【C++】C++11 新特性总结 | C++ 常见设计模式总结（秋招篇）

今天一定要洛必达 于 2023-08-06 23:46:11 发布

文章目录

前言

提示：这里可以添加本文要记录的大概内容：

例如：随着 人工智能 的不断发展，机器学习这门技术也越来越重要，很多人都开启了学习机器学习，本文就介绍了机器学习的基础内容。

介绍几种C++11新特性

C++11引入了许多新的特性和改进，可以分为以下几个方面：

1.自动类型推导（Type Inference）：引入了auto和decltype关键字，使得编译器能够根据初始化表达式自动推导变量的类型。

2.列表初始化（Uniform Initialization）：使用统一的语法{}进行初始化，可以用于初始化各种类型的对象，包括基本类型、数组和类对象。

3.右值引用（Rvalue References）和移动语义（Move Semantics）：引入了新的引用类型&&，允许将右值（临时对象）绑定到右值引用上，并且可以通过移动语义实现高效的资源管理。

4.Lambda表达式：允许在代码中定义匿名函数，方便编写简洁的函数对象或闭包。

5.智能指针（Smart Pointers）：引入了shared_ptr、unique_ptr和weak_ptr等智能指针类型，用于管理动态分配的对象，避免内存泄漏和资源管理问题。

6.并发编程支持：引入了原子操作、线程库和互斥量等机制，使得多线程编程更加方便和安全。

7.新的标准库组件：引入了一些新的标准库组件，如、、和<unordered_map>等，提供了更多的数据结构和算法支持。

8.强类型枚举（Strongly Typed Enumerations）：引入了强类型枚举，使得枚举类型更加类型安全和可控。

9.nullptr关键字：引入了nullptr关键字，用于表示空指针，取代了传统的NULL宏。

10.静态断言（Static Assertion）：引入了static_assert关键字，用于在编译时进行静态断言，检查一些编译期常量表达式的真假。

11.委托构造函数（Delegating Constructors）：允许一个构造函数调用同一个类的其他构造函数，简化了构造函数的实现。

12.可变参数模板（Variadic Templates）：引入了可变参数模板，允许函数模板和类模板接受任意数量和类型的参数。

13.范围基于的for循环（Range-based for loop）：引入了for循环的新语法，可以方便地遍历容器或其他可迭代对象的元素。

…等
这些是C++11引入的一些主要特性，它们使得C++语言更加现代化、安全和高效，为开发人员提供了更多的工具和选择。

介绍一下自动类型推导auto和decltype关键字的用法

auto和decltype是C++11引入的关键字，用于进行**类型推导**（Type Inference），使得编译器能够根据初始化表达式自动推导变量的类型，而无需手动指定类型。这样可以让代码更加简洁、易读且更容易维护。（这就有点像python了）

1.auto关键字：

在声明变量时使用auto关键字，编译器会根据初始化表达式的类型自动推导出变量的类型。

auto适用于大部分情况，如基本数据类型、对象、函数返回值等。

auto还可以结合范围基于的for循环，让编译器推导出范围内元素的类型。

示例：

```
auto num = 42; // num被推导为int类型
auto name = "John"; // name被推导为const char*类型
auto pi = 3.14159; // pi被推导为double类型

std::vector<int> numbers = {1, 2, 3, 4, 5};
for (auto& num : numbers) {
    num *= 2;
}
```

这里auto& num 加上&符号是为了避免复制带来的开销，节省内存

2.decltype关键字：

decltype用于获取表达式的类型，而不是根据初始化表达式来推导变量类型。

可以用于获取变量、函数返回值、表达式等的类型，包括cv限定符（const和volatile）和引用。

示例：

```cpp
int x = 5;
const int& y = x;

decltype(x) a = x; // a被推导为int类型
decltype(y) b = x; // b被推导为const int&类型
decltype(x * y) c = x * y; // c被推导为const int&类型，因为x和y中有const限定符
```

auto和decltype的引入，使得代码更加灵活，减少了类型相关的冗余代码，同时还能保持代码的类型安全性。使用这些关键字可以让开发者更专注于逻辑而不是繁琐的类型声明。但同时，**过度使用这些关键字可能会使代码可读性下降，所以需要适度使用并结合良好的命名和注释。**

举例讲一下范围基于的for循环

这个语法很像python里的for i in range()

当使用范围基于的for循环时，可以通过简单的语法来遍历容器中的元素，而无需使用迭代器或索引。下面是一个使用范围基于的for循环的示例：

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // 使用范围基于的for循环遍历容器中的元素
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

在上面的示例中，我们创建了一个整数类型的vector numbers，然后使用范围基于的for循环遍历它。循环的语法是for (const auto& num : numbers)，其中：

const auto& num 是循环变量的声明，auto关键字会自动推导出num的类型为容器元素的类型，const关键字表示在循环中我们不会修改元素。 加上&符号是为了避免复制带来的开销，节省内存

numbers 表示要遍历的范围，即numbers容器中的元素。

在每次循环迭代中，循环变量num会依次取到numbers容器中的每个元素的值，并通过std::cout输出到控制台。输出结果将会是：1 2 3 4 5

通过范围基于的for循环，我们可以更简洁地遍历容器中的元素，无需担心索引或迭代器的细节。这使得代码更加易读和简洁。

介绍一下列表初始化

列表初始化（Uniform Initialization）是C++11引入的一个特性，它允许我们使用统一的语法{}来初始化各种类型的对象，包括基本类型、数组、结构体和类对象。这种初始化方式在语法上更加一致，并且能够避免一些初始化相关的问题。以下是一些例子来说明列表初始化的用法：

1.初始化基本类型和自定义类型：

```
int num1 = 5;
int num2{10}; // 使用列表初始化

std::string text1 = "Hello";
std::string text2{"World"};

struct Point {
    int x;
    int y;
};

Point p1 = {1, 2};
Point p2{3, 4};
```

2.初始化数组：

```
int arr1[3] = {1, 2, 3};
int arr2[] = {4, 5, 6}; // 自动推导数组大小

std::vector<int> numbers = {7, 8, 9};
```

3.初始化类对象：

```
class Person {
public:
    Person(std::string n, int a) : name(n), age(a) {}

    std::string name;
    int age;
};

Person person1("Alice", 25);
Person person2{"Bob", 30};
```

4.初始化嵌套结构和容器：

```
struct Rectangle {
    int width;
    int height;
};

Rectangle rect1{4, 5};

std::vector<std::string> fruits = {"apple", "banana", "orange"};
```

列表初始化通过统一的语法{}提供了更一致和直观的初始化方式，它适用于各种不同类型的对象，包括基本类型、数组、结构体和类对象。它还可以避免一些常见的初始化问题，如窄化转换（narrowing conversion）等。这是C++11引入的一个重要特性，能够使代码更加简洁和可读。

讲一下右值引用，和左值引用的区别，有何作用?

右值引用和左值引用是C++11引入的两种不同类型的引用，它们在语义和用途上有着明显的区别。

区别：

左值引用：使用单个 ampersand & 表示，绑定到具有名称且可寻址的对象，即左值。例**如，变量、对象或函数返回的左值都可以绑定到左值引用。**左值引用主要用于传递和修改对象的值。

右值引用：使用双 ampersand && 表示，**绑定到临时对象、字面常量和表达式等即将要被销毁的临时值，即右值。**例如，函数返回的临时对象或使用 std::move() 转换后的对象都可以绑定到右值引用。右值引用主要用于支持转移语义和移动语义，提高性能和资源管理效率。

作用：

左值引用的主要作用是允许函数修改传入的参数，避免不必要的拷贝开销。通过左值引用，函数可以直接操作传入的对象，而不需要进行额外的拷贝。这在传递大型对象时非常有用，可以提高性能并避免内存开销。

右值引用的主要作用是支持转移语义和移动语义。通过右值引用，我们可以将资源从一个对象转移到另一个对象，而不进行深拷贝，从而提高性能和资源管理效率。右值引用特别适用于大型对象的移动操作，例如在实现移动构造函数和移动赋值运算符时。

