C++

类型输入输出

C与C++的区别：1.C++对类型（指针比较严格。变 量没有所谓【cosnt int a;int b=a】也可以的）的检查更加的严格【指针申请空间的时候，一定要有类型，也就是要实现强转char \*p=(char \*)malloc(100)】【用const（放在\*前面）修饰的变量要赋给指针地址的时候，指针也必须是const（放在\*前面）修饰的指针const int a=5;const int \*p=&a】【把一个 const 类型的指针赋给非 const 类型的指针。c 语言中可以通的过，但是在 c++中则编不过去】

2.C++中有bool类型，而C语言没有【C语言中用0表示假，非0表示真】【C++中采用bool类型，bool flag =true】【bool类型为1个字节sizeof(bool)】

3.C++中的枚举是真正的枚举【C语言中的枚举，在初始化的时候，还可以取不是枚举里面的变量（enum A{a,b};enum A s=100;）】【C++中的枚举，在初始化的时候，只能取枚举里面的变量】

4.输入与输出的不同【C语言是printf和scanf】【C++是cin（流输入）和cout（流输出）】【C语言输入字符串的时候是这样的char buf[30];可以用scanf或者gets这两个都是容易越界的，用fgets(buf,30,stdin)就不会越界了】【C++中用cin>>buf也容易越界，用cin.get(buf,30)就不会越界了】

5.输入输出的格式控制不同【对于实型C++默认输出六位】【C语言中右对齐的输出宽度是%3d，左对齐的输出宽度是%-3d】【C++输出宽度是setw(3)（头文件是iomanip）（这个只是影响它后面的一个输出而已，如果要使每个输出都这样，就都要加setw(3)，简单来说就是影响局部），左对齐是setiosflags(ios::left)（这个影响它后面所有的输出，简单来说就是全局），cout<<setiosflags(ios::left)<<setw(3)<<’c’<<setw(3)<<5<<endl，填充域宽的时候，可以采用setfill(‘0’)（这个影响它后面所有的输出，简单来说就是全局），cout<<setw(5)<<setfill(‘0’)<<10<<endl】

6.输出浮点数【C++中，输出的浮点数，从第一个不为0的数字算起，有效数字为6位，如果浮点数多于6位的话，以四舍五入的形式输出6位（cout<<12.345678），如果要完全的输出浮点数的话，采用setiosflags(ios::fixed)，自己控制浮点数要输出小数点后面几位的话，可以采用setiosflags(ios::fixed)<<setprecision(2)，连用的形式，输出小数点后面两位】

7.输出进制数【C语言中，八进制是%o,十六进制是%x】【C++中，八进制是oct，十六进制是hex，十进制是dec（默认是十进制）(cout<<hex<<100<<endl)，还可以通过setbase(8)或者setbase(16)来输出进制数】

8.函数重载【C语言中没有存在函数重载】

9.结构体的使用不一样【C语言中，比如定义一个结构体变量struct complex a，要这样定义】【C++中，定义一个结构体变量complex a，它可以不用带struct】

10.运算符重载【C语言中也有运算符重载，只不过是C本身就重载好的（\*可以表示乘法也可以表示节引用）（&可以表示位与也可以表示引用）】【C++中的重载是可以自己写的】

11.引用【C语言中没有引用，只有取地址&】【C++中有引用，用来取别名】

12.const【C语言中的const修饰局部变量虽然说这个变量的值不能再进行修改了，但是还是可以用指针取地址来进行修改（const修饰全局变量就不可以了，相当于把全局变量放在只读数据段）】【C++中的const修饰局部变量就是真的不可以修改了，也不可以用指针取地址进行修改】

13.字符串【C语言的字符串用char buf[]】【C++中有字符串类型string】

14.堆内存的申请和释放【C语言用malloc和free（这是系统函数）【需要判断是否申请成功（if(p==NULL)），因为malloc有返回值】】【C++用new和delete（这是关键字）【不需要判断申请成功，因为new没有返回值】】【new会自动调用构造函数，delete会自动调用析构函数，但是malloc和free不会】

15.内联函数【C语言中没有内联函数，用的是宏（宏的优点：内嵌到目标代码【就是单纯替换】，减少函数的调用）（宏的缺点：在预处理阶段就被处理了，可能带来语意上的差错）】【C++中替代宏的函数（优点：完成某一些操作的抽象，避免相同功能的重复开发）（缺点：无法避免压栈和出栈的开销）】【C++用内联函数可以具有宏和函数的优点，还多了个类型检查，并且没有了它们的缺点】（inline int sqt(int a){}）

16.命名空间【C语言没有命名空间】【C++有命名空间（namespace NN{int a}）】

17.返回值可以做左值【C语言返回值不可以做左值（(a=b)=4）是不可以的】【C++返回值可以做左值（(a=b)=4）是可以的】

18.强转类型【C语言中的强转(float)a/b】【C++中的强转static\_cast<float>a/b】

函数重载

函数重载规则：1.函数名相同。2.参数类型或者个数或者顺序不同（一个不同就可以），就可以构成重载。3.返回值类型不同不能构成重载（这个是说，如果参数不变的情况下，返回值类型不同不能构成重载，如果参数改变的情况下，返回值类型不同还是可以构成重载的）【比如：int func(int a)和double func(int a)这样就不能构成重载，int func(int a)和double func(double a)这样是可以构成重载的】

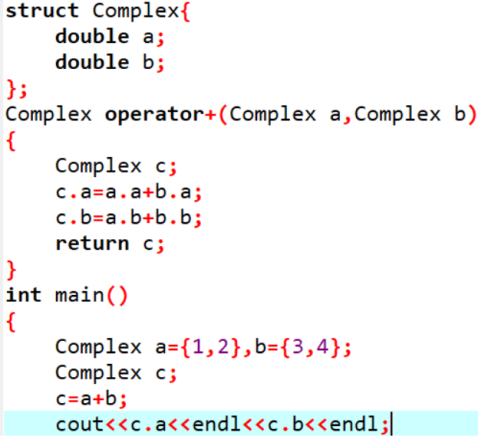
倾轧技术：C++利用 name mangling(倾轧)技术，来改名函数名，区分参数不同的同名函数。

name mangling 发生在两个阶段，.cpp 编译阶段，和.h 的声明阶段。只有两个阶段同时进行，才能匹配调用。

操作符重载

C++中，允许int到long和double的隐式类型转化，如果函数只有void func(long a)和void func(double a)这两个函数，没有int的形参函数的话，func(1)就不知道该调用哪个函数（出现了二义性ambiguous），所以我们要强给1long或者double类型，比如func(1l)或者func((double)1)

运算符重载：格式：operator运算符（比如Complex operator+(Complex a,Complex b)【Complex是一个结构体】,然后下面就可以用+进行结构体相加【Complex a,Complex b,Complex c,c=a+b】）【跟写函数一样，但又不同于函数】



