C++语言---顾甡源的总结

目录

[c++ 1](#_Toc4598390)

[C++中最基本的头文件和命名空间： 1](#_Toc4598391)

[标准输入与输出： 2](#_Toc4598392)

[关于c++中的类： 3](#_Toc4598393)

[为什么使用成员变量没有写成员函数会导致乱码： 4](#_Toc4598394)

[调用类中的函数方法： 4](#_Toc4598395)

[C++对c的加强的地方： 4](#_Toc4598396)

[实用性加强： 5](#_Toc4598397)

[Register关键字的增强： 5](#_Toc4598398)

[变量检测加强： 5](#_Toc4598399)

[Struct类型加强： 5](#_Toc4598400)

[对变量和函数的类型检查更加严格： 5](#_Toc4598401)

[C++中新增的bool类型关键字： 6](#_Toc4598402)

[C++中的三目运算符： 6](#_Toc4598403)

[Const的基础和const符号表机制研究： 6](#_Toc4598404)

[Const的好处： 6](#_Toc4598405)

[c++的引用： 8](#_Toc4598406)

[封装： 9](#_Toc4598407)

[继承： 10](#_Toc4598408)

[多态： 10](#_Toc4598409)

[Inline内联函数： 11](#_Toc4598410)

[函数参数 11](#_Toc4598411)

[函数重载： 12](#_Toc4598412)

[类的封装： 14](#_Toc4598413)

[构造和析造的使用： 16](#_Toc4598414)

[C++当中的析构函数格式： 16](#_Toc4598415)

[**引用：** 17](#_Toc4598416)

[引用应用 18](#_Toc4598417)

[常引用 19](#_Toc4598418)

[引用作为返回值 20](#_Toc4598419)

[引用和多态 22](#_Toc4598420)

[引用总结 23](#_Toc4598421)

[总结 24](#_Toc4598422)

[**第一章** 66](#_Toc4598423)

[**作用域限定符** 66](#_Toc4598424)

[**const常量** 67](#_Toc4598425)

[**常引用** 67](#_Toc4598426)

[**常对象** 69](#_Toc4598427)

[**常成员函数** 69](#_Toc4598428)

[**常数据成员** 70](#_Toc4598429)

[**带默认参数的函数** 70](#_Toc4598430)

[**new和delete** 70](#_Toc4598431)

[**预处理命令** 71](#_Toc4598432)

[第二章 71](#_Toc4598433)

[面向对象思想 71](#_Toc4598434)

[类的定义 71](#_Toc4598435)

[类的访问权限 72](#_Toc4598436)

[构造函数与析构函数 72](#_Toc4598437)

[拷贝构造函数 72](#_Toc4598438)

[深拷贝和浅拷贝 73](#_Toc4598439)

[析构函数 73](#_Toc4598440)

[This指针 73](#_Toc4598441)

[静态数据成员 74](#_Toc4598442)

[静态成员函数 75](#_Toc4598443)

[友元关系 75](#_Toc4598444)

[友元类 76](#_Toc4598445)

[第三章 76](#_Toc4598446)

[继承语法格式 76](#_Toc4598447)

[继承方式public，protected，private 76](#_Toc4598448)

[构造函数初始化格式 77](#_Toc4598449)

[析构顺序与构造顺序相反。 77](#_Toc4598450)

[多继承的构造函数 78](#_Toc4598451)

[虚基类 78](#_Toc4598452)

[虚基类的初始化 79](#_Toc4598453)

[**第四章** 79](#_Toc4598454)

[**多态** 79](#_Toc4598455)

[**动态绑定的条件：** 79](#_Toc4598456)

[**虚函数的覆盖** 80](#_Toc4598457)

[**虚函数的传递性** 80](#_Toc4598458)

[**抽象类** 80](#_Toc4598459)

[**函数重载** 81](#_Toc4598460)

[**调用拷贝构造函数的情况：** 82](#_Toc4598461)

[第五章 92](#_Toc4598462)

[函数模板 92](#_Toc4598463)

[函数模板重载 93](#_Toc4598464)

[类模板 93](#_Toc4598465)

[类模板的派生 94](#_Toc4598466)

[第六章 94](#_Toc4598467)

[标准模板库（STL） 94](#_Toc4598468)

[功能 94](#_Toc4598469)

[头文件 94](#_Toc4598470)

[说明 94](#_Toc4598471)

[函数对象 95](#_Toc4598472)

[常用通用容器 95](#_Toc4598473)

[list 97](#_Toc4598474)

[deque容器 98](#_Toc4598475)

[queue和stack容器 99](#_Toc4598476)

[STL常用算法 100](#_Toc4598477)

[第七章 102](#_Toc4598478)

[标准输入输出 102](#_Toc4598479)

[文件流 104](#_Toc4598480)

[打开外存文件 104](#_Toc4598481)

[打开文件包括两个方面内容 104](#_Toc4598482)

[可通过两种方式打开文件 105](#_Toc4598483)

[读写文本文件 106](#_Toc4598484)

[读写二进制文件 106](#_Toc4598485)

[第八章 107](#_Toc4598486)

[动态内存管理 107](#_Toc4598487)

[new运算符与delete运算符 108](#_Toc4598488)

[new使用方式 108](#_Toc4598489)

[delete使用方式 108](#_Toc4598490)

[关于内存的注意事项： 109](#_Toc4598491)

[常见动态内存错误 110](#_Toc4598492)

[指针和数组的差异 110](#_Toc4598493)

[利用指针传递内存 112](#_Toc4598494)

[函数参数传递方式 112](#_Toc4598495)

[类型转换 113](#_Toc4598496)

[基本类型转换为类类型 113](#_Toc4598497)

[类类型转换为基本类型 113](#_Toc4598498)

[命名空间 114](#_Toc4598499)

[c++ 135](#_Toc4598500)

[C++中最基本的头文件和命名空间： 135](#_Toc4598501)

[标准输入与输出： 136](#_Toc4598502)

[关于c++中的类： 137](#_Toc4598503)

[为什么使用成员变量没有写成员函数会导致乱码： 138](#_Toc4598504)

[调用类中的函数方法： 138](#_Toc4598505)

[C++对c的加强的地方： 138](#_Toc4598506)

[c++的引用： 142](#_Toc4598507)

[继承： 144](#_Toc4598508)

[多态： 144](#_Toc4598509)

[Inline内联函数： 145](#_Toc4598510)

[函数重载： 146](#_Toc4598511)

[类的封装： 148](#_Toc4598512)

[构造和析造的使用： 150](#_Toc4598513)

[**一、** **引用简介** 151](#_Toc4598514)

[引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。 152](#_Toc4598515)

[**二、引用应用** 152](#_Toc4598516)

[**三、引用总结** 158](#_Toc4598517)

[一、什么是引用 158](#_Toc4598518)

[二、引用的应用 161](#_Toc4598519)

[A.引用作为函数的参数 161](#_Toc4598520)

[B.常引用 163](#_Toc4598521)

[C.引用作为函数的返回值 165](#_Toc4598522)

[D.用引用实现多态 169](#_Toc4598523)

[三、总结 170](#_Toc4598524)

# c++

c++名字的由来：

C语言＋面向对象的方法论=objective c/c++

# C++中最基本的头文件和命名空间：

Namespace（命名空间）：

Namespace是指标识符的各种可见范围。C++标准程序库中的所有标识符都被定义于一个名为std的namespace中

<iostream>后缀为.h的头文件c++已经不支持。当使用iostream的时候，相当于在c中调用库函数，使用的是全局命名空间，必须使用namespace std这样才能正确使用cout

对于namespace的概念，使用c++标准的程序库的任何标志符的时候，可以有三种选择：

1. 直接指定标识符，比如std：：ostream而不是ostream，完整语句：std：：cout<<std::hex<<3.4<<endl;
2. 使用using关键字。Using std::cout;using std::endl;using std::cin;以上程序可以写为：cout<<std::hex<<3.4<<endl;
3. 最方便的就是using namespace std；为了避免程序员在选择在选择类的名称或函数名的时候可能引起冲突，所以把标准库中的一切全部放在空间std中

在c++文件中，在一开始一定要协商#include“iostream”//它的意思是包含c++的头文件

Using namespace std；//使用命名空间std标准的命名空间，在这个命名空间中定义了很多标准定义。

C语言中只有一个全局作用域，导致c语言所有的全部标识符共享一个作用域，这就导致了变量符之间可能发生冲突

C++提出了命名空间的概念，命名空间讲全局作用域分成不同的部分，不同的命名空间中的标识符可以同名而不会发生冲突。命名空间可以相互嵌套，全局作用域也叫默认命名空间

定义命名空间的格式：

Namespace XXX

{…}

命名空间里面可以嵌套

命名空间的使用：在主函数里面：using namespace XXX；

举一个例子：namespace C是嵌套在namespace B里面的，struct teacher是namespace C里面的一个结构体，这时候要调用这一个结构体是 using namespace B::namespace C：：teacher； 如果不这么使用那么在每次调用这个teacher结构体的t2数据的时候就是namespace B：：namespace C：：teacher t2；（相对于第一种方法显得比较繁琐）

# 标准输入与输出：

Cout相当于标准输出到黑屏幕上（cmd控制台）

<<左移操作符在c++中进行了功能的增强，这是c++语言操作符重载

Endl在c++里面是回车换行的意思

cout<<”hello”<<endl；可以理解为把hello流入到cout的黑窗户上，然后把endl的换行流入到黑窗户上

cin代表标准输入，代表键盘，表示从键盘输入一个数再流入这个变量

标准输入cin和标准输出cout一般都写在<<>>操作符的左边。

Cout和endl是定义在std中的，如果在文件的首端没有使用using namespace std；那么在后续使用cout和endl的时候要这么用：std：：cout；std：：endl；也就是说文件中iostream没有引入标准的std，需要程序员手工的写。

定义命名空间：命名空间的定义是可以嵌套的，只需要 namespace XXX就定义了一个命名空间

使用命名空间：using XXXX就可以在主函数中使用该命名空间里面的数据了。如果需要输出的变量在调用的多个命名空间中存在不止一个，那么使用方法是：XXXX：：a；

# 关于c++中的类：

Class是在struct的基础上建立起来的，c++是兼容c的

在c++中，把物体的特性叫做物体的属性

面向对象加工的是一个个的函数/语句，面向对象加工的是一个个的类，在类的里面包含函数和数据。

类就是一个抽象的数据类型，它和结构体不一样的地方之一是类能够在类里面写函数。

类的抽象：

1. 成员变量和成员函数
2. 实例化，就是用类定义变量，就是对象，类似于c语言中给生产多个结构体。
3. 使用类，求出结果
4. Main集成测试

类代码不是一步一步指向的

类是一个数据类型（固定大小内存块的别名），定义一个类，是一个抽象的概念，不会分配内存，用数据类型定义变量的时候，才会分配内存

为什么使用成员变量没有写成员函数会导致乱码：

因为一开始的时候成员变量是随机取值的，之后进行的操作只是改变了这个成员变量在当时的情况，在取出这个成员变量的时候会导致乱码（取出的是之前的随机量）取出的时候是从变量所标志的内存空间中拿值，并没有执行我们想要的赋值的情况，在通过类型定义变量的时候，值已经固定下来了。我的认为是在从外部传入数据的时候，不能改变private属性中的变量

从深层次的讲：

成员变量在类的范围内，存在于堆内存中，会有默认的初始值，在申明的同时已经给变量赋值了。再次赋值其实使用的是语句，而类里面只能出现变量和方法，不能出现语句。

调用类中的函数方法：类名.类中的函数名（参数表）；

C++对c的加强的地方：

1. namespace命名空间
2. 实用性的加强
3. Register关键字的加强
4. 变量检测的加强
5. Struct类型加强
6. C++中所有的变量和函数都必须有类型
7. 新增bool类型关键字
8. 三目运算符功能增强

实用性加强：

原来c语言中变量要提前定义，就是说c语言中的变量必须在作用域开始的位置定义，而c++中更强调语言的“实用性“，所有的变量都可以在需要使用的时候再定义。

Register关键字的增强：

请求编译器让变量a直接放在寄存器里面，速度快

再c语言中，register修饰的变量不能取地址，但是在c++中做出了改变：

C语言无法取得register变量地址，在c++中依然支持register关键字，c++编译器由自己的优化方式，不适用register也可能做优化，c++中可以取得register变量的地址

C++编译器发现程序需要取register变量的地址的时候，register对变量的声明变得无效，早期的c语言编译器不会对代码进行优化，因此register变量是一个很好的补充

在c语言中，寄存器变量上是不能取地址的。奇妙的是：在频繁使用某一个变量的时候c++编译器会自动优化这个变量为寄存器变量

变量检测加强：

在c语言中，重复定义多个同名的全局变量是合法的，在c++中，不允许，不允许定义多个同名的全局变量，c语言中多个同名的全局变量最终会连接到全局数据的同一个地址空间上，而c++直接拒绝这种二义性的做法。

Struct类型加强：

C语言中struct定义了一组变量的集合，c编译器并不认为这是一种新的类型

C++中的struct是一个新类型的定义声明，c++对struct关键字做了功能的增强

就是有些在c中部通过的结构体，在c++中通过了（比如struct关键字在c++中，可以在主函数外面定义。

Struct关键字和class关键字完成的功能是一样的，但是是有区别的

对变量和函数的类型检查更加严格：

C++中的变量和函数必须有类型（c++对变量类型的检查更加严格）：

比如在c语言中可以写 g（）{return 5}；但是在c++中是不行的，比如c++中对函数的参数表和函数的类型的检查更加严格，这些变量和函数都必须有类型。

C++中新增的bool类型关键字：

比如：bool b1=true；布尔类型占用的内存空间是一个字节

Bool关键字告诉c+++关键字给变量分配1个字节的内存、

对于bool变量，要么是1，要么是0。对于布尔变量，正数和负数都是真值，只有0布尔变量为假。

理论上bool只占用一个字节，如果多个bool变量定义在一起，可能会各占一个bit，这取决于编译器的实现。

C++中的三目运算符：

在c语言中，表达式的运算结果放在寄存器里面，表达式返回的是值，而不是内存中，它返回的是一个数，三目运算符不可以做左值。左值应该是一个变量的性质，不是一个值。

在c++中，表达式返回的是变量的本身，三目运算符返回的是一个地址，可以作为左值。

那么如何让c语言也做到：让表达式返回一个内存地址。把这个值导向变成地址导向（使用指针）

本质：c++编译器帮助程序员完成了取地址的工作

# Const的基础和const符号表机制研究：

被const修饰的东西不能被修改。需要观察const修饰的是指针内存还是变量。

顺带一提：const int a；和int const a；是一样的，代表一个常整型数。

Const int \*c;//c是一个指向常整型数的指针，所指向的内存数据不能更改，但是本身是可以更改的

Int\*const d;//d常指针，指针变量不能被修改，但是它指向的内存空间可以被修改

Const int\*const e;//e是一个指向常整形的常指针，指针和它指向的内存空间都不可以修改

Const的好处：

1、指针做函数参数，可以有效的提高代码可读性，减少bug

2、清除的分清参数的输入和输出特性

C语言中的const是一个冒牌货，虽然直接赋值是不可行的，但是通过指针地址来进行间接赋值还是可以的。

但是在c++中，只要使用了const，无论直接赋值还是间接赋值，c++都只承认第一次的赋值。在c++中，const是一个真正的常量

因为c++编译器在看到const的修饰的时候，不再给const的变量分配内存。在c++中，const的量是存在于符号表中的，他的值被固定。当发现使用常量则直接从符号表中寻找对应的值兑换。编译过程中如果发现对const使用了extern或者&操作符，则给对应的常量分配内存空间（兼容c）。但是当像c++编译器索取const量的地址的时候，c++编译器会给这个量开辟一块内存空间。

Const和define的相同之处：

举例子：int a=10；

Int b=20；

Int array【a+b】；//这种情况下编译是不通过的，但是在linux内核里面是可以通过的，因为编译linux内核的gcc编译器支持，c和c++都不支持。

但是，如果在上文中，a和b是用const 或者宏定义#define做的，那么能够编译

const和define的不同之处：

const常量是由编译器处理的，提供类型检查和作用域检查

宏定义由预处理器处理，单纯的文本替换

注：undef和define互为逆

const int \*p //指向 const int 类型对象的指针

int const \*q; //同上

int \*const x; //指向 int 类型对象的 const 指针；注意 const 的位置

const int \*const r; //指向 const int 类型对象的 const 指针

int const \*const t; //同上

# c++的引用：

在c++中新增加了引用的概念，引用必须要初始化，依赖于一个变量。

引用可以看作一个一定义变量的别名，两个东西本质上就是一个东西了

引用语法举例：

Int&a=b；引用作为函数参数声明时不进行初始化

复杂数据类型做函数引用：

引用（起别名）也可以用于结构体的成员，或整个结构体名。

引用的意义：

引用作为其他变量的别名而存在，因此在一些场合可以取代指针

引用相对于指针来说具有更好的可读性和实用性

引用本质探究：

普通引用也占有内存空间，很像是指针占据的空间

引用在c++内部实现是一个常量指针，因此引用所占用的空间大小和指针相同。

从使用的角度，引用会让人误会只是一个别名，没有自己的存储空间，这是c++为了实用性而做出的细节隐藏

引用的结论：

引用在实现上，只不过是把间接赋值成立的三个条件的后两步合二为一

当实参传给形参引用的时候，只不过是c++编译器帮我们程序员手工取了一个实参地址，传给形参引用（常量指针）

函数的返回值当引用：

比如：int getAA190

int& getAA2（）

c++引用使用时的难点：

当函数返回值为引用时，不能成为其他引用的初始值，不能作为左值引用

若返回静态变量或全局变量，可以成为其他引用的初始值，即可以作为右值使用，也可以作为左值使用

# 封装：

将数据和处理数据的程序组合起来，仅对外公开接口，达到信息隐藏的功能。

优点：能减少耦合。

耦合：某两个事物之间存在一种相互作用、相互影响的关系。

C++的类对其成员（包括数据成员、函数成员）分为三种封装状态：

公有（public）：类别的用户可以访问、使用该类别的此种成员。

保护（protected）：该类别的派生类可以访问、使用该类别的此成员。外部程序代码不可以访问、使用这种成员。

私有（private）：只有类别自身的成员函数可以访问、使用该类别的此成员。一般可以将C++类的对外接口设定为公有成员；类内部使用的数据、函数设定为私有成员；供派生自该类别的子类使用的数据、函数设定为保护成员。

# 继承：

派生类继承基类，会自动获取超类别除私有特质外的全部特质，同一类别的所有实体都会自动有该类别的全部特质，做到代码再用。C++只支持类别构成式继承，虽然同一类别的所有实体都有该类别的全部特质，但是实体能够共享的实体成员只限成员函数，类别的任何实体数据成员乃每个实体独立一份，因此对象间并不能共享状态，除非特质为参考类型的属性，或使用指针来间接共享。C++支持的继承关系为：

公有继承：最常用继承关系，含义是“is-a”关系，代表了在完全使用公有继承的对象类别之间的层次关系。

受保护继承：基类的公有或保护内容可以被派生类，以及由此派生的其他类别使用。但是基类对外界用户是不可见的。派生类的用户不能访问基类的成员、不能把派生类别转换（造型）为基类的指针或引用。

私有继承：基类的公有或保护内容仅可以被派生类访问。但基类对派生类的子类或派生类的用户都是不可见的。派生类的子类或派生类的用户都不能访问基类的内容、不能把派生类转换为基类的指针或引用。

C++支持多继承。

# 多态：

除了封装与继承外，C++还提供了多态功能，面向对象的精神在于多态，一般的多态，是指动态多态，系使用继承和动态绑定实现，使用多态可创建起继承体系。

类与继承只是达成多态中的一种手段，所以称面向对象而非面向类。  
多态又分成静态多态与动态多态。C++语言支持的动态多态必须结合继承和动态绑定方式实现。静态多态是指编译时决定的多态，包括重载和以模板实现多态的方法即参数化类型，是使用宏的“程序代码膨胀法”达到多态效果。

类型转换也是一种非参数化多态的概念，C++提供dynamic\_cast, static\_cast等运算符来实现强制类型转换。

操作数重载或函数重载也算是多态的概念。

在c语言中，&时取地址符，用在函数传参中的指针赋值；但是在c++中，&还能够表 示引用，可以提高代码执行效率，增加代码质量。

# Inline内联函数：

内联函数必须和函数体实现的地方写在一起

C++编译器直接将定义的内联函数的函数体插入到函数调用的地方

内联函数在最终生成的代码中是没有意义的

Inline函数只是编译器的加工，是一种请求，但是编译器不一定允许这种请求

内联函数省去了普通函数调用时压栈，跳转和返回的开销

关于内联函数，会接触到宏替换，宏替换的时候会碰到关于宏展开的知识点

内联函数在定义的时候需要注明时inline形式

# 函数参数

默认参数

c++在函数定义的时候为参数定义一个默认值，也就是说在c++中在调用自己定义的函数的时候，如果调用的括号里没有给参数赋值，那么c++编译器会自动给它一个默认值，而不会直接报错。

函数默认参数的规则：

只有函数列表后面部分的参数才可以提供默认参数值

一旦在一个函数调用中开始使用默认参数值，那么在这个自定义函数里面的后面的所有的这个参数都必须使用这一个参数值。

函数的占位参数：

在函数调用的时候必须写够参数。占位参数就是在函数的参数表里面有一个int ，但是没有变量名

默认参数和占位参数：

可以将占位参数与默认参数结合起来使用，它的意义就是为以后的程序拓展留下线索，兼容c语言程序中可能出现的不规则的写法

# 函数重载：

重载（overload）函数名相同，但是搭配不同的参数（个数，类型）

用一个函数名定义不同的参数，当参数不一样的时候，达到不同的蕴意，同一个函数名的函数能够得到不同的对参数的处理。函数名和不同参数搭配的时候函数的含义不同

函数的返回值不是函数重载的判断标准，除此之外函数名前面的返回值类型在函数重载上是不可以不一样的

重载函数的调用标准：

将所有同名函数作为候选者

尝试寻找可行的候选参数

精确匹配实参

通过默认参数能够匹配实参

匹配失败

最终寻找到的可行候选参数不唯一，那么出现二义性，编译失败

无法匹配所有候选者，函数未定义，编译失败

函数重载在本质上是不同的函数（静态链编）

重载函数的函数类型是不同的

函数返回值不能作为函数重载的依据

函数重载是由函数名和参数列表决定的

当默认参数遇见函数重载：

这样子函数调用时出现二义性，会失败

函数重载和函数指针在一起：

函数可以赋值给指针，之后就可以这样子使用P（1）；

函数指针：

声明一个函数类型

定义一个指针，这个指针指向函数的入口地址

声明一个指针的数据类型

通过函数指针类型定义一个函数指针

定义一个函数指针 变量

类的封装和访问控制：

面向对象的三大概念：封装，继承，多态

# 类的封装：

封装的含义：

1. 对属性和方法进行封装
2. 对成员的量和成员进行访问控制

一个理念：

类的内部和类的外部：

定义类的函数体

类的外部：在定义类的部分

类的访问控制关键字：

Public：修饰的成员变量和函数，可以在类的内部和类的外部访问

Private：修饰的成员变量和成员函数，只能在类的内部被访问，不能在类的外部访问

Protected：修饰的变量和函数，只能在类的内部被访问，不能在类的外部访问。但是在继承里面，protected可以在子类里面使用。

如果在类的外部使用调用非public的成员，会被编译器报错

类的封装，public是这个封装的接口

在c++中，没有权限修饰的成员变量函数都默认为私有属性

Struct和class关键字的区别：

Struct关键字定义的类，默认是public

类的声明和类的实现：

就是定义一个自己想要的类，然后在主函数中使用这些类，调用出所需要的类中的成员

对象的构造和析构：

创造一个对象的时候，常常需要做某些初始化的工作，例如对数据成员赋初值。注意，类的数据成员是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，而是在建立对象的时候自动执行。

