

标准C++编程

C/C++教学体系

“

个人简介
闵卫

minwei@tarena.com.cn”

“

编程基础

”

全程目标

- 背景介绍
- 更好的C
- 第一个C++程序
- 名字空间
- 结构、联合和枚举
- 布尔类型
- 操作符别名
- 重载
- 缺省参数和哑元
- 内联
- 动态内存分配
- 引用
- 显式类型转换
- 来自C++社区的建议



背景介绍

- C++的江湖地位

Position Jul 2013	Position Jul 2012	Delta in Position	Programming Language	Ratings Jul 2013	Delta Jul 2012	Status
1	1	=	C	17.628%	-0.70%	A
2	2	=	Java	15.906%	-0.18%	A
3	3	=	Objective-C	10.248%	+0.91%	A
4	4	=	C++	8.749%	-0.37%	A
5	7	↑↑	PHP	7.186%	+2.17%	A
6	5	↓	C#	6.212%	-0.46%	A
7	6	↓	(Visual) Basic	4.336%	-1.36%	A
8	8	=	Python	4.035%	+0.03%	A
9	9	=	Perl	2.148%	+0.10%	A
10	11	↑	JavaScript	1.844%	+0.39%	A



Mother Tongues

Tracing the roots of computer languages through the ages

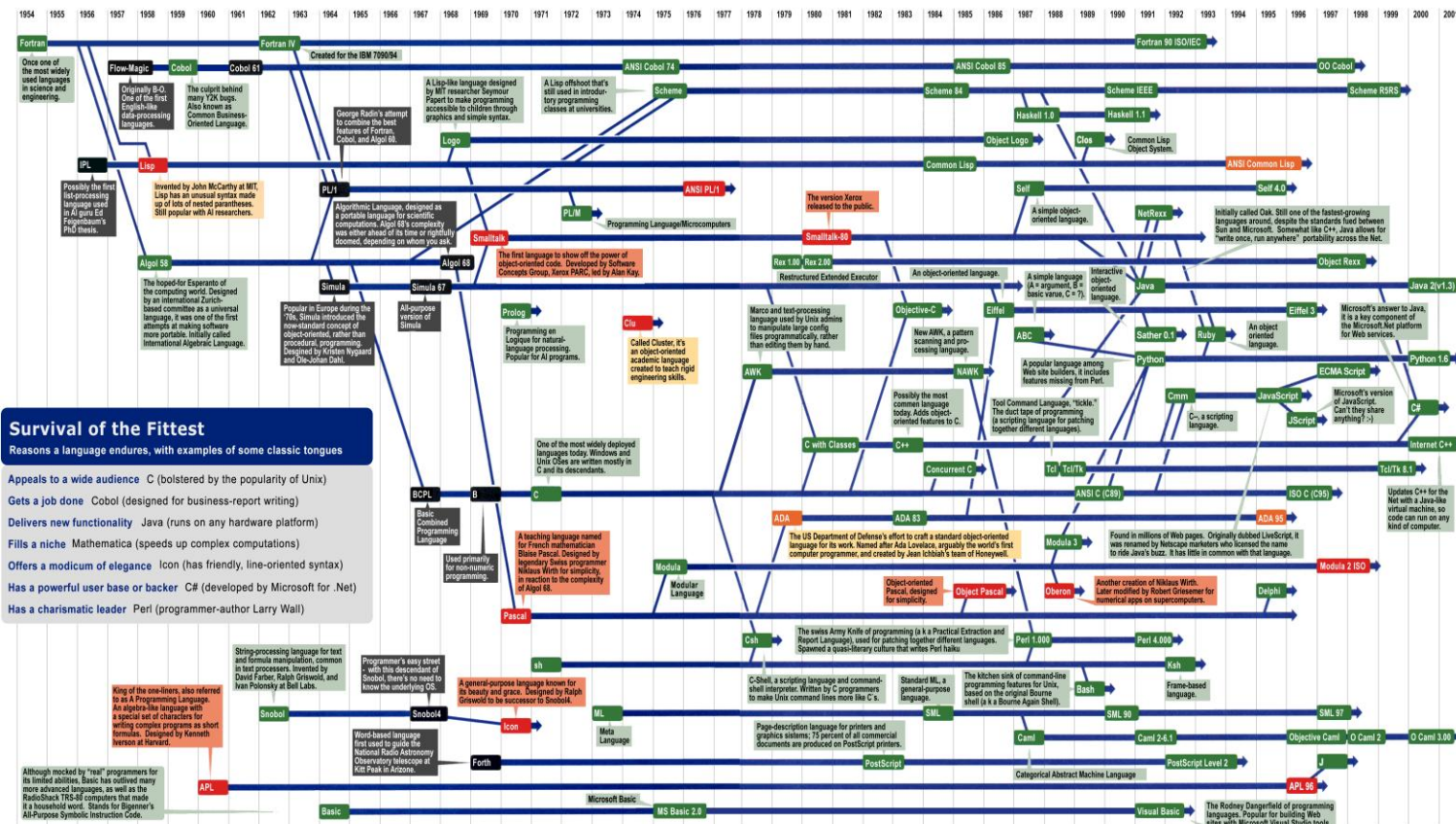
Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouse C/C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will-aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua frangas. Among the most endangered are Ada, APL, B (the predecessor of C), Lsp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at [HTTP://www.informatik.uni-freiburg.de/Java/misc/lang_list.html](http://www.informatik.uni-freiburg.de/Java/misc/lang_list.html). - Michael Mendeno

Key

- 1954 Year introduced
- Active: thousands of users
- Protected: taught at universities; compilers available
- Endangered: using dropping off
- Extinct: no known active users or up-to-date compilers
- Lineage continues



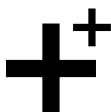
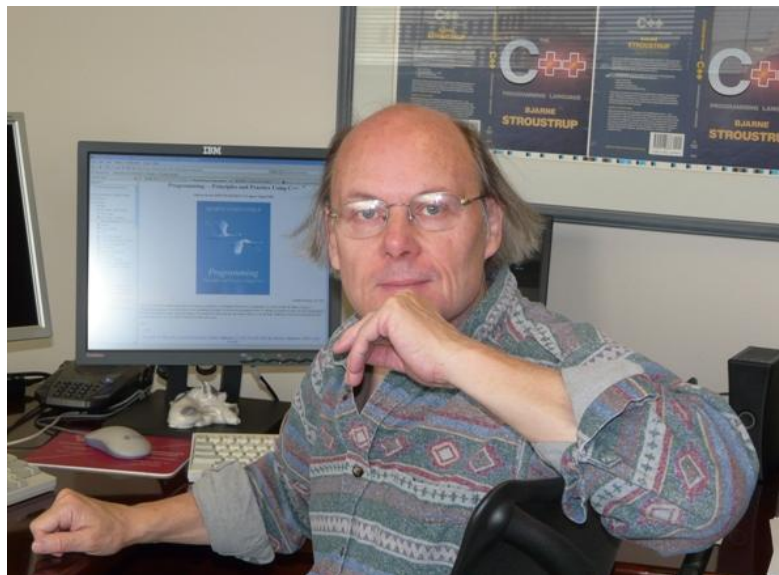
背景介绍

- 致敬与缅怀
 - Ken Thompson (肯·汤普逊),
1943- , **B语言之父**、UNIX发明人之一
 - Dennis M. Ritchie (丹尼斯·里奇),
1941-2011 ,
C语言之父、UNIX之父、黑客之父



背景介绍

- 致敬与缅怀
 - Bjarne Stroustrup (本贾尼·斯特劳斯特卢普), 1950- , C++之父



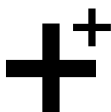
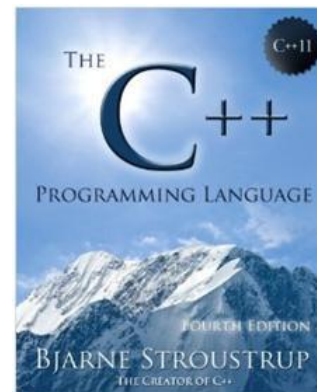
背景介绍

- C++之父的伟大贡献
 - 1979年4月，Bjarne博士试图寻找一种有效的工具，以更加模块化的方法，分析UNIX系统由于内核分布而造成的网络流量
 - 1979年10月，Bjarne博士完成了一个预处理程序，Cpre，为C增加了类似Simula的类机制
 - 1983年，Bjarne博士开发了一种全新的编程语言，C with Classes，即后来的C++
 - C++在运行时间、代码紧凑性和数据紧凑性方面，完全可以和C语言相媲美



背景介绍

- C++之父的伟大贡献
 - C++吸收了很多其它语言的精华
 - ✓ Simula的类
 - ✓ Algol68的运算符重载和引用
 - ✓ BCPL的 “//” 注释
 - ✓ Ada的模板和名字空间
 - ✓ Clu和ML的异常
 - 1985年，CFront 1.0发布，Bjarne博士推出经典巨著《The C++ Programming Language》第一版



背景介绍

- C++大事记
 - 1979 : Bjarne博士在贝尔实验室完成了Cpre
 - 1983 : 第一个C with Classes实现投入使用
 - 1985 : CFront 1.0发布
 - 1987 : GNU C++发布
 - 1990 : Borland C++发布
 - 1992 : Microsoft C++发布 , IBM C++发布
 - 1998 : ISO标准被批准 , 即C++98
 - 2003 : ISO对C++98进行修订 , 即C++03
 - 2011 : ISO发布ISO/IEC 15882:2011 , 即C++11



背景介绍

- 无处不在的C++

- 游戏

C++在性能和效率方面的突出表现，是一个很重要的原因

- 科学计算

曾经被FORTRAN一统天下，但近年来，C++凭借先进的数值计算库、泛型编程等优势在这一领域也应用颇多

- 网络和分布式应用

C++拥有很多成熟的用于网络通信的库，其中最具有代表性的是跨平台的、重量级的ACE库，该库可以说是C++语言最重要的成果之一，在许多重要的企业、部门甚至军方项目中都有应用



背景介绍

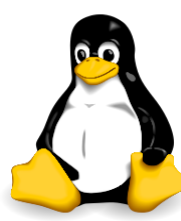
- 无处不在的C++

- 操作系统和设备驱动

在该领域，C语言依然是主要使用的编程语言，但是C++凭借其对于C良好的兼容和面向对象特性，也开始在该领域崭露头角

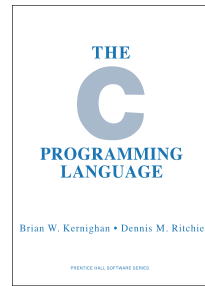
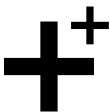
- 移动终端、嵌入式系统、编程语言、行业应用

C++、Java和C#这3种语言中，C++是最早出现的，保持了对C的高度兼容，允许程序员显式地使用指针，进而高效地管理和使用内存，尽管这也是最容易导致问题的地方



背景介绍

- C++的竞争对手
 - 一方面，在企业级系统(数据密集，业务规则复杂多变)开发中，C++正在遭受Java、C#等语言的围剿
 - 另一方面，在系统编程和嵌入式等更接近硬件的领域，C++又遭到C语言的强烈狙击
 - 总之，C++是目前唯一同时具备高性能和良好抽象建模能力的编程语言



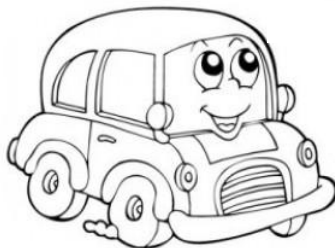
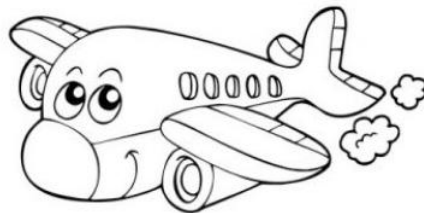
背景介绍

- C++还是++C？
 - 后++表达式的值是操作数自增以前的值！
 - C++程序员们是否还在以C的方式使用C++呢？



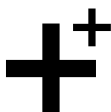
更好的C

- C++和C一样，都属于**编译型**语言
- C++和C一样，都属于**强类型**语言
- C++对C完全兼容，对其做了优化并提供更多特性
 - 语言风格更加**简洁**
 - 类型检查更加**严格**
 - 支持**面向对象**编程
 - 支持**操作符重载**
 - 支持**异常**处理
 - 支持**泛型**编程



练习时间

第一个C++程序：hello.cpp
体验C++程序和C程序的差别



第一个C++程序

- 编译器：**g++**
 - 也可以用gcc，但要加上-lstdc++
- 扩展名：**.cpp/.cc/.C/.cxx**
 - 也可以用.c，但要加上-x c++
- 头文件：**#include <iostream>**
 - 也可以用#include <cstdio>
- 流操作：**cout<</cin>>**
 - 也可以用scanf/printf
- 所有标准类型、对象和函数都位于**std**命名空间中



名字空间

- 为什么需要名字空间——**WHY** ?
 - 划分逻辑**单元**
 - 避免名字**冲突**



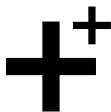
名字空间

- 什么是名字空间——WHAT ?
 - 名字空间定义
 - 名字空间合并
 - 声明定义分开
- 怎样用名字空间——HOW ?
 - 作用域限定符
 - 名字空间指令
 - 名字空间声明



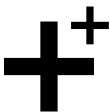
名字空间

- 无名名字空间
 - 不属于任何有名名字空间的标识符，隶属于**无名**命名空间
 - 无名命名空间的成员，**直接**通过 “::” 访问
 - 名字空间嵌套与名字空间别名
 - 内层标识符**隐藏**外层同名标识符
 - 嵌套的名字空间需要**逐层**分解
 - 可通过名字空间**别名**简化书写
- namespace ns_four = ns1::ns2::ns3::ns4;



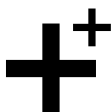
结构、联合和枚举

- C++的结构
 - 声明或定义结构型变量，可以省略**struct**关键字
 - 可以定义**成员函数**，在结构体的成员函数中可以直接访问该结构体的成员变量，无需通过 “.” 或 “->”
- C++的联合
 - 声明或定义联合型变量，可以省略**union**关键字
 - 支持**匿名**联合
- C++的枚举
 - 声明或定义枚举型变量，可以省略**enum**关键字
 - **独立**的类型，和整型之间不能隐式相互转换



布尔类型

- 表示布尔量的数据类型
 - **bool**
- 布尔类型的字面值常量
 - **true**表示真
 - **false**表示假
- 布尔类型的本质
 - 单字节**整数**，用**1**和**0**表示真和假
- 任何基本类型都可以被**隐式**转换为布尔类型
 - 非0即真，0即假



操作符别名

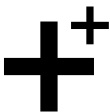
主名	别名	主名	别名	主名	别名
{	<%	&&	and		bitor
}	%>		or	=	or_eq
[<:	!	not	^	xor
]	:>	!=	not_eq	^=	xor_eq
#	%:	&	bitand	~	compl
##	%:::	&=	and_eq		

- 某些欧洲国家语言使用的字母比26个基本拉丁字母多，占用了ASCII码表及键盘中&等特殊符号的位置，因此规定了一些操作符的别名，以便使用这些国家的键盘也能正确输入C++源代码



重载

- 重载关系
 - 同一作用域中，函数名相同，参数表不同的函数
 - 只有同一作用域中的同名函数才涉及重载问题，不同作用域中同名函数遵循标识符隐藏原则
- 重载解析
 - 完全匹配>常量转换>升级转换>标准转换>自定义转换>省略号匹配
- 函数指针的类型决定其匹配的重载版本



重载

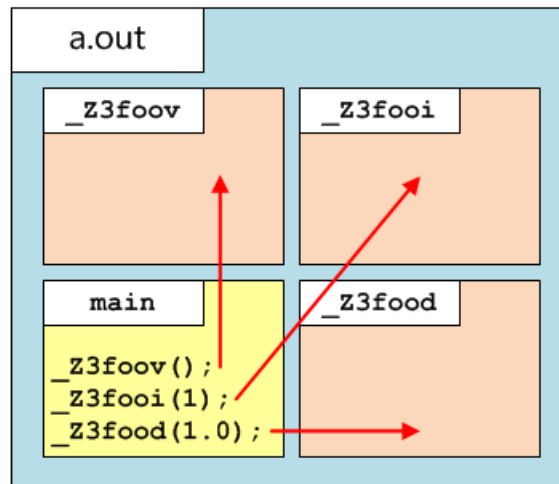
- 重载是通过C++**换名**实现的
- 通过**extern "C"**可以要求C++编译器按照C方式处理函数接口，即不做换名，当然也就无法重载

```

overload.cpp

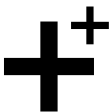
void foo (void) { ... }
void foo (int i) { ... }
void foo (double d) { ... }

int main (void) {
    foo ();
    foo (1);
    foo (1.0);
    return 0;
}
    
```



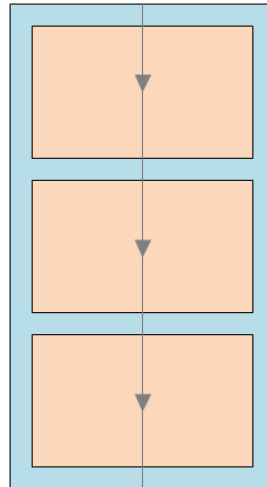
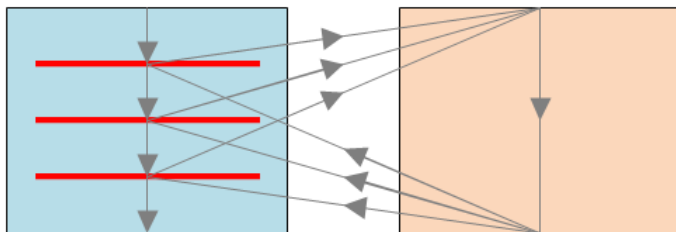
缺省参数和哑元

- 可以为函数的参数指定**缺省值**，调用该函数时若未指定实参，则与该实参相对应的形参取缺省值
- 函数参数的缺省值只能在函数**声明**中指定
- 如果函数的某一个参数具有缺省值，那么该参数**后面的所有**参数必须都具有缺省值
- 不要因为使用缺省参数而导致**重载歧义**
- 只指定类型而不指定名称的函数参数，谓之**哑元**
 - 保证函数的向下兼容
 - 形成函数的重载版本



内联

- 内联就是用函数已被编译好的二进制代码，**替换**对该函数的调用指令
- 内联在保证函数特性的同时，避免了函数**调用开销**
- 注意内联与**有参宏**(宏函数)的区别



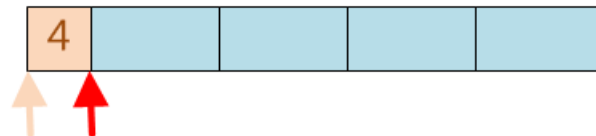
内联

- 内联会使可执行文件的**体积**和进程代码的**内存**变大，因此只有**频繁**调用的**简单**函数才适合内联
- 若函数在类声明中**直接定义**，则自动被优化为**内联**，否则可在其声明处加上**inline**关键字
- **inline**关键字仅表示**期望**该函数被优化为内联，但是否适合内联则完全由编译器决定
- **稀少**被调用的**复杂**函数和**递归**函数都不适合内联



动态内存分配

- 继续使用标准C库函数malloc/calloc/realloc/free
- 使用new/delete操作符在堆中分配/释放内存
 - `int* pi = new int;`
`delete pi;`
- 在分配内存的同时初始化
 - `int* pi = new int (100);`
- 以数组方式new的也要以数组方式delete
 - `int* pi = new int [4] {1, 2};`
`delete[] pi;`



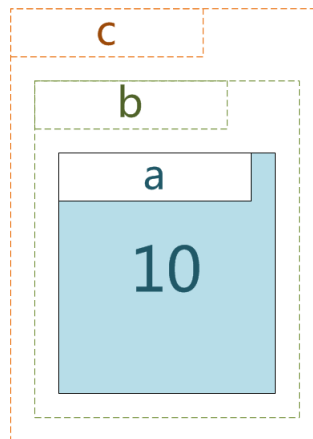
动态内存分配

- 通过new操作符分配N维数组，返回N-1维数组指针
 - `int (*prow) [4] = new int [3][4];`
 - `int (*ppage) [4][5] = new int [3][4][5];`
- 不能通过delete操作符释放已释放过的内存
- delete野指针后果未定义，delete空指针安全
- 内存分配失败，new操作符抛出bad_alloc异常
- 定位分配
 - `new (指针) 类型 (初值);`
 - 在一个已分配的内存空间中创建对象



