RAII resource Acquisition is Initialization 资源获取即初始化

在构造函数中申请分配资源，在析构函数中释放资源。因为C++语言机制保证了当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。

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++语言机制，实现资源和状态的安全管理，智能指针是RAII的最好例子

# 内存结构

内核态1G，用户态3G

用户态： 代码段（机器码和字符串常量），数据段，ssd段，堆，映射区（动态链接文件的映射），栈

# 结构体对齐

假设第一个元素起始地址是0，那么其他元素起始地址应为自身对齐大小的整数倍；

结构体整体的大小应该为最大元素大小的整数倍

作用：便于存取，提高存储效率。

一方面是和cpu的架构有关，一个地址（1字节），有的cpu架构从偶数位开始读，如果int变量从奇地址开始，那么就需要2个读周期

# 源代码到可执行文件

预处理：g++ -E mian.cpp -o mian.ii -E代表预编译。

编译：g++ -S main.ii -o main.s -S表示只编译。编译是在预处理文件基础上经过一系列词法分析、语法分析及优化后（涉及到指令重排）生成汇编代码

汇编：g++ -c main.s -o main.o 将会汇编代码转换成机器可执行指令(也称目标文件)

链接 g++ main.o mani.out 解决符号之间重定位问题（将多段机器码组合成可执行）。由于cpp中会用到其他模块中的函数或全局变量，在编译单个cpp的时候是没法之傲他们准确地址。连接器将各种没有准确地址的符号（函数、变量等）设置位正确的值，这样组装在一起就可以形成一个完整的可执行程序。

静态链接：.a。window中是lib静态链接即使用到的地方都拷贝一份。

动态链接：.so。window中是dll用到的时候再从磁盘加载到内存。运行时绑定地址。通过页请求调入。

装载 将可执行程序加载到内存

.ko(kernal object)内核的目标文件，即内核模块

# 大端模式和小端模式

大端就是数据的高低位和存储的高低地址一致

表格

描述已自动生成

# New和malloc的区别

void \*malloc(size\_t size)

new xxclass();

malloc需要指定大小，返回的是void指针；new会调用自定义类的构造函数返回该对象的指针。

Malloc是C的库函数依赖于头文件，new是关键字需要编译器编译

Malloc用free释放，new用delete

New做的事情比malloc多：分配内存和调用类的构造函数；delete调用类的解析函数和释放内存。

# malloc底层原理

malloc底层是个内存池，双向链表管理空闲内存；找到第一块大的一份为二；空闲出来后就紧缩，将小块的连续的合并。Brk

mmap是在映射区做映射

2、进程调用A=malloc(30K)以后  
malloc函数会调用brk系统调用，将\_edata指针往高地址推30K，就完成虚拟内存分配。  
你可能会问：只要把\_edata+30K就完成内存分配了？

事实是这样的，\_edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生缺页中断，这个时候，内核才分配A这块内存对应的物理页。也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。图示

描述已自动生成

**情况二、malloc大于128k的内存**

malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)，如下图：

图示

描述已自动生成

## 有了 malloc/free 为什么还要 new/delete ？

malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 malloc/free 无法满足动态对象的要求。

对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。 因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。

# Heap和stack区别

自由存储区是C++中通过new和delete动态分配是释放对象的抽象概念。而堆是操作系统的术语。

堆是动态存储区，new

栈存放临时变量

栈是从高地址向低地址生长

堆是低地址向高地址生长

# C++是不是类型安全的

类型安全很大程度上等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。而C++两个不同类型的指针之间可以强制转换（reinterpret cast）.

