一、C++程序基本常识

1.编译方式

1. gcc xx.cpp/xx.c -lstdc++
2. g++ xx.cpp

2.文件扩展名：

add.cpp(最常用)/add.cc/add.C/add.cxx

3.头文件

#include <iostream> //in,out,stream/输入,输出流

#include <cstdio> <==等价==> #include<stdio.h>

4.输入和输出

用cin表示标准输入，用cout表示标准输出

"<<":插入运算符

printf("%d,%lf,%s\n",a,d,str);

cout << a << ',' << d << ',' << str << endl; //红字为打印’\n’

">>":提取运算符

scanf("%d%d",&number,&number2);

cin >> number >> number2;

""表示输出字符串，''输出单个字符

<< endl表示插入'\n'

<< flush表示刷新输出缓冲区

==================================

二、名字空间

1.为什么需要名字空间？

1)划分逻辑单元，改善程序的可读性和可维护性。

2)避免名字冲突。

举例："std::"标准名字空间

2.名字空间的定义和声明

1)定义

namespace 名字空间名 { ... }

2)合并

同名名字空间，无论是在一个源文件中，还是分布于不同的源文件中，编译器都会把它们合并到一个名字空间中。

3)声明和定义可以分开

定义语句必须带有作用域限定。(类当中构造函数和成员函数同理)

namespace ns {

extern int x;

void foo (void);

}

int ns::x = 100;

void ns::foo (void) { ... }

3.怎样使用名字空间

1)作用域限定符

cout << ns::x << endl;

ns::foo ();

1. 名字空间指令

指定名字空间中的成员在之后的代码中都可见，相当于全局变量可见

using namespace ns;

cout << x << endl;

foo ();

只有在不发生名字冲突的前提下，才可以使用这种语法。

1. 名字空间声明

将名子空间的特定成员引入当前作用域，相当于定义的局部变量

using ns::x;

cout << x << endl;

只有在不发生名字冲突的前提下，才可以使用这种语法。

4.无名(匿名)名字空间

1)不属于任何有名名字空间的标识符，会被编译器自动地放入一个无名名字空间中。

2)无名名字空间中成员，无需声明，默认为可见

3)需要区别访问时：直接在该成员名前面使用“::”。

5.名字空间嵌套和别名

1)多层名字空间

namespace ns1 {

namespace ns2 {

namespace ns3 {

int name;

void foo (void) { ... }

}

}

}

2)访问内层名字空间的成员

ns1::ns2::ns3::name

using namespace ns1::ns2::ns3;

foo ();

3)名字空间别名

namespace ns123 = ns1::ns2::ns3;

cout << ns123::x << endl;

ns123::foo ();

==================================

三、结构、联合和枚举

1.C++中的结构、联合和枚举型变量在定义时可以省略struct，union和enum关键字。

2.C++的结构中可以包含成员函数，在成员函数中可以直接访问同一个结构中的成员变量。结构的成员函数必须通过结构类型的变量或者指针加上"."或者"->"运算符调用。

3.C++中可以使用匿名联合，将其中包含的变量约束为以内存共享的方式布局。

4.C++中的枚举是独立的数据类型，不能从整型隐式转换(不可赋值)，但可以被隐式转换为整型。

================================

四、字符串(类型：string，底层通过malloc分配一定大小内存)

1.string类型的对象

在C++中，字符串表示为一个string类型的对象，所以需要实例化

1)实例化

在栈中隐式实例化：

A. string a;

B. string b ("字符串内容");

在栈中显式实例化：

C. string s = string("Hello,World!");

在堆中显式实例化：

D.string \*s = new string("Hello,World!");

2)C字符串与C++字符串转换方式：

C -> C++: 隐式转换过程(通过类型转换构造函数)

string s = "abc";

C++ -> C: 调用成员函数string::c\_str()

char const\* pc = s.c\_str();

2.字符串运算

1)赋值/拷贝(strcpy)：=

2)拼接(同下)：+

3)拼接赋值(strcat)：+=

4)大小关系(strcmp)：>/>=/</<=/==/!=(逻辑运算符)

5)字符串下标访问：[] 不检查下标越界，有可能报错或者不报错

3.相关成员函数

1)string::size()/string::length()

2)resize(调整大小)/clear(清空字符串)/empty(判断是否为空)

3)C++标准模板函数swap()，交换两个变量

===============================

五、布尔类型

1.bool(1个字节)：true/false

2.任何基本类型都可以被隐式转换为bool类型，内存非零即true，零即false。

bool b = false;

cout << b << endl; //0

//插入流控制符boolalpha,以字符串形式打印布尔值，并且后面代码也有效

cout << boolalpha << b << endl; //false

===============================

一、函数重载

1.重载条件：

（1）作用域相同

（2）函数名相同

（3）参数表不同(包括顺序)

（4）相同类型的引用和非引用不构成重载

（5）指针和引用的常属性可以构成重载(原理与常函数重载一样，非指针和引用的常属性不行)。nm a.out

2.重载解析：类型匹配，数据安全，最小工作量。

举例：

(1)工作量(数据转换)相同：double => int和double => float

(2)匹配先后顺序：完全匹配>常量装换>升级转换>过分升级转换=降级转换>省略号匹配

ps：1.升级转换：数据类型变大(数据安全)

2.隐式转换工作量相同时，编译器报歧义

1. 函数指针：根据函数指针的类型(返回值及参数表)选择重载版本。

4.重载原理：通过C++换名实现(函数名按顺序加上形参类型缩写)，用命令nm a.out查看。

5.extern "C"关键字：抑制C++编译器的换名。

-C调C++，在C++中：extern “C” + 函数声明 (抑制C++换名)

-C++调C，在C++中：extern “C” {C头文件} (抑制C头文件)

===============================

二、缺省参数

1.可以为函数的参数指定缺省值，如果调用该函数不指定部分或全部实参，编译器会用对应参数的缺省值作为实参。

2.函数的缺省参数必须靠后。

3.函数的缺省参数是在编译阶段解决的，因此只能用常量、常量表达式或者全局变量等非局部化数值作为缺省参数

(1)不能用局部变量作为函数参数的缺省值。

void foo (int a, int b = a); //error

(2)如果需要将函数的声明与定义分开，函数的缺省参数只能出现在函数的声明部分。

4.使用缺省参数时避免与重载发生冲突。

-------------------------------------

补充(了解)：

哑元参数定义：只定义类型没有变量名的形参称为哑元参数

例：void foo(int a，int/\*哑元\*/)

1. 哑元参数接收的参数不起作用
2. 作用：(1)为了升级算法库后，兼容旧代码 (2)运算符重载

==============================

1. 内联函数 (可以代替宏)(编译时的优化策略)

inline void foo (void) { ... }

1.用函数被编译完以后的二进制模块替换代码中对该函数的调用指令。

效果：减少调用函数时产生的跳转，代码执行流畅；生成的可执行文件较大。

与宏定义的区别：宏定义发生在预处理，内联发生在编译时

2.通过牺牲空间换取时间的优化策略。

3.对于频繁调用的简单函数适合通过内联优化，而对于稀少调用的复杂函数内联优化的效果并不明显。若函数在类或结构体的内部直接定义，则该函数被自动优化为内联函数。

4.适合优化的，编译器会自行隐式内联。

5.加上inline关键字表示显示内联。但仅表示一种建议，具体是否执行有编译器决定。

6.带有递归调用或动态绑定特性的函数不能内联。

---------------------------------------------

函数发生跳转，不但有内存开销，还有时间开销(嵌入式)

inline关键字的理解：(编写程序时，尽量减少程序因多层循环或者调用函数产生的跳转)

1.为了减少程序中调用函数产生的跳转

2.使用循环也会产生跳转

例如下面两种循环：

循环次数一样，但是第二种执行时，由于外层循环少，跳转少，执行效率高

for(int i = 0; i<1000000; i++){ //跳转次数(1+1000000) x 2

for(int j = 0; j<1000; j++){

//...

