

入门C++中的RAII

原创 Mr.pyZhang 已于 2025-02-05 16:44:27 修改 阅读量985 收藏 16 点赞数 30

分类专栏: #RAII 文章标签: c++ 面试 开发语言

RAII RAII 专栏收录该内容

0 订阅 4 篇文章 订阅专栏

RAII 概念介绍

Resource Acquisition Is Initialization

- 资源获取视为 [初始化](#) ，
- 资源释放视为销毁

为什么资源释放视为销毁

C++ 的解构函数（destructor）是显式的，与 Java，Python 等垃圾回收语言不同。

- 可以显式调用: 通过 `malloc free` ， `new delete` 和 `new[] delete[]` 手动创建释放
- 或离开作用域自动销毁

在使用上，我们希望 离开 `{ }` 作用域自动释放。有好处也有坏处，对高性能计算而言利大于弊

对C而言 - 避免内存泄漏

答：避免犯错误， 果没有解构函数自动调用：

- 每个带有返回的分支都要手动释放所有之前的资源
- 用 new/delete 或者 malloc/free 就很容易出现忘记释放内存的情况，造成内存泄露。

而例 `vector` 会在离开作用域时，自动调用解构函数，释放内存，就不必手动释放了，更安全。

```
1 | #include <vector>
2 | #include <iostream>
3 |
4 | int main() {
5 |     std::vector<int> v(4, 0);
6 |     int sum = 0;
7 |     for (size_t i = 0; i < v.size(); i++) {
8 |         sum += v[i];
9 |     }
10 |    std::cout << sum << std::endl;
11 |    // 不用手动释放
```

对java而言

使用需求

为什么很多面向对象语言，比如 Java，都没有构造函数全家桶这些概念？

因为java的业务需求大多是在和资源打交道，从而基本都是要explicit删除拷贝函数的那一类。

- 需求举例：打开数据库，增删改查学生数据，打开一个窗口，写入一个文件，正则匹配是不是电邮地址，应答 HTTP 请求等。
 - 解决这种需求，几乎总是在用 `shared_ptr<GLShader>` 的模式
- 于是 Java 和 Python 干脆简化：一切非基础类型的对象都是浅拷贝，引用计数由 [垃圾回收机制](#) 自动管理。

因此，以系统级编程、算法数据结构、高性能计算为主要业务的 C++

- 发展出了RAII思想
- 将拷贝/移动/指针/可变性/多线程等概念作为语言基本元素存在。

这些在我们的业务里面是非常重要的，不可替代。

异常安全的不同处理


异常安全（exception-safe）

C++ 标准保证当异常发生时，会触发栈解旋，因此 C++ 中没有（也不需要） `finally` 语句。

栈解旋：依次调用已创建对象的解构函数

```
1 | // C++ 标准保证当异常发生时，会调用已创建对象的解构函数。（栈解旋）
2 | // C++ 中没有（也不需要） finally 语句。
3 | try {
4 |     test();
5 | } catch (std::exception const &e) {
6 |     std::cout << "捕获异常: " << e.what() << std::endl;
7 | }
```

而对java而言，必须使用 `finally` 语句： 因为 果此处才

 Mr.pyZhang 关注

👍 30 🗨 16 💬 0 📄 分享 ... 专栏目录

但若对时序有要求就不能依靠 GC: 比 mutex 忘记 unlock 造成死锁等等..... 更不要说, 依赖GC会对性能存在影响。

```
1 | Connection c = driver.getConnection();
2 | try {
3 |     ...
4 | } catch (SQLException e) {
5 |     ...
6 | } finally {
7 |     c.close();
8 | }
```

初始化小技巧

构造函数 {}

使用 {} 和 () 调用构造函数, 有什么区别?

谷歌在其 Code Style 中也明确提出别再通过 () 调用构造函数

- 更安全: {} 是非强制转换, 即不支持强制转换
 - int(3.14f) 不会出错, 但是 int{3.14f} 会出错
 - Pig("佩奇", 3.14f) 不会出错, 但是 Pig{"佩奇", 3.14f} 会出错
- 可读性:
 - Pig(1, 2) Pig 有可能是个函数,
 - Pig{1, 2} 看起来更明确, 一定是构造函数。

需要类型转换时, 显式调用 static_cast<> 而不是 构造函数() 例 int(float f)

谷歌在其 Code Style 中也明确提出别再通过 () 调用构造函数, 需要类型转换时应该用:

- static_cast<int>(3.14f) 而不是 int(3.14f)
- reinterpret_cast<void *>(0xb8000) 而不是 (void *)0xb8000

这样可以更加明确用的哪一种类型转换 (cast) , 从而避免一些像是 static_cast<int>(ptr) 的错误。

explicit

explicit 拒绝隐式转换。

比 std::vector 的构造函数 vector(size_t n) 也是 explicit 的。

- 推荐为拷贝构造, 移动构造设置 explicit 禁止通过 = 调用拷贝构造, 移动构造
- 场景: 必须用 () 强制转换
 - 单参数
 - 拒绝 operator= 的隐式转换
 - 多个参数时
 - 禁止从一个 {} 表达式初始化。

在一个返回 Pig 的函数里用: return {"佩奇", 80};的话, 就不要加 explicit。

```
1 | class Pig {
2 |     explicit Pig(int weight)
3 |         : m_name("一只重达" + std::to_string(weight) + "公斤的猪")
4 |         , m_weight(weight){}
5 | }
6 | // 不加 explicit
7 | show(80); // 编译通过!  希望输入int, 却被隐式转换为pig
8 | Pig pig = 10; // 编译通过
9 | Pig pig(10); // 编译通过
10 |
11 | ..  ..  ..  ..
```

常引用

常引用, 值引用

常引用实际只传递了一个指针, 避免了拷贝。

以拷贝贝赋值函数而言:

- 常引用 RAII const & raii
 - RAII & operator=(RAII const & raii)
 - (推荐使用)
- 值引用 RAII & raii
 - RAII & operator=(RAII & raii)

 Mr.pyZhang

关注

 30

 16

 0

 分享

 ...

专栏目录

- （不推荐）

参数类型优化

函数参数类型优化规则：按

- 常引用、

```
1 | // 是数据容器类型 (比 vector, string) 则按常引用传递:
2 | int sumArray(std::vector<int> const &arr);
```

- 值引用、
- 值?

```
1 | // 是基础类型 (比 int, float) 则按值传递:
2 | float squareRoot(float val);
3 | // 是原始指针 (比 int *, Object *) 则按值传递:
4 | void doSomethingWith(Object *ptr);
5 | // 数据容器不大 (比 tuple<int, int>) , 则其实可以按值传递:
6 | glm::vec3 calculateGravityAt(glm::vec3 pos);
```

```
1 | // 智能指针 (比 shared_ptr) ,
2 | // - 且需要生命周期控制权, 则按值传递:
3 | void addObject(std::shared_ptr<Object> obj);
4 | // - 但不需要生命周期, 则通过 .get() 获取原始指针后, 按值传递:
5 | void modifyObject(Object *obj);
```

委托构造函数

一个构造函数委托同类型的另一个构造函数对对象进行初始化。

- 委派构造函数:
 - 不能同时使用 初始化列表
- 执行顺序
 - 将控制权交给目标构造函数
 - 在目标构造函数执行完之后, 再执行委托构造函数的主体。
 - 果委派构造函数要给变量赋初值, 初始化代码必须放在函数体中。
- 目标构造函数:
 - 被调用“基版本”构造函数就是目标构造函数。
 - 作为 “被委托函数” 可以使用初始化列表 (果本身不再作为委派构造)

注意避免构造死循环。

```
1 | class Person
2 | {
3 | public:
4 |     // 语法: 在委托构造函数的初始化列表中调用目标构造函数。
5 |     // 委派构造函数:
6 |     Person() :Person(1, 'a') {}
7 |     Person(int i) : Person(i, 'a') {}
8 |     Person(char ch) : Person(1, ch) {}
9 |
10 | private:
11 |     // 目标构造函数
```



你需要遵守的三五法则

移动/拷贝 F4

定义

```
1 | class RAII {
2 | public:
3 |     int* mIdPtr = new int(0);
4 |     std::string mName = "张三";
5 |     // 有参/无参构造
6 |     RAII() {
7 |         std::cout << "RAII() 无参构造 " << this->mName << *(this->mIdPtr) << std::endl;
8 |     }
9 |     RAII(int id, std::string name) {
10 |         *(this->mIdPtr) = id;
11 |         this->mName = name;
```



