Lecture 8 Templates and Functions

#CS106L

•Implicit Interface

•Concepts

•Functions and Lambdas

•Algorithm Library

•Adaptors

•Ranges

Q1:how to lift this function into its most generic form?

```
tuple<bool, int, int> mismatch(const vector<int>& vec1,const vector<int>& vec2)

size_t i = 0;

while (i < vec1.size() && vec1[i] == vec2[i])  ++i;

if (i == vec1.size()) return {true, 0, 0};

else                  return {false, vec1[i], vec2[i]};
```

Stanford version:

```
template <typename InputIt1 typename InputIt2>

pair<int, int> mismatch(InputIt1 first1, InputIt1 last1,

InputIt2 first2)

while (first1 != last1 && *first1 == *first2){

++first1; ++first2;
}

return {first1, first2};

}
```

another version:

```
template <typename T, typename U>

tuple<bool, typename T::value_type, typename U::value_type> mismatch(const T&
container1, const U& container2) {

    auto i = begin(container1);auto j = begin(container2);

    while (i != end(container1) && j != end(container2) && *i == *j) {

        ++i; ++j;

    }

    if (i == end(container1)) return { true, typename T::value_type{}, typename
U::value_type{} };

    else return { false, *i, *j };

}
```

Q2:why

```
        typename T::value_type, typename U::value_type
```

can't rewritten as

```
        typename value_type,typename value_type
```

But if the two containers contain different types, the comparison is impossible?

If you write `typename value_type, typename value_type` the compiler would not know to which type `value_type` belongs to.

`typename T::value_type` and `typename U::value_type` specifies that `value_type` is a nested type of container type `T` and `U` respectively. This way the compiler knows that `value_type` is a member of the container class `T` and `U`, and it can deduce the type correctly.

On the other hand, if you wrote `typename value_type, typename value_type`, the compiler would not know to which class `value_type` belongs to. It would assume that `value_type` is a type defined in the current scope, which would cause a compile error if it is not defined.

Additionally, `typename T::value_type` and `typename U::value_type` allows the function to return the same type of data that the container holds, because it uses the `value_type` typedef of the container.

In conclusion, it is important to specify to which class or struct the member type belongs to, using `T::value_type` or `U::value_type` to avoid confusion and to make sure the correct type is deduced by the compiler.

You're correct that if the two containers contain different types, the comparison is impossible, the function would not be able to work as intended. The problem with the approach of `typename value_type,typename value_type` is that it will not indicate to the compiler that the types of the two containers must be the same.

It is important to note that the function I provided is just an example and it doesn't account for the scenario where the types of the two containers are different, in this case, it will cause a compile error.

If you want to make sure that the types of the two containers are the same, you can add a template constraint to the function, using `std::enable_if` for example, to make sure that both container types are the same.

```
template <typename T, typename U>

typename enable_if<is_same<typename T::value_type, typename
U::value_type>::value,tuple<bool, typename T::value_type, typename
U::value_type>>::type mismatch(const T& container1, const U& container2) {

    // ...

}
```

This will make sure that the function is only instantiated when the types of the two containers are the same.

In conclusion, the `typename T::value_type, typename U::value_type` is more accurate because it specifies that `value_type` is a nested type of container type `T` and `U` respectively, and it allows the function to return the same type of data that container holds. But it doesn't account for the scenario where the two containers contain different types, in that case, you should add a template constraint to ensure that both container types are the same.

Q3:implicit interfaces?

An implicit interface is a set of functions or member functions that a type must provide in order to be used in a certain way.

For example, in C++, a type that can be used in a range-based for loop must provide the `begin()` and `end()` member functions, and a type that can be used in an `std::sort()` function must provide the `<` operator.

These interfaces are not explicit, they are not defined by a specific interface or class, but by the behavior of the standard library functions and by the language itself.

For example, the `std::sort()` function does not require that the type passed to it explicitly inherits from a specific class or implements a specific interface, it only requires that the type has a `<` operator, this is considered an implicit interface.

Similarly, the range-based for loop requires that the type passed to it has `begin()` and `end()` member functions, again, this is considered an implicit interface.

C++20 concepts are a way to make these implicit interfaces explicit, they allow you to define the requirements that a type must satisfy in order to be used in a certain way, and by doing so, they improve the readability, maintainability and expressiveness of the code.

Q4:concepts?

