第3课 C++11/14/17 常考新特性

C++ 编译过程与原理

C++ 编译过程通常分为以下几个基本阶段：

1. **预处理（Preprocessing）**
   - 在编译源代码之前，预处理器首先对源文件进行操作。这个阶段主要包括：
     - **宏定义展开**：使用 `#define` 定义的宏会被替换为其内容。
     - **处理头文件**：`#include` 指令会将指定头文件的内容插入到源代码中。
     - **删除注释**：预处理器会移除所有注释内容，以简化后续处理。

示例：

```cpp
#define PI 3.14
#include <iostream>

int main() {
    std::cout << "圆周率是: " << PI << std::endl;
    return 0;
}
```

预处理后，代码变为：

```cpp
#include <iostream>

int main() {
    std::cout << "圆周率是: " << 3.14 << std::endl;
    return 0;
}
```

M学长的考研OP帮

2. **词法分析（Lexical Analysis）**
   - 预处理后的文本被编译器分解成一系列符号或标记（tokens）。这些tokens包括关键字、标识符、常量、运算符和分隔符等。
   - **示例**：对于语句 `int a = 5;`，会分解为以下tokens：`int`、`a`、`=`、`5` 和 `;`。

3. **语法分析（Syntax Analysis 或 Parsing）**
   - 编译器根据C++的语法规则，将tokens组合成结构化的数据结构，即抽象语法树（AST）。
   - **示例**：

```
int main() {
    int x = 5;
    int y = 3;
    int z = x + y;
    return z;
}
```

AST示例（简化描述）：

```
Program
|
+-- FunctionDeclaration (int main())
    |
    +-- CompoundStatement
        |
        +-- VariableDeclaration (int x)
        |   +-- IntegerLiteral (5)
        |
        +-- VariableDeclaration (int y)
        |   +-- IntegerLiteral (3)
        |
        +-- VariableDeclaration (int z)
            +-- BinaryExpression (+)
                +-- Identifier (x)
                +-- Identifier (y)
        |
        +-- ReturnStatement
            +-- Identifier (z)
```

M学长的考研top帮

为什么AST方便编译器分析和优化：

结构化表示

- AST以树状结构清晰展示程序的嵌套和层级，如函数、语句块和变量。
- 帮助编译器快速理解代码的整体架构和各部分关系。

便于遍历和操作

- 编译器可以高效地遍历AST的每个节点，进行分析和修改。
- 例如，轻松定位并优化特定的变量或表达式。

抽象化细节

- AST去除了具体的语法符号（如分号、括号），专注于代码的逻辑结构。
- 简化了分析过程，使编译器能更专注于优化逻辑和数据流。

支持语义分析

- 基于AST，编译器可以执行类型检查、作用域解析等，确保代码正确性。
- 例如，验证变量是否正确声明和初始化。

优化机会

- AST使编译器能应用各种优化，如常量折叠和死代码消除。
- 例如，将z = 5 + 3优化为z = 8，减少运行时计算。

代码生成的基础

- 编译器根据AST生成中间代码或机器代码，确保逻辑一致。
- AST提供所有必要的信息，支持准确的代码生成。

4. **语义分析（Semantic Analysis）**
   - 在构建AST的同时，编译器进行语义检查，以确保所有变量和函数的声明与使用符合语义规则。
   - 包括类型检查、作用域解析和其他静态语义规则的验证。例如，如果尝试将字符串赋值给整型变量，编译器将在这一阶段报告错误。

5. **中间代码生成（Intermediate Code Generation）**
   - 编译器生成一种与特定机器无关的中间表示形式，如字节码或三地址码。这种形式便于后续的优化和跨平台移植。
   - **示例**：将 `int z = x + y;` 转换为中间代码可能类似于：

```
t1 = x + y
z = t1
```

考研帮

6. **优化（Optimization）**
   - 编译器对中间代码进行各种优化，以提高运行效率和减少资源消耗。
   - **优化示例**：
     - **循环优化**：减少循环内重复计算的次数。
     - **死代码删除**：移除永远不会执行的代码。
     - **常量传播**：将常量值直接替换到表达式中。