默认参数

默认参数：如果定义在前（就是不用声明了，这时候默认参数是定义的），如果定义在后，声明在前（这时候默认参数是声明的，而且定义的时候就不用再进行默认了，声明的时候默认就可以了，如果两边都有的话，就会报错）【【一个函数不能既做重载函数，又做默认参数的函数】】

1.默认单个形参，就是在定义函数的时候，给形参赋一个值，然后在下面调用函数的时候，就可以不用输入实参，也可以进行调用（void func(int a=10)）

2.默认多个形参，默认的时候，要注意默认顺序，要从右往左（void func(int a,int b,int c)，要从右往左进行默认，并且不能跳跃，如果是这样void func(int a,int b=1,int c)，这样函数调用的时候，实参写不了）

引用及提高

引用：格式：int a;int &b=a;（相当于取别名，给a取别名，只是取别名而已，类型、大小、地址都是一样的【也可以说它是一种声明的关系，声明的时候必须初始化，并且只能进行一次初始化】【可以对引用再次引用（但不能定义引用的引用），让一个变量有多个别名】）【区分取地址和引用：在=左边是引用，在=右边是取地址】

引用：应用：以前C语言中，要交换两个变量的值，用的是传地址（传指针）【Swap(int \*a,int \*b)】，现在在C++中，可以用传引用来实现【Swap(int &a,int &b)】（也就是取别名来实现）【引用的本质就是对指针的包装（引用是常指针type \*const p）】所以引用的大小是4字节（32位机）引用的本质就是指针（地址）的包装，所以在定义引用的指针（相当于取地址）是没有意思的

引用：提高：可以定义指针的引用（但不能定义引用的指针int &\*p=&a）【int \*p;int \*&rp=p;（p是int\*型，所以写成int \*rp）】【也可以来实现两个字符的交换（char \*p=”aaa”,char \*q=”bbb”,Swap(char \*&p,char \*&q)），就可以不用升级到二级指针来进行交换】

不可以定义引用的引用【int a;int &p=a;int &&pp=p】（但是可以对引用再次引用）

指针和引用最大的区别：指针可以定义指针的指针（int a;int \*p=&a;int \*\*pp=&p），但是引用不能定义引用的引用【int a;int &p=a;int &&pp=p】

可以定义指针数组【int \*p[]={&a,&b}】，但不能定义引用数组【int &p[]={a,b}】，可以定义数组引用（也就是对数组取别名，它的定义跟数组指针的定义是一样的）【int arr[]={1,2};int (&a)[]=arr】【注意要有括号】

constRef引用本质

const：C++中的const修饰的局部变量是真的不能进行修改的（所以const修饰的变量不能这样取引用【const int a;int &b=a是不可以的】，但是可以这样取引用，const修饰的变量的引用必须是const的【const int a;const int &b=a是可以的】），不能通过指针取地址进行修改

const引用和非const引用的区别：非const引用不能引用常量，但是const引用可以引用常量【const int &a=1（可以），int &a=1（不可以）】非const引用不能引用不同类型的变量，但是const引用可以引用不同类型的变量【double a;const int &b=a（可以）【相当于开辟一个未命名的空间，以只读的方式进行引用，不管后面a的值如何修改，它还是第一次a的值】，double a;int &b=a（不可以）】非const引用不可以引用表达式，但是const引用可以引用表达式【double a;const int &b=a+2（可以）double a;double &b=a+2（不可以）】

const 对象的引用必须是 const 的，将普通引用绑定到 const 对象是不合法的。

这个原因比较简单。既然对象是 const 的，表示不能被修改，引用当然也不能修改，必

须使用 const 引用。

new delete

能用new和delete解决的问题不一定能用malloc和free解决，能用malloc和free解决的问题能用new和delete解决【new和delete是关键字，而malloc和free是系统函数】

开辟单变量地址空间：int \*p=new int;（初始化有两种方法：1.int \*p=new int(300) 2.int \*p=new int;\*p=300）【释放：delete p】

开辟多变量地址空间（相当于数组）：int \*p=new int[100];（初始化的方法：int \*p=new int[100]{1}【这样子是数组的部分初始化，第一个元素为1，后面全为0】）【释放：delete []p】

开辟二维空间：int (\*p)[3]=(int (\*)[3])malloc(2\*3\*4) int(\*p)[3]=new int[2][3]【释放：delete []p】

内联函数inline

内联函数：定义：inline int sqr(int a){}，具有宏和函数的优点，并且含有类型检查，没有了宏和函数的缺点，但是内联函数也是有缺点的，会增加代码段的空间【内联函数的本质：以牺牲代码段空间为代价，提高程序的运行时间的效率】

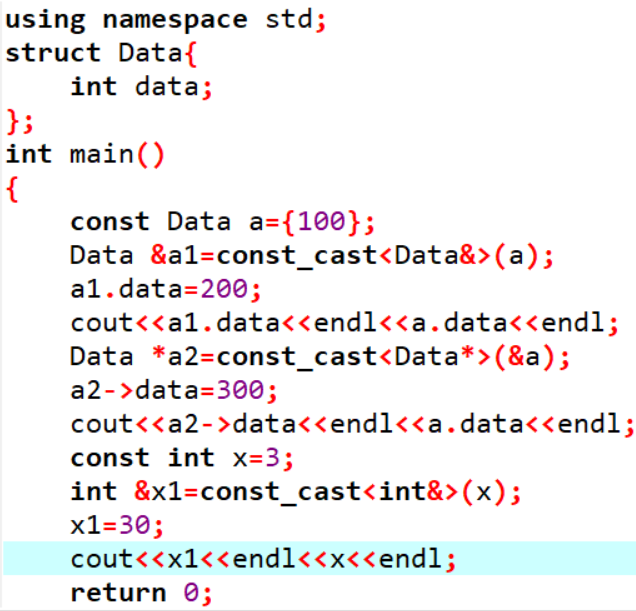
内联函数：使用场景：函数体积很小，且频繁被调用

constCast

const：C++的const是用来取代C语言的宏的，宏定义的数是没有类型的，而const定义的数是有类型的

const修饰的变量通常称为常变量：可以修饰常量（const int a=5;【这个无论通过什么方法，都无法改变a的值，脱const化也不行】）也可以修饰变量

const\_cast：使用const\_cast<typeid>(待转化变量）【typeid只能是引用或指针（就是类型）【int&或int\*】，不可以直接是对象】（const int a;int &b=const\_cast<int&>(a)）或者（const int a;int \*b=const\_cast<int\*>(&a)）虽然是脱const化了，但是const修饰的变量的值还是改变不了的，非const修饰的变量的值是可以进行修改的



结构体的脱const化是有效果的，可以改变const修饰的结构体变量的值。

命名空间

作用域运算符（：：） 作用域限定符（{}）

全局命名空间是无名的：怎么使用呢，定义一个全局变量a和一个局部变量a，我要输出全局变量a的值，就要用到全局命名空间和作用域运算符（cout<<::a<<endl），这样就是输出全局变量a的值【作用域运算符前面是要有一个类名的，因为全局命名空间是无名的，所以不用加】【函数也可以这样使用（::func（）），但是一般这样使用很少，因为main函数里面不能定义函数，所以函数就没有什么局部函数和全局函数，所以使用很少】