}

}

for(int i = 0; i<1000; i++){ //跳转次数(1+1000) x 2

for(int j = 0; j<1000000; j++){

//...

}

}

===============================

四、动态内存分配 (对象创建操作符/对象销毁操作符)

1.沿用C库函数：malloc/calloc/realloc/free

2.新的运算符：new/new[]/delete/delete[]，因为在给对象分配空间时，new可以调用构造函数

1)凡是new出来的数组一定要用delete[]释放。

2)new一定要和delete配对，malloc一定要和free配对。

3)野指针和空指针。

指针前的\*是解引用运算符

int \*pa = new int(100);

//new数组同时初始化,C++11中支持

int \*pa = new int[10]{0,1,2,3,4,5,6,7,8,9};

===============================

补充：(尤其是对象数组)

1. 某些C++实现,用new[]操作符动态分配数组时，会在数组首元素前面多分配4或8个字节，用以存放数组的长度
2. new[]操作符所返回的地址是数组首元素的地址，而非所分配内存的起始地址
3. 如果将new[]操作符返回的地址直接交给delete操作符，将导致无效指针(invalid pointer)异常
4. delete[]操作符会将交给它的地址向低地址方向偏移4或8个字节，避免导致内存泄漏

重析构(double free)：

•不能通过delete操作符释放已释放过的内存

int\* p = new int;

delete p;

delete p; // 核心转储

标准库检测到重析构异常后将进程杀死，并转储进程映像

•delete野指针后果未定义，可能会出错，delete空指针安全

int\* p = new int;

delete p;

p = NULL;

delete p; // 什么也不做

==============================================

一、引用(Reference)

1.定义：引用即别名。前提是存在实体

注意：

1. 引用必须在定义的同时初始化，不允许先定义再赋值

int a = 10;

int& b = a; //ok

int& b; //error

1. 引用不能为空

int& r = NULL; //error

3.但“野引用”或称“悬空引用”确实是存在的

int& r = \*new int(1); //熟悉这个写法

++r;

cout << r << endl; // 2

delete &r;

++r; // 未定义

解引用一个野引用，就跟解引用一个野指针一样，其结果将是未定义的，可能导致崩溃(段错误)，可能意外地修改了其它有效堆内存中的数据，也可能什么也没有发生而且结果还很正确，但没有人能保证到底会是哪种结果

4.引用不能更换目标

int &const c = b; //这种写法不存在

引用一经定义就不能再引用别的变量

int a = 10,b = 20;

int& c = a;

c = b; //将b的值赋给a，而非令c引用b

5.‘&’放在引用的类型后面

const char\* s = “hello”;

const char\*& p = s; //引用常量指针

=======================

补充：

1.指针和引用的常识：

const char\* p; <==> char const\* p;

const int& c = b; <==> int const& c = b;

char const\* p; //常量指针，指向可改

char \*const p; //指针常量，内容可改

2.关于左值和右值概念：

1）左值：可以放在赋值运算符的左边，可以被修改，可以被取地址。

常见左值(表示存储区)：

1. 变量
2. 前缀表达式的值(++a/--a)
3. //解引用表达式\*p（常属性的变量，不可修改）
4. 赋值表达式的值
5. 左值引用型函数返回值

举例:

A.前缀表达式的值

++i = 10; //前缀表达式为左值

++++i; //ok

B.赋值表达式的值

(a = b) = 30;

C.引用型函数返回值(常引用除外)

int& foo (int& n) { return n; }

foo(a) = 100; //ok

2）右值：只能放在赋值运算符右边，不可以被修改，不可以被取地址。

常见右值：

1. 字面值常量：10，”hello”
2. 常引用型函数返回值：(在类型前加了const，效果逼近右值)
3. 临时变量：(即将亡右值)

将亡右值包括以下三种：

A.大部分表达式的值(不包括&a，二级指针)

i++ = 20; //error,后缀表达式为右值

i++++; //error

(a + b) = 10; //error

B.(非引用型)函数返回值

int func(void){

int a = 100;

return a; //将a保存到一个临时变量中(temp=a)

}

int res = func();//int res = temp

C.隐式类型转换

const int& a = 'A'; //65

---------------------------

注意：函数返回的类类型变量是临时变量(右值)，但却可以直接调用成员函数或者类型转换构造函数修改,使用起来和左值相似。

class A{

int m\_data;

A(int data = 0):m\_data(data){}

};

A foo(void){}

int main(){

A a;

foo() = a; //编译器理解为调用成员函数operator=

Foo() = 100; //调用类型转换构造函数

}

=========================================================

1. 常量引用（只读引用,万能引用）

普通引用只能引用左值，常量引用既能引用左值，又能引用右值

1.引用常量(纯右值)

const int& r = 100; //常引用表示纯右值

const char\* s =“hello”;

引用临时变量(将亡右值)

//常引用可以延长临时变量(将亡右值)寿命,

2.大部分表达式

int a = 10, b = 20;

const int& c = a+b; //常引用表示将亡右值

3.非引用型函数返回值

//int& res = foo(); //error

const int& res = foo();

4.引用别的数据类型(隐式转换)

char ch = 'A';

//int& rch = ch; //error

const int& rch = ch;

7.引用型函数参数(引用不占内存空间)

1)通过引用型参数输出数据；

2)引用型参数避免按值传参过程中内存复制的开销(提高传参效率)，对于有复杂数据结构的参数类型而言意义非常；

======================

\*\*\*\*\*\*\*\*\*\*

函数传参和返回过程：

int& foo (int& n) { return n; }

foo(a);

传参过程：int& n = a; 返回值过程：int& temp = n;

int& bar (int\* n) { return \*n; }

bar(&a);

传参过程：int\* n = &a; 返回值过程：int& temp = (\*(&a));

\*\*\*\*\*\*\*\*\*\*

======================

1. 引用型返回值

1)返回左值；

2)避免返回值的过程中内存复制的开销；

3)不能返回局部变量的引用；

因为：局部变量的内存会在函数返回以后被释放，并标记为可以被其它变量使用(这一点类似指针)

返回局部变量引用或地址(危险)：

int& foo (void) { int n = 123; return n; } //危险

int& hum (int n) { return n; } //危险

int\* bar (void) { int n = 456; return &n; } //危险

int\* hum (int n) { return &n; } //危险

int\*\* bar (int\* n) { return &n; } //危险，局部指针的地址

int\*& bar (int\* n) { return n; } //危险，局部指针的引用

int& foo (int& n) { return n; }

int\* foo (int\* n) { return n; }

int& bar (int\* n) { return \*n; } //\*n表示的变量，函数结束后还在

int\* bar (int& n) { return &n; }

4)如果函数返回值需要右值属性，可以返回一个常左值引用，使用效果接近右值

int const& foo (int& n) { return n; }

//以下写法勉强可以(编译警告)，但没有必要使用，因为指针本身空间就不大

int\* const& foo (int& n) { return &n; } //常引用返回将亡右值

===============================

\*\*\*9.总结引用和指针的区别

1)指针可以用NULL初始化，引用不可以

~~int& r = NULL;~~ // 错误

1. 指针可以不初始化，指针的指向可以修改(除了指针常量)；引用必须初始化，初始化以后不能更换目标。

~~int& r;~~ //error

int& r = a; // r -> a

r = b; //把b的值赋给了a

(引用只是一个别名，不是实体，运算时直接用原名替换；引用不占内存，没有地址空间)