调用规则

总览

```
1 | std::cout << "-----有参构造-----\n";
~ | -----有参构造-----
```



```
6 | RAII raii_2_1 = raii_1_1;
7 | RAII raii_2_2 = RAII(raii_1_1);
8 | RAII raii_2_3(raii_1_1);
9 | RAII raii_2_4 = FuncRetObj(); // 果没有实现移动构造
10 |
11 | std::cout << "\n-----拷贝赋值-----\n";
12 | RAII raii_4_1 = RAII(4, "张三4 1");
13 | raii_4_1 = raii_3_1;
14 |
```

拷贝 还是 构造

果其中一个成员不支持 拷贝构造函数，那么 拷贝构造函数将不会被编译器自动生成。

其他函数同理。

```
1 | // 拷贝构造：直接未初始化的内存上构造1
2 | int x = 1; // 拷贝构造函数 int(int const &myint);
3 | // 拷贝赋值：先销毁现有的 1，再重新构造2
4 | x = 2; // 拷贝赋值函数 int &operator=(int const &myint)
```

拷贝

```
1 | std::cout << "\n-----拷贝构造-----\n";
2 | RAII raii_2_1 = raii_1_1;
3 | RAII raii_2_2 = RAII(raii_1_1);
4 | RAII raii_2_3(raii_1_1);
5 | RAII raii_2_4 = FuncRetObj(); // 果没有实现移动构造
6 |
7 | std::cout << "\n-----拷贝赋值-----\n";
8 | RAII raii_4_1 = RAII(4, "张三4 1");
9 | raii_4_1 = raii_3_1;
10 |
```

拷贝构造

直接在未初始化的内存上构造

参数

- 参数必须是引用——RAII(RAI raii_) 是错误的
- 常引用RAII(RAI const & raii_) 优于 值引用RAII(RAI & raii_)。
- 常引用 和 值引用可以同时存在。重载的强大之处~

何时触发

- 显示调用，类型作为参数
- 函数返回值。没有移动构造时，指向拷贝构造。

形参	对应函数	评价
RAII obj_new = obj_old;	RAII(RAI const & raii)	直接在未初始化的内存上构造
RAII obj_new = RAII(obj_old);	同上	同上
RAII obj_new(obj_old);	同上	同上
RAII obj_new = funcRet();	未定义移动构造函数	低效

拷贝赋值

先销毁现有的 再重新构造

- 值引用 不推荐
- 常引用RAII & operator=(RAI const & raii)

形参	对应函数	评价
obj_exists = RAII(1);	RAII(int) ~RAII() 以及RAII & operator=(RAI const & raii)	低效 创建临时对象又马上被销毁
obj_exists = obj_old;	RAII & operator=(RAI const & raii)	正确做法

移动

正确实现移动语义需要你的对象容纳一个“空状态”，移动时需要将源对象置空，析构时也需要判空。

```
1 | std::cout << "\n-----移动构造-----\n";
2 | RAII raii_3_1 = std::move(RAI(3, "张三3 1")); // 低效 创建临时对象又马上被销毁
3 | RAII raii_3_2(std::move(raii_2_1));
4 | RAII raii_3_3 = RAII(std::move(raii_1_1));
5 | RAII raii_3_4 = FuncRetObj(); // 函数返回值
6 |
7 | std::cout << "\n-----移动赋值-----\n";
8 | RAII raii_5_1 = RAII(5, "张三5 1");
9 | RAII raii_5_2 = RAII(5, "张三5 2");
10 | raii_5_1 = std::move(raii_2_2);
11 | raii_5_2 = RAII(42, "张三5 tmp"); // 低效 创建了临时对象
```


移动构造

目标：移动构造RAII(RAI && raii)

何时触发

- 显示调用，类型作为参数
- 函数返回值。没有移动构造时，指向拷贝构造。

形参

 Mr.pyZhang [关注](#)

👍 30

💬

🌟 16

💬 0

🔗 分享

⋮

[专栏目录](#)

形参	对应函数	评价
<code>RAII obj_new(std::move(obj_old));</code>	<code>RAII(RAII && raii)</code>	正确做法
<code>RAII obj_new = RAII(std::move(obj_old));</code>	同上	同上
<code>RAII obj_new = funcRet();</code>	同上	同上
<code>RAII obj_new = std::move(RAII(1));</code>	<code>RAII(int)</code> <code>~RAII()</code> 以及 <code>RAII(RAII && raii)</code>	低效 创建临时对象又马上被销毁