命名空间：作用：是对全局作用域的再次划分，比如再全局作用域里面定义一个int a=5;还想再定义一个int a=5;这时候就要用到命名空间（namespace mySpace{int a=5;int x;}）

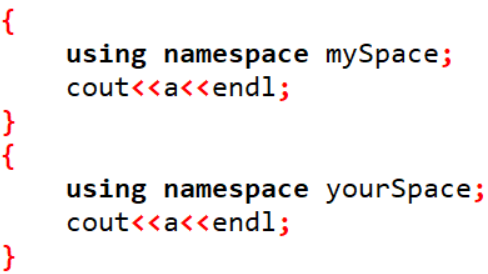
对命名空间的访问：1.cout<<mySpace::a<<endl;

1. using mySpace::a;（这样子下面就可以直接使用a了，不用像1那样那么麻烦了）cout<<a<<endl;using mySpace::x;cout<<x<<endl;
2. using namespace mySpace;（这样子下面就全部都可以使用过了，不用管命名空间里面的哪个变量，就是全部都可以使用就对了。。。。。。。。。。）cout<<a<<endl;cout<<x<<endl;

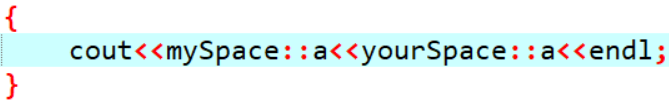
using namespace std：这个std就是C++标准库的一个命名空间，所以在C++文件里面加这个的话，cin和cout就不用在加作用域运算符了（：：）

如果遇到两个命名空间里面的变量有一样的话，又要同时使用的话，有两种方法来解决：

1. 使用作用域限定符：



1. 使用带命名空间名字的方法：



命名空间具有累计效应：namespace NN{int a=5}; namespace NN{int b=6;}，编译的时候这两个是合在一起的，累计起来的

命名空间还可以支持嵌套

string类的使用

string类型的三种初始化方法：1.string str;str=”yuzhangbin”;

2.string str=”yuzhangbin”;

3.string str(“yuzhangbin”);

string类型的变量还可以进行加法：string s1=”yu”;string s2=”zhang”;string s3=s1+s2;

string类型的变量可以直接求字符串的长度：cout<<s1.length()<<endl;

string类型的变量还可以读出字符串中的字符：cout<<(s1.c\_str())[2]<<endl;

string类型的变量还可以用来直接比较大小

string类型还可以定义数组

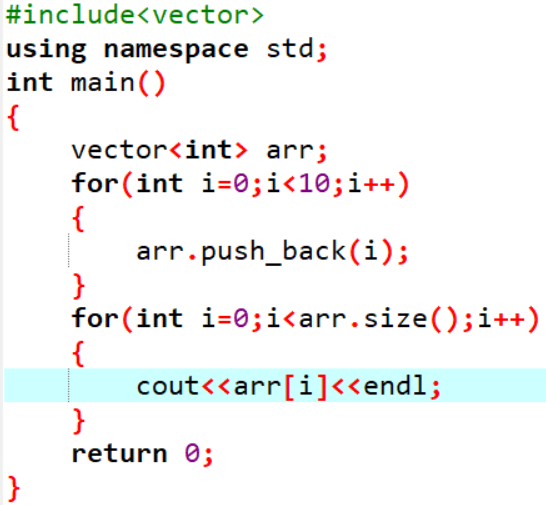
vector的使用

vector是用来替换数组的：【C语言中的数组（int a[];for(int i=0;i<sizeof(a)/sizeof(a[0]);i++）】【

C++中的数组vector（vector <类型> 数组名）】

arr.capacity()：表示申请的空间有多大

vector<int> arr(5)：表示申请5个int空间

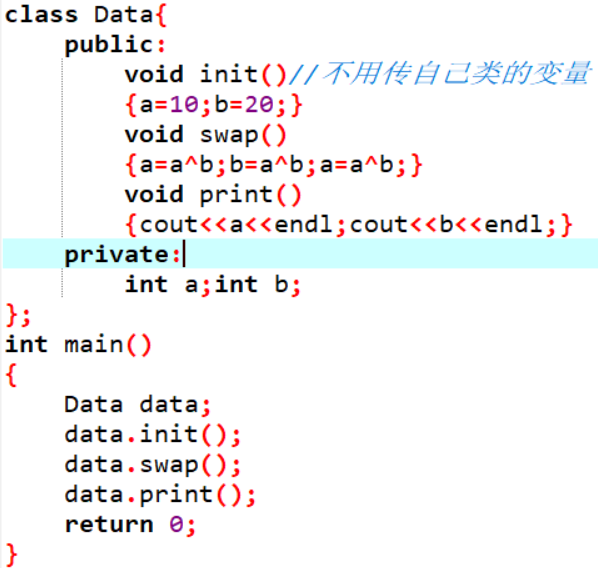
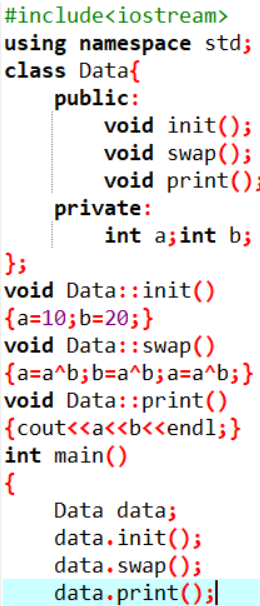


封装

结构体是对数据的封装，行为是对数据的处理和分析（比如，对结构体变量的初始化（一个函数），打印（一个函数），判断（一个函数））【类跟结构体是很像的（包括用法）】

结构体中的变量是公有的，而类class中的变量是默认私有的

用结构体封装只能封装数据，并且都是公有的。而用类封装既能封装数据，又能封装行为，并且是私有还是公有的由自己决定（默认是私有的）【把函数写在类里面，如果需要传类里面定义的数据的时候，则函数定义的时候可以不用写形参，因为都是封在一起的，都是自家人】【类名本身就是一个命名空间，所以在类里面可以只写函数的声明就可以了，把函数的实现写在类外面，并且加作用域运算符，而且不论是函数声明还是函数的实现，都不用传形参，直接用就可以了】

构造器

类：构造器【这三种情况都是调用构造器（string s;string s=”yuzhangbin”;string s(“yuzhangbin”)）】、析构器、拷贝构造函数【由已存在的对象来创建新对象】【这两种情况是调用拷贝构造函数（string s=s1;string s(s1);）】、运算符重载【这种情况是赋值运算符重载（string s1;string s5;s5=s1;）【两个都是已经存在的对象】】

构造器：1.与类同名，无返回值，在类对象创建时，自动调用【Data data;创建对象的时候它就调用的】，完成类对象的初始化【就是可以把初始化写在这个构造器里面，这样就不用写初始化函数了】，尤其是动态对内存的申请