示例：

```
#include <iostream>

void modifyValue(int& value) {
    value = 100;
}

```

```cpp
void processValue(int&& value) {
    std::cout << "Received Rvalue: " << value << std::endl;
}

int main() {
    int x = 42; // x 是左值
    int&& rref = 100; // rref 是右值引用，绑定到临时值

    modifyValue(x); // 传递左值
    std::cout << "x after modifyValue: " << x << std::endl; // 输出 100

    processValue(123); // 传递右值

    return 0;
}
```

在上述示例中，我们定义了一个使用左值引用参数的函数 modifyValue 和一个使用右值引用参数的函数 processValue。在 main 函数中，我们创建了一个左值 x 和一个右值引用 rref。

总结：

右值引用允许我们在C++中实现转移语义和移动语义，以提高性能和资源管理效率。同时，左值引用则用于传递和修改对象的值，让函数能够直接操作传入的对象而不需要进行额外的拷贝。这两种引用类型在C++中一起工作，为代码的优化和性能提供了很好的支持。

讲一下转移语义和移动语义（移动构造函数）

转移语义（Move Semantics）和移动语义（Move Semantics）是 C++11 引入的概念，旨在优化对象的资源管理，提高性能并减少不必要的拷贝操作。

转移语义（Move Semantics）：

转移语义是指将一个对象的资源（例如堆分配的内存、文件句柄等）从一个对象转移到另一个对象，同时使原始对象保持在有效但已被“掏空”的状态。这意味着在转移资源的同时，避免了深拷贝，从而提高了性能。

转移语义的核心是通过右值引用来实现，右值引用允许我们获取对临时值（右值）的引用，这些临时值即将被销毁。通过移动资源而不是复制，可以避免不必要的开销。

移动语义（Move Semantics）：

移动语义是实现转移语义的机制，它允许我们在对象的拷贝构造函数和拷贝赋值运算符中检测并优化移动操作。通过移动构造函数和移动赋值运算符，我们可以将资源从一个对象转移到另一个对象，避免额外的拷贝开销。

移动构造函数用于从右值创建新对象，将资源从右值转移到新对象中，并将原始右值置于有效但空状态。移动赋值运算符用于将资源从一个对象移动到另一个对象，同时使原始对象保持在有效但空状态。

示例：

```cpp
#include <iostream>

class MyString {
public:
    MyString(const char* str) {
        size = strlen(str);
        data = new char[size + 1];
        strcpy(data, str);
    }

    // 移动构造函数
    MyString(MyString&& other) noexcept : data(other.data), size(other.size) {
        other.data = nullptr;
        other.size = 0;
    }

    // 移动赋值运算符
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            size = other.size;
            other.data = nullptr;
            other.size = 0;
        }
        return *this;
    }

    ~MyString() {
        delete[] data;
    }

private:
    char* data;
    size_t size;
};

int main() {
    MyString str1("Hello");
    MyString str2 = std::move(str1); // 使用移动构造函数
```

```
    MyString str3("World");
    str3 = std::move(str2); // 使用移动赋值运算符

    return 0;
}
```

在上述示例中，我们定义了一个简单的 MyString 类，实现了移动构造函数和移动赋值运算符。这些函数允许在创建对象和赋值对象时执行资源的转移，而不是进行**深拷贝**。这有助于提高性能，特别是对于大型对象或需要频繁传递的对象。
总之，转移语义和移动语义在 C++ 中是优化性能和资源管理的重要机制，通过避免不必要的拷贝操作，提高了代码的效率和性能。

举例介绍一下Lambda表达式

Lambda表达式是C++11引入的一种用于创建匿名函数的方法，它允许在需要函数的地方内联定义函数，从而避免了显式定义独立函数或函数对象的繁琐过程。Lambda表达式的语法相对简洁，并且可以捕获外部变量，使得它们在某些情况下非常方便。

Lambda表达式的语法结构如下：

```
[capture_list](parameter_list) -> return_type {
    // 函数体
    // 返回语句（如果有）
}
```

其中：

capture_list：用于捕获外部变量，可以为空或者包含多个外部变量。捕获方式有以下几种：

[]：不捕获任何外部变量。

[var]：捕获变量var，但不可修改。

[&var]：以引用方式捕获变量var，可以修改。

[=]：以值方式捕获所有外部变量。

[&]：以引用方式捕获所有外部变量。

[=, &var]：以值方式捕获所有外部变量，并以引用方式捕获变量var。

parameter_list：用于指定Lambda表达式的参数列表，类似于函数的参数列表。参数可以有类型声明，也可以使用auto自动推导类型。

return_type：Lambda表达式的返回类型声明，可以使用auto自动推导返回类型，也可以显式指定返回类型。

{}：Lambda表达式的函数体，用于实现Lambda表达式的功能。

下面是一些具体的Lambda表达式示例：

Lambda表达式没有捕获外部变量，并接受两个整数参数并返回它们的和：

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

捕获外部变量x，并接受一个整数参数，返回x与参数的乘积：

```
int x = 10;
[=] (int value) -> int {
    return x * value;
}
```

捕获外部变量y以引用方式，并接受一个整数参数，返回y与参数的差：

```
int y = 5;
[&] (int value) -> int {
    return y - value;
}
```

捕获多个外部变量并接受无参数，输出它们的值：

```
int a = 1, b = 2, c = 3;
[=] {
    std::cout << "a: " << a << ", b: " << b << ", c: " << c << std::endl;
}
```

Lambda表达式的使用非常灵活，它可以在需要函数的地方直接定义，并且通过捕获外部变量，能够很方便地处理一些上下文相关的逻辑。

具体的几个例子如下：
1.基本的Lambda表达式：

```
auto add = [](int a, int b) { return a + b; };
int result = add(3, 5); // 结果为8
```

2.捕获外部变量：

```
int x = 10;
int y = 5;

auto multiply = [&x, y](int value) { return x * y * value; };
int product = multiply(2); // 结果为100, x = 10, y = 5
```

3.Lambda作为参数传递给函数：

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

// 使用Lambda表达式作为参数来筛选偶数
std::vector<int> evenNumbers;
std::copy_if(numbers.begin(), numbers.end(), std::back_inserter(evenNumbers),
             [](int num) { return num % 2 == 0; });
```

4.Lambda表达式可以带有返回类型的声明：

```
auto power = [](int base, int exponent) -> int {
    int result = 1;
    for (int i = 0; i < exponent; ++i) {
        result *= base;
    }
    return result;
};
```

int value = power(2, 3); // 结果为8

5.在标准算法中使用Lambda表达式：

```
std::vector<int> data = {7, 3, 9, 1, 6};
// 使用Lambda表达式自定义排序规则
std::sort(data.begin(), data.end(), [](int a, int b) { return a > b; });
// data变为{9, 7, 6, 3, 1}
```

介绍一下C++11的智能指针

具体的内容我之前单独写了一篇博客：C++11智能指针
有时间大家可以自己尝试手写一个智能指针

智能指针是C++中用于管理动态内存的智能对象，它们是一种RAII（Resource Acquisition Is Initialization）技术的实现，用于自动管理动态分配的内存和资源，避免了内存泄漏和资源泄漏等问题。智能指针通过在对象的生命周期结束时自动释放分配的内存，从而提供更安全、更简洁的内存管理。
C++标准库中提供了三种主要类型的智能指针：

1）std::unique_ptr: 独占式智能指针，表示对于一个对象或者数组的唯一所有权。当unique_ptr被销毁时，它所管理的对象或数组也会被自动释放。它不允许多个unique_ptr共享同一个对象。

2）std::shared_ptr: 共享式智能指针，可以让多个shared_ptr共享同一个对象的所有权。当最后一个shared_ptr被销毁时，它会自动释放所管理的对象。

3）std::weak_ptr: 弱引用指针，它是用来解决shared_ptr的循环引用问题的。weak_ptr允许对由shared_ptr管理的对象进行观测，但并不拥有对象的所有权。当最后一个shared_ptr销毁后，即使有weak_ptr引用存在，对象也会被释放。

使用智能指针的好处包括：

自动内存管理：**不需要手动调用delete来释放内存**，智能指针会在适当的时候自动释放资源。

避免内存泄漏：智能指针确保在不再需要时正确释放资源，避免了因忘记释放内存而造成的内存泄漏问题。

异常安全：在使用智能指针的过程中，即使出现异常，资源也会被正确释放，确保程序的异常安全性。

使用智能指针时，需要注意避免循环引用，即两个或多个对象之间形成环状引用，导致资源无法正确释放。
下面是一个简单的示例展示如何使用std::unique_ptr和std::shared_ptr：