移动赋值

移动赋值 `RAII & operator=(RAII && raii)`

形参	对应函数	评价
<code>obj_exists = std::move(RAII(1));</code>	<code>RAII(int)</code> <code>~RAII()</code> 以及 <code>RAII & operator=(RAII && raii)</code>	低效 创建临时对象又马上被销毁
<code>obj_exists = std::move(obj_old);</code>	<code>RAII & operator=(RAII const & raii)</code>	正确做法

缺省实现

移动语义

果不定义移动构造和移动赋值，编译器为保证不出错，会自动实现默认的缺省实现：

虽然低效，但至少可以保证不出错。

- 缺省 移动构造
 - `≈拷贝构造+他解构+他默认构造`
- 缺省 移动赋值
 - 未自定义 移动构造
 - `≈拷贝赋值+他解构+他默认构造`
 - 自定义 移动构造
 - `≈解构+移动构造`

拷贝

拷贝赋值

三五法则

概念

修改任意一个，就需要改3个：

- 析构函数，拷贝构造，拷贝赋值。

自定义了 **析构函数** ，那就

- 把移动构造函数和拷贝构造函数全部delete掉！
- 果确实需要移动
 - 自己定义或default掉移动构造函数。（不建议尝试）
 - 使用unique_ptr。

果对提高性不能感兴趣，可以忽略

- 移动构造
- 移动赋值

要实现移动语义，需要实现5个：

- 析构函数，拷贝构造，拷贝赋值，移动构造，移动赋值。
- 正确实现移动语义需要你的对象容纳一个“空状态”，
移动时需要将源对象置空，析构时也需要判空。

不能做的事情：

如果类定义了		必须同时	函数	错误原因	解决原理
解构函数	<code>~RAII()</code>	定义	拷贝构造函数 和 拷贝赋值函数	避免浅拷贝指针导致多次释放同一内存。	“封装：不变性”服务。即：保证任何单个操作前后，对象都是处于正确的状态，从而避免程序读到错误数据（空悬指针）的情况。
		或 删除	-	-	我们压根就不允许这么用，在编译期就发现错误。
移动构造函数	<code>RAII(RAII &)</code>	定义 或 删除(删除仍然低效)	移动赋值函数		
拷贝构造函数	<code>RAII(RAII const &)</code>	定义 或 删除(仍然低效)	拷贝赋值函数		内存的销毁重新分配可以通过realloc，从而就地利用当前现有的m_data，避免重新分配。
		定义	移动构造函数 (否则低效)		
拷贝赋值函数		定义	移动赋值函数 (否则低效)		

判断安全

安全

 **Mr.pyZhang** [关注](#)

 30



 16

 0

 分享

 ...

[专栏目录](#)

一般来说，可以认为符合三五法则的类型是安全的。

判断方式：

- 果不需要自定义的解构函数，那么这个类就不需要担心。
- 否则，往往意味着类成员中，包含有不安全的类型。

果类所有成员都是安全的类型，类自动就是安全的。

则五大函数都

- 无需声明
- 或声明为 = default

不安全

不安全：一般无外乎两种情况：

- 类管理着资源。
 - 这个类管理着某种资源，资源往往不能被“复制”。
 - 删除拷贝函数，统一用智能指针管理
 - 避免每个资源类实现一遍原子引用计数器（不推荐）
- 类是数据结构：你精心设计
 - 考虑定义拷贝和移动
 - 数据结构是否支持拷贝（比 Vector 就可以），
 - 支持：自定义。
 - 不支持：删除（= delete）。

例子

以下类型是安全的：

```
1 | // 基础类型
2 | int id;
3 | // STL 容器
4 | std::vector<int> arr;
5 | // 智能指针
6 | std::shared_ptr<Object> child;
7 | // 原始指针， 果是从智能指针里 .get() 出来的
8 | Object *parent;
```

以下对象是不安全的：

```
1 | // 原始指针， 果是通过 malloc/free 或 new/delete 分配的
2 | char *ptr;
3 | // 是基础类型 int，但是对应着某种资源
4 | GLint tex;
5 | // STL 容器，但存了不安全的对象
6 | std::vector<Object *> objs;
```

默认生成规则

f4的打包删除

f4 删除一个而其它的没有显式定义，则编译器自动删除其它三个。

```
1 | class Resource {
2 |     Resource();
3 |     Resource(Resource &&) = delete; // 其他三个也会被删除
4 | };
```