2.系统有默认的无参空构造器（就是一个空的函数，里面什么也没有），一经自定义，则不再提供默认的【比如说一开始没有自定义的构造器，我们这样（Data data）就可以编译成功，因为它调用的是无参空构造器，但是我们自定义一个带参数的构造器之后，（Data data）就不可以编译成功，这就是一经定义，则不再提供默认的，相当于默认的失效了（那么这里可以有一种解决的办法，就是给自定义的构造器默认参数，这样Data data也可以成功编译）】

3.构造器可以重载

构造器析构器形参列表

析构器：1.与类同名，在名字前面加~，无返回值，无参数【所以不能重载】，在对象消失的时候【栈对象离开作用域就是消失，堆对象是delete的时候是消失】，自动被调用

2.功能：完成对象销毁前的处理工作，处理构造器申请的堆内存空间

3.没有指针要释放的时候，析构器为空就可以了

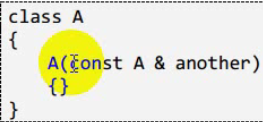
构造器参数列表：可以用构造器参数列表初始化，效率比较高【Data(int x,int y):a(x),b(y)（在还没调用函数的时候，就已经初始化了，因为没有this->）】【构造器参数列表初始化的顺序，与前后的赋值（a(x),b(y)）无关，但是与变量的创建有关（就是private里面变量的创建顺序）】

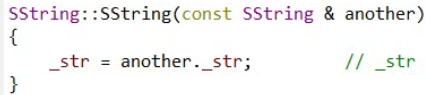
拷贝构造器

拷贝构造器：1.系统提供默认的拷贝构造器，一经自己自定义的拷贝拷贝构造器，系统提供的不复存在 2.系统提供的默认拷贝是等位拷贝，就是浅拷贝【类中包含的数据元素全部在栈上】

3.要实现深拷贝【有堆上的数据元素】，必须要自定义

拷贝构造器（由已经存在的一个对象拷贝给另一个对象）：1.与类同名，参数为本类对象的const引用

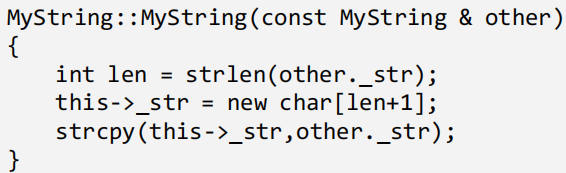




这个\_str是类中定义的private变量。

同类对象间无隐私

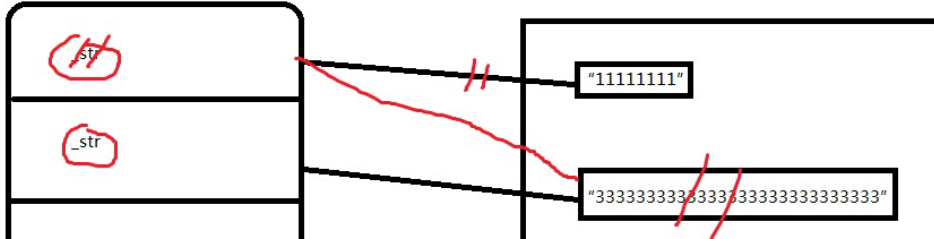
深拷贝：



赋值运算符重载

赋值运算符重载【两个都是已经存在的对象】：1.系统提供默认赋值运算符重载，一经自定义，不复存在

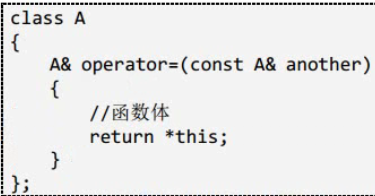
2.系统提供默认赋值运算符重载是等位拷贝，也就是浅拷贝，那么容易出现内存泄漏和重析构（原来它就指向一个，然后还没释放就重新指向另一个，那么这样释放的话，就把那个空间释放两次，重析构，而最开始的那个空间已经泄漏了）



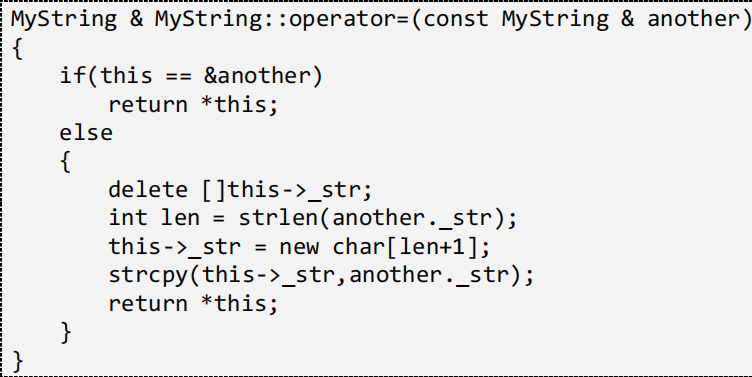
3.要实现深拷贝，需要自定义，那么深拷贝需要解决的问题（1.自赋值2.内存泄漏3.重析构）

4.返回值类型一定要是对象的引用

赋值运算符的重载的返回值类型一定是对象的引用，因为可以实现连续赋值



返回值是A&类型，可以实现连续的赋值



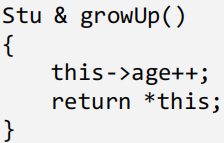
自赋值

this指针

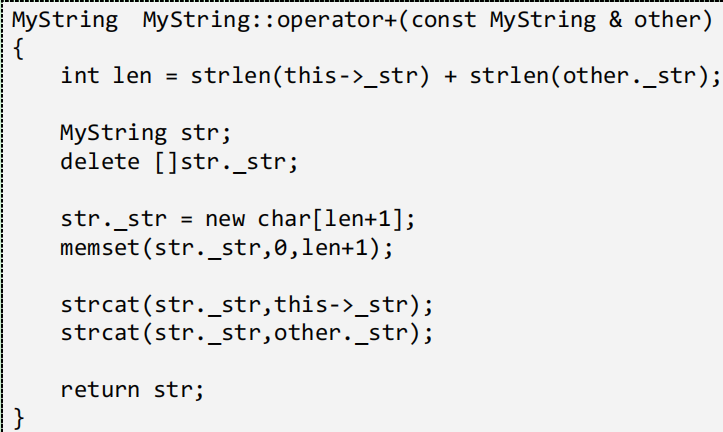
this：是指向当前对象的常指针（哪个对象调用这个函数，this就指向那个对象）

this：用途：1.用来区分成员和形参。如果形参的名字和私有成员的名字一样的时候，并且在函数里要赋值给私有成员的时候（Data(int a,int b){a=a;b=b;}），这样子赋值是会发生错乱的，所以就需要this指针（Data(int a,int b){this->a=a;this->b=b;}）