3)可以定义二级指针，但是不可以定义二级引用

int a;

int& r = a;

~~int&& rr = r;~~ //错误

&& - C++98中非法

C++11中合法，但是它并不表示二级引用，而是表示右值引用

int&& rr = 100;

4)可以定义引用指针的引用，但是不可以定义指向引用的指针

int\* p;

int\*& r = p;

int a;

int& r = a;

~~int&\* p = &r;~~ //错误

5)可以定义指针数组，但是不可以定义引用数组，却可以定义数组引用(引用数组的引用)

int\* pa[5];

~~int& ra[5] = { ... };~~ //错误

int a[5];

int (&ar)[5] = a; //数组引用

int (\*ap)[5] = &a; //数组指针

6)函数指针和函数引用语法基本一样。

void func(int a,double d){...}

int main(void)

{

//函数指针

void (\*pfunc)(int,double) = func;

pfunc(100,3.14);

//函数引用

void (&rfunc)(int,double) = func;

rfunc(200,4.13);

}

1. 引用的本质就是指针，但是建议尽量使用引用而不是指针！

11.同一个程序中，避免指针和引用混用。

原因：如果返回引用的函数调用了返回指针的函数，就会出现返回临时变量(指针)引用的情况。

=======================================================

C++类型转换

1. 隐式类型转换

---》发生的情况有3种：1.赋值、2.传参、3.函数返回值

1. 基本类型之间可以互相隐式转换
2. 任意指针 ---> viod \*类型指针可行，反之不行

int\* i = NULL;

void\* p = i; //ok

void\* pv = ...;

int\* pi = pv; //error

int\* pi = (int\*)pv;

1. 显式类型转换

C++中支持的两种强制类型转换：

int i;

char c = (char)i; //C的强制转换

char c = char(i); //C++风格的强制转换

C++中新增四种转换，会自动检查错误：

1.静态类型转换(常用)

static\_cast: 隐式类型转换的逆转换、自定义类型转换

1. 编译器会检查,不兼容时会报错

char\* pc = "hello";

int\* pi = static\_cast<int\*> (pc); //error

1. 原类型和目标类型只要在一个方向上可以做隐式类型转换，那么在两个方向上就都可以做静态类型转换：反转

A -隐式-> B

A <-静态-> B

------------------------------

2.dynamic\_cast动态类型转换运算符

1）用于在构成多态继承的父子类中，将基类类型的指针或引用向下造型。

补充：

A.如果没有构成多态继承，无法使用该转换。

B.多态继承：基类包含至少一个虚函数。

举例：

class X{virtual void foo(){}};

class Y:public X{void foo(){}};

class Z:public X{void foo(){}};

X\* px = new Y;

Z\* pz = static\_cast<Z\*>(px); //ok，但不合理

Z\* pz = dynamic\_cast<Z\*>(px); //error

1. dynamic\_cast在转换过程中，会检查目标对象和期望转换的对象是否一致，如果一致则转换成功，否则失败；如果转换的是指针，失败返回NULL，如果转换是引用，失败将会抛出异常"bad\_cast"。

-------------------------------------

3.常量类型转换(去常转换)

const\_cast: 同类型变量，去除指针或引用的常属性

4.重解释类型转换(重新解释数据含义，最接近强制类型转换)

reinterpret\_cast: 适用场景：

1. 任意类型指针或引用之间的转换

2.指针和整型数之间的转换

========================================

补充：

寄存器知识：计算机有寄存器，所有数据使用都要先从内存读入寄存器中，在寄存器里处理

const 编译时，编译器会优化，直接取寄存器里先前存放的副本，这样读取速度快

volatile 表示变量易变，告诉编译器停止优化，每次都内存中重新读取数据

这种情况一般只会在嵌入式传感器中碰到(变量改变由硬件状态引起)

===============================

const常量关键字表示：存放的变量内存只读

注意：

1. 定义的常量必须初始化，因为之后不可修改
2. 只能定义常量指针指向const修饰的常量
3. 可以使用去常转换，使普通指针指向常量，并修改

例：

const int ci = 100;

//多了一个可以修改的途径

//int\* pci = &ci; //error

const int\* pci = &ci; //ok

int\* pci = const\_cast<int\*>(&ci);

============================================

类

1.类的一般形式

class/struct 类名：继承表{

public/private/protected:

类名 (构造形参表){ ... } // 构造函数

~类名(void){...} // 析构函数

返回类型 函数名(形参表){ ... } // 成员函数

数据类型 变量名; // 成员变量

};

封装一个类，要通过数据抽象的方法。

数据抽象：在描述对象时，把细节东西剥离出去，只考虑一般性的、有规律性的、统一的东西。

2.访问控制属性

公有(public) - 谁都能访问

私有(private) - 只有自己可以访问

保护(protected) – 只有自己和自己的子类可以访问(继承)

注:class的缺省访问控制属性为私有(private),struct的缺省访问控制属性为公有(public)

3.构造函数(Constructor)

1)构造函数特点：没有返回类型，函数名必须和类名完全相同。

类名(构造形参表):[初始化表]{...}

2)调用构造函数的时机：每次创建对象时只被系统自动调用一次。

1. 构造函数作用：设置对象初始状态，分配资源，为对象的功能做准备。

================================================

1. 执行过程：基类子对象->成员子对象->子类构造代码

成员子对象：类类型的成员对象

基类子对象：子类对象包含的基类部分成员

对象创建过程：分配内存->构造基类->构造成员->执行构造函数代码

基类：按继承顺序，从左至右，依次构造

成员：按声明顺序，从上至下，依次构造

==================================================

4.对象的实例化：

1）对象定义语句调用构造函数

User user("张飞",25); //隐式构造，User(&user(this),"张飞",25)调用构造函数;

User user = User("张飞",25); //显式构造

//栈中的对象数组

User user[3] = {User("张飞",25),User("张飞",25),User("张飞",25)};

2）new操作符调用构造函数

User\* user = new User("张飞",25); //堆中的单个对象

//堆中的对象数组

User\* user = new User[3]{User("张飞",25),User("张飞",25),User("张飞",25)};

注意：

1.一创建对象就自动调用构造函数： 例如：A a1;

A.栈对象，编译器生成调用构造函数的代码；

B.堆对象，new运算符负责调用构造函数。

2.如果构造函数不传实参(调用无参构造)，只有两种写法

类名 对象;

类名 对象 = 类名();

~~类名 对象();~~ //error,编译器会理解为函数声明

注意：

1.new数组同时初始化,C++11中支持

int \*pa = new int[10]{0,1,2,3,4,5,6,7,8,9};

-------------------------------------

一、构造函数和初始化表

1.构造函数可以重载

2.几个特殊的构造函数

1)缺省构造函数(或叫无参构造函数)

