1 ```cpp #include class Base { public: Base() { std::cout << "Base Constructor\n"; print(); // 在构造函数中调用虚函数 } virtual ~Base() { std::cout << "Base Destructor\n"; print(); // 在析构函数中调用虚函数 } virtual void print() const { std::cout << "Base::print()\n"; } }; class Derived : public Base { public: Derived() { std::cout << "Derived Constructor\n"; } ~Derived() { std::cout << "Derived Destructor\n"; } void print() const override { std::cout << "Derived::print()\n"; } }; int main() { Base* obj = new Derived(); delete obj; return 0; } ``` **问题：上述程序的输出是什么？为什么？** Base Constructor Base::print() Derived Constructor Derived Destructor Base Destructor Base::print() --- # 2 ```cpp #include class A { public: A() { std::cout << "A Constructor\n"; } virtual ~A() { std::cout << "A Destructor\n"; } }; class B : virtual public A { public: B() { std::cout << "B Constructor\n"; } ~B() { std::cout << "B Destructor\n"; } }; class C : virtual public A { public: C() { std::cout << "C Constructor\n"; } ~C() { std::cout << "C Destructor\n"; } }; class D : public B, public C { public: D() { std::cout << "D Constructor\n"; } ~D() { std::cout << "D Destructor\n"; } }; int main() { D obj; return 0; } ``` **问题：上述程序的输出是什么？为什么？** A Constructor B Constructor C Constructor D Constructor D Destructor C Destructor B Destructor A Destructor --- # 3 ```cpp #include class Example { private: int value; mutable int cachedValue; // 可变成员 public: Example(int v) : value(v), cachedValue(0) {} int getValue() const { cachedValue++; // 允许修改 return value; } int getCachedValue() const { return cachedValue; } }; int main() { const Example e(10); // 常量对象 std::cout << "Value: " << e.getValue() << std::endl; // 输出什么？ std::cout << "Cached Value: " << e.getCachedValue() << std::endl; // 输出什么？ return 0; } ``` **问题：上述程序的输出是什么？为什么？** Value: 10 Cached Value: 1 --- # 4 ```cpp #include void func(int a) { std::cout << "Function with int: " << a << std::endl; } void func(double d) { std::cout << "Function with double: " << d << std::endl; } int main() { short s = 10; func(s); // 调用哪个版本的func？ return 0; } ``` --- # 5 ```cpp #include class A { public: static int value; A() { value++; } }; int A::value = 0; // 静态成员变量初始化 int main() { A a1; A a2; std::cout << A::value << std::endl; // 输出什么？ return 0; } ``` **问题：上述程序的输出是什么？为什么？** 2 --- # 6 ```cpp #include #include #include class Test { public: std::string data; Test(const std::string& str) : data(str) { std::cout << "Constructed with data: " << data << std::endl; } Test(Test&& other) noexcept : data(std::move(other.data)) { std::cout << "Move constructor called\n"; } ~Test() { std::cout << "Destructing object with data: " << data << std::endl; } }; int main() { Test t1("Hello"); Test t2(std::move(t1)); std::cout << "After move, t1 data: " << t1.data << std::endl; return 0; } ``` **问题：程序输出是什么？为什么？** Constructed with data: Hello Move constructor called Destructing object with data: Destructing object with data: Hello After move, t1 data: --- # 7 ```cpp #include constexpr int factorial(int n) { return n <= 1 ? 1 : (n * factorial(n - 1)); } int main() { constexpr int val = factorial(5); // 编译时计算 std::cout << "Factorial of 5 is: " << val << std::endl; int n = 4; int runtime_val = factorial(n); // 运行时计算 std::cout << "Factorial of 4 is: " << runtime_val << std::endl; return 0; } ``` **问题：程序输出是什么？为什么？** Factorial of 5 is: 120 Factorial of 4 is: 24 如果所有输入都是常量，constexpr函数会在编译时计算结果，否则会在运行时计算。 --- # 8 ```cpp #include #include std::optional divide(int a, int b) { if (b == 0) return std::nullopt; return a / b; } int main() { auto result = divide(10, 0); std::cout << "Result: " << result.value_or(-1) << std::endl; // 这里输出是什么？ result = divide(10, 2); std::cout << "Result: " << *result << std::endl; // 这里输出是什么？ return 0; } ``` **问题：上述程序的输出是什么？为什么？** Result: -1 Result: 5 std::optional 提供了一个安全的方式来表示"无值"的情况。 divide(10, 0)返回std::nullopt，因此result.value_or(-1)返回-1。 result解引用可选对象并返回其值（如果有值）。对于divide(10, 2)，结果是5。 --- # 9 ```cpp #include #include #include int main() { std::variant var = "Hello"; try { std::cout << std::get<0>(var) << std::endl; // 尝试访问错误的类型 } catch (const std::bad_variant_access& e) { std::cout << "Exception: " << e.what() << std::endl; } var = 42; // 改变为int类型 std::cout <<