引用

- 引用即**别名**
 - `int a = 10;`
`int& b = a;`
`int& c = b;`
- 引用必须**初始化**
 - `int& b; // ERROR !`
`int& b = a;`
`const int& b = 10;`
- 引用不能为**空**
- 引用不能更换**目标**



引用

- 引用型参数，函数的形参是实参的别名
 - 在函数中修改实参值
 - 避免对象复制的开销
- 常引用型参数
 - 接受常量型实参
 - 防止对实参的意外修改

```
fun(actual);
```

actual

10

```
void fun(int& formal){
```

```
...
```

formal

```
}
```



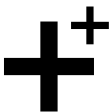
引用

- 引用型返回值，从函数中返回引用，一定要保证在函数返回以后，该引用的目标依然有效
 - 返回左值
 - 可以返回全局、静态乃至成员变量的引用
 - 可以返回在堆中动态创建的对象引用
 - 可以返回调用对象自身的引用
 - 可以返回引用型参数本身
 - 不能返回局部变量的引用
- 常引用型返回值
 - 返回右值



引用

- 在实现层面，引用就是指针，但在语言层面，引用不是**实体**类型，因此与指针存在明显的差别
 - 指针可以不初始化，其目标可在初始化后随意变更（除非是指针常量），而引用必须**初始化**，且一旦初始化就**无法变更**其目标
 - 存在空指针，不存在**空引用**
 - 存在指向指针的指针，不存在**引用引用的引用**
 - 存在引用指针的引用，不存在**指向引用的指针**
 - 存在指针数组，不存在**引用数组**，但存在**数组引用**



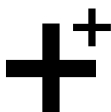
显式类型转换

- C风格的显式类型转换
 - (目标类型)源类型变量
- C++风格的显式类型转换
 - 目标类型(源类型变量)
- 静态类型转换
 - `static_cast<目标类型>` (源类型变量)
 - 隐式类型转换的逆转换
 - 自定义类型转换



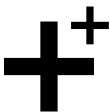
显式类型转换

- 动态类型转换
 - `dynamic_cast<目标类型>` (源类型变量)
 - 多态父子类指针或引用之间的转换
- 常类型转换
 - `const_cast<目标类型>` (源类型变量)
 - 去除指针或引用上的const属性
- 重解释类型转换
 - `reinterpret_cast<目标类型>` (源类型变量)
 - 任意类型的指针或引用之间的转换
 - 任意类型的指针和整型之间的转换



来自C++社区的建议

- 用`const`取代常量宏，用`enum`取代唯一标识宏，用`inline`取代参数宏，用`namespace`取代条件编译解决名字冲突
- 变量随用随声明，并立即初始化
- 少用`malloc/free`，`new/delete`更好
- 避免使用`void*`、指针算术、联合和强制类型转换
- 用`string`和STL容器取代低级数组
- 树立面向对象的编程思想，程序设计的过程是用类描述世界的过程，而非用函数处理数据的过程



“

类和对象

”

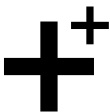
全程目标

- 面向对象
- 类和对象
- 类的定义与实例化
- 构造函数与初始化表
- this指针
- 常函数与常对象
- 析构函数
- 拷贝构造与拷贝赋值
- 静态成员
- 成员指针



面向对象

- 为什么要面向对象——**WHY** ?
 - 相比于分而治之的结构化程序设计，强调大处着眼的面向对象程序设计思想，更适合于开发**大型软件**
 - 得益于数据抽象、代码复用等面向对象的固有特征，软件开发的**效率**获得极大地提升，**成本**却大幅降低
 - 面向对象技术在数据库、网络通信、图形界面等领域的广泛应用，已催生出各种**设计模式**和**应用框架**
 - 面向对象技术的表现如此出众，以至于那些原本并不直接支持面向对象特性的语言(例如C)，也在越来越多地通过各种方法**模拟**一些面向对象的软件结构



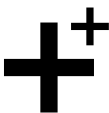
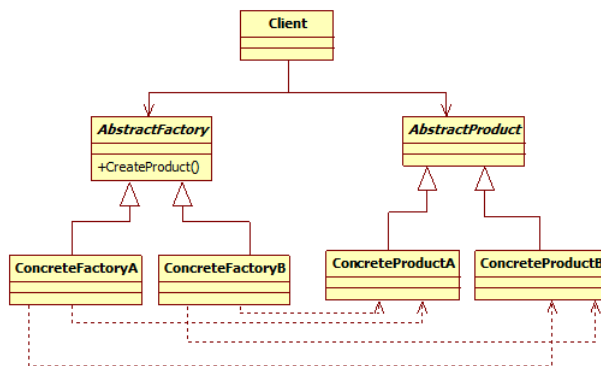
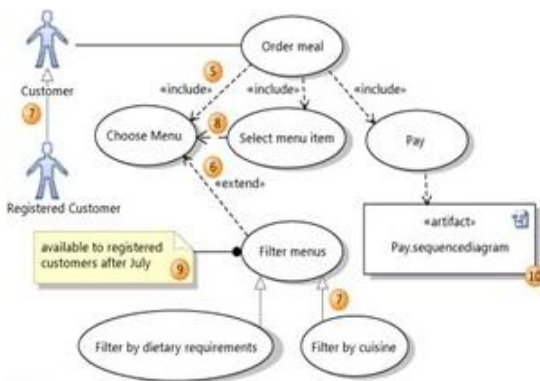
面向对象

- 什么是面向对象——WHAT？
 - 万物皆对象，这是人类面对世界最朴素，最自然的感觉、想法和观点
 - 把大型软件看成是一个由对象组成的社会
 - 对象拥有足够的智能，能够理解来自其它对象的信息，并以适当的行为作出反应
 - 对象能够从高层对象继承属性和行为，并允许低层对象从自己继承属性和行为等
 - 编写程序的过程就是描述对象属性和行为的过程，凭借这种能力使问题域和解域获得最大程度的统一
 - 面向对象的三大要件：封装、继承和多态



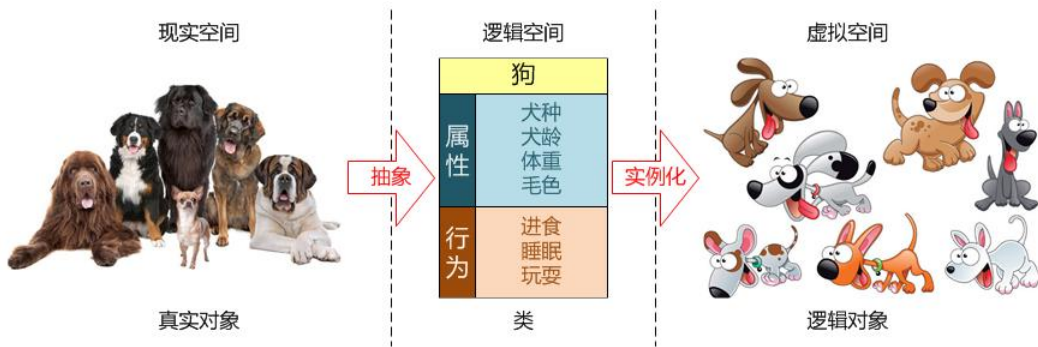
面向对象

- 怎样面向对象——**HOW**？
 - 至少掌握一种面向对象的**程序设计语言**，如C++
 - 深入理解**封装、继承和多态**等面向对象的重要概念
 - 精通一种**元语言**，如UML，在概念层次上描述设计
 - 学习**设计模式**，源自多年成功经验的积累和总结



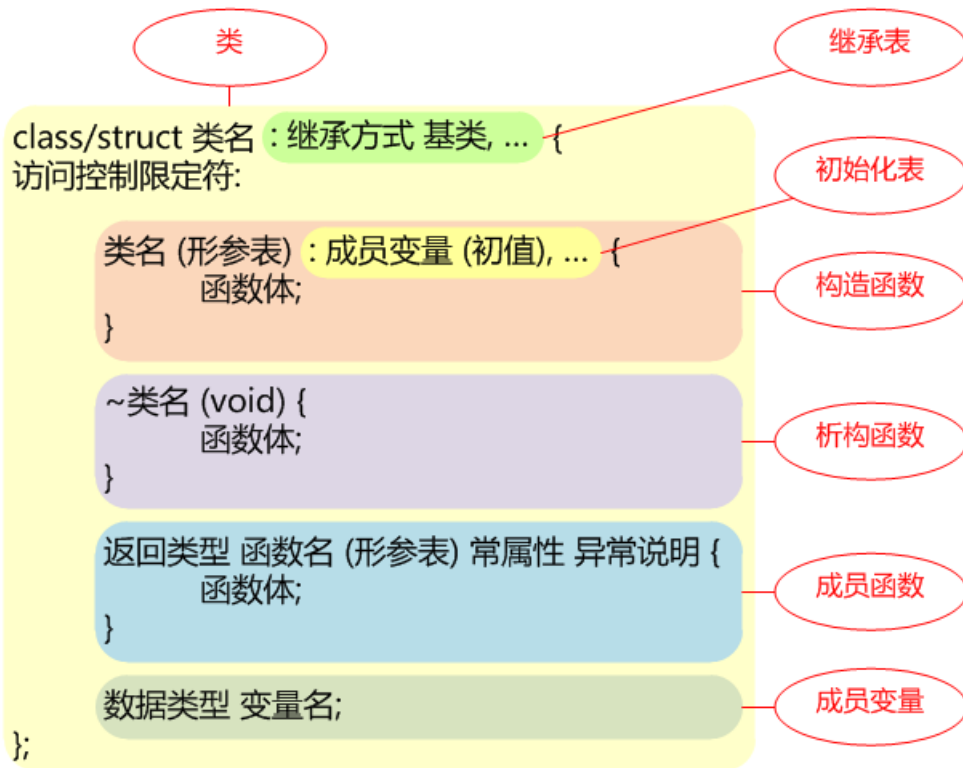
类和对象

- 拥有相同**属性**和**行为**的对象被分成一组，即一个类
- 类可用于表达那些不能直接与内置类型建立自然映射关系的**逻辑抽象**
- 类是一种用户自定义的复合数据类型，即包括表达属性的**成员变量**，也包括表达行为的**成员函数**
- 类是现实世界的**抽象**，对象是类在虚拟世界的**实例**



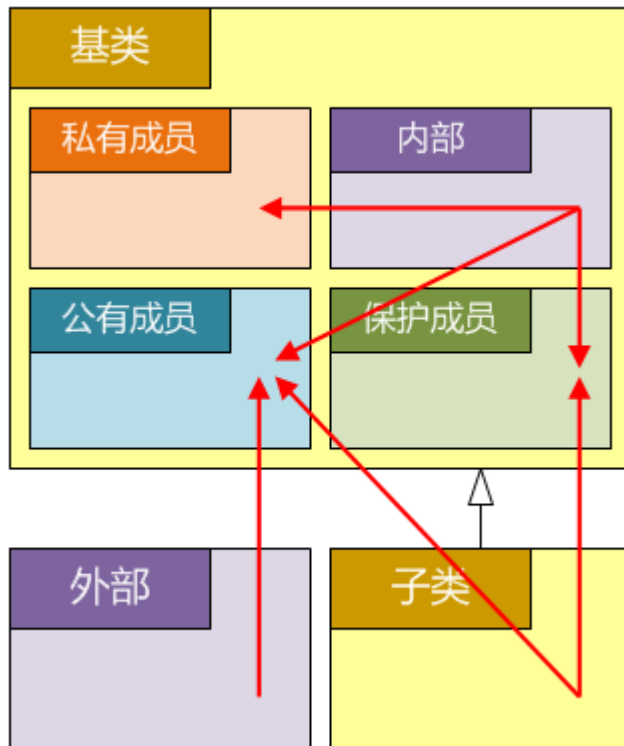
类的定义与实例化

- 类的一般形式



类的定义与实例化

- 访问控制限定符
 - public
公有成员
谁都可以访问
 - protected
保护成员
只有自己和子类可以访问
 - private
私有成员
只有自己可以访问



类的定义与实例化

- 访问控制限定符
 - 在C++中，类(class)和结构(struct)已没有本质性的差别，唯一的不同在于
 - ✓ 类的缺省访问控制属性为私有(private)
 - ✓ 结构的缺省访问控制属性为公有(public)
 - 访问控制限定符仅作用于类，而非作用于对象，因此同一个类的不同对象，可以互相访问非公有部分
 - 对不同成员的访问控制属性加以区分，体现了C++作为面向对象程序设计语言的封装特性



类的定义与实例化

- 构造函数
 - 函数名与**类名**相同，且**没有返回类型**
 - 在创建对象时**自动**被调用，且仅被调用**一次**
 - ✓ 对象定义语句
 - ✓ new操作符
 - 为成员变量赋初值，分配资源，设置对象的**初始状态**
 - 对象的创建过程
 - ✓ 为整个对象分配**内存**空间
 - ✓ 以构造实参调用**构造**函数
 - 构造**基类**部分
 - 构造**成员**变量
 - 执行构造**代码**



类的定义与实例化

- 类的声明与实现可以分开

```
class 类名 {
    返回类型 函数名 (形参表);
};
```

```
返回类型 类名::函数名 (形参表) {
    函数体;
}
```



类的定义与实例化

- 对象的创建与销毁
 - 在栈中创建单个对象
 类名 对象; // 注意不要加空括号
 类名 对象 (实参表);
 - 在栈中创建对象数组
 类名 对象数组[元素个数];
 类名 对象数组[元素个数] = {类名 (实参表), ...};
 类名 对象数组[] = {类名 (实参表), ...};



类的定义与实例化

- 对象的创建与销毁

- 在堆中创建/销毁单个对象

类名* 对象指针 = new 类名;

类名* 对象指针 = new 类名 ();

类名* 对象指针 = new 类名 (实参表);

delete 对象指针;

- 在堆中创建/销毁对象数组

类名* 对象数组指针 = new 类名[元素个数];

类名* 对象数组指针 = new 类名[元素个数] {类名 (实参表), ...};

// 上面的写法需要编译器支持C++11标准

delete[] 对象数组指针;



类的定义与实例化

- 将类的声明、实现与使用分别放在不同的文件里

user.h

```
class User {
public:
    User (string name, int age);
    void print (void);
private:
    string m_name;
    int m_age;
};
```

声明类

user.cpp

```
#include "user.h"
User::User (string name, int age) :
    m_name (name), m_age (age) {}
void User::print (void) {
    cout << m_name << ", "
        << m_age << endl;
}
```

实现类

main.cpp

```
#include "user.h"
int main (void) {
    User user ("Zaphod", 49);
    user.print ();
    return 0;
}
```

使用类

使用



声明



实现



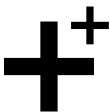
练习时间

实现一个电子时钟类，其构造函数接受当前系统时间，以秒为单位持续运行



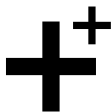
构造函数与初始化表

- 构造函数可以重载
 - 构造函数也可以通过参数表的差别化形成重载
 - 重载的构造函数通过构造实参的类型选择匹配
 - 不同的构造函数版本表示不同的对象创建方式
 - 使用缺省参数可以减少构造函数重载版本数量
 - 某些构造函数具体特殊的意义
 - ✓ 缺省构造函数：按缺省方式构造
 - ✓ 类型转换构造函数：从不同类型的对象构造
 - ✓ 拷贝构造函数：从相同类型的对象构造



构造函数与初始化表

- 缺省构造函数
 - 缺省构造函数亦称**无参**构造函数，但其未必真的没有任何参数，为一个有参构造函数的每个参数都提供一个**缺省值**，同样可以达到无参构造函数的效果
 - 如果一个类**没有**定义任何构造函数，那么编译器会为其提供一个**缺省**构造函数
 - ✓ 对基本类型的成员变量，**不做**初始化
 - ✓ 对类类型的成员变量和基类子对象，调用相应类型的**缺省构造**函数初始化
 - 对于已经定义至少一个构造函数的类，无论其构造函数是否带有参数，编译器都**不会**为其提供缺省构造函数



构造函数与初始化表

- 缺省构造函数
 - 有时必须自己定义缺省构造函数，即使它什么也不做，尤其是在使用数组或容器的时候，某些基于早期C++标准的编译器不支持对象数组的初始化语法
 - 有时必须为一个类提供缺省构造函数，仅仅因为它可能作为另一个类的子对象而被缺省构造
 - 若子对象不宜缺省构造，则需要为父对象提供缺省构造函数，并在其中显式地以非缺省方式构造该子对象



构造函数与初始化表

- 类型转换构造函数
 - 在目标类型中，可以接受单个源类型对象实参的构造函数，支持从源类型到目标类型的隐式类型转换


```
class 目标类型 {
    目标类型 (const 源类型& src) { ... }
};
```
 - 通过explicit关键字，可以强制这种通过构造函数实现的类型转换必须显式地进行


```
class 目标类型 {
    explicit 目标类型 (const 源类型& src) { ... }
};
```



构造函数与初始化表

- 拷贝构造函数

- 形如

```
class 类名 {
    类名 (const 类名& that) { ... }
};
```

的构造函数被称为拷贝构造函数，用于从一个已定义的对象构造其同类型的副本，即**对象克隆**

- 如果一个类**没有**定义拷贝构造函数，那么编译器会为其提供一个**缺省**拷贝构造函数

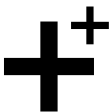
- ✓ 对基本类型成员变量，按**字节**复制

- ✓ 对类类型成员变量和基类子对象，调用相应类型的**拷贝构造函数**



构造函数与初始化表

- 拷贝构造函数
 - 如果自己定义了拷贝构造函数，编译器将不再提供缺省拷贝构造函数，这时所有与成员复制有关的操作，都必须在自定义拷贝构造函数中编写代码完成
 - 若缺省拷贝构造函数不能满足要求，则需自己定义
 - 拷贝构造的时机
 - ✓ 用已定义对象作为同类型对象的构造实参
 - ✓ 以对象的形式向函数传递参数
 - ✓ 从函数中返回对象
 - ✓ 某些拷贝构造过程会因编译优化而被省略



构造函数与初始化表

- 自定义构造函数和系统定义构造函数

自定义构造函数	系统定义构造函数
无	缺省构造函数 缺省拷贝构造函数
除拷贝构造函数以外的 任何构造函数	缺省拷贝构造函数
拷贝构造函数	无

所有系统定义的构造函数，其访问控制属性均为公有(public)



构造函数与初始化表

- 初始化表
 - 通过在类的构造函数中使用初始化表，指明该类的成员变量如何被初始化
 - 数组和结构型成员变量需要用花括号 “{}” 初始化
 - 类的类类型成员变量和基类子对象，必须在初始化表中显式初始化，否则将调动相应类型的缺省构造函数初始化
 - 类的常量型和引用型成员变量，必须在初始化表中显式初始化
 - 类的成员变量按其在类中的声明顺序依次被初始化，而与其在初始化表中的顺序无关



练习时间

将之前的电子时钟类，改为用初始化表初始化，并提供另一种初始化方式，使之兼具计时器的功能



this指针

- C++对象模型
 - 相同类型的不同对象各自拥有**独立的**成员变量实例
 - 相同类型的不同对象彼此共享**同一份**成员函数代码
 - 在代码区中，为相同类型的不同对象所**共享**的成员函数，如何区分所访问的成员变量隶属于哪个**对象**
 - 类的每个成员函数、构造函数和析构函数，都有一个隐藏的指针型参数**this**，指向**调用**该成员函数、正在被**构造**或正在被**析构**的对象，这就是this指针
 - 在类的成员函数、构造函数和析构函数中，对所有成员的访问，都是通过**this指针**进行的





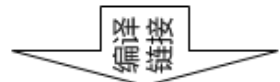
编辑

```
class User {
public:
    User (string name, int age) {
        m_name = name;
        m_age = age;
    }
    void print (void) {
        cout << m_name << ", "
            << m_age << endl;
    }
private:
    string m_name;
    int m_age;
};

int main (void) {
    ...
}
```

源代码

类



00401184	push	edx
00401185	call	edi
00401187	lea	eax,[esp+0Ch]
0040118B	push	eax
0040118C	call	ebx
0040118E	push	0
00401190	push	0
00401192	push	0
00401194	lea	ecx,[esp+18h]
00401198	push	ecx
00401199	call	esi
0040119B	test	eax, eax
0040119F	pop	ebx
004011A0	mov	eax, dword ptr
004011A4	pop	esi
004011A5	pop	edi
004011A6	add	esp, 1Ch
004011A9	ret	10h