关于构造函数：

C++中的类可以定义与类名相同的特殊成员函数，这种与类名相同的成员函数叫做构造函数

构造函数在定义的时候可以由参数

没有任何返回类型的声明

构造函数的调用：

自动调用：一般情况下，c++编译器会自动调用构造函数

手动调用：在一些情况下则需要手工调用构造函数

有关构造函数：

析构函数的定义及调用：

C++中的类可以定义一个特殊的成员函数清理对象，这个特殊的成员函数叫做析构函数

语法：~calssname（）

析构函数没有参数也没有任何返回类型的声明

析构函数在对象销毁时自动被调用

析构函数的调用机制：c++编译器自动调用

析构函数是使用在类的内部的

# 构造和析造的使用：

析构函数与构造函数相反，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统自动执行析构函数。析构函数往往用来做“清理善后” 的工作（例如在建立对象时用new开辟了一片内存空间，delete会自动调用析构函数后释放内存）。

析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。 [1]

C++当中的析构函数格式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class <类名>  {       public:         ~<类名>();  };  <类名>::~<类名>()  {      //函数体  } |

如以下定义是合法的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class T  {     public:      ~T()；  };      T::~T()  {      //函数体  } |

当程序中没有析构函数时，系统会自动生成以下析构函数：

<类名>::~<类名>(){}，即不执行任何操作。

**引用：**

　　引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。

　　引用的声明方法：类型标识符 &引用名=目标变量名；

　　【例1】：int a; int &ra=a; //定义引用ra,它是变量a的引用，即别名

　　说明：

　　（1）&在此不是求地址运算，而是起标识作用。

　　（2）类型标识符是指目标变量的类型。

　　（3）声明引用时，必须同时对其进行初始化。

　　（4）引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，且不能再把该引用名作为其他变量名的别名。

　　 ra=1; 等价于 a=1;

　　（5）声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。故：对引用求地址，就是对目标变量求地址。&ra与&a相等。

　　（6）不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。

引用应用

**1、引用作为参数**

　　引用的一个重要作用就是作为函数的参数。以前的C语言中函数参数传递是值传递，如果有大块数据作为参数传递的时候，采用的方案往往是指针，因为 这样可以避免将整块数据全部压栈，可以提高程序的效率。但是现在（C++中）又增加了一种同样有效率的选择（在某些特殊情况下又是必须的选择），就是引 用。

　　【例2】：

void swap(int &p1, int &p2) //此处函数的形参p1, p2都是引用

{ int p; p=p1; p1=p2; p2=p; }

　　为在程序中调用该函数，则相应的主调函数的调用点处，直接以变量作为实参进行调用即可，而不需要实参变量有任何的特殊要求。如：对应上面定义的swap函数，相应的主调函数可写为：

main( )

{

　int a,b;

　cin>>a>>b; //输入a,b两变量的值

　swap(a,b); //直接以变量a和b作为实参调用swap函数

　cout<<a<< ' ' <<b; //输出结果

}

　　上述程序运行时，如果输入数据10 20并回车后，则输出结果为20 10。

　　由【例2】可看出：

　　（1）传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

　　（2）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给 形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效 率和所占空间都好。

　　（3）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用"\*指针变量名"的 形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

　　如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。

常引用

　　常引用声明方式：const 类型标识符 &引用名=目标变量名；

　　用这种方式声明的引用，不能通过引用对目标变量的值进行修改,从而使引用的目标成为const，达到了引用的安全性。

　　【例3】：

int a ;

const int &ra=a;

ra=1; //错误

a=1; //正确

　　这不光是让代码更健壮，也有些其它方面的需要。

　　【例4】：假设有如下函数声明：

string foo( );

void bar(string & s);

　　那么下面的表达式将是非法的：

bar(foo( ));

bar("hello world");

　　原因在于foo( )和"hello world"串都会产生一个临时对象，而在C++中，这些临时对象都是const类型的。因此上面的表达式就是试图将一个const类型的对象转换为非const类型，这是非法的。

　　引用型参数应该在能被定义为const的情况下，尽量定义为const 。

引用作为返回值

　　要以引用返回函数值，则函数定义时要按以下格式：

类型标识符 &函数名（形参列表及类型说明）  
{函数体}

　　说明：

　　（1）以引用返回函数值，定义函数时需要在函数名前加&

　　（2）用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

　　【例5】以下程序中定义了一个普通的函数fn1（它用返回值的方法返回函数值），另外一个函数fn2，它以引用的方法返回函数值。

#include <iostream.h>

float temp; //定义全局变量temp

float fn1(float r); //声明函数fn1

float &fn2(float r); //声明函数fn2

float fn1(float r) //定义函数fn1，它以返回值的方法返回函数值

{

　temp=(float)(r\*r\*3.14);

　return temp;

}

float &fn2(float r) //定义函数fn2，它以引用方式返回函数值

{

　temp=(float)(r\*r\*3.14);

　return temp;

}

void main() //主函数

{

　float a=fn1(10.0); //第1种情况，系统生成要返回值的副本（即临时变量）

　float &b=fn1(10.0); //第2种情况，可能会出错（不同 C++系统有不同规定）

　//不能从被调函数中返回一个临时变量或局部变量的引用

　float c=fn2(10.0); //第3种情况，系统不生成返回值的副本

　//可以从被调函数中返回一个全局变量的引用

　float &d=fn2(10.0); //第4种情况，系统不生成返回值的副本

　//可以从被调函数中返回一个全局变量的引用

　cout<<a<<c<<d;

}

　　引用作为返回值，必须遵守以下规则：

　　（1）不能返回局部变量的引用。这条可以参照Effective C++[1]的Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

　　（2）不能返回函数内部new分配的内存的引用。这条可以参照Effective C++[1]的Item 31。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一 个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak。

　　（3）可以返回类成员的引用，但最好是const。这条原则可以参照Effective C++[1]的Item 30。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常 量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

　　（4）引用与一些操作符的重载：

　　流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl;　因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回 一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于返回一 个流指针则不能连续使用<<操作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这 就是C++语言中引入引用这个概念的原因吧。 赋值操作符=。这个操作符象流操作符一样，是可以连续使用的，例如：x = j = 10;或者(x=10)=100;赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

　　【例6】 测试用返回引用的函数值作为赋值表达式的左值。

#include <iostream.h>

int &put(int n);

int vals[10];

int error=-1;

void main()

{

put(0)=10; //以put(0)函数值作为左值，等价于vals[0]=10;

put(9)=20; //以put(9)函数值作为左值，等价于vals[9]=20;

cout<<vals[0];

cout<<vals[9];

}

int &put(int n)

{

if (n>=0 && n<=9 ) return vals[n];

else { cout<<"subscript error"; return error; }

}

　　（5）在另外的一些操作符中，却千万不能返回引用：+-\*/ 四则运算符。它们不能返回引用，Effective C++[1]的Item23详细的讨论了这个问题。主要原因是这四个操作符没有side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、返回一个局部变量的引用，返回一个new分配的对象的引用、返回一 个静态对象引用。根据前面提到的引用作为返回值的三个规则，第2、3两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为true而导致错误。所以可选的只剩下返回一个对象了。

引用和多态

　　引用是除指针外另一个可以产生多态效果的手段。这意味着，一个基类的引用可以指向它的派生类实例。

　　【例7】：

class 　A;

class 　B：public A{……};

B 　b;

A 　&Ref = b; // 用派生类对象初始化基类对象的引用

　　Ref 只能用来访问派生类对象中从基类继承下来的成员，是基类引用指向派生类。如果A类中定义有虚函数，并且在B类中重写了这个虚函数，就可以通过Ref产生多态效果。

引用总结

　　（1）在引用的使用中，单纯给某个变量取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不如意的问题。

　　（2）用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，且通过const的使用，保证了引用传递的安全性。

　　（3）引用与指针的区别是，指针通过某个指针变量指向一个对象后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。

　　（4）使用引用的时机。流操作符<<和>>、赋值操作符=的返回值、拷贝构造函数的参数、赋值操作符=的参数、其它情况都推荐使用引用。

**语法：类型 &引用名=目标变量名；**

**特别注意：**

1.&不是求地址运算符，而是起标志作用

2.引用的类型必须和其所绑定的变量的类型相同

3.声明引用的同时**必须对其初始化**，否则系统会报错

4.引用相当于变量或对象的别名，因此**不能再将已有的引用名作为其他变量或对象的名字或别名**

5.引用不是定义一个新的变量或对象，因此**内存不会为引用开辟新的空间存储这个引用**

6.对数组的引用

**语法：类型 (&引用名)[数组中元素数量]=数组名；**

7.对指针的引用

**语法：类型 \*&引用名=指针名;//可以理解为：（类型\*） &引用名=指针名，即将指针的类型当成类型\***

**特别注意：**

1.当用引用作为函数的参数时，其效果**和用指针作为函数参数的效果相当**。当调用函数时，函数中的形参就会被当成实参变量或对象的一个别名来使用，也就是说**此时函数中对形参的各种操作实际上是对实参本身进行操作，而非简单的将实参变量或对象的值拷贝给形参**。

2.通常函数调用时，系统采用值传递的方式将实参变量的值传递给函数的形参变量。此时，系统会在内存中开辟空间用来存储形参变量，并将实参变量的值拷贝给形参变量，也就是说形参变量只是实参变量的副本而已；并且如果函数传递的是类的对象，系统还会调用类中的拷贝构造函数来构造形参对象。而**使用引用作为函数的形参时，由于此时形参只是要传递给函数的实参变量或对象的别名而非副本，故系统不会耗费时间来在内存中开辟空间来存储形参**。因此**如果参数传递的数据较大时，建议使用引用作为函数的形参，这样会提高函数的时间效率，并节省内存空间**。

3.使用指针作为函数的形参虽然达到的效果和使用引用一样，但**当调用函数时仍需要为形参指针变量在内存中分配空间**，而引用则不需要这样，故**在C++中推荐使用引用而非指针作为函数的参数**

4.**如果在编程过程中既希望通过让引用作为函数的参数来提高函数的编程效率，又希望保护传递的参数使其在函数中不被改变，则此时应当使用对常量的引用作为函数的参数。**

5.数组的引用作为函数的参数：C++的数组类型是带有长度信息的，**引用传递时如果指明的是数组则必须指定数组的长度**

总结

1.在引用的使用中，单纯给某个变量去别名是毫无意义的，**引用的目的主要用于在函数参数的传递中，解决大块数据或对象的传递效率和空间不如意的问题**

2.用引用传递函数的参数，能**保证参数在传递的过程中不产生副本**，从而提高传递效率，同时**通过const的使用，还可以保证参数在传递过程中的安全性**

3.引用本身是目标变量或对象的别名，对引用的操作本质上就是对目标变量或对象的操作。因此**能使用引用时尽量使用引用而非指针**

C++对函数的扩展：

Inline内联函数：

内联函数可以替换宏代码片段，使用inline声明内联函数，声明的时候inline关键字必须和定义结合在一起，否则编译器会直接忽略内联请求

Day3：

copy构造函数的调用：

函数返回值是匿名对象的去和留的剖析：

浅拷贝问题：

深拷贝问题：

构造函数的初始化列表：

构造和析构调用顺序：

匿名对象生命周期：

构造中调用构造（产生匿名对象）：

New和delete：

静态成员变量和静态成员函数：

This指针：

全局函数和成员函数：

友元函数：

友元类：

运算符重载：

成员函数和友元函数完成一元运算符重载：

友元函数实现操作符重载知识总结：

重载符号操作符：

绪论：

机器语言：

CPU指令系统，由0，1序列构成的指令码组成。

比如：10000000表示加，10010000表示减

汇编语言：

用助记符号描述的指令系统，如：ADD A,B以上两者是面向机器的语言

程序设计是数据被加工的过程：

客观世界可以分类，对象是类的实例

对象是数据和方法的封装

对象间通过发送和接受消息发生联系

面向对象的语言：

出发点：更加直接的描述客观世界中存在的事物（对象）以及他们之间关系

特点：高级语言

将 客观事物看作具有属性和行为的对象

通过抽象找出同一类对象的共同属性和行为，形成类

通过类的继承与多态实现代码重用

面向对象：

将数据以及对数据的操作方法封装在一起作为一个相互依存，不可分离的整体—对象

对同类型对象抽象出其共性，形成类

类通过一个简单的外部接口，与外界发生关系

对象与对象之间通过消息进行通信

对象具有：静态特征：可以用某数据来描述

动态特征：对象所表现的行为或具有的功能

对象：由一组属性和一组行为构成

属性：用来描述对象静态特征的数据项

行为：用来描述对象动态特性的操作序列

类：

分类是人类通常的思维方式

抽象是分类所依据的原则

抽象：忽略事物的非本质特性，只注意那些与当前目标有关的本质特征，从而找出事物的共性，把具有共同性质的事物划分为一类，得出一个抽象的概念

封装：

把对象的属性和服务结合成一个独立的系统单元

尽可能地隐蔽对象地内部细节，对外形成一个边界（或者说一个屏障），只保留有限地对外接口使之与外部发生关系。

继承：

特殊类地对象拥有其一般类地全部属性与服务，称作特殊类对一般类地继承

多态性：

在一般类中定义地属性或行为，被特殊类继承以后，可以具有不同地数据类型或表现出不同地行为。这使得同一个属性或行为在一般类以及各个特殊类中具有不同地语义

比如：数地加法—实数地加法—复数地加法

注意：对补码再次求补得到地是原码

若运算结果溢出，那么负数之和得正数，或正数之和得负数

关于变量名：

源代码中，变量依靠变量名来标识

目标代码中，变量依靠地址来标识，每个变量地地址互不相同

第三章节 函数

函数的定义和调用

函数的定义

函数是面向对象的程序设计中，对功能的抽象

函数定义的语法形式：

类型标识符/\*如果没有返回值，写void\*/ 函数符 （形式参数表）/\*是被初始化的内部变量，寿命和可见性都仅限在函数的内部）

函数的调用：

调用前先申明函数

如果函数的定义在调用点之前，则无需另外说明

如果函数定义在调用点之前，则需要在调用函数前按下如下形式申明函数原型：

类型标识符 被调用函数名（含类型变量的形参表）

调用形式：函数名（实参列表）

嵌套调用：函数可以嵌套调用，但是函数不可以嵌套定义

在c++中：

Rand

函数原型： int rand(void)

所需要的头文件：<cstdlib>

功能和返回值：求出并返回一个伪随机数

Srand

函数原型：void srand (unsigned int seed);

参数：seed产生随机数的种子

所需头文件：<cstdlib>

功能：为使rand（）产生一系列伪随机整数而设置起始点。使用1作为seed参数，乐意重新初始化rand（）

函数的参数传递机制—传递参数值

在函数被调用时才分配形参的存储单元

实参可以是常量、变量或表达式

实参类型必须与形参相符

传递时是传递参数值，即单向传递

函数的参数传递—用引用做形参

引用（&）是标识符的别名

声明一个引用时，必须同时对它进行初始化，使它指向一个已存在的对象，一旦一个引用被初始化后，就不能改为指向其他对象

引用可以作为形式参数

内联函数的声明与使用

声明时使用关键字inline

编译时在调用处用函数体进行替换，节省了参数擦混敌、控制转移等开销

注意：内联函数体内不能有循环语句和switch语句

内联函数的声明必须出现在内联函数第一次被调用之前

对内联函数不能进行异常接口声明

缺省形参值的作用

函数在声明时可以预先给出缺省的形参值，调用时如给出实参，则采用参数值，否则采用预先给出的缺省形参值

Int add（int x=5，int y=6）

{

Return x+y；

}

int main()

{

Add(10);//10+6

}

缺省形参数的形参必须在形参列表的最后也就是说缺省形参值的右边不能有无缺省值的参数。因为调用时实参与形参的结合是从左向右的顺序

重载函数的声明：

C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成重载。方便使用，便于记忆。

比如：int add(int x,int y)

Float add(float x,float y);

以上两个是形参类型不同

还有的情况是形参个数不同

注：重载函数的形参必须不同：个数不同或者类型不同

编译程序将根据实参和形参的类型以及个数的最佳匹配来选择调用哪一个函数

不要将不同功能的函数声明为重载函数，以免出现调用结果的误解，混淆

标准函数和非标准函数：

标准c++函数

1、c++标准中规定的函数

2、各种编译环境普遍支持，因此使用标准函数的程序移植性好

3、很多标准c++函数继承自标准c，头文件以c开头：cmath cstdlib cstdio ctime

非标准c++函数

1、与特定操作系统或编译环境相关

2、在处理和操作相关事务时常常需要调用

再次提一下栈：

栈是一种容纳数据的容器，数据只能从栈的一段存入（压入栈），数据只能从栈的同一段取出（弹出栈）

运行栈：

运行栈是一段区域的内存空间

运行栈分为一个一个栈帧。每一个栈帧对应一次函数调用

栈帧中包括：

本次函数调用的形参值

控制信息

局部变量值

一些临时数据

函数返回时，会有一个栈帧被压入栈，返回时会有一个栈帧被弹出

第四章节 类与对象

OOP的基本特点：抽象

抽象是对具体对象（问题）进行概括，抽出这一类对象的公共性质并加以描述的过程

先注意问题的本质及描述，其次是实现过程或细节

数据抽象：描述某类对象的属性或状态（对象相互区别的物理量）

代码抽象：描述某类对象的共有的行为特征或具有的功能

抽象的实现：通过类的定义

抽象的实例：钟表

数据抽象：int hour,int minute,int second

代码抽象：setTime(),showTime()

抽象实例—钟表类

Class Clock{

Public:

Void setTime(int newH,int newM,int newS);//这是一个外部接口

Void showTime();//这也是一个外部接口

Private://特定的访问权限

Int hour,minute,second;

};

//这边叫做边界

封装：将抽象出的数据成员、代码成员相结合，将它们视为一个整体

目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只需要通过外部接口，以特定的访问权限，来使用类的成员。

实现封装：类定义中的{}

继承与派生：是c++中支持层次分类的一种机制，允许程序员在保持原有类特性的基础上进行更具体的说明。

实现：定义派生类

多态性：多态：同一名称，不同功能实现方式

目的：达到行为标志统一，减少程序中标识符的个数

实现：重载函数和虚函数

C++中的类：

类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分

利用类可以实现数据的封装、隐藏、集成和派生

利用类易于编写大型复杂的程序，并将其模拟化程度比c中采用函数更高

类的定义形式：类是一种用户自定义类型，它的定义形式为：

Class类名称

{

Public：

公有成员（外部接口）

Private：私有成员

Protected：

保护型成员

}

公有类型成员：

在关键字public后面声明，他们是类与外部的接口，任何外部函数都可以访问共有类型数据和函数

私有类型成员：

在关键字private后面声明，只允许本类中的函数访问，而类外部的函数都不能访问。

如果紧跟在类名称后面声明私有成员，则关键字private可以省略

保护类型：与private类似，其差别表现在继承与派生类的影响不同

类的成员：

class clock{

public:

void setTime(int newH,int newM,int newS);

void showTime();

private:

int hour,minute,second;

};//其中setTime,showTime是成员函数，hour,minute,second是成员数据

成员数据：与一般的变量声明相同，但需要将它放在类的定义体中

成员函数：在类中说明原型，可以在类外给出函数体实现，并在函数名前使用类名加以限定。也可以直接在类中给出函数体，形成内连成员函数。

允许声明重载函数和带默认形参值的函数

内联成员函数：

为了提高运行时的效率，对于比较简单的函数可以声明为内联形式

内联函数体中不要有复杂结构

在类中声明内联成员函数的方式：将函数体放在类的定义中，使用inline关键字

对象：类的对象时该类的某一特定实体，即类类型的变量

定义形式：类名 对象名；

比如：Clock myClock；

类中成员的访问方式：

类中成员互访：直接使用成员名

类外访问：使用“对象名.成员名”方式访问public属性的成员

构造函数：构造函数的作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态

在对象创建时由系统自动调用

如果程序中并未说明，则系统自动产生一个隐含的参数列表为空的构造函数

允许为内联函数，重载函数，带默认形参值的函数

拷贝构造函数：

拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用。

class 类名{

public:

类名（形参）；//构造函数

类名（类名&对象名）；//拷贝构造函数

}；

类名：：类（类名&对象名）//拷贝构造函数的实现

{函数体}

当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现复制

若函数的形参为类对象，调用函数时。实参赋值给形参，系统自动调用拷贝构造函数

当函数的返回值是类对象时，系统自动调用拷贝构造函数

隐含的老贝构造函数：

如果程序员没有为类声明拷贝初始化构造函数，则编译器自己生成一个隐含的拷贝构造函数

这个构造函数执行的功能是：用作为初始值的对象的每一个数值成员的值，初始化将要建立的对象的对应数据成员

后遭函数：

完成对象被删除前的一些清理工作

在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间

如果程序中未声明析构函数，编译器将自动产生一个隐含的析构函数

组合的概念：

类中的成员数据是另一个类的对象

可以在已有抽象的基础上实现更复杂的抽象

类组合的构造函数设计：

原则：不仅要负责对本类中的基本类型成员赋初值，也要对对象成员初始化。

定义形式：类名：：类名（对象成员所需的形参，本类成员形参）：对象1（参数），对象2（参数）

类组合的构造函数调用：

构造函数调用顺序：先调用内嵌对象的构造函数）按照内嵌时的声明顺序，先声明者先构造）。然后调用本类的构造函数（析构函数的调用顺序相反）

初始化列表中未出现的内嵌现象，用默认构造函数）（即无形参的）初始化

系统自动生成的隐含的默认构造函数中，内嵌对象全部用默认构造函数初始化

不能定义该类的对象，也不能再内联成员函数中使用该类的对象

结构体时一种特殊形态的类：

与类的唯一区别：类的缺省访问权限时private，结构体的缺省访问权限是public

结构体存在的主要原因：与c语言保持兼容

什么时候用结构体不用类：

定义主要用来保存数据而没有什么操作的类型

人们习惯将结构体的数据成员设为公有，因此这个时候使用结构体会更加方便

第五章节：数据的共享与保护

函数原型的作用域，函数原型中的参数，其作用域始于（，结束于）

局部作用域：函数的形参，在块中声明的标识符，其作用域自声明处起，

C++语言中给变量赋值的两个方式：

Int count(0);int count=0;

