

现代 C++ 题目

卢瑟帝国

2024 年 2 月 6 日

目录

1	实现管道运算符	2
2	实现自定义字面量 <code>_f</code>	2
3	实现 <code>print</code> 以及特化 <code>std::formatter</code>	3
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	4
5	实现 <code>scope_guard</code> 类型	5
6	解释 <code>std::atomic</code> 初始化	7
7	<code>throw new MyException</code>	8
8	定义 <code>array</code> 推导指引	9
9	名字查找的问题	10
10	遍历任意聚合类数据成员	11
11	<code>emplace_back()</code> 的问题	12
12	实现 <code>make_vector()</code>	12
13	关于 <code>return std::move(expr)</code>	14
14	以特殊方法修改命名空间中声明的对象	15
15	表达式模板	16
16	制造传递函数模板的宏	19

暂时只有 15 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。

```
1  int main(){
2      std::vector v{1, 2, 3};
3      std::function f [](const int& i) {std::cout << i << ' '; } };
4      auto f2 = [](int& i) {i *= i; };
5      v | f2 | f;
6  }
```

要求运行结果

1 4 9

• 难度：★★☆☆☆

提示：T& [operator](#)|(T& v, const T& f) 。

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。

```
1  int main(){
2      std::cout << " 乐 :{} *\n"_f(5);
3      std::cout << " 乐 :{0} {0} *\n"_f(5);
4      std::cout << " 乐 :{:b} *\n"_f(0b01010101);
5      std::cout << "{:.*<10}"_f(" 卢瑟");
6      std::cout << '\n';
7      int n{};
8      std::cin >> n;
```

```

9      std::cout << " : {:.{}f}\n"_f(std::numbers::pi_v<double>, n);
10 }

```

要求运行结果

```

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

```

- 难度: ★ ★ ☆ ☆ ☆

提示: C++11 用户定义字面量、C++20 format 库。

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:

```

1 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)

```

```

1 struct Frac {
2     int a, b;
3 };

```

给出自定义类型 Frac, 要求支持以下:

```

1 Frac f{ 1,10 };
2 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。

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

```

1  #include<iostream>
2  class ComponentBase{
3  protected:
4      static inline std::size_t component_type_count = 0;
5  };
6  template<typename T>
7  class Component : public ComponentBase{
8  public:
9      //todo...
10     //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
11
12     //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
13     //要求：不能使用 std::type_info（禁用 typeid 关键字），所有 id 从 0 开始连续。
14 };
15 class A : public Component<A>
16 {};
17 class B : public Component<B>
18 {};
19 class C : public Component<C>
20 {};
21 int main()
22 {
23     std::cout << A::component_type_id() << std::endl;
24     std::cout << B::component_type_id() << std::endl;
25     std::cout << B::component_type_id() << std::endl;
26     std::cout << A::component_type_id() << std::endl;
27     std::cout << A::component_type_id() << std::endl;
28     std::cout << C::component_type_id() << std::endl;

```

29 }

要求运行结果

```

0
1
1
0
0
2

```

- 难度：★ ☆ ☆ ☆ ☆

提示：初始化。

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）。

```

1  #include <cstdio>
2  #include <cassert>
3
4  #include <stdexcept>
5  #include <iostream>
6  #include <functional>
7
8  struct X {
9      X() { puts("X()"); }
10     X(const X&) { puts("X(const X&)"); }
11     X(X&&) noexcept { puts("X(X&&)"); }
12     ~X() { puts("~X()"); }
13 };
14
15 int main() {
16     {
17         // 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
18         // 这也是本题的基础要求

```

```
19     FILE * fp = nullptr;
20     try{
21         fp = fopen("test.txt", "a");
22         auto guard = scope_guard([&] {
23             fclose(fp);
24             fp = nullptr;
25         });
26
27         throw std::runtime_error{"Test"};
28     } catch(std::exception & e){
29         puts(e.what());
30     }
31     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