```cpp
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass(int value) : data(value) {
        std::cout << "Constructor called. Value: " << data << std::endl;
    }

    ~MyClass() {
        std::cout << "Destructor called. Value: " << data << std::endl;
    }

    void print() const {
```

```
        std::cout << "Value: " << data << std::endl;
    }

private:
    int data;
};

int main() {
    // 使用 std::unique_ptr
    std::unique_ptr<MyClass> uniquePtr = std::make_unique<MyClass>(42);
    uniquePtr->print();

    // 使用 std::shared_ptr
    std::shared_ptr<MyClass> sharedPtr1 = std::make_shared<MyClass>(100);
    std::shared_ptr<MyClass> sharedPtr2 = sharedPtr1;
    sharedPtr1->print();
    sharedPtr2->print();

    return 0;
}
```

在上面的示例中，我们首先使用std::unique_ptr创建了一个对象，并输出其值。std::unique_ptr确保在其作用域结束时，所管理的对象会被正确释放。
然后，我们使用std::shared_ptr创建了一个对象，并使用多个shared_ptr共享同一个对象。在输出时，我们可以看到只有在最后一个shared_ptr销毁时，对象的析构函数才会被调用，因为shared_ptr共享着同一个对象的所有权。
总体而言，智能指针是C++中非常有用且推荐使用的特性，可以大大简化动态内存管理的工作，提高代码的安全性和可维护性。

智能指针的原理是什么?

智能指针可以自动回收内存的原理是通过引用计数来管理资源的生命周期。智**能指针是一种特殊的数据结构，它包装了原始指针，并在其内部维护了一个引用计数。引用计数跟踪有多少个智能指针共享同一个对象资源。**
当创建一个智能指针时，引用计数会被初始化为1。当有新的智能指针指向同一个资源时，引用计数会增加。当智能指针被销毁或不再指向某个资源时，引用计数会减少。当引用计数变为0时，表示没有任何智能指针指向该资源，这意味着资源可以被安全地释放。
以下是智能指针自动回收内存的基本原理：

初始化：当一个对象或资源被动态分配时，一个智能指针通过构造函数接管该对象的所有权，并初始化引用计数为1。

复制：当使用另一个智能指针来复制已有的智能指针时，复制构造函数或赋值操作符会增加资源的引用计数，而不会重新分配新的资源。这样，多个智能指针可以共享同一个资源。

销毁：当一个智能指针的作用域结束时（例如，离开代码块），智能指针的析构函数会减少资源的引用计数。如果引用计数变为0，表示没有智能指针指向该资源，那么资源会被销毁，内存会被自动释放。

引用计数的方式使得智能指针可以自动地管理资源的生命周期，避免了手动释放内存和资源泄漏的问题。然而，引用计数的方式也会带来一些性能开销和可能的循环引用问题。为了解决循环引用问题，C++ 提供了 std::weak_ptr，它可以被用来观测而不影响资源的引用计数。这样就可以打破循环引用，使资源得以正确释放。

讲一下三种智能指针的区别

当涉及到智能指针时，三种常用的类型是std::unique_ptr、std::shared_ptr和std::weak_ptr。它们之间有一些重要的区别，主要包括所有权、引用计数和解决循环引用的能力。

1）std::unique_ptr：

所有权：std::unique_ptr **是独占式智能指针**，意味着一个对象只能由一个std::unique_ptr拥有。当拥有对象的std::unique_ptr被销毁或者重置时，它所管理的对象也会被销毁。因此，不能将同一个对象的所有权交给多个std::unique_ptr。

引用计数：std::unique_ptr没有引用计数的概念，因为它是独占的，不需要追踪其他对象是否引用了相同的资源。

解决循环引用：由于独占性，std::unique_ptr不能解决循环引用问题。如果两个对象通过std::unique_ptr相互引用，将导致循环引用，从而导致资源无法正确释放。

2）std::shared_ptr：

所有权：std::shared_ptr 是共享式智能指针，允许多个std::shared_ptr共享同一个对象的所有权。当最后一个std::shared_ptr被销毁或者重置时，它所管理的对象会被销毁。因此，可以将同一个对象的所有权交给多个std::shared_ptr。

引用计数：std::shared_ptr使用引用计数来追踪有多少个std::shared_ptr共享同一个对象。每当新的std::shared_ptr指向对象时，引用计数会增加；当std::shared_ptr被销毁或者重置时，引用计数会减少。当引用计数为零时，对象会被销毁。

解决循环引用：std::shared_ptr 无法直接解决循环引用问题。如果两个或多个对象通过std::shared_ptr相互引用，可能会导致循环引用，从而导致资源无法正确释放。为了避免循环引用，可以使用std::weak_ptr。
下面举一个简单的例子来说明std::shared_ptr的共享特性：

```
#include <iostream>
#include <string>
#include <memory>

class Person {
public:
    Person(const std::string& name) : name(name) {
```

```
        std::cout << "Constructing " << name << std::endl;
    }

    ~Person() {
        std::cout << "Destructing " << name << std::endl;
    }

    void SayHello() {
        std::cout << "Hello, my name is " << name << std::endl;
    }

private:
    std::string name;
};

int main() {
    // 创建一个std::shared_ptr来共享管理一个Person对象
    std::shared_ptr<Person> personPtr1 = std::make_shared<Person>("Alice");

    // 另一个std::shared_ptr也指向同一个Person对象
    std::shared_ptr<Person> personPtr2 = personPtr1;

    // 通过任意一个shared_ptr都可以操作Person对象
    personPtr1->SayHello();
    personPtr2->SayHello();

    // 此时，当所有的shared_ptr超出作用域时，Person对象的引用计数减为0，会自动销毁
    return 0;
}
```

在这个例子中，我们创建了两个std::shared_ptr，它们都指向同一个Person对象。当personPtr1和personPtr2同时拥有该对象时，该对象的引用计数为2。无论我们通过哪个std::shared_ptr来操作Person对象，都会正确地输出"Hello, my name is Alice"。当所有的std::shared_ptr超出作用域时（例如，当main()函数结束时），Person对象的引用计数减为0，会自动销毁，调用Person对象的析构函数输出"Destructing Alice"。这样，std::shared_ptr实现了多个指针共享拥有同一个资源的功能。

3）std::weak_ptr：

所有权：std::weak_ptr是弱引用指针，它不拥有对象的所有权。当最后一个std::shared_ptr指向对象时，即使有std::weak_ptr存在，对象也会被销毁。因此，std::weak_ptr不影响对象的生命周期。

引用计数：std::weak_ptr不参与引用计数，它仅充当了一个观察者的角色，用于检查std::shared_ptr是否还在管理对象。

解决循环引用：std::weak_ptr可以用于解决std::shared_ptr的循环引用问题。通过将循环引用中的某些std::shared_ptr替换为std::weak_ptr，打破循环引用，当最后一个std::shared_ptr被销毁时，对象可以正确释放。

总结：

std::unique_ptr适用于独占资源的场景，不涉及资源的共享和引用计数。

std::shared_ptr适用于多个智能指针共享同一个资源的场景，使用引用计数来管理资源的生命周期。

std::weak_ptr适用于解决std::shared_ptr的循环引用问题，以及需要观察资源是否存在的场景。它不影响资源的生命周期。

讲一下三种智能指针的初始化方式，以及引用计数器的变化

这里推荐各位阅读大丙老师的博客：大丙老师C++智能指针

讲一下std::make_shared 的用法

std::make_shared 是 C++ 标准库提供的一个函数模板，**用于方便地创建一个指定类型的 std::shared_ptr 智能指针。它可以避免直接使用 new 操作符来创建对象并手动管理智能指针，从而减少代码中的资源泄漏的风险。**
std::make_shared 函数的语法如下：
template <typename T, typename… Args>
std::shared_ptr make_shared(Args&&… args);

其中，T 是要创建的对象的类型，Args 是用于构造 T 对象的参数类型列表。可以用 Args&&… args 表示可以接受任意数量和类型的参数。
使用 std::make_shared 的优势是它将分配内存和对象的构造结合在一起，从而减少了额外的内存分配和构造函数调用开销，使得代码更加高效。
下面是一个使用 std::make_shared 的示例：