在c++中，字符串是类的类型，用双引号表示，可以执行字符串的比较，连接。

如果一个函数没有参数列表，那么可以不写，但是括号一定要有

在函数使用地址传递的时候，内部参数地址的改变会影响外部参数的改变，这就是地址传递的特性。

程序的内存区域：

1. 代码区，存放程序的代码，即各个程序中的代码块
2. 全局数据区，存放程序全局变量和静态数据
3. 堆区，存放程序的动态数据
4. 栈区，存放程序的局部变量，即各个程序中的数据

strcpy(dst, src) 将字符从 src拷贝到dst。函数的返回值是dst的地址

strncpy(dst, src, n) 至多从 src 拷贝n个字符到dst。函数的返回值是dst的地址

strcat(dst, src) 将 src 接到 dst 后。函数的返回值是dst的地址

strncat(dst, src, n) 从 src 至多取 n 个字符接到 dst 后。函数的返回值是dst的地址

strlen(s) 返回s的长度

strcmp(s1, s2) 比较 s1 和 s2。如 s1 > s2 返回值为正数，s1=s1返回值为0，s1< s2返回值为负数

strncmp(s1, s2, n) 如 strcmp，但至多比较n个字符 strchr(s, ch) 返回一个指向s中第一次出现ch的地址 strrchr(s, ch) 返回一个指向s中最后一次出现ch的地址 strstr(s1, s2)

返回一个指向s1中第一次出现s2的地址

在c++中的动态分配：

运算符new用于进行内存分配：

P=new type；

P=new type【size】；//分配数组

比如：p=new int【100】；

P=new type（初值）；

运算符delete释放new分配的内存：

Delete p；

Delete 【】 p；

在c++语言中输入一整个字符串（可以包括空格的情况）这么使用：getline（cin，str）；

对象具有动态特性和静态特性：

静态特性：可以用某数据来描述

动态特性：对象所表现的行为或具有的功能

对象：由一组数据和一组行为构成

属性：用来描述对象静态特征的数据项

行为：用来描述对象动态特征的操作序列

类：分类是人类通常的思维方式

抽象是分类所依据的原则

抽象：忽略事情的非本质特性，只注意那些与当前目标有关的本质特征，从而找出事情的共性，把具有共同性质的书屋划分为一类，得出一个抽象的概念

封装：

把对象的属性和服务结合成一个独立的系统单元

尽可能隐蔽对象的内部细节，对外形成一个边界（或者说一个屏障），只保留有限的对外接口使之与外部发生联系

继承：

特殊类的对象拥有其一般类的全部属性与服务，称作特殊类对一般类的继承。

继承是面向对象的方法的一个主要特征。继承是使用已有的类定义作为基础建立新类的定义技术。已有的类可以当作基类来引用，则新类相应的可以作为派生类来引用。广义的说，继承是指能够直接获得已有的性质和特征，而不必重复定义他们。

及城市一种使用户得以在一个类地基础上建立新的类地技术。新的类自动继承旧的类地对象数据属性和行为属性，并可具备某些附加地特征或某些限制。新类称作酒类地子类，旧类称作新类地父类。继承能有效地支持软件构件地重用，使得当需要在系统中增加新的特性的时候，所需要地新的代码最少。并且当继承和多态，动态联编结合使用地时候，为修改系统所需变动地源代码最少。继承机制地强有力地地方还在于它允许程序设计人员可以重用一个班未必完全符合要求地类，允许对该类进行修改而不至于在该类地其他部分引起有害地副作用。

继承分为单继承和多继承。单继承是指一个类只允许有一个父类，即类等级为树形结构。多重继承是指一个类允许有多个父类，多重继承的类可以组合多个父类的性质构成所需要的性质。但是使用多重继承的时候要注意避免二义性。继承性的优点就是相似的对象可以共享程序代码和数据结构，从而大大减少了程序中的冗余信息，提高软件的可重用性，便于软件修改维护

多态性：

在一般类中定义的属性或行为，被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一个属性或行为在一般类及其各个特殊类中具有不同的语义。

例子：数的加法—实数的加法---复数的加法

对于三元运算符这里再强调一下XXX?XXX:XXX如果一开始的表达式为真，那么取引号前面的一项。

第五章节：数据的共享与保护

作用域：在一个地方这个声明的量起作用

类作用域：一般用于特定的成员名

可见性：从对标识符的引用的角度来谈的。具体指的是从内层作用域往外层作用域看的时候能够看到什么。如果标识符在某处可见，则就可以在该处引用此标识符

一般来说：文件的作用域大于类作用域大于块的作用域

在同一作用域内的对象名，函数名，枚举常量名会隐藏同名的类名或枚举类型名

重载的函数可以有相同的函数名

关于c++的总结：

一、new和malloc的区别

1、new和delete配对，释放数组需要用delete[]。new和delete实际上调用了malloc和free，另外调用了类的构造函数和析构函数。

2、malloc和free配对，malloc返回的是void指针，需要强转。

3、new申请的内存保存在堆中，malloc申请的内存保存在自由存储区。

二、C++运算符

1、取模操作符：%

2、逻辑否、与、或：!, &&, ||

3、三元操作符：

c = (a>b) ? a : b;

4、按位与、或、非

& AND 逻辑与 Logic AND

| OR 逻辑或Logic OR

~ NOT 对1取补（位反转）Complement to one (bit inversion)

5、按位移：

<< SHL 左移Shift Left

>> SHR 右移Shift Right

三、&: 取地址运算符、定义变量引用

&操作符用于取地址时的用法是：int\* x=&y;。

然而，另外一种用法是定义变量别名，这种用法不能和取地址简单等同。用于传递函数输入参数时很好理解，但定义变量时容易引起理解错误，特别是和指针的区别：

从内存的角度看，指针和引用是完全不同的。指针,内存要为它分配一个存储空间。引用,内存不分配空间的，引用只是一个别名。我认为就是在符号表里增加一个标志而已，对于语句int &y=x; (&x=&y)为true。

实际上“引用”可以做的任何事情“指针”也都能够做，为什么还要“引用”这东西？答案是“用适当的工具做恰如其分的工作”。当重载某个操作符时，你应该使用引用。最普通的例子是操作符[]。这个操作符典型的用法是返回一个目标对象，其能被赋值。如果操作符[]返回一个指针，那么后一个语句就得这样写：

\*v[5] = 10;

但是这样会使得v看上去象是一个向量指针。因此你会选择让操作符返回一个引用。

引用的一些规则如下：

（1）引用被创建的同时必须被初始化（指针则可以在任何时候被初始化），否则会报编译错误。

（2）一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

以下示例程序中，k被初始化为i的引用。语句k = j并不能将k 修改成为j 的引用，只是把k的值改变成为6。由于k是i的引用，所以i 的值也变成了6。

int i = 5;

int j = 6;

int &k = i;

k = j; // k 和i 的值都变成了6;

（3）不能有NULL 引用，引用必须与合法的存储单元关联（指针则可以是NULL）。

以下的写法将地址指向一个位置的内存，是错误的。结果将是不确定的（编译器能产生一些输出，导致任何事情都有可能发生）

char \*pc = 0; // 设置指针为空值

char& rc = \*pc; // 让引用指向空值

（4）“sizeof 引用”得到的是所指向的变量(对象)的大小，但是当引用作为成员时，其占用空间与指针相同（没找到标准的规定）。

（5）引用只能指向一个实际的变量，不能指向指针或引用

(int\*) \* p1; // p1是指针的指针

(int\*) & p2; // p2是指向整型指针的引用

引用不能指向指针或引用！

(int&) \* p3; // ERROR: 不能有指向引用的指针，因为引用只是一个别名

(int&) & p4; // ERROR: 不能有指向引用的引用，因为引用只是一个别名

（6）指针和引用在内部的实现其实是没多大的区别的。但使用时有些地方是要注意的。因为引用具有对象行为，这一点很重要。引用复制时会调用对象的复制函数，在涉及多态时，这地方很容易出错。

class A{...};

class B:public A{...};

void f(A&a1,A&a2)

{

a1=a2;//此处调用的只有基类A的复制函数，而B部分不会被进行复制，之将导致数据的不一致（即B部分的数据没有被复制）；

a1.fun();

}

四、关于const

一般的const变量：

下面两个声明都指向一个const int类型的指针，指针所指向的内存不能被修改，但指针可以指向另一个内存：

const int \*p;

int const \*q;

int类型的const指针应该这样声明。指针所指向的内存可以被修改，但指针不能指向另一个内存

int \* const r= &n;

声明一个指向const int类型的const指针：

const int \* const p=&n;

const在函数声明中的含义：

const int& SetPoint(const int& param) const

第一个const:

函数的返回值限定为const,即返回值不能被修改。const int a=SetPoint(...) a在此之后便不能被修改。

第二个const:

指函数的形参为const类型,函数体内不能被修改.

第三个const:

表明这个函数不会对这个类对象的数据成员（准确地说是非静态数据成员）作任何改变。

类的const和static成员变量的初始化：

对于static成员变量，如果同时是const的，可以在类定义中初始化，否则只能在类定义外部初始化。

非static的const成员变量只能在构造函数的初始化列表中初始化。（ClassName():m\_1(1){};）

五、一些数据类型和变量赋值语法

1、union 中的所有被声明的元素占据同一段内存空间，其大小取声明中最长的元素的大小。union 的用途之一是将一种较长的基本类型与由其它比较小的数据类型组成的结构(structure)或数组(array)联合使用。

2、long double和float变量的赋值方法：

3.14159L // long double

6.02e23f // float

3、容易引起理解错误的定义语句：int\* p,q;

第一眼看去，好像是p和q都是int\*类型的，但事实上，只有p是一个指针，而q是一个最简单的int型变量。同时定义两个指针的语法是：int \*p1, \*p2;

4、定义一个指向int[4]数组的指针变量

int (\*p)[4]=RollNum;

这里，p被声明为一个指向一个4元素（int类型）数组的指针。

5、未指定size情况下，char数组的大小由初始化字符串决定：

我们可以用下面两种方法的任何一种来初始化字符串mystring：

char mystring [ ] = { 'H', 'e', 'l', 'l', 'o', '/0' };

char mystring [ ] = "Hello";

在两种情况下字符串或数组mystring都被定义为6个字符长(元素类型为字符char)：组成Hello的5个字符加上最后的空字符('/0')。在第二种用双引号的情况下，空字符('/0')是被自动加上的。两种情况下sizeof应该都是6，strlen都是5。

六、常用的几个标准C++函数

1、cout和cin的用法：

cout << "xxx" << endl;

cin >> "yyy";

2、常用的字符串函数：

strcat //字符串拼接

strcpy

strncpy

strcmp //字符串比较，相同返回0

七、switch-case的写法

switch (expression)

{

case constant1:

block of instructions 1

break;

case constant2:

block of instructions 2

break;

.

.

.

default:

default block of instructions

}

八、函数的几个属性和用法

1、指定函数的默认参数值

int divide (int a, int b=2) {

2、什么是函数重载(Overloaded functions)

两个不同的函数可以用同样的名字，只要它们的参量(arguments)的原型(prototype)不同，也就是说你可以把同一个名字给多个函数，如果它们用不同数量的参数，或不同类型的参数。

3、内联函数

inline 指令可以被放在函数声明之前，要求该函数必须在被调用的地方以代码形式被编译。这相当于一个宏定义(macro)。它的好处只对短小的函数有效，这种情况下因为避免了调用函数的一些常规操作的时间(overhead)，如参数堆栈操作的时间，所以编译结果的运行代码会更快一些。

调用函数的时候并不需要写关键字inline ，只有在函数声明前需要写。

4、将数组作为参数传入函数，传的是引用而不是值。

void procedure (int myarray[ ][3][4])

九、函数指针的用法

使用函数指针的几种方法：

（1）简单调用函数指针；

void (\*pfunc)(int);

pfunc=callback\_funcname;

callback\_funcname(1);

其中声明函数指针原型的代码可以在调用处写，也可以写成全局的。这种方法使用简单，适用于临时调用。

（2）使用typedef调用函数指针：

typedef void(\*PFUNC)(int);

PFUNC pfunc;

pfunc=callback\_funcname;

callback\_funcname(1);

这种方法适用于多次调用，先全局定义PFUNC，再在每个调用的地方声明临时变量后调用。

（3）C++类中调用成员函数指针（不使用typedef）：

void (\*MyClass::pfunc)(int);

pfunc=&MyClass::callback\_funcname;

(this->\*callback\_funcname)(1);

和方法1类似，注意语法的不同。

（4）C++类中调用成员函数指针（使用用typedef）：

typedef void(\*PFUNC)(int); //在类中typedef

PUNC pfunc;

pfunc=&MyClass::callback\_funcname;

(this->\*callback\_funcname)(1);

和方法2类似，注意语法的不同。

十、typedef的不常用用法

typedef的一般用法是：

typedef int UINT32;

但用来定义一个数组类型或指针函数时，比较特殊：

typedef char CARRAY[32]; //定义了一个CARRAY的类型，代表char[32]

typedef void(\*PFUN)(int); //定义了一个指向指针函数的变量类型，函数原型为void xxx(int yyy);

十一、类的private/protected/public属性

1、类的成员如果没有指定访问域，默认是private的。

2、标识符protected 与 private类似，它们的唯一区别在继承时才表现出来。当定义一个子类的时候，基类的protected 成员可以被子类的其它成员所使用，然而private 成员就不可以。

3、public/protected/private继承的区别：

（1）public继承：父类的public依然是public，protected依然是protected，private不可访问；

（2）protected继承：父类的public称为protected，protected称为private；

（3）private继承：父类的所有成员全部变成private。

十二、关于空类

编译器为一个空类提供哪些默认函数？

1、C++编译器会提供默认的构造函数，析构函数， 拷贝构造函数和拷贝赋值操作符（请参考著名的Effective C++的第三版的第5条）

当我们定义一个class而没有明确定义构造函数的时候，编译器会自动假设两个重载的构造函数 (默认构造函数"default constructor" 和复制构造函数"copy constructor")。拷贝构造函数是一个只有一个参数的构造函数（原型：ClassName(ClassName &cn){};），该参数是这个class的一个对象，这个函数的功能是将被传入的对象（object）的所有非静态（non-static）成员变量的值都复制给自身这个object。

必须注意：这两个默认构造函数（empty construction 和 copy constructor ）只有在没有其它构造函数被明确定义的情况下才存在。

一个类包含一个对赋值操作符assignation operator (=)的默认定义，该操作符用于两个同类对象之间。这个操作符将其参数对象(符号右边的对象) 的所有非静态 (non-static) 数据成员复制给其左边的对象。

2、用class obj;的方式声明一个对象，如果构造函数没有参数，或只有默认构造函数，后面不能加()，因为编译器会误以为这是一个没有参数的函数声明;

3、如果任何其它有任意参数的构造函数被定义了，默认构造函数和拷贝构造函数就都不存在了。在这种情况下，如果你想要有empty construction和copy constructor ，就必需要自己定义它们。

4、对基本类型，在c++里面，为了模板template,规定他们可以使用类似于类的默认构造函数的方式（仅仅是类似的方式而已） 赋初始值0。这叫做基本类型的显示初始化， 请参考 C++标准程序库（The C++ Standard Library）的14页，2.2.2 基本型别的显示初始化，书中举的例子就是

int i1;//未初始化

int i2 = int(); //初始化为0

sizeof一个空类等于多少？

sizeof一个空类返回1。所谓类的实例化就是在内存中分配一块地址，每个实例在内存中都有独一无二的地址。同样空类也会被实例化，所以编译器会给空类隐含的添加一个字节，这样空类实例化之后就有了独一无二的地址了。所以空类的sizeof为1。C++编译器不允许对象为零长度。试想一个长度为0的对象在内存中怎么存放？怎么获取它的地址？为了避免这种情况，C++强制给这种类插入一个缺省成员，长度为1。如果有自定义的变量，变量将取代这个缺省成员。

十三、继承或多重继承情况下构造函数的调用顺序

（1）如果声明为Derive: public Super1, public Super2{AnotherClass m\_obj;}; 构造函数的调用顺序是：Super1, Super2, AnotherClass, Derive.

（2）如果父类有默认构造函数，或没有参数的构造函数，不需要在子类的构造函数定义中显式调用父类构造函数，否则需要调用。成员对象也是一样道理。以上面的例子说明，Derive的构造函数写法是：

Derive(int i): Super1(i), Super2(i), m\_obj(1){...}

注意，成员对象初始化时应该指明对象名称，而不是类名。

析构函数的调用顺序应该是依次反过来的。

十四、虚函数、纯虚函数和抽象类、虚析构函数

虚函数的作用和运行原理

（1）多态是面向对象编程中的核心概念，就是说一个基类类型的指针实际上可能指向的是一个子类对象。只有在运行时才能根据实际情况来决定执行哪个函数，也就是动态联编。和动态联编对应的是静态联编，也就是说在编译时就决定了调用哪个函数。为了实现动态联编，必须将父类的函数声明为virtual。如果没有声明为virtual，可能得到的结果不是预期中的。

对于析构函数而言，虚函数保证子类和父类的析构函数都会被执行。

参考：

http://jiamingjun03.blog.163.com/blog/static/11687677620099297435263/

（2）对于包含了至少一个虚函数的类（或其父类包含虚函数），编译器需要为这个类增加4个字节，用来保存指向虚函数表VTABLE的指针。

纯虚函数和抽象类

包含了纯虚函数的类不能被直接实例化，可以称为抽象类。定义方法：

virtual void func()=0;

子类override一个虚函数，不一定要加virtual关键字。

什么情况下需要指定析构函数为virtual?

（1）析构函数不一定需要定义为虚函数，只有当这个类要作为其他类的父类使用时，才需要定义为虚函数。如果父类的析构函数没有定义为虚函数，则子类对象销毁时，父类析构函数不会被调用。

（2）对于一个抽象类，析构函数可以被定义为纯虚的。

（3）父类和子类之间的虚函数动态联编不会因为private发生影响。

（4）一个类的虚函数在它自己的构造函数和析构函数中被调用的时候，它们就变成普通函数了，不“虚”了。也就是说不能在构造函数和析构函数中让自己“多态”。

（5）在虚函数和纯虚函数的定义中不能有static标识符，原因很简单，被static修饰的函数在编译时候要求前期bind,然而虚函数却是动态绑定（run-time bind），而且被两者修饰的函数生命周期（life recycle）也不一样。

十五、多重继承情况下如何引用父类的同名成员？

重继承情况下，如果多个基类有同名成员，引用方法是：

pDeriveObj->BaseClass1::Member;

十六、运算符重载

语法：

Type Type::operator +(const Type &i){}

十七、友元

友元可以实现外部对private和protected成员的访问。有两种实现：

（1）友元函数。语法：在函数声明前加上friend。友元函数并不是类的成员函数，实现函数体或调用函数时不加ClassName::。

（2）友元类。在类的声明中加入：friend class VisitorClass;

友元不会被子类继承。

十八、模板

函数模板

实现语法:

在函数的声明和实现前加上template<typename T> ，(typename和class等价)，可以写在一行里面，也可以分成两行写，注意>后面没有分号。如果声明和实现分开写，两个地方都要写上template<typename T>。（1）一行代码中同时声明并实现函数

template <typename T>void func(T t1, T t2)

{...};

（2）分两行代码声明并实现函数

template <class T>

void func(T t1, T t2)

{...};

（3）在两个地方分别声明和实现函数

template <class T>

void func(T t1, T t2);

...

template <class T>

void func(T t1, T t2)

{...}

引用语法：

func<int>(5, 6);

类模板

声明方法：

类模板可以实现在一个类中有一个通用类型的成员变量。

template <class T> class ClassName

{public:

T \*m\_pVariable;

};

引用方法：

ClassName<int> obj;

模板特殊化

模板特殊化可以专门为某种数据类型定义特殊的行为。类的定义必须和通用的模板类完全一致，除了用专门语法，并将T修改为专门的类型，并定义特殊行为。

template<> class<int>{定义通用函数，定义特殊函数};

定义模板的默认值

template <class T = char> // 有一个默认值。

模板的参数值

除了模板参数前面跟关键字class 或 typename 表示一个通用类型外，函数模板和类模板还可以包含其它不是代表一个类型的参数，例如代表一个常数，这些通常是基本数据类型的。

二十、类型转换和C++高级类型转换

基本类型强转有两种写法：

int i;

float f = 3.14;

i = (int) f;

i = int ( f );

高级类型转换

ANSI-C++ 标准定义了4种新的类型转换操作符： reinterpret\_cast, static\_cast, dynamic\_cast 和const\_cast。

reinterpret\_cast可以将一个指针转换为任意其它类型的指针。

ClassA\* pa;

ClassB\* pb=reinterpret\_cast<ClassB\*>pa;

static\_cast可以执行所有能够隐含执行的类型转换，以及它们的反向操作（即使这种方向操作是不允许隐含执行的）。用于类的指针，也就是说，它允许将一个引申类的指针转换为其基类类型（这是可以被隐含执行的有效转换），同时也允许进行相反的转换：将一个基类转换为一个引申类类型。不会检查被转换的基类是否真正完全是目标类型的。

Derive\* pa;

Super\* pb;

pa = static\_cast<Derive\*> pb;

pb = static\_cast<Super\*> pa;

static\_cast除了能够对类指针进行操作，还可以被用来进行类中明确定义的转换，以及对基本类型的标准转换：

double d=3.14159265;

int i = static\_cast<int>(d);

dynamic\_cast 完全被用来进行指针的操作。它可以用来进行任何可以隐含进行的转换操作以及它们被用于多态类情况下的方向操作。然而与static\_cast不同的是， dynamic\_cast 会检查后一种情况的操作是否合法，也就是说它会检查类型转换操作是否会返回一个被要求类型的有效的完整的对象。

在不合法的情况下，如果用于指针，将返回NULL；如果用于引用，抛出异常。

Derive\* pa = new Derive();

Super\* pb = new Super();

pa = dynamic\_cast<Derive\*> pb; //失败，返回NULL

pb = dynamic\_cast<Super\*> pa; //成功

const\_cast类型转换对常量const 进行设置或取消操作。

class C {};

const C \* a = new C;

C \* b = const\_cast<C\*> (a);

typeid (object\_pointer)

这个操作符返回一个类型为type\_info的常量对象指针，这种类型定义在标准头函数中。type\_info::name()返回对象的类名。

二十一、命名空间

定义一个命名空间：

namespace ns1{...}

设置默认命名空间：

using namespace ns1;

引用其他命名空间的类型：

ns2::variable = xx;

二十二、预处理命令

#undef 完成与 #define相反的工作，它取消对传入的参数的宏定义

#ifdef, #ifndef, #if, #endif, #else and #elif

指令#line 可以使我们对这两点进行控制，也就是说当出错时显示文件中的行数以及我们希望显示的文件名。它的格式是：

#line number "filename"