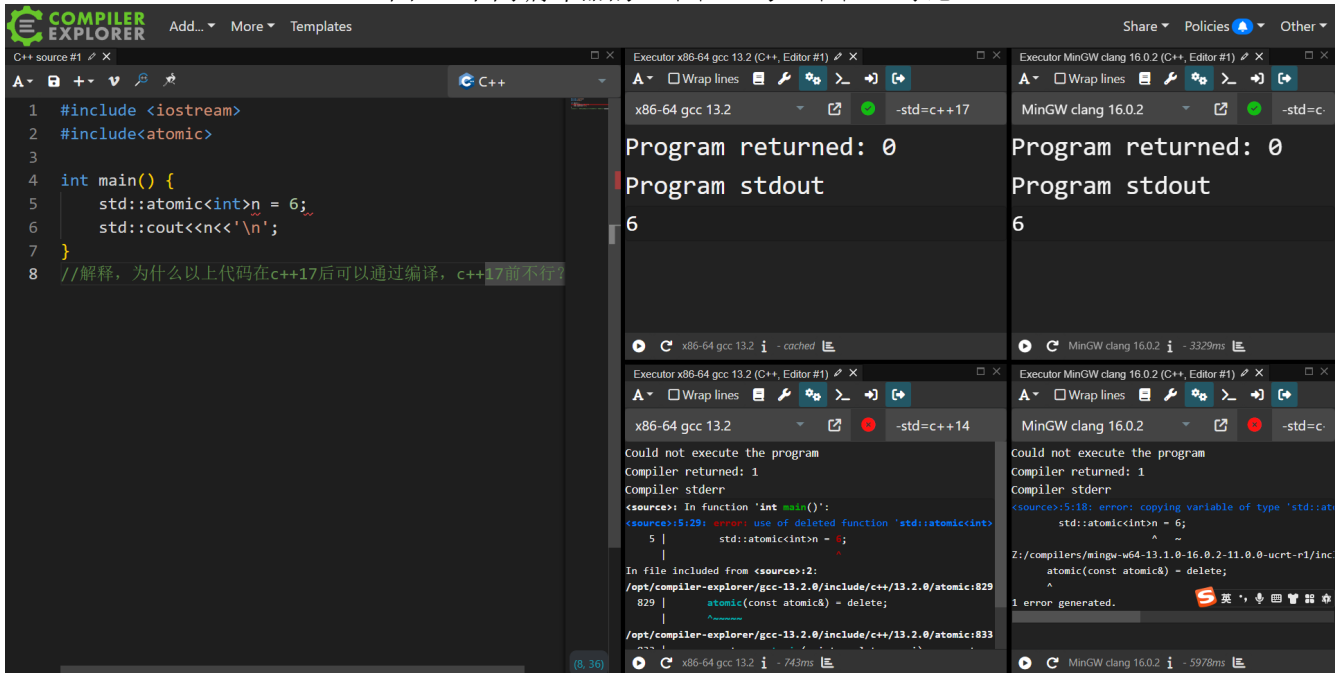
6 解释 std::atomic 初始化

日期: 2023/8/2 出题人: mq 白

```
1  #include <iostream>  
2  #include <atomic>  
3  int main() {  
4      std::atomic<int> n = 6;  
5      std::cout << n << '\n';  
6  }
```

详细解释, 为什么以上代码在 C++17 后可以通过编译, C++17 前不行?