重载和重写的区别

重载：允许多个同名函数，而这些函数的参数表不同；

重写：子类重新定义子类重新定义父类的方法。动态调用。运行时一个父类指针指向子类，

# 多态是如何实现的？运行时覆盖

一个类一个虚表，同一个类的实例共用一个虚表。

即使子类中没有虚函数，也会有一个虚表，里面放着父类的虚函数地址。

由于pb实际指向的对象类型是Derive，因此vptr指向的Derive类的vtable，当调用pb->func()时，根据虚表中的函数地址找到的就是Derive类的func()函数。在定义一个类的时候，就会生成相应的虚表（虚函数地址）；

定义一个子类的时候，不会增加虚表

在运行的时候，实例化了一个子类，构造过程如下：

1. 基类base1构造：虚表赋值，成员数据
2. 基类base2构造，虚表赋值，成员数据
3. 派生类AB构造：虚表覆盖，成员数据

每个对象都有一个虚表指针指向父类虚表，在所有父类中国覆盖掉同名函数指针，子类独有的虚函数添加到第一个父类虚表末尾。

图示

描述已自动生成 表格

描述已自动生成

**情况一：多继承**

1. A中有虚函数fun
2. B中有虚函数fun
3. Class C:public A,public B 也有个fun虚函数

A:Vtable（\*fun）

B::Vtable(\*fun)

C把A,B虚表中的fun覆盖成自己fun所在地址

**情况二：多继承+独有虚函数**

情况一中除了fun还有其他虚函数func

Func添加到ATable末尾

情况三：父类A被B，C继承

B会和情况2一样

C会拷贝一张父类表，把自己独有虚函数加载末尾

Class B: public A

Class C: public A

B b； C c；

编译时一个虚表？运行时会产生两个虚表？都是A:vtable?

Class B: publicA

Class C :public B

编译时一个虚表AVtable,

C c；//在虚表中加入B和C独有的函数

如果时

C c; B b; A a; 运行时有几个虚表

**情况四：虚继承Class B :virtual public A**

成员数据在内存中存储的结构：

假设没有virtual虚继承关键字，the对象在内存中的结构如下:

A::m\_nDataA

B::m\_nDataB

Class B :virtual public A

现在我们的源码中有virtual继承关键字，那么内存结构必然会有区别,那么内存结构是怎么样的呢？如下：

B::base Offset m\_nDataA     // A::m\_nDataA数据的偏移，只保留一份基类数据，数据偏移和虚表偏移不同

B::m\_nDataB

A::m\_nDataA

编译器为什么要这么做呢？这个偏移值是什么？这么做的意义又何在？

首先，这么做是为了只存在于一份虚基类数据。B::base Offset 偏移的值，一般为全局数据区中。编译器为了和虚表区别。这个指针指向的地址的值一般为：0x00000000, 或者某些特殊值

由于是虚继承,所以虚基类只会产生一份拷贝.内存结构必然如下：

# 面向对象的三个特征

封装继承多态

C++中四种类型转换方式

Dynamic\_cast

运行时强制转化。子类转父类指针时安全的；父类转成子类指针时不安全的会返回nullptr

向上转型：子类对象指针->父类指针/引用（不需要转换）  
向下转型：父类对象指针->子类指针/引用（用dynamic\_cast转型是安全的）

Static\_cast

隐式转换都是statc\_cast实现的。进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性需要开发者来维护。

Const\_cast

1、常量指针被转化成非常量的指针，并且仍然指向原来的对象；   
2、常量引用被转换成非常量的引用，并且仍然指向原来的对象；

文本

描述已自动生成

Reinterpret\_cast 不相关类型强制转换

# 深拷贝/浅拷贝/临时对象

深拷贝拷贝了资源，与源对象独立；浅拷贝只是拷贝了指针，指向的时同一个资源；

浅拷贝在析构的时候会析构两次，造成程序奔溃。临时对象的开销比局部对象小些。

临时对象就是右值

移动构造函数与拷贝构造函数的区别是，拷贝构造的参数是const MyString& str，是*常量左值引用*，而移动构造的参数是MyString&& str，是*右值引用*，而MyString("hello")是个临时对象，是个右值，优先进入**移动构造函数**而不是拷贝构造函数。而移动构造函数与拷贝构造不同，它并不是重新分配一块新的空间，将要拷贝的对象复制过来，而是"偷"了过来，将自己的指针指向别人的资源，然后将别人的指针修改为nullptr，这一步很重要，如果不将别人的指针修改为空，那么临时对象析构的时候就会释放掉这个资源，"偷"也白偷了。下面这张图可以解释copy和move的区别。手表的钟表