下面这段代码将会产生一个错误，显示为在文件"assigning variable", line 1 。

#line 1 "assigning variable"

int a?;

这个指令将中断编译过程并返回一个参数中定义的出错信息

#error

这个指令是用来对编译器进行配置的，针对你所使用的平台和编译器而有所不同。

#pragma

二十三、预定义宏

\_\_LINE\_\_ 整数值，表示当前正在编译的行在源文件中的行数。

\_\_FILE\_\_ 字符串，表示被编译的源文件的文件名。

\_\_DATE\_\_ 一个格式为 "Mmm dd yyyy" 的字符串，存储编译开始的日期。

\_\_TIME\_\_ 一个格式为 "hh:mm:ss" 的字符串，存储编译开始的时间。

\_\_cplusplus 整数值，所有C++编译器都定义了这个常量为某个值。如果这个编译器是完全遵守C++标准的，它的值应该等于或大于199711L，具体值取决于它遵守的是哪个版本的标准。

**第一章**

**作用域限定符**

限定符：：成员名

限定符一般就是类名，也可能是命名空间，用于类成员的访问及定义，用于说明某一成员属于哪一个类，属于哪一个命名空间。若限定符省略，则说明后面的成员是一个全局变量

**const常量**

1定义时必须初始化

2为右结合，但若是简单数据类型，const也可以写在数据类型的左侧，如

const int i=5;

int const i=5;

const还可修饰函数的形参，表示相应的实际参数在该函数内不能被修改

**常引用**

引用声明时必须初始化，即指定它引用的是哪一个变量

系统不会为变量的引用另外分配存储空间，引用和被引用的变量共享同一内存单元

常引用语法

const 数据类型 &引用名

如果在声明引用时用const修饰，那么该引用就是一个常引用。意味着所引用的对象不能更改

首先进一步理解引用： int &a=b 相当于 int \*const a=b。即引用是一个指针常量（又称常指针，即一个常量，其类型是指针,表示这个指针只能指向这个地址，不能改变指向其他地址）。

每当编译器遇到引用变量a，就会自动执行 \* 操作。

而常引用：const int &a=b就相当于 const int \* const a=b。不仅仅是a这个地址不可修改，而且其指向的内存空间也不可修改。

常引用的作用有两个，一是让变量所指向的内存空间只读，二是能够指向常量

int \* const p1=&a;//这里必须赋值，因为是常指针

p1不能指向别处

//  const int \*p2;

int const \*p2;//这两种定义方式都可以

不能通过p2修改其所指向地方的值

const int \* const p3=&a;//这里必须直接赋值

包括以上两种情况，p3既不能指向别处，也不能通过p3修改被指向地方的值

int &c = a;//引用c和a等价，可以互相改变（相当于常指针，即只能指向a的指针）

int & const d = a;//个人感觉这里const没啥用啊！等价于int &d=a

const int & e = a;//等价于int const &e=a。不能通过e修改a的值

const int & const f=a;//f不能赋值，但是a却可以赋值。效果和上一个相等

const int \* &p12 = p10;//p12是一个指针引用，是p10的引用，所以p12也是一个指针，和p10指向同一个地方，const在最前面，所以不能通过p12改变所指地方的值。但是p12还能指向其他地方。

int \*const &p16 = p14;//引用，p16与p14等价，这里p16只能和p14一样指向a，不能指向其他地方了。但是可以改变a的值。

**常对象**

如果希望一个对象自创建时起就不能被修改状态，可以用const修饰这个对象，将它声明为常对象。

类名 const 对象名（实参表）

const 类名 对象名（实参表）

常对象在生命的同时必须初始化，并从此不能改变其状态。

This指针不能指向常对象，所以常对象不能调用成员函数。通常是将常对象的引用作为参数传递给函数。

常对象只能调用类的常成员函数和静态成员函数。

**常成员函数**

返回类型 函数名（参数表） const

常对象不能调用普通成员函数，能调用常成员函数

常成员函数只能访问对象的数据成员，而不能修改他们的值，也不能在函数体中调用非const修饰的成员函数。

当两个成员函数原型相同，只区别在有无const修饰的时候，系统可以视之为重载函数，并根据调用的对象来匹配对应的函数：常对象调用常成员函数，普通对象调用普通成员函数。

**常数据成员**

构造函数对常数据成员初始化必须使用初始化列表。

对象创建之后，任何函数不能修改常数据成员的值。

如果类有多个重载构造函数，每个构造函数都需要初始化常数据成员。

内联函数inline

**带默认参数的函数**

Void fun(int a, int b,int c=199)

默认参数的形参位置必须放在最右边

**new和delete**

new的返回值：若是非数组类型，返回指向该类型对象的指针；若是数组类型，返回数组的首地址

new不一定总会成功，不成功返回NULL

new为数组申请内存时无法提供初值

delete操作符格式：

delete 指针变量

delete []指针变量

**预处理命令**

以#号引导

预处理命令包括宏定义命令，文件包含命令，条件编译命令

# **第二章**

## 面向对象思想

封装、抽象、继承、多态

## 类的定义

Class  <类名>

{

Public:

<成员函数或数据成员的声明>

Private:

<成员函数或数据成员的声明>

};

<各个成员函数的类外实现>

## 类的访问权限

Public：类对外界公开的部分，允许类外访问

Private：只能被类内部的成员函数访问

Protected：可以被类内成员函数访问，也可以被派生类的成员函数访问。

## 构造函数与析构函数

构造函数的作用是在对象被创建的时候为其分配内存空间，初始化。

构造函数的函数名与类名相同，可以有任意类型的参数，但不具有返回类型。创建对象时，系统会自动调用构造函数，如果没有人为编写的构造函数，系统会自动生成一个默认的构造函数。

## 拷贝构造函数

类名(const 类名&对象名);

拷贝构造函数的函数名是类名，同时没有返回类型，由此可见它是一种构造函数。

拷贝构造函数只能有一个参数，而且是本类类型。Const关键字约束参数不可改变，这样可以确保复制源的同类对象不会被改变。采用传引用的方式，是因为引用不占内存空间，减小系统开销。

### 深拷贝和浅拷贝

如果类中不包含指针类型的数据成员，那么拷贝构造函数执行的是浅拷贝，即两个对象进行简单的数据成员的值复制。像这样的拷贝构造函数没有必要显示定义，因为系统会自动生成。

如果类中包含指针数据成员，就需要进行深拷贝，即定义一个深拷贝构造函数，在复制之前先动态申请一块内存，然后进行复制，而不是指针值的复制（因为这实际是地址的复制）。

## 析构函数

~类名();

做一些对象消亡的扫尾工作，如释放内存。

析构顺序：先创建的对象后消亡，后创建的对象先消亡。

## This指针

对于同类对象而言，他们各自拥有一套数据成员，并分别占有相同大小的存储空间。

类的成员函数则不同，对于类的所有对象而言都只有一份。

当不同的对象调用相同的成员函数时，系统该怎样判断函数所操纵的数据成员是属于哪一个对象的呢？每个类都拥有一个自动生成、自动隐藏的私有成员，它存在于类的非静态成员函数中，用来指向正在调用某个成员函数的对象的地址，这就是this指针。

This的作用域在类内部，当在类的非静态成员函数中访问类的非静态数据成员时，编译器会自动将对象本身的地址作为一个隐含参数传递给非静态成员函数的this指针，所以函数成员的原型应该是void Move(Point\* this,float x1,float y1);

this 一般不用显示写出，但是在函数返回调用函数的对象本身的时候，可以使用return \*this;

当参数与数据成员同名时，使用this可以增强可读性。

This->n=n;

This指针只能在类的非静态成员函数中使用，指向调用该成员函数的对象。

## 静态数据成员

Static

不专属于某个对象，生命周期与整个程序相同，即使某个对象消亡了，它还会在程序中继续存在，被类的其他对象共享。

保存在静态数据区，该类的所有对象共享这块存储空间，所有该类对象都可以修改该数据。

静态数据成员也有访问权限控制

初始化不能在构造函数中，要在类外初始化静态数据成员。

由于只属于类所有，静态数据成员的访问形式一般通过类名和作用域限定符的形式来访问。

## 静态成员函数

静态成员函数主要用来操纵同类中的静态数据成员。不能访问普通数据成员。因为普通成员函数都有this指针，可以指明普通数据成员所属的对象，而静态成员函数没有this指针，因此当遇到普通数据成员时，无法指明该数据成员属于哪个对象，所以出错。

静态成员函数的访问权限，公有的可以通过类名：：调用，也可以通过对象名调用。私有的只能通过对象名来调用。

## 友元关系

通常，只有类中的成员函数才能访问类的私有成员，类外无法访问。

友元函数：由friend修饰，虽然在类中声明，但不是类的成员函数，可以是普通函数，也可以是其他类的成员函数。如果要操纵类的数据成员，需要将类的对象作为参数传递进去，因为友元函数不是成员函数，所以没有this指针，不能直接访问数据成员，只能同过类来访问数据成员，所以需要传递整个类作为参数。但是函数体内类对象可以直接访问类的数据成员。

友元函数可以直接访问类的所有成员，但它不是该类的成员函数，可以放在类声明的任何位置，既可以在private区也可以在public区，但不受访问权限的控制。

友元函数若不是成员函数，则不需要通过对象或对象指针来调用，直接调用即可。

如果友元函数是其他类的成员函数，在声明时之选哦在该成员函数的函数名前加上“类名：：”即可。

Class A{

Friend void B::function();

}

## 友元类

如果类B是类A的友元类，那么B的所有成员函数都是A的友元函数，都可以直接访问A的所有数据成员。

# **第三章**

## 继承语法格式

Class 派生类名：继承方式 基类名1，继承方式 基类名2……

{

派生类新增成员定义；

}

设计一个派生类主要包括三部分：吸收基类成员，改造基类成员，新增自己成员。

## 继承方式public，protected，private

公有继承：基类的公有和保护成员在派生类中仍是公有和保护成员，可以由派生类的成员函数来访问。基类的私有成员，无论派生类的成员函数还是类对象都不能访问。

私有继承：基类的所有非私有成员全部变为私有成员，派生类只能通过它的成员函数来访问，派生类对象不能访问。至于基类的私有成员，只能通过基类的成员函数来访问。

省略继承方式默认为私有继承

保护继承：基类的非私有成员都变为保护成员，可以被派生类的成员函数访问，但不能被派生类的对象访问。基类的私有成员仍然在派生类中不可见。

派生类不能继承基类的构造函数和析构函数，必须自己定义

## 构造函数初始化格式

派生类名（参数表）：基类名1（参数表1），基类名2（参数表2）……

{

派生类新增成员初始化

}

对基类成员和子对象成员的初始化必须在初始化列表中进行，新增成员可在初始化列表中进行也可以在函数体内进行。

派生类构造函数初始化顺序是：先调用基类构造函数初始化基类数据成员，如果有子对象，随后进行子对象的初始化，最后初始化派生类新增成员。

派生类有多个基类时，处于同一层次的各个基类的构造顺序取决于积累的声明顺序（从左向右）

如果派生类的基类也是一个派生类，则每个派生类只需负责其直接基类的初始化，依次上溯。

## 析构顺序与构造顺序相反。

## 多继承的构造函数

多继承中，派生类需要初始化多个基类，应按照派生类声明各个基类的顺序来逐一提供参数并调用相应基类的构造函数来完成。

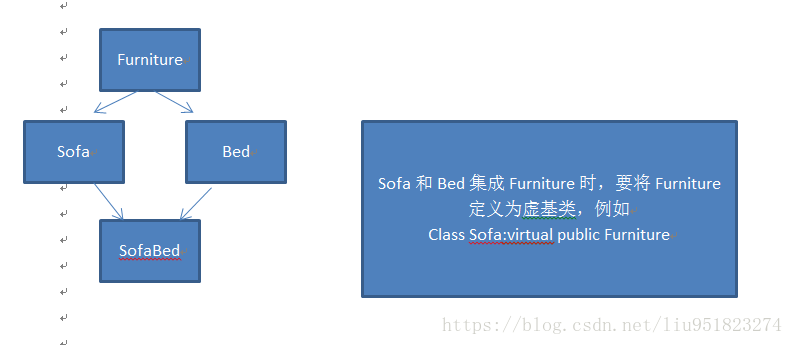
## 虚基类

消除多继承中的二义性问题。

如果在多条继承路径上有一个公共基类，那么由它派生出的多条路径的汇合处的派生类会存在多套继承自该公共基类的同名数据成员。

虚基类用virtual标识

Class 派生类名：virtual 继承方式 基类名



## 虚基类的初始化

与一般基类相同，只是调用顺序不同

1. 虚基类的构造函数在非虚基类之前调用
2. 若同一层有多个虚基类，则按照声明顺序调用
3. 若虚基类由非虚基类派生而来，则先调用更高级别的基类的构造函数，在按照（1）（2）的顺序

**第四章**

**多态**

静态多态性：编译阶段绑定（早期联编），通过函数重载实现

动态多态性：运行时动态绑定（之后联编），通过虚函数和继承实现

**动态绑定的条件：**

1. 在虚基类中定义虚函数
2. 使用指向基类的指针或引用调用虚函数

虚函数必须是成员函数，且必须是非静态成员函数。（静态成员函数都是早期联编的）。

当基类中某个成员函数定义为虚函数时，在派生类中就要重新定义该函数，且函数签名必须完全一致。

虚函数不能是内联函数

构造函数不能是虚函数

析构函数可以是虚函数

**虚函数的覆盖**

出原型完全相同的虚函数在基类、派生类中形成覆盖关系外，其他同名函数形成隐藏关系。隐藏和覆盖都让派生类对象无法访问基类同名函数。但当通过基类指针或引用访问派生类中的隐藏或覆盖函数时会有不同结果。覆盖函数会调用派生类中的新定义的函数，隐藏函数会调用基类中的函数而不能调用派生类中的新定义的函数。

**虚函数的传递性**

虚函数具有传递性，即在多重继承中基类派生的所有派生类的原型相同的函数都是虚函数，具有虚函数的一切特性。通过基类的指针或引用可以调用该基类派生的所有派生类的所有原型相同的函数，且都具有多态性。

在有继承关系时，析构函数应设计为虚函数。

构造函数尽量采用初始化列表的方式初始化派生类的构造函数，以提高效率。因为初始化列表会在函数体内的赋值语句执行前执行，且效率比函数体内的赋值语句高。

**抽象类**

Class 类名

{

Public:

    Virtual 函数返回类型 函数名（参数表）=0；

}

虚函数必须有函数实现，纯虚函数一般在基类中没有实现。

纯虚函数首先必须是成员函数。

纯虚函数是虚函数的一种特例，具有虚函数的性质——通过它可以实现动态多态性。

如果在派生类中没有对基类的纯虚函数进行定义，那么他在派生类中依然是纯虚函数，派生类则依然是抽象类。

一个类如果含有纯虚函数，那么该类只能作为基类用于派生新类，不可以实例化。（类似与java中的接口）

抽象类主要表达一个类族的共同特征或者行为，以方便该类族的派生类继承和实现，从而更好的实现多态性。

不能直接用抽象类对象作为函数参数类型、函数返回值类型或显示转换类型。

函数参数设置为抽象基类的指针或引用时，使用抽象基类的指针或引用（指向子类）传进去可动态绑定。

**函数重载**

函数名相同，但是参数表不同。

运算符重载

<函数返回值类型> operator <运算符> (<形参表>)

{

<函数体>

}

成员运算符重载，参数个数必须必运算符原来的参数个数少1，表示默认隐含的另一个参数是该类的对象，该类的对象被隐含作为第一操作数。

非成员运算符重载，参数个数与运算符原来的个数相同。<<和>>运算符只能重载为非成员运算符。

单目运算符最好重载为成员运算符，双目运算符最好重载为非成员运算符。

重载前缀++或—

<函数类型> operator ++();

重载后缀++

<函数类型> operator ++(int);             int仅作为与前缀的区别

赋值运算符重载类似于拷贝构造函数

通常凡事包含动态分配成员或包含指针成员的类，都需要提供拷贝构造函数；提供拷贝构造函数的同时，还应该考虑重载赋值运算符。

**调用拷贝构造函数的情况：**

1当以对象作为函数参数，以值传递的方式传入函数体时；2当以对象作为函数返回值，以值传递的方式从函数返回对象时；3当用对象初始化另一个对象时。

1、C++ 可执行程序的内存分布

BSS：Block Started by Symbol

存放程序中未初始化的全局变量

数据段：

存放已初始化的全局变量，静态内存分配

代码段：

存放执行代码

堆heap：

存放运行中动态分配的内存段，可扩张或缩减

栈stack：

存放程序的局部变量，在函数被调用时，栈还用来传递参数和返回值

2、C++编译原理

1、预编译：宏展开过程

2、编译：inline在此时展开，从C++代码到汇编语言

3、汇编：汇编语言到机器语言

4、链接

引申：内联函数和宏：

两者都可以节省函数栈的开销，提高运行效率。因为函数在运行时需要保持现场和上下文后恢复现场，有时间和空间的开销

宏在调用时直接展开，更有效率，但不能使用私有成员，不能调试，容易产生二义性

内联函数由编译器展开

3、C++ extern关键字的作用

加在变量或函数前，表明该内容的定义在其他文件中。

extern “C” 提示C++编译器按C语言的规则编译函数，举例：

C++生成的动态库.so文件，如果按C++命名，则函数的名称会因为继承等原因变化，比如func()会变成func\_%20()，其他程序调用dlopen动态打开该库时，按照func名称会搜索不到相应的函数。

4、static关键字的作用

在全局变量中加static，称为静态全局变量有以下特点：

在数据段中存放

会被自动初始化成0

在文件之外不可见

另外，static修饰函数，可以使其成为类的静态成员函数，成员变量和类相关，和类的对象无关。

及时没有具体对象，也能调用类的静态成员函数——单例模式

5、const关键字的作用

主要有四个方面：常量、函数输入、函数输出、成员函数，都是为了让被修饰的内容受到保护，防止意外修改

1、const修饰常量，最常用，常量不让修改

2、const修饰输入参数：

void StringCopy(char dest,const char source)，函数体内任何修改source的行为都会报错

3、如果输入参数采用值传递，由于函数会在栈中自动产生临时变量的复制（拷贝构造函数），因此参数不用加const

void func(int x)不能用void func(const int x)

4、引用传递比值传递效率高，但是缺点是参数可能被修改，因此也可以加const修饰

void func(A& a) –> void func(const A& a)

5、用const修饰函数的返回值

如果以指针传递的方式返回，且被const修饰，则赋值给其他指针变量的时候，要求指针也得用const声明：

const char \* GetString();

char \* str=GetString();——报错

const char \*str=GetString();——正确

如果是值传递，由于函数会自动复制的关系，加const没用

6、const修饰成员函数

任何不会修改数据成员的函数都应声明为const类型

1、在const成员函数中，不慎修改了数据成员，会报错

2、在const成员函数中，不慎调用了其他非const成员函数，报错

7、static、const、static const的初始化

1、在类里创建一个const类型变量时，不能给他初值

理由：编译器必须为const类型变量分配存储空间，但类的声明又不占用存储空间

class foo

{

private:

const int a= 100;——报错

}

只能在构造函数的初始化列表里初始化：

A::A(int a):bb(a)，其中bb的声明为 const int bb;

2、static修饰的类成员变量，既不能在类定义中初始化，也不能在初始化列表里初始化，只能在外头

int foo::a=100;

3、static const也是在类外初始化

8、C++中，new和malloc的区别

new是c++的操作符，malloc是c中的函数

new可以调用构造函数 delete调用析构函数

malloc只是单纯的分配内存，不会初始化成员

9、关于C++ 多态和函数表的关系

多态的定义：一个接口，多种方法

多态和非多态的区别，可以理解为静态编译（重载）和动态编译（虚函数）

多态是通过虚函数表来实现的，重载在编译时就确定了函数的地址，虚函数的运行地址只能在运行过程中才能确定

10、C++虚函数与构造、析构函数的关系

1、构造函数不能使虚的

理由：存在虚函数的话，会存在虚函数表vtable，构建的对象会有一个vptr指向虚函数表，所以不能把构造函数放虚函数里

2、析构函数一般都是虚的

如果是实的，那么当有一个基类指针指向派生类，那么析构时只会析构基类的成员，造成派生类的成员没有正确释放

只有是虚的，才会先在虚函数表中找到派生类，再找基类

如果类中有虚函数，不管有多少个，它派生出来的对象大小都会多4字节，是vptr：指向虚函数表的指针

11、C++ 构造函数、拷贝构造函数、赋值函数

1、构造函数、拷贝构造函数

特点都是“无中生有”

A a——通过默认构造函数创建一个对象

2、拷贝构造函数

用一个类的对象去初始化另一个对象时：

A b=a;

A b(a);

都使用的拷贝构造函数

浅拷贝：复制一个指针，指向同一个地址空间

深拷贝：在内存中开辟新的空间，两个对象独立

拷贝构造函数的声明：

A(const A& a)

赋值运算函数：

A& operator=(const A& a)

Linux技术

1、epoll和select

当一个程序要读两个以上IO时，如果使用阻塞式IO，那么一个文件描述符阻塞，另一个文件描述符有数据却读不出来，效率很低。

解决方案有以下几个：

1、开一个描述符就起一个线程，维护起来比较复杂

2、用非阻塞IO，得一直轮询

3、用异步IO，通过信号量或者回调触发

4、IO多路复用（epoll和select）

函数

函数模板：

只适用于函数体相同、函数的参数个数相同而类型不同的情况，如果参数的个数不同，则不能用函数模板

函数重载：

参数的个数和类型可以都不相同。但不能只有函数的类型不同而参数的个数和类型相同。

默认参数：

实参与形参的结合是从左至右顺序进行的。因此指定默认值的参数必须放在形参表列中的最右端，否则出错。

函数的声明和定义：

函数的声明是函数的原型，而函数的定义是函数功能的确立；

变量的声明和定义：

把建立存储空间的声明称为定义，把不需要建立存储空间的声明称为声明；

指针和数组

多维数组名作函数参数：

必须指定第二维（即列）的大小，且应与实参的第二维的大小相同

变量与指针：

变量的指针就是变量的地址。用来存放变量地址的变量就是指针变量。

指针变量：

一般的C++编译系统都为指针变量分配4个字节的存储单元。但在定义指针变量时必须指定基类型。

函数调用：

实参变量和形参变量之间的数据传递是单向的“值传递”方式，调用函数时不会改变实参指针变量的值，但可以改变实参指针变量所指向变量的值。

\*(p++)、\*(++p):

前者先取\*p的值，然后使p加1。后者是先使p加1，再取\*p

函数指针：

一个函数在编译时被分配一个入口地址，这个函数入口地址就成为函数的指针。

指针函数：

返回指针值的函数，eg. 类型名 \*函数名(参数列表)