可执行程序

编译

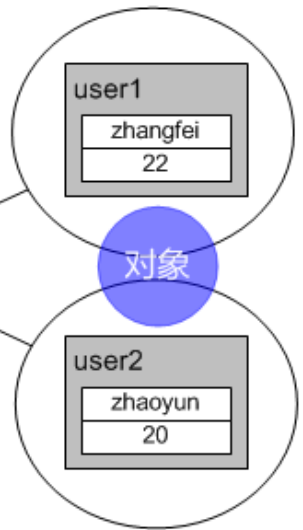
进程空间

高地址



低地址

运行




```
class User {
public:
    ...
    void print (void) {
        cout << m_name << ", "
            << m_age << endl;
    }
    ...
};

int main (void) {
    User user1 (...), user2 (...);
    user1.print ();
    user2.print ();
}
```

```
void _ZN4User5printEv (User* this) {
    cout << this->m_name << ", "
        << this->m_age << endl;
}

int main (void) {
    User user1 (...), user2 (...);
    _ZN4User5printEv (&user1);
    _ZN4User5printEv (&user2);
}
```



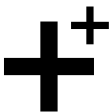
this指针

- this指针的应用
 - 多数情况下，程序并**不需要**显式地使用this指针
 - 有时为了方便，将一个类的某个成员变量与该类构造函数的相应参数取**相同标识符**，这时在构造函数内部，可通过this指针将二者加以区分
 - 返回基于this指针的自引用，以支持**串连调用**
 - 将this指针作为函数的参数，以实现**对象交互**



常函数与常对象

- 在类成员函数的形参表之后，函数体之前加上 `const` 关键字，该成员函数的 `this` 指针即具有常属性，这样的成员函数被称为常函数
 - `class` 类名 {
 返回类型 函数名 (形参表) `const` {
 函数体;
 }
};
- 在常函数内部无法修改成员变量的值，除非该成员变量被 `mutable` 关键字修饰



```
class User {
public:
    ...
    void print (void) const {
        cout << m_name << ", "
            << m_age << endl;
    }
    ...
};

int main (void) {
    User user1 (...), user2 (...);
    user1.print ();
    user2.print ();
}
```

```
void _ZNK4User5printEv (const User* this) {
    cout << this->m_name << ", "
        << this->m_age << endl;
}

int main (void) {
    User user1 (...), user2 (...);
    _ZN4User5printEv (&user1);
    _ZN4User5printEv (&user2);
}
```



常函数与常对象

- 被`const`关键字修饰的对象、对象指针或对象引用，统称为常对象
 - `const User user (...);`
 - `const User* cptr = &user;`
 - `const User& cref = user;`
- 通过常对象只能调用常函数，通过非常对象既可以调用常函数，也可以调用非常函数
- 原型相同的成员函数，常版本和非常版本构成重载
 - 常对象只能选择常版本
 - 非常对象优先选择非常版本，如果没有非常版本，也能选择常版本



常函数与常对象

```
class Integer {  
public:  
    Integer (const int& val = 0) : m_val (val) {}  
    int& value (void) {  
        return m_val;  
    }  
    const int& value (void) const {  
        return m_val;  
    }  
private:  
    int m_val;  
};
```

```
int main (void) {  
    Integer i (100);  
    i.value () = 200;  
    const Integer& cr = i;  
    cout << cr.value () << endl;  
}
```



练习时间

实现一个最多保存100个整型数的容器类，
通过`at(size_t index)`成员函数接受从0开始
的下标，要求根据容器的常属性返回其中元
素的左值或右值



析构函数

- 析构函数的函数名就是在**类名**前面加 “~” ，**没有返回类型也没有参数**，不能重载
- 在销毁对象时**自动**被调用，且仅被调用**一次**
 - 对象离开作用域
 - delete操作符
- **释放**在对象的构造过程或生命周期内所获得的**资源**
- 其功能并不局限在释放资源上，它可以执行任何类的设计者希望在**最后一次**使用对象之后执行的动作
- 通常情况下，若对象在其生命周期的最终时刻，**并不持有**任何动态分配的资源，可以**不定义**析构函数



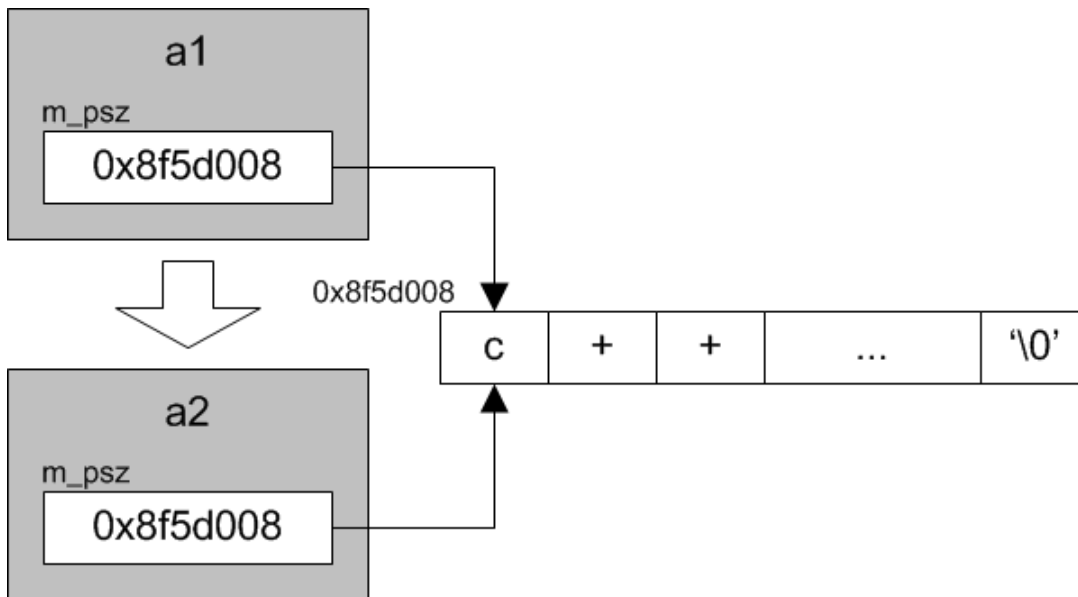
析构函数

- 如果一个类**没有**定义析构函数，那么编译器会为其提供一个**缺省**析构函数
 - 对基本类型的成员变量，什么也**不做**
 - 对类类型的成员变量和基类子对象，调用相应类型的**析构**函数
- 对象的销毁过程
 - 调用**析构**函数
 - ✓ 执行析构**代码**
 - ✓ 析构**成员**变量
 - ✓ 析构**基类**部分
 - 释放整个对象所占用的**内存**空间



拷贝构造与拷贝赋值

- 缺省方式的拷贝构造和拷贝赋值，对包括指针在内的基本类型成员变量按字节复制，导致浅拷贝问题



拷贝构造与拷贝赋值

- 为了获得完整意义上的对象副本，必须自己定义拷贝构造和拷贝赋值，针对指针型成员变量做**深拷贝**



拷贝构造与拷贝赋值

- 相对于拷贝构造，**拷贝赋值**需要做更多的工作
 - 避免自赋值
 - 分配新资源
 - 拷贝新内容
 - 释放旧资源
 - 返回自引用
- 尽量**复用**拷贝构造函数和析构函数中的代码
 - 拷贝构造：分配新资源、拷贝新内容
 - 析构函数：释放旧资源



拷贝构造与拷贝赋值

- 无论是拷贝构造还是拷贝赋值，其缺省实现对类类型成员变量和基类子对象，都会调用相应类型的拷贝构造函数和拷贝赋值运算符函数，而不是简单地按字节复制，因此应尽量**避免**使用**指针型**成员变量
- 尽量通过**引用**或**指针**向函数传递对象型**参数**，既可以降低参数传递的开销，也能减少拷贝构造的机会
- 出于具体原因的考虑，确实无法实现完整意义上的拷贝构造和拷贝赋值，可将它们**私有化**，以防误用
- 如果为一个类提供了自定义的拷贝构造函数，就没有理由不提供实现**相同**逻辑的拷贝赋值运算符函数



练习时间

实现一个字符串类String，为其提供可接受C风格字符串的构造函数、析构函数、拷贝构造函数和拷贝赋值运算符函数



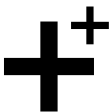
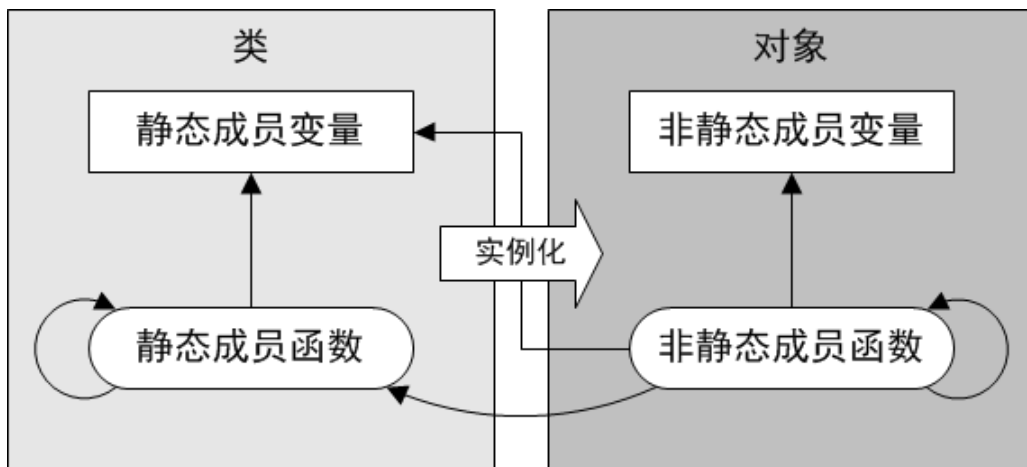
静态成员

- 静态成员**属于类**而不属于对象
 - 静态成员变量**不包含**在对象实例中，**进程级**生命期
 - 静态成员函数**没有this**指针，也**没有常属性**
 - 静态成员依然受**类作用域**和**访问控制**限定符的约束
- 静态成员变量的定义和初始化，只能在**类的外部**而不能在构造函数中进行
- 静态成员变量为该类的所有对象实例所**共享**
- 访问静态成员，既可以**通过类**也可以通过对象
- 静态成员函数**只能**访问静态成员，而非静态成员函数既可以访问静态成员，也可以访问非静态成员



静态成员

- 事实上，类的静态成员变量和静态成员函数，更象是普通的全局变量和全局函数，只是多了一层类作用域和访问控制属性的限制，相当于具有**成员访问性**的**全局变量**和**全局函数**



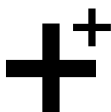
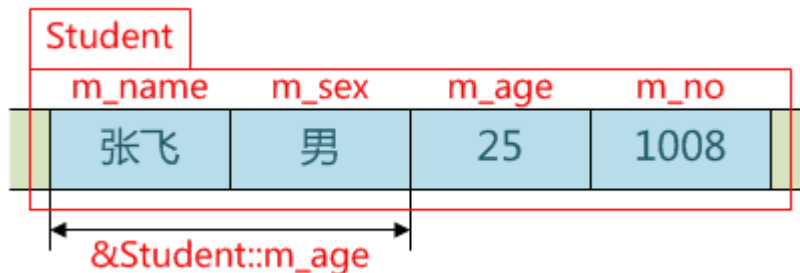
单例模式(Singleton)

- 一个类**仅有一个**实例，通过全局访问点获取之
- 将包括拷贝构造函数在内的所有构造函数**私有化**，防止类的使用者从类的外部创建对象
- 公有静态成员函数**getInstance()**是获取对象实例的唯一渠道
- **饿汉式**：无论用不用，程序启动即创建
- **懒汉式**：用的时候创建，不用了即销毁
 - 永不销毁
 - 引用计数
 - 线程安全



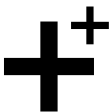
成员指针

- 成员变量指针
 - 类型 类名::*成员变量指针;
 - 成员变量指针 = &类名::成员变量;
 - 对象.*成员变量指针
对象指针->*成员变量指针
 - 其本质就是特定成员变量在对象实例中的**相对地址**，
解引用时再根据调用对象的地址计算出该成员变量的绝对地址



成员指针

- 成员函数指针
 - 返回类型 (类名::*成员函数指针) (形参表);
 - 成员函数指针 = &类名::成员函数名;
 - (对象.*成员函数指针) (实参表)
(对象指针->.*成员函数指针) (实参表)
 - 虽然成员函数并不存储在对象中，但也要通过对象或者对象指针对成员函数指针解引用，其目的只有一个，即提供this指针



成员指针

- 静态成员变量指针
 - 类型* 静态成员变量指针;
 - 静态成员变量指针 = &类名::静态成员变量;
 - *静态成员变量指针
- 静态成员函数指针
 - 返回类型 (*静态成员函数指针) (形参表);
 - 静态成员函数指针 = 类名::静态成员函数名;
 - 静态成员函数指针 (实参表)
- 静态成员与对象无关，因此静态成员指针与普通指针没有任何本质性区别



练习时间

实现一个银行账户类，支持存款、取款、查询余额、结息和调整利率



“

操作符重载

”

全程目标

- 操作符标记与操作符函数
- 典型双目操作符的重载
- 典型单目操作符的重载
- 输入输出操作符的重载
- 其它操作符的重载
- 不能被重载的操作符
- 操作符重载的局限性



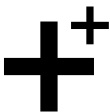
操作符标记与操作符函数

- 操作符标记
 - 单目操作符：-、++、--、*、->等
 - 双目操作符：+、-、+=、-=、>>、<<、[]等
 - 三目操作符：？：
- 操作符函数
 - 在特定条件下，编译器有能力把一个由操作数和操作符共同组成的表达式，解释为对一个**全局或成员函数**的调用，该全局或成员函数被称为操作符函数
 - 通过定义操作符函数，可以实现针对**自定义**类型的**运算法则**，并使之与内置类型一样参与各种表达式



操作符标记与操作符函数

- 双目操作符表达式：L#R
 - 成员函数形式：L.operator# (R)
左操作数是调用对象，右操作数是参数对象
 - 全局函数形式：::operator# (L, R)
左操作数是第一参数，右操作数是第二参数
- 单目操作符表达式：#O/O#
 - 成员函数形式：O.operator# ()
 - 全局函数形式：::operator# (O)
- 三目操作符表达式：F#S#T
 - 无法重载



典型双目操作符的重载

- 运算类双目操作符：+、-、*、/等

- 左右操作数均可为左值或右值

- 表达式的值为右值

- 成员函数形式

```
class LEFT {
    const RESULT operator# (
        const RIGHT& right) const { ... }
};
```

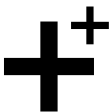
- 全局函数形式

```
const RESULT operator# (
    const LEFT& left, const RIGHT& right) { ... }
```



友元

- 可以通过friend关键字，把一个全局函数、另一个类的成员函数或者另一个类整体，声明为某个类的友元
- 友元拥有访问授权类任何非公有成员的特权
- 友元声明可以出现在授权类的公有、私有或者保护等任何区域，且不受访问控制限定符的约束
- 友元不是成员，其作用域并不隶属于授权类，也不拥有授权类类型的this指针
- 操作符函数常被声明为其参数类型的友元



典型双目操作符的重载

- 赋值类双目操作符：`=`、`+=`、`-=`、`*=`、`/=`等
 - 右操作数为左值或右值，但左操作数必须是左值
 - 表达式的值为左值，且为左操作数本身(而非副本)
 - 成员函数形式


```
class LEFT {
    LEFT& operator# (
        const RIGHT& right) { ... }
};
```
 - 全局函数形式


```
LEFT& operator# (
    LEFT& left, const RIGHT& right) { ... }
```

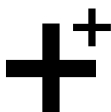


典型单目操作符的重载

- 运算类单目操作符：-、~、！等
 - 操作数为左值或右值
 - 表达式的值为右值
 - 成员函数形式


```
class OPERAND {
    const RESULT operator# (void) const { ... }
};
```
 - 全局函数形式


```
const RESULT operator# (
    const OPERAND& operand) { ... }
```



典型单目操作符的重载

- 前自增减类单目操作符：前++、前--
 - 操作数为左值
 - 表达式的值为左值，且为操作数本身(而非副本)
 - 成员函数形式


```
class OPERAND {
    OPERAND& operator# (void) { ... }
};
```
 - 全局函数形式


```
OPERAND& operator# (OPERAND& operand) { ... }
```

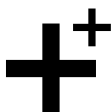


典型单目操作符的重载

- 后自增减类单目操作符：后++、后--
 - 操作数为左值
 - 表达式的值为右值，且为自增减以前的值
 - 成员函数形式


```
class OPERAND {
    const OPERAND operator# (int) { ... }
};
```
 - 全局函数形式


```
const OPERAND operator# (
    OPERAND& operand, int) { ... }
```



输入输出操作符的重载

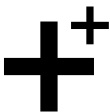
- 输出操作符：`<<`
 - 左操作数为左值形式的输出流(ostream)对象，右操作数为左值或右值
 - 表达式的值为左值，且为左操作数本身(而非副本)
 - 左操作数的类型为ostream，若以成员函数形式重载该操作符，就应将其定义为ostream类的成员，该类为标准库提供，无法添加新的成员，因此只能以全局函数形式重载该操作符
- ```
ostream& operator<< (ostream& os,
 const RIGHT& right) { ... }
```





# 输入输出操作符的重载

- 输入操作符：`>>`
    - 左操作数为左值形式的输入流(istream)对象，右操作数为左值
    - 表达式的值为左值，且为左操作数本身(而非副本)
    - 左操作数的类型为istream，若以成员函数形式重载该操作符，就应将其定义为istream类的成员，该类为标准库提供，无法添加新的成员，因此只能以全局函数形式重载该操作符
- ```
istream& operator>> (istream& is,  
    RIGHT& right) { ... }
```



其它操作符的重载

- 下标操作符：[]
 - 常用于在容器类型中以**下标**方式获取数据元素
 - **非常**容器的元素为**左值**，**常**容器的元素为**右值**

```
class Array {  
public:  
    int& operator[] (size_t i) {  
        return m_array[i];  
    }  
    const int& operator[] (size_t i) const {  
        return const_cast<Array&> (*this)[i];  
    }  
private:  
    int m_array[256];  
};
```

```
Array array;  
array[100] = 1000;  
// array.operator[] (100) = 1000;  
const Array& carr = array;  
cout << carr[100] << endl;  
// cout << carr.operator[] (100) << endl;
```



其它操作符的重载

- 函数操作符：()
 - 如果一个类重载了函数操作符，那么该类的对象就可以被当做函数来调用，其参数和返回值就是函数操作符函数的参数和返回值
 - 参数的个数、类型以及返回值的类型，没有限制
 - 唯一可以带有缺省参数的操作符函数

```
class Less {  
public:  
    bool operator() (int a, int b) const {  
        return a < b;  
    }  
};
```

```
Less less;  
cout << less (100, 200) << endl;  
// cout << less.operator() (100, 200) << endl;
```



其它操作符的重载

- 解引用和间接成员访问操作符：`*`、`->`
 - 如果一个类重载了解引用和间接成员访问操作符，那么该类的对象就可以被当做**指针**来使用

```
class Integer {
public:
    Integer (const int& val = 0) : m_val (val) {}
    int& value (void) {
        return m_val;
    }
    const int& value (void) const {
        return m_val;
    }
private:
    int m_val;
};
```

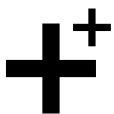
```
class IntegerPointer {
public:
    IntegerPointer (Integer* p = NULL) : m_p (p) {}
    Integer& operator* (void) const {
        return *m_p;
    }
    Integer* operator-> (void) const {
        return m_p;
    }
private:
    Integer* m_p;
};
```

```
IntegerPointer ip (new Integer (100));
(*ip).value ()++;
// ip.operator* ().value ()++;
cout << ip->value () << endl;
// cout << ip.operator-> ()->value () << endl;
```



智能指针(auto_ptr)

- 常规指针的缺点
 - 当一个常规指针离开它的作用域时，只有该指针变量**本身**所占据的内存空间(通常是4个字节)会被释放，而它所**指向**的动态内存并未得到释放
 - 在某些特殊情况下，包含free/delete/delete[]的代码根本就**执行不到**，形成内存泄漏
- 智能指针的优点
 - 智能指针是一个**封装**了常规指针的类类型**对象**，当它离开作用域时，其**析构函数**负责释放该常规指针所指向的动态内存
 - 以正确方式创建的智能指针，其析构函数**总会**执行