7. **目标代码生成（Code Generation）**
   - 编译器将优化后的中间代码转换为特定计算机架构的目标代码（Object Code）。
   - 目标代码通常为汇编语言或直接是二进制格式，能被CPU执行。例如，中间代码可能转换为如下汇编语言：

```
mov eax, x
add eax, y
mov z, eax
```

8. **链接（Linking）**
   - 最后，链接器将多个目标代码文件及必要的库文件合并，形成最终的可执行文件（Executable）。
   - 链接过程中，链接器会解决符号引用，将函数调用与其实现绑定，并确定全局变量的存储位置。

示例：

- 如果有两个源文件 `main.cpp` 和 `utils.cpp`，它们都有函数 `void foo()`，链接器会将 `main.cpp` 中的 `foo()` 调用与 `utils.cpp` 中的实现关联起来。

C++11/14/17新特性

M学长的考研C0P帮

自动类型推导 (auto)

C++11 引入了 `auto` 关键字，允许编译器基于变量的初始化表达式自动推导其类型。

实现原理：

- 编译时，编译器根据变量的初始化表达式推导出变量的精确类型。

优势：（这里不用记，你就想他自动了，肯定就方便高效了）

- 简化代码，提高开发效率。
- 减少因类型错误导致的编译错误。
- 适用于迭代器和复杂函数返回类型的场景。

解决的问题：

- 减少重复类型声明，使代码更易于维护。

使用前后对比示例代码：

使用前：

```
std::vector<int> vec = {1, 2, 3, 4};
std::vector<int>::iterator it = vec.begin(); // 需要明确指定迭代器的类型
```

使用后：

```
std::vector<int> vec = {1, 2, 3, 4};
auto it = vec.begin(); // 自动推导迭代器类型
```

M学长的考研top帮

Lambda 表达式

Lambda 表达式是 C++11 中引入的，用于定义匿名函数对象。在讲Lambda之前我们先说一下重载的概念

重载

C++中的重载（Overloading）是指在同一作用域内，可以定义多个同名函数，但这些函数的参数列表必须不同。这里的“不同”指的是参数的数量、类型或者顺序的不同，而不是仅仅返回值类型的不同。通过函数重载，可以根据传入参数的不同自动调用相应的函数版本，使得代码更具有可读性和灵活性。

（上面看着晕这里结合代码看看）

```cpp
#include <iostream>

// 重载函数 Eat
void Eat() {
    std::cout << "I am eating something." << std::endl;
}

// 参数类型为整数时的Eat
void Eat(int quantity) {
    std::cout << "I am eating " << quantity << " pieces of food." << std::endl;
}

// 参数类型为字符串时的Eat
void Eat(const std::string& dishName) {
    std::cout << "I am eating " << dishName << "." << std::endl;
}

int main() {
    // 调用无参数版本的Eat
    Eat();

    // 调用接受整数参数的Eat
    Eat(3);

    // 调用接受字符串参数的Eat
    Eat("apple");

    return 0;
}
```

小学长的考研top帮

在上述例子中，我们定义了三个名为 `Eat` 的函数，它们因为参数列表不同而被编译器区分开来，这样在调用时可以根据实际传递给函数的参数类型和数量确定调用哪个函数实现。这就是C++中函数重载的概念与应用。此外，C++还支持运算符重载，即对预定义的运算符赋予用户自定义类型上的特定行为，这也是通过重载机制实现的。

回调函数

回调函数是一种编程设计模式，它允许你在执行一个操作时传递一个函数（或方法）作为参数，然后在特定的时机或条件满足时，由被调用方来调用这个传递进来的函数。简单来说，回调函数就是“在未来某个时刻会被调用的函数”。

什么是Lambda表达式?

Lambda表达式是一种轻量级的、匿名的函数对象，允许你在需要函数的地方内联定义和使用函数逻辑。它们通常用于需要临时、小型函数的场景，如算法中的回调、事件处理等。

Lambda表达式的基本语法

```
[capture](parameters) -> return_type {
    // 函数体
};
```

- **`capture`**：指定Lambda捕获外部变量的方式，可以是值捕获、引用捕获或混合捕获。
- **`parameters`**：函数参数列表，与普通函数类似。
- **`return_type`**（可选）：指定返回类型，编译器通常可以自动推导。
- **`函数体`**：Lambda执行的代码块。

示例

```
#include <iostream>
#include <vector>
#include <algorithm>

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

