现代C++题目（答案与解析）

卢瑟帝国

2023 年 11 月 24 日

目录

暂时只有 13 道题目，并无特别难度，有疑问可看视频教程或答案解析。

1 实现管道运算符

日期：2023/7/21 出题人：mq白

给出以下代码，在不修改已给出代码的前提下使它满足**运行结果**。

```
int main(){
    std::vector v{1, 2, 3};
    std::function f {[](const int& i) {std::cout << i << ' '; } };
    auto f2 = [](int& i) {i *= i; };
    v | f2 | f;
}
```

要求运行结果

```
1 4 9
```

- 难度： ★★☆☆☆
  提示：`T& operator|(T& v, const T& f)` 。

1.1 答案

```
template<typename U, typename F>
    requires std::regular_invocable<F, U&>//可加可不加，不会就不加
std::vector<U>& operator|(std::vector<U>& v1, F f) {
    for (auto& i : v1) {
        f(i);
    }
    return v1;
}
```

不使用模板：

```
std::vector<int>& operator|(std::vector<int>& v1, const std::function<void(int&)>& f) {
    for (auto& i : v1) {
```

```
        f(i);
    }
    return v1;
}
```

不使用范围 for，使用 C++20 简写函数模板：

```
std::vector<int>& operator|(auto& v1, const auto& f) {
    std::ranges::for_each(v1, f);
    return v1;
}
```

各种其他答案的范式无非就是这些改来改去了，没必要再写。

1.2 解析

很明显我们需要重载管道运算符 |，根据我们的调用形式 v | f2 | f, 这种**链式**的调用，以及根据给出运行结果，我们可以知道，重载函数应当返回 v 的引用，并且 v 会被修改。

v | f2 调用 operator |，operator | 中使用 f2 遍历了 v 中的每一个元素，然后返回 v 的引用，再 | f。

形式和原理很简单，那么接下来就是实现；最简单的方式无非就是写一个模板

```
template<typename T, typename F>
T& operator|(T& v1, F f) {
    for (auto& i : v1) {
        f(i);
    }
    return v1;
}
```

当然了，这个模板还不够好，我们知道了第一个参数会是 vector，模板完全可以再准确一点：

```
template<typename U, typename F>
std::vector<U>& operator|(std::vector<U>& v1, F f)
```

考虑到 std::function 的复制开销也不小，第二个模板形参也可以加 const&。

范围 for，以及 requires 不再介绍。

2 实现自定义字面量 _f

日期：2023/7/22 出题人：mq白

给出以下代码，在不修改已给出代码的前提下使它满足**运行结果**。 6 为输入，决定 π 的小数点后的位数，可自行输入更大或更小数字。

```cpp
int main(){
    std::cout << "乐 :{} *\n"_f(5);
    std::cout << "乐 :{0} {0} *\n"_f(5);
    std::cout << "乐 :{:b} *\n"_f(0b01010101);
    std::cout << "{:*<10}"_f("卢瑟");
    std::cout << '\n';
    int n{};
    std::cin >> n;
    std::cout << "π：{:.{}f}\n"_f(std::numbers::pi_v<double>, n);
}
```

要求运行结果

```
乐 :5 *
乐 :5 5 *
乐 :1010101 *
卢瑟******
6
π：3.141593
```

- **难度：** ★ ★ ☆ ☆ ☆

  提示：C++11 用户定义字面量、C++20 format 库。

2.1 答案

```cpp
constexpr auto operator""_f(const char* fmt, size_t) {
    return [=]<typename... T>(T&&... Args) {
        return std::vformat(fmt, std::make_format_args(std::forward<T>(Args)...));
    };
}
```

2.2 解析

我们需要使用到 C++11 用户定义字面量，""_f 正是用户自定义字面量，但**字面量运算符**（用户定义字面量所调用的函数被称为字面量运算符）的形参列表有一些限制，我们需要的是（const char *,std::size_t）这样的形参列表，恰好这是允许的；字面量运算符的返回类型，我们需要自定义，这个类型需要在内部重载 () 运算符，以满足上述字面量像函数一样调用的要求。

我们一步一步来：

```
1  void operator""_test(const char* str, std::size_t){
2      std::cout << str << '\n';
3  }
4
5  "luse"_test; //调用了字面量运算符，打印 luse
6
7  std::size_t operator""_test(const char* , std::size_t len){
8      return len;
9  }
10
11 std::size_t len = "luse"_test; //调用了字面量运算符，返回 luse 的长度 4
```

上面这段代码的两个使用示例展示了我们用户定义字面量的基本使用，尤其注意第二段，返回值。如果要做到像 "xxx"_f(xxx) 这样调用，就得在返回类型上做点手脚。

```
1  struct X{
2      std::size_t operator()(std::size_t n)const{
3          return n;
4      }
5  };
6
7  X operator""_test(const char* , std::size_t){
8      return {};
9  }
10
11 std::cout<<"无意义"_test(1); //打印 1
```