智能指针(auto_ptr)



- 智能指针与常规指针的一致性
 - 为了使智能指针也能象常规指针一样，通过 “*” 操作符解引用，通过 “->” 操作符访问其目标的成员，就需要对这两个操作符进行**重载**
- 智能指针与常规指针的不一致性
 - 任何时候，针对同一个对象，只允许有一个**智能指针**持有其地址，否则该对象将在多个智能指针中被析构多次(double free)
 - 智能指针的**拷贝构造**和**拷贝赋值**需要做特殊处理，对其所持有的对象地址，以指针间的**转移**代替**复制**
 - 智能指针的**转移语义**与常规指针的复制语义不一致



其它操作符的重载

- 自定义类型转换

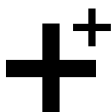
- 通过构造函数实现自定义类型转换

```
class 目标类型 {  
    [explicit] 目标类型 (const 源类型& src) { ... }  
};
```

- 通过类型转换操作符函数实现自定义类型转换

```
class 源类型 {  
    [explicit] operator 目标类型 (void) const { ... }  
};
```

- 若源类型是基本类型，则只能通过构造函数实现自定义类型转换



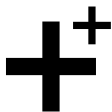
其它操作符的重载

- 自定义类型转换

- 若**目标**类型是**基本**类型，则只能通过**类型转换操作符函数**实现自定义类型转换
- 若**源**类型和**目标**类型都**不是基本**类型，则既可以通过**构造函数**也可以通过**类型转换操作符函数**实现自定义类型转换，但不要两者同时使用，引发歧义错
- 若**源**类型和**目标**类型都是**基本**类型，则**无法实现**自定义类型转换，基本类型间的类型转换规则完全由编译器**内置**

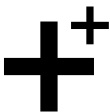
```
class Integer {
public:
    Integer (const int& val = 0) : m_val (val) {}
    operator int (void) const {
        return m_val;
    }
private:
    int m_val;
};

Integer integer (100);
int i = integer;
// int i = integer.operator int ();
```



其它操作符的重载

- 对象创建操作符：new/new[]
 - 如果一个类重载了new/new[]操作符，那么当通过new/new[]创建该类的对象/对象数组时，将首先调用该操作符函数分配内存，然后再调用该类的构造函数
- ```
class 类名 {
 static void* operator new (size_t size) { ... }
 static void* operator new[] (size_t size) { ... }
};
```
- 包含自定义析构函数的类，通过new[]创建对象数组，所分配的内存会在低地址部分预留出sizeof(size\_t)个字节，存放数组长度



# 其它操作符的重载

- 对象创建操作符：new/new[]

```
class Dummy {
public:
 Dummy (void) {}
 ~Dummy (void) {}
 static void* operator new (size_t size) {
 return malloc (size);
 }
 static void* operator new[] (size_t size)
 return malloc (size);
}
};
```

```
Dummy* dummy = new Dummy;
// Dummy* dummy = (Dummy*)(Dummy::operator new (sizeof (Dummy)));
// dummy->Dummy ();
Dummy* dummies = new Dummy[10];
// Dummy* dummies = (Dummy*)((size_t*)Dummy::operator new[] (
// sizeof (size_t) + 10 * sizeof (Dummy)) + 1);
// *((size_t*)dummies - 1) = 10;
// for (size_t i = 0; i < *((size_t*)dummies - 1); ++i)
// (dummies + i)->Dummy ();
```



# 其它操作符的重载

- 对象销毁操作符：delete/delete[]
  - 如果一个类重载了delete/delete[]操作符，那么当通过delete/delete[]销毁该类的对象/对象数组时，将首先调用该类的析构函数，然后再调用该操作符函数释放内存

```
class 类名 {
 static void operator delete (void* p) { ... }
 static void operator delete[] (void* p) { ... }
};
```

- 包含自定义析构函数的类，通过delete[]销毁对象数组，会根据低地址部分预存的数组长度，从高地址到低地址依次对每个数组元素调用析构函数

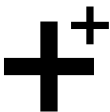


# 其它操作符的重载

- 对象销毁操作符：delete/delete[]

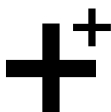
```
class Dummy {
public:
 Dummy (void) {}
 ~Dummy (void) {}
 static void operator delete (void* p) {
 free (p);
 }
 static void operator delete[] (void* p) {
 free (p);
 }
};
```

```
delete dummy;
// dummy->~Dummy ();
// Dummy::operator delete (dummy);
delete[] dummies;
// for (size_t i = *((size_t*)dummies - 1) - 1; ; --i) {
// (dummies + i)->~Dummy ();
// if (i == 0) break;
// }
// Dummy::operator delete[] ((size_t*)dummies - 1);
```



# 操作符重载的限制

- 不是所有的操作符都能重载，以下操作符不能重载
  - 作用域限定操作符(::)
  - 直接成员访问操作符(.)
  - 直接成员指针解引用操作符(.\*)
  - 条件操作符(?:)
  - 字节长度操作符(sizeof)
  - 类型信息操作符 typeid
- 无法重载所有操作数均为基本类型的操作符
  - $1 + 1 = 8$  ?



# 操作符重载的限制


- 无法通过操作符重载改变操作符的**优先级**
  - $a + b \wedge c \Leftrightarrow a + b^c ?$
- 无法通过操作符重载改变**操作数**的个数
  - $50\% \Leftrightarrow 0.5 ?$
- 无法通过操作符重载**发明**新的操作符
  - $x ** y \Leftrightarrow x^y ?$
- 操作符重载着力于对**一致性**的追求
  - $(3+4i)+(1+2i) = 2+2i ?$



# 操作符重载的限制

- 操作符重载的价值在于提高代码的**可读性**，而不是沦为少数“算符控”们赖以卖弄的**奇技淫巧**！

```
class Stack {
public:
 Stack (void) { ... };
 ~Stack (void) { ... };
 void operator+ (int data) { ... } // 压入
 int operator- (void) { ... } // 弹出
 int operator* (void) const { ... } // 栈顶
 operator bool (void) const { ... } // 判空
 ...
};
```



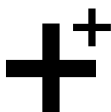
```
Stack stack;
for (int i = 0; i < 10; ++i)
 stack + i;
while (! stack) {
 cout << *stack << endl;
 -stack;
}
```



# 练习时间

实现一个3X3的矩阵类支持如下操作符：

$*/-(减)/*/+ = /- = /* = /-(负)/*++(前后)/--(前后)/[]/<<$





“

**继承**

”

# 全程目标

- 继承的基本概念和语法
- 公有继承的基本特点
- 继承方式与访问控制
- 子类的构造与析构
- 子类的拷贝构造与拷贝赋值
- 子类的操作符重载
- 名字隐藏与重载
- 私有继承与保护继承
- 多重继承、钻石继承与虚继承



# 继承的基本概念和语法

- 共性与个性

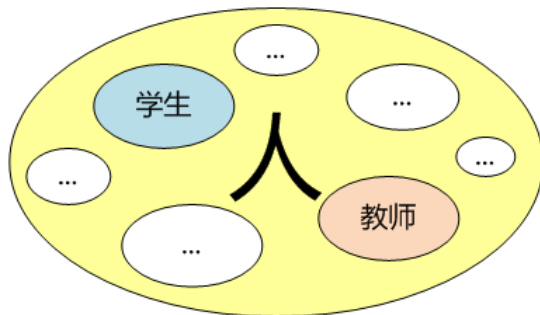
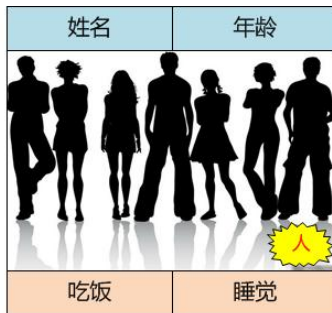
| 姓名                                                                                | 年龄 | 学号 |
|-----------------------------------------------------------------------------------|----|----|
|  |    |    |
| 吃饭                                                                                | 睡觉 | 学习 |

| 姓名                                                                                 | 年龄 | 工资 |
|------------------------------------------------------------------------------------|----|----|
|  |    |    |
| 吃饭                                                                                 | 睡觉 | 授课 |



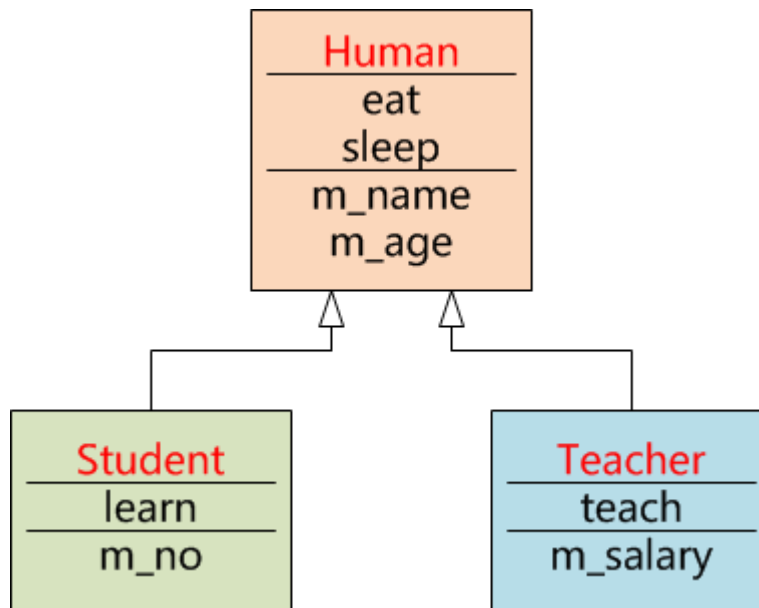
# 继承的基本概念和语法

- 超集与子集



# 继承的基本概念和语法

- 基类与子类



# 继承的基本概念和语法

- 继承与派生

```
class Human {
public:
 void eat (const string& food) { ... }
 void sleep (int hours) { ... }
 string m_name;
 int m_age;
};
```

```
class Student : public Human {
public:
 void learn (
 const string& course) { ... }
 int m_no;
};
```

```
class Teacher : public Human {
public:
 void teach (
 const string& course) { ... }
 float m_salary;
};
```

# 继承的基本概念和语法

- 继承的语法
  - class 子类 : 继承方式1 基类1, 继承方式2 基类2, ... {
    - ...
  - };
- 继承方式
  - 公有继承 : public
  - 保护继承 : protected
  - 私有继承 : private



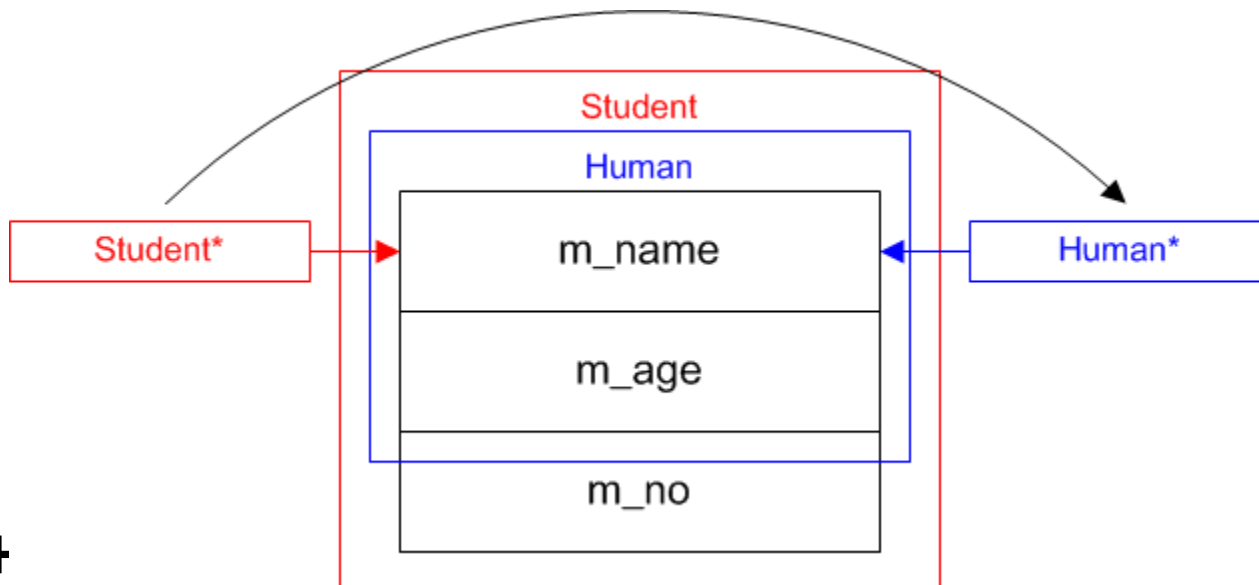
# 公有继承的基本特点

- 子类对象任何时候都可以被当做其基类类型的对象
  - ```
class Human { ... };  
class Student : public Human { ... };  
Student student (...);  
Human* phuman = &student;  
Human& rhuman = student;
```
 - 编译器认为访问范围缩小是安全的



公有继承的基本特点

- 子类对象任何时候都可以被当做其基类类型的对象



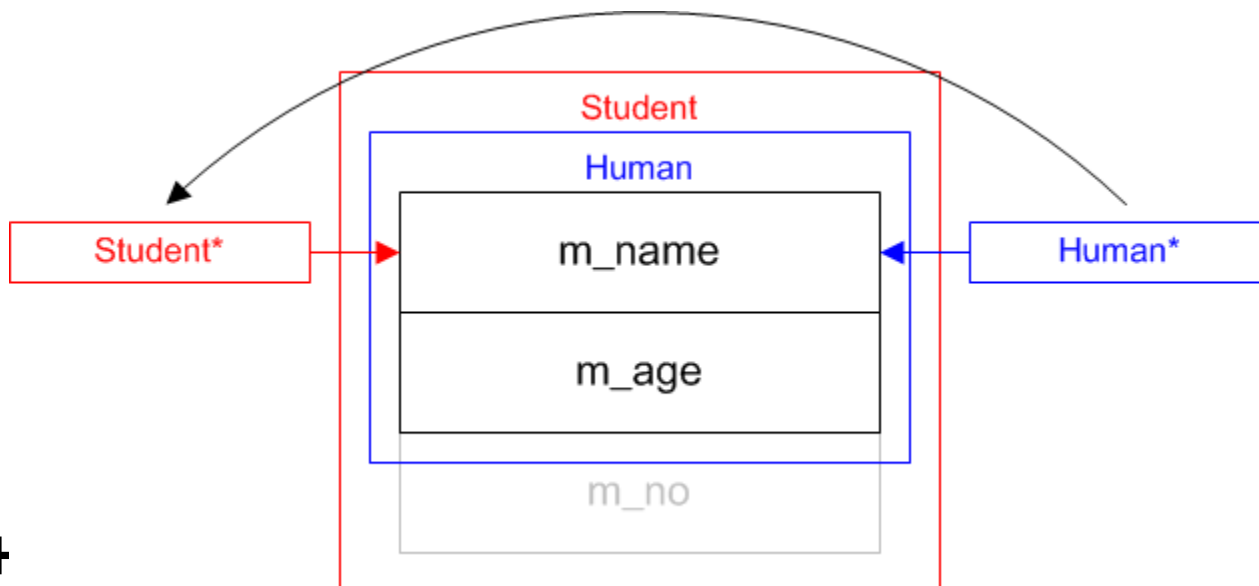
公有继承的基本特点

- 基类类型的指针或者引用不能隐式转换为子类类型
 - ```
class Human { ... };
class Student : public Human { ... };
Human human (...);
Student* pstudent =
 static_cast<Student*> (&human);
Student& rstudent =
 static_cast<Student&> (human);
```
  - 编译器认为访问范围扩大是危险的



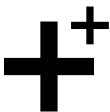
# 公有继承的基本特点

- 基类类型的指针或者引用不能隐式转换为子类类型



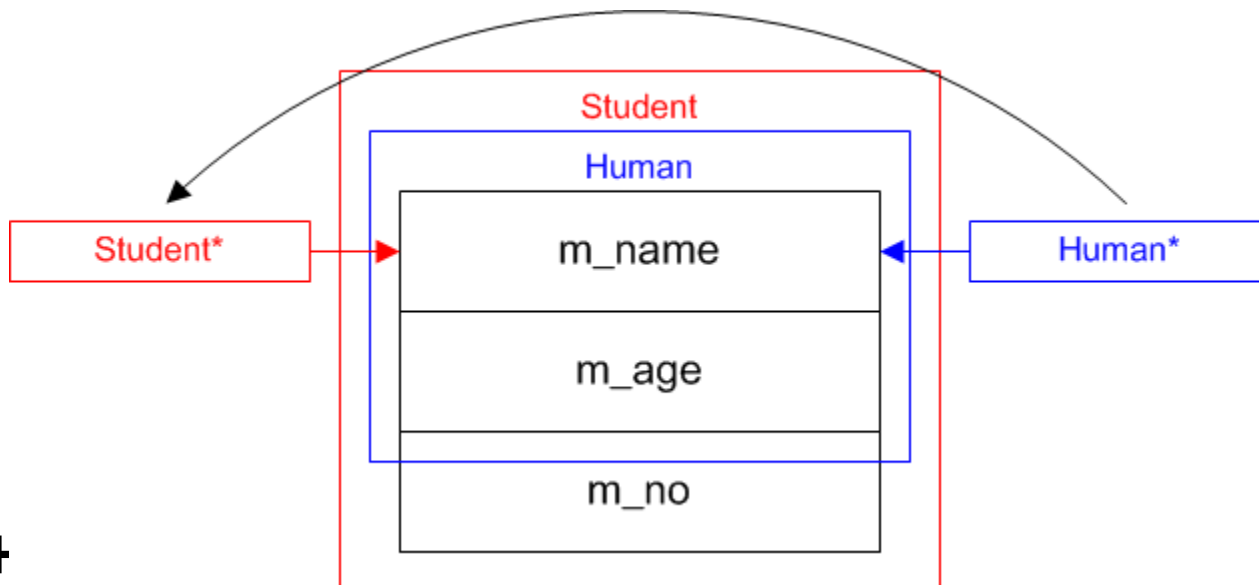
# 公有继承的基本特点

- 编译器对类型安全的检测仅仅基于指针或引用本身
  - ```
class Human { ... };  
class Student : public Human { ... };  
Student student (...);  
Human* phuman = &student;  
Human& rhuman = student;  
Student* pstudent =  
    static_cast<Student*> (phuman);  
Student& rstudent =  
    static_cast<Student&> (rhuman);
```
 - 基类指针或引用的实际目标，究竟是不是子类对象，完全由程序员自己判断



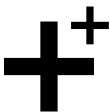
公有继承的基本特点

- 编译器对类型安全的检测仅仅基于指针或引用本身



公有继承的基本特点

- 在子类中可以**直接访问**基类的所有**公有**和**保护**成员，就如同它们是在子类中声明的一样
- 基类的**私有**成员在子类中虽然存在却**不可见**，故无法直接访问
- 尽管基类的公有和保护成员在子类中直接可见，但仍然可以在子类中重新定义这些名字，子类中的名字会**隐藏**所有基类中的同名定义
- 如果需要在子类中或通过子类访问一个在基类中定义却为子类所隐藏的名字，可以借助作用域限定操作符 “::” 实现



继承方式与访问控制

- 类成员的访问控制限定符与访问控制属性

访问控制 限定符	访问控制 属性	基类	子类	外部	友元
public	公有成员	OK	OK	OK	OK
protected	保护成员	OK	OK	NO	OK
private	私有成员	OK	NO	NO	OK



继承方式与访问控制

- 基类中的公有、保护和私有成员，在其公有、保护和私有子类中的访问控制属性，会因继承方式而异

基类中的	在公有子类中 变成	在保护子类中 变成	在私有子类中 变成
公有成员	公有成员	保护成员	私有成员
保护成员	保护成员	保护成员	私有成员
私有成员	私有成员	私有成员	私有成员

- 当通过子类访问其所继承的基类的成员时，需要考虑继承方式对访问控制属性的影响

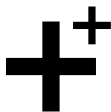


<pre> class A { public: int m_pub; protected: int m_pro; private: int m_pri; }; </pre>			
<pre> class B : { void foo (void) { m_pub = 10; m_pro = 10; m_pri = 10; } }; </pre>	public A	protected A	private A
	Ok	Ok	Ok
	Ok	Ok	Ok
	No	No	No
<pre> class C : { void foo (void) { m_pub = 10; m_pro = 10; m_pri = 10; } }; </pre>	public/protected/private B		
	Ok	Ok	No
	Ok	Ok	No
	No	No	No
<pre> int main (void) { B b; b.m_pub = 10; b.m_pro = 10; b.m_pri = 10; return 0; } </pre>			
	Ok	No	No
	No	No	No
	No	No	No