何时生成默认拷贝构造

何时编译器生成 默认拷贝构造：编译器觉得 果没有的话你会出错，所以给你整了一个

果不提供默认拷贝构造函数，编译器会按照位拷贝进行拷贝（位拷贝指的是按字节进行拷贝，有些时候位拷贝出现的不是我们预期的行为，会取消一些特性）

以下是编译器需要强制提供默认拷贝构造函数的必要条件：来自知乎

- 类成员
 1. 存在类成员，是一个有拷贝构造函数的类。
 - 为了让成员类的拷贝构造函数能够被调用到，不得不为类生成默认拷贝构造函数。
 2. 有类成员，包含一个或多个 **虚函数** 。
 - 其类成员的虚函数表指针，需要调用其拷贝构造函数才不会丢失。
 - 需要为类生成默认拷贝构造函数，完成类成员拷贝构造函数的调用 & 类成员虚函数表指针的拷贝，从而完成虚函数表指针的拷贝。
- 基类
 3. 基类，是一个有拷贝构造函数的类。
 - 子类执行拷贝构造函数时，先调用父类的拷贝构造函数，
 - 为了能够调用到父类的拷贝构造，不得不生成默认的拷贝构造函数。
 4. 基类，有一个或多个虚函数。

 **Mr.pyZhang** [关注](#)

 30

 16

 0

 分享

 ...

[专栏目录](#)

- 果不提供默认拷贝构造函数，
 - 会进行位拷贝。类成员的拷贝构造函数不被调用
 - 从而基类的虚函数表指针（可能）会丢失
 - 需要为类生成默认拷贝构造函数，调用基类的拷贝构造函数
- 完成基类拷贝构造函数的调用，从而完成虚函数表指针的拷贝。

我认为以上解释存在存在虚函数的成员时，“为了避免浅拷贝”这个理由是错的。

- 浅拷贝 相当于多个对象 共用一个指针，由于没有人能确保所有权，其指向可能被释放
- 但是，虽然 虚表指针是一个指针 可对于同一个类，
 - 其指向是固定的 其虚函数表在rodata区。
 - 任何一个对象释放都不会去释放这个属于类的虚函数表。

那么为什么，有虚函数表的类 编译器会帮我们 默认拷贝构造函数呢？

我认为使用 其同样拥有虚函数表的 子类 进行拷贝时。确保虚表指针的正确(存疑)，以及正确拷贝内存中的内容 不要包含子类部分。

C++ Qt零基础入门进阶与企业级项目实战教程与学习方法分享

瑶娜猫地 4113

介绍Qt环境搭建、QtCreator / VS2019的基本使用方法、Qt整体架构、Qt信号机制、Qt内存管理等知识。

C++ 资源管理 (RAII)

weixin_39640298的博客 857

1、概述所谓资源，就是程序员从操作系统中获取的硬件资源，我们申请了资源，使用完毕之后还要还给系统。C++最常使用的是内存资源，其它还包括文件描述符、互斥锁、图形界面中的字体和笔刷、数据库连接、以及网络socket等。尝试在...

C++中RAII详解_c++ raii

2-18

为了减少这些问题,C++提供了一种强大的设计模式:RAII(Resource Acquisition Is Initialization)即资源获取即初始化。RAII为C++的内存管理提供了一个结构化、可靠的解决方案,使程序员能够高效、安全地管理资源。本文将深入介绍RAII...

全面理解C++中的RAII机制

2-18

在C++中,RAII(Resource Acquisition Is Initialization 资源获取即初始化)是一种核心编程范式,通过对象的生命周期管理资源(内存、文件句柄、网络连接等),确保资源的自动获取和安全释放。以下是RAII的详细解析: 一、RAII的核心思想 资源...

RAII

weixin_34354173的博客 213

RAII是Resource Acquisition Is Initialization的简称,是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象,在其构造时获取对应的资源,在对象生命周期内控制对资源的...

《面试必问》C++ RAII 详解 最新发布

码事漫谈 813

RAII是C++中一种强大的资源管理技术,通过将资源的生命周期与对象的生命周期绑定,确保资源在对象销毁时自动释放。RAII不仅简化了资源管理,还提高了代码的异常安全性和可维护性。在实际开发中,RAII广泛应用于智能指针、文...