A.表示对象缺省状态；

B.可以不带参数，也可以带有缺省参数；

C.编译器在未定义任何构造函数时会提供缺省构造函数。

1. 类型转换构造函数(单参构造函数)

[explicit] 目标类型(const 源类型& src){...}

1. 用一种类型的对象作为源对象，类型转换构造出另一种类型的对象(支持隐式转换)。

B.explicit关键字：如果加上，表示强制这种转换显示进行。

3)拷贝构造函数

形如：A (A const& that);

不可以写成：A (A that);，原因：拷贝对象以值传递的方式传递给形参时，本生就会发生拷贝，这样会递归调用拷贝构造函数，形成死循环。

--------------------------------------

A.用一种类型的对象作为源对象构造出同类型的另一个对象――对象克隆。

B.编译器在未定义拷贝构造函数时会提供缺省拷贝构造函数。

--》对基本类型的成员变量，按字节复制

--》对类类型的成员变量，调用相应类型的拷贝构造函数(顺序和构造函数一样)

C.调用拷贝构造函数的时机

a)直接构造对象副本

A a1;

A a2 (a1); //隐式拷贝，A(&a2,a1)调用拷贝构造;

A a3 = a2; //显示拷贝

b)以值的形式传递形参

c)从函数中返回对象(发生2次，有时会被编译器优化掉)

注释：返回给中间量temp时要拷贝，temp再拷贝给等待接收的变量

(接收的变量引用了函数返回值，去掉优化 -fno-elide-constructors)

\*\*\*当自定义一个拷贝构造函数时，类里就只有唯一构造函数

1. 初始化表 (效率比普通方式高出一点，可以忽略)

1)语法形式

class A{

A(int i=0):m\_data(i){}

int m\_data;

};

2)必须使用初始化表的场合

A.成员子对象(或者基类子对象)没有无参构造函数。

注解：先调用成员子对象的构造函数，如果子对象没有无参构造，就不能先创建再给成员子对象赋值，而必须创建的同时，用初始化表给子对象初始化。

B.包含常量或者引用类型的成员变量。("const"或"引用&")

1. 成员变量的初始化顺序由其在类中被声明的顺序决定，而与其在初始化表中出现的顺序无关――尽量避免成员变量在初始化过程中的相互依赖，如果避免不了，用注释说明。

参见代码：04initlist.cpp

提议：

一般情况下使用初始化表，无法避免成员变量在初始化过程中的互相依赖可以使用普通方式。

===============================

二、this指针与常函数

1.this指针

1)本质：类的非静态成员函数都隐藏着一个该类类型的指针参数，该指针名为this。

--》对于普通成员函数，this指针指向调用该函数的对象(调用对象)

--》对于构造和析构函数，this指针指向正在被构造或析构的对象

在成员函数中对成员变量和其它成员函数的访问，实际上都是通过this指针实现的。

---------------------------------

2)必须使用的场景：

--》区分成员变量和同名的局部或全局变量。(区分作用域)

以下三条总结为：在成员函数内部访问其调用对象本身

--》从成员函数内部给值结果参数形式传递调用对象自身。

--》从成员函数内部向调用者返回调用对象的自引用或者自拷贝。

--》从类的内部销毁堆区的对象自身（对象自销毁）

--》this指针作为函数的实参，实现对象之间的交互(了解)(句意不理解，用法参见05this.cpp)

---------------------------------------

1. 常函数

1)在一个成员函数的参数表后面加上const，这个成员函数就称为常函数。

形式：返回类型 函数名(形参表)const{函数体}

注意：const关键字修饰该成员函数的this指针。

2)与其他原理融合的特点：

A.mutable关键字：修饰的成员变量会在使用时去除常属性(单个成员变量去常属性)

B.常对象只能调用常函数，同理常函数(常函数会把this指针常量化)中只能调用常函数。

注解：1.道理等同于，必须拿常量指针指向常量，否则会扩大访问权限，编译器会报错。

2.常对象包含常指针和常引用

class A {

void foo (void) {}

void bar (void) const {}

void hum (void) {

this->foo (); //非->非

this->bar (); //非->常

}

void fun (void) const {

this->foo (); //常->非，报错

const\_cast<A\*>(this)->foo (); //非->非

this->bar (); //常->常

}

};

C.同一个类中，签名相同的成员函数常版本和非常版本构成重载关系。常对象选择常版本，非常对象选择非常版本，但是如果没有非常版本，非常对象也可以选择常版本。

建议：成员函数返回不是左值，没有对成员变量做修改，都要加上const关键字(修饰this指针)

--------------------------------------

三、析构函数(Destructor)

1.原型

~类名(void){ ...}

析构函数没有参数，不能重载。

2.调用析构函数的时机

1)栈内对象：右“}”调用，右“}”叫作用域终止运算符，用于释放作用域内的局部变量

2)堆内对象：delete/delete[]调用。

3.缺省析构函数(编译器提供)

4.功能

1）释放资源

2）善后处理注解：析构函数的功能不仅限于释放资源，它可以执行任何作为类的设计者希望在最后一次使用对象之后执行的动作

举例：当类中有指针型成员，即创建对象时需要开辟动态内存时，不但需要重写拷贝构造，还需要重写析构函数，防止内存泄漏。

5.对象销毁过程：执行析构代码->析构成员->析构基类->释放内存

基类：按继承顺序，从右至左，依次析构

成员：按声明顺序，从下至上，依次析构

===============================

一、拷贝构造和拷贝赋值

1.浅拷贝和深拷贝

1)浅拷贝：针对类的指针型成员变量，在获得对象副本时只复制指针本身，而没有复制指针所指内容的拷贝构造函数。

缺点：导致在对象被析构时引发double free异常，同时导致多个对象因引用相同的资源而发生耦合。

2)深拷贝：不复制指针所指向的地址，而是复制指针所指向内容的拷贝构造函数。

1. 缺省的拷贝构造函数和拷贝赋值运算符函数

1)如果没有为一个类显式定义拷贝构造函数和拷贝赋值运算符函数，编译器会为其提供缺省版本，但是缺省版本只做浅拷贝。

2)如果一个类拥有指针形式的成员变量，并用该指针维护动态分配的内存资源，那么就应该为该类定义支持深拷贝的拷贝构造函数和拷贝赋值运算符函数。

-----------------------------

3.实现深拷贝(重点)

1)自定义拷贝构造：A::A(A const& that){}

过程：开辟新内存复制指针指向的内容

2)自定义拷贝赋值运算符函数：

形如：A::A& operator=(A const& that){}

过程先后顺序：

1)防止给自己赋值

2)分配新内存，复制新内容

3)释放旧内存

4)返回自引用

注解：

1. 因为指针指向的内存大小不一定一样，为了防止内存泄漏或者越界访问所以要分配新内存。

2.确保new成功后(new失败后会报异常并跳转)，再释放旧内存，防止原数据先释放

3.学会代码复用：复用拷贝构造和析构函数(拷贝一个临时变量，交换指针类型成员的指向)