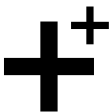
子类的构造与析构

- 子类构造函数隐式调用基类构造函数
 - 如果子类的构造函数没有显式指明其基类部分的构造方式，那么编译器会选择其基类的缺省构造函数，构造该子类对象中的基类子对象
- 子类构造函数显式调用基类构造函数
 - 子类的构造函数可以在初始化表中显式指明其基类部分的构造方式，即通过其基类的特定构造函数，构造该子类对象中的基类子对象
- 子类对象的构造过程
 - 构造基类子对象->构造成员变量->执行构造代码



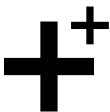
子类的构造与析构

- 阻断继承
 - 子类的构造函数**无论如何**都会调用基类的构造函数，构造子类对象中的基类子对象
 - 如果把基类的构造函数定义为**私有**，那么该类的子类就永远无法被实例化为对象
 - 在C++中可以用这种方法**阻断**一个类被扩展



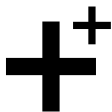
子类的构造与析构

- 子类析构函数隐式调用基类析构函数
 - 子类的析构函数在执行完其中的析构代码，并析构完所有的成员变量以后，会**自动调用**其基类的析构函数，析构该子类对象中的基类子对象
- 基类析构函数不会调用子类析构函数
 - 通过基类指针析构子类对象，实际被析构的仅仅是子类对象中的**基类子对象**，子类的扩展部分将失去被析构的机会，极有可能形成内存泄漏
- 子类对象的析构过程
 - 执行**析构代码**->析构**成员**变量->析构**基类**子对象



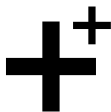
子类的拷贝构造与拷贝赋值

- 子类没有定义拷贝构造函数
 - 编译器为子类提供的缺省拷贝构造函数，会自动调用其基类的**拷贝构造函数**，构造该子类对象中的基类子对象
- 子类定义了拷贝构造函数，但没有显式指明其基类部分的构造方式
 - 编译器会选择其基类的**缺省构造函数**，构造该子类对象中的基类子对象
- 子类定义了拷贝构造函数，同时显式指明了其基类部分以拷贝方式构造
 - 子类对象中的**基类部分**和**扩展部分**一起被复制



子类的拷贝构造与拷贝赋值

- 子类没有定义拷贝赋值运算符函数
 - 编译器为子类提供的缺省拷贝赋值运算符函数，会自动调用其基类的**拷贝赋值运算符函数**，复制该子类对象中的基类子对象
- 子类定义了拷贝赋值运算符函数，但没有显式调用其基类的拷贝赋值运算符函数
 - 子类对象中的基类子对象将**得不到复制**
- 子类定义了拷贝赋值运算符函数，同时显式调用了其基类的拷贝赋值运算符函数
 - 子类对象中的**基类部分**和**扩展部分**一起被复制



子类的操作符重载

- 在为子类提供操作符重载定义时，往往需要调用其**基类**针对该操作符所做的重载定义，完成部分工作
- 通过将子类对象的指针或引用**向上造型**为其基类类型的指针或引用，可以迫使针对基类的操作符重载函数在针对子类的操作符重载函数中被调用

```
ostream& operator<< (ostream& os,  
    const Manager& manager) {  
    return os << (Employee&)manager << ", "  
        << manager.m_title;  
}
```



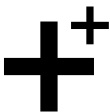
名字隐藏与重载

- 继承不会改变类成员的**作用域**，基类的成员永远都是基类的成员，并不会因为继承而变成子类的成员
- 因为作用域的不同，分别在子类和基类中定义的**同名**成员函数(包括静态成员函数)，并不构成重载关系，相反是一种**隐藏**关系，除非通过**using**声明将基类的成员函数引入子类的作用域，形成重载
- 任何时候，无论在子类的内部还是外部，总可以通过作用域限定操作符“**::**”，显式地调用那些在基类中定义却为子类所隐藏的成员函数



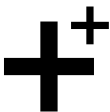
私有继承与保护继承

- 私有继承亦称实现继承，旨在于子类中将其基类的公有和保护成员**私有化**，既禁止从外部通过该子类访问这些成员，也禁止在该子类的子类中访问这些成员
- 保护继承是一种特殊形式的实现继承，旨在于子类中将其基类的公有和保护成员进行**有限的私有化**，只禁止从外部通过该子类访问这些成员，但并不禁止在该子类的子类中访问这些成员
- 私有子类和保护子类类型的指针或引用，**不能隐式转换**为其基类类型的指针或引用



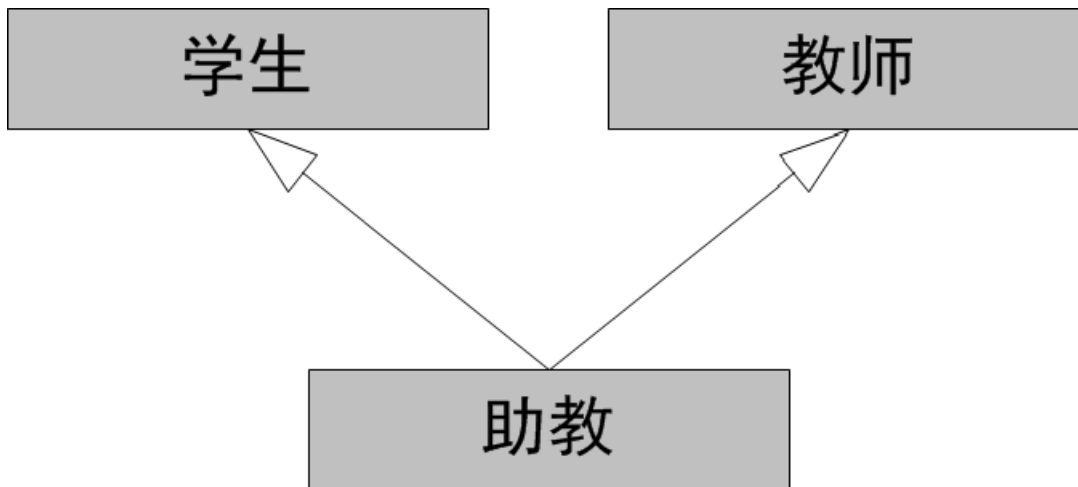
多重继承、钻石继承与虚继承

- 一个类可以同时从多个基类继承实现代码
 - 智能手机系统



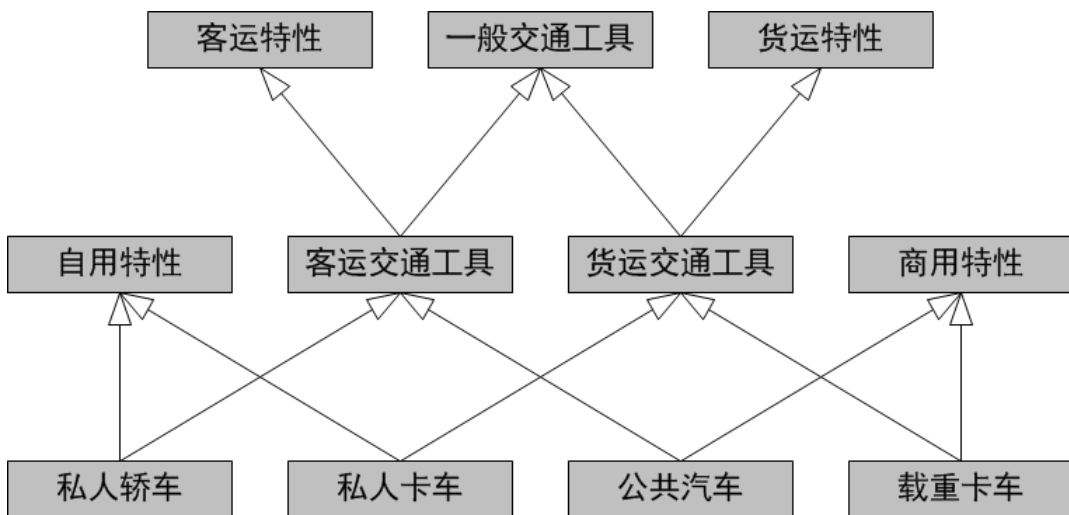
多重继承、钻石继承与虚继承

- 一个类可以同时从多个基类继承实现代码
 - 课堂教学系统



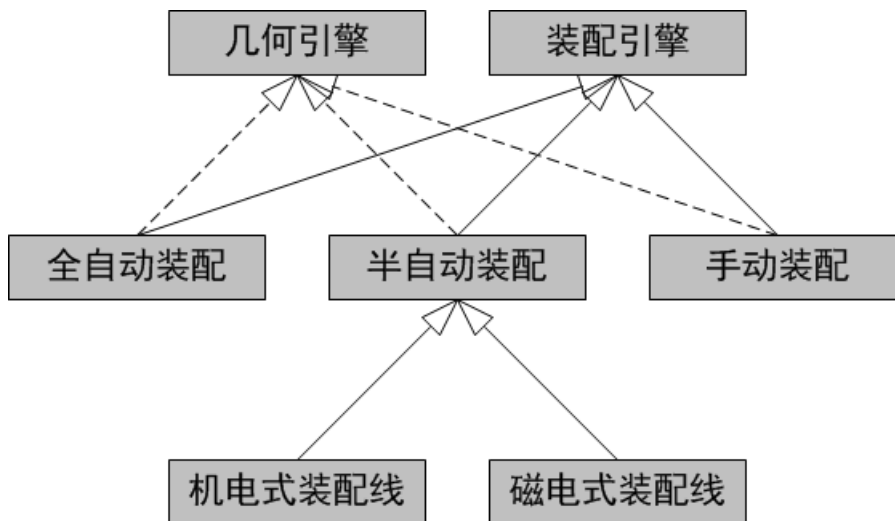
多重继承、钻石继承与虚继承

- 一个类可以同时从多个基类继承实现代码
 - 交通工具系统



多重继承、钻石继承与虚继承

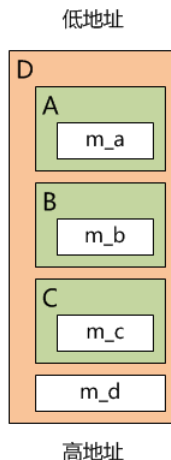
- 一个类可以同时从多个基类继承实现代码
 - 零件装配系统



多重继承、钻石继承与虚继承

- 多重继承的内存布局和类型转换
 - 子类对象中的多个基类子对象，按照继承表的顺序依次被构造，并从低地址到高地址排列，析构的顺序则与构造严格相反

```
class A {  
public:  
    int m_a;  
};  
class B {  
public:  
    int m_b;  
};  
class C {  
public:  
    int m_c;  
};  
class D : public A, public B, public C {  
public:  
    int m_d;  
};
```



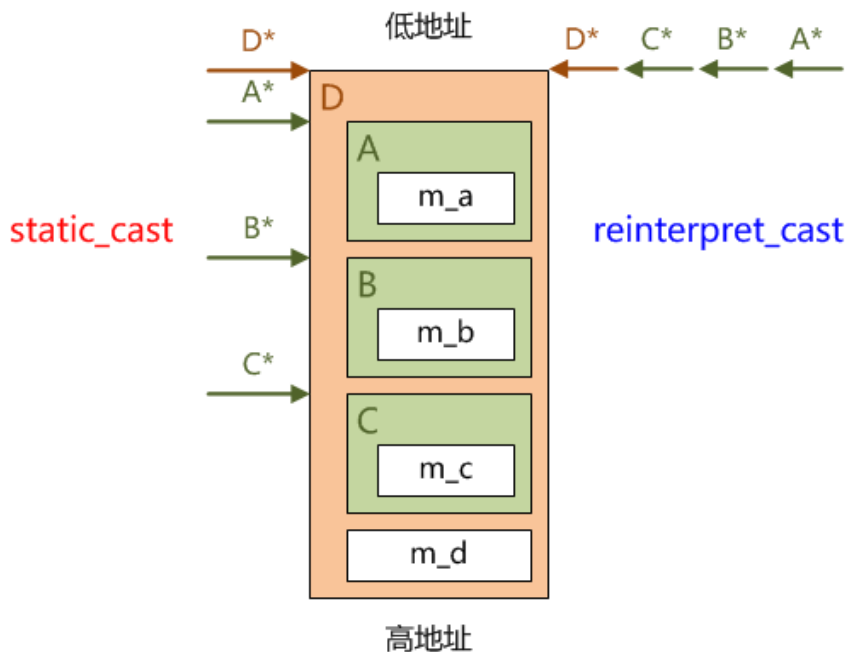
多重继承、钻石继承与虚继承

- 多重继承的内存布局和类型转换
 - 将继承自多个基类的子类类型的指针，隐式或静态转换为它的基类类型，编译器会根据各个基类对象在子类对象中的内存布局，**进行适当的偏移计算**，以保证指针的类型与其所指向目标对象的类型一致
 - 反之，将该子类的任何一个基类类型的指针静态转换为子类类型，编译器同样会**进行适当的偏移计算**
 - 无论在哪个方向上，重解释类型转换(reinterpret_cast)都**不进行任何偏移计算**
 - 引用的情况与指针类似，因为引用的本质就是指针



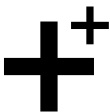
多重继承、钻石继承与虚继承

- 多重继承的内存布局和类型转换



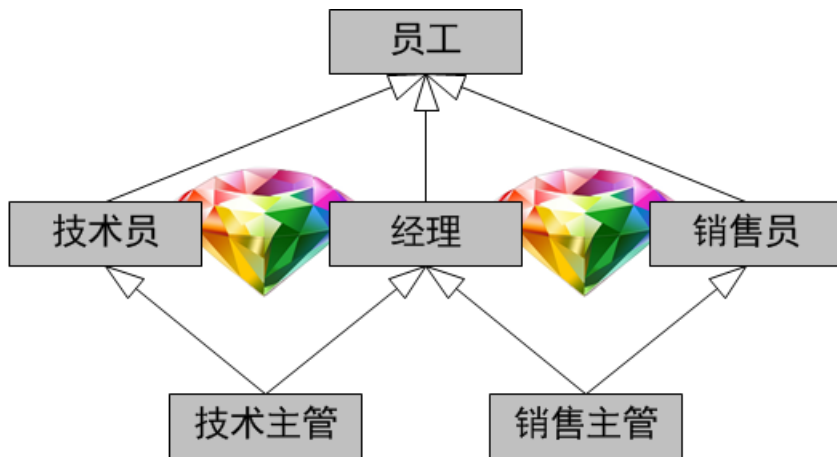
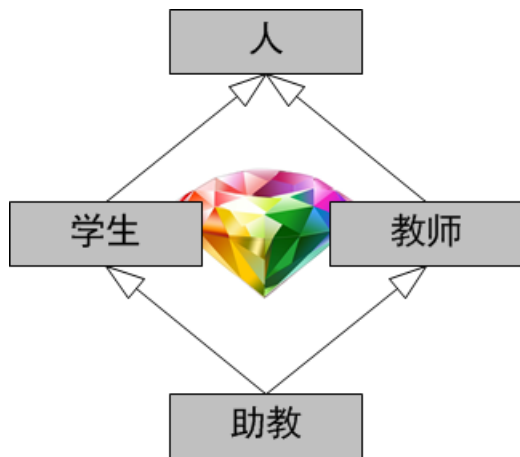
多重继承、钻石继承与虚继承

- 围绕多重继承，历来争议颇多
 - 现实世界中的实体本来就具有同时从多个来源共同继承的特性，因此多重继承有助于面向现实世界的问题域直接建立逻辑模型
 - 多重继承可能会在大型程序中引入令人难以察觉的BUG，并极大地增加对类层次体系进行扩展的难度
- 名字冲突问题
 - 如果在子类的多个基类中，存在同名的标识符，而且子类又没有隐藏该名字，那么任何试图在子类中，或通过子类对象访问该名字的操作，都将引发歧义，除非通过作用域限定操作符“::”显式指明所属基类



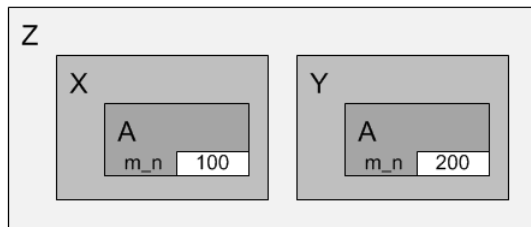
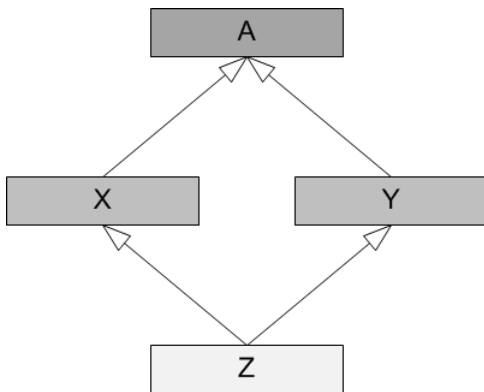
多重继承、钻石继承与虚继承

- 钻石继承问题
 - 一个子类继承自多个基类，而这些基类又源自**共同**的祖先，这样的继承结构称为钻石继承(菱形继承)



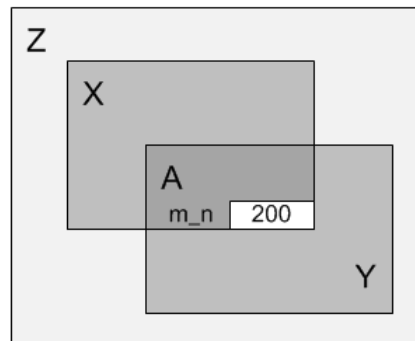
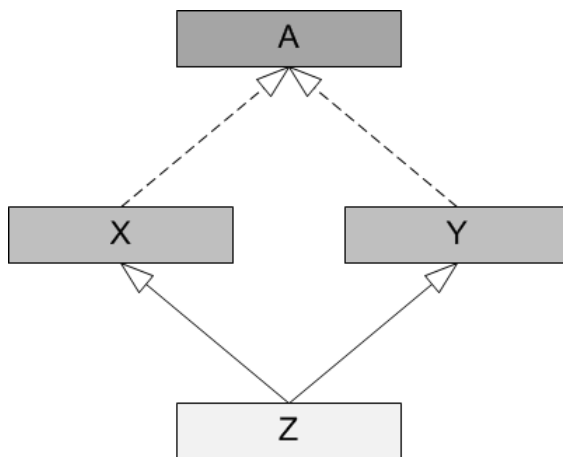
多重继承、钻石继承与虚继承

- 钻石继承问题
 - 派生多个中间子类的**公共基类**子对象，在继承自多个中间子类的**汇聚子类**对象中，存在**多个实例**
 - 在汇聚子类中，或通过汇聚子类对象，访问公共基类的成员，会因继承路径的不同而导致**不一致**



多重继承、钻石继承与虚继承

- 钻石继承问题
 - 通过虚继承，可以保证公共基类子对象在汇聚子类对象中，仅存一份实例，且为多个中间子类对象所共享



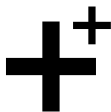
多重继承、钻石继承与虚继承

- 虚继承、虚基类、虚表和虚表指针
 - 在继承表中使用**virtual**关键字
 - 位于继承链**最末端**的子类的构造函数负责构造虚基类子对象
 - 虚基类的**所有**子类(无论直接的还是间接的)都必须在其构造函数中**显式指明**该虚基类子对象的**构造方式**，否则编译器将选择以缺省方式构造该子对象
 - 虚基类的**所有**子类(无论直接的还是间接的)都必须在其拷贝构造函数中**显式指明**以**拷贝方式**构造该虚基类子对象，否则编译器将选择以缺省方式构造该子对象