C++: RAII是什么——使用对象来管理资源

2-22

在本文中,我们介绍了C++中的RAII技术,它是一种管理资源的方法,可以帮助我们避免内存泄漏和资源泄漏等问题。RAII技术的核心思想是将资源的获取和释放绑定在对象的生命周期中,这样可以确保资源在不再需要时被正确释放。我们还介绍...

C++ 编程基础(10)RAII(Resource Acquisition Is Initialization)机制...

2-17

RAII是C++编程中的一种重要惯用法,它通过将资源的分配与对象的生命周期绑定在一起,简化了资源管理的逻辑,提高了程序的可靠性和可维护性。在实际编程中,应该充分利用RAII机制来管理各种资源,以确保程序的正确性和稳定性。

C++之RAII (Resource Acquisition Is Initialization)

ly13438189519的博客 522

转载: https://www.jianshu.com/p/b77fe79498be 什么是RAII? RAII是Resource Acquisition Is Initialization (wiki上面翻译成“资源获取就是初始化”)的简称,是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会...

C++RAII

CSDN-ZCL 527

RAII 资源获取即初始化 (Resource Acquisition Is Initialization)

C++中什么是RAII?_c++ raii

2-19

RAII可以应用于多种资源管理场景,包括但不限于:内存管理: std::unique_ptr和std::shared_ptr在C++11中就是RAII的典型实现,自动管理动态分配的内存。 文件管理: std::fstream在打开文件时自动获取文件句柄,在析构时关闭文件。 锁管理...

C++11新增特性:智能指针(RAII)_c++ raii

2-19

一、RAII 二、智能指针原理 1、auto_ptr 2)、unique_ptr 3)、shared_ptr 增加定制删除器 4)、weak_ptr 1 (weak_ptr由来和用途 2 (循环引用问题 3 (weak_ptr 三、C++11和boost中智能指针的关系 一、RAII RAII(Resource Acqui...

C++ RAII

马德里小铁匠的铁匠铺 396

RAII的核心思想是将资源或者状态与对象的生命周期绑定,通过C++的语言机制,实现资源和状态的安全管理。理解和使用RAII能使软件设计更清晰,代码更健壮。资源管理RAII是C++的发明者Bjarne Stroustrup提出的概念,RAII全称是“Res...

C/C++ 学习入门代码案例 - RAII样例代码

06-05

掌握 auto_ptr (C++17 已经被正式从C++标准里删除了)、unique_ptr、shared_ptr、weak_ptr 原理及用法; 样例代码中用法都有介绍。 重点掌握 unique_ptr 原理, shared_ptr 原理 (引用计数管理方法)、weak_ptr...

【C++】RAII--C++中最厉害的编程范式(小白一看就懂!!!)_c++ raii-CSDN...

2-14

RAII(Resource Acquisition Is Initialization 资源获取即初始化)是一种在C++中常见的编程范式,主要用于管理资源(动态内存、文件句柄、网络连接等)。其核心思想是将资源的生命周期绑定到对象的生命周期,通过对象的构造函数来获取资源...

【C++】智能指针(RAII思想)_c++ raii

2-13

RAII(Resource Acquisition Is Initialization 资源获取即初始化)是一种资源管理类的设计思想,广泛应用于C++等支持对象导向编程的语言中。它的核心思想是将资源的管理与对象的生命周期紧密绑定,通过在对象的构造函数中获取资源,并在析...

Cplusplus基础与提高.zip_C++ 基础与提高_c++入门和提高

09-15

3. **RAII (Resource Acquisition Is Initialization) **：一种编程原则，确保资源在对象生命周期内正确管理。 4. **Lambda表达式**：C++11引入的新特性，用于创建匿名函数，简化代码并提高可读性。 5. **多线程...

C++ RAII资源管理方法与智能指针使用案例解析

资源摘要信息“本资源摘要旨在详细解读《C/C++ 学习入门代码案例 - RAII样例代码》文件的内容，针对标题、描述、标签及文件列表提供深入的知识点介绍。以下内容将围绕RAII的概念、智能指针的原理及用法以及资源管理...

C++:什么是RAII?智能指针

2-15

一、什么是RAII? RAII(Resource Acquisition Is Initialization)是由c++之父Bjarne Stroustrup提出的,中文翻译为资源获取即初始化,使用局部对象来管理资源的技术称为资源获取即初始化;这里的资源主要是指操作系统中有限的东西 内存(heap)...