```

```cpp
    // 使用Lambda表达式打印每个元素
    std::for_each(numbers.begin(), numbers.end(), [](int n) {
        std::cout << n << " ";
    });
    std::cout << std::endl;

    // 使用Lambda表达式计算总和
    int sum = 0;
    std::for_each(numbers.begin(), numbers.end(), [&](int n) {
        sum += n;
    });
    std::cout << "总和: " << sum << std::endl;

    return 0;
}
```

输出：

```
1 2 3 4 5
总和: 15
```

Lambda表达式的优势

1. **简洁性**：无需为简单操作创建单独的函数，减少代码量。
2. **灵活性**：可以在定义时捕获外部变量，适应不同的上下文需求。
3. **可读性**：将操作逻辑直接嵌入使用场景，增强代码的可读性和维护性。
4. **内联优化**：编译器更容易优化内联，提高性能。

实现原理

1. 匿名函数对象：

IT学长的考研top帮

- Lambda表达式允许你创建一个没有名称的函数。例如：`[](int x, int y) { return x + y; }` 这就是一个简单的加法lambda表达式。
- 编译器在背后做的事情非常巧妙，它会根据lambda表达式的结构生成一个编译期临时的、未命名的类（通常称为闭包类或lambda类），这个类包含了一个重载了调用运算符 `operator()` 的成员函数，这样当你像调用函数一样使用lambda表达式的实例时，实际上就是在调用这个类的 `operator()` 方法。

2. 闭包（Closure）：（这块看完下面代码再回来看看）
   - 闭包是指Lambda表达式能够“记住”它所在上下文中的变量，并能在自己的执行环境中访问它们的能力。
   - 捕获外部变量的方式有多种：
     - 值捕获（`[=]`）：将外部变量以值方式复制到lambda内部，修改lambda内的副本不影响外部变量。
     - 引用捕获（`[&]`）：通过引用捕获外部变量，对lambda内变量的修改会影响到外部对应的变量。
     - 显式捕获列表（如 `[x, &y]`）：可以混合使用值和引用捕获特定的变量。
     - 隐式捕获（`[]`）：默认不捕获任何外部变量，但如果在lambda体中使用了外部作用域的非静态局部变量，则需要显式声明捕获方式。
   - 值捕获用于将外部作用域内的变量“携带”进lambda内部，而传参则是为了接收在调用lambda函数时提供的数据。两者共同决定了lambda如何访问和操作其内外的数据。
3. 类型推导：
   - 在lambda表达式中，编译器可以根据函数体的逻辑自动推断出返回类型，从而省去了显式指定返回类型的步骤，这使得代码更为简洁。
   - 如果lambda体中包含return语句或者表达式可以唯一确定类型，编译器就能完成类型推导。
   - 若需要明确指定返回类型，可以在参数列表后使用 `-> 返回类型` 来显式指明。

基本结构

- **捕获列表（Capture List）**：捕获列表使用方括号 [] 括起来，用于指定Lambda表达式可以访问的外部变量。捕获列表有以下几种形式：
  - `[ ]`：不捕获任何外部变量。
  - `[=]`：以值捕获方式捕获所有外部变量。
  - `[&]`：以引用捕获方式捕获所有外部变量。
  - `[a, &b]`：以值捕获变量a，以引用捕获变量b，可以同时使用多个变量。

- **参数列表（Parameter List）**：参数列表使用小括号 () 括起来，类似于普通函数的参数列表，用于定义Lambda表达式的参数，如果没有参数可以留空。

- **函数体（Function Body）**：函数体使用花括号 {} 括起来，包含Lambda表达式的具体实现代码，就像普通函数的函数体一样。

示例解释

以下是一个Lambda表达式的示例：

```
[](int n) { return n % 2 == 0; }
```

- 捕获列表 `[]` 表示不捕获任何外部变量。
- 参数列表 `(int n)` 表示Lambda表达式接受一个整数参数n。
- 函数体 `{ return n % 2 == 0; }` 实现了一个简单的逻辑，检查n是否为偶数。

这个Lambda表达式等效于以下的普通函数：

```
bool isEven(int n) {
    return n % 2 == 0;
}
```

Lambda表达式的优势在于它可以在需要的地方直接定义和使用，无需预先声明一个单独的函数，从而使代码更加紧凑和易于理解。Lambda表达式在C++中特别适用于作为回调函数传递给算法或其他函数，或者用于定义短小的行为片段，提高了代码的灵活性和可读性。