2.连续赋值。

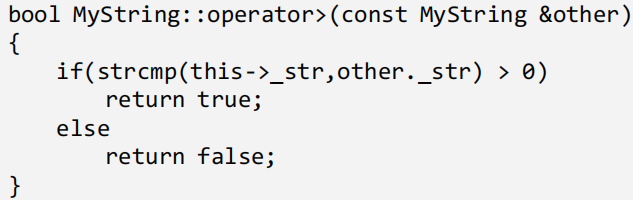


string类的算术运算

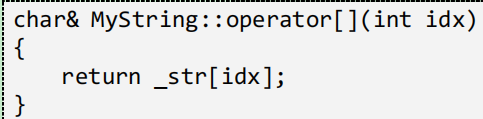


因为有默认的无参构造函数，所以说要先释放一下

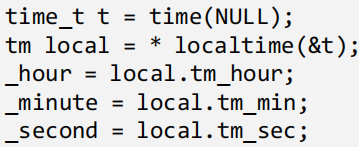
string类的逻辑运算符



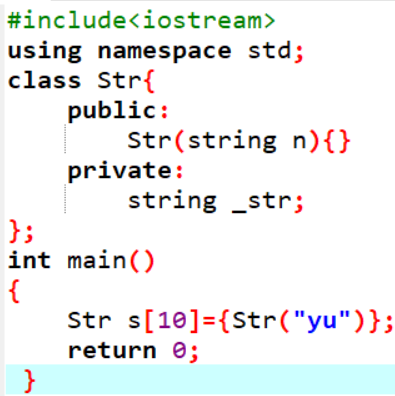
可以实现><和==



获取当前时间的函数



类对象构成的数组



所以我们在写构造函数的时候，不管有没有用到默认的构造函数，都把它包含进去，默认参数为空就是包含了

但是这种情况要怎么样编译才能过去呢，要给自定义的构造函数的参数默认为空（Str(string n=””）就可以编译过去了

这样的代码是编译不过去的

如果构造函数是默认无参的构造函数的话，那么是可以编译过去的，但是因为这个是自定义的构造函数，又因为是类对象构成的数组，但是它只初始化了一个元素，这种数组跟之前那种数组不一样，这种数组没有部分初始化这种功能，不会说初始化一个，接下去就全部默认为0这样子。

成员函数的存储方式

用类定义对象的时候，只为数据成员分配空间，而与成员函数无关，所以在sizeof(类名)的时候，它的大小是由类中的数据成员决定的【注意：static修饰的数据成员不占用空间，它存放在RW段，所以sizeof的时候，不把它算在内】

const修饰类成员

const修饰类成员：1.修饰数据成员：const修饰的成员变量变成成员常量，不可以修改，只能在初始化列表中赋值（就是构造函数的初始化列表【Str(string a):\_str(a)】（用上面的类举例），不可以在函数中赋值），可以被const和非const函数调用，并且不可以修改

2.修饰成员函数：（1）const放的位置有点不太一样，放在函数的末尾【void func() const】（声明和实现都需要加const）【放在函数前面的const是返回值，放在函数后面的是const函数】（2）const函数可以构成重载【void func()和void func() const】，那么const修饰的类对象只会调用const修饰的成员函数，因为只有const函数保证不会修改数据（不管这个数据有没有用const修饰，都不会修改）（3）const修饰的函数不能调用非const的函数，只能调用const修饰的函数，非const也可以调用const函数

3.修饰类对象：（1）const修饰类对象，该对象只能调用const修饰的函数（2）可以访问const和非const修饰的成员，并且不能修改

static修饰类成员

static修饰类成员【作用就是共享】：1.static修饰类数据成员：（1）定义在类里面（static int a），初始化只能在类外面（int a=10【初始化不需要带static】）【当数据成员既有const又有static的时候，只能在定义的时候初始化（const static int a=10或者static const int a=10）】（2）static修饰的成员变量不占用存储空间，它存放在RW段，静态数据成员既属于类又属于对象，能通过类名和对象名进行访问【调用类的静态数据成员，类名::类的静态数据成员，对象.类的静态数据成员】

2.static修饰成员函数（静态成员函数）：静态成员函数只能访问静态数据成员，静态成员函数属于类，而不属于对象，所以只能用类名调用，而不能用对象名调用初始化可以在类外面也可以在里面

静态成员函数只能访问静态数据成员。原因：非静态成员函数，在调用时 this 指

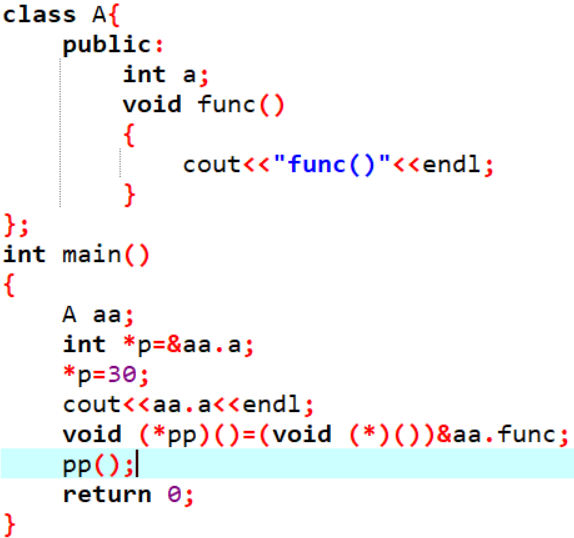
针时被当作参数传进。而静态成员函数属于类，而不属于对象，没有 this 指针。

指向类成员的指针

指向类成员的指针：1.指向类的静态数据成员和静态成员函数的指针：【C语言的用法】

2.指向类的成员函数和成员变量的指针：【C语言的用法】，这个函数用到了强转

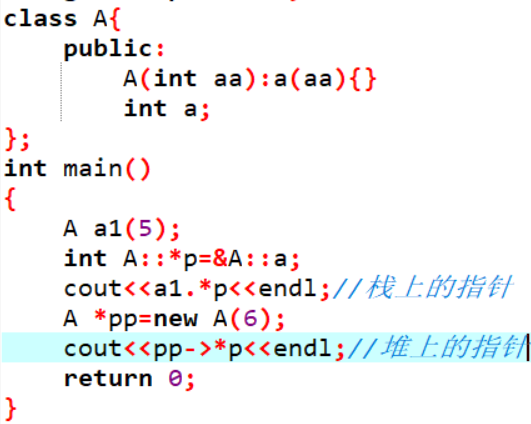
C语言的用法都是先有对象，然后指针指向对象的数据【C++的用法是不需要有对象，因为指针是指向类，所以用类名就可以了，然后在定义对象，通过对象调用指针来使用】



3.指向类的数据成员的指针：【C++的用法】（1）指针的定义：

<数据类型><类名>::\*<指针名>（2）指针的初始化：

<数据类型><类名>::\*<指针名>=&<类名>::（非静态数据成员）（3）指针的使用：<对象名>.\*<指针名>



4.指向类的成员函数的指针：【C++用法】：（1）指针的定义：

<类型>(<类名>::\*<指针名>)(<参数>)（2）指针的初始化：

<类型>(<类名>::\*<指针名>)(<参数>)=&<类名>::<非静态成员函数>（3）指针的使用：(<对象名>.\*<指针名>)()