===================================================

补充：当需要自定义含有基类或成员子对象的子类深拷贝时，需要注意实现全拷贝和全赋值

1. 编译器提供的缺省全拷贝和全拷贝赋值运算符函数，会自动调用基类和成员子对象的拷贝构造

\*\*\*\*自定义全拷贝和局部拷贝

2.子类的拷贝构造函数

如果需要自己定义子类的拷贝构造函数，不要忘记在拷贝构造函数中调用基类子对象和成员子对象的拷贝构造函数（一般写在初始化表）。

其中：基类子对象参数直接传子类对象(that)，编译器会隐式向上造型，完成类型匹配。

\*\*\*\*自定义全赋值和局部赋值

3.子类的拷贝赋值运算符

如果需要自己定义子类的拷贝赋值运算符函数，不要忘记显式调用基类和成员类的拷贝赋值运算符函数，复制子类对象中的基类和成员子对象。

====================================================

4.忠告：

1)尽量避免使用指针型成员；

2)尽量通过引用或指针向函数传递对象型参数，和接收对象型返回值；

注解：这样既可以降低参数传递的开销，也能减少拷贝构造的机会

1. 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

4)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

==============================================

二、静态成员和单例模式

1.静态成员的共同特点：

a）静态成员不属于对象，属于类。

b）既可以通过对象访问，也可以通过“类名::”直接访问

c）静态成员受到访问控制属性的约束。

2.关于静态成员变量：

a）存放在全局区(数据区/BSS区)，具有进程级生命期

b）不能在构造函数中对其初始化，必须在类外单独定义和初始化。

c）不参与类型字节数的计算

3.关于静态成员函数：

a）静态成员函数没有this指针，也没有常属性，所以不能访问非静态成员。

b）静态成员函数中只能访问静态成员变量或调用静态成员函数；而在非静态成员函数中既可以访问静态成员，也可以访问非静态成员(静静,非静,静非静)。

注解：因为没有this指针，无法区分不同对象的同名成员

--------------------------------------

4.单例模式(掌握设计思想) 代码参见(静态成员/单例模式)

只允许存在唯一的对象实例。

1)禁止在类的外部创建对象：私有化所有构造函数

2)类内部自己维护的唯一对象：私有的该类类型的静态成员对象

3)提供外部的访问接口：返回静态对象引用类型的静态成员函数

4)两种创建模式：

---》饿汉方式：程序启动就创建、线程安全、资源效率不高。

---》懒汉方式：当使用时才创建、线程风险、资源效率较高。

===========================================

三、成员指针(只能指向定义的类中的成员变量)(看到要懂，了解)

1.成员变量指针

1)定义和初始化

形如：int A::\*p1 = &A::m\_d;

成员变量指针本质：就是类中指定成员变量在对象中的相对地址，偏移量。

class A{

public:

int m\_a;

int m\_b;

};

A a;

A\* pa = &a;

int A::\*pm = &A::m\_a;//成员变量指针

a.\*pm = 100; //a.m\_a = 100

pa->\*pm = 200; //pa->m\_a = 200

pm = &A::m\_b; //指向下一个成员变量

a.\*pm = 100; //a.m\_b = 100

pa->\*pm = 200; //pa->m\_b = 200

2)成员变量指针使用

对象 .\* 成员指针变量名；

对象指针 ->\* 成员指针变量名；

“.\*”：成员指针解引用运算符

“->\*”：间接成员指针解引用运算符

-------------------------------------------------

2.成员函数指针

1)定义和初始化

返回类型 (类名::\*成员函数指针) (形参表) = &类名::成员函数;

形如：int (A::\*pfoo)(int) = &A::foo;

1. 调用指向的成员函数

注解：虽然成员函数并不存储在对象中，不存在根据相对地址计算绝对地址的问题，但也要通过对象或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。

(对象.\*成员函数指针) (实参表)

(对象指针->\*成员函数指针) (实参表)

class A{

public:

int m\_a;

int m\_b;

int foo(int x){}

};

A a;

A\* pa = &a;

void (A::\*pfoo)(int) = &A::foo; //成员函数指针

(a.\*pfoo)(5); //对象调用函数

(pa->\*pfoo)(5); //对象指针调用函数

-------------------------------------------------------

静态成员指针

3.静态成员的指针与普通指针的用法一样，区别只是静态成员与对象无关。

A.静态成员变量指针

1)定义和初始化

类型\* 静态成员变量指针 = &类名::静态成员变量;

class A{public: static int m\_data;};

int\* pm = &A::m\_data;

2)解引用

\*pm = 100; //A::m\_data = 100;

注解：类的静态成员变量不是对象的一部分，不需要根据相对地址计算绝对地址，也不需要通过对象或其指针解引用

----------------------------

B.静态成员函数指针

1)定义和初始化

返回类型 (\*静态成员函数指针) (形参表) = 类名::静态成员函数名;

class Student{public: static Student\* create(void);}

Student\* (\*p\_create)(void) = Student::create;

2)调用指向的静态成员函数

静态成员函数指针(实参表)

Student\* student = p\_create();

注解：类的静态成员函数没有this指针，无需调用对象

4.对所有的成员指针的初始化或赋值，受访问控制属性的限制(私有的无法指向)。

===============================

操作符的重载

注意：

1.为现实对对象的直接运算，需要定义操作符重载函数

2.两个对象中间加上一个操作符的表达式会被编译器理解为调用操作符重载函数，该函数的返回值就是表达式的值。

举例：

1. 成员函数处理：L # R; ==> L.operator#(R);
2. 全局函数处理：L # R; ==> ::operator#(L,R);

-----------------------------------------------

\*\*\*友元关键字，friend：

1.friend关键字，可以把以下几样声明为某个类的友元 (声明必须在该类中)：

（1）一个全局函数

（2）另一个类的成员函数

（3）另一个类整体

2.友元可以声明在授权类的公有、私有或者保护等任何区域，且不受访问控制限定符的约束

3.友元不是成员，即便将其定义在授权类的内部，其作用域也不属于授权类，当然也不拥有授权类的this指针

1. 因为不隶属于同一个作用域，友元和成员也不可能构成重载
2. 可以通过调用一个类的成员函数，访问另一个对象的任何成员变量，

方法：定义在调用对象类中(最好定义常函数)，在用friend关键字在另一个类型中做函数声明

--------------------------------------------------

一.双目操作符

表达式处理方式：

成员函数：L#R; ==> L.operator#(R);

全局函数：L#R; ==> ::operator#(L,R);

1.运算类双目操作符(+/-...)

left + right

--》表达式结果是右值

--》左右操作数可以是左值也可以是右值

1)成员函数：

Complex const operator+(Complex const& right)const{...}

2)全局函数：

Complex const operator+(Complex const& left, Complex const& right){...}

注意：操作符重载时，需要注意语义(接收什么值，返回什么值)。

上面函数从左到右加上3个const的目的：

1）修饰返回值，让其返回右值

2）常引用：为了能接收右值(常量型的右操作数)

3）常函数：为了能让常对象调用(支持常左操作数)

--------------------------------------

2.赋值类的双目操作符(+=/-=....)

left += right;

--》表达式结果是左值，就是操作数自身

--》左操作数必须是左值

--》右操作数既可以是左值也可以是右值

1)成员函数：

Complex& operator+=(Complex const& right)const{

...

return \*this;

}

2)全局函数：

Complex& operator+=(Complex& left, Complex const& right){

...

return \*this;

}

---------------------------------------------------------------------

1. 单目操作符

表达式处理方式：

成员函数：#c; ==> c.operator#();

全局函数：#c; ==> ::operator#(c);

1.计算类单目操作符(~,-,!...)