游戏编程入门_directX C++ 课件_代码_第二5课

01-03

本课件集合了DirectX与C++的结合,旨在帮助初学者入门游戏编程。在游戏编程中,DirectX提供了音频、视频、图形和输入等关键功能的API,让开发者能够直接与硬件交互,实现高性能的游戏效果。Direct3D是DirectX的一...

C++中的RAII 热门推荐

一簪烟雨任平生_也无风雨也无晴 5万+

有很多东西我们一直在用,但是不知道他的名字。什么是RAII? RAII是Resource Acquisition Is Initialization的缩写,用普通话讲就是“资源获取即初始化”为什么需要RAII? 看一段代码: RawResourceHandle* handle=createNewResource(); ha...

【C++设计技巧】C++中的RAII机制

221

作者: gnuhpc 出处: http://www.cnblogs.com/gnuhpc/ 1.概念 Resource Acquisition Is Initialization 机制是Bjarne Stroustrup首先提出的。要解决的是这样一个问题:在C++中,如果在这个程序段结束时需要完成一些资源释放工作,那么正常...

C++ RAII机制

Arcohaleno 300

最近一直在碰到这个RAII机制,但是似乎没搞清楚啥意思。现在大概明白了,C++的RAII机制就是类似于C#或者Java的GC机制。垃圾回收。合理的回收系统资源,避免程序员大量的写重复的delete代码来手动回收。转载于: https://blog.cs...

C++中的RAII机制

Lce的专栏 615

什么是RAII? RAII是Resource Acquisition Is Initialization的简称,是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象,在其构造时获取对应的资源,在对象生命期...

微软Visual Studio中文版C++教程: RAII与异常处理

4. **C++入门与学习路径** - 提供了针对C++初学者的推荐教程,包括从基础开始,在Visual Studio中安装C/C++支持和创建HelloWorld程序。 5. **Visual Studio集成开发环境** - 教授何在Visual Studio中进行C++...

C++中文离线手册.rar

它涵盖了C++的基础到高级概念,旨在帮助初学者快速入门C++。

Mr.pyZhang 关注

30 16 0 分享 ... 专栏目录

12-17

C++中RAII机制

RAII是“资源获取就是初始化”的简称，是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命周期内控制对资源的访问，使之始...

关于我们 招贤纳士 商务合作 寻求报道 400-660-0108 kefu@csdn.net 在线客服 工作时间 8:30-22:00

公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心 家长监护 网络110报警服务 中国互联网举报中心 Chrome商店下载 账号管理规范 版权与免费声明 版权申诉 出版物许可证 营业执照

©1999-2025北京创新乐知网络技术有限公司

Mr.pyZhang

VP

码龄7年 暂无认证

36

1万+

2万+

3万+

原创

周排名

总排名

访问

等级

1141

437

812

0

590

积分

粉丝

获赞

评论

收藏

私信

关注

2024博客之星
年度评选已开启

MAC mini、大疆无人机、华为手机等精美奖品等你来拿！

立即去查看

搜博文

热门文章

CMake快速上手 · 2087

一文速通C++中宏的现代用法 · 1618

设计模式杂谈 · 1504

RAII - 安卓中的智能指针 · 1483

编译链接的基础概念 · 1451

分类专栏

Linux3篇

C++

内存那些事4篇

编译期那些事6篇

多线程安全7篇

语法糖1篇

大家在看

DeepSeek-V3 技术报告 · 1040

Spring Boot 项目整合 XXL - Job 使用流程详解 · 796

毕业设计：基于springboot+vue开发的酒店住宿管理系统【源码+sql+可运行】_【50227】 · 191

莆田鞋纯原鞋与正品鞋之间的十大区别,看完你也变莆田鞋专家

动手学深度学习：线性回归神经网络

最新文章

Linux/POSIX 多路IO复用

C++17并行化加速STL算法——std::execution

静态多态——CRTP奇异性模板递归

2025年 17篇

2024年 19篇

目录

RAII 概念介绍

为什么资源释放视为销毁

对C而言 - 避免内存泄漏

对java而言

初始化小技巧

构造函数0

explicit

常引用

委托构造函数

你需要遵守的三五法则

移动/拷贝 F4

Mr.pyZhang

关注

30

16

0

分享

专栏目录

第8页 共9页

2025/2/23 18:58

[三五法则](#)

[默认生成规则](#)



Mr.pyZhang

[关注](#)

30



16

0

[分享](#)



[专栏目录](#)