Template

Template function

- How to define a template function? Are the keywords `typename` and `class` different in template function declaration? Implement a template function `max(a, b)`, which can return the maximum value of `a` and `b`, whatever the type of `a` and `b`.
- Compared the usage of `std::make_shared` and `std::max`, show the difference of them. You **don't need** to know the meaning of `std::make_shared`, just consider the **calling form** of it.
- If we use `std::max(1.0, 2)`, it will tell us a compile error, what does the error mean? How to fix it?
- What can `...` be in `template<...>`? Give an example of it.
- When we define a template function without using, does the compiler exactly generate a function? (Hint: you can use Compiler Explorer) What it will be when we use it? Is the template function a true function?
- What is the output of following program?

```
#include <iostream>
template<typename T>
void f(T) { std::cout << "Template Function!\n"; }
void f(int) { std::cout << "Int Function!\n"; }
int main() { f(12); }
```

- Implement a function `for_each()`. We can give it a container and a function, `for_each()` will execute the function with the parameter in container. Here is an example of `for_each()`.

```
template<typename T>
void print(const T& v) {
    std::cout << v << " ";
}
template<typename T>
void println(const T& v) {
    std::cout << v << std::endl;
}
int main() {
    std::vector<int> vec{1, 2, 3, 4, 5};
    std::deque<std::string> deq{"1234", "abcd", "ddddd", "dptee"};
    for_each(vec, print<int>); std::cout << std::endl;
    for_each(deq, println<std::string>);
    return 0;
}
```

Outputs are:

```
1 2 3 4 5
1234
abcd
ddddd
dptee
```

- Pipe operator is an important operator in Linux. In this problem, you should implement a simple pipe operator in C++. In `a | b`, The return value or value of `a` is the argument of function `b`. And this operator can be repeated. Here is an example of it.

```cpp
int main() {
    int value = 10;
    auto func1 = [] (int x) { return x * x; };
    auto func2 = [] (int x) { return x + 998; };
    auto x = value | func1 | func2;
    std::cout << x << std::endl;
}
```

Output is `1098`. You can simply assume that functions in pipe operator are lambda expressions or `std::function` and have only one parameter. (Hint: consider to overload operator `|`).

Template class

- Explain the definition of template class. Give an example of template class in STL.
- Compared with template function, is a template class a true class?
- What can a `...` be in template class `template<...> class a {};`? Comparing with template function.
- We can use template function without declaring its template arguments, e.g. `std::max(1, 2)`, can we do the same things in template class? Consider the different standard of C++. Give an example whether we can or not.
- In many situations, we desperate the definition and implementation in different file, which the head file (`.h` file) contains definitions, and source file (`.cpp` file) contains implementations. Can we do the same things for template classes and template functions? Give an example and guess the reason. (Hint: consider the `vector` in C++ STL and the previous questions)
- If we want the template class is different from general template class **for some special template arguments**, what should we do? For example, we have defined `template<typename T>struct A{ ... };`, now we find the implementation is different for `T=bool`, what should we do? (You can check the definition of `std::allocator<void>`). Complete the class implementation of `is_bool` for the following usage. (Hint: you can use `static` variable, and `static` variable can be initialized in class with `const`/`constexpr`)

```cpp
int main() {
    std::cout << std::boolalpha;
    std::cout << is_bool<int>::value << std::endl;  // print 'false'
    std::cout << is_bool<bool>::value << std::endl; // print 'true'
    return 0;
}
```

- Here is a template class `template<typename T> strut A{ ... };`, and **for some types with common characteristics**, it has different implementation, such as pointer, array, const

type, ... What should we do? Complete the class implementation of `is_pointer` for the following usage.

```cpp
int main() {
    std::cout << std::boolalpha;
    std::cout << is_pointer<int*>::value << std::endl;  // print 'true'
    std::cout << is_pointer<char>::value << std::endl;  // print 'false'
    return 0;
}
```

- Use the **recursive template class** to calculate the Fibonacci number. (Hint: if there is a class `template<int N>struct A{};`, you can use `A<N-1>` and `A<N-2>` in `A<N>`) Here is an example usage of class.

```cpp
int main() {
    std::cout << Fib<5>::value << std::endl;
    std::cout << Fib<10>::value << std::endl;
    std::cout << Fib<15>::value << std::endl;
    return 0;
}
```

If you **have no idea**, you can put it down, and complete it **after class**.

- Here is a `vector` derived from `std::vector`, it is added a new assignment operator. Now you should implement a template class `vector_expr`, and overload `+-*/` for `vector`, so that we can directly use arithmetic operators between `vector`.

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <ranges>
#include <vector>

template <typename T>
class vector : public std::vector<T> {
public:
    using std::vector<T>::vector;
    using std::vector<T>::size;
    using std::vector<T>::operator[];
    template <typename E>
    vector<T>& operator=(const E& e) {
        const auto count = std::min(size(), e.size());
        this->resize(count);
        for (std::size_t idx{0}; idx < count; ++idx) {
            this->operator[](idx) = e[idx];
        }
        return *this;
    }
};

/*
// Implement the template class and operator overload.
// You can add other functions if needed.
template<...>
struct vector_expr {
```

```
};

// operator+
// operator-
// operator*
// operator/
*/


/*
!!! You should Not modify the following codes !!!
*/

void print(vector<double> vec) {
    for (auto&& v: vec) {
        std::cout << v << " ";
    }
    std::cout << std::endl;
}
int main() {
    const vector<double> a { 1.2764, 1.3536, 1.2806, 1.9124, 1.8871, 1.7455 };
    const vector<double> b { 2.1258, 2.9679, 2.7635, 2.3796, 2.4820, 2.4195 };
    const vector<double> c { 3.9064, 3.7327, 3.4760, 3.5705, 3.8394, 3.8993 };
    const vector<double> d { 4.7337, 4.5371, 4.5517, 4.2110, 4.6760, 4.3139 };
    const vector<double> e { 5.2126, 5.1452, 5.8678, 5.1879, 5.8816, 5.6282 };

    {
        std::cout << "Standard outputs:\n";
        vector<double> result(6);
        for (std::size_t idx = 0; idx < 6; idx++) {
            result[idx] = a[idx] - b[idx] * c[idx] / d[idx] + e[idx];
        }
        print(result);
    }
    {
        std::cout << "Your outputs:\n";
        vector<double> result(6);
        result = a - b * c / d + e; // use the expression template to calculate.
        print(result);
    }
    return 0;
}
```

You can look up these references: C++语言的表达式模板：表达式模板的入门性介绍, Wiki Pedia - Expression templates.

If you **have no idea** about this, you could put it down, and complete it **after class**.