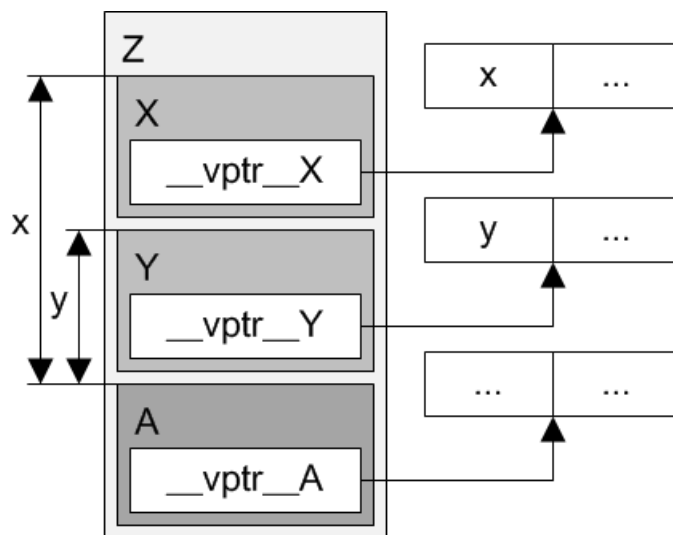
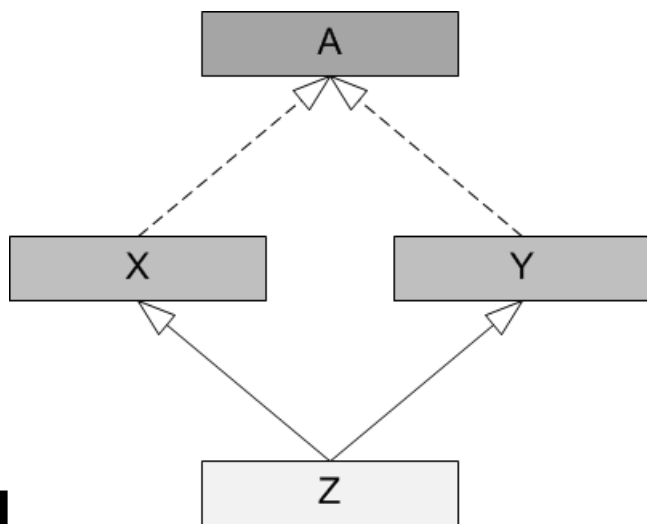
多重继承、钻石继承与虚继承

- 虚继承、虚基类、虚表和虚表指针
 - 与构造函数和拷贝构造函数的情况不同，无论是否存在虚基类，**拷贝赋值运算符函数**的实现没有区别
 - 汇聚子类对象中的每个中间子类子对象都持有一个**虚表指针**，该指针指向一个被称为**虚表**的指针数组的中部，该数组的高地址侧存放虚函数指针，低地址侧则存放所有虚基类子对象相对于每个中间子类子对象起始地址的**偏移量**
 - 某些C++实现会将虚基类子对象的**绝对地址**直接存放在中间子类子对象中，而另一些实现(比如微软)则提供了单独的**虚基类表**，但它们的基本原理都一样



多重继承、钻石继承与虚继承

- 虚继承、虚基类、虚表和虚表指针
 - 包含虚继承的对象模型



练习时间

薪酬计算



所有员工：姓名、工号、等级

经理：绩效奖金

技术员：研发津贴(元/小时)

销售员：提成比率

薪资=基本工资+绩效工资

基本工资=等级额度X出勤率

绩效工资的计算因职务而异

普通员工：基本工资的一半

经理：绩效奖金X绩效因数(输入)

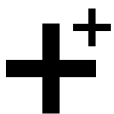
技术员：研发津贴X工作小时数X进度因数(输入)

销售员：销售额(输入)X提成比率

技术主管：技术员绩效工资和经理绩效工资的平均数

销售主管：销售员绩效工资和经理绩效工资的平均数

打印员工信息，输入必要数据，计算薪酬



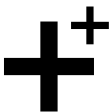
“

多态

”

全程目标

- 非虚的世界
- 虚函数、覆盖和多态
- 覆盖的条件
- 多态的条件
- 纯虚函数、抽象类和纯抽象类
- 好莱坞模式
- 虚函数表与动态绑定
- 运行时类型信息(RTTI)
- 虚析构
- 抽象工厂模式



非虚的世界

- 对象的自恰性
 - 对同样的函数调用，每个对象都会做出**恰当**的响应
- 通过指向子类对象的基类指针调用函数
 - 只能调用**基类**的成员函数，虽然指针指向子类对象
 - 一旦调用子类所特有的成员函数，将引发**编译错误**
- 通过指向基类对象的子类指针调用函数
 - 可以调用**子类**的成员函数，尽管指针指向基类对象
 - 直接或间接地访问子类的成员变量，后果**不可预知**
- 名字隐藏
 - 子类的成员函数**隐藏**基类的同名成员函数

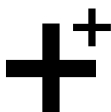


虚函数、覆盖和多态

- 虚函数
 - 形如

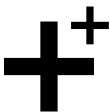

```
class 类名 {
    virtual 返回类型 函数名 (形参表) { ... }
};
```

 的成员函数，称为虚函数或方法
- 覆盖
 - 如果子类的成员函数和基类的虚函数**具有相同的函数原型**，那么该成员函数就也是虚函数，无论其是否带有virtual关键字，且对基类的虚函数构成**覆盖**



虚函数、覆盖和多态

- 多态
 - 如果子类提供了对基类虚函数的有效覆盖，那么通过一个指向子类对象的基类指针，或者引用子类对象的基类引用，调用该虚函数，实际被调用的将是子类中的覆盖版本，而非基类中的原始版本，这种现象称为多态
 - 多态的重要意义在于，一般情况下，调用哪个类的成员函数是由调用者指针或引用本身~~本身~~的类型决定的，而当多态发生时，调用哪个类的成员函数则完全由调用者指针或引用的实际目标对象的类型决定



覆盖的条件

- 有效的虚函数覆盖需要满足如下条件
 - 该函数必须是**成员函数**，既不能是全局函数也不能是静态成员函数
 - 该函数必须在基类中用**virtual**关键字声明为虚函数
 - 覆盖版本与基类版本必须拥有**完全相同的签名**，即函数名、形参表和常属性严格一致
 - 如果基类版本返回**基本**类型数据，那么覆盖版本**必须返回相同**类型的数据
 - 如果基类版本返回**类**类型对象的指针或引用，那么覆盖版本**可以返回其子类**类型对象的指针或引用



覆盖的条件

- 有效的虚函数覆盖需要满足如下条件
 - 如果基类版本带有异常说明，那么覆盖版本**不能说**
明比基类版本抛出**更多的异常**
 - 无论基类版本位于基类的公有、私有还是保护部分，覆盖版本都可以出现在子类包括公有、私有和保护在内的**任何部分**

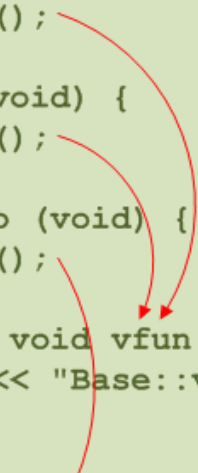


多态的条件

- 多态特性除了需要在基类中定义虚函数以外，还必须借助**指针**或者**引用**调用该虚函数，才能表现出来
- 调用虚函数的指针也可能是基类中的**this**指针，同样能满足多态的条件，但在**构造**和**析构**函数中除外
- 全局函数形式的操作符函数无法实现多态，只有**成员函数**形式的操作符函数，才能借助于虚函数，表现出多态特性

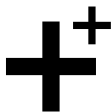


```
class Base {  
public:  
    Base (void) {  
        vfun ();  
    }  
    ~Base (void) {  
        vfun ();  
    }  
    void foo (void) {  
        vfun ();  
    }  
    virtual void vfun (void) {  
        cout << "Base::vfun()" << endl;  
    }  
};
```



```
class Derived : public Base {  
public:  
    void vfun (void) {  
        cout << "Derived::vfun()" << endl;  
    }  
};
```

```
Derived d;  
d.foo ();
```



纯虚函数、抽象类和纯抽象类

- 纯虚函数
 - 形如

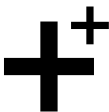
```
class 类名 {  
    virtual 返回类型 函数名 (形参表) = 0;  
};
```

的虚函数，称为纯虚函数或抽象方法
- 抽象类
 - 至少拥有一个纯虚函数的类称为抽象类
 - 抽象类不能实例化为对象
 - 抽象类的子类如果不对基类中的全部纯虚函数提供有效的覆盖，那么该子类就也是抽象类



纯虚函数、抽象类和纯抽象类

- 纯抽象类
 - **全部**由纯虚函数构成的抽象类称为纯抽象类或接口



好莱坞模式—职责分离

Don't call me, I'll call you !

```
class Parser {
public:
    void parse (const char* file, ...) {
        ...
        onText (...);
        ...
        onRect (...);
        ...
        onCircle (...);
        ...
        onImage (...);
        ...
    }
private:
    virtual void onText    (...) = 0;
    virtual void onRect    (...) = 0;
    virtual void onCircle  (...) = 0;
    virtual void onImage   (...) = 0;
};
```

```
class Render : public Parser {
private:
    void onText    (...) { ... }
    void onRect    (...) { ... }
    void onCircle  (...) { ... }
    void onImage   (...) { ... }
};
```

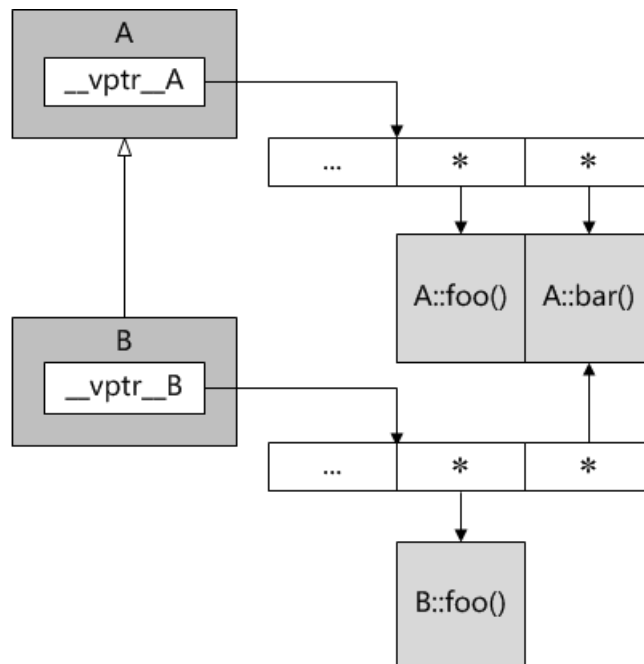
```
Render render (...);
render.parse ("c++primer.pdf", ...);
```

虚函数表与动态绑定

- 虚函数表

```
class A {
public:
    virtual void foo (void) { ... }
    virtual void bar (void) { ... }
};

class B : public A {
public:
    void foo (void) { ... }
};
```



虚函数表与动态绑定

- 动态绑定
 - 当编译器看到通过指针或引用调用虚函数的语句时，并不急于生成有关函数调用的指令，相反它会用一段代码替代该语句，这段代码在运行时被执行，完成如下操作：
 1. 确定调用指针或引用的目标对象的真实类型
 2. 从调用指针或引用的目标对象中找到虚函数表，并从虚函数表中获取所调用虚函数的入口地址
 3. 根据入口地址，调用该函数



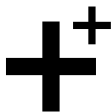
虚函数表与动态绑定

- 动态绑定对性能的影响
 - 虚函数表本身会增加内存空间的开销
 - 与普通函数调用相比，虚函数调用要多出几个步骤，增加运行时间的开销
 - 动态绑定会妨碍编译器通过内联来优化代码
 - 只有在确实需要多态特性的场合才使用虚函数，否则尽量使用普通函数



运行时类型信息(RTTI)

- 动态类型转换(dynamic_cast)
 - 用于将基类类型的指针或引用转换为其子类类型的指针或引用，前提是子类必须从基类多态继承，即基类包含至少一个虚函数
 - 动态类型转换会对所需转换的基类指针或引用做检查，如果其目标确实为期望得到的子类类型的对象，则转换成功，否则转换失败
 - 针对指针的动态类型转换，以返回空指针(NULL)表示失败，针对引用的动态类型转换，以抛出bad_cast异常表示失败



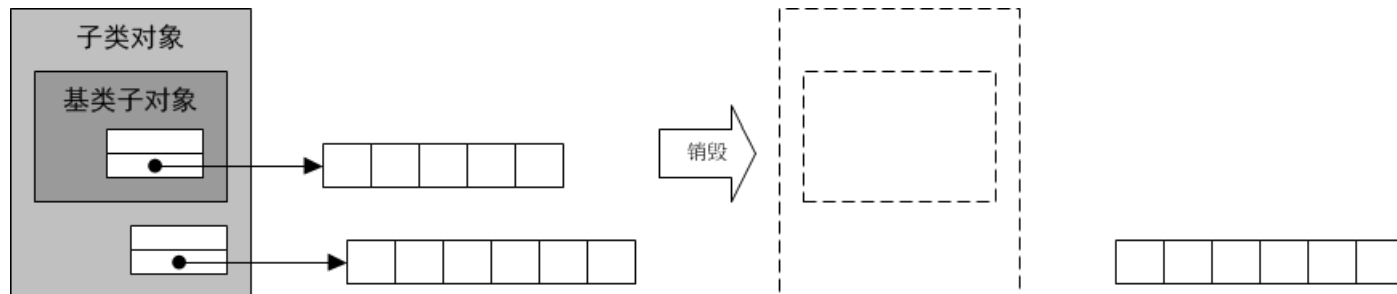
运行时类型信息(RTTI)

- typeid操作符
 - #include <typeinfo>
 - 返回typeinfo类型对象的常引用
 - ✓ typeid类的成员函数name(), 返回空字符结尾的类型名字符串首地址
 - ✓ typeid类支持 “==” 和 “!=” 操作符, 可直接用于类型相同与否的判断
 - 当其作用于基类类型的指针或引用的目标时, 若基类包含至少一个虚函数, 即存在多态继承, typeid所返回类型信息将由该指针或引用的实际目标对象的类型决定, 否则由该指针或引用本身的类型决定



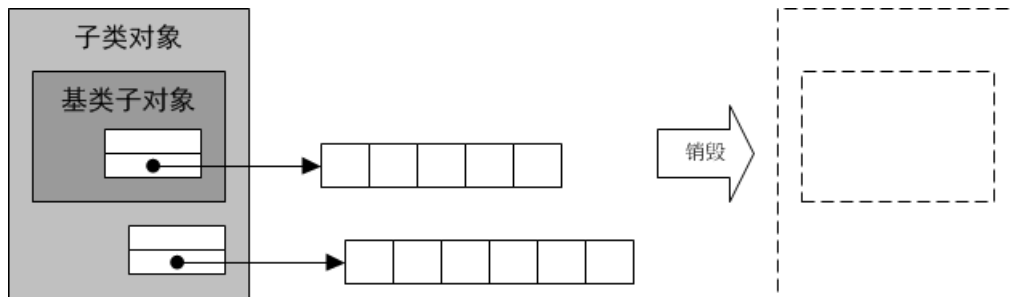
虚析构

- delete一个指向子类对象的基类指针
 - 实际被调用的仅仅是基类的析构函数
 - 基类的析构函数负责析构子类对象中的基类子对象
 - 基类的析构函数不会调用子类的析构函数
 - 在子类中分配的资源将形成内存泄漏



虚析构

- delete一个指向子类对象的基类指针
 - 如果将**基类**的**析构函数**声明为**虚函数**，那么实际被调用的将是子类的析构函数
 - 子类的析构函数将首先析构子类对象的**扩展部分**，然后再通过基类的析构函数析构该对象**基类部分**，最终实现完美的资源释放



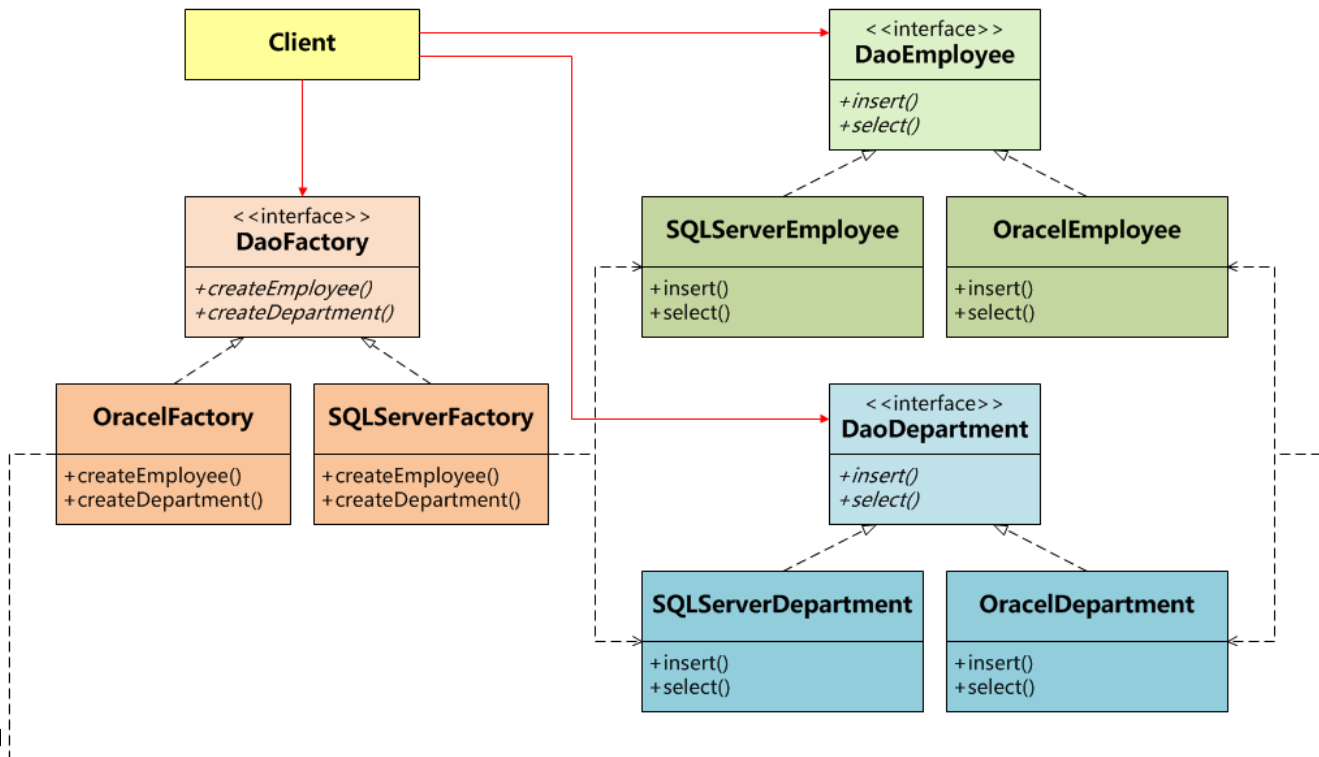
虚析构

- 空虚析构造函数
 - 没有分配任何资源的类，**无需定义**析构造函数
 - 没有定义析构造函数的类，编译器会为其提供一个缺省析构造函数，但**缺省析构造函数并不是虚函数**
 - 为了保证delete一个指向子类对象的基类指针时，能够正确调用子类的析构造函数，就必须把基类的析构造函数定义为**虚函数**，即使它是一个**空函数**
 - 任何时候，为基类定义一个虚析构造函数总是**无害的**
- 一个类中，除了**构造函数**和**静态成员函数**外，任何函数都可以被声明为虚函数



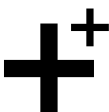
抽象工厂模式—接口与解耦

知识案例



练习时间

将前面薪酬计算的程序改用多态实现，看是否可以使代码更加精炼



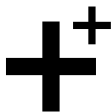
“

异常

”

全程目标

- 错误与错误处理
- 异常处理语法
- 异常处理流程
- 异常说明
- 异常处理模式
- 构造函数中的异常
- 析构函数中的异常
- 标准库异常



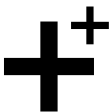
错误与错误处理

- 何为错误？
 - 语法错误：程序员在编码阶段解决并通过**编译**
 - 逻辑错误：程序员借助于**调试**工具诊断并修改
 - 功能错误：程序员修改**代码**，测试员回归验证
 - 设计缺陷：设计员修改**设计**，程序员重新编码
 - 需求不符：分析员修改**需求**，设计员调整设计
 - 环境异常：客服协助用户调整程序的**运行环境**
 - 操作不当：客服指导用户按照**正确的方法**操作
- 错误处理主要针对在实际运行环境中发生，却在设计、编码和测试阶段**无法预料**的，各种**潜在**的异常