图 1: 不同编译器的 C++17 与 C++14 对比



- 难度: ★★☆☆☆

提示: 复制消除。

7 throw new MyException

日期: 2023/8/6 出题人: mq 白

给出代码:

```

1 struct MyException : std::exception {
2     const char* data{};
3     MyException(const char* s) : data(s) { puts("MyException()"); }
4     ~MyException() { puts("~MyException()"); }
5     const char* what() const noexcept { return data; }
6 };
7 void f2() {
8     throw new MyException("new Exception 异常....");
9 }
10 int main(){
11     f2();

```


12 }

灵感来源自 Java 人写 C++。

在 main 函数中自行修改代码，接取 f2() 函数抛出的异常（try catch）。

要求运行结果

```
MyException()
new Exception 异常....
MyException()
```

- 难度：★ ☆ ☆ ☆ ☆

提示：std::exception, try catch

8 定义 array 推导指引

日期：2023/8/12 出题人：mq 白

给出代码：

```
1  template<class Ty, std::size_t size>
2  struct array {
3      Ty* begin() { return arr; };
4      Ty* end() { return arr + size; };
5      Ty arr[size];
6  };
7  int main() {
8      ::array arr{1, 2, 3, 4, 5};
9      for (const auto& i : arr) {
10         std::cout << i << ' ';
11     }
12 }
```

要求自定义推导指引，不更改已给出代码，使得代码成功编译并满足运行结果。

要求运行结果

```
1 2 3 4 5
```

- 难度: ★ ★ ★ ☆ ☆

提示: 参考 `std::array` 实现, C++17 类模板推导指引

9 名字查找的问题

日期: 2023/8/15 出题人: mq 白

```
1  #include<iostream>
2
3  template<class T>
4  struct X {
5      void f()const { std::cout << "X\n"; }
6  };
7
8  void f() { std::cout << " 全局\n"; }
9
10 template<class T>
11 struct Y : X<T> {
12     void t()const {
13         this->f();
14     }
15     void t2()const {
16         f();
17     }
18 };
19
20 int main() {
21     Y<void>y;
22     y.t();
23     y.t2();
24 }
```

给出以上代码, 要求解释其运行结果。

要求运行结果

X
全局

- 难度：★★★★☆☆

提示：名字查找。本问题堪称经典，在某著名 template 书籍也有提过（虽然它完全没有讲清楚）。并且从浅薄的角度来说，本题也可以让你向其他人证明加 this 访问类成员，和不加，是有很多区别的。

10 遍历任意聚合类数据成员

日期：2023/8/18 出题人：mq 白

题目的要求非常简单，在很多其他语言里也经常提供这种东西（一般是反射）。但是显而易见 C++ 没有反射。

我们给出代码：

```
1  int main() {
2      struct X { std::string s{ " " }; }x;
3      struct Y { double a{}, b{}, c{}, d{}; }y;
4      std::cout << size<X>() << '\n';
5      std::cout << size<Y>() << '\n';
6
7      auto print = [](const auto& member) {
8          std::cout << member << ' ';
9      };
10     for_each_member(x, print);
11     for_each_member(y, print);
12 }
```

要求自行实现 for_each_member 以及 size 模板函数。要求支持任意自定义类类型（聚合体）的数据成员遍历（聚合体中存储数组这种情况不需要处理）。这需要打表，那么我们的要求是支持聚合体拥有 0 到 4 个数据成员的遍历。

要求运行结果

```
1
4
0 0 0 0
```

- 难度：★★★★☆☆

提示：学习，boost::pfr。

11 `emplace_back()` 的问题

日期: 2023/8/20 出题人: jacky

思考: 以下代码为什么在 C++20 以下的版本中无法成功编译, 而在 C++20 及以后却可以?

```
1  #include <vector>
2
3  struct Pos {
4      int x;
5      int y;
6  };
7
8  int main(){
9      std::vector<Pos> vec;
10     vec.emplace_back(1, 5);
11 }
```