以上这段简单的代码很好的完成了我们需要的调用形式，那么是时候完成题目要求的功能了。最简单的方式是直接使用 C++20 format 库进行格式化。

```cpp
namespace impl {
    struct Helper {
        const std::string_view s;
        Helper(const char* s, std::size_t len): s(s, len) {}
        template <typename... Args>
        std::string operator()(Args&&... args) const {
            return std::vformat(s,
            ↪  std::make_format_args(std::forward<Args>(args)...));
        }
    };
} // namespace impl
impl::Helper operator""_f(const char* s, std::size_t len) noexcept {
    return {s, len};
}
```

operator""_f 本身非常简单，只是用来把传入的参数（格式字符串）和长度，构造 mpl::Helper 对象再返回。Helper 类型使用了一个 string_veiw 作为数据成员，存储了格式字符串，以供后面格式化使用。

重点只在于 **operator**()。它是一个变参模板，用来接取我们传入的任意类型和个数的参数，然后返回格式化后的字符串。

这里用到的是 std::vformat 进行格式化，它的第一个参数是格式字符串，也就是我们要按照什么样的规则去格式化；第二个参数是要格式化的参数，但是我们没有办法直接进行形参包展开，它第二个参数的类型实际上是 std::format_args。我们必须使用 std::make_format_args 函数传入我们的参数，它会返回 std::format_args 类型，其实也就是相当于转换一下，合理。

不过显然标准答案不是这样的，还能简化，直接让 ""_f 返回一个 lambda 表达式即可。

3 实现 print 以及特化 std::formatter

日期：2023/7/24 出题人：mq白

实现一个 print，如果你做了上一个作业，我相信这很简单。要求调用形式为:

```
print(格式字符串，任意类型和个数的符合格式字符串要求的参数)
```

```
struct Frac {
    int a, b;
 };
```

给出自定义类型Frace，要求支持以下：

```
Frac f{ 1,10 };
print("{}", f);// 结果为1/10
```

要求运行结果

1/10

- **难度：** ★★★☆☆
  提示：std::formatter。

禁止面向结果编程，使用宏等等方式，本题主要考察和学习 format 库，记得测试至少三个不同编译器。

3.1 标准答案

```
template<>
struct std::formatter<Frac>:std::formatter<char>{
    auto format(const auto& frac, auto& ctx)const{//const修饰是必须的
        return std::format_to(ctx.out(), "{}/{}", frac.a, frac.b);
    }
};
void print(std::string_view fmt,auto&&...args){
```

```
8       std::cout << std::vformat(fmt,
        ↪  std::make_format_args(std::forward<decltype(args)>(args)...));
9   }
```

3.2 解析

实现一个 print 很简单，我们只要按第二题的思路来就行了，一个格式化字符串，用 std::string_view 做第一个形参，另外需要接受任意参数和个数的参数，使用形参包即可。

```
1   void print(std::string_view fmt,auto&&...args){
2       std::cout << std::vformat(fmt,
        ↪  std::make_format_args(std::forward<decltype(args)>(args)...));
3   }
```

由于使用了 C++20 简写函数模板，此处的完美转发就只能使用 decltype，显得有点诡异，其实很合理

展开的形式无非就是：

```
1   std::forward<decltype(args1)>(args1),
2   std::forward<decltype(args2)>(arsg2),
3   std::forward<decltype(args3)>(args3),...
```

这两个格式化函数没必要再介绍，和第第二题的作用一样，很简单的调库。

4 给定类模板修改，让其对每一个不同类型实例化有不同 ID

日期：2023/7/25 出题人：Maxy

```cpp
#include <iostream>
class ComponentBase{
protected:
    static inline size_t component_type_count = 0;
};
template<typename T>
class Component : public ComponentBase{
public:
    //todo...
    //使用任意方式更改当前模板类，使得对于任意类型X，若其继承自Component

    //则X::component_type_id()会得到一个独一无二的size_t类型的id（对于不同的X类型返回的值应不同）
    //要求：不能使用std::type_info（禁用typeid关键字），所有id从0开始连续。
};
class A : public Component<A>
{};
class B : public Component<B>
{};
class C : public Component<C>
{};
int main()
{
    std::cout << A::component_type_id() << std::endl;
    std::cout << B::component_type_id() << std::endl;
    std::cout << B::component_type_id() << std::endl;
    std::cout << A::component_type_id() << std::endl;
    std::cout << A::component_type_id() << std::endl;
    std::cout << C::component_type_id() << std::endl;
}
```