指向非静态成员函数时，必须用类名作限定符，使用时则必须用类的实例【即对象】作限定符。

指向静态成员函数时，则不需要使用类名作限定符。

使用指向类的成员函数的指针可以提供更加隐蔽的接口：【创建一个函数指针数组】

#include<iostream>

using namespace std;

class A{

public:

A()

{

p[0]=&f;

p[1]=&g;

p[2]=&h;

p[3]=&i;

}

void select(int i,int val)

{

(this->\*p[i])(val);

}

private:

void f(int val){cout<<"void f() "<<val<<endl;}

void g(int val){cout<<"void g() "<<val<<endl;}

void h(int val){cout<<"void h() "<<val<<endl;}

void i(int val){cout<<"void i() "<<val<<endl;}

void (A::\*p[4])(int);

};

int main()

{

A a;

for(int i=0;i<4;i++)

{

a.select(i,1);

}

return 0;

}

友元

函数不是类的成员函数，但需要经常用到类的数据成员，又因为经常用类的成员函数得到类的数据成员的时候，这样子效率很低，所以就用友元函数【友元函数是可以直接访问类的私有成员的非成员函数。】

友元函数：1.优点：提高程序的运行效率（即减少了类型检查和安全性检查等都需要时间开销）2.缺点：破坏了类的封装性，使得非该类成员函数能访问类的数据成员

友元函数的注意点：（1）声明放在哪里都可以：public private protected（2）友元函数不能被继承（3）友元关系是单向的（A是B的友元，B不一定是A的友元）（4）友元关系不具有传递性（A是B的友元，C是A的友元，C不一定是B的友元）

友元函数的使用：（1）全局函数做友元函数：在函数声明的时候，函数名前加一个friend，【声明在类里面，实现在类外面，实现的时候不需要加friend】然后就相当于类的成员函数，通过该类的对象来访问数据成员（就是形参一般是引用该类的对象）

1. 类成员函数做友元函数：

#include<iostream>

using namespace std;

class Data;//要有Data的前向声明

class DataM{

public:

static void func1(Data &other);//要声明为静态函数，因为后面调用的时候，必须加上类名，所以说要声明为静态

};

class Data{

public:

Data(int aa)

:a(aa){}

friend void func(Data &other);

friend void DataM::func1(Data &other);//声明为友元函数的时候也要有作用域

private:

int a;

};

void DataM::func1(Data &other)//要有作用域，函数的实现要写在Data后面，而类的定义要写在Data前面

{

int b;

b=other.a;

cout<<b<<endl;

}

void func(Data &other)

{

int b;

b=other.a;

}

int main()

{

Data p(1);

DataM::func1(p);//因为必须要带类名来使用函数，所以把这个函数定义为静态，因为静态函数只能用类名来访问，不能用对象名来访问，用static修饰了函数属于类不属于对象，而用static修饰的成员变量既属于类又属于对象

return 0;

}

（3）友元对象（就是类做友元，可以访问private和protected）：友元类的所有成员函数都是另一个类的友元函数【实现：就是把一个类声明到另个一类里面，加上friend（friend class DataM）】

前向声明：是一种不完全的声明，只需要提供类名即可，不需要提供类的实现，所以1.不能定义类的对象（但是可以声明为类的形参）2.可以用于定义指向这个类的指针和引用

运算符重载

operator 运算符名称 在一起构成了新的函数名。

运算符重载有两种情况（当两者同时存在的时候，会出现二义性）：

1.友元+全局（也可以是局部）【因为是全局，所以在写的时候不用作用域了】（friend A operator+(const A &other,const A &another)）【双目运算符最好写成友元】

2.类成员函数（const A A::operator+(const A &other){return (this->x-other.y)【就是用this就可以】）

【二者同时存在时先调用成员函数】

因为这个运算符重载会出现一个问题，就是我们原本没重载之前（a+b）=5这种用法是错误的，但是重载之后是可以的，所以为了要让这个不能实现，在把重载函数弄为const类型就可以了

运算符重载规则：1.只能重载已有的运算符，不能自己定义一个

2.不能重载的运算符只有五个：成员选择符（.）成员对象选择符（.\*）作用域运算符（：：）条件运算符（？：）长度运算符（sizeof()）



3.重载不能改变运算符的操作数个数（所以重载运算符函数不能有默认的参数）

4.重载不能改变运算符的优先级

5.重载不能改变运算符的结合性

6.重载的参数至少有一个是类对象或者是类对象的引用，不能都是原来的int double float这样的类型（如果都是跟原来一样的，那就不用重载了哈哈）

7.=、&不必重载

重载+=：为了使（a+=b）+=c【a、b、c都是对象】能够成立，所有重载函数返回类的引用【A & operator+=(A&other)return \*this】【凡是有这种连等式的都要返回类的引用（++++）】

重载-（负号）：为了使-a=100不能成立，所以重载函数返回的是const对象，又因为const对象只能调用const函数，所以要加两个const【const A operator-() const{return A(-this->变量名)}】

重载++（前++）：它也是可以实现连等式（++++n可以），所以要返回类的引用【A& operator++(){this->变量名++;return \*this;}】【重载前++和后++如何区分呢，前++是这样的：A& operator++()，后++是这样的：const A& operator++(int)这个int叫做哑元，用来区分而已】

重载++（后++）：它实现不了连等式（n++++不可以），所以要返回const对象，因为const对象只能调用const函数，所以就是实现不了连等式，因为是后++，所以在写函数的时候，要写把值记下来，再去++，最后返回记下来的时候的值【const A operator++(int){A t(\*this);this->变量名++;return t;}】 这是一个拷贝构造函数初始化

重载>>（流输入）【要实现输入对象而不是变量】：因为要实现cin>>a>>b（a、b都是对象）【这个也可以用operator>>(operator>>(cin,a),b)】【这相当于连等式】，所以要返回istream的引用，所以是这样的【friend istream& operator>>(istream& in,A& c){in>>c.x>>x.y;return in}】，为什么要用友元函数呢，因为避免修改C++的标准库

重载<<（流输出）【要实现输出对象而不是变量】：因为要实现cout<<a<<b（a、b都是对象）【这个也可以用operator<<(operator<<(cout,a),b)】【这相当于连等式】，所以要返回ostream的引用，所以是这样的【friend ostream& operator<<(ostream& out,A& c){out<<c.x<<c.y;return out;}】，为什么要用友元函数呢，因为避免修改C++的标准库