使用前后对比示例代码：

使用前：

```
bool descending(int a, int b) {
    return a > b;
}
std::vector<int> vec = {1, 2, 3, 4};
std::sort(vec.begin(), vec.end(), descending); // 需要单独定义比较函数
```

Mr长的考研top帮

使用后：

```
std::vector<int> vec = {1, 2, 3, 4};
std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; }); // 直接使用 Lambda 表达式
```

优势

1. 简洁性：Lambda 表达式提供了一种非常简洁的方式来定义一个函数对象。使用 Lambda 表达式，你可以在需要的地方直接编写函数逻辑，而不需要单独定义一个函数。
2. 方便的闭包：Lambda 允许你方便地捕获外部变量，使得你可以在函数体内使用这些变量，而无需显式地将它们作为参数传递。
3. 易于使用的回调：由于上面的原因，Lambda 表达式特别适用于定义回调函数，比如作为参数传递给算法或者异步操作的处理器。
4. 增强的可读性和维护性：由于 Lambda 表达式通常是紧凑的，它们可以使得代码更易于阅读和维护。特别是在使用 STL 算法或者需要定义小的行为片段时。

这里补充一下上面提到的回调函数

回调函数是一种在特定事件或条件发生时，由一个函数调用另一个预先定义好的函数的设计模式。在程序设计中，回调函数通常作为参数传递给另一个函数，这个接收回调函数的函数会在适当的时间点调用它。这样做的好处是可以实现模块之间的解耦和灵活的事件处理机制。

举个简单的例子来理解：

```
// 假设有一个函数用于异步读取文件内容
void readFile(const char* filename, void (*callback)(const std::string& content));

// 定义一个处理读取到内容的回调函数
void handleFileContent(const std::string& content) {
    std::cout << "File content: " << content << std::endl;
}

// 使用回调函数读取文件
readFile("example.txt", handleFileContent);

// 在readFile内部完成文件读取后，会调用传入的handleFileContent函数处理内容
```

在这个例子中：

- `readFile` 是一个接受回调函数作为参数的函数。
- `handleFileContent` 是被传入的回调函数，它的作用是在读取文件操作完成后处理文件内容。
- 当 `readFile` 完成文件读取任务后，它将调用 `handleFileContent` 函数，传入读取到的内容。

通过这种方式， `readFile` 不需要知道如何具体处理文件内容，只需专注于执行读取操作，而具体的处理逻辑则交给了回调函数。这就是回调函数的核心思想：把部分功能委托给使用者自定义的函数，实现了低耦合、高扩展性。

智能指针

智能指针是模板类，用于自动管理内存。

普通指针

在C++编程语言中，指针是一个变量，它的值是另一个变量（通常是对象或数组）的内存地址。换句话说，指针存储的是内存中某个数据的地址，而不是数据本身。

例如，如果你声明一个整型变量 `int x = 10;` ，那么可以创建一个指向这个整数变量的指针：

```
int x = 10;
int* ptr = &x; // ptr 现在指向变量 x 的内存地址
```

在这里， `ptr` 就是一个指针变量，使用 `&` 运算符获取了变量 `x` 的地址并将其赋值给 `ptr` 。通过解引用指针（使用 `*` 运算符），我们可以访问或者修改该地址所对应的变量的值：

```
std::cout << *ptr; // 输出：10，因为 *ptr 指向的是 x 的值
*ptr = 20;         // 修改指针指向的内容，现在 x 的值变为 20
```

M学长的考研top学

指针的主要作用包括：

- 访问和操作内存中的数据。
- 动态内存管理，如使用 `new` 和 `delete` 创建和释放动态分配的对象。
- 实现函数之间的参数传递，尤其是对于大型数据结构，通过传递指针可提高效率，避免复制整个对象。
- 实现复杂的数据结构，如链表、树等，其中每个节点通常包含指向下一个节点或其他相关节点的指针。

然而，由于指针直接与内存打交道，如果不正确地使用它们可能会导致各种问题，比如内存泄漏、野指针引用、空指针解引用等错误。因此，理解并谨慎处理指针是C++编程中的关键技能之一。