```
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass(int value) : value(value) {
        std::cout << "Constructing MyClass with value: " << value << std::endl;
    }

    ~MyClass() {
        std::cout << "Destructing MyClass with value: " << value << std::endl;
    }

```

```
    void PrintValue() {
        std::cout << "Value: " << value << std::endl;
    }

private:
    int value;
};

int main() {
    // 使用 std::make_shared 创建一个 MyClass 对象的 shared_ptr
    std::shared_ptr<MyClass> myPtr = std::make_shared<MyClass>(42);

    // 使用 shared_ptr 访问对象的成员函数
    myPtr->PrintValue();

    // 当 myPtr 所有者超出作用域时，对象会自动销毁，输出 Destructing MyClass with value: 42
    return 0;
}
```

在上面的例子中，我们使用 std::make_shared 创建了一个 MyClass 对象的 std::shared_ptr，该智能指针拥有这个对象，并在合适的时候自动释放内存。当 myPtr 超出作用域时，MyClass 对象会被自动销毁，输出 “Destructing MyClass with value: 42”。这样就避免了手动释放资源和内存泄漏的问题。

讲一下std::shared_ptr 的循环引用问题

循环引用（Circular Reference）是指两个或多个对象之间相互引用，形成一个环状结构。这种情况在使用智能指针（如std::shared_ptr）来管理对象生命周期时可能会引发问题，因为循环引用会导致资源无法正确地释放，从而造成内存泄漏。
让我们通过一个例子来说明循环引用的问题：
假设我们有两个类，Person 和 Car，每个类都包含一个指向另一个类对象的 std::shared_ptr，形成了循环引用：

```
#include <iostream>
#include <memory>

class Car;
class Person;

class Person {
public:
```

```cpp
    Person(const std::string& name) : name(name) {
        std::cout << "Constructing Person: " << name << std::endl;
    }

    ~Person() {
        std::cout << "Destructing Person: " << name << std::endl;
    }

    void SetCar(std::shared_ptr<Car> car) {
        ownedCar = car;
    }

private:
    std::string name;
    std::shared_ptr<Car> ownedCar;
};

class Car {
public:
    Car(const std::string& model) : model(model) {
        std::cout << "Constructing Car: " << model << std::endl;
    }

    ~Car() {
        std::cout << "Destructing Car: " << model << std::endl;
    }

    void SetOwner(std::shared_ptr<Person> person) {
        owner = person;
    }

private:
    std::string model;
    std::shared_ptr<Person> owner;
};

int main() {
    std::shared_ptr<Person> alice = std::make_shared<Person>("Alice");
    std::shared_ptr<Car> car = std::make_shared<Car>("Sedan");

    alice->SetCar(car);
    car->SetOwner(alice);

```

```
    // 在这个例子中，alice和car互相引用，形成了循环引用

    return 0;
}
```

在这个例子中，Person 和 Car 两个类相互引用，形成了循环引用。当 main() 函数结束时，alice 和 car 的引用计数都不会减到0，因为它们之间存在循环引用，导致它们指向的内存不会被释放，从而造成了内存泄漏。
为了避免循环引用带来的问题，可以使用std::weak_ptr 来打破循环引用，std::weak_ptr 不会增加对象的引用计数，而只是提供了一种观察资源是否存在的方式。这样当不再需要资源时，资源可以被正确地释放。

介绍一些C++11支持并发编程的库和函数

在C++11及以后的版本中，引入了原子操作、线程库和互斥量等机制，以支持更方便和安全的多线程编程。这些特性使得并发编程更加容易管理和避免常见的线程竞争问题。

1.原子操作：
原子操作是一种在多线程环境中进行共享变量的原子读写操作，确保在同一时刻只有一个线程能够访问该共享变量。C++中的原子操作可以通过std::atomic模板类来实现。例如，可以使用原子操作来实现计数器的安全自增操作：

```cpp
#include <iostream>
#include <atomic>
#include <thread>

std::atomic<int> counter = 0;

void incrementCounter() {
    for (int i = 0; i < 100000; ++i) {
        counter++; // 使用原子操作自增计数器
    }
}

int main() {
    std::thread t1(incrementCounter);
    std::thread t2(incrementCounter);

    t1.join();
    t2.join();
```

```
    std::cout << "Counter value: " << counter << std::endl;
    return 0;
}
```

在这个例子中，**由于counter变量是原子类型的**，所以它的自增操作是原子的，不会出现竞态条件。两个线程同时对counter变量进行操作，但是由于原子操作的保证，这些操作不会相互干扰，最终输出的结果应该是一个比较大的值（接近200000）。

2.线程库：
C++标准库提供了头文件，其中包含用于创建、管理和同步线程的相关类和函数。下面是一个简单的多线程例子，用于计算两个数的乘积：

```
#include <iostream>
#include <thread>

void multiply(int a, int b, int& result) {
    result = a * b;
}

int main() {
    int result = 0;
    int x = 5, y = 10;

    std::thread t(multiply, x, y, std::ref(result));
    t.join();

    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

当然linux一般还是用使用POSIX线程库（Pthreads）
3.互斥量：
互斥量（mutex）是一种用于保护共享资源的同步机制，它确保在同一时刻只有一个线程能够访问共享资源。C++中的std::mutex类提供了互斥量的实现。下面是一个使用互斥量保护共享资源的例子：

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;
int sharedValue = 0;

void incrementSharedValue() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(mtx); // 使用互斥量保护共享资源
        sharedValue++;
    }
}

int main() {
    std::thread t1(incrementSharedValue);
    std::thread t2(incrementSharedValue);

    t1.join();
    t2.join();

    std::cout << "Shared value: " << sharedValue << std::endl;
    return 0;
}
```

以上例子分别展示了原子操作、线程库和互斥量的用法。这些机制都能有效地帮助我们处理多线程编程中的并发问题，确保线程安全性并避免数据竞争。然而，在实际应用中，对于并发编程，还需要谨慎考虑多线程之间的协作和同步，以免产生死锁和其他并发问题。

介绍一下强类型枚举（Strongly Typed Enumerations）

强类型枚举是C++11引入的特性，它使得枚举类型更加类型安全和可控。在传统的C风格枚举中，枚举值被视为整数类型，可以隐式地转换为整数，这可能导致错误的用法和编程错误。强类型枚举通过限制枚举值的隐式转换，可以更好地防止潜在的错误。
以下是一个使用强类型枚举的简单示例：

```cpp
#include <iostream>

enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

```cpp

void printDay(Day day) {
    switch (day) {
        case Day::Monday:
            std::cout << "Monday" << std::endl;
            break;
        case Day::Tuesday:
            std::cout << "Tuesday" << std::endl;
            break;
        case Day::Wednesday:
            std::cout << "Wednesday" << std::endl;
            break;
        case Day::Thursday:
            std::cout << "Thursday" << std::endl;
            break;
        case Day::Friday:
            std::cout << "Friday" << std::endl;
            break;
        case Day::Saturday:
            std::cout << "Saturday" << std::endl;
            break;
        case Day::Sunday:
            std::cout << "Sunday" << std::endl;
            break;
    }
}

int main() {
    Day today = Day::Thursday;
    printDay(today);

    // 错误示例，无法隐式转换为整数
    // int dayInt = today;

    // 正确示例，需要显式转换为整数
    int dayInt = static_cast<int>(today);
    std::cout << "Day as integer: " << dayInt << std::endl;

    return 0;
}
```

在上面的示例中，Day是一个强类型枚举。在函数printDay中，我们使用了switch语句来打印不同的枚举值对应的字符串表示。在main函数中，我们声明了一个today变量，并将其赋值为Day::Thursday。这里Day::Thursday是一个具体的枚举值。
注意在错误示例中，我们试图将today直接赋值给dayInt，但由于强类型枚举的限制，无法直接隐式地将枚举值转换为整数。
在正确示例中，我们使用static_cast(today)将枚举值转换为整数，这是一种显式转换的方法。
使用强类型枚举可以防止意外的类型转换，增加了代码的可读性和类型安全性。它是C++中一个非常有用的特性。