要求运行结果

```
0
1
1
0
0
2
```

- 难度： ★ ☆ ☆ ☆ ☆
  提示：初始化。

4.1 标准答案

```cpp
template<typename T>
class Component : public ComponentBase{
public:
    static size_t component_type_id(){
        static size_t ID = component_type_count++;
        return ID;
    }
};
```

4.2 解析

我们需要实现 Component 的静态成员函数 component_type_id。这是从给出代码得知的：

```cpp
class A : public Component<A>
{};
A::component_type_id()
```

C++ 的模板不是具体类型，实例化之后才是（即函数模板不是函数，类模板不是类），类模板的静态成员或静态成员函数也不属于模板，而是属于实例化后的具体类型我们可以用一段代码来展示结论：

```cpp
#include <iostream>

template<typename T>
```

```
4  struct Test{
5      inline static int n = 10;
6  };
7
8  int main(){
9      Test<int>::n = 1;
10     std::cout << Test<void>::n << '\n';//10
11     std::cout << Test<int>::n << '\n';//1
12 }
```

这段代码很轻易的就展示了静态数据成员属于模板实例化后的具体类型。`Test<void>::n` 和 `Test<int>::n` 不是相同的 n ，并且 `Test<void>` 和 `Test<int>` 也不是一种类型（静态成员函数同理）。

所以我们的解法利用的是：不同的类型实例化 Component 类模板，也是不同的静态成员函数，静态成员函数里面的静态局部变量也都是唯一的，并且在第一次调用的时候才会初始化，后面就不会。

5 实现 scope_guard 类型

日期：2023/7/29 出题人：Da'Inihlus

要求实现 **scope_guard** 类型（即支持传入任意可调用类型，析构的时候同时调用）。

```cpp
#include <cstdio>
#include <cassert>

#include <stdexcept>
#include <iostream>
#include <functional>

struct X {
    X() { puts("X()"); }
    X(const X&) { puts("X(const X&)"); }
    X(X&&) noexcept { puts("X(X&&)"); }
    ~X() { puts("~X()"); }
};

int main() {
    {
        // scope_guard的作用之一，是让各种C风格指针接口作为局部变量时也能得到RAII支持
        // 这也是本题的基础要求
        FILE * fp = nullptr;
        try{
            fp = fopen("test.txt","a");
            auto guard = scope_guard([&] {
                fclose(fp);
                fp = nullptr;
            });

            throw std::runtime_error{"Test"};
        } catch(std::exception & e){
            puts(e.what());
        }
        assert(fp == nullptr);
```

```
32      }
33      puts("----------");
34      {
35          // 附加要求1，支持函数对象调用
36          struct Test {
37              void operator()(X* x) {
38                  delete x;
39              }
40          } t;
41          auto x = new X{};
42          auto guard = scope_guard(t, x);
43      }
44      puts("----------");
45      {
46          // 附加要求2，支持成员函数和std::ref
47          auto x = new X{};
48          {
49              struct Test {
50                  void f(X*& px) {
51                      delete px;
52                      px = nullptr;
53                  }
54              } t;
55              auto guard = scope_guard{&Test::f, &t, std::ref(x)};
56          }
57          assert(x == nullptr);
58      }
59  }
```

要求运行结果

Test
———-
X()
X()
———-
X()
X()

- 难度： ★ ★ ★ ★ ☆
  提示：C++11 形参包，成员指针，完美转发，std::tuple，std::apply，C++17 类推导指引，std::invoke，std::function

5.1 标准答案

使用 std::function 并擦除类型

```
struct scope_guard {
    std::function<void()>f;
    template<typename Func, typename...Args> requires std::invocable<Func,
    ↪ std::unwrap_reference_t<Args>...>
    scope_guard(Func&& func, Args&&...args) :f{ [func = std::forward<Func>(func),
    ↪ ...args = std::forward<Args>(args)]() mutable {
            std::invoke(std::forward<std::decay_t<Func>>(func),
            ↪ std::unwrap_reference_t<Args>(std::forward<Args>(args))...);
        } }{}
    ~scope_guard() { f(); }
    scope_guard(const scope_guard&) = delete;
    scope_guard& operator=(const scope_guard&) = delete;
};
```

使用 std::tuple+std::apply

```
template<typename F, typename...Args>
    requires requires(F f, Args...args) { std::invoke(f, args...); }
struct scope_guard {
```

```
    F f;
    std::tuple<Args...>values;

    template<typename Fn, typename...Ts>
    scope_guard(Fn&& func, Ts&&...args) :f{ std::forward<Fn>(func) }, values{
    ↪  std::forward<Ts>(args)... } {}
    ~scope_guard() {
        std::apply(f, values);
    }
    scope_guard(const scope_guard&) = delete;
};

template<typename F, typename...Args>//推导指引非常重要
scope_guard(F&&, Args&&...) -> scope_guard<std::decay_t<F>, std::decay_t<Args>...>;
```

5.2 解析