根据语义处理

-------------------------

2.自增减单目操作符(++/--)

1)前缀自增减

--》返回值是左值，就是操作数本身

--》操作数必须是左值

1)成员函数：

Complex& operator++(void){

...

return \*this;

}

2)全局函数：

Complex& operator++(Complex& c){

...

return \*this;

}

------------------------

2）后缀自增减(为了区分前++，增加一个哑元参数)

--》返回值是右值，返回自增减之前的副本

--》操作数必须是左值

成员函数形式：c#; ==> c.operator#(int);

全局函数形式：c#; ==> ::operator#(c,int);

1)成员函数：

Complex const operator++(int){...}

2)全局函数：

Complex const operator++(Complex& c, int){...}

-------------------------------------------------------------------

1. 其它操作符

1.插入/提取操作符: << >>

I/O流文件：

#include <iostream>

class istream{};

class ostream{};

ostream cout; //输出流对象

istream cin; //输入流对象

---------------------------

由于I/O流文件无法修改，所以可以在想要直接插入、提取对象的类里，声明友元的全局函数

（1）插入运算符：<<

friend ostream& operator<<(ostream& os,Complex const& c){

...

return os;

}

（2）提取运算符：>>

friend ostream& operator>>(istream& is,Complex& c){

...

return is;

}

----------------------------------------------------------------------

2.下标操作符: [](原理与数组相同,常用于在容器类型中以下标方式获取数据元素)

注:非常容器的元素(包括基本类型或对象)为左值，常容器的元素为右值

class Array{

public:

//非常容器对象:[]

int& operator[](size\_t i){

return m\_data[i];

}

//常容器对象：[]

int const& operator[](size\_t i)const{

//直接复用上面非常版本

return const\_cast<Array&>(\*this)[i];

}

private:

int m\_array[100];

};

int main(void)

{

Array a(5);

//非常容器返回左值

a[0] = 10; //a.operator[](0)

cout << a[0] << endl; //10

const Array& ra = a;

//常容器返回右值

//ra[0] = 20; //error

cout << ra[0] << endl; //10

return 0;

}

-------------------------------------

3.函数运算符：()（几乎不用）

可以将对象当做函数来使用。

A a;

a(10,20);//==> a.operator()(10,20)

----------------------------------------------

4.解引用/间接成员访问运算符：\*/->

\*\*\*智能指针是一个封装了常规指针（通常指向堆区）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，避免内存泄露。

注解：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(smart pointer)的一种通用实现技术是使用[引用计数](http://baike.baidu.com/item/%E5%BC%95%E7%94%A8%E8%AE%A1%E6%95%B0" \t "http://baike.baidu.com/_blank)(reference count)。智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象的指针指向同一对象。调用[析构函数](http://baike.baidu.com/item/%E6%9E%90%E6%9E%84%E5%87%BD%E6%95%B0" \t "http://baike.baidu.com/_blank)时，析构函数减少引用计数（如果引用计数减至0，则删除基础对象）。

---------------------------------------------------------------

5.类型转换操作符（隐式转换，类型转换构造的逆向转换）

自定义类型转换：

1）基类类型 --》类类型：通过类型转换构造函数实现自定义转换

2）类类型 --》基本类型：通过类型转换操作符函数实现自定义转换

operator 基本类型(void)const{...}

注意：当定义的类型转换操作符函数前加了explicit，就只能显示转换

3）类类型 --》类型型：上面两种方式都可以

4）基本类型 --》基本类型：无法实现

----------------------------------------------------------------

6.创建/销毁运算符：new/delete，new[]/delete[]（了解）

static void\* operator new(size\_t size){...}

static void operator delete(void\* p){...}

-----------------------------------------------------------------

三、操作符重载的限制

1.有些运算符是不能重载的。

--》作用域限制操作符（::）

--》直接成员访问操作符（.）

--》直接成员指针你解引用操作符（.\*）

--》条件操作符（?:）

--》字节长度操作符（sizeof）

--》类型信息操作符(typeid)

2.如果一个操作符的所有的操作数都是基本类型，则该操作符无法重载；

3.不能通过运算符重载改变运算符的优先级。

4.不能改变操作符的操作数个数。

5.不能发明新的运算符。

6.保持一致性和可读性。

一、继承(Inheritance)的基本概念和语法

1.继承的基本概念

1)共性和个性：基类(基本类)表示共性，子类(派生类)表示个性。

2)继承和派生：基类派生子类，子类继承基类。

2.继承的语法

1)继承表

class 子类:继承方式1 基类1,继承方式2 基类2,...{...};

2)继承方式

公有继承：public

保护继承：protected

私有继承：private

------------------------------------

二、公有继承的基本特点

1.皆然性：子类对象任何时候都可以被当做其基类类型的对象。

逻辑结构：子类的逻辑空间小于基类(学生属于人类)。

物理结构：任何一个子类对象中一定都包含着一个基类的子对象。

-----------------------

1. 向上和向下造型

(缩小指针的操作范围，安全，可以隐式完成)

向上造型：将子类类型的指针或引用转换为基类类型的指针或引用。这种操作性缩小的类型转换，在编译器看来是安全的，可以隐式完成。

(扩大指针的操作范围，危险，必须显示转换)

向下造型：将基类类型的指针或引用转换为子类类型的指针或引用。这种操作性扩大的类型转换，在编译器看来是危险的，必须显示完成(static\_cast<>)。

---------------------------

3.成员的继承

1）在子类中或者通过子类，可以直接访问基类中的公有成员和保护成员，如同它们是在子类被声明一样。

2）基类的私有成员在子类中存在但是不可见，所以无法直接访问，但是可以通过基类提供的公有或保护的成员函数间接访问

3）基类的构造函数和析构函数，子类无法继承，但是在子类自己的构造函数中，可以通过初始化表显示指明基类部分的初始化方式。

成员子对象：类类型的成员对象

基类子对象：子类对象包含的基类部分成员

-----------------------------

4.名字隐藏(子类隐藏基类的同名成员，常有同名变量和函数)

子类中的标识符会隐藏基类中的同名标识符。

标识符：变量名、函数名、类型名、名字空间名等。

1. 基类中任何被子类隐藏的标识符，都可以通过作用域限定操作符“::”被子类或类的外部访问。
2. 同名函数因为作用域不同，不能构成重载关系，而只是隐藏关系
3. 通过using声明将基类的成员函数引入到子类的作用域，形成重载，但一般不这么做。

---------------------------------------

1. 继承方式对访问控制的影响
2. 只有继承的成员受继承方式的影响

非继承成员的访问控制属性：

=================================================

访问限定符 访问属性 内部 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

==================================================

继承方式对访问控制的影响：（继承部分变为保护/私有）

公有继承：没有任何影响

保护继承：公有->保护

私有继承：公有/保护->私有

============================================

| 继承方式

--------------------------------------------

基类访控属性| 公有继承 保护继承 私有继承

--------------------------------------------

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

=============================================

注：向上造型不适用私有和保护继承，私有的子类和保护子类类型的指针或引用，不能转换为基类类型的指针或引用。

----------------------------------------------------------------------

四、子类的构造、析构

1.子类的构造

1)基类的隐式构造：不说基类怎么构造，一定是缺省构造

2)基类的显式构造：在子类构造函数的初始化表中显式指明基类子对象的构造方式。

3)构造顺序：基类子对象->成员子对象->子类构造代码

2.子类的析构

1)子类的析构函数会自动调用基类的析构函数，但是基类的析构函数不会自动调用子类的析构函数。

2)delete一个指向子类对象的基类指针，子类的析构函数不会被调用，导致内存泄漏。

举例：

class A{...};

class B:public A{...};

int main(void){

A\* pa = new B; //pa是指向子类对象的基类指针

delete pa; //导致内存泄漏

}

3)析构顺序：析构代码->成员子对象->基类子对象

===============================

1. 多继承

定义：子类继承多个基类

1.名字冲突问题

1）名字冲突：子类的多个基类存在相同名字的成员，当通过子类访问这些成员时，编译器会报歧义错误；

2）解决名字冲突的一般做法就是通过作用域限定(推荐)，指明所访问的成员出处。

3）同名函数不能重载：产生名字冲突的成员函数，不能重载，因为作用域不同。

4）可以通过using声明，消除作用域限制，在子类中构成重载关系。(不推荐，代码可读性差)