std::get(var) << std::endl; // 这里输出什么？ return 0; } ``` **问题：上述程序的输出是什么？为什么？** Exception: bad_variant_access 42 --- # 10 ```cpp #include #include void overloaded(int& x) { std::cout << "Lvalue reference overload: " << x << std::endl; } void overloaded(int&& x) { std::cout << "Rvalue reference overload: " << x << std::endl; } template void forwarding(T&& arg) { overloaded(std::forward(arg)); // 完美转发 } int main() { int a = 10; forwarding(a); // 调用哪个版本？ forwarding(20); // 调用哪个版本？ return 0; } ``` **问题：上述程序的输出是什么？为什么？** Lvalue reference overload: 10 Rvalue reference overload: 20 --- # 11 ```cpp #include constexpr int dangerous(int x) { int* ptr = nullptr; if (x > 0) { ptr = &x; } return *ptr; // 可能的未定义行为 } int main() { constexpr int val = dangerous(0); // 这里会发生什么？ std::cout << val << std::endl; // 如果上面没有问题，这里会输出什么？ return 0; } ``` **问题：程序会成功编译和运行吗？如果没有，是什么原因？** 编译时错误。 --- # 12 ```cpp #include #include template decltype(auto) forward_type(T&& t) { return std::forward(t); // 返回什么类型？ } int main() { int a = 5; auto result = forward_type(a); // result的类型是什么？ static_assert(std::is_same_v, "Type mismatch!"); // 这里会通过吗？ const int b = 10; auto result2 = forward_type(b); // result2的类型是什么？ static_assert(std::is_same_v, "Type mismatch!"); // 这里会通过吗？ auto result3 = forward_type(20); // result3的类型是什么？ static_assert(std::is_same_v, "Type mismatch!"); // 这里会通过吗？ return 0; } ``` **问题：static_assert语句是否会通过？如果没有，是什么原因？** 所有的static_assert都会通过。 decltype(auto)会保留表达式的值类别和const性。 std::forward(t)根据T的类型会保持原类型。 forward_type(a)推导为int&。 forward_type(b)推导为const int&。 forward_type(20)推导为int&&。 --- # 13 ```cpp #include #include #include std::atomic counter(0); void increment(int n) { for (int i = 0; i < n; ++i) { counter.fetch_add(1, std::memory_order_relaxed); } } int main() { std::thread t1(increment, 1000); std::thread t2(increment, 1000); t1.join(); t2.join(); std::cout << "Final counter value: " << counter << std::endl; // 结果总是2000吗？ return 0; } ``` **问题：程序输出的结果总是2000吗？为什么？** 总是2000。 std::atomic保证对counter的操作是原子的，使用fetch_add并指定memory_order_relaxed的内存顺序选项也可以确保多个线程安全地进行自增操作。 虽然memory_order_relaxed不提供同步，但它保证了操作的原子性，所以不会导致数据竞争。因此，结果总是2000。 --- # 14 ```cpp #include #include struct Awaiter { bool await_ready() { return false; } void await_suspend(std::coroutine_handle<>) {} void await_resume() { std::cout << "Resumed!\n"; } }; struct Task { struct promise_type { Task get_return_object() { return {}; } std::suspend_never initial_suspend() { return {}; } std::suspend_always final_suspend() noexcept { return {}; } void return_void() {} void unhandled_exception() {} }; Awaiter operator co_await() const { return {}; } // 隐式等待 }; Task example() { co_await Task{}; std::cout << "After co_await\n"; } int main() { example(); // 输出什么？ return 0; } ``` **问题：程序的输出是什么？为什么？** Resumed! After co_await --- # 15 ```cpp #include #include template struct Factorial { static constexpr int value = N * Factorial::value; }; template<> struct Factorial<0> { static constexpr int value = 1; }; template void printFactorial(T t) { if constexpr (std::is_integral_v) { std::cout << Factorial::value << std::endl; // 这里会出什么问题？ } else { std::cout << "Non-integral type not supported" << std::endl; } } int main() { printFactorial(5); // 输出什么？ printFactorial(3.5); // 输出什么？ return 0; } ``` **程序会编译通过吗？如果通过，输出是什么？如果没有，是什么原因？** 不会编译通过。 虽然if constexpr语句看起来像是一个编译时的条件分支，但编译器仍会解析整个模板实例化的表达式。 Factorial::value会在编译时被实例化，而t是一个非类型模板参数，不能为double。 因此，printFactorial(3.5)的调用会导致编译错误。需要一个静态断言来避免实例化时的错误。 --- # 16 ```cpp #include template void print(T val) { std::cout << "General template: " << val << std::endl; } template<> void print(int val) { std::cout << "Specialized template for int: " << val << std::endl; }