中度可信度描述已自动生成

MyString str1("hello"); //调用构造函数

MyString str2("world"); //调用构造函数

MyString str3(str1); //调用拷贝构造函数

MyString str4(std::move(str1)); // 调用移动构造函数、

// cout << str1.get\_c\_str() << endl; // 此时str1的内部指针已经失效了！不要使用

//注意：虽然str1中的m\_dat已经称为了空，但是str1这个对象还活着，知道出了它的作用域才会析构！而不是move完了立刻析构

MyString str5;

str5 = str2; //调用拷贝赋值函数

MyString str6;

str6 = std::move(str2); // str2的内容也失效了，不要再使用

str6 = std::move(str2)，虽然将str2的资源给了str6，但是str2并没有立刻析构，只有在str2离开了自己的作用域的时候才会析构，所以，如果继续使用str2的m\_data变量，可能会发生意想不到的错误。

# 为什么提出右值

第一个问题就是临时对象非必要的昂贵的拷贝操作，第二个问题是在模板函数中如何按照参数的实际类型进行转发。

C++11中所有的值必属于左值、将亡值、纯右值三者之一。

int i = getVar();

上面的这行代码很简单，从getVar()函数获取一个整形值，然而，这行代码会产生几种类型的值呢？答案是会产生两种类型的值，一种是左值i

T&& k = getVar();

getVar()产生的临时值不会像第一行代码那样，在表达式结束之后就销毁了，而是会被“续命”，他的生命周期将会通过右值引用得以延续，和变量k的声明周期一样长。

Inline含义时什么？优缺点？

inline与函数的定义体放在一起，使该函数称为内联。inline是一种用于实现的关键字，而不是用于声明的关键字。

内联函数的特点：使用内联函数的目的是为了提高函数的运行效率。内联函数体的代码不能过长，因为内联函数省去调用函数的时间是以代码膨胀为代价的。内联函数不能包含循环语句，因为执行循环语句要比调用函数的开销大。

### 多态类中的虚函数表是Compile-Time，还是Run-Time时建立的?

虚拟函数表是在编译期就建立了,各个虚拟函数这时被组织成了一个虚拟函数的入口地址的数组.而对象的隐藏成员--虚拟函数表指针是在运行期--也就是构造函数被调用时进行初始化的,这是实现多态的关键。