注意：不同类型同名，编译也会报歧义；比如函数名和变量名冲突。

1. 子类中的标识符依旧会隐藏基类中的同名标识符。

3.类型转换：隐式的向上造型和静态的向下造型，编译器都会自动计算指针的偏移量，保证指针或引用的类型与其实际指向的对象的类型严格一致。

注意：引用的情况和指针类似，重解释转换不会计算偏移量。

=======================================================

1. 钻石继承和虚继承

1.钻石继承：一个子类的多个基类源自共同的基类祖先。

A(公共基类)

/ \

B C(中间子类)

\ /

D(汇聚子类)

1. 多分基类对象：派生多个中间子类的公共基类子对象(A),在末端的汇聚子类对象(D)中,存在多个实例。在汇聚子类(D)中，访问公共基类的成员，会因为路径的不同导致不一致。
2. 通过虚继承可以令公共基类子对象(A)在汇聚子类对象(D)的中实例唯一,并为所有的中间子类(B、C)所共享，这样即使沿着不同的继承路径，所访问到的公共基类成员一定是一致的。

(虚继承存在是为了面向对象的语法符合逻辑)

4.虚继承语法

1）中间子类的继承表中使用virtual关键字

2）虚基类的所有子类(包括汇聚子类和中间子类)，都必须在其构造函数中显示说明虚基类子对象的初始化方式，否则将以无参方式初始化。

补充：

虚继承原理：中间子类多存放了一个三级指针

内存模型：虚指针->虚基类表->虚基类子对象相对于中间基类子对象的偏移量

===============================

三、多态和虚函数

1.多态基本概念

如果将基类中的某个成员函数声明为虚函数，那么其子类中与该函数具有相同函数原型的成员函数就也是虚函数，并且与基类中的版本构成覆盖关系。

这时，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）调用虚函数，实际被调用的是子类中的覆盖版本，而非基类中的原始版本，这种语法现象称为多态。

-------------------------------------------------------

2.虚函数覆盖的条件

1）只有成员函数可以被声明为虚函数，构造函数、全局函数、静态成员函数(因为没有this指针)都不能声明为虚函数。

2）只有基类中的虚函数(virtual关键修饰)才能被子类覆盖，子类中覆盖版本的virtual加不加都可以。

3）子类中的覆盖版本和基类中的原始版本，必须拥有相同的函数签名。

注解：函数签名：函数名、形参表、常属性

1. 如果基类中的虚函数返回的不是类类型的指针或引用，那么覆盖版本必须和原始版本返回类型一致。

注意：当函数签名相同，返回类型不同时，子类版本试图覆盖基类版本虚函数，但是覆盖出错。

5）如果基类中的虚函数返回类类型的指针或引用，那么该函数在子类中的覆盖版本可以返回其基类版本返回类型的公有子类的指针或引用--类型协变

6）无论基类中虚函数的访控属性如何，该函数在子类中的覆盖版本可以是任何访控属性。

7）如果基类的虚函数带有异常说明，那么该函数在子类中覆盖版本，不能说明比基类版本抛出更多的异常（否则会因为“放松throw的限定”而导致编译失败）。

补充一句：这种情况下，虽然实际调用的是子类中的覆盖版本，但是访控属性受限于基类。

------------------------

协变覆盖代码举例：

class A{};

class B:public A{};

class Base{

public:

virtual A\* hum(int a = 0){

cout << "Base::hum" << endl;

}

};

class Derived:public Base{

public:

//返回类型是类类型指针或引用，并且具有继承关系，可以形成协变覆盖

B\* hum(int a = 0){

cout << "Derived::hum" << endl;

}

};

-------------------------------------

异常范围覆盖代码举例：

class Base{

virtual void func(void)throw(int,char){...}

};

class Derived:public Base{

void func(void)throw(int,char){...} //ok

void func(void)throw(int){...} //ok

void func(void)throw(double){...} //覆盖失败

}

----------------------------------------------

3.发生多态的条件

1）表现多态除了需要在子类中形成对基类中虚函数的覆盖，还必须通过指针或引用来调用该虚函数。

2）调用虚函数的指针也可以是成员函数中的this指针，只要是指向子类对象的基类指针。

举例：子类对象直接调用基类已被覆盖的虚函数（重点掌握）。

原理：子类调用对象在基类中生成了指向子类对象的基类类型this指针。

1. 在基类的构造和析构函数中调用虚函数，不会表现出多态。

原因：子类调用对象尚未构造或已析构完毕，它只表现出基类类型的外观和行为，这时只能调用基类版本虚函数。

另外：在构造或析构函数中通过已构造完毕或尚未析构的对象调用虚函数，其多态性不受任何影响。

4）显示指明调用基类中虚函数，不会表现出多态（函数地址不同）

--------------------------------------------------------------------------

4.纯虚函数、抽象类、纯抽象类

抽象类举例：

形状(位置) (抽象类)

/ \

圆形(半径) 矩形(长、宽)

（1）纯虚函数：（存在就是为了让子类函数覆盖）

virtual 返回类型 函数名(形参表)[const] = 0;

（2）抽象类：包含纯虚函数的类，不能实例化对象。

（3）抽象属性的继承：如果子类没有覆盖其基类的全部纯虚函数，那么子类也是抽象类。

（4）纯抽象类：全部成员函数(构造和析构除外)都是纯虚函数。

-------------

补充扩展：

1. 工厂方法模式（好莱坞模式）
2. 纯虚函数可以不定义，但如果定义，必须写在类的外部

void Shape::draw (void) const { ... }

3.在基类的构造和析构函数中调用纯虚函数，结果将是未定义的。通常会在链接阶段报告失败。如果该纯虚函数有定义，编译器将在给出警告之后，选择调用基类版本

------------------------------------------------------------------------

5.虚函数表和动态绑定

补充（了解）：多态实现原理

1）编译器会为包含虚函数的类生成一张虚函数表，用来存放每个虚函数地址，简称虚表，每个虚函数都对应虚函数表中的索引号。

2）同时会增加一个隐式的成员变量，用于存放虚表的地址，该变量称为虚函数表指针，简称虚指针(vptr);