指针数组 类型名 \*数组名[数组长度]

数组指针 类型名 (\*指针名)[数组长度]

1

2

引用：

引用不是一种独立的数据类型，必须指定其代表某一类型的实体(如变量、类对象)，不能定义引用数组，不能定义指向引用的指针。引用变量主要用作函数参数，它可以提高效率，而且保持程序良好的可读性。

结构体和对象

动态分配撤销内存：

new和 delete是运算符，不是函数

类类型访问限制：

如果在类的定义中既不指定private，也不指定public，则系统就默认为是私有的。

结构体vs类：

用struct声明的类，如果对其成员不作private或public的声明，系统将其默认为public，而用class声明的类，如果不作private或public的声明，系统将其成员默认为private。

成员函数：

不论成员函数在类内定义还是在类外定义，成员函数的代码段都用同一种方式存储，即都不占用对象的存储空间。

信息隐蔽：

类的公用接口与私有实现的分离。当接口与实现（对数据的操作）分离时，只要类的接口没有改变，对私有实现的修改就不会影响程序的其他部分。

默认构造函数：

调用构造函数时不必给出实参的构造函数。一个类只能有一个默认构造函数。由于不需要实参也可以调用构造函数，因此全部参数都制定了默认值的构造函数也属于默认构造函数。

析构函数：

作用不是删除对象，是在撤销对象占用的内存之前完成一些清理工作，由于没有函数参数，所以不可以被重载。

构造函数、析构函数执行顺序：

先构造的后析构，后构造的先析构。相当于一个栈，先进后出。

静态数据成员：

在内存中只占一份空间，每个对象都可以引用这个静态数据成员。静态数据成员在程序编译时被分配空间，到程序结束时才释放空间。

静态成员函数/非静态成员函数：

非静态成员函数有this指针，而静态成员函数没有this指针。静态成员函数主要用来访问静态数据成员。

提前引用声明：

在正式声明一个类之前，先声明一个类名，表示此类稍后声明。目的为了解决“鸡生蛋，蛋生鸡”的问题。

保护成员/私有成员：

两者的不同之处，在于把保护成员的访问范围扩展到派生类中。

运算符

自增/减运算符重载函数：

如果在自增（自减）运算符重载函数中，增加一个int型形参，就是后置自增（自减）运算符函数。

转换构造函数/类型转换函数：

在需要的时候，编译系统会自动调用这些函数，建立一个无名的临时对象。将运算符“+”函数重载为友元函数，在进行对象相加时，可以用交换律。将运算符“+”函数重载为类的成员函数，交换律不适用。所以，双目运算符函数重载为友元函数，单目运算符则多重载为成员函数。

运算符重载：

一般将单目运算符重载为成员函数，将双目运算符重载为友元函数（原因如上一条）

继承和多态

派生类和基类：

派生类是基类的具体化，基类是派生类的抽象；一个派生类可以从一个基类派生，也可以从多个基类派生。构造派生类通常包含三部分：从基类接受成员，调整从基类接受的成员，在声明派生类时增加的成员。构造函数和析构函数不能从基类继承。

重名函数覆盖：

不同的成员函数，只有在函数名和参数个数相同、类型相匹配的情况下才发生同名覆盖，如果只有函数名相同而参数不同，不会发生同名覆盖，而属于函数重载。

虚基类继承：

由于虚基类在派生类中只有一份数据成员，所以这份数据成员的初始化必须由派生类直接给出。

基类和派生类之间的转换：

只能用子类对象对其基类对象赋值，而不能用基类对象对其子类对象赋值。通过指向基类对象的指针，只能访问派生类中的基类成员，不能访问派生类增加的成员。

多态(polymorphism)：

具有不同功能的函数可以用同一个函数名

虚函数：

允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。在基类用virtual声明成员函数时为虚函数，在类外定义虚函数时，无需再加virtual。当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。

纯虚函数：

在声明虚函数时被初始化为0的函数，作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。如果在基类中没有保留函数名字，就无法实现多态性。

cerr/clog：

都是在终端显示器上显示出错信息，但是cerr是不经过缓冲区，直接向显示器输出相关信息，clog中的信息是存放在缓冲区中，缓冲区满后或遇到endl时向显示器输出。

C++工具

异常检测：

Catch块是try-catch结构中的一部分，必须紧跟在try块之后，不能单独使用，两者之间不能插入其他语句。

指向常量的指针vs常量指针

const出现在\*左边，表述被指物是常量，const出现在\*右边，表示指针是常量。如果出现在\*两遍，表示被指物和指针两者都是常量

最新的C++特性，登录boost查看。暂时整理这些，随时更新。

# **第五章**

## 函数模板

Template <class Type>     //声明函数模板

Type GetMax(Type a,Type b, Type c);

//定义函数模板

Template <class Type>

Type GetMax(Type a,Type b, Type c){

……

}

然后调用函数GetMax时就可以传入不同类型的参数

## 函数模板重载

1首先匹配类型完全相同的重载函数

2其次才寻求函数模板来匹配

## 类模板

//声明类模板

Template <class Type>

Class TClass

{

Public:

         TClass(Type);

         Void SetItem(Type);

Private:

         Type m\_item;

};

//定义类模板的构造函数

Template<class Type>

TClass<Type>::TClass(Type item)  
{

         M\_item=item;

}

每个成员函数定义前必须用与声明该了模板一样的template形式加以声明。

## 类模板的派生

1类模板直接派生出新的类模板

2用类模板生成的模板类派生新的非模板类。

# **第六章**

## 标准模板库（STL）

主要包括三个部分：迭代器，容器，算法

|  |  |  |
| --- | --- | --- |
| 功能 | 头文件 | 说明 |
| 迭代器 | iterator |  |
| 输入输出流 | iostream | 标准输入输出流 |
| 文件输入输出流 | fstream | 文件输入输出流 |
| 字符串 | sstream  string | 字符串输入输出流 |
| 函数对象 | functional |  |
| 通用容器 | vector  list  queue  deque  stack | 向量  链表  队列  双端队列  栈 |
| 通用算法 | algorithm | 通用算法容器 |

## 函数对象

通过重载类中的函数运算符（）实现函数对象，可以对容器中的数据进行各种操作，同时还能维护自己的状态。

## 常用通用容器

vector：向量（-动态数组）

**构造函数模板**

vector（）：创建一个空vector

vector（int n）：创建一个元素个数为n的vector

vector（int nSize，const Type& e）：创建一个vector，元素个数为nSize，且各元素的值为e

vector（const vector&）：拷贝构造函数

**增加函数模板**

void push\_back（const Type& e）：向量尾部添加一个元素e

iterator  insert（iterator it，const Type& e）：向量中某一元素前增加一个元素e

void insert（iterator it，int n，const Type& e）：向量中某一元素前增加n个相同元素e

**删除函数模板**

iterator erase（iterator it）：删除向量中某一元素

iterator erase（iterator first，iterator last）：删除向量中[first,last]中的元素

void pop\_back（）：删除向量中最后一个元素

void clear（）：清空向量

**遍历函数模板**

Type& at（int pos）：返回pos位置的元素的引用

Type &front（）：返回首元素的引用

Type &back（）：返回尾元素的引用

iterator begin（）：返回向量头迭代器，指向第一个元素

iterator end（）：返回向量尾迭代器，指向最后一个元素之后的空

reverse\_iterator rbegin（）：反向迭代器，返回最后一个元素的迭代器，用于逆向遍历容器

reverse\_iterator rend（）：反向迭代器，返回第一个元素之前的迭代器，用于逆向遍历容器

**其他函数模板**

void assign（int n，const Type& e）：将向量中第n个元素设置为e

bool empty（）：向量是否为空

int size（） const：返回向量中元素的个数

## list

list是双向链表，每次插入或删除一个元素就配置或释放一个元素控件。

**构造函数模板**

list（）：空list

list（const list&）：拷贝构造函数

**增加、删除函数模板**

void push\_back（const Type& e）：尾部增加e

void push\_front（const Type& e）：首部插入e

void pop\_back（）：当且仅当容器不为空时，删除尾部元素

void pop\_front（）：当且仅当容器不为空时，删除首部元素

void remove（const Type& e）：删除容器中所有元素值等于e的元素

void clear（）：清空容器

iterator insert（iterator it，const Type &e=Type（））：在迭代器it前插入元素e，返回指向e的迭代器

void insert（iterator it，int n，const Type& e）：在迭代器it前插入n个相同的元素e

iterator erase（iterator it）：删除迭代器it对应的元素

iterator erase（iterator first，iterator last）：删除迭代器[first,last)间的元素

**遍历函数模板**

iterator begin（）

iterator end（）

reverse\_iterator rbegin（）

reverse\_iterator rend（）

Type& front（）：返回首元素的引用

Type& back（）：返回尾元素的引用

**其他函数模板**

void sort（）：容器内所有元素按升序排序

void reverse（）：反转容器中元素顺序

int size（）const

bool empty（）const

## deque容器

支持高效插入和删除容器的首尾部元素。也叫双端队列。

**构造函数模板**

deque（）：空deque

deque（const deque&）：拷贝构造函数

**增加函数模板**

void push\_back（const Type& e）：队列尾部增加e

void push\_front（const Type& e）：队列首部增加e

**删除函数模板**

void pop\_front（）：删除队首元素

void pop\_back（）：删除队尾元素

void clear（）：清空

**遍历函数模板**

Type& front（）

Type& back（）

iterator begin（）

iterator end（）

reverse\_iterator rbegin（）

reverse\_iterator rend（）

**其他函数模板**

int size（） const

bool empty（） const

## queue和stack容器

queue称为队列，只允许在表的一端插入，另一端删除，允许插入的叫队尾（入队），允许删除的叫队首（出队），是先进先出的线性表。

stack称为栈，只允许在一端进行插入和删除操作，后进先出。

queue和stack只是起一个中介作用，就是一个容器配接器，是对其他基础容器类（lsit、vector、deque）的再封装，底层调用的就是其他基础容器类的函数。所以queue和stack在创建的时候，要指明底层使用哪一个基础容器类。

**构造函数模板**

queue（class Type，class Container=deque<Type>）：创建元素类型为Type的空队列，默认容器是deque

stack（class Type，class Container=deque<Type>）：创建元素类型为Type的空栈，默认容器是deque

示例：

stack<int> s; //创建类型为int的栈，选用了默认容器deque<int>

stack<int,vector<int>> s;  //创建类型为int的栈，使用了容器vector<int>

stack<int,list<int>> s;   //创建类型为int的栈，使用了容器list<int>

queue<int> q;  //创建类型为int的队列，使用默认容器deque<int>

以此类推……

**队列和栈公有函数模板**

bool empty（）

int size（）

void push（const Type& e）：把元素e压入队尾（栈顶）

void pop（）：在队列（栈）非空情况下，删除队头（栈顶）元素

**队列独有函数模板**

Type& front（）

Type& back（）

**栈独有函数模板**

Type& top（）：在栈非空情况下，返回栈顶元素的引用

## STL常用算法

**遍历算法for\_each（）**

原型

template<class It, class Fun>

Fun for\_each(It first, It last, Fun& f);   //对[first,last)的元素，作为函数f的参数传入并执行函数功能。

**查询算法find（）**

原型

template<class It, class Type>

It find(It first, It last, const Type& e);

查询[first,last)间对应的元素值等于e的，若有则返回迭代器，否则返回last

**排序算法sort（）**

原型

//将[first,last)间的元素默认升序排序

template<class It>

void sort(It first, It last);

//将[first,last)间的元素根据比较函数pr（x，y）代替operator<(x,y)，进行排序。

template<class It, class Pred>

void sort(It first, It last, Pred pr);

示例

sort(v.begin(),v.end());//默认升序排序

sort(v.begin(),v.end(),greater<int>());

//用greater<int>函数对象作为参数，改为降序排序。若换位函数对象less<int>则为升序排序

# **第七章**

## 标准输入输出

输出流对象都是ostream类，cout是ostream类的一个实例，用于向显示屏输出。

**cout**

cout<<”hello,world”<<endl;

输出流ostream类成员函数

put()：输出单个字符

输出流对象.put（ch）

**cin**

输入流对象都是istream类，cin是istream类的一个实例，用于接受从键盘的输入。

cin>>n;

cin 输入会自动跳过空格、制表符、回车等空白符。

输入流istream类成员函数

**get（）**

输入单个字符，使用方法：输入流对象.get();

函数返回所输入的字符。若遇到输入流中的文件结束，则返回结束标志EOF。

**getline（）**

从输入流对象中输入一行字符

输入流对象.getline(字符指针，字符个数)；

如果一行字符个数小于n，则字符指针指向的的字符串存储实际输入的字符；如果字符个数大于n或等于n，则字符指针指向的字符串存储n-1个字符，因为末尾要存’\0’字符。

**示例：**

char s[256];

cin.getline（s，8）；

eof（）

输入流对象.eof()

eof在读取完最后一个数据后，仍是False，当再次试图读一个数据时，由于发现没数据可读了，才知道到末尾了，此时才修改标志，eof变为True。

**peek（）**

返回输入流中当前的字符，但输入流对象的位置不变，不后移。如果当前字符为结束符，返回EOF

**putback（）**

输入流对象.putback(ch)

将前面用get函数从输入流中得到的字符返回到输入流中，插入到当前束流对象的当前位置。

## 文件流

如果要对外存文件进行操作，应先定义一个文件流类对象，然后通过文件流类对象操作数据。

ofstream outFile;   //输出文件流

ifstream inFile;   //输入文件流

fstream file;   //输入输出文件流

## 打开外存文件

文件操作方式用在类ios中定义的枚举值来表示

|  |  |
| --- | --- |
| 文件操作方式 | 功能 |
| ios::in | 以输入方式打开文件，如果文件不存在则出错，否则打开成功，是文件流类ifstream的默认打开方式，打开后文件指针当前位置在文件的开始处 |
| ios：：out | 以输出方式打开文件，如果文件不存在将创建一个新文件，否则将文件清空，是文件流类ofstream的默认打开方式，打开后文件指针在当前文件的开始处 |
| ios：：app | 以追加的方式打开文件，如果文件已存在则不清楚原有内容，否则出现运行时错误，打开后文件指针的当前位置在文件结尾处 |
| ios：：binary | 以二进制方式打开文件，如不指定此方式则默认为ASCII方式打开文件 |

## 打开文件包括两个方面内容

1为文件流对象和特定的外存文件建立关联

2指定文件的操作方式

## 可通过两种方式打开文件

1.使用文件流类的成员函数open（磁盘文件名，文件操作方式）

ofstream outFile；

outFile.open（“text.txt”，ios：：out）；

2.在定义文件流类时用带有参数的构造函数，其中包含了文件名和打开方式。

ofstream（“text.txt”，ios：：out）；

**fail()**

如果打开操作失败，文件流成员函数fail（）返回非0值，否则返回0。fail（）函数使用方法如下

文件流对象.fail（）

打开文件完整示例：

ofstream outFile(“text.txt”,ios::out);

if(outFile.fail())

{//打开文件失败}

//打开文件成功，进行后续操作……

**关闭文件**

文件流对象.close（）

## 读写文本文件

1可以使用<<和 >>进行读写

2可以使用文件流类成员函数put（）、get（）、getline（）读写

## **读写二进制文件**

二进制文件将按内存中数据存储形式不加转换地传送到外存文件。

1用文件流类成员函数read（）和write（）实现

文件流对象.read（字符指针，长度）

文件流对象.write（字符指针，长度）

其中字符指针用于指向内存中一块存储空间，长度指读、写的字节数

示例：

int a[]={1,5,2,3};

f.write（（char\*）&a[0],sizeof(a)）;

2采用与文件指针有关的流成员函数来实现随机访问二进制文件

对文件指针的操作

|  |  |
| --- | --- |
| 文件指针操作 | 功能 |
| tellg（） | 返回输入文件指针的当前位置 |
| seekg（位置） | 将输入文件中指针移到指定位置 |
| seekg（位移量，参照位置） | 以参照位置为基础移动若干个字节 |
| tellp（） | 返回输出文件指针的当前位置 |
| seekp（位置） | 将输出文件的指针移到指定位置 |
| seekp（位移量，参照位置） | 以参照位置为基础移动若干字节 |

函数中的“位置”和“位移量”都是长整型，以字节为单位。对于“位移量”，为正数时，向后移，为负数时，向前移。“参照位置”可以是下面三个枚举量之一

ios：：beg        文件开头

ios：：cur                 当前位置

ios：：end        文件结尾

# **第八章**

## 动态内存管理

内存分为堆区，栈区，静态数据区

静态数据区：程序编译时九尾静态变量和静态对象分配了静态数据存储区。在静态数据区中存储的变量或对象在该程序的整个运行期间都存在（会反应在生成的exe执行程序的大小上。另外执行程序的大小会有优化，如果静态变量没有初始化或者初始化为0，也不会反应在exe执行程序的大小上，如果静态数据初始化为非默认值，就会反应在exe执行程序的大小上），它们的生命周期贯穿整个程序的运行周期。比如全局变量、static变量等就存储在静态数据区。

栈区：调用函数时，函数内的局部变量和形式参数等在栈区中分配存储单元。这部分变量的生命周期与函数的生命周期相同。函数执行结束，这些变量的的存储单元也就被释放了。

堆区：程序员自己在程序中使用new运算符时，系统工会在堆区分配空间。它是唯一一个生存周期可以由程序员自己控制的存储区。可以自己用new申请空间，当不用时必须用delete释放空间。

## new运算符与delete运算符

在C语言中使用malloc和free申请和释放内存空间

new运算符将根据用户自定义类型或者系统定义类型自动执行对应的构造函数，分配相应的空间。delete运算符可根据对象类型自动执行其析构函数，完成对象的销毁和内存空间的释放。

new可以自动计算所申请空间的大小，而malloc需要程序员指定所需申请的空间大小。所以new比malloc功能更强大

## new使用方式

<指针名>=new<类型>(<实参表>)

当内存不足时，new失败，返回NULL

使用new创建数组时，不能指定初始值。

new生成一个对象数组时不能带参数，即生成对象数组时只能调用无参构造函数或者全部参数都有默认值的构造函数。如果类中定义了构造函数，但没有定义无参构造函数，且没有定义全部都有初始值的构造函数，则该类不能用默认构造函数创建，也因此不能用new创建该类的对象数组。

## delete使用方式

delete <指针名>

delete []<指针名>

delete执行后，原指针所指向的内存空间就被释放了，而指向该动态内存的指针不会发生变化，依然指向这块动态内存。但是这部分内存已经被释放，而且以后随时有可能被分配给其他变量，所以此后通过该指针变量对该内存的任何操作都可能引发系统崩溃。此时用（p！=NULL）防错毫无意义。所以应该在delete掉内存之后，立即手动将指针置为NULL。

如果new和delete使用不当，可能发生内存泄漏。例如，delete以后不对指针变量进行任何处理，会造成“指针悬挂”；指针变量声明以后，没有初始化就通过它访问内存空间，可能产生“野指针”问题等等。

### 关于内存的注意事项：

1定义指针变量的同时初始化。如不初始化就设置为NULL，避免该指针指向一个不确定的地方，引发误操作。

2delete指针以后，第一时间将其设置为NULL。

3当指针指向数组时，谨防越界。

4不要用指针传递栈内存。不要返回一个即将自动消失的局部变量或局部对象的地址。栈内存中的局部变量和局部对象不能跨函数生存。要随时注意指针变量本身以及指针所指向的对象的生存周期。（函数内部的局部变量在函数结束后就会被销毁，所以不能通过指针返回一个局部变量的地址）

5注意区分指针变量和指针指向的内存空间之间的差异：指针变量p和它所指向的内存空间是两个不同的概念。

例如：

void Fun（）{

char\* p=new char[100]

}

指针变量p位于栈内存，长度100的数组位于动态内存空间，位于堆内存，所以函数结束后，指针变量p被销毁，而长度100的数据没有被销毁。造成了内存泄漏。

6关于指针和动态内存注意两点：指针消亡了，不表示它所指向的动态内存会被自动释放、自动消亡；动态内存释放了，不代表指向该动态内存的指针会消亡或者自动变成NULL。

## 常见动态内存错误

1动态内存分配失败却继续操作

2动态内存空间未经初始化就使用

3动态内存空间越界使用

4内存泄漏

5继续使用已经被释放的动态内存控件

## 指针和数组的差异

数组名代表数组所在的内存空间的首地址。为数组所分配的内存空间大小以及数组元素的多少等在数组的生命期内不能改变，但数组的内容可以改变。

指针变量就是一个可用于存放地址的变量，不同类型的地址需要不同类型的指针变量来存放。由于系统的所有地址都是等长的，所以不同类型的指针变量都是等长的，而且不同类型的地址之间可以很方便的相互转换。换句话说，一个指针变量几乎可以存放任意数据类型的地址值，即它可以任意指向任何类型的内存块。

示例：

//数组定义：内存中有数组空间，没有字符串常量

char a1[]=”hello Iron Man!”;    //定义数组并初始化

a1[14]=’2’;    //数组的元素值是可变的

//指针变量定义并初始化，内存中有指针变量，并在静态存储区还有一个字符串常量

//内存中分配两个空间：指针变量空间，字符串常量空间

char\* p1=”hello world”;    //p1指向字符串常量

p1[0]不可修改，因为p1指向常量

//字符串数组赋值方法

char a2[]=”hello baby!”;

char b2[81];

strcpy(b2,a2);

//复制指针所指向的字符串的方法

int len=strlen(a2);

char\* p2=new char[len+1];

if(p2==NULL)

{

         exit(0);

}

strcpy(p2,a2);

//字符指针变量的长度和该指针所指向的字符串长度的区别

cout<<sizeof(a2)<<endl;  //12字节，a2是一个字符数组

cout<<sizeof(p2)<<endl;  //4字节，p2是一个字符指针变量

## 利用指针传递内存

可以使用指针、指针的引用以及函数返回指针来传递动态内存。

## 函数参数传递方式

传值、传引用、传指针、传指针的引用

传值：常用于简单基本类型的参数的传递，即复制一份，将新的复制的值传入函数

传引用：加&符号，即将该值本身传入函数，在函数中修改该值，函数外也被修改。

传指针：复制一个指针，和源指针指向相同位置，将复制后的指针传入函数中。

传指针的引用：将该指针本身传入函数。

**利用指针传值**

char\* TestPointerForwardValue(char\* pChar2);

**利用指针的引用传值**

char\* TestReferencePointer(char\* &rpChar);

## 类型转换

简单基本类型的类型转换

int a=10;

double b=(double)a;

double b=double(a);

### 基本类型转换为类类型

需要类定义有只有一个参数的构造函数或者其他参数都有默认值的构造函数

### 类类型转换为基本类型

需要类重载类型转换函数

operator <基本数据类型名>()

{

         //……

         return <基本数据类型值>;

}

示例：

operator double()

{

         return m\_value;

}

与以前的重载运算符函数不同的是，类类型转换函数没有返回值类型，因为<基本数据类型名>就代表了返回类型。类类型转换函数也没有任何参数，在调用过程中要传入一个对象实参。