介绍一下静态断言static_assert关键字

静态断言是C++11引入的特性，它使用 static_assert 关键字来在编译时进行断言检查，用于验证一些编译期常量表达式的真假。如果断言条件为真，则编译继续进行，如果条件为假，则会产生编译错误，并给出相应的错误消息。
下面是一个使用静态断言的简单示例：

```cpp
#include <iostream>

template <typename T, int Size>
class MyArray {
public:
    static_assert(Size > 0, "Size of the array must be greater than 0");

    MyArray() {
        // Some code here
    }

private:
    T data[Size];
};

int main() {
    // 使用静态断言创建 MyArray 对象，传入的 Size 是编译期常量表达式
    MyArray<int, 5> myArray; // 正确示例，Size 是大于 0 的常量表达式
    // MyArray<int, 0> myArray; // 错误示例，Size 是 0，静态断言会触发编译错误

    return 0;
}
```

在上面的示例中，我们定义了一个模板类 MyArray，它有两个模板参数 T 和 Size，其中 Size 是一个编译期常量表达式表示数组的大小。我们在类中使用了 static_assert 来检查 Size 是否大于 0。如果传入的 Size 不满足条件，就会触发编译错误，并输出错误消息 “Size of the array must be greater than 0”。
在 main 函数中，我们演示了正确示例和错误示例。MyArray<int, 5> 是一个正确示例，因为 Size 是大于 0 的常量表达式。而 MyArray<int, 0> 是一个错误示例，因为 Size 是 0，静态断言会触发编译错误。
静态断言在编译时进行检查，能够帮助开发人员在编译阶段尽早发现错误，提高代码的可靠性和稳定性。

介绍一下委托构造函数

委托构造函数是C++11引入的特性，它允许一个构造函数在初始化列表中调用同一个类的其他构造函数，从而避免了重复代码，提高了代码的可读性和维护性。
下面是一个使用委托构造函数的简单示例：

```cpp
#include <iostream>

class Person {
public:
    Person() : Person("Unknown", 0) {} // 委托构造函数

    Person(const std::string& name) : Person(name, 0) {} // 委托构造函数

    Person(const std::string& name, int age) : name(name), age(age) {}

    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }

private:
    std::string name;
    int age;
};

int main() {
    Person person1; // 使用无参数的构造函数
    person1.display();

    Person person2("Alice"); // 使用带一个参数的构造函数
    person2.display();
```

```
    Person person3("Bob", 30); // 使用带两个参数的构造函数
    person3.display();

    return 0;
}
```

在上面的示例中，我们定义了一个名为 Person 的类，它有三个构造函数：一个无参数构造函数，一个带一个参数的构造函数，以及一个带两个参数的构造函数。
在无参数构造函数中，我们使用了委托构造函数的方式来调用带两个参数的构造函数，并提供了默认参数值，这样在构造 Person 对象时，如果没有传入任何参数，就会调用这个无参数构造函数，实际上是通过委托调用带两个参数的构造函数。
在带一个参数的构造函数中，同样使用了委托构造函数的方式来调用带两个参数的构造函数，并将第二个参数设置为默认值 0。
在带两个参数的构造函数中，我们完成了实际的成员变量初始化工作。
在 main 函数中，我们演示了使用不同构造函数创建 Person 对象的方法，包括使用无参数、一个参数和两个参数的构造函数。
通过委托构造函数，我们可以在一个构造函数中调用其他构造函数，从而避免了重复的初始化代码，简化了构造函数的实现，并且更加灵活地创建对象。

介绍一下可变参数模板

可变参数模板是C++11引入的重要特性，它允许函数模板和类模板接受**任意数量和类型的参数**，从而更加灵活和通用化。
下面分别通过函数模板和类模板来举例介绍可变参数模板的用法：

函数模板示例：

```
#include <iostream>

// 使用递归展开可变参数的函数模板
void printValues() {} // 基本情况：没有参数时终止递归

template <typename T, typename... Args>
void printValues(const T& value, const Args&... args) {
    std::cout << value << " ";
    printValues(args...); // 递归调用，展开剩余的参数
}

int main() {
    printValues(1, 2.5, "Hello", 'c');
    return 0;
}
```

在上面的函数模板示例中，我们定义了一个名为 printValues 的函数模板。它使用递归展开可变参数的方式来实现任意数量和类型参数的打印功能。当没有参数时，递归终止。在递归调用中，我们打印当前参数的值，并通过 args... 来展开剩余的参数。
在 main 函数中，我们调用 printValues 分别传入整数 1，浮点数 2.5，字符串 “Hello” 和字符 ‘c’，输出结果为：1 2.5 Hello c。

类模板示例：

```cpp
#include <iostream>

template <typename... Args>
class Tuple {
public:
    Tuple(const Args&... args) : elements{args...} {}

    void print() const {
        printElements<Args...>(elements);
    }

private:
    template <typename T, typename... Rest>
    void printElements(const T& value, const Rest&... rest) const {
        std::cout << value << " ";
        printElements(rest...); // 递归调用，展开剩余的元素
    }

    void printElements() const {} // 基本情况：没有元素时终止递归

    std::tuple<Args...> elements;
};

int main() {
    Tuple<int, double, std::string> myTuple(42, 3.14, "Hello");
    myTuple.print();
    return 0;
}
```

在上面的类模板示例中，我们定义了一个名为 Tuple 的类模板，它允许接受任意数量和类型的参数。我们使用了递归展开参数的方式，在构造函数中将传入的参数存储在 std::tuple 中。

类模板中的 printElements 函数使用递归方式打印每个元素的值。当没有元素时，递归终止。
在 main 函数中，我们创建了一个 Tuple 对象，并传入整数 42，浮点数 3.14，以及字符串 “Hello”。然后调用 print 方法来打印存储在 Tuple 中的值，输出结果为：42 3.14 Hello。
通过可变参数模板，我们可以编写更通用、灵活的函数模板和类模板，能够接受不同数量和类型的参数，提高代码的复用性和适用性。

----------以下为设计模式部分----------

介绍一下常见的设计模式

设计模式是一套被广泛接受的解决特定问题的最佳实践。以下是几种常见的C++设计模式：

单例模式（Singleton Pattern）：确保一个类只有一个实例，并提供一个全局访问点来访问该实例。

工厂模式（Factory Pattern）：定义一个用于创建对象的接口，但将具体的对象创建延迟到子类中。

观察者模式（Observer Pattern）：定义了对象之间的一对多依赖关系，当一个对象状态发生改变时，其依赖的所有对象都会收到通知并自动更新。

适配器模式（Adapter Pattern）：将一个类的接口转换成客户端所期望的另一个接口，以解决接口不兼容的问题。

策略模式（Strategy Pattern）：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，使得算法可以独立于客户端而变化。

装饰器模式（Decorator Pattern）：动态地将责任附加到对象上，以扩展对象的功能，同时又不改变其接口。

模板方法模式（Template Method Pattern）：定义一个算法的骨架，将一些步骤延迟到子类中实现。

命令模式（Command Pattern）：将请求封装成对象，以便可以用不同的请求对客户进行参数化。

这只是一小部分常见的设计模式，还有很多其他的设计模式可以用于不同的情况。每种设计模式都有其特定的应用场景和优缺点

单例模式适合哪些场景?