# Static用法

**修饰全局变量**：本文件范围作用域。

**修饰局部变量**：程序生命周期内分配只分配一次。函数结束，内存仍在，在此进入，操作的是同一块内存。

**修饰类成员变量**：对象共享。所有对象共享一个变量

**修饰类成员函数**：对象共享，使用类名定义范围解析运算符来调用。只能访问静态成员，不能访问非静态成员。

**修饰类对象：**和修饰变量一样，作用域本文件。整个生命周期。

# Const关键字有哪些作用

只读不能改变，在常量区。

**修饰变量**：变量内容不可变

**修饰指针**，指针指向不能变

**修饰形参**，在函数内形参不能被改变

**修饰成员函数**：不能改变类成员变量。函数名（ 形参表 ）const {函数体}

**修饰对象**：只能访问const常函数

**修饰返回值**：确保返回的不是左值。

修饰成员变量：只能在初始化列表中

Const不能和static同时修饰类成员函数，静态成员函数是类调用const必须要有实例

### const与#define 的比较，const有什么优点?

（1） const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应）。

（2）有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

# 智能指针

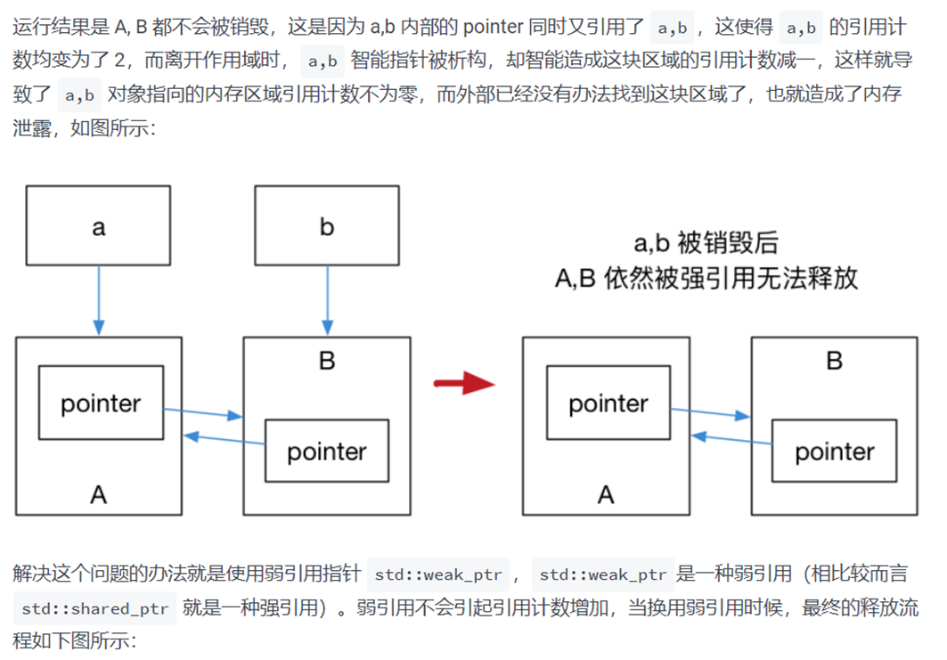
**Unique\_ptr**:独占智能指针，禁止其他智能指针共享一个的的对象，保证了代码安全性。不可复制，单可以使用std::move将其转移给其他unique\_ptr

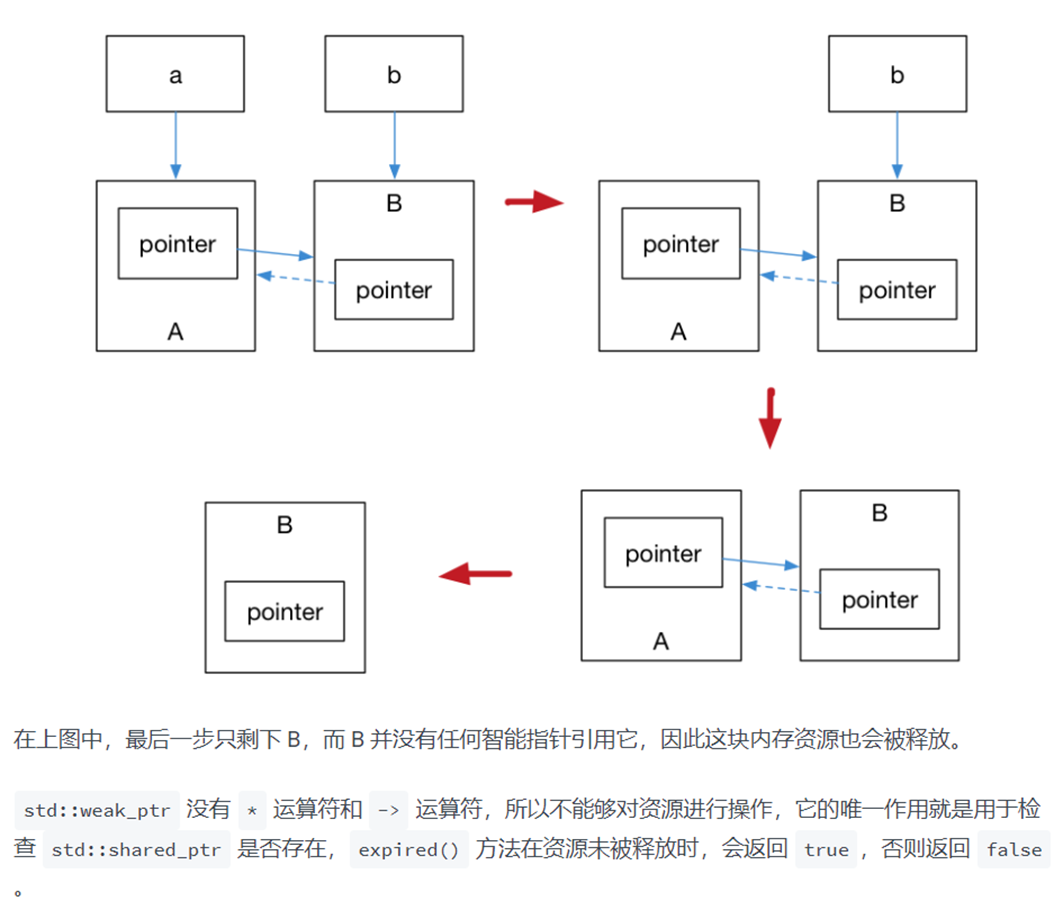
**Shared\_ptr**: shared\_ptr 共同指向一个对象，从而消除显示的调用 delete，当引用计数变为零的时候就会将对象自动删除。std::make\_shared 就能够用来消除显示的使用 new，所以std::make\_shared 会分配创建传入参数中的对象，并返回这个对象类型的std::shared\_ptr指针。