- 难度: ★★☆☆☆

提示: `new`, 聚合初始化。

12 实现 `make_vector()`

日期: 2023/8/28 出题人: jacky

请实现函数 `make_vector(...)`, 使以下代码编译通过 (C++20):

```
1  #include <cstdio>
2  #include <vector>
3
4  inline void dbg(const char* msg)
5  {
6      std::puts(msg);
7      std::fflush(stdout);
8  }
9
10 struct X {
```

```
11     X() noexcept
12     {
13         dbg("X()");
14     };
15
16     ~X() noexcept
17     {
18         dbg("~X()");
19     };
20
21     X(const X&)
22     {
23         dbg("X(const X&)");
24     }
25
26     X(X&&) noexcept
27     {
28         dbg("X(X&&)");
29     }
30 };
31
32 void test()
33 {
34     static_assert(requires {
35         {
36             make_vector(std::vector{1, 2, 3})
37         } -> std::same_as<std::vector<std::vector<int>>>>;
38         {
39             make_vector(1, 2, 3)
40         } -> std::same_as<std::vector<int>>>;
41         make_vector(1, 2, 3).size() == 3;
42     });
43     X    x1;
44     X    x2;
45     auto vec = make_vector(x1, std::move(x2));
46 }
47
```

```

48  int main()
49  {
50      test();
51      dbg("test end");
52  }

```

要求运行结果

```

X()
X()
X(const X&)
X(X&&)
X(const X&)
X(const X&)
X()
X()
X()
X()
X()
X()
test end

```

• 难度: ★ ★ ★ ☆ ☆

提示: 重载决议

13 关于 `return std::move(expr)`

日期: 2023/9/6 出题人: mq 白

我们会给出三段使用到了 `return std::move(expr)` 代码。

解释说明这些代码是否有问题, 问题在哪, 或者没问题, 那么为什么要这样使用。

1. 全局函数, 返回局部对象, 使用 `std::move`。

```

1      #include <iostream>
2
3      struct X{//后续代码不再重复 X 类
4          X() { puts("X()"); }

```

```
5         X(const X&) { puts("X(const X&)"); }
6         X(X&&)noexcept { puts("X(X&&)"); }
7         ~X() { puts("~X()"); }
8     };
9
10    X f(){
11        X x;
12        return std::move(x);
13    }
14
15    int main(){
16        X x = f();
17    }
```

2. 全局函数，返回局部的引用，使用 `std::move`。

```
1     X&& f(){
2         X x;
3         return std::move(x);
4     }
```

3. 类中成员函数，返回数据成员，使用 `std::move`。

```
1     struct Test {
2         X x;
3         X f() {
4             return std::move(x);
5         }
6     };
```

• 难度：★★★☆☆

提示：return 重载决议。

14 以特殊方法修改命名空间中声明的对象

日期：2023/12/5 出题人：mq 白

给出以下代码，不得修改，要求不得以

- `ss::a`
- `using namespace ss`
- `namespace x = ss, x::a`
- `using ss::a`
- 直接在 `ss` 命名空间中通过声明引用或指针指向 `a`，然后再去修改 `a`

这些方式去修改命名空间 `ss` 中的对象 `a`，并且满足运行结果。

需要真的修改了 `a`，而不是别的东西，诸如更改入口函数等。

不要求你的做法完全符合标准。

```
1  #include<iostream>
2
3  namespace ss {
4      int a = 0;
5  }
6
7  int main() {
8      // todo..
9      std::cout << ss::a << '\n';
10 }
```