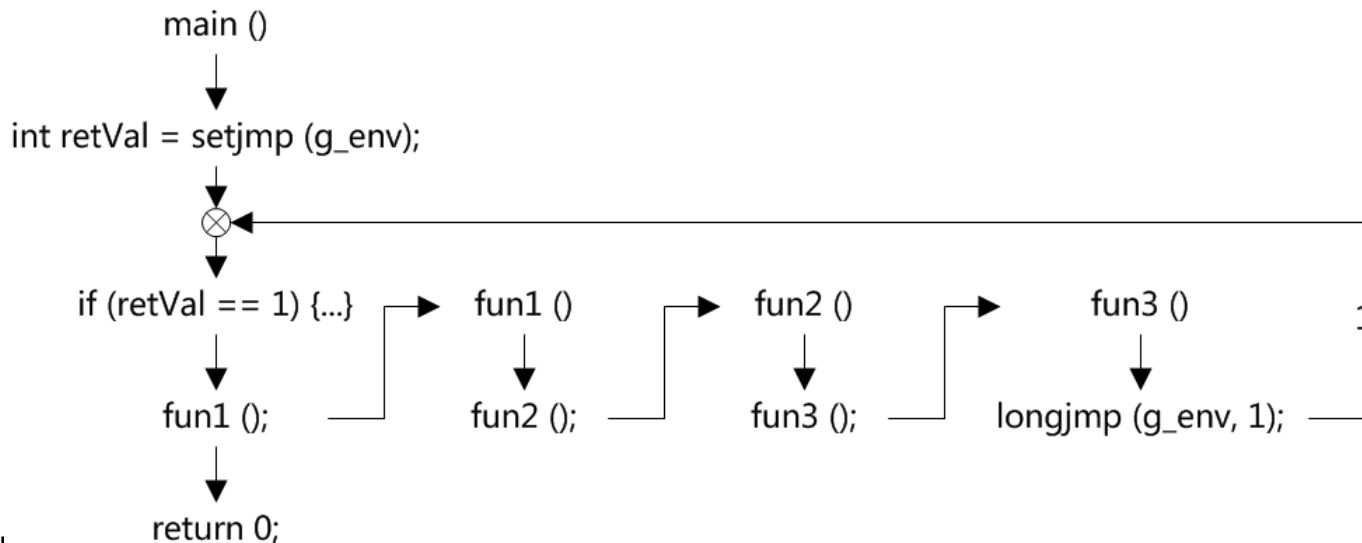
错误与错误处理

- 三种典型的错误处理机制
 - 通过返回值返回错误信息
 - ✓ 所有局部对象都能正确地被析构
 - ✓ 逐层判断，流程繁琐
 - 借助setjmp/longjmp远程跳转
 - ✓ 一步到位，流程简单
 - ✓ 某些局部对象可能因此丧失被析构的机会
 - 抛出——捕获异常对象
 - ✓ 形式上一步到位，流程简单
 - ✓ 实际上逐层析构局部对象，避免内存泄漏



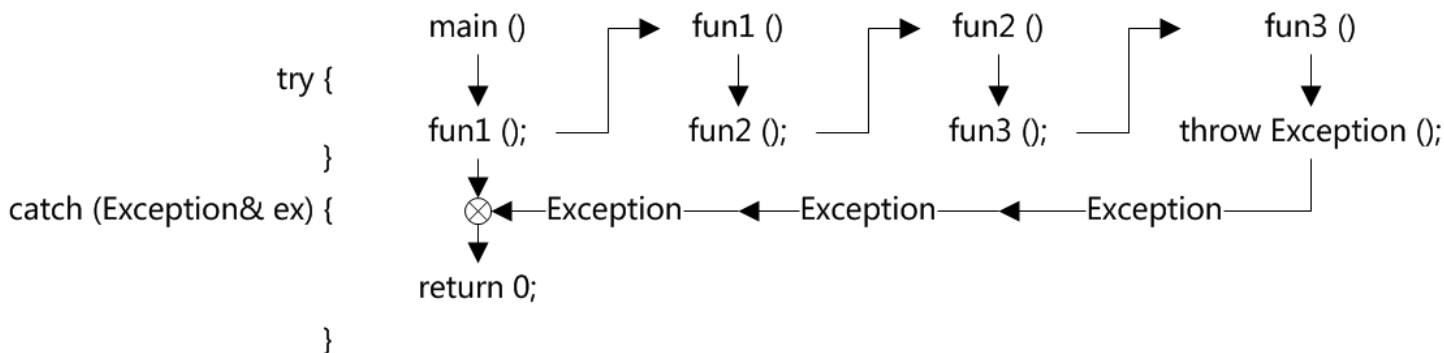
错误与错误处理

- 三种典型的错误处理机制



错误与错误处理

- 三种典型的错误处理机制



异常处理语法

- 抛出异常
 - **throw** 异常对象;
 - 可以抛出**基本类型**的对象，如：


```
throw -1;
throw "内存分配失败！";
```
 - 也可以抛出**类类型**的对象，如：


```
MemoryException ex;
throw ex;
throw MemoryException ();
```
 - 但不要抛出**局部对象的指针**，如：


```
MemoryException ex;
throw &ex; // 错误！
```



异常处理语法

- 捕获异常
 - try {
 - 可能引发异常的语句; }
 - catch (异常类型1& ex) {
 - 针对异常类型1的异常处理; }
 - catch (异常类型2& ex) {
 - 针对异常类型2的异常处理; }
 - ...
 - catch (...) {
 - 针对其它异常类型的异常处理; }



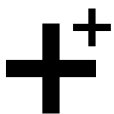
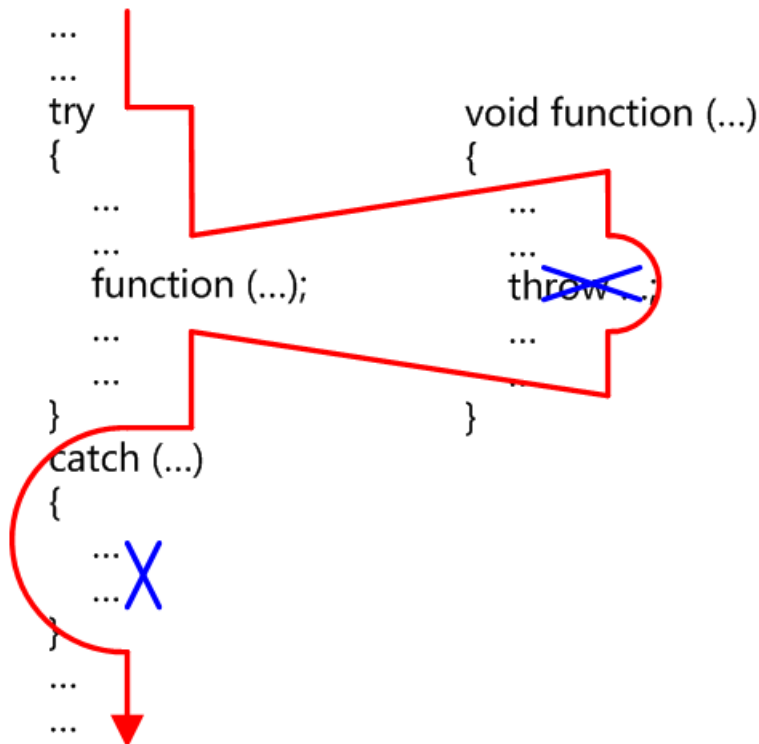
异常处理语法

- 捕获异常
 - 根据异常对象的类型自上至下**顺序匹配**，而非最优匹配，因此对子类类型异常的捕获不要放在对基类类型异常的捕获后面
 - 建议在catch子句中使用**引用**接收异常对象，避免因拷贝构造带来性能损失，或引发新的异常
- 异常对象
 - 为每一种异常定义相对应的**异常类型**
 - 推荐以**匿名临时对象**的形式抛出异常
 - 异常对象必须允许被**拷贝构造**和析构
 - 建议从**标准库异常**中派生自己的异常



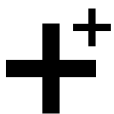
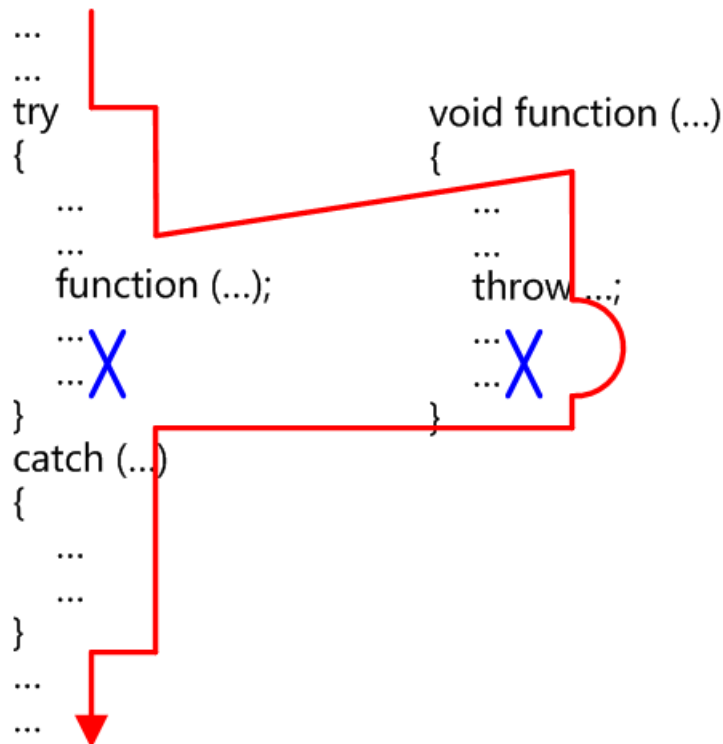
异常处理流程

- 未发生异常时的流程



异常处理流程

- 发生异常时的流程



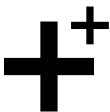
异常说明

- 异常说明是函数原型的一部分，旨在说明函数可能抛出的异常类型
 - 返回类型 函数名 (形参表) **throw** (异常类型1, 异常类型2, ...) {
函数体;
}
- 异常说明是一种承诺，承诺函数不会抛出异常说明以外的异常类型
 - 如果函数抛出了异常说明以外的异常类型，那么该异常将**无法被捕获**，并导致**进程中止**
 - `std::unexpected()` -> `std::terminate()` -> `abort()`
- **隐式**抛出异常的函数也可以列出它的异常说明



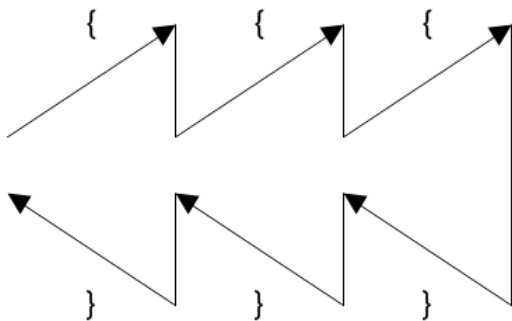
异常说明

- 异常说明可以没有也可以为空
 - 没有异常说明，表示可能抛出任何类型的异常
`void foo (void) { ... }`
 - 异常说明为空，表示不会抛出任何类型的异常
`void foo (void) throw () { ... }`
- 异常说明是虚函数覆盖的条件之一
 - 如果基类中的虚函数带有异常说明，那么子类的覆盖版本不能说明比基类版本抛出更多的异常
- 异常说明在函数的声明和定义中必须保持严格一致，否则将导致编译错误

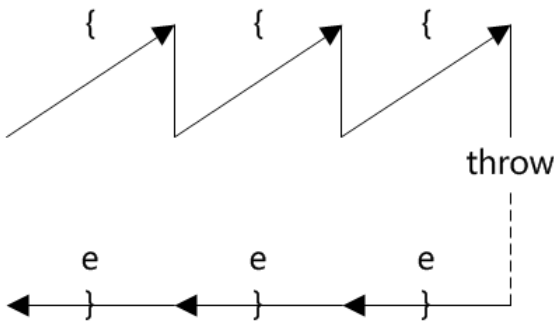


异常处理模式

- 抛出**基本类型**的异常，根据异常对象的值分别处理
- 抛出**类类型**的异常，根据异常对象的类型分别处理
- 利用类类型的异常，携带更多**诊断信息**，以便查错
- **忽略异常**，不做处理



未发生异常



发生异常



异常处理模式

- 从catch块中**继续抛出**所捕获的异常，或其它异常
- 任何未被捕获的异常，都会由std::unexpected()函数处理，缺省的处理方式就是**中止进程**
- 抛出**标准库异常**，或**标准库异常的子类**异常，以允许用户以与标准库一致的方式捕获该异常

```
class StackUnderflow :  
    public std::runtime_error {  
public:  
    explicit StackUnderflow (  
        const char* msg = "堆栈下溢!") :  
        std::runtime_error (msg) {}  
};
```



构造函数中的异常

- 构造函数可以抛出异常，某些时候还必须抛出异常
 - 构造过程中可能遇到各种错误，比如内存分配失败
 - 构造函数没有返回值，无法通过返回值通知调用者
- 构造函数抛出异常，对象将被不完整构造，而一个被不完整构造的对象，其析构函数永远不会被执行
 - 构造函数的回滚机制，可以保证所有对象形式的成员变量，在抛出异常的瞬间，都能得到正确地析构
 - 所有动态分配的资源，必须在抛出异常之前，手动释放，否则将形成内存泄漏，除非使用了智能指针



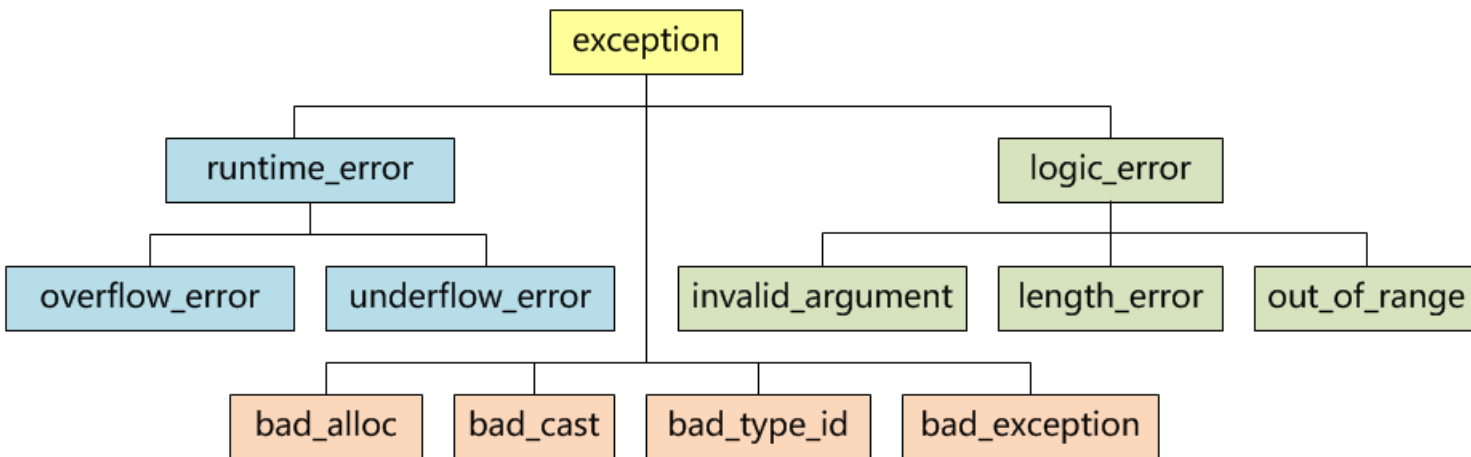
析构函数中的异常

- 不要从析构函数中**主动抛出异常**
 - 在两种情况下，析构函数会被调用
 - ✓ 正常销毁对象，离开作用域或显式delete
 - ✓ 在异常传递的**堆栈辗转开解**(stack-unwinding)过程中，由异常处理系统销毁对象
 - 对于第二种情况，异常正处于激活状态，而析构函数又抛出了异常，并试图将流程移至析构函数之外，这时C++将通过std::terminate()函数，令**进程中止**
- 对于可能引发异常的操作，**尽量内部消化**
 - try { ... } **catch (...)** { ... }



标准库异常

`#include <stdexcept>`



“

I/O流

”

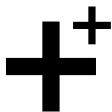
全程目标

- I/O流的基本概念
- I/O流类库
- I/O流的打开与关闭
- I/O流的状态
- 格式化I/O
- 非格式化I/O
- 二进制I/O
- 读写指针与随机访问
- 字符串流



I/O流的基本概念

- 流(Stream)
 - 字节序列形式的**数据**，有如流水一般，**从一个对象流向另一个对象**
- 输入流(Input Stream)
 - 数据字节从表示**输入设备**(如键盘、磁盘文件等)的对象流向**内存**对象，如：
`cin >> student;`
- 输出流(Output Stream)
 - 数据字节从**内存**对象流向表示**输出设备**(如显示器、打印机、磁盘文件等)的对象，如：
`cout << student;`



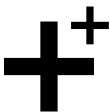
I/O流的基本概念

- 流数据(Stream Data)
 - 各种形式的**字节序列**，二进制数据、文本字符、图形图像、音频视频，等等
- 流缓冲(Stream Buffer)
 - 介于各种I/O设备和内存对象之间的**内存缓冲区**
 - 当从键盘输入时，数据首先进入键盘缓冲区，直到按下回车键，才将键盘缓冲区中的数据灌注到**输入流缓冲区**，之后再通过流操作符“>>”，进入内存对象
 - 当向显示器输出时，数据首先通过流操作符“<<”，从内存对象进入**输出流缓冲区**，直到缓冲区满或遇到换行符，才将其中的数据灌注到显示器上显示出来



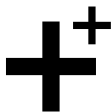
I/O流的基本概念

- 流对象(Stream Object)
 - 表示各种**输入输出设备**的对象，如键盘、显示器、打印机、磁盘文件等，因其皆以流的方式接收或提供数据，故称为流对象
 - 向下访问各种**物理设备**接口，向上与**应用程序**交互，中间维护**流缓冲区**
 - 四个预定义的标准流对象
 - ✓ **cin**：标准输入设备——键盘
 - ✓ **cout**：标准输出设备——显示器
 - ✓ **cerr**：标准错误输出设备——显示器，不带缓冲
 - ✓ **clog**：标准错误输出设备——显示器

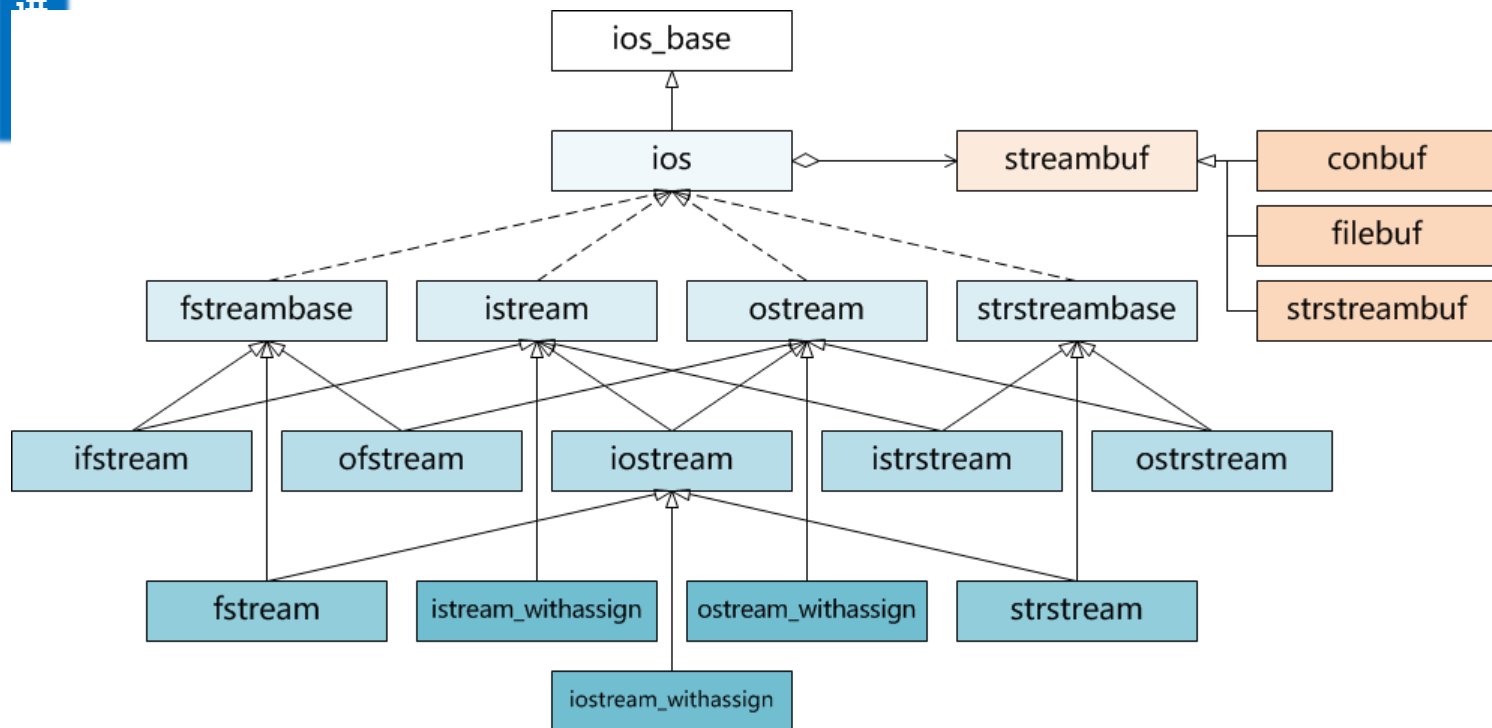


I/O流的基本概念

- 流类(Stream Class)
 - 用于实例化流对象的类
 - cin和cout分别是istream_withassign和ostream_withassign类的对象
- 流类库(Stream Class Library)
 - C++以继承的方式定义了一组流类，并将其作为标准C++库的一部分提供给用户
 - 基于流类库可以构建三种形式的流对象
 - ✓ 面向控制台的I/O流
 - ✓ 面向文件的I/O流
 - ✓ 面向内存的I/O流