std::shared\_ptr 可以通过 get() 方法来获取原始指针，通过 reset() 来减少一个引用计数，并通过get\_count()来查看一个对象的引用计数。

weak\_ptr 弱引用

它的构造和析构不会引起引用记数的增加或减少. weak\_ptr 在功能上类似于普通指针, 然而一个比较大的区别是, 弱引用能检测到所管理的对象是否已经被释放, 从而避免访问非法内存。





## 解决hash冲突的方法

<https://www.jianshu.com/p/4d3cb99d7580>

解决哈希冲突的方法一般有：开放定址法、链地址法（拉链法）、再哈希法、建立公共溢出区等方法。

开放定址法：线行探查法、平方探查法、伪随机探测法。

# 引用和指针

都是指向一个对象内存，对对象进行操作。不涉及到拷贝对象。

引用的对象必须被初始化

引用初始化后不能被改变，指针可以

不存在指向空值的引用，但是存在指向空值的指针

## 内存碎片？

内部碎片就是已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间；

外部碎片指的是还没有被分配出去（不属于任何进程），但由于太小了无法分配给申请内存空间的新进程的内存空闲区域。

分页就是为了减小内部碎片的大小。分页不存在外部碎片。都是以页位单位非连续分配

## STL容器是线程安全的吗？

多个读线程没有写则是安全的。如果有写线程，在对同一个容器进行多线程的读写、写操作；在每次调用容器的成员函数期间都要锁定该容器；在每个在容器上调用的算法执行期间锁定该容器。

在工厂中多线程STL的场景应该比较常见，一个典型例子就是用其来左生产者-消费者模型的队列或者其他共享队列，这样为了应对线程安全问题我们必须自己对容器进行封装。

## STL迭代器的作用

通过迭代器可以在不里哦阿姐内部原理的情况下遍历 容器。

是容器和算法的粘接剂，通过提供统一的迭代器接口，同一套算啊代码可以应用在不同容器中。

## STL迭代器？列举几个迭代器失效的场景

文本

中度可信度描述已自动生成

# 程序加载过程

https://blog.csdn.net/wzq2009/article/details/103147255

在内核中每个进程都有个task\_struct,（就是PCB）里面有个mm\_struct是虚拟内存地址相关数据。mm\_struct中有pgd页表指针和mmap（指向vm\_area\_struct链表是是虚拟内存空间的部分地址）

图示

描述已自动生成

代码中用的是虚拟内存地址

实际运行的时候，cpu读取逻辑地址数据。

受限将虚拟地址发送给MMU单元返回物理地址。

那么MMU是如何实现虚拟内存和物理内存的映射呢？

数据和代码一开始在磁盘地址中，mmap将磁盘文件映射到虚拟地址（初始化vm\_area\_struct）到此完成了内存映射

当读到映射区的虚拟地址时候，MMU查询高速缓存页如果没有对应的物理地址；就查询页表项，然后找到页表，如果没有加载到物理内存；就发生页中断，到硬盘的swap区查找；如果还是找不到就根据mmap映射关系将磁盘文件加载到内存；然后更新页表和高速缓存页。