运行结果

100

- 难度：★★★★☆

15 表达式模板

日期：2024/1/7 出题人：Matrix-A（#159）

1. 使用表达式模板补全下面的代码，实现表达式计算；

2. 指出表达式模板和 STL Ranges 库中哪些视图类似，并指出它们的异同和优缺点。

```
1  #include <algorithm>
2  #include <functional>
3  #include <iostream>
4  #include <ranges>
5  #include <vector>
6
7  // 为 std::vector 增加一个自定义的赋值函数
8  template <typename T>
9      requires std::disjunction_v<std::is_integral<T>, std::is_floating_point<T>>
10 class vector : public std::vector<T> {
11 public:
12     using std::vector<T>::vector;
13     using std::vector<T>::size;
14     using std::vector<T>::operator[];
15     template <typename E>
16     vector<T>& operator=(const E& e)
17     {
18         const auto count = std::min(size(), e.size());
19         this->resize(count);
20         for (std::size_t idx { 0 }; idx < count; ++idx) {
21             this->operator[](idx) = e[idx];
22         }
23         return *this;
24     }
25 };
26
27 /*
28 // 实现表达式模板类及相关函数
29 template<...>
30 struct vector_expr {
31
32 };
33
34 // operator+
35 // operator-
```

```

36 // operator*
37 // operator/
38 */
39
40 int main()
41 {
42     auto print = [](const auto& v) {
43         std::ranges::copy(v, std::ostream_iterator<std::ranges::range_value_t<decltype(v)>> { s
44         std::cout << std::endl;
45     };
46     const vector<double> a { 1.2764, 1.3536, 1.2806, 1.9124, 1.8871, 1.7455 };
47     const vector<double> b { 2.1258, 2.9679, 2.7635, 2.3796, 2.4820, 2.4195 };
48     const vector<double> c { 3.9064, 3.7327, 3.4760, 3.5705, 3.8394, 3.8993 };
49     const vector<double> d { 4.7337, 4.5371, 4.5517, 4.2110, 4.6760, 4.3139 };
50     const vector<double> e { 5.2126, 5.1452, 5.8678, 5.1879, 5.8816, 5.6282 };
51
52     {
53         vector<double> result(6);
54         for (std::size_t idx = 0; idx < 6; idx++) {
55             result[idx] = a[idx] - b[idx] * c[idx] / d[idx] + e[idx];
56         }
57         print(result);
58     }
59     {
60         vector<double> result(6);
61         result = a - b * c / d + e; // 使用表达式模板计算
62         print(result);
63     }
64     return 0;
65 }

```

运行结果

```

4.73472, 4.05709, 5.038, 5.08264, 5.73076, 5.18673,
4.73472, 4.05709, 5.038, 5.08264, 5.73076, 5.18673,

```

- 难度：待定

学习链接：

- [Wikipedia - Expression templates](#)
- 我们不需要臭名昭著的表达式模板（英文）
- C++ 语言的表达式模板：表达式模板的入门性介绍
- `std::valarray` 在一些 STL 实现中使用了表达式模板

16 制造传递函数模板的宏

日期：2024/2/6 出题人：mq 白

函数模板不是函数，所以如果要传递，必须实例化，指明模板参数，比如以下代码无法通过编译：

```
1  template< class T >
2  const T& min(const T& a, const T& b) {
3      return a < b ? a : b;
4  }
5
6  template<typename F, class...Args>
7  auto foo(F f, Args&&...args) {
8      return f(std::forward<Args>(args)...);
9  }
10
11 int main() {
12     const auto result = foo(::min, 2, 3);
13     std::cout << result << '\n';
14 }
```

除非我们把里面的 `::min` 换成 `::min<int>`。这显然很麻烦，而且很多时候依然会有问题，比如这个函数模板有多个类型模板参数，又或者它有多个重载：

```
1  template< class T >
2  const T& min(const T& a, const T& b) {
3      return a < b ? a : b;
4  }
5
6  template< class T >
7  T min(std::initializer_list<T> ilist) {
```

```
7     return std::min(ilist, std::less<>{});
8 }
9
10 template<typename F, class...Args>
11 auto foo(F f, Args&&...args) {
12     return f(std::forward<Args>(args)...);
13 }
14
15 int main() {
16     const auto result = foo(::min<int>, 2, 3);
17     std::cout << result << '\n';
18 }
```

重载决议根本没有任何办法处理。因此，**通透函数对象**（Transparent function objects）应运而生。

我们要解决以上问题，实现 BY_NAME 宏，从函数模板名创造通透函数对象，使以下代码能够成功编译：

```
1 template<typename F, class...Args>
2 auto foo(F f, Args&&...args) {
3     return f(std::forward<Args>(args)...);
4 }
5
6 int main() {
7     const auto result = foo(BY_NAME(std::min), 2, 3);
8     std::cout << result << '\n';
9 }
```

• 难度：★★☆☆☆