单例模式适合以下场景：

资源共享：当一个类的实例需要在整个应用程序中共享时，可以使用单例模式。通过单例模式，可以确保只有一个实例存在，从而避免资源的浪费和冲突。

全局配置：在应用程序中，有些配置信息可能需要被多个对象访问，比如日志记录器的配置、数据库连接配置等。使用单例模式可以确保这些配置信息只需要加载一次，并且可以在任何地方访问。

管理共享资源：在多线程环境中，单例模式可以用于管理共享资源，比如线程池、数据库连接池等。通过单例模式，可以确保所有线程共享同一个资源池，避免资源竞争和浪费。

缓存：在需要缓存数据的场景中，单例模式可以用于管理缓存，确保缓存数据只有一个实例，避免内存占用过大。

日志记录器：在记录日志的场景中，单例模式可以用于管理日志记录器，确保所有的日志信息都被统一记录到同一个日志文件中。
任务队列：通过单例模式，可以确保任务队列只有一个实例，不会重复创建多个队列，避免了资源浪费。而且，在多线程环境下，由于单例模式只创建一个实例，也就免了多线程同时访问多个队列造成的资源竞争和冲突。
因此，在任务队列的场景下，单例模式是一个很好的设计选择，能够确保任务队列在整个应用程序中只有一个实例，并且能够被所有需要的线程共享和使用。这样可以简化任务调度和管理，提高代码的可维护性和可靠性。

介绍一下懒汉模式和饿汉模式

懒汉模式（Lazy Initialization）和饿汉模式（Eager Initialization）是单例模式的两种实现方式。
懒汉模式是指在需要获取单例实例时才进行初始化。具体实现方式是在类中定义一个私有的静态成员变量作为单例实例，然后提供一个公共的静态方法来获取该实例。在该方法中，首先检查实例是否已经被创建，如果没有则进行实例化。懒汉模式的特点是延迟加载，即只有在需要时才会创建实例。
饿汉模式是指在类加载时就进行初始化。具体实现方式是在类中定义一个私有的静态成员变量，并在类定义的同时直接进行实例化。然后提供一个公共的静态方法来获取该实例。饿汉模式的特点是立即加载，即在类加载时就会创建实例。
两种模式各有优缺点：

懒汉模式的优点是延迟加载，只有在需要时才会创建实例，节省了资源。缺点是在多线程环境下需要考虑线程安全问题，需要进行额外的同步处理。

饿汉模式的优点是简单直观，没有线程安全的问题。缺点是在程序启动时就会创建实例，可能会浪费一些资源。

选择使用哪种模式取决于具体的需求和场景。如果资源消耗较大，且不需要立即加载实例，可以选择懒汉模式。如果实例创建比较简单，且需要保证线程安全，可以选择饿汉模式。

一个常见的例子是创建一个日志记录器的单例。
懒汉模式的实现如下：

```
class Logger {
private:
    static Logger* instance;
    Logger() {} // 私有构造函数，防止外部实例化
    //单例模式很喜欢这样定义，私有部分放一个构造函数，再放一个类的指针
public:
    static Logger* getInstance() {
        if (instance == nullptr) {
            instance = new Logger();
        }
        return instance;
    }

    void log(const std::string& message) {
        // 日志记录逻辑
        std::cout << "Logging: " << message << std::endl;
```

```
    }
};

Logger* Logger::instance = nullptr; // 初始化为nullptr

int main() {
    Logger* logger = Logger::getInstance();
    logger->log("Hello, World!");

    return 0;
}
```

在懒汉模式中，**Logger类的实例在第一次调用getInstance()方法时才会被创建。**
（Logger* Logger::instance = nullptr;这里是静态成员的调用方法，不会的话可以复习一下前面的知识）
饿汉模式的实现如下：

```
class Logger {
private:
    static Logger* instance;
    Logger() {} // 私有构造函数，防止外部实例化

public:
    static Logger* getInstance() {
        return instance;
    }

    void log(const std::string& message) {
        // 日志记录逻辑
        std::cout << "Logging: " << message << std::endl;
    }
};

Logger* Logger::instance = new Logger(); // 在类定义时直接创建实例

int main() {
    Logger* logger = Logger::getInstance();
    logger->log("Hello, World!");

```

```
    return 0;
}
```

在饿汉模式中，**Logger类的实例在程序启动时就会被创建。**
无论是懒汉模式还是饿汉模式，都可以通过getInstance()方法获取Logger类的单例实例，并进行日志记录操作。

单例模式的构造函数通常被放在私有部分，以防止外部代码直接通过实例化来创建多个对象。

讲一下懒汉模式和饿汉模式的线程安全问题

懒汉模式存在线程安全的问题！：

懒汉模式是指在需要使用实例时才进行实例化。也就是说，实例在第一次调用getInstance()方法时才会被创建。**当多个线程同时调用getInstance()方法并发现实例尚未创建时，它们可能会同时进入实例的创建过程，导致创建多个实例，从而违反了单例模式的初衷**

饿汉模式无线程安全问题:

饿汉模式是指在类加载的时候就进行实例化。也就是说，实例在程序启动时就会被创建。
（可以这样理解：**实例在加载时就已经创建，不需要考虑线程安全性问题。 因为实例已经是全局变量了 所以后面的线程都不会再单独创建实例了，直接用就完事了**）

饿汉模式的实现相对简单，因为实例在加载时就已经创建，不需要考虑线程安全性问题。

饿汉模式在某种程度上提供了线程安全，因为实例在加载时就已经被创建，但是它可能造成资源浪费，因为即使程序在运行过程中没有使用这个实例，它也会一直被创建。

解决懒汉模式线程安全的方法有哪些?

1）最简单的懒汉模式实现是通过**双重检查锁定**(DCL)来保证线程安全性。即在第一次检查时判断实例是否已创建，如果没有，再通过加锁的方式创建实例，避免多个线程同时创建实例。
下面是个**手动加锁解锁**的例子：

```
#include <iostream>
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;

```

```cpp
    // 私有构造函数，防止从外部实例化对象
    Singleton() {}

public:
    // 获取单例实例的静态方法
    static Singleton* getInstance() {
        // 第一次检查，如果已经创建了实例，直接返回，避免获取锁
        if (instance == nullptr) {
            mtx.lock();
            // 第二次检查，防止在第一个线程获取锁之前，其他线程已经创建了实例
            if (instance == nullptr) {
                instance = new Singleton();
            }
            mtx.unlock();
        }
        return instance;
        // 函数结束时lock_guard的析构函数会自动解锁mtx
    }

    void showMessage() {
        std::cout << "Hello from Singleton!" << std::endl;
    }
};

// 初始化静态成员变量
Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;

int main() {
    Singleton* singleton1 = Singleton::getInstance();
    Singleton* singleton2 = Singleton::getInstance();

    if (singleton1 == singleton2) {
        std::cout << "Both pointers point to the same instance. Singleton is working." << std::endl;
    } else {
        std::cout << "Oops! Different instances. Singleton implementation is incorrect." << std::endl;
    }

    return 0;
```

在上面的示例中，getInstance() 方法使用了双重检查锁定的思想。第一次检查用于判断实例是否已经创建，如果已经创建了，就直接返回已有的实例，避免获取锁，提高性能。如果没有实例，就获取互斥锁，并在临界区内再次检查实例是否为空，然后创建新的实例。

由于手动加锁和解锁需要程序员明确管理锁的生命周期，容易出现遗漏或错误的加锁解锁操作，从而引入新的竞态条件或死锁。因此，更推荐使用RAII（资源获取即初始化）机制，例如C++中的std::lock_guard，来自动管理锁的加锁和解锁，减少程序员手动管理锁带来的问题。
下面是个例子：

```cpp
#include <iostream>
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;

    // 私有构造函数，防止从外部实例化对象
    Singleton() {}

public:
    // 获取单例实例的静态方法
    static Singleton* getInstance() {
        // 第一次检查，如果已经创建了实例，直接返回，避免获取锁
        if (instance == nullptr) {
            std::lock_guard<std::mutex> lock(mtx); // 获取互斥锁
            // 第二次检查，防止在第一个线程获取锁之前，其他线程已经创建了实例
            if (instance == nullptr) {
                instance = new Singleton();
            }// 这里的花括号结束lock_guard的作用域，它的析构函数会在这里自动解锁mtx
        }
        return instance;

    }

    void showMessage() {
        std::cout << "Hello from Singleton!" << std::endl;
    }
};
```

```cpp
// 初始化静态成员变量
Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;

int main() {
    Singleton* singleton1 = Singleton::getInstance();
    Singleton* singleton2 = Singleton::getInstance();

    if (singleton1 == singleton2) {
        std::cout << "Both pointers point to the same instance. Singleton is working." << std::endl;
    } else {
        std::cout << "Oops! Different instances. Singleton implementation is incorrect." << std::endl;
    }

    return 0;
}
```

std::lock_guardstd::mutex确实没有显式的解锁操作。它是C++标准库中提供的一种用于管理互斥锁的RAII（资源获取即初始化）类。
std::lock_guard的构造函数会自动锁定所管理的互斥锁，并在其析构函数中自动解锁。这样，当std::lock_guard对象离开其作用域时（比如**函数结束或代码块结束**），会自动调用析构函数，从而自动释放互斥锁。**这里是在instance = new Singleton()之后函数结束时（}花括号那里）就开始释放。**

然而，**懒汉模式的双重检查锁定并不是线程安全的。**在C++11之前，由于编译器和硬件的优化行为，**指令重排**可能导致在一个线程还没完成实例的创建和初始化，而另一个线程就已经获取到了未初始化的实例。因此，需要在双重检查锁定中使用适当的内存栅栏或特殊的指令来防止指令重排序。

2）在C++11及以后，可以使用原子操作来避免这个问题。
在C++11及以后，可以使用std::call_once或原子操作来避免这个问题。

懒汉模式的双重检查锁定为何不安全?