图示

描述已自动生成

图示

描述已自动生成

图示

描述已自动生成

PTEA：page table entry table

PA: physical address

## C++共享内存实现

https://blog.csdn.net/qq\_33611327/article/details/81738195

图形用户界面, 文本

描述已自动生成

# Mmap原理

内存copy次数少了，本来要先从磁盘读入内核的页缓存，再从页缓存读入用户内存；

Mmap是建立文件和虚拟内存的映射。使得IO读写加快。读写次数从2次降到了1次。

图表, 箱线图

描述已自动生成

一种是open一个文件，然后使用read系统调用读取文件的一部分或全部。这个read过程是这样的：内核将文件中的数据从磁盘区域读取到内核页高速缓冲区，再从内核的高速缓冲区读取到用户进程的地址空间。这里就涉及到了数据的两次拷贝：磁盘->内核，内核->用户态。而且当存在多个进程同时读取同一个文件时，每一个进程中的地址空间都会保存一份副本，这样肯定不是最优方式的，造成了物理内存的浪费

图示

描述已自动生成

当多个进程需要同时访问同一个文件时，每个进程都将文件所存储的内核高速缓冲映射到自己的进程地址空间。当第一个进程访问内核中的缓冲区时候，前面讲过并没有实际拷贝数据，这时MMU在地址映射表中是无法找到与ptr相对应的物理地址的，也就是MMU失败，就会触发缺页中断。内核将文件的这一页数据读入到内核高速缓冲区中，并更新进程的页表，使页表指向内核缓冲中的这一页。之后有其他的进程再次访问这一页的时候，该页已经在内存中了，内核只需要将进程的页表登记并且指向内核的页高速缓冲区即可。

# Struct和class区别

默认继承权限：struct是public，class是private

默认的成员访问权限：struct是public,class是private

# C++和C的区别

C是面向过程，C++是面向对象的

C适合代码小，效率高的场合，如嵌入式；C++适合更上层的，复杂的

# Const符号常量

看左边

# 在类中如何使用const的

有时我们希望某些常量只在类中有效。由于#define 定义的宏常量是全局的，不能达到目的，于是想当然地觉得应该用 const 修饰数据成员来实现。const 数据成员的确是存在的，但其含义却不是我们所期望的。const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的，因为类可以创建多个对象，不同的对象其 const 数据成员的值可以不同。 不能在类声明中初始化 const 数据成员。

const 数据成员的初始化只能在类构造函数的初始化表中进行。

### 以下三条输出语句分别输出什么？

char str1[] = "abc";

char str2[] = "abc";

const char str3[] = "abc";

const char str4[] = "abc";

const char\* str5 = "abc";

const char\* str6 = "abc";

cout << boolalpha << ( str1==str2 ) << endl; // 输出什么？

cout << boolalpha << ( str3==str4 ) << endl; // 输出什么？

cout << boolalpha << ( str5==str6 ) << endl; // 输出什么？

分别输出false,false,true。

str1和str2都是字符数组，每个都有其自己的存储区，它们的值则是各存储区首地址，不等；

str3和str4同上，只是按const语义，它们所指向的数据区不能修改。

str5和str6并非数组而是字符指针，并不分配存储区，其后的“abc”以常量形式存于静态数据区，而它们自己仅是指向该区首地址的指针，相等。

# C++11特性

模板

Template<class T>

Class A

{

}

右值 int&&

Lamada表达 [](){}s

# 如何判断一段程序是C++还是C

图形用户界面, 应用程序

描述已自动生成

文本

描述已自动生成

图形用户界面, 文本

描述已自动生成

文本

描述已自动生成

# 重定义

一个变量被直接或者间接多次定义导致的错误

最常见的是头文件多次包含

 #pragma once则由编译器提供保证：同一个文件不会被包含多次。

#ifndef XXXX\_H

ifndef的方式依赖于宏名字不能冲突，这不光可以保证同一个文件不会被包含多次，也能保证内容完全相同的两个文件不会被不小心同时包含。

# 友元函数

#include <iostream> using namespace std; class Box { double width; public: friend void printWidth( Box box ); void setWidth( double wid ); }; // 成员函数定义 void Box::setWidth( double wid ) { width = wid; } // 请注意：printWidth() 不是任何类的成员函数 void printWidth( Box box ) { /\* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 \*/ cout << "Width of box : " << box.width <<endl; }