原始指针的问题

在C++中，原始指针（raw pointers）用于动态分配内存并跟踪对象的生命周期。但原始指针存在一些问题：

1. 内存泄漏：如果忘记释放动态分配的内存，将导致内存泄漏，使程序占用的内存不断增加。
2. 悬挂指针：当指向已被释放的内存或已销毁的对象时，原始指针可能变成悬挂指针，访问它们会导致未定义的行为。（也叫野指针）
   a. 空指针与野指针不同，空指针是一个已初始化的指针变量，其值为零或nullptr
3. 难以维护：手动管理内存需要仔细考虑对象的生命周期，容易出错，使代码变得难以维护。

实现原理

- `std::unique_ptr` 是一种独占拥有权的智能指针，**只能有一个 `unique_ptr` 指向对象**。其原理包括：
  - 禁止复制构造和赋值：**`unique_ptr` 不允许复制构造和赋值，因此它只能有一个所有**权。
  - 移动语义：通过移动语义，`unique_ptr` 可以将所有权从一个指针转移到另一个，使得资源的管理更高效。

这种机制确保了对象的独占拥有权，避免了资源的重复释放和多个指针同时指向一个对象的问题。

- `std::shared_ptr` 是一种共享拥有权的智能指针，**多个 `shared_ptr` 可以指向相同的对象**。它的原理包括：

- 内部引用计数：`shared_ptr` 内部维护一个引用计数，记录有多少个 `shared_ptr` 共享同一对象。
- 拥有权：当一个 `shared_ptr` 指向一个对象时，引用计数增加。当 `shared_ptr` 超出作用域或被显式重置时，引用计数减少。当引用计数为零时，对象被自动释放。

这种机制确保了对象的生命周期与 `shared_ptr` 的生命周期一致，避免了内存泄漏和悬挂指针的问题。

`shared_ptr` **存在循环引用的问题**，具体是指两个或多个智能指针相互指向对方，导致它们的引用计数器永远无法归零，因此即使程序不再需要这些对象，它们也无法被正确地释放。这是因为每个 `shared_ptr` 都会维护一个引用计数器，每次创建新的 `shared_ptr` 指向同一个对象时，该对象的引用计数增加；当 `shared_ptr` 被销毁或者其指向其他对象时，引用计数减少。

- `std::weak_ptr`：**配合 `shared_ptr` 使用，作为观察者，不影响引用计数。解决了循环引用的问题**

优势

1. 自动内存管理：智能指针自动管理内存，避免了内存泄漏和悬挂指针的问题。
2. 方便且安全：使用智能指针可以更方便地处理对象的生命周期，使代码更安全，减少错误。
3. 资源管理：智能指针不仅可以用于内存管理，还可以用于管理其他资源，如文件句柄。
4. 多线程安全：`std::shared_ptr` 提供了多线程安全的引用计数，适用于多线程环境。

使用前后对比示例代码

使用 `unique_ptr` 前：

```
int* p = new int(10);
// ... 使用 p
delete p; // 必须手动删除
```

使用 `unique_ptr` 后：

```
std::unique_ptr<int> up(new int(10)); // 自动管理内存
// 当 up
```

```
3
4   离开作用域时，它指向的内存会被自动释放
```

使用 `shared_ptr` 前：

```
1 int* p = new int(10);
2 int* q = p; // p 和 q 共享同一个资源
3 // ... 使用 p 和 q
4 delete p; // p 被删除
5 q = nullptr; // 必须手动处理 q
```

使用 `shared_ptr` 后：

```
1 std::shared_ptr<int> sp1 = std::make_shared<int>(10);
2 std::shared_ptr<int> sp2 = sp1; // sp1 和 sp2 自动共享资源
3 // 当最后一个 shared_ptr 离开作用域时，资源自动释放
```

但实际上，在追求性能的场景（比如低延迟服务器里）还是会使用传统指针，因为智能指针的“智能”要求了更多资源（比如引用技术）（可以老师问到的时候装一下）

其他新特性list（了解即可，背一下加粗的部分，具体原理感兴趣可以查一下，问到的概率不高）

C++11 新特性

1. **范围for循环（Range-based for loop）：允许方便地遍历容器。**
2. **右值引用（Rvalue References）和移动语义（Move Semantics）**：允许资源的转移而非拷贝，提高性能。
3. **初始化列表（Initializer lists）**：用于容器和对象的统一初始化方式。
4. **线程支持库（Threading Library）**：支持多线程编程。
5. **原子操作库（Atomic Operations Library）**：提供了原子操作的支持，用于多线程中的数据同步。
6. **正则表达式库（Regular Expressions Library）**：用于字符串模式匹配和搜索。