I/O流类库



I/O流类库

- 流缓冲类(streambuf)
 - 实现**对流缓冲区的低级操作**，如设置缓冲区、控制缓冲区指针、从缓冲区提取字符、向缓冲区存储字符等
- 控制台流缓冲类(conbuf)
 - 从流缓冲类(streambuf)派生
 - 实现控制光标、设置颜色、定义活动窗口、清屏等功能，为面向**控制台**的输入输出提供缓冲区管理
- 文件流缓冲类(filebuf)
 - 从流缓冲类(streambuf)派生
 - 实现文件的打开关闭、读取写入、随机访问等功能，为面向**文件**的输入输出提供缓冲区管理



I/O流类库

- 字符串流缓冲类(strstreambuf)
 - 从流缓冲类(streambuf)派生
 - 提供了在内存中进行提取和插入操作的缓冲区管理
- I/O流类(ios)及其子类
 - I/O流类(ios)作为其所有子类的公共抽象虚基类，主要定义用于格式化输入输出和错误处理等成员函数
 - 在I/O流类(ios)和它的各级子类中，均含有一个指向流缓冲类(streambuf)特定子类对象的指针
 - 各种流对象的实际输入和输出功能，都是通过相应类型的流缓冲对象实现的



I/O流类库

目标	抽象	输入	输出	输入输出
抽象	ios	istream	ostream	iostream
控制台	——	istream_ withassign	ostream_ withassign	iostream_ withassign
文件	fstreambase	ifstream	ofstream	fstream
内存	strstreambase	istrstream	ostrstream	strstream

- 以上只有蓝色和红色的9个类，针对具体目标执行具体操作
- 其中蓝色的三个类已经预定义了cin/cout/cerr/clog流对象
- 实际编程中主要使用红色的6个类实现针对文件和内存的I/O
- 出于某些原因，所有I/O流类都不支持拷贝构造和拷贝赋值



I/O流类库

- `#include <iostream>`
 - `ios`、`istream`、`ostream`、`iostream`
 - `istream_withassign`、`ostream_withassign`、`iostream_withassign`
- `#include <fstream>`
 - `ifstream`、`ofstream`、`fstream`
- `#include <sstream>`
 - `istrstream`、`ostrstream`、`strstream`
- `#include <sstream>`
 - `istringstream`、`ostingstream`、`stringstream`



I/O流的打开与关闭

- 通过构造函数打开I/O流

- 打开输入流

- ```
ifstream (const char* filename, openmode mode);
```

- 打开输出流

- ```
ofstream (const char* filename, openmode mode);
```

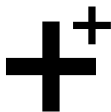
- 打开输入输出流

- ```
fstream (const char* filename, openmode mode);
```

- 通过成员函数打开I/O流

- ```
void open(const char* filename, openmode mode);
```

- 其中filename表示文件路径，mode表示打开模式



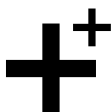
I/O流的打开与关闭

- 打开模式
 - ios::in
 - ✓ 打开文件用于**读取**，不存在则失败，存在不清空
 - ✓ 适用于ifstream(缺省)/fstream
 - ios::out
 - ✓ 打开文件用于**写入**，不存在则创建，存在则清空
 - ✓ 适用于ofstream(缺省)/fstream
 - ios::app
 - ✓ 打开文件用于**追加**，不存在则创建，存在不清空
 - ✓ 适用于ofstream/fstream



I/O流的打开与关闭

- 打开模式
 - ios::trunc
 - ✓ 打开时清空原内容
 - ✓ 适用于ofstream/fstream
 - ios::ate
 - ✓ 打开时定位文件尾
 - ✓ 适用于ifstream/ofstream/fstream
 - ios::binary
 - ✓ 以二进制模式读写
 - ✓ 适用于ifstream/ofstream/fstream



I/O流的打开与关闭

- 打开模式
 - 打开模式可以组合使用，比如
`ios::in | ios::out`
表示既读取又写入
 - 打开模式不能随意组合，比如
`ios::in | ios::trunc`
清空同时读取没有意义，但是
`ios::in | ios::out | ios::trunc`
是合理的，清空原内容，写入新内容，同时读取
- 析构函数和`close()`成员函数都可以关闭I/O流



I/O流的状态

- I/O流类内部维护当前状态，其值为以下常量的位或
 - `ios::goodbit` : 0，一切正常
 - `ios::badbit` : 1，发生致命错误
 - `ios::eofbit` : 2，遇到文件尾
 - `ios::failbit` : 4，实际读写字节数未达预期
- I/O流类支持到bool类型的隐式转换
 - 发生致命错误或遇到文件尾，返回false，否则返回true
 - 将I/O流对象直接应用到布尔上下文中，即可实现转换
- 处于致命错误或文件尾状态的流，在复位前无法工作



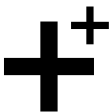
I/O流的状态成员函数

状态成员函数	说明
<code>bool ios::good (void);</code>	流可用，状态位全零，返回true
<code>bool ios::bad (void);</code>	badbit位被设置否
<code>bool ios::eof (void);</code>	eofbit位被设置否
<code>bool ios::fail (void);</code>	badbit或failbit位被设置否
<code>iostate ios::rdstate (void);</code>	获取当状态
<code>void ios::clear (iostate s = ios::goodbit);</code>	设置(复位)流状态
<code>void ios::setstate (iostate s);</code>	添加流状态



格式化I/O

- 流函数
 - I/O流类(ios)定义了一组用于控制输入输出格式的公有**成员函数**，调用这些函数可以改变I/O流对象内部的格式状态，进而影响后续输入输出的格式化方式
- 流控制符(Stream Manipulator)
 - 标准库提供了一组特殊的**全局函数**，它们有的带有参数(在iomanip头文件中声明)，有的不带参数(在iostream头文件中声明)，可被直接嵌入到输入输出表达式中，影响后续输入输出格式，称为流控制符



I/O流格式化函数

格式化函数	说明
<code>int ios::precision (int);</code>	设置浮点精度，返回原精度
<code>int ios::precision (void) const;</code>	获取浮点精度
<code>int ios::width (int);</code>	设置显示域宽，返回原域宽
<code>int ios::width (void) const;</code>	获取显示域宽
<code>char ios::fill (char);</code>	设置填充字符，返回原字符
<code>char ios::fill (void) const;</code>	获取填充字符
<code>long ios::flags (long);</code>	设置格式标志，返回原标志
<code>long ios::flags (void) const;</code>	获取格式标志



I/O流格式化函数

格式化函数	说明
<code>long ios::setf (long);</code>	添加格式标志位，返回原标志
<code>long ios::setf (long, long);</code>	添加格式标志位，返回原标志 先用第二个参数将互斥域清零
<code>long ios::unsetf (long);</code>	清除格式标志位，返回原标志

- 一般而言，对I/O流格式的改变都是**持久**的，即只要不再设置新格式，当前格式将始终保持下去
- 显示**域宽**是个例外，通过`ios::width(int)`所设置的显示域宽，只影响紧随其后的**第一次**输出，再往后的输出又恢复到默认状态



I/O流格式标志

格式标志位	互斥域	说明
ios::left	ios::adjustfield	左对齐
ios::right		右对齐
ios::internal		数值右对齐，符号左对齐
ios::dec	ios::basefield	十进制
ios::oct		八进制
ios::hex		十六进制
ios::fixed	ios::floatfield	用定点小数表示浮点数
ios::scientific		用科学计数法表示浮点数



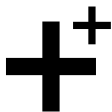
I/O流格式标志

格式标志位	说明
<code>ios::showpos</code>	正整数前面显示+号
<code>ios::showbase</code>	显示进制前缀0或0x
<code>ios::showpoint</code>	显示小数点和尾数0
<code>ios::uppercase</code>	数中字母显示为大写
<code>ios::boolalpha</code>	用字符串表示布尔值
<code>ios::unitbuf</code>	每次插入都刷流缓冲
<code>ios::skipws</code>	以空白字符作分隔符



举例

```
cout.precision (10);  
cout << sqrt (200) << endl; // 14.14213562  
cout << cout.precision () << endl; // 10  
cout.setf (ios::scientific, ios::floatfield);  
cout << sqrt (200) << endl; // 1.4142135624e+01  
cout.width (10);  
cout.fill ('-');  
cout.setf (ios::internal, ios::adjustfield);  
cout.setf (ios::showpos);  
cout << 12345 << endl; // +----12345
```



I/O流格式化控制符

格式化控制符	说明	输入	输出
left	左对齐	<input type="checkbox"/>	<input checked="" type="checkbox"/>
right	右对齐	<input type="checkbox"/>	<input checked="" type="checkbox"/>
internal	数值右对齐，符号左对齐	<input type="checkbox"/>	<input checked="" type="checkbox"/>
dec	十进制	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
oct	八进制	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
hex	十六进制	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
fixed	用定点小数表示浮点数	<input type="checkbox"/>	<input checked="" type="checkbox"/>
scientific	用科学计数法表示浮点数	<input type="checkbox"/>	<input checked="" type="checkbox"/>



I/O流格式化控制符

格式化控制符	说明	输入	输出
(no)showpos	正整数前面(不)显示+号	<input type="checkbox"/>	<input checked="" type="checkbox"/>
(no)showbase	(不)显示进制前缀0或0x	<input type="checkbox"/>	<input checked="" type="checkbox"/>
(no)showpoint	(不)显示小数点和尾数0	<input type="checkbox"/>	<input checked="" type="checkbox"/>
(no)uppercase	数中字母(不)显示为大写	<input type="checkbox"/>	<input checked="" type="checkbox"/>
(no)boolalpha	(不)用字符串表示布尔值	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
(no)unitbuf	(不)每次插入都刷流缓冲	<input type="checkbox"/>	<input checked="" type="checkbox"/>
(no)skipws	(不)以空白字符作分隔符	<input checked="" type="checkbox"/>	<input type="checkbox"/>
ws	跳过前导空白字符	<input checked="" type="checkbox"/>	<input type="checkbox"/>



I/O流格式化控制符

格式化控制符	说明	输入	输出
ends	空字符	<input type="checkbox"/>	<input checked="" type="checkbox"/>
endl	换行符，刷流缓冲	<input type="checkbox"/>	<input checked="" type="checkbox"/>
flush	刷流缓冲	<input type="checkbox"/>	<input checked="" type="checkbox"/>
setprecision (int)	设置浮点精度	<input type="checkbox"/>	<input checked="" type="checkbox"/>
setw (int)	设置显示域宽	<input type="checkbox"/>	<input checked="" type="checkbox"/>
setfill (int)	设置填充字符	<input type="checkbox"/>	<input checked="" type="checkbox"/>
setiosflags (long)	设置格式标志	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
resetiosflags (long)	清除格式标志	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>



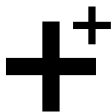
举例

```
cout << setprecision (10) << sqrt (200) << endl;  
// 14.14213562
```

```
cout << cout.precision () << endl;  
// 10
```

```
cout << scientific << sqrt (200) << endl;  
// 1.4142135624e+01
```

```
cout << setw (10) << setfill ('-') << internal  
      << showpos << 12345 << endl;  
// +----12345
```



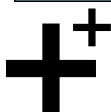
I/O流格式化控制符

- 自定义无参流控制符
 - 流类型& 流控制符 (流类型&);

```
ostream& datetime (ostream& os) {
    time_t t = time (NULL);
    struct tm* local = localtime (&t);
    return os << '['
        << local->tm_year + 1900 << '-'
        << local->tm_mon + 1 << '-'
        << local->tm_mday << ' '
        << local->tm_hour << ':'
        << local->tm_min << ':'
        << local->tm_sec << "]" << " ";
}
```

```
cout << datetime
    << "Hello, World !" << endl;
```

```
[2013-9-18 14:43:18] Hello, World !
```



I/O流格式化控制符

- 自定义单参流控制符
 - `#include <iomanip>`
 - 流类型& 流控制符 (流类型&, 参数类型);
 - `[i/o/io]manip<参数类型> 流控制符 (参数类型);`

```
template<typename T> class omanip {
public:
    omanip (ostream& (*fun) (ostream&, T), T arg) :
        m_fun (fun), m_arg (arg) {}
    friend ostream& operator<< (ostream& os,
        const omanip<T>& om) {
        return om.m_fun (os, om.m_arg);
    }
private:
    ostream& (*m_fun) (ostream&, T);
    T m_arg;
};
```

```
ostream& line (ostream& os, int no) {
    return os << "<LINE #" << no << "> ";
}
omanip<int> line (int no) {
    return omanip<int> (line, no);
}
```

```
cout << line (__LINE__)
    << "Hello, World !" << endl;
```

```
<LINE #28> Hello, World !
```

I/O流格式化控制符

- 自定义多参流控制符
 - `#include <iomanip>`
 - 流类型& 流控制符 (流类型&, 参数集);
 - `[i/o/io]manip<参数集>` 流控制符 (参数表);

```
template<typename T> class omanip {
    ...
};
```

```
class argset {
public:
    argset (const char* file, int line) :
        m_file (file), m_line (line) {}
    string m_file;
    int m_line;
};
```

```
ostream& debug (ostream& os,
    argset args) {
    return os << args.m_file << ':'
        << args.m_line << ' ';
}
omanip<argset> debug (const char* file,
    int line) {
    return omanip<argset> (debug,
        argset (file, line));
}
```

```
cout << debug (__FILE__, __LINE__)
    << "Hello, World !" << endl;
```

```
omanip3.cpp:37 Hello, World !
```



非格式化I/O

- 读取字符
 - `int istream::get (void);`
 - ✓ 成功返回读到的字符，失败或遇到文件尾返回EOF
 - `istream& istream::get (char& ch);`
 - ✓ 返回输入流本身，其在布尔上下文中的值，成功为true，失败或遇到文件尾为false
 - `istream& istream::get (char* buffer, streamsize num, char delim = '\n');`
 - ✓ 读取字符到buffer中，直到读满num-1个字符，或遇到文件尾，或遇到定界符delim(缺省为换行符)，追加结尾空字符，返回流本身
 - ✓ 如果因为遇到定界符返回，**定界符并不被读取**，读指针停在该定界符处



非格式化I/O

- 读取行
 - `istream& istream::getline (char* buffer, streamsize num, char delim = '\n');`
 - ✓ 读取字符到buffer中，直到读满num-1个字符，或遇到文件尾，或遇到定界符delim(缺省为换行符)，追加结尾空字符，返回流本身
 - ✓ 如果因为遇到定界符返回，**定界符被读取并丢弃**，读指针停在**该定界符的下一个位置**
- 放回字符
 - `istream& istream::putback (char ch);`
 - ✓ 将字符ch放回输入流，返回流本身



非格式化I/O

- 窥视字符
 - `int istream::peek (void);`
 - ✓ 返回但不读取读指针当前位置处的字符，失败或遇到文件尾返回EOF
- 忽略字符
 - `istream& istream::ignore (streamsize num = 1, int delim = EOF);`
 - ✓ 忽略输入流中的num(缺省1)个字符，或遇到定界符delim(缺省到文件尾)，返回流本身
 - ✓ 如果因为遇到定界符返回，定界符并不被忽略，读指针停在该定界符处



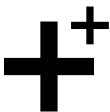
非格式化I/O

- 获取读长度
 - `streamsize istream::gcount (void);`
 - ✓ 返回最后一次从输入流中读取的字节数
- 写入字符
 - `ostream& ostream::put (char ch);`
 - ✓ 一次向输出流写入一个字符，返回流本身
- 刷输出流
 - `ostream& ostream::flush (void);`
 - ✓ 将输出流缓冲区中的数据刷到设备上，返回流本身



二进制I/O

- 读取二进制数据
 - `istream& istream::read (char* buffer, streamsize num);`
 - ✓ 从输入流中读取num个字节到缓冲区buffer中
 - ✓ 返回流本身，其在布尔上下文中的值，成功(读满)为true，失败(没读满)为false
 - ✓ 如果没读满num个字节，函数就返回了，比如遇到文件尾，最后一次读到缓冲区buffer中的字节数，可以通过`istream::gcount()`函数获得



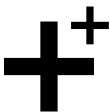
二进制I/O

- 写入二进制数据
 - `ostream& ostream::write (const char* buffer, streamsize num);`
 - ✓ 将缓冲区buffer中的num个字节写入到输出流中
 - ✓ 返回流本身，其在布尔上下文中的值，成功(写满)为true，失败(没写满)为false



读写指针与随机访问

- 设置读/写指针位置
 - `istream& istream::seekg (off_type offset, ios::seekdir origin);`
`ostream& ostream::seekp (off_type offset, ios::seekdir origin);`
 - ✓ `origin`表示偏移量`offset`的起点
 - `ios::beg`：从文件的第一个字节
 - `ios::cur`：从文件的当前位置
 - `ios::end`：从文件最后一个字节的下一个位置
 - ✓ `offset`为负/正表示向文件头/尾的方向偏移
 - ✓ 读/写指针被移到文件头之前或文件尾之后，则失败



读写指针与随机访问

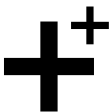
- 获取读/写指针位置
 - `pos_type istream::tellg (void);`
`pos_type ostream::tellp (void);`
 - ✓ 返回读/写指针当前位置相对于文件头的字节偏移量
- `iostream`的子类，如`fstream`
 - 同时拥有针对读/写指针位置的两套设置/获取函数
 - 理论上应该拥有两个相互独立的读/写指针
 - 多数编译器仍然使用一个指针记录文件当前位置
 - 建议读取时用`seekg/tellg`，写入时用`seekp/tellp`



字符串流

- 输入字符串流

```
#include <sstream>
string is ("1234 56.78 ABCD");
// istreamstringstream iss;
// iss.str (is);
istreamstringstream iss (is);
int i;
double d;
string s;
iss >> i >> d >> s;
```



字符串流

- 输出字符串流

```
#include <sstream>

ostringstream oss;

oss << 1234 << ' ' << 56.78 << ' ' << "ABCD";

string os = oss.str ();
```



练习时间

利用二进制I/O，实现对任意文件的异或加解密程序

密钥=随机数%256

密文=明文^密钥

明文=密文^密钥



推荐书目

- 入门
 - C++ 程序设计原理与实践
Bjarne Stroustrup
 - C++ Primer
Stanley B. Lippman
- 进阶
 - Effective C++: 改善程序与设计的55个具体做法
Scott Meyers
 - More Effective C++: 35个改善编程与设计的有效方法
Scott Meyers



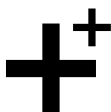
推荐书目

- 深研
 - 深度探索C++对象模型
Stanley B. Lippman
 - 设计模式：可复用面向对象软件的基础
Erich Gamma, Richard Helm, ...
- 拓展
 - 深入理解C++11: C++11新特性解析与应用
IBM XL编译器中国开发团队
 - Boost程序库完全开发指南：深入C++ “准” 标准库
罗剑锋



推荐书目

- 休闲
 - C++语言99个常见编程错误
Stephen C. Dewhurst
 - C++语言的设计与演化
Bjarne Stroustrup
 - 大话设计模式
程杰



“

再见

”