3）所谓的虚函数覆盖，本质就是用子类中的虚函数的地址覆盖基类虚表中对应基类虚函数的地址

4）当编译器看到通过指针或引用调用虚函数时，不会直接生成函数调用的语句，而是用一段代码来替换该语句，这段代码在运行阶段被执行，完成如下功能：

--》确定指针或引用的目标对象的真实类型，并找到虚指针

--》根据虚指针找到相应的虚函数表，并从中获取调用虚函数的入口地址

--》根据函数入口地址执行虚函数代码

这个函数绑定过程发生在程序的运行阶段，因此被称为动态绑定。

6.动态绑定对程序性能影响

1）虚函数表会增加内存开销

2）虚函数调用会增加时间的开销

3）虚函数不能被内联优化

注：如果没有多态的语法要求，最好不要使用虚函数

-----------------------------------------------------------

7.虚析构函数

1）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，基类的析构函数只负责虚构基类子对象的动态自动，不会调用子类的析构函数，这样子类中特有的动态资源就会形成内存泄露。

2）将基类的析构函数声明为虚函数，那么子类的析构函数就也将是虚函数，并且对基类的虚函数构成有效的覆盖，可以表现多态的特性。这时，delete一个指向子类对象的基类指针，实际被调用的将是子类的析构函数，子类析构函数在释放完它自己特有的动态资源之后，会自动的调用基类的析构函数，释放基类子对象的动态资源，避免内存泄露。

------------------------------------------------

四、运行时类型信息（RTTI）

1.typeid运算符

#include <typeinfo>

1）typeid操作符既可用于类型也可用于对象（用法类似sizeof()）

int x;

typeid(int);

typeid(x);

2）typeid操作符返回typeinfo对象的常引用

--》typeinfo类的成员函数name()，返回空字符结尾的类型名字符串

--》typeinfo类支持“==”和“!=”操作符，可直接用于类型相同与否的判断

if(typeid(human) == typeid(Student)){ ... }

1. 当typeid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，该操作符所返回的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。

多态继承：基类包含至少一个虚函数。

多态继承的举例：

class A{};

class B:public A{virtual void foo (void){}}; //继承

class C:public B{}; //多态继承

=================================

二十四、异常(Exception)

1.C语言错误处理：

1)返回值：层层判断，代码臃肿，局部对象能被析构。

2)远程跳转(setjmp/longjmp)：一步到位，代码精炼，局部对象可能造成内存泄漏。

--------------------------

2.C++异常机制：

结合两种传统错误处理的优点，同时避免它们的缺点，在形式上实现一步到位错误处理，同时保证所有局部对象，在异常传递的堆栈辗转开解(stack-unwinding)过程中，被异常处理系统销毁。

---------------------------------------------------

3.异常语法

3.1异常抛出

1）throw 异常对象

2）抛出异常对象可以是基本类型

举例：

throw -1;

throw "errror";

1. 也可以抛出类类型的异常对象

class FileError{...};

FileError ex("file error");

throw ex;

注意：推荐以匿名临时对象的形式抛出异常

throw FileError("file error");

---------------------

3.2 异常捕获

1）代码框架：

try{

可能引发异常的语句；

}

catch(异常类型1& ex){

针对异常类型1的处理；

}

catch(异常类型2& ex){

针对异常类型2的处理；

}

...

catch(...){

针对其它异常类型的处理；

}

1. catch子句根据异常对象的类型自上而下顺序匹配，而非最优匹配，因此对子类类型的异常捕获应该写在前面，避免发生隐式的向上造型，而被基类的异常捕获语句提前截获。
2. 建议在catch子句中使用引用接收异常对象，避免因为拷贝构造带来性能损失，或引发新的异常

-------------------------------

3.3 标准异常类(C++库中异常类)

class exception{

public:

exception() throw(){}

virtual ~exception() throw(){}

virtual const char\* what()throw()const;

};

---------------------

class A：public exception{

const char\* what()const throw(){

cout << "A::exception" << endl;

}

}

class B：public exception{

const char\* what()const throw(){

cout << "B::exception" << endl;

}

}

int main(void){

try{

func(); //可能抛出A或B类型异常

}

catch(exception& ex){ //多态

cout <<　ex.what() << endl;

}

}

注意：建议从标准库异常中派生自己的异常类，并在子异常类中覆盖what()函数，以构成多态。

---------------------------

1. 异常说明

形如：

返回类型 函数名 (形参表) [const] throw (异常类型1, 异常类型2, ...);

1）异常说明是函数原型的一部分，显示说明，调用者自行捕获并处理被抛出的异常范围。

2) 如果抛出该类型范围以外的异常，会被系统捕获（导致进程终止）。

3）函数没有异常说明，表示抛出的任何类型异常都让系统处理（终止进程）。

4）函数异常说明为空（如：throw()）,表示抛出的任何类型异常都由调用者自行处理。

1. 异常说明在函数的声明和定义中必须保持一致的类型，否则将导致编译错误（顺序无所谓）。
2. 如果基类的虚函数带有异常说明，那么该函数在子类中覆盖版本，不能说明比基类版本抛出更多的异常（否则会因为“放松throw的限定”而导致编译失败）。

举例：

class Base{

virtual void func(void)throw(int,char){...}

};

class Derived:public Base{

void func(void)throw(int,char){...} //ok

void func(void)throw(int){...} //ok

void func(void)throw(double){...} //覆盖失败

}

-------------------------------------------------------

5.构造函数中抛出异常

1. 构造函数可以抛出异常，某些时候还必须抛出异常，因为：

--》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，无法通过返回值回复调用者结果。

1. 如果构造函数抛出异常，对象将被不完整构造，这样其析构函数永远不会被执行。

--》因此必须在构造函数抛出异常之前，手动释放所有动态分配的资源，避免内存泄漏。

6.析构函数中的异常（了解）

1. 析构函数最好不要抛出异常。

原因：如果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，这时会形成死递归，系统会直接终止进程。

2）如果处于某种特殊原因，析构函数抛出了异常，也应该在析构函数内部捕获处理。

===================================

二十五 I/O流(了解)

1 主要I/O流类

ios

/ \

istream ostream

/ | \ / | \

istrstream ifstream iostream ofstream ostrstream

/ \

strsteam fstream

2 格式化I/O

1）格式化函数

eg：

cout.precision(5);//设置cout输出浮点数的精度，5个有效数字

cout << 3.1415926 << endl;//3.1415

2）流控制符

eg:

cout << setprecision(5) << 3.1415926 <<endl;//3.1415

3 字符串流

#include <strstream>

istrstream\ostrstream\strstream

#include <sstream>//更流行

istringstream\ostringstream\stringstream

4 文件流

#include <fstream>

ifstream、ofstream、fstream

5 二进制I/O

//类似UC：write

ostream& ostream::write(const char\* buf,size\_t num);

//类似UC：read

istream& istream::read(char\* buf,streamsize num);

==============================================================

避免一步中多次隐式转换，因为编译器可能会报错。 --------闵卫

缩小防控权限，安全，可以转换；扩大防控权限，危险，一般报错。

小结：C++之父给C程序的建议

1.慎用宏

#define PAI 3.14 ==》const double PAI = 3.14；

#define STATE\_SLEEP 0

#define STATE\_RUN 1

#define STATE\_STOP 2

==>

enum STATE{SLEEP,RUN,STOP}

#define max(a,b) ((a)>(b)?(a):(b))

==>inline int max(int a,int b){

return a >b?a:b;

}

2 变量随用随声明，同时初始化

3 尽量使用new/delete取带malloc/free

4 少用void\*,指针计算、联合体、强制转换

5 尽量用string表示字符串，少用C风格的char\*

===============================

什么是面向对象？

把数据和对数据的操作分类，抽象成一个整体，使编写程序的方式更接近人的逻辑方式。

面向对象的3个特点：封装，继承，多态