示例：

int a;

//类类型à基本类型

a=int(oCF);或者a=oCF;

//基本类型à类类型

oCF=10;

## 命名空间

**命名空间的使用**

namespace namespace\_name

{

         //各种名字的定义

}

**使用命名空间语法规则**

1命名空间名字：：命名空间成员

2 using namespace 命名空间名字

可以对命名空间的名字起别名

示例：

namespace N2=YourNamespace；

编程的学习学无止境，只掌握一门语言是远远不够的，现在我们开始C++的学习之路，下面先看下C++ 与C 的区别

一、C++概述

1、发展历史

1980年，Bjarne Stroustrup博士开始着手创建一种模拟语言，能够具有面向对象的程序设计特色。在当时，面向对象编程还是一个比较新的理念，Stroustrup博士并不是从头开始设计新语言，而是在C语言的基础上进行创建。这就是C++语言。

1985年，C++开始在外面慢慢流行。经过多年的发展，C++已经有了多个版本。为次，ANSI和ISO的联合委员会于1989年着手为C++制定标准。1994年2月，该委员会出版了第一份非正式草案，1998年正式推出了C++的国际标准。

2、C和C++

C++是C的超集，也可以说C是C++的子集，因为C先出现。按常理说，C++编译器能够编译任何C程序，但是C和C++还是有一些小差别。

例如C++增加了C不具有的关键字。这些关键字能作为函数和变量的标识符在C程序中使用，尽管C++包含了所有的C，但显然没有任何C++编译器能编译这样的C程序。

C程序员可以省略函数原型，而C++不可以，一个不带参数的C函数原型必须把void写出来。而C++可以使用空参数列表。

C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。

标准C++中的字符串类取代了C标准C函数库头文件中的字符数组处理函数（C中没有字符串类型）。

C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。

C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。

二、关键字和变量

C++相对与C增加了一些关键字，如下：

typename bool dynamic\_cast mutable namespace

static\_cast using catch explicit new

virtual operator false private template

volatile const protected this wchar\_t

const\_cast public throw friend true

reinterpret\_cast try

bitor xor\_e and\_eq compl or\_eq

not\_eq bitand

在C++中还增加了bool型变量和wchar\_t型变量：

布尔型变量是有两种逻辑状态的变量，它包含两个值：真和假。如果在表达式中使用了布尔型变量，那么将根据变量值的真假而赋予整型值1或0。要把一个整型变量转换成布尔型变量，如果整型值为0，则其布尔型值为假；反之如果整型值为非0，则其布尔型值为真。布儿型变量在运行时通常用做标志，比如进行逻辑测试以改变程序流程。

#include iostream.h

int main()

{

bool flag;

flag = true;

if(flag)

cout << true << endl;

return 0;

}

C++中还包括wchar\_t数据类型，wchar\_t也是字符类型，但是是那些宽度超过8位的数据类型。许多外文字符集所含的数目超过256个，char字符类型无法完全囊括。wchar\_t数据类型一般为16位。

标准C++的iostream类库中包括了可以支持宽字符的类和对象。用wout替代cout即可。

#include iostream.h

int main()

{

wchar\_t wc;

wc = 'b';

wout << wc;

wc = 'y';

wout << wc;

wc = 'e';

wout << wc << endl;

return 0;

}

说明一下：某些编译器无法编译该程序(不支持该数据类型)。

三、强制类型转换

有时候，根据表达式的需要，某个数据需要被当成另外的数据类型来处理，这时，就需要强制编译器把变量或常数由声明时的类型转换成需要的类型。为此，就要使用强制类型转换说明，格式如下：

int\* iptr=(int\*) &table;

表达式的前缀(int\*)就是传统C风格的强制类型转换说明(typecast)，又可称为强制转换说明(cast)。强制转换说明告诉编译器把表达式转换成指定的类型。有些情况下强制转换是禁用的，例如不能把一个结构类型转换成其他任何类型。数字类型和数字类型、指针和指针之间可以相互转换。当然，数字类型和指针类型也可以相互转换，但通常认为这样做是不安全而且也是没必要的。强制类型转换可以避免编译器的警告。

long int el = 123;

short i = (int) el;

float m = 34.56;

int i = (int) m;

上面两个都是C风格的强制类型转换，C++还增加了一种转换方式，比较一下上面和下面这个书写方式的不同：

long int el = 123;

short i = int (el);

float m = 34.56;

int i = int (m);

使用强制类型转换的最大好处就是：禁止编译器对你故意去做的事发出警告。但是，利用强制类型转换说明使得编译器的类型检查机制失效，这不是明智的选择。通常，是不提倡进行强制类型转换的。除非不可避免，如要调用malloc()函数时要用的void型指针转换成指定类型指针。

四、标准输入输出流

在C语言中，输入输出是使用语句scanf()和printf()来实现的，而C++中是使用类来实现的。

#include iostream.h

main() //C++中main()函数默认为int型，而C语言中默认为void型。

{

int a;

cout << input a number: ;

cin >> a; /\*输入一个数值\*/

cout << a << endl; //输出并回车换行

return 0;

}

cin,cout,endl对象，他们本身并不是C++语言的组成部分。虽然他们已经是ANSI标准C++中被定义，但是他们不是语言的内在组成部分。在C++中不提供内在的输入输出运算符，这与其他语言是不同的。输入和输出是通过C++类来实现的，cin和cout是这些类的实例，他们是在C++语言的外部实现。

在C++语言中，有了一种新的注释方法，就是‘//’，在该行//后的所有说明都被编译器认为是注释，这种注释不能换行。C++中仍然保留了传统C语言的注释风格/\*……\*/。

C++也可采用格式化输出的方法：

#include iostream.h

int main()

{

int a;

cout << input a number: ;

cin >> a;

cout << dec << a << ' ' //输出十进制数

<< oct << a << ' ' //输出八进制数

<< hex << a << endl; //输出十六进制数

return 0;

}

从上面也可以看出，dec,oct,hex也不可作为变量的标识符在程序中出现。

五、函数参数问题

1、无名的函数形参

声明函数时可以包含一个或多个用不到的形式参数。这种情况多出现在用一个通用的函数指针调用多个函数的场合，其中有些函数不需要函数指针声明中的所有参数。看下面的例子：

int fun(int x,int y)

{

return x\*2;

}

尽管这样的用法是正确的，但大多数C和C++的编译器都会给出一个警告，说参数y在程序中没有被用到。为了避免这样的警告，C++允许声明一个无名形参，以告诉编译器存在该参数，且调用者需要为其传递一个实际参数，但是函数不会用到这个参数。下面给出使用了无名参数的C++函数代码：

int fun(int x,int) //注意不同点

{

return x\*2;

}

2、函数的默认参数

C++函数的原型中可以声明一个或多个带有默认值的参数。如果调用函数时，省略了相应的实际参数，那么编译器就会把默认值作为实际参数。可以这样来声明具有默认参数的C++函数原型：

#include iostream.h

void show(int = 1,float = 2.3,long = 6);

int main()

{

show();

show(2);

show(4,5.6);

show(8,12.34,50L);

return 0;

}

void show(int first,float second,long third)

{

cout << first =<< first

<< second =<< second

<< third =<< third << endl;

}

上面例子中，第一次调用show()函数时，让编译器自动提供函数原型中指定的所有默认参数，第二次调用提供了第一个参数，而让编译器提供剩下的两个，第三次调用则提供了前面两个参数，编译器只需提供最后一个，最后一个调用则给出了所有三个参数，没有用到默认参数。

六、函数重载

在C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在C语言中是不允许的。

1、参数个数不同

#include iostream.h

void a(int,int);

void a(int);

int main()

{

a(5);

a(6,7);

return 0;

}

void a(int i)

{

cout << i << endl; //输出5

}

void a(int i,int j)

{

cout << i << j << endl; //输出67

}

2.参数格式不同

#include iostream.h

void a(int,int);

void a(int,float);

int main()

{

a(5,6);

a(6,7.0);

return 0;

}

void a(int i,int j)

{

cout << i << j <<endl; //输出56

}

void a(int i,float j)

{

cout << i << j << endl; //输出67.0

}

七、变量作用域

C++语言中，允许变量定义语句在程序中的任何地方，只要在是使用它之前就可以；而C语言中，必须要在函数开头部分。而且C++允许重复定义变量，C语言也是做不到这一点的。看下面的程序：

#include iostream.h

int a;

int main()

{

cin >> a;

for(int i = 1;i <= 10; i++) //C语言中，不允许在这里定义变量

{

static int a = 0; //C语言中，同一函数块，不允许有同名变量

a += i;

cout<<::a<< <<a<<endl;

}

return 0;

}

八、new和delete运算符

在C++语言中，仍然支持malloc()和free()来分配和释放内存，同时增加了new和delete来管理内存。

1.为固定大小的数组分配内存

#include iostream.h

int main()

{

int \*birthday = new int[3];

birthday[0] = 6;

birthday[1] = 24;

birthday[2] = 1940;

cout << I was born on

<< birthday[0] << '/' << birthday[1] << '/' << birthday[2] << endl;

delete [] birthday; //注意这儿

return 0;

}

在删除数组时，delete运算符后要有一对方括号。

2.为动态数组分配内存

#include iostream.h

#include stdlib.h

int main()

{

int size;

cin >> size;

int \*array = new int[size];

for(int i = 0;i < size;i++)

array[i] = rand();

for(i = 0;i < size;i++)

cout << '\n' << array[i];

delete [] array;

return 0;

}

九、引用型变量

在C++中，引用是一个经常使用的概念。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。

1.引用是一个别名

C++中的引用是其他变量的别名。声明一个引用型变量，需要给他一个初始化值，在变量的生存周期内，该值不会改变。& 运算符定义了一个引用型变量：

int a;

int& b=a;

先声明一个名为a的变量，它还有一个别名b。我们可以认为是一个人，有一个真名，一个外号，以后不管是喊他a还是b，都是叫他这个人。同样，作为变量，以后对这两个标识符操作都会产生相同的效果。

#include iostream.h

int main()

{

int a = 123;

int& b = a;

cout << a << ','<< b << endl; //输出123,123

a++;

cout << a << ','<< b << endl; //输出124,124

b++;

cout << a<< ',' << b << endl; //输出125,125

return 0;

}

2.引用的初始化

和指针不同，引用变量的值不可改变。引用作为真实对象的别名，必须进行初始化，除非满足下列条件之一：

(1) 引用变量被声明为外部的，它可以在任何地方初始化

(2) 引用变量作为类的成员，在构造函数里对它进行初始化

(3) 引用变量作为函数声明的形参，在函数调用时，用调用者的实参来进行初始化

3.作为函数形参的引用

引用常常被用作函数的形参。以引用代替拷贝作为形参的优点：

引用避免了传递大型数据结构带来的额外开销

引用无须象指针那样需要使用\*和->等运算符

#include iostream.h

void func1(s p);

void func2(s& p);

struct s

{

int n;

char text[10];

};

int main()

{

static s str = {123,China};

func1(str);

func2(str);

return 0;

}

void func1(s p)

{

cout << p.n << endl;

cout << p.text << endl;

}

void func2(s& p)

{

cout << p.n << endl;

cout << p.text << endl;

}

从表面上看，这两个函数没有明显区别，不过他们所花的时间却有很大差异，func2()函数所用的时间开销会比func2()函数少很多。它们还有一个差别，如果程序递归func1()，随着递归的深入，会因为栈的耗尽而崩溃，但func2()没有这样的担忧。

4.以引用方式调用

当函数把引用作为参数传递给另一个函数时，被调用函数将直接对参数在调用者中的拷贝进行操作，而不是产生一个局部的拷贝(传递变量本身是这样的)。这就称为以引用方式调用。把参数的值传递到被调用函数内部的拷贝中则称为以传值方式调用。

#include iostream.h

void display(const Date&,const char\*);

void swapper(Date&,Date&);

struct Date

{

int month,day,year;

};

int main()

{

static Date now={2,23,90};

static Date then={9,10,60};

display(now,Now: );

display(then,Then: );

swapper(now,then);

display(now,Now: );

display(then,Then: );

return 0;

}

void swapper(Date& dt1,Date& dt2)

{

Date save;

save=dt1;

dt1=dt2;

dt2=save;

}

void display(const Date& dt,const char \*s)

{

cout << s;

cout << dt.month << '/' << dt.day << '/'<< dt.year << endl;

}

5.以引用作为返回值

#include iostream.h

struct Date

{

int month,day,year;

};

Date birthdays[]=

{

{12,12,60};

{10,25,85};

{5,20,73};

};

const Date& getdate(int n)

{

return birthdays[n-1];

}

int main()

{

int dt=1;

while(dt!=0)

{

cout<<Enter date # (1-3,0 to quit)<<endl;

cin>>dt;

if(dt>0 && dt<4)

{

const Date& bd = getdate(dt);

cout << bd.month << '/' << bd.day << '/'<< bd.year << endl;

}

}

return 0;

}

# c++

c++名字的由来：

C语言＋面向对象的方法论=objective c/c++

# C++中最基本的头文件和命名空间：

Namespace：

Namespace是指标识符的各种可见范围。C++标准程序库中的所有标识符都被定义于一个名为std的namespace中

<iostream>后缀为.h的头文件c++已经不支持。当使用iostream的时候，相当于在c中调用库函数，使用的是全局命名空间，必须使用namespace std这样才能正确使用cout

对于namespace的概念，使用c++标准的程序库的任何标志符的时候，可以有三种选择

1. 直接指定标识符，比如std：：ostream而不是ostream，完整语句：std：：cout<<std::hex<<3.4<<endl;
2. 使用using关键字。Using std::cout;using std::endl;using std::cin;以上程序可以写为：cout<<std::hex<<3.4<<endl;
3. 最方便的就是using namespace std；为了避免程序员在选择在选择类的名称或函数名的时候可能引起冲突，所以把标准库中的一切全部放在空间std中

在c++文件中，在一开始一定要协商#include“iostream”//它的意思是包含c++的头文件

Using namespace std；//使用命名空间std标准的命名空间（在这个命名空间中定义了很多标准定义。

C语言中只有一个全局作用域，导致c语言所有的全部标识符共享一个作用域，这就导致了变量符之间可能发生冲突

C++提出了命名空间的概念，命名空间讲全局作用域分成不同的部分，不同的命名空间中的标识符可以同名而不会发生冲突。命名空间可以相互嵌套，全局作用域也叫默认命名空间

定义命名空间的格式：

Namespace XXX

{…}

命名空间里面可以嵌套

命名空间的使用：在主函数里面：using namespace XXX；

举一个例子：namespace C是嵌套在namespace B里面的，struct teacher是namespace C里面的一个结构体，这时候要调用这一个结构体是 using namespace B::namespace C：：teacher； 如果不这么使用那么在每次调用这个teacher结构体的t2数据的时候就是namespace B：：namespace C：：teacher t2；（相对于第一种方法显得比较繁琐）

# 标准输入与输出：

Cout相当于标准输出到黑屏幕上（cmd控制台）

<<左移操作符在c++中进行了功能的增强，这是c++语言操作符重载

Endl在c++里面是回车换行的意思

cout<<”hello”<<endl；可以理解为把hello流入到cout的黑窗户上，然后把endl的换行流入到黑窗户上

cin代表标准输入，代表键盘，表示从键盘输入一个数再流入这个变量

标准输入cin和标准输出cout一般都写在<<>>操作符的左边。

Cout和endl是定义在std中的，如果在文件的首端没有使用using namespace std；那么在后续使用cout和endl的时候要这么用：std：：cout；std：：endl；也就是说文件中iostream没有引入标准的std，需要程序员手工的写。

定义命名空间：命名空间的定义是可以嵌套的，只需要 namespace XXX就定义了一个命名空间

使用命名空间：using XXXX就可以在主函数中使用该命名空间里面的数据了。如果需要输出的变量在调用的多个命名空间中存在不止一个，那么使用方法是：XXXX：：a；

# 关于c++中的类：

Class是在struct的基础上建立起来的，c++是兼容c的

在c++中，把物体的特性叫做物体的属性

面向对象加工的是一个个的函数/语句，面向对象加工的是一个个的类，在类的里面包含函数和数据。

类就是一个抽象的数据类型，它和结构体不一样的地方之一是类能够在类里面写函数。

类的抽象：

1. 成员变量和成员函数
2. 实例化，就是用类定义变量，就是对象，类似于c语言中给生产多个结构体。
3. 使用类，求出结果
4. Main集成测试

类代码不是一步一步指向的

类是一个数据类型（固定大小内存块的别名），定义一个类，是一个抽象的概念，不会分配内存，用数据类型定义变量的时候，才会分配内存

为什么使用成员变量没有写成员函数会导致乱码：

因为一开始的时候成员变量是随机取值的，之后进行的操作只是改变了这个成员变量在当时的情况，在取出这个成员变量的时候会导致乱码（取出的是之前的随机量）取出的时候是从变量所标志的内存空间中拿值，并没有执行我们想要的赋值的情况，在通过类型定义变量的时候，值已经固定下来了。

从深层次的讲：

成员变量在类的范围内，存在于堆内存中，会有默认的初始值，在申明的同时已经给变量赋值了。再次赋值其实使用的是语句，而类里面只能出现变量和方法，不能出现语句。

调用类中的函数方法：类名.类中的函数名（参数表）；

C++对c的加强的地方：

1. namespace命名空间
2. 实用性的加强
3. Register关键字的加强
4. 变量检测的加强
5. Struct类型加强
6. C++中所有的变量和函数都必须有类型
7. 新增bool类型关键字
8. 三目运算符功能增强

实用性加强：

原来c语言中变量要提前定义，就是说c语言中的变量必须在作用域开始的位置定义，erc++中更强调语言的“实用性“，所有的变量都可以在需要使用的时候再定义。

Register关键字的增强：

请求编译器让变量a直接放在寄存器里面，速度快

再c语言中，register修饰的变量不能取地址，但是在c++中做了内容。

C语言无法取得register变量地址，再c++中依然支持register关键字，c++编译器由自己的优化方式，不适用register也可能做优化，c++中可以取得register变量的地址

C++编译器发现程序需要取register变量的地址的时候，register对变量的声明变得无效，早期的c语言编译器不会对代码进行优化，因此register变量是一个很好的补充

在c语言中，寄存器变量上是不能取地址的

在频繁使用某一个变量的时候c++编译器会自动优化这个变量为寄存器变量

变量检测加强：

在c语言中，重复定义多个同名的全局变量是合法的，在c++中，不允许，不允许定义多个同名的全局变量，c语言中多个同名的全局变量最终会连接到全局数据的同一个地址空间上，而c++直接拒绝这种二义性的做法。

Struct类型加强：

C语言中struct定义了一组变量的集合，c编译器并不认为这是一种新的类型

C++中的struct是一个新类型的定义声明，c++对struct关键字做了功能的增强

就是有些在c中部通过的结构体，在c++中通过了（比如struct关键字在c++中，可以在主函数外面定义。

Struct关键字和class关键字完成的功能是一样的，但是是有区别的

C++中的变量和函数必须有类型（c++对变量类型的检查更加严格）：

比如在c语言中可以写 g（）{return 5}；但是在c++中是不行的，比如c++中对函数的参数表和函数的类型的检查更加严格，这些变量和函数都必须有类型。

C++中新增的bool类型关键字：

比如：bool b1=true；布尔类型占用的内存空间是一个字节

Bool关键字告诉c+++关键字给变量分配1个字节的内存、

对于bool变量，要么是1，要么是0。对于布尔变量，正数和负数都是真值，只有0布尔变量为假。

理论上bool只占用一个字节，如果多个bool变量定义在一起，可能会各占一个bit，这取决于编译器的实现。

C++中的三目运算符：

在c语言中，表达式的运算结果放在寄存器里面，表达式返回的是值，而不是内存中，它返回的是一个数，三目运算符不可以做左值。左值应该是一个变量的性质，不是一个值。

在c++中，表达式返回的是变量的本身，三目运算符返回的是一个地址，可以作为左值。

那么如何让c语言也做到：让表达式返回一个内存地址。把这个值导向变成地址导向（使用指针）

本质：c++编译器帮助程序员完成了取地址的工作

Const的基础和const符号表机制研究：

被const修饰的东西不能被修改。需要观察const修饰的是指针内存还是变量。

顺带一提：const int a；和int const a；是一样的，代表一个常整型数。

Const int \*c;//c是一个指向常整型数的指针，所指向的内存数据不能更改，但是本身是可以更改的

Int\*const d;//d常指针，指针变量不能被修改，但是它指向的内存空间可以被修改

Const int\*const e;//e是一个指向常整形的常指针，指针和它指向的内存空间都不可以修改

Const的好处：

合理使用const 1、指针做函数参数，可以有效的提高代码可读性，减少bug 2、清除的分清参数的输入和输出特性

C语言中的const是一个冒牌货，虽然直接赋值是不可行的，但是通过指针地址来进行间接赋值还是可以的。

但是在c++中，只要使用了const，无论直接赋值还是间接赋值，c++都只承认第一次的赋值。在c++中，const是一个真正的常量

因为c++编译器在看到const的修饰的时候，不再给const的变量分配内存。在c++中，const的量是存在于符号表中的，他的值被固定。当发现使用常量则直接从符号表中寻找对应的值兑换。编译过程中如果发现对const使用了extern或者&操作符，则给对应的常量分配内存空间（兼容c）。但是当像c++编译器索取const量的地址的时候，c++编译器会给这个量开辟一块内存空间。

Const’和define的相同之处：

举例子：int a=10；

Int b=20；

Int array【a+b】；//这种情况下编译是不通过的，但是在linux内核里面是可以通过的，因为编译linux内核的gcc编译器支持，c和c++都不支持。

但是，如果在上文中，a和b是用const 或者宏定义#define做的，那么能够编译

const和define的不同之处：

const常量是由编译器处理的，提供类型检查和作用域检查

宏定义由预处理器处理，单纯的文本替换

注：undef和define互为逆

const int \*p //指向 const int 类型对象的指针

int const \*q; //同上

int \*const x; //指向 int 类型对象的 const 指针；注意 const 的位置

const int \*const r; //指向 const int 类型对象的 const 指针

int const \*const t; //同上

# c++的引用：

在c++中新增加了引用的概念，引用必须要初始化，依赖于一个变量。

引用可以看作一个一定义变量的别名，两个东西本质上就是一个东西了

引用语法举例：

Int&a=b；引用作为函数参数声明时不进行初始化

复杂数据类型做函数引用：

引用（起别名）也可以用于结构体的成员，或整个结构体名。

引用的意义：

引用作为其他变量的别名而存在，因此在一些场合可以取代指针

引用相对于指针来说具有更好的可读性和实用性

引用本质探究：

普通引用也占有内存空间，很像是指针占据的空间

引用在c++内部实现是一个常量指针，因此引用所占用的空间大小和指针相同。

从使用的角度，引用会让人误会只是一个别名，没有自己的存储空间，这是c++为了实用性而做出的细节隐藏

引用的结论：

引用在实现上，只不过是把间接赋值成立的三个条件的后两步合二为一

当实参传给形参引用的时候，只不过是c++编译器帮我们程序员手工取了一个实参地址，传给形参引用（常量指针）

函数的返回值当引用：

比如：int getAA190

int& getAA2（）

c++引用使用时的难点：

当函数返回值为引用时，不能成为其他引用的初始值，不能作为左值引用

若返回静态变量或全局变量，可以成为其他引用的初始值，即可以作为右值使用，也可以作为左值使用

封装：

将数据和处理数据的程序组合起来，仅对外公开接口，达到信息隐藏的功能。

优点：能减少耦合。

耦合：某两个事物之间存在一种相互作用、相互影响的关系。

C++的类对其成员（包括数据成员、函数成员）分为三种封装状态：

公有（public）：类别的用户可以访问、使用该类别的此种成员。

保护（protected）：该类别的派生类可以访问、使用该类别的此成员。外部程序代码不可以访问、使用这种成员。

私有（private）：只有类别自身的成员函数可以访问、使用该类别的此成员。一般可以将C++类的对外接口设定为公有成员；类内部使用的数据、函数设定为私有成员；供派生自该类别的子类使用的数据、函数设定为保护成员。