7. **类型推导** `decltype`：用于获取表达式的类型。
8. **用户定义字面量（User-defined literals）**：允许给字面量自定义解释。
9. **静态断言（Static assertions）**：编译时断言。

C++14 新特性

1. **返回类型推导（Return type deduction）**：函数返回类型可以用 `auto` 关键字自动推导。
2. **泛型Lambda表达式（Generic lambdas）**：Lambda表达式可以使用 `auto` 在参数中实现参数类型推导。
3. **二进制字面量（Binary literals）**：支持二进制数表示。
4. **数字分隔符（Digit separators）**：可以使用单引号作为数字字面量的分隔符，提高可读性。

C++17 新特性

1. **结构化绑定（Structured bindings）**：允许将对象或数组的多个成员绑定到一组变量。
2. **编译时if（constexpr if）**：允许在编译时进行条件编译。
3. **字符串视图（std::string_view）**：提供了对字符串的轻量级非拥有引用。
4. **文件系统库（Filesystem library）**：提供了对文件系统的操作能力。
5. **并行算法（Parallel algorithms）**：标准库算法的并行版本，用于提高性能。
6. **可选类型（std::optional）**：表示可能不存在的值。
7. **任意类型（std::any）**：安全地存储任意类型的值。
8. **变体类型（std::variant）**：存储并访问固定集合中的任意类型。

面试常问问题

自动类型推导（`auto`）相关问题

Q1: `auto` 的作用是什么？在什么情况下应该使用 `auto`？

A: `auto` 是 C++ 提供的类型推导关键字，可以让编译器自动推导变量的类型，减少冗长的类型声明。适合在以下场景使用：

- 变量的类型较长且复杂，如迭代器、模板类型等。
- 避免重复声明变量的类型，提高代码可读性。
- 当类型明显且不容易引起混淆时（比如用 `auto` 初始化一个返回值类型复杂的表达式）。

Q2: 什么时候不应该使用 `auto`？

A：尽量避免在以下情况使用 `auto`：

- **类型不清晰**：如果 `auto` 隐藏了类型信息，可能导致代码可读性差。

- **类型推导不符合预期**：例如，某些表达式返回 `const` 类型，而你希望保持原始的可变类型。

- **引用类型**：在使用引用类型时，`auto` 会丢弃 `const` 和引用修饰符，导致错误行为。

```
const int x = 5;
auto y = x; // y 的类型是 int 而非 const int
y = 10;     // 没有错误，因为 y 不是 const
```

智能指针相关问题

Q1: 为什么要使用智能指针，而不是原生指针？

A：智能指针自动管理内存，避免了手动 `delete`，减少了内存泄漏、悬空指针和双重释放等问题。在 C++ 中，常用的智能指针有 `std::unique_ptr`、`std::shared_ptr` 和 `std::weak_ptr`。

Q2: `std::unique_ptr` 和 `std::shared_ptr` 有什么区别？

A：

- **`std::unique_ptr`**：独占资源的所有权，不能复制，只能转移（`std::move`）。适合需要唯一所有权的场景。

- **`std::shared_ptr`**：多个 `shared_ptr` 可以共享资源的所有权，内部有引用计数，当计数为 0 时释放资源。适合需要共享资源的场景。

M学长的考研top帮

Q3: 为什么需要 `std::weak_ptr`？

A: `std::weak_ptr` 用于避免 `std::shared_ptr` 之间的**循环引用**。当两个对象互相持有 `shared_ptr` 指向对方时，会导致引用计数无法归零，导致内存泄漏。`weak_ptr` 不增加引用计数，提供了一种对资源的**弱引用**。

```
#include <memory>

struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev; // 避免循环引用
};
```

其他问题

Q1: `nullptr` 和 `NULL` 有什么区别？为什么 C++11 中引入了 `nullptr`

A:

`nullptr` 和 `NULL` 的主要区别在于**类型安全**和**表达的语义**：

- **`NULL` 是 `0` 的宏**：

在 C++ 中，`NULL` 通常被定义为整数 `0`，这意味着 `NULL` 是一个**整数常量**。所以，`NULL` 可以被隐式转换为任何指针类型、整数类型，甚至可以导致类型不明确的问题。

```
#define NULL 0
```

- **`nullptr` 是一个指针类型**：

在 C++11 中引入的 `nullptr` 是一个新的关键字，表示一个**空指针常量**。它有一种特殊的类型，叫做 **`std::nullptr_t`**，可以隐式转换为任意指针或成员指针类型，但不能转换为整数类型。这让 `nullptr` 更加类型安全，避免了 `NULL` 带来的歧义。

引入 `nullptr` 的原因：

学长的考研top帮

- **类型安全**：`nullptr` 不会被转换为整数类型，减少了因类型不匹配而引发的错误。

- **可读性**：`nullptr` 明确表示这是一个指针，`NULL` 在不同环境中可能会有不同的定义，有时会造成困惑。

示例：

```
void foo(int);
void foo(char*);

foo(NULL);    // 调用的是 foo(int)，因为 NULL 是 0
foo(nullptr); // 调用的是 foo(char*)，因为 nullptr 是一个指针类型
```

Q2:智能指针是如何实现自动内存管理的？它们的内部原理是什么？

A：**智能指针的核心原理**

1. **RAII（Resource Acquisition Is Initialization）**：
   智能指针将资源管理封装在其对象的生命周期内，当智能指针对象被构造时获取资源（分配内存），当智能指针对象被销毁时释放资源（释放内存）。
2. **引用计数和控制块**：
   `shared_ptr` 使用**引用计数**来跟踪所有 `shared_ptr` 实例对同一对象的引用，确保在最后一个 `shared_ptr` 离开作用域时自动释放对象。
3. **析构函数**：
   智能指针的析构函数会自动调用 `delete` 或其他自定义删除器，确保所管理对象被正确释放。

自动内存管理：

`unique_ptr` 在超出作用域时，自动调用**析构函数**来释放所管理的对象。

多个 `shared_ptr` 可以共享同一个对象的所有权，所有 `shared_ptr` 之间通过引用计数来管理该对象的生命周期。当最后一个 `shared_ptr` 离开作用域时，引用计数归零，所管理的对象被自动释放。

Q3:循环引用的问题如何避免

A:**循环引用**发生在两个或多个 `std::shared_ptr` 互相引用对方，导致**引用计数无法归零**，从而内存无法被释放。

- **避免循环引用的方法：**

使用 `std::weak_ptr`：

使用 `std::weak_ptr` 代替 `std::shared_ptr`，打破循环引用。`weak_ptr` 只持有对象的弱引用，不会增加引用计数

Q4：什么是 lambda 表达式？C++11 中如何使用它们？

A:Lambda 表达式是一种匿名函数对象，允许在代码中定义内联的、临时的函数。它们常用于需要短期使用函数的场景，如算法中的回调函数或事件处理。

- **优点：**

- 提高代码的灵活性和可读性。
- 减少不必要的函数定义。

Q5：C++11 引入了哪些智能指针？它们有什么区别？

A:C++11 引入了三种主要的智能指针：`std::unique_ptr`、`std::shared_ptr` 和 `std::weak_ptr`，它们用于自动管理动态分配的内存，防止内存泄漏和悬挂指针。

1. **`std::unique_ptr`**：
   - **特点**：独占所有权，不能被复制，只能被移动。
   - **适用场景**：适用于单一所有者的资源管理。
2. **`std::shared_ptr`**：
   - **特点**：共享所有权，多个 `shared_ptr` 可以指向同一个对象，通过引用计数管理生命周期。
   - **适用场景**：适用于多个所有者需要共享同一资源的情况。
3. **`std::weak_ptr`**：
   - **特点**：不拥有资源所有权，用于打破 `shared_ptr` 之间的循环引用。
   - **适用场景**：适用于需要观察 `shared_ptr` 管理的资源但不拥有它的场景。

- **区别总结：**

- `unique_ptr` 适用于独占资源。
- `shared_ptr` 适用于共享资源。
- `weak_ptr` 适用于观察资源，防止循环引用。

学长的考研top帮