只能重载为成员函数的运算符：=（赋值运算符）[]（下标运算符）（）（函数运算符）->（间接成员访问）

一个操作符的左右操作数不一定是相同类型的对象，这就涉及到将该操作符函 数

定义为谁的友元，谁的成员问题。【通常是左操作数的成员，右操作数的友元】

TypeCast

为什么要有转化函数：对于用户自定义的类型。编译系统则不知道怎么进行转化。解决这个问题的办法是，自定义专门的函数。

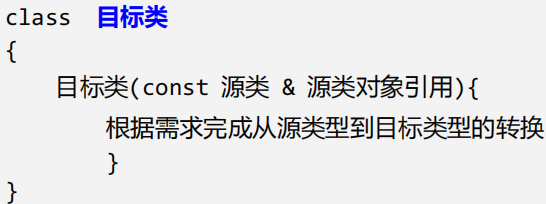
转化函数是一种特殊的构造函数，往往只有一个参数

转化函数的作用：在目标类内部，实现其它类或基本数据类到本类的转化

关键字explicit：显式的，就相当于要求要用强转的格式写出来

转化函数定义在目标类内部，是目标类的成员函数。

转化类的两种方法：1.用类型转换构造函数进行类型转换，其格式如下：



这种方法它也有两种方式：（1）隐式转换

#include<iostream>

using namespace std;

class P3D;

class P2D{

public:

P2D(int x,int y)

:a(x),b(y){

}

void func()

{

cout<<a<<","<<b<<endl;

}

friend P3D;

private:

int a;

int b;

};

class P3D{

public:

P3D(int x,int y,int z)

:a(x),b(y),c(z){

}

void func()

{

cout<<a<<","<<b<<","<<c<<endl;

}

P3D(P2D &p)//类型转化构造函数

{

this->a=p.a;

this->b=p.b;

this->c=0;

}

private:

int a;

int b;

int c;

};

int main()

{

P2D p2(1,2);

p2.func();

P3D p3(3,4,5);

p3.func();

P3D p3a=p2;

p3a.func();

return 0;

}

（2）显示转换：显示转换就要用到关键字explicit，并且格式就是相当于用强转的格式

#include<iostream>

using namespace std;

class P3D;

class P2D{

public:

P2D(int x,int y)

:a(x),b(y){

}

void func()

{

cout<<a<<","<<b<<endl;

}

friend P3D;

private:

int a;

int b;

};

class P3D{

public:

P3D(int x,int y,int z)

:a(x),b(y),c(z){

}

void func()

{

cout<<a<<","<<b<<","<<c<<endl;

}

explicit P3D(P2D &p)//显式转化

{

this->a=p.a;

this->b=p.b;

this->c=0;

}

private:

int a;

int b;

int c;

};

int main()

{

P2D p2(1,2);

p2.func();

P3D p3(3,4,5);

p3.func();

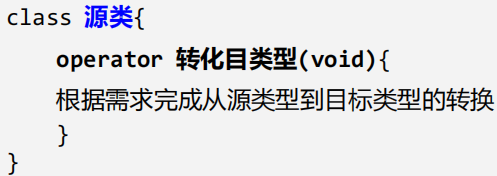
P3D p3a=static\_cast<P3D>(p2);显式转化需要用强转的格式

p3a.func();

return 0;

}

2.用类型转换操作符函数进行转换



#include<iostream>

using namespace std;

class P3D;

class P2D{

public:

P2D(int x,int y)

:a(x),b(y){

}

void func()

{

cout<<a<<","<<b<<endl;

}

operator P3D();//为什么实现要写在类外面呢，因为实现用到了类的构造函数，又因为上面类的前向声明，声明引用和指针而已，声明不了构造函数，所以要写在外面

private:

int a;

int b;

};

class P3D{

public:

P3D(int x,int y,int z)

:a(x),b(y),c(z){

}

void func()

{

cout<<a<<","<<b<<","<<c<<endl;

}

private:

int a;

int b;

int c;

};

P2D::operator P3D()

{

return P3D(this->a,this->b,0);//没有返回值的类型还可以return

}

int main()

{

P2D p2(1,2);

p2.func();

P3D p3(3,4,5);

p3.func();

P3D p3a=p2;

p3a.func();

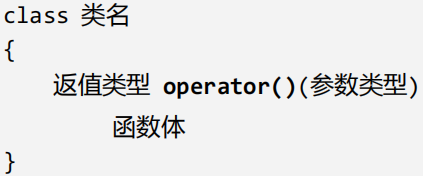
return 0;

}

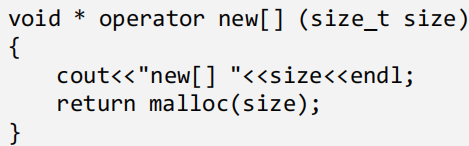
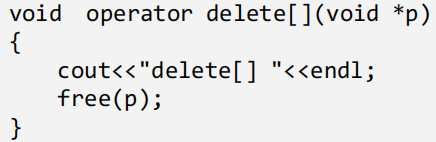
运算符重载提高

只要是对象进行加减，就一定都是对象，比如A=B+2；（A、B为同类的对象），那么这个2一定会用到这个类的构造函数进行转化

重载函数操作符（）【重载之后叫仿函数】：像使用函数名一样使用类名【比如本来是这样使用函数名func()，现在A()（参数）【A是类名】】



重载new和delete：size\_t就是unsigned int

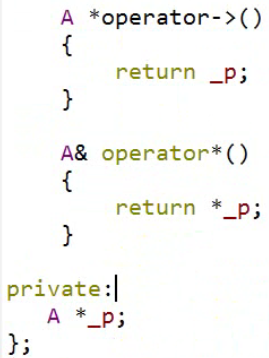
 

那么这个顺序是（比如A \*p=new A;delete p;）先到重载的new里面，再到构造函数里面，再到析构函数里面，到重载的delete里面【new会调用构造函数，new几个调用几次，delete会调用析构函数】

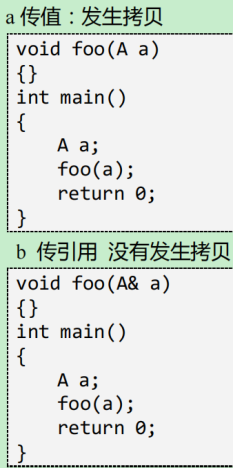
智能指针

我们平时的new都要自己delete，那么用智能指针的话，它自己就可以自动delete了，智能指针的格式：【auto\_ptr<类名> 对象名（new 类名）】（头文件memory），这个对象名具有指针的特点，比如说对象名是p，那么可以这样用p->、\*p，那么所以智能指针的本质是重载了->、\*

重载->、\*:



如何判断是否是调用构造函数还是拷贝构造函数：传值就是构造函数，传引用就是拷贝构造函数



继承方式

继承与派生（同一个问题的两种描述）：继承是站在儿子的角度说的，派生是站在父亲的角度说的【继承和派生的功能是实现代码的重用性】

父类（基类）与子类（派生类或者超类）：比如说一个是学生，一个是老师，它们的共同的有姓名、年龄，这些共同的就可以作为父类（基类），它们的不同的有学号、工号，这些不同的就可以作为子类（派生类），这就会实现代码的重用性