# 继承：

派生类继承基类，会自动获取超类别除私有特质外的全部特质，同一类别的所有实体都会自动有该类别的全部特质，做到代码再用。C++只支持类别构成式继承，虽然同一类别的所有实体都有该类别的全部特质，但是实体能够共享的实体成员只限成员函数，类别的任何实体数据成员乃每个实体独立一份，因此对象间并不能共享状态，除非特质为参考类型的属性，或使用指针来间接共享。C++支持的继承关系为：

公有继承：最常用继承关系，含义是“is-a”关系，代表了在完全使用公有继承的对象类别之间的层次关系。

受保护继承：基类的公有或保护内容可以被派生类，以及由此派生的其他类别使用。但是基类对外界用户是不可见的。派生类的用户不能访问基类的成员、不能把派生类别转换（造型）为基类的指针或引用。

私有继承：基类的公有或保护内容仅可以被派生类访问。但基类对派生类的子类或派生类的用户都是不可见的。派生类的子类或派生类的用户都不能访问基类的内容、不能把派生类转换为基类的指针或引用。

C++支持多继承。

# 多态：

除了封装与继承外，C++还提供了多态功能，面向对象的精神在于多态，一般的多态，是指动态多态，系使用继承和动态绑定实现，使用多态可创建起继承体系。

类与继承只是达成多态中的一种手段，所以称面向对象而非面向类。  
多态又分成静态多态与动态多态。C++语言支持的动态多态必须结合继承和动态绑定方式实现。静态多态是指编译时决定的多态，包括重载和以模板实现多态的方法即参数化类型，是使用宏的“程序代码膨胀法”达到多态效果。

类型转换也是一种非参数化多态的概念，C++提供dynamic\_cast, static\_cast等运算符来实现强制类型转换。

操作数重载或函数重载也算是多态的概念。

在c语言中，&时取地址符，用在函数传参中的指针赋值；但是在c++中，&还能够表 示引用，可以提高代码执行效率，增加代码质量。

# Inline内联函数：

内联函数必须和函数体实现的地方写在一起

C++编译器直接将定义的内联函数的函数体插入到函数调用的地方

内联函数在最终生成的代码中是没有意义的

Inline函数只是编译器的加工，是一种请求，但是编译器不一定允许这种请求

内联函数省去了普通函数调用时压栈，跳转和返回的开销

关于内联函数，会接触到宏替换，宏替换的时候会碰到关于宏展开的知识点

内联函数在定义的时候需要注明时inline形式

函数参数

默认参数

c++在函数定义的时候为参数定义一个默认值，也就是说在c++中在调用自己定义的函数的时候，如果调用的括号里没有给参数赋值，那么c++编译器会自动给它一个默认值，而不会直接报错。

函数默认参数的规则：

只有函数列表后面部分的参数才可以提供默认参数值

一旦在一个函数调用中开始使用默认参数值，那么在这个自定义函数里面的后面的所有的这个参数都必须使用这一个参数值。

函数的占位参数：

在函数调用的时候必须写够参数。占位参数就是在函数的参数表里面有一个int ，但是没有变量名

默认参数和占位参数：

可以将占位参数与默认参数结合起来使用，它的意义就是为以后的程序拓展留下线索，兼容c语言程序中可能出现的不规则的写法

# 函数重载：

重载（overload）函数名相同，但是搭配不同的参数（个数，类型）

用一个函数名定义不同的参数，当参数不一样的时候，达到不同的蕴意，同一个函数名的函数能够得到不同的对参数的处理。函数名和不同参数搭配的时候函数的含义不同

函数的返回值不是函数重载的判断标准，除此之外函数名前面的返回值类型在函数重载上是不可以不一样的

重载函数的调用标准：

将所有同名函数作为候选者

尝试寻找可行的候选参数

精确匹配实参

通过默认参数能够匹配实参

匹配失败

最终寻找到的可行候选参数不唯一，那么出现二义性，编译失败

无法匹配所有候选者，函数未定义，编译失败

函数重载在本质上是不同的函数（静态链编）

重载函数的函数类型是不同的

函数返回值不能作为函数重载的依据

函数重载是由函数名和参数列表决定的

当默认参数遇见函数重载：

这样子函数调用时出现二义性，会失败

函数重载和函数指针在一起：

函数可以赋值给指针，之后就可以这样子使用P（1）；

函数指针：

声明一个函数类型

定义一个指针，这个指针指向函数的入口地址

声明一个指针的数据类型

通过函数指针类型定义一个函数指针

定义一个函数指针 变量

类的封装和访问控制：

面向对象的三大概念：封装，继承，多态

# 类的封装：

封装的含义：

1. 对属性和方法进行封装
2. 对成员的量和成员进行访问控制

一个理念：

类的内部和类的外部：

定义类的函数体

类的外部：在定义类的部分

类的访问控制关键字：

Public：修饰的成员变量和函数，可以在类的内部和类的外部访问

Private：修饰的成员变量和成员函数，只能在类的内部被访问，不能在类的外部访问

Protected：修饰的变量和函数，只能在类的内部被访问，不能在类的外部访问。但是在继承里面，protected可以在子类里面使用。

如果在类的外部使用调用非public的成员，会被编译器报错

类的封装，public是这个封装的接口

在c++中，没有权限修饰的成员变量函数都默认为私有属性

Struct和class关键字的区别：

Struct关键字定义的类，默认是public

类的声明和类的实现：

就是定义一个自己想要的类，然后在主函数中使用这些类，调用出所需要的类中的成员

对象的构造和析构：

创造一个对象的时候，常常需要做某些初始化的工作，例如对数据成员赋初值。注意，类的数据成员是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，而是在建立对象的时候自动执行。

关于构造函数：

C++中的类可以定义与类名相同的特殊成员函数，这种与类名相同的成员函数叫做构造函数

构造函数在定义的时候可以由参数

没有任何返回类型的声明

构造函数的调用：

自动调用：一般情况下，c++编译器会自动调用构造函数

手动调用：在一些情况下则需要手工调用构造函数

有关构造函数：

析构函数的定义及调用：

C++中的类可以定义一个特殊的成员函数清理对象，这个特殊的成员函数叫做析构函数

语法：~calssname（）

析构函数没有参数也没有任何返回类型的声明

析构函数在对象销毁时自动被调用

析构函数的调用机制：c++编译器自动调用

析构函数是使用在类的内部的

# 构造和析造的使用：

析构函数(destructor) 与构造函数相反，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统自动执行析构函数。析构函数往往用来做“清理善后” 的工作（例如在建立对象时用new开辟了一片内存空间，delete会自动调用析构函数后释放内存）。

与构造函数相反，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。以C++语言为例：析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。 [1]

C++当中的析构函数格式如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class <类名>  {       public:         ~<类名>();  };  <类名>::~<类名>()  {      //函数体  } |

如以下定义是合法的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class T  {     public:      ~T()；  };      T::~T()  {      //函数体  } |

当程序中没有析构函数时，系统会自动生成以下析构函数：

<类名>::~<类名>(){}，即不执行任何操作。

1. **引用简介**

引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。

　　引用的声明方法：类型标识符 &引用名=目标变量名；

　　【例1】：int a; int &ra=a; //定义引用ra,它是变量a的引用，即别名

　　说明：

　　（1）&在此不是求地址运算，而是起标识作用。

　　（2）类型标识符是指目标变量的类型。

　　（3）声明引用时，必须同时对其进行初始化。

　　（4）引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，且不能再把该引用名作为其他变量名的别名。

　　 ra=1; 等价于 a=1;

　　（5）声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。故：对引用求地址，就是对目标变量求地址。&ra与&a相等。

　　（6）不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。

**二、引用应用**

**1、引用作为参数**

　　引用的一个重要作用就是作为函数的参数。以前的C语言中函数参数传递是值传递，如果有大块数据作为参数传递的时候，采用的方案往往是指针，因为 这样可以避免将整块数据全部压栈，可以提高程序的效率。但是现在（C++中）又增加了一种同样有效率的选择（在某些特殊情况下又是必须的选择），就是引 用。

　　【例2】：

void swap(int &p1, int &p2) //此处函数的形参p1, p2都是引用

{ int p; p=p1; p1=p2; p2=p; }

　　为在程序中调用该函数，则相应的主调函数的调用点处，直接以变量作为实参进行调用即可，而不需要实参变量有任何的特殊要求。如：对应上面定义的swap函数，相应的主调函数可写为：

[复制代码](javascript:void(0);)

main( )

{

　int a,b;

　cin>>a>>b; //输入a,b两变量的值

　swap(a,b); //直接以变量a和b作为实参调用swap函数

　cout<<a<< ' ' <<b; //输出结果

}

[复制代码](javascript:void(0);)

　　上述程序运行时，如果输入数据10 20并回车后，则输出结果为20 10。

　　由【例2】可看出：

　　（1）传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

　　（2）使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给 形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效 率和所占空间都好。

　　（3）使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用"\*指针变量名"的 形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

　　如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。

**2、常引用**

　　常引用声明方式：const 类型标识符 &引用名=目标变量名；

　　用这种方式声明的引用，不能通过引用对目标变量的值进行修改,从而使引用的目标成为const，达到了引用的安全性。

　　【例3】：

int a ;

const int &ra=a;

ra=1; //错误

a=1; //正确

　　这不光是让代码更健壮，也有些其它方面的需要。

　　【例4】：假设有如下函数声明：

string foo( );

void bar(string & s);

　　那么下面的表达式将是非法的：

bar(foo( ));

bar("hello world");

　　原因在于foo( )和"hello world"串都会产生一个临时对象，而在C++中，这些临时对象都是const类型的。因此上面的表达式就是试图将一个const类型的对象转换为非const类型，这是非法的。

　　引用型参数应该在能被定义为const的情况下，尽量定义为const 。

　　**3、引用作为返回值**

　　要以引用返回函数值，则函数定义时要按以下格式：

类型标识符 &函数名（形参列表及类型说明）  
{函数体}

　　说明：

　　（1）以引用返回函数值，定义函数时需要在函数名前加&

　　（2）用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

　　【例5】以下程序中定义了一个普通的函数fn1（它用返回值的方法返回函数值），另外一个函数fn2，它以引用的方法返回函数值。

[复制代码](javascript:void(0);)

#include <iostream.h>

float temp; //定义全局变量temp

float fn1(float r); //声明函数fn1

float &fn2(float r); //声明函数fn2

float fn1(float r) //定义函数fn1，它以返回值的方法返回函数值

{

　temp=(float)(r\*r\*3.14);

　return temp;

}

float &fn2(float r) //定义函数fn2，它以引用方式返回函数值

{

　temp=(float)(r\*r\*3.14);

　return temp;

}

void main() //主函数

{

　float a=fn1(10.0); //第1种情况，系统生成要返回值的副本（即临时变量）

　float &b=fn1(10.0); //第2种情况，可能会出错（不同 C++系统有不同规定）

　//不能从被调函数中返回一个临时变量或局部变量的引用

　float c=fn2(10.0); //第3种情况，系统不生成返回值的副本

　//可以从被调函数中返回一个全局变量的引用

　float &d=fn2(10.0); //第4种情况，系统不生成返回值的副本

　//可以从被调函数中返回一个全局变量的引用

　cout<<a<<c<<d;

}

[复制代码](javascript:void(0);)

　　引用作为返回值，必须遵守以下规则：

　　（1）不能返回局部变量的引用。这条可以参照Effective C++[1]的Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

　　（2）不能返回函数内部new分配的内存的引用。这条可以参照Effective C++[1]的Item 31。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一 个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak。

　　（3）可以返回类成员的引用，但最好是const。这条原则可以参照Effective C++[1]的Item 30。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常 量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

　　（4）引用与一些操作符的重载：

　　流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl;　因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回 一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于返回一 个流指针则不能连续使用<<操作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这 就是C++语言中引入引用这个概念的原因吧。 赋值操作符=。这个操作符象流操作符一样，是可以连续使用的，例如：x = j = 10;或者(x=10)=100;赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

　　【例6】 测试用返回引用的函数值作为赋值表达式的左值。

[复制代码](javascript:void(0);)

#include <iostream.h>

int &put(int n);

int vals[10];

int error=-1;

void main()

{

put(0)=10; //以put(0)函数值作为左值，等价于vals[0]=10;

put(9)=20; //以put(9)函数值作为左值，等价于vals[9]=20;

cout<<vals[0];

cout<<vals[9];

}

int &put(int n)

{

if (n>=0 && n<=9 ) return vals[n];

else { cout<<"subscript error"; return error; }

}

[复制代码](javascript:void(0);)

　　（5）在另外的一些操作符中，却千万不能返回引用：+-\*/ 四则运算符。它们不能返回引用，Effective C++[1]的Item23详细的讨论了这个问题。主要原因是这四个操作符没有side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、返回一个局部变量的引用，返回一个new分配的对象的引用、返回一 个静态对象引用。根据前面提到的引用作为返回值的三个规则，第2、3两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为true而导致错误。所以可选的只剩下返回一个对象了。

**4、引用和多态**

　　引用是除指针外另一个可以产生多态效果的手段。这意味着，一个基类的引用可以指向它的派生类实例。

　　【例7】：

class 　A;

class 　B：public A{……};

B 　b;

A 　&Ref = b; // 用派生类对象初始化基类对象的引用

　　Ref 只能用来访问派生类对象中从基类继承下来的成员，是基类引用指向派生类。如果A类中定义有虚函数，并且在B类中重写了这个虚函数，就可以通过Ref产生多态效果。

**三、引用总结**

　　（1）在引用的使用中，单纯给某个变量取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不如意的问题。

　　（2）用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，且通过const的使用，保证了引用传递的安全性。

　　（3）引用与指针的区别是，指针通过某个指针变量指向一个对象后，对它所指向的变量间接操作。程序中使用指针，程序的可读性差；而引用本身就是目标变量的别名，对引用的操作就是对目标变量的操作。

　　（4）使用引用的时机。流操作符<<和>>、赋值操作符=的返回值、拷贝构造函数的参数、赋值操作符=的参数、其它情况都推荐使用引用。

## 一、什么是引用

引用，[顾名思义](https://www.baidu.com/s?wd=%E9%A1%BE%E5%90%8D%E6%80%9D%E4%B9%89&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)是某一个变量或对象的**别名**，对引用的操作与对其所绑定的变量或对象的操作完全等价

**语法：类型 &引用名=目标变量名；**

**特别注意：**

1.&不是求地址运算符，而是起标志作用

2.引用的类型必须和其所绑定的变量的类型相同

复制代码

1 #include<iostream>

2 using namespace std;

3 int main(){

4 double a=10.3;

5 int &b=a; //错误，引用的类型必须和其所绑定的变量的类型相同

6 cout<<b<<endl;

7 }

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824021027589-776757471.png

3.声明引用的同时**必须对其初始化**，否则系统会报错

1 #include<iostream>

2 using namespace std;

3 int main(){

4 int &a; //错误！声明引用的同时必须对其初始化

5 return 0;

6 }

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824021416636-977580793.png

4.引用相当于变量或对象的别名，因此**不能再将已有的引用名作为其他变量或对象的名字或别名**

5.引用不是定义一个新的变量或对象，因此**内存不会为引用开辟新的空间存储这个引用**

1 #include<iostream>

2 using namespace std;

3 int main(){

4 int value=10;

5 int &new\_value=value;

6 cout<<"value在内存中的地址为："<<&value<<endl;

7 cout<<"new\_value在内存中的地址为："<<&new\_value<<endl;

8 return 0;

9 }

复制代码

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824022144808-68850484.png

6.对数组的引用

**语法：类型 (&引用名)[数组中元素数量]=数组名；**

复制代码

1 #include<iostream>

2 using namespace std;

3 int main(){

4 int a[3]={1,2,3};

5 int (&b)[3]=a;//对数组的引用

6 cout<<&a[0]<<" "<<&b[0]<<endl;

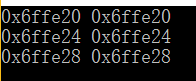
7 cout<<&a[1]<<" "<<&b[1]<<endl;

8 cout<<&a[2]<<" "<<&b[2]<<endl;

9 return 0;

10 }

复制代码



7.对指针的引用

**语法：类型 \*&引用名=指针名;//可以理解为：（类型\*） &引用名=指针名，即将指针的类型当成类型\***

复制代码

1 #include<iostream>

2 using namespace std;

3 int main(){

4 int a=10;

5 int \*ptr=&a;

6 int \*&new\_ptr=ptr;

7 cout<<&ptr<<" "<<&new\_ptr<<endl;

8 return 0;

9 }

复制代码

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824023719027-1561726255.png

## 二、引用的应用

### A.引用作为函数的参数

复制代码

1 #include<iostream>

2 using namespace std;

3 void swap(int &a,int &b){//引用作为函数的参数

4 int temp=a;

5 a=b;

6 b=temp;

7 }

8 int main(){

9 int value1=10,value2=20;

10 cout<<"----------------------交换前----------------------------"<<endl;

11 cout<<"value1的值为："<<value1<<endl;

12 cout<<"value2的值为："<<value2<<endl;

13 swap(value1,value2);

14 cout<<"----------------------交换后----------------------------"<<endl;

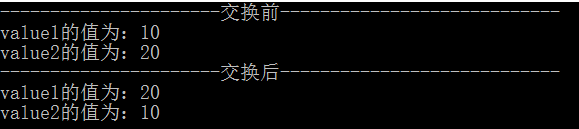
15 cout<<"value1的值为："<<value1<<endl;

16 cout<<"value2的值为："<<value2<<endl;

17 return 0;

18 }

复制代码



**特别注意：**

1.当用引用作为函数的参数时，其效果**和用指针作为函数参数的效果相当**。当调用函数时，函数中的形参就会被当成实参变量或对象的一个别名来使用，也就是说**此时函数中对形参的各种操作实际上是对实参本身进行操作，而非简单的将实参变量或对象的值拷贝给形参**。

2.通常函数调用时，系统采用值传递的方式将实参变量的值传递给函数的形参变量。此时，系统会在内存中开辟空间用来存储形参变量，并将实参变量的值拷贝给形参变量，也就是说形参变量只是实参变量的副本而已；并且如果函数传递的是类的对象，系统还会调用类中的拷贝构造函数来构造形参对象。而**使用引用作为函数的形参时，由于此时形参只是要传递给函数的实参变量或对象的别名而非副本，故系统不会耗费时间来在内存中开辟空间来存储形参**。因此**如果参数传递的数据较大时，建议使用引用作为函数的形参，这样会提高函数的时间效率，并节省内存空间**。

3.使用指针作为函数的形参虽然达到的效果和使用引用一样，但**当调用函数时仍需要为形参指针变量在内存中分配空间**，而引用则不需要这样，故**在C++中推荐使用引用而非指针作为函数的参数**

4.**如果在编程过程中既希望通过让引用作为函数的参数来提高函数的编程效率，又希望保护传递的参数使其在函数中不被改变，则此时应当使用对常量的引用作为函数的参数。**

5.数组的引用作为函数的参数：C++的数组类型是带有长度信息的，**引用传递时如果指明的是数组则必须指定数组的长度**

复制代码

1 #include<iostream>

2 using namespace std;

3 void func(int(&a)[5]){//数组引用作为函数的参数，必须指明数组的长度

4 //函数体

5 }

6 int main(){

7 int number[5]={0,1,2,3,4};

8 func(number);

9 return 0;

10 }

复制代码

## B.常引用

**语法：const 类型 &引用名=目标变量名；**

**常引用不允许通过该引用对其所绑定的变量或对象进行修改**

复制代码

1 #include<iostream>

2 using namespace std;

3 int main(){

4 int a=10;

5 const int &new\_a=a;

6 new\_a=11;//错误！不允许通过常引用对其所绑定的变量或对象进行修改

7 return 0;

8 }

复制代码

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824042102746-132436409.png

**特别注意：**

先看下面的例子

复制代码

1 #include<iostream>

2 #include<string>

3 using namespace std;

4 string func1(){

5 string temp="This is func1";

6 return temp;

7 }

8 void func2(string &str){

9 cout<<str<<endl;

10 }

11 int main(){

12 func2(func1());

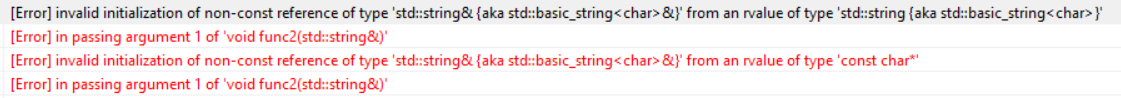
13 func2("Tomwenxing");

14 return 0;

15 }

复制代码

运行上面的程序编译器会报错



这是由于func1()和“Tomwenxing”都会在系统中产生一个临时对象（string对象）来存储它们，而**在C++中所有的临时对象都是const类型的**，而上面的程序试图将const对象赋值给非const对象，这必然会使程序报错。如果在函数func2的参数前添加const，则程序便可正常运行了

复制代码

1 #include<iostream>

2 #include<string>

3 using namespace std;

4 string func1(){

5 string temp="This is func1";

6 return temp;

7 }

8 void func2(**const** string &str){

9 cout<<str<<endl;

10 }

11 int main(){

12 func2(func1());

13 func2("Tomwenxing");

14 return 0;

15 }

复制代码

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824043535808-172238642.png

### C.引用作为函数的返回值

**语法：类型 &函数名（形参列表）{ 函数体 }**

**特别注意：**

1.引用作为函数的返回值时，**必须在定义函数时在函数名前将&**

2.用引用作函数的返回值的**最大的好处是在内存中不产生返回值的副本**

复制代码

1 //代码来源：RUNOOB

2 #include<iostream>

3 using namespace std;

4 float temp;

5 float fn1(float r){

6 temp=r\*r\*3.14;

7 return temp;