懒汉模式的双重检查锁定（Double-Checked Locking）在某些编程语言和编译器优化条件下可能是不安全的。主要原因是由于编译器对指令重排（Out-of-order Execution）和内存可见性的优化，可能导致在多线程环境下出现问题。
以下是导致双重检查锁定不安全的原因：

1）**指令重排：在现代计算机架构中，为了提高执行效率，编译器和处理器可能会对指令进行重排**，这种重排是不影响单线程执行结果的，但在多线程环境下可能会导致问题。例如，在双重检查锁定中，可能会先分配内存（在第一个检查之后）然后再进行实例化，这样在多线程环境下可能会导致一个线程获取到未完全初始化的实例。

下面是是大丙老师网站对指令重排的解释这里直接截个图图个方便哈哈：大丙老师网站

但是实际上 `m_taskQ = new TaskQueue;` 在执行过程中对应的机器指令可能会被重新排序。正常过程如下：

- 第一步：分配内存用于保存 TaskQueue 对象。
- 第二步：在分配的内存中构造一个 TaskQueue 对象（初始化内存）。
- 第三步：使用 m_taskQ 指针指向分配的内存。

但是被重新排序以后执行顺序可能会变成这样：

- 第一步：分配内存用于保存 TaskQueue 对象。
- 第二步：使用 m_taskQ 指针指向分配的内存。
- 第三步：在分配的内存中构造一个 TaskQueue 对象（初始化内存）。

这样重排序并不影响单线程的执行结果，但是在多线程中就会出问题。如果线程A按照第二种顺序执行机器指令，执行完前两步之后失去CPU时间片被挂起了，此时线程B在第3行处进行指针判断的时候 `m_taskQ` 指针是不为空的，但这个指针指向的内存却没有被初始化，最后线程 B 使用了一个没有被初始化的队列对象就出问题了（出现这种情况是概率问题，需要反复的大量测试问题才可能会出现）。

2）内存可见性：在多线程环境下，如果一个线程修改了共享的状态，其他线程可能无法立即看到这个修改，而是从各自的线程缓存中读取。这就可能导致在第二次检查时，一个线程看到了instance不为空（因为第一个线程已经创建了实例），但实际上实例还没有被完全初始化。

3）C++11之前的缺陷：在C++11之前，对于静态局部变量的初始化，不同的编译器有不同的实现，导致双重检查锁定的正确性在某些编译器上无法保证。

介绍一下工厂模式

工厂模式（Factory Pattern）是一种创建型设计模式，用于封装对象的创建过程。它提供了一种统一的接口来创建对象，而无需客户端代码直接关注对象的具体实现。**工厂模式可以将对象的实例化与客户端代码解耦，从而提高代码的可维护性和灵活性。**
工厂模式通常包括以下角色：

产品（Product）接口或基类：定义了工厂所创建的对象的通用接口。具体产品类将实现这个接口。

具体产品（Concrete Product）：实现了产品接口的具体对象，是工厂创建的实际产品。

工厂（Factory）接口或基类：声明了一个用于创建产品对象的工厂方法，返回的类型通常是产品接口或基类。

具体工厂（Concrete Factory）：实现了工厂接口，负责创建具体产品的对象。

工厂模式可以有不同的变体，包括简单工厂模式、工厂方法模式和抽象工厂模式。

简单工厂模式（Simple Factory Pattern）：
简单工厂模式并不是GoF（Gang of Four）所定义的23种设计模式之一，它只有一个具体的工厂类，根据传入的参数决定创建哪种具体产品。这种模式相对简单，适用于只有一个工厂类负责所有产品创建的情况。

工厂方法模式（Factory Method Pattern）：
工厂方法模式是GoF所定义的23种设计模式之一。它引入了工厂接口或基类，每个具体产品都有对应的工厂类，负责创建该产品。客户端代码通过调用工厂方法来创建产品，具体的产品创建逻辑由相应的工厂类实现。工厂方法模式适用于需要添加新产品时不需要修改现有客户端代码的情况。

抽象工厂模式（Abstract Factory Pattern）：
抽象工厂模式是GoF所定义的23种设计模式之一。它提供了一组相关或依赖的产品族，每个工厂类负责创建一整组产品。客户端代码通过使用抽象工厂接口，可以创建多个不同类型的产品。抽象工厂模式适用于需要创建一组相关产品的情况。

工厂模式的主要优点是将对象的创建和客户端代码分离，使得客户端代码不需要了解具体产品的实现细节。这样可以降低代码的耦合度，提高代码的可维护性和可扩展性。
示例（工厂方法模式）：

```cpp
#include <iostream>

// 产品接口
class Product {
public:
    virtual void use() = 0;
};

// 具体产品A
class ConcreteProductA : public Product {
public:
    void use() override {
        std::cout << "Using ConcreteProductA" << std::endl;
```

```
    }
};

// 具体产品B
class ConcreteProductB : public Product {
public:
    void use() override {
        std::cout << "Using ConcreteProductB" << std::endl;
    }
};

// 工厂接口
class Factory {
public:
    virtual Product* createProduct() = 0;
};

// 具体工厂A
class ConcreteFactoryA : public Factory {
public:
    Product* createProduct() override {
        return new ConcreteProductA();
    }
};

// 具体工厂B
class ConcreteFactoryB : public Factory {
public:
    Product* createProduct() override {
        return new ConcreteProductB();
    }
};

int main() {
    Factory* factoryA = new ConcreteFactoryA();
    Factory* factoryB = new ConcreteFactoryB();

    Product* productA = factoryA->createProduct();
    Product* productB = factoryB->createProduct();

    productA->use(); // Output: Using ConcreteProductA
    productB->use(); // Output: Using ConcreteProductB

```

```
    delete factoryA;
    delete factoryB;
    delete productA;
    delete productB;

    return 0;
}
```

在上述示例中，我们定义了一个产品接口 Product 和两个具体产品类 ConcreteProductA 和 ConcreteProductB，它们实现了产品接口。然后，我们定义了一个工厂接口 Factory 和两个具体工厂类 ConcreteFactoryA 和 ConcreteFactoryB，它们分别负责创建对应的具体产品。客户端代码通过调用工厂的方法来创建产品，从而实现了客户端代码与具体产品的解耦。

总结：

工厂模式是一种创建型设计模式，它通过提供一个统一的接口来创建对象，将对象的实例化过程与客户端代码分离，从而增加了代码的可维护性和灵活性。**工厂模式有多种变体，包括简单工厂模式、工厂方法模式和抽象工厂模式，适用于不同的场景需求。**

举例介绍一下简单工厂模式和工厂模式

简单工厂模式示例：

假设我们有一个几何图形类，它有两个子类：圆形和正方形。我们可以使用简单工厂模式来创建这些图形对象。

```
#include <iostream>

// 抽象图形类
class Shape {
public:
    virtual void draw() = 0;
};

// 圆形类
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

// 正方形类
```

```cpp
class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a square" << std::endl;
    }
};

// 简单工厂类
class ShapeFactory {
public:
    static Shape* createShape(const std::string& shapeType) {
        if (shapeType == "circle") {
            return new Circle();
        } else if (shapeType == "square") {
            return new Square();
        } else {
            throw std::invalid_argument("Invalid shape type");
        }
    }
};

// 客户端代码
int main() {
    Shape* circle = ShapeFactory::createShape("circle");
    Shape* square = ShapeFactory::createShape("square");

    circle->draw();  // 输出: Drawing a circle
    square->draw();  // 输出: Drawing a square

    delete circle;
    delete square;

    return 0;
}
```

