang@icloud.com

1. A constant expression that designates the address of a complete object with static storage duration and external or internal linkage or a function with external or internal linkage, including function templates and function *template-ids* but excluding non-static class members, expressed (ignoring parentheses) as `&` *id-expression*, where the *id-expression* is the name of an object or function, except that the `&` may be omitted if the name refers to a function or array shall be omitted if the corresponding *template-parameter* is a reference.↩

2. In some library and language (Javascript for example) implementations, their reduce will return the initial value when they comes across 1 parameters. So do we.↩