template void print(const T* val) { std::cout << "Pointer specialization: " << *val << std::endl; } int main() { int x = 5; print(x); // 输出什么？ print(&x); // 输出什么？ print("Hello"); // 输出什么？ return 0; } ``` **程序的输出是什么？为什么？** Specialized template for int: 5 Pointer specialization: 5 Pointer specialization: H print(x)调用了int的完全特化版本。 print(&x)调用了指针的模板特化版本。 print("Hello")调用了指针特化版本，由于const char*指针的解引用输出'H'。 --- # 17 ```cpp #include #include struct A { int* p; A() : p(new int(42)) {} A(A&& other) noexcept : p(other.p) { other.p = nullptr; } ~A() { if (p) delete p; } }; int main() { A a; A b = std::move(a); if (a.p) // 检查是否为nullptr std::cout << "a.p is not nullptr, value: " << *a.p << std::endl; // 会输出什么？ else std::cout << "a.p is nullptr" << std::endl; return 0; } ``` **程序的输出是什么？是否有未定义行为**是UB --- # 18 ```cpp #include #include int main() { std::vector v(3, true); // 创建一个bool类型的vector v[0] = false; bool* p = &v[0]; // 获取指向第一个元素的指针 if (p[0] == false) std::cout << "p[0] is false" << std::endl; else std::cout << "p[0] is true" << std::endl; p[1] = false; // 修改第二个元素，这里会出问题吗？ std::cout << "v[1] is " << (v[1] ? "true" : "false") << std::endl; // 输出什么？ return 0; } ``` **程序的输出是什么？会出现什么问题？** 是UB。 std::vector是一个特例，它不存储bool类型，而是使用了位字段（Bitfield）进行压缩。因此，&v[0]得到的指针并不是真正的bool*，而是一个代理类（Proxy）对象。 当你尝试使用p指针进行指针运算或解引用时，会导致未定义行为（UB），因为这些操作并不适用于代理对象。 --- # 18 ```cpp #include #include class WeirdContainer { int data[5]; public: WeirdContainer() : data{1, 2, 3, 4, 5} {} int& operator[](std::size_t i) { return data[i]; } const int& operator[](std::size_t i) const { return data[i]; } auto get_span() { return std::span(data, 5); } // 返回一个span }; int main() { WeirdContainer wc; auto sp = wc.get_span(); // 获取一个std::span对象 wc[1] = 10; // 直接修改WeirdContainer的元素 std::cout << "wc[1]: " << wc[1] << ", sp[1]: " << sp[1] << std::endl; // 会输出什么？ sp[1] = 20; // 通过span修改元素 std::cout << "wc[1]: " << wc[1] << ", sp[1]: " << sp[1] << std::endl; // 会输出什么？ return 0; } ``` **程序的输出是什么？为什么？** wc[1]: 10, sp[1]: 10 wc[1]: 20, sp[1]: 20 std::span是一个指向连续内存的轻量级视图，不管理内存，直接引用原始数据。 当通过wc[1]或sp[1]修改数据时，都是修改同一个底层数组data。因此，修改通过wc和sp都会反映到对方。 --- ```cpp #include constexpr int deep_recursion(int n) { return (n <= 0) ? 1 : (1 + deep_recursion(n - 1)); // 会导致栈溢出吗？ } int main() { constexpr int val = deep_recursion(100000); // 这里的n取100000 std::cout << "Value: " << val << std::endl; return 0; } ``` **程序能成功编译和运行吗？如果没有，是什么原因？** 不知道。 --- # 19 ```cpp #include #include struct Base { int x; }; struct Derived : Base { int y; }; int main() { alignas(Derived) char buffer[sizeof(Derived)]; // 手动分配内存 auto* d = new (buffer) Derived{ {1}, 2 }; // 在buffer中构造Derived Base* b = d; std::cout << b->x << std::endl; // 输出1 b->~Base(); // 显式销毁Base new (b) Base{3}; // 在同一位置构造新的Base std::cout << std::launder(&d->x)->x << std::endl; // 这里会输出什么？ return 0; } ``` **程序的输出是什么？为什么？** 是UB。 --- # 20 ```cpp #include #include struct S { S(int, double) { std::cout << "S(int, double)" << std::endl; } S(std::initializer_list) { std::cout << "S(initializer_list)" << std::endl; } }; int main() { S s1{1, 3.14}; // 调用哪个构造函数？ S s2({1, 2}); // 调用哪个构造函数？ S s3 = {1, 3.14}; // 调用哪个构造函数？ return 0; } ``` **每个构造函数调用的结果是什么？为什么？** S(initializer_list) S(initializer_list) S(int, double) S s1{1, 3.14}：因为1和3.14都可以转换为initializer_list中的int，所以优先调用S(std::initializer_list)。 S s2({1, 2})：直接传递initializer_list，调用S(std::initializer_list)。 S s3 = {1, 3.14}：使用拷贝初始化，编译器优先使用S(int, double)构造函数，因为这种初始化要求匹配的构造函数。 ---