在这个例子中，ShapeFactory 是一个简单工厂类，通过传入不同的参数来创建不同类型的图形对象。

工厂模式示例：

假设我们有一个手机制造工厂，它可以制造不同品牌的手机，包括苹果手机和小米手机。我们可以使用工厂模式来创建这些手机对象。

```cpp
#include <iostream>

// 抽象产品类
class Phone {
public:
    virtual void displayInfo() = 0;
};

// 具体产品类：苹果手机
class ApplePhone : public Phone {
public:
    void displayInfo() override {
        std::cout << "This is an Apple phone" << std::endl;
    }
};

// 具体产品类：小米手机
class XiaomiPhone : public Phone {
public:
    void displayInfo() override {
        std::cout << "This is a Xiaomi phone" << std::endl;
    }
};

// 抽象工厂类
class PhoneFactory {
public:
    virtual Phone* createPhone() = 0;
};

// 具体工厂类：苹果手机工厂
class ApplePhoneFactory : public PhoneFactory {
public:
    Phone* createPhone() override {
        return new ApplePhone();
    }
};

// 具体工厂类：小米手机工厂
```

```
class XiaomiPhoneFactory : public PhoneFactory {
public:
    Phone* createPhone() override {
        return new XiaomiPhone();
    }
};

// 客户端代码
int main() {
    PhoneFactory* appleFactory = new ApplePhoneFactory();
    PhoneFactory* xiaomiFactory = new XiaomiPhoneFactory();

    Phone* applePhone = appleFactory->createPhone();
    Phone* xiaomiPhone = xiaomiFactory->createPhone();

    applePhone->displayInfo();  // 输出: This is an Apple phone
    xiaomiPhone->displayInfo(); // 输出: This is a Xiaomi phone

    delete applePhone;
    delete xiaomiPhone;
    delete appleFactory;
    delete xiaomiFactory;

    return 0;
}
```

在这个例子中，PhoneFactory 是一个抽象工厂类，它定义了创建手机对象的接口，而 ApplePhoneFactory 和 XiaomiPhoneFactory 是具体工厂类，分别创建苹果手机和小米手机对象。这样，客户端代码只需要通过不同的工厂类来创建不同品牌的手机对象，无需直接与具体的手机类耦合。这符合了工厂模式的特点。
简单工厂类只有一个工厂，工厂类有多个工厂

举例介绍一下抽象工厂模式

下面是一个使用C++实现的抽象工厂模式的示例，展示了如何使用抽象工厂模式创建不同操作系统的GUI组件。

```
#include <iostream>

// 抽象产品接口
class Button {
```

```
public:
    virtual void paint() = 0;
};
class TextField {
public:
    virtual void paint() = 0;
};

// 具体产品类
class WindowsButton : public Button {
public:
    void paint() override {
        std::cout << "Windows Button" << std::endl;
    }
};

class WindowsTextField : public TextField {
public:
    void paint() override {
        std::cout << "Windows TextField" << std::endl;
    }
};

class MacButton : public Button {
public:
    void paint() override {
        std::cout << "Mac Button" << std::endl;
    }
};

class MacTextField : public TextField {
public:
    void paint() override {
        std::cout << "Mac TextField" << std::endl;
    }
};

// 抽象工厂接口
class GUIFactory {
public:
    virtual Button* createButton() = 0;
    virtual TextField* createTextField() = 0;
```

```
};

// 具体工厂类
class WindowsFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new WindowsButton();
    }

    TextField* createTextField() override {
        return new WindowsTextField();
    }
};

class MacFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new MacButton();
    }

    TextField* createTextField() override {
        return new MacTextField();
    }
};

// 客户端代码
int main() {
    GUIFactory* windowsFactory = new WindowsFactory();
    Button* windowsButton = windowsFactory->createButton();
    TextField* windowsTextField = windowsFactory->createTextField();

    windowsButton->paint();       // 输出: Windows Button
    windowsTextField->paint();    // 输出: Windows TextField

    GUIFactory* macFactory = new MacFactory();
    Button* macButton = macFactory->createButton();
    TextField* macTextField = macFactory->createTextField();

    macButton->paint();           // 输出: Mac Button
    macTextField->paint();        // 输出: Mac TextField

    delete windowsFactory;
    delete windowsButton;
```

```
    delete windowsTextField;
    delete macFactory;
    delete macButton;
    delete macTextField;

    return 0;
}
```

在这个示例中，我们首先定义了两个抽象产品接口 Button 和 TextField，它们都包含一个纯虚函数 paint()。然后我们创建了具体产品类 WindowsButton、WindowsTextField、MacButton 和 MacTextField，它们分别继承自抽象产品接口并实现了 paint() 方法。
接着，我们定义了一个抽象工厂接口 GUIFactory，其中包含两个纯虚函数 createButton() 和 createTextField()，用于创建不同操作系统的GUI组件。然后我们创建了具体工厂类 WindowsFactory 和 MacFactory，它们分别继承自抽象工厂接口并实现了对应的方法，用于创建不同操作系统的GUI组件对象。
最后，客户端代码通过具体工厂来创建并使用不同操作系统的GUI组件对象，而无需关心具体的对象创建过程。这使得客户端代码与具体产品的实现解耦，并且可以轻松切换不同的工厂以创建不同的产品组合。
抽象工厂模式适用于需要创建一系列相关或相互依赖的对象，且客户端代码不应直接依赖于具体产品类的情况。它提供了一种灵活的方式来创建不同产品族的对象，同时保持了客户端代码与具体产品的解耦。

讲一下三种工厂模式的区别

工厂模式是一种创建对象的设计模式，它将对象的创建与使用分离，通过一个工厂类来创建对象，从而降低了系统的耦合性，增加了系统的灵活性和可维护性。在工厂模式中，有三种常见的变体：简单工厂模式、工厂方法模式和抽象工厂模式。它们之间的区别如下：

简单工厂模式：
简单工厂模式是最简单的工厂模式，它由一个工厂类来负责创建所有产品的实例。客户端通过向工厂类传递不同的参数，来获取不同的产品实例。简单工厂模式实现了对象的创建和使用的分离，但是如果要新增产品，需要修改工厂类的代码，**不符合开闭原则。**

工厂方法模式：
工厂方法模式是简单工厂模式的扩展，它定义了一个创建对象的接口，但由具体的子类来实现创建对象的方法。每个具体的子工厂类负责创建一种产品，客户端可以通过选择不同的工厂类来创建不同的产品实例。工厂方法模式符合开闭原则，但需要为每个产品定义一个具体的工厂类，增加了类的个数。

抽象工厂模式：
抽象工厂模式是工厂方法模式的扩展，它定义了一个创建一系列相关或相互依赖对象的接口，而不需要指定具体的类。抽象工厂模式包含多个工厂方法，每个工厂方法负责创建一个系列的产品。客户端通过选择不同的抽象工厂来创建不同系列的产品。抽象工厂模式能够创建一组相关的产品，但难以支持新种类产品的增加。

总结：

简单工厂模式适用于创建的对象较少且不经常变化的情况，但不符合开闭原则。

工厂方法模式适用于创建的对象有多个类型，且需要支持新增类型的情况。

抽象工厂模式适用于创建一系列相关对象，但难以支持新增产品种类的情况。

如果看不懂 可以参考大丙老师的博客：三种工厂模式的区别

什么是开闭原则?

开闭原则（Open-Closed Principle，OCP）是面向对象设计中的一个重要原则，由著名的计算机科学家Bertrand Meyer于1988年提出。它是SOLID原则中的一部分，SOLID是面向对象设计中的五个基本原则之一。
开闭原则的定义如下：
“软件实体（类、模块、函数等）应该对扩展开放，对修改关闭。”
简而言之，开闭原则要求一个软件实体在需要改变其行为时，不应该修改其源代码。而应该通过扩展该实体，添加新的代码来实现新的行为，从而保持原有代码的稳定性和可复用性。
开闭原则的目标是尽量减少系统的维护和修改，从而降低引入新功能时的风险。通过遵循开闭原则，可以使软件系统更加稳定、灵活、可扩展，并且更易于维护和升级。
遵循开闭原则的一种常见方法是使用抽象类、接口和多态性。通过定义抽象类或接口，然后派生具体的子类来实现不同的行为。在需要新增功能时，只需要增加新的子类，而不需要修改原有的代码。
开闭原则是面向对象设计的基石之一，它在软件设计和架构中起到重要的指导作用。符合开闭原则的设计可以使系统更加健壮、可维护和可扩展。