8 }

9 float &fn2(float r){ **//&说明返回的是temp的引用，换句话说就是返回temp本身**

10 temp=r\*r\*3.14;

11 return temp;

12 }

13 int main(){

14 float a=fn1(5.0); //**case 1：返回值**

15 //float &b=fn1(5.0); **//case 2:用函数的返回值作为引用的初始化值 [Error] invalid initialization of non-const reference of type 'float&' from an rvalue of type 'float'**

**//（有些编译器可以成功编译该语句，但会给出一个warning）**

16 float c=fn2(5.0);//**case 3：返回引用**

17 float &d=fn2(5.0);//**case 4：用函数返回的引用作为新引用的初始化值**

18 cout<<a<<endl;//78.5

19 //cout<<b<<endl;//78.5

20 cout<<c<<endl;//78.5

21 cout<<d<<endl;//78.5

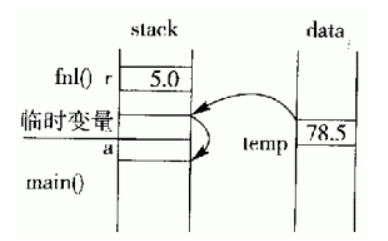
22 return 0;

23 }

复制代码

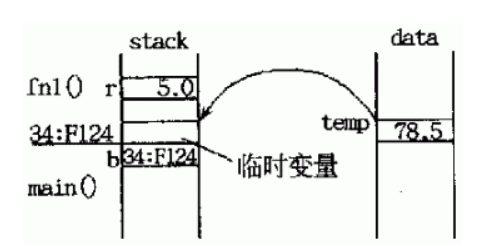
**上例中4个case的说明解释：**

**case 1：用返回值方式调用函数（如下图，图片来源：伯乐在线）：**



**返回全局变量temp的值时，C++会在内存中创建临时变量并将temp的值拷贝给该临时变量。当返回到主函数main后，赋值语句a=fn1(5.0)会把临时变量的值再拷贝给变量a**

**case 2：用函数的返回值初始化引用的方式调用函数（如下图，图片来源：伯乐在线）**



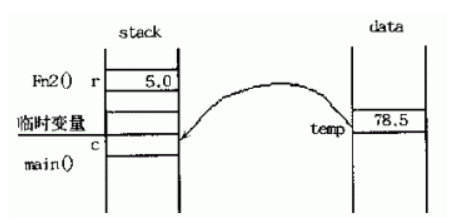
**这种情况下，函数fn1()是以值方式返回到，返回时，首先拷贝temp的值给临时变量。返回到主函数后，用临时变量来初始化引用变量b，使得b成为该临时变量到的别名。由于临时变量的作用域短暂（在C++标准中，临时变量或对象的生命周期在一个完整的语句表达式结束后便宣告结束，也就是在语句float &b=fn1(5.0);之后） ，所以b面临无效的危险，很有可能以后的值是个无法确定的值。**

**如果真的希望用函数的返回值来初始化一个引用，应当先创建一个变量，将函数的返回值赋给这个变量，然后再用该变量来初始化引用：**

1 int x=fn1(5.0);

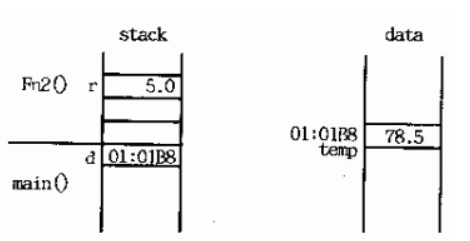
2 int &b=x;

**case 3:用返回引用的方式调用函数（如下图，图片来源：伯乐在线）**



**这种情况下，函数fn2()的返回值不产生副本，而是直接将变量temp返回给主函数，即主函数的赋值语句中的左值是直接从变量temp中拷贝而来（也就是说c只是变量temp的一个拷贝而非别名） ，这样就避免了临时变量的产生。尤其当变量temp是一个用户自定义的类的对象时，这样还避免了调用类中的拷贝构造函数在内存中创建临时对象的过程，提高了程序的时间和空间的使用效率。**

**case 4:用函数返回的引用作为新引用的初始化值的方式来调用函数（如下图，图片来源：伯乐在线）**



**这种情况下，函数fn2()的返回值不产生副本，而是直接将变量temp返回给主函数。在主函数中，一个引用声明d用该返回值初始化，也就是说此时d成为变量temp的别名。由于temp是全局变量，所以在d的有效期内temp始终保持有效，故这种做法是安全的。**

3.**不能返回局部变量的引用。**如上面的例子，如果temp是局部变量，那么它会在函数返回后被销毁，此时对temp的引用就会成为“无所指”的引用，程序会进入未知状态。

4.**不能返回函数内部通过new分配的内存的引用**。虽然不存在局部变量的被动销毁问题，但如果被返回的函数的引用只是作为一个临时变量出现，而没有将其赋值给一个实际的变量，那么就可能造成这个引用所指向的空间（有new分配）无法释放的情况（由于没有具体的变量名，故无法用delete手动释放该内存），从而造成内存泄漏。因此应当避免这种情况的发生

5**当返回类成员的引用时，最好是const引用**。这样可以避免在无意的情况下破坏该类的成员。

6.可以用函数返回的引用作为赋值表达式中的左值

复制代码

1 #include<iostream>

2 using namespace std;

3 int value[10];

4 int error=-1;

5 int &func(int n){

6 if(n>=0&&n<=9)

7 return value[n];//返回的引用所绑定的变量一定是全局变量，不能是函数中定义的局部变量

8 else

9 return error;

10 }

11

12 int main(){

13 func(0)=10;

14 func(4)=12;

15 cout<<value[0]<<endl;

16 cout<<value[4]<<endl;

17 return 0;

18 }

复制代码

https://images2017.cnblogs.com/blog/900715/201708/900715-20170824233843558-1731339574.png

### D.用引用实现多态

在C++中，**引用是除了指针外另一个可以产生多态效果的手段**。也就是说一个基类的引用可以用来绑定其派生类的实例

class Father;**//基类（父类）**

class Son：public Father{.....}**//Son是Father的派生类**

Son son;**//son是类Son的一个实例**

Father &ptr=son;**//用派生类的对象初始化基类对象的使用**

**特别注意：**

**ptr只能用来访问派生类对象中从基类继承下来的成员**。**如果基类（类Father）中定义的有虚函数，那么就可以通过在派生类（类Son）中重写这个虚函数来实现类的多态。**

## 三、总结

1.在引用的使用中，单纯给某个变量去别名是毫无意义的，**引用的目的主要用于在函数参数的传递中，解决大块数据或对象的传递效率和空间不如意的问题**

2.用引用传递函数的参数，能**保证参数在传递的过程中不产生副本**，从而提高传递效率，同时**通过const的使用，还可以保证参数在传递过程中的安全性**

3.引用本身是目标变量或对象的别名，对引用的操作本质上就是对目标变量或对象的操作。因此**能使用引用时尽量使用引用而非指针**

C++对函数的扩展：

Inline内联函数：

内联函数可以替换宏代码片段，使用inline声明内联函数，声明的时候inline关键字必须和定义结合在一起，否则编译器会直接忽略内联请求

Day3：

copy构造函数的调用：

函数返回值是匿名对象的去和留的剖析：

浅拷贝问题：

深拷贝问题：

构造函数的初始化列表：

构造和析构调用顺序：

匿名对象生命周期：

构造中调用构造（产生匿名对象）：

New和delete：

静态成员变量和静态成员函数：

This指针：

全局函数和成员函数：

友元函数：

友元类：

运算符重载：

成员函数和友元函数完成一元运算符重载：

友元函数实现操作符重载知识总结：

重载符号操作符：

绪论：

机器语言：

CPU指令系统，由0，1序列构成的指令码组成。

比如：10000000表示加，10010000表示减

汇编语言：

用助记符号描述的指令系统，如：ADD A,B以上两者是面向机器的语言

程序设计是数据被加工的过程：

客观世界可以分类，对象是类的实例

对象是数据和方法的封装

对象间通过发送和接受消息发生联系

面向对象的语言：

出发点：更加直接的描述客观世界中存在的事物（对象）以及他们之间关系

特点：高级语言

将 客观事物看作具有属性和行为的对象

通过抽象找出同一类对象的共同属性和行为，形成类

通过类的继承与多态实现代码重用

面向对象：

将数据以及对数据的操作方法封装在一起作为一个相互依存，不可分离的整体—对象

对同类型对象抽象出其共性，形成类

类通过一个简单的外部接口，与外界发生关系

对象与对象之间通过消息进行通信

对象具有：静态特征：可以用某数据来描述

动态特征：对象所表现的行为或具有的功能

对象：由一组属性和一组行为构成

属性：用来描述对象静态特征的数据项

行为：用来描述对象动态特性的操作序列

类：

分类是人类通常的思维方式

抽象是分类所依据的原则

抽象：忽略事物的非本质特性，只注意那些与当前目标有关的本质特征，从而找出事物的共性，把具有共同性质的事物划分为一类，得出一个抽象的概念

封装：

把对象的属性和服务结合成一个独立的系统单元

尽可能地隐蔽对象地内部细节，对外形成一个边界（或者说一个屏障），只保留有限地对外接口使之与外部发生关系。

继承：

特殊类地对象拥有其一般类地全部属性与服务，称作特殊类对一般类地继承

多态性：

在一般类中定义地属性或行为，被特殊类继承以后，可以具有不同地数据类型或表现出不同地行为。这使得同一个属性或行为在一般类以及各个特殊类中具有不同地语义

比如：数地加法—实数地加法—复数地加法

注意：对补码再次求补得到地是原码

若运算结果溢出，那么负数之和得正数，或正数之和得负数

关于变量名：

源代码中，变量依靠变量名来标识

目标代码中，变量依靠地址来标识，每个变量地地址互不相同

第三章节 函数

函数的定义和调用

函数的定义

函数是面向对象的程序设计中，对功能的抽象

函数定义的语法形式：

类型标识符/\*如果没有返回值，写void\*/ 函数符 （形式参数表）/\*是被初始化的内部变量，寿命和可见性都仅限在函数的内部）

函数的调用：

调用前先申明函数

如果函数的定义在调用点之前，则无需另外说明

如果函数定义在调用点之前，则需要在调用函数前按下如下形式申明函数原型：

类型标识符 被调用函数名（含类型变量的形参表）

调用形式：函数名（实参列表）

嵌套调用：函数可以嵌套调用，但是函数不可以嵌套定义

在c++中：

Rand

函数原型： int rand(void)

所需要的头文件：<cstdlib>

功能和返回值：求出并返回一个伪随机数

Srand

函数原型：void srand (unsigned int seed);

参数：seed产生随机数的种子

所需头文件：<cstdlib>

功能：为使rand（）产生一系列伪随机整数而设置起始点。使用1作为seed参数，乐意重新初始化rand（）

函数的参数传递机制—传递参数值

在函数被调用时才分配形参的存储单元

实参可以是常量、变量或表达式

实参类型必须与形参相符

传递时是传递参数值，即单向传递

函数的参数传递—用引用做形参

引用（&）是标识符的别名

声明一个引用时，必须同时对它进行初始化，使它指向一个已存在的对象，一旦一个引用被初始化后，就不能改为指向其他对象

引用可以作为形式参数

内联函数的声明与使用

声明时使用关键字inline

编译时在调用处用函数体进行替换，节省了参数擦混敌、控制转移等开销

注意：内联函数体内不能有循环语句和switch语句

内联函数的声明必须出现在内联函数第一次被调用之前

对内联函数不能进行异常接口声明

缺省形参值的作用

函数在声明时可以预先给出缺省的形参值，调用时如给出实参，则采用参数值，否则采用预先给出的缺省形参值

Int add（int x=5，int y=6）

{

Return x+y；

}

int main()

{

Add(10);//10+6

}

缺省形参数的形参必须在形参列表的最后也就是说缺省形参值的右边不能有无缺省值的参数。因为调用时实参与形参的结合是从左向右的顺序

重载函数的声明：

C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成重载。方便使用，便于记忆。

比如：int add(int x,int y)

Float add(float x,float y);

以上两个是形参类型不同

还有的情况是形参个数不同

注：重载函数的形参必须不同：个数不同或者类型不同

编译程序将根据实参和形参的类型以及个数的最佳匹配来选择调用哪一个函数

不要将不同功能的函数声明为重载函数，以免出现调用结果的误解，混淆

标准函数和非标准函数：

标准c++函数

1、c++标准中规定的函数

2、各种编译环境普遍支持，因此使用标准函数的程序移植性好

3、很多标准c++函数继承自标准c，头文件以c开头：cmath cstdlib cstdio ctime

非标准c++函数

1、与特定操作系统或编译环境相关

2、在处理和操作相关事务时常常需要调用

再次提一下栈：

栈是一种容纳数据的容器，数据只能从栈的一段存入（压入栈），数据只能从栈的同一段取出（弹出栈）

运行栈：

运行栈是一段区域的内存空间

运行栈分为一个一个栈帧。每一个栈帧对应一次函数调用

栈帧中包括：

本次函数调用的形参值

控制信息

局部变量值

一些临时数据

函数返回时，会有一个栈帧被压入栈，返回时会有一个栈帧被弹出

第四章节 类与对象

OOP的基本特点：抽象

抽象是对具体对象（问题）进行概括，抽出这一类对象的公共性质并加以描述的过程

先注意问题的本质及描述，其次是实现过程或细节

数据抽象：描述某类对象的属性或状态（对象相互区别的物理量）

代码抽象：描述某类对象的共有的行为特征或具有的功能

抽象的实现：通过类的定义

抽象的实例：钟表

数据抽象：int hour,int minute,int second

代码抽象：setTime(),showTime()

抽象实例—钟表类

Class Clock{

Public:

Void setTime(int newH,int newM,int newS);//这是一个外部接口

Void showTime();//这也是一个外部接口

Private://特定的访问权限

Int hour,minute,second;

};

//这边叫做边界

封装：将抽象出的数据成员、代码成员相结合，将它们视为一个整体

目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只需要通过外部接口，以特定的访问权限，来使用类的成员。

实现封装：类定义中的{}

继承与派生：是c++中支持层次分类的一种机制，允许程序员在保持原有类特性的基础上进行更具体的说明。

实现：定义派生类

多态性：多态：同一名称，不同功能实现方式

目的：达到行为标志统一，减少程序中标识符的个数

实现：重载函数和虚函数

C++中的类：

类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分

利用类可以实现数据的封装、隐藏、集成和派生

利用类易于编写大型复杂的程序，并将其模拟化程度比c中采用函数更高

类的定义形式：类是一种用户自定义类型，它的定义形式为：

Class类名称

{

Public：

公有成员（外部接口）

Private：私有成员

Protected：

保护型成员

}

公有类型成员：

在关键字public后面声明，他们是类与外部的接口，任何外部函数都可以访问共有类型数据和函数

私有类型成员：

在关键字private后面声明，只允许本类中的函数访问，而类外部的函数都不能访问。

如果紧跟在类名称后面声明私有成员，则关键字private可以省略

保护类型：与private类似，其差别表现在继承与派生类的影响不同

类的成员：

class clock{

public:

void setTime(int newH,int newM,int newS);

void showTime();

private:

int hour,minute,second;

};//其中setTime,showTime是成员函数，hour,minute,second是成员数据

成员数据：与一般的变量声明相同，但需要将它放在类的定义体中

成员函数：在类中说明原型，可以在类外给出函数体实现，并在函数名前使用类名加以限定。也可以直接在类中给出函数体，形成内连成员函数。

允许声明重载函数和带默认形参值的函数

内联成员函数：

为了提高运行时的效率，对于比较简单的函数可以声明为内联形式

内联函数体中不要有复杂结构

在类中声明内联成员函数的方式：将函数体放在类的定义中，使用inline关键字

对象：类的对象时该类的某一特定实体，即类类型的变量

定义形式：类名 对象名；

比如：Clock myClock；

类中成员的访问方式：

类中成员互访：直接使用成员名

类外访问：使用“对象名.成员名”方式访问public属性的成员

构造函数：构造函数的作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态

在对象创建时由系统自动调用

如果程序中并未说明，则系统自动产生一个隐含的参数列表为空的构造函数

允许为内联函数，重载函数，带默认形参值的函数

拷贝构造函数：

拷贝构造函数是一种特殊的构造函数，其形参为本类的对象引用。

class 类名{

public:

类名（形参）；//构造函数

类名（类名&对象名）；//拷贝构造函数

}；

类名：：类（类名&对象名）//拷贝构造函数的实现

{函数体}

当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现复制

若函数的形参为类对象，调用函数时。实参赋值给形参，系统自动调用拷贝构造函数

当函数的返回值是类对象时，系统自动调用拷贝构造函数

隐含的老贝构造函数：

如果程序员没有为类声明拷贝初始化构造函数，则编译器自己生成一个隐含的拷贝构造函数

这个构造函数执行的功能是：用作为初始值的对象的每一个数值成员的值，初始化将要建立的对象的对应数据成员

后遭函数：

完成对象被删除前的一些清理工作

在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间

如果程序中未声明析构函数，编译器将自动产生一个隐含的析构函数

组合的概念：

类中的成员数据是另一个类的对象

可以在已有抽象的基础上实现更复杂的抽象

类组合的构造函数设计：

原则：不仅要负责对本类中的基本类型成员赋初值，也要对对象成员初始化。

定义形式：类名：：类名（对象成员所需的形参，本类成员形参）：对象1（参数），对象2（参数）

类组合的构造函数调用：

构造函数调用顺序：先调用内嵌对象的构造函数）按照内嵌时的声明顺序，先声明者先构造）。然后调用本类的构造函数（析构函数的调用顺序相反）

初始化列表中未出现的内嵌现象，用默认构造函数）（即无形参的）初始化

系统自动生成的隐含的默认构造函数中，内嵌对象全部用默认构造函数初始化

不能定义该类的对象，也不能再内联成员函数中使用该类的对象

结构体时一种特殊形态的类：

与类的唯一区别：类的缺省访问权限时private，结构体的缺省访问权限是public

结构体存在的主要原因：与c语言保持兼容

什么时候用结构体不用类：

定义主要用来保存数据而没有什么操作的类型

人们习惯将结构体的数据成员设为公有，因此这个时候使用结构体会更加方便

第五章节：数据的共享与保护

函数原型的作用域，函数原型中的参数，其作用域始于（，结束于）

局部作用域：函数的形参，在块中声明的标识符，其作用域自声明处起，

C++语言中给变量赋值的两个方式：

Int count(0);int count=0;

在c++中，字符串是类的类型，用双引号表示，可以执行字符串的比较，连接。

如果一个函数没有参数列表，那么可以不写，但是括号一定要有

在函数使用地址传递的时候，内部参数地址的改变会影响外部参数的改变，这就是地址传递的特性。

程序的内存区域：

1. 代码区，存放程序的代码，即各个程序中的代码块
2. 全局数据区，存放程序全局变量和静态数据
3. 堆区，存放程序的动态数据
4. 栈区，存放程序的局部变量，即各个程序中的数据

strcpy(dst, src) 将字符从 src拷贝到dst。函数的返回值是dst的地址

strncpy(dst, src, n) 至多从 src 拷贝n个字符到dst。函数的返回值是dst的地址

strcat(dst, src) 将 src 接到 dst 后。函数的返回值是dst的地址

strncat(dst, src, n) 从 src 至多取 n 个字符接到 dst 后。函数的返回值是dst的地址

strlen(s) 返回s的长度

strcmp(s1, s2) 比较 s1 和 s2。如 s1 > s2 返回值为正数，s1=s1返回值为0，s1< s2返回值为负数

strncmp(s1, s2, n) 如 strcmp，但至多比较n个字符 strchr(s, ch) 返回一个指向s中第一次出现ch的地址 strrchr(s, ch) 返回一个指向s中最后一次出现ch的地址 strstr(s1, s2)

返回一个指向s1中第一次出现s2的地址

在c++中的动态分配：

运算符new用于进行内存分配：

P=new type；

P=new type【size】；//分配数组

比如：p=new int【100】；

P=new type（初值）；

运算符delete释放new分配的内存：

Delete p；

Delete 【】 p；

在c++语言中输入一整个字符串（可以包括空格的情况）这么使用：getline（cin，str）；

对象具有动态特性和静态特性：

静态特性：可以用某数据来描述

动态特性：对象所表现的行为或具有的功能

对象：由一组数据和一组行为构成

属性：用来描述对象静态特征的数据项

行为：用来描述对象动态特征的操作序列

类：分类是人类通常的思维方式

抽象是分类所依据的原则

抽象：忽略事情的非本质特性，只注意那些与当前目标有关的本质特征，从而找出事情的共性，把具有共同性质的书屋划分为一类，得出一个抽象的概念

封装：

把对象的属性和服务结合成一个独立的系统单元

尽可能隐蔽对象的内部细节，对外形成一个边界（或者说一个屏障），只保留有限的对外接口使之与外部发生联系

继承：

特殊类的对象拥有其一般类的全部属性与服务，称作特殊类对一般类的继承。

继承是面向对象的方法的一个主要特征。继承是使用已有的类定义作为基础建立新类的定义技术。已有的类可以当作基类来引用，则新类相应的可以作为派生类来引用。广义的说，继承是指能够直接获得已有的性质和特征，而不必重复定义他们。

及城市一种使用户得以在一个类地基础上建立新的类地技术。新的类自动继承旧的类地对象数据属性和行为属性，并可具备某些附加地特征或某些限制。新类称作酒类地子类，旧类称作新类地父类。继承能有效地支持软件构件地重用，使得当需要在系统中增加新的特性的时候，所需要地新的代码最少。并且当继承和多态，动态联编结合使用地时候，为修改系统所需变动地源代码最少。继承机制地强有力地地方还在于它允许程序设计人员可以重用一个班未必完全符合要求地类，允许对该类进行修改而不至于在该类地其他部分引起有害地副作用。

继承分为单继承和多继承。单继承是指一个类只允许有一个父类，即类等级为树形结构。多重继承是指一个类允许有多个父类，多重继承的类可以组合多个父类的性质构成所需要的性质。但是使用多重继承的时候要注意避免二义性。继承性的优点就是相似的对象可以共享程序代码和数据结构，从而大大减少了程序中的冗余信息，提高软件的可重用性，便于软件修改维护

多态性：

在一般类中定义的属性或行为，被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一个属性或行为在一般类及其各个特殊类中具有不同的语义。

例子：数的加法—实数的加法---复数的加法

对于三元运算符这里再强调一下XXX?XXX:XXX如果一开始的表达式为真，那么取引号前面的一项。

第五章节：数据的共享与保护

作用域：在一个地方这个声明的量起作用

类作用域：一般用于特定的成员名

可见性：从对标识符的引用的角度来谈的。具体指的是从内层作用域往外层作用域看的时候能够看到什么。如果标识符在某处可见，则就可以在该处引用此标识符

一般来说：文件的作用域大于类作用域大于块的作用域

在同一作用域内的对象名，函数名，枚举常量名会隐藏同名的类名或枚举类型名

重载的函数可以有相同的函数名