Concepts are a new feature introduced in C++20, they are a way to express constraints on template arguments. A concept defines a set of requirements that a template argument must satisfy in order for the template to be instantiated.

The syntax for a concept is similar to a function template, it starts with the keyword `concept` followed by the name of the concept and a set of template parameters. The body of the concept is used to specify the requirements that the template argument must satisfy.

For example, the following is a concept for a type that supports the `==` operator:

```
template<typename T>

concept EqualityComparable = requires(T a, T b) {

    { a == b } -> bool;

    { a != b } -> bool;

};
```

This concept can be used to constrain a template function to only work with types that support the `==` operator:

```
template <EqualityComparable T>

void compare(const T& a, const T& b) {

    if (a == b) {

        cout << "a and b are equal" << endl;

    } else {

        cout << "a and b are not equal" << endl;

    }

}
```

Concepts can also be combined with logical operators (`&&`, `||` and `!`) to create more complex constraints. For example, you can use the `&&` operator to specify that a template argument must satisfy multiple concepts, such as:

```
template <typename T>

concept C1 = ...;

template <typename T>
```

```
concept C2 = ...;

template <typename T>

concept C3 = C1<T> && C2<T>;
```

This defines a new concept `C3` that requires that a template argument must satisfy both `C1` and `C2`.

You can also use the `||` operator to specify that a template argument must satisfy one of multiple concepts, such as:

```
template <typename T>

concept C4 = C1<T> || C2<T>;
```

This defines a new concept `C4` that requires that a template argument must satisfy either `C1` or `C2`.

Additionally, you can use the `!` operator to specify that a template argument must not satisfy a certain concept, such as:

```
template <typename T>

concept C5 = !C1<T>;
```

This defines a new concept `C5` that requires that a template argument must not satisfy the concept `C1` .

Concepts can be used in various parts of the code, for example, to constrain template functions, classes, and member functions. They can also be used to constrain the type of a variable, for example:

```
EqualityComparable auto x = ...;
```

This declares a variable `x` of a type that satisfies the concept `EqualityComparable`.

Concepts improve the readability, maintainability and expressiveness of the code by making the requirements of the template arguments explicit and by providing better compile-time error messages. They also make the code more efficient by allowing the compiler to perform more aggressive optimization and by reducing the need for type traits and SFINAE.

Q5:lambda function?

In C++, a lambda function (also known as a lambda expression or simply a lambda) is a way to create anonymous (unnamed) functions. A lambda function is similar to a regular function, but it can be defined and executed on the fly, without having to give it a name.

A lambda function is defined using the following syntax:

```
        [ auto list ] (parameter list) -> return_type { function_body }
```

The capture list is used to capture variables from the enclosing scope(capture everything by value or reference), the parameter list is used to define the parameters of the function(only in lambda function can we use auto !), the return type is used to specify the type of the value returned by the function, and the function body is used to define the behavior of the function.

For example, the following lambda function takes two integers as parameters and returns their sum:

```
        [](int a, int b) { return a + b; }
```

Lambda functions can be used in many ways, for example, they can be stored in a variable, passed as an argument to a function, or returned as a value from a function. They can be used as a function pointer, as a functor, and as a std::function.

Lambda functions are very useful in C++, they provide a more concise and expressive way to define small, anonymous functions, they can be used to define callbacks and to pass functions as arguments to other functions. They are widely used in C++11/14/17/20 in STL and other libraries, for example:

1. Using lambda functions with `std::sort()`:

```
        std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};

        std::sort(v.begin(), v.end(), [](int a, int b) { return a < b; });
```

2. Using lambda functions with `std::for_each()`:

```
        std::vector<int> v = {1, 2, 3, 4, 5};

        std::for_each(v.begin(), v.end(), [](int& x) { x *= x; });
```

3. Using lambda functions with `std::thread`:

```
        std::thread t([]() { std::cout << "Hello from thread\n"; });

        t.join();
```

4. Using lambda functions with `std::bind()`:

```
        std::function<int(int, int)> add = [](int a, int b) { return a + b; };

        auto add5 = std::bind(add, 5, std::placeholders::_1);

        std::cout << add5(3) << std::endl; // prints 8
```

5. Using lambda functions with `std::mem_fn()`:

```
class A {

public:

    void f() { std::cout << "Hello\n"; }

};

A a;

auto f = std::mem_fn(&A::f);

f(a);
```