继承的格式：比如有一个父类Person，一个子类要继承父类，那么子类的格式为

class Student:public Person【这个public如果省略不写的话，默认是私有的】

继承方式和访问方式（二者是不同的概念）：继承方式有三种（public、private、protected），访问方式有三种（public、private、protected）【在访问方式上，protected和private是一样的】

public继承：父类是public（子类还是public，子类中能访问，也可以被子类的对象访问），父类是protected（子类还是protected，子类中能访问，不可以被子类的对象访问），父类是private（子类中还是private，子类中不可以访问，也不可以被子类的对象访问【相当于继承不了的】）

派生类的初始化

派生类不继承父类的构造器和析构器，因为自己有，那么自己的构造器在构造的时候，也是要对父类的数据成员进行初始化，那么怎么初始化呢【Student是父类】相当于是显式的调用父类的构造器【如果不是显式的，也就是把下面的Student（sn,ia,cs）删掉，它也会自动的调用父类的默认无参构造器或者是有默认参数的构造器（就是会调用就是了）】构造器和析构器的顺序是父构子构子析父析

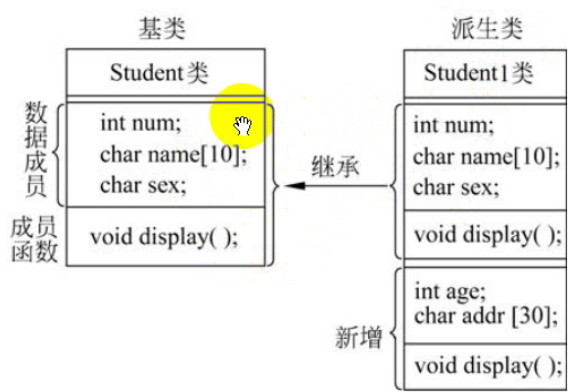


如果有多个父类，调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左到右）；

C++的特性：封装、继承、多态、模板

友元为了提高效率，static为了共享，const为了避免修改，继承为了代码的重用性

箭头是由派生类指向父类的



派生类的拷贝与赋值

子类的默认拷贝构造器（就是自己没写拷贝构造器）会调用父类的默认拷贝构造器或者自己写的拷贝构造器，如果子类中自己有写拷贝构造器，那么就要显式的调用父类的拷贝构造器

这个就是显式的调用父类的拷贝构造器，称为派生类的对象可以初始化父类的引用



student是父类

子类继承父类的时候，如果有成员函数的名字一样的时候，给父类的成员函数加一个

作用域运算符就可以了，因为是成员函数也是会继承下来的

子类的默认赋值运算符重载函数（就是自己没写赋值运算符重载函数）会调用父类的默认赋值运算符重载函数或者自己写的赋值运算符重载函数，如果子类中自己有写赋值运算符重载函数，不会再调用父类的默认的赋值运算符重载函数（注意：不会报错【没有下面的手动调用也不会报错】，如果是拷贝构造器就会报错【拷贝构造器需要显式的调用】），自实现需要手动的调用父类的赋值运算符重载函数（也就是自己写的）



这个就是手动的调用

派生类的析构顺序跟构造顺序相反：子类->成员->基类

继承方式和访问控制

shadow：就是父类中有一个函数，子类中有一个跟它同名的函数，那么子类对象调用这个函数的时候，父类中的函数就会被shadow（就是父类中的这个函数不被调用），如果非要调用父类的这个函数，使用作用域运算符就可以了【比如func（）是重名的这个函数，那么子类中func(){func()}，那么这样会造成死循环】

构成shadow的条件只有一个就是函数同名

解决shadow的方法：使用作用域运算符，两种情况：1.在子类中使用作用域【比如父类的名字为Father，函数是func()，那么就可以这样使用：Father::func()】

2.在子类的对象中使用作用域【比如子类是son，子类的对象为son s;那么就可以这样使用：son.Father::func()】

继承方式：

私有继承：父类中的public和protected在子类中仍然可以访问得到，但是子类的对象不可以访问

解释一下protected：比如说在父类的成员是protected，那么使用公有继承的话，在子类中可以访问到父类的protected的成员，但是子类的对象就不可以访问父类的protected的成员【图中的不可见的意思是连子类中也无法访问父类中的私有成员】



多继承问题

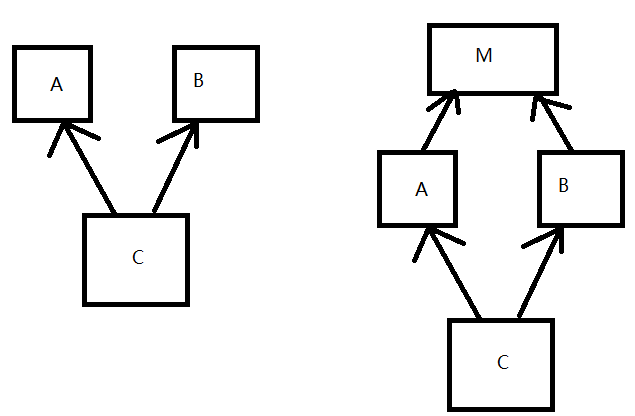
继承分为单继承和多继承，单继承就是一个父类，多继承就是多个父类

在继承的时候，每一个父类前面都要有继承的方式



在多继承中，有时会出现多个父类中有相同的数据成员（比如说data），那么在子类中如何解决呢？可以用提取公因式+虚继承来解决，那么这样它们都是操作同一个data了，相当于由三角问题变成了四角问题【成员函数也可以是这样用的】

A、B称为中间类，中间类的使用不受virtual的影响，像普通类一样正常使用，virtual在生成孙子类的时候生效



#include<iostream>

using namespace std;

class M{//提取公因式

M称为虚基类

public:

M(int i):data(i){}

int data;

};

class A:virtual public M{

public:

A(int i):M(i){}

void setData(int i)

{

data=i;

}

};

class B:virtual public M{

public:

B(int i):M(i){}

void getData()

{

cout<<data<<endl;

}

};

class C:public A,public B{

public://这里的A和B已经是形式了，没有意义，主要M起作用

C():A(2),B(3),M(4){}

void func()

{

cout<<A::data<<endl;

cout<<B::data<<endl;

}

};

int main()

{

C c;

c.func();

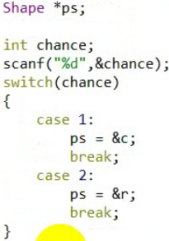
return 0;

}

虚继承和虚函数：虚继承解决了避免从父类中继承重名的成员的问题【子类继承父类，孙子类继承子类，那么孙子类就有两份爷爷类，所以就可以用虚继承】，虚函数解决了多态的问题【重写】

赋值兼容

函数重载也是一种多态的行为，称为静多态【在编译阶段就决定了调用情况】。动多态是在运行时发生的【下面那个就是动多态】



这个就是动多态，在运行的时候才决定调用情况

发出同一个信息，产生不同的行为就称为多态，比如说双击，有的打开QQ，有的打开微信，这就是多态的行为

声明型：virtual、static、friend【声明型就是说在类里面定义函数的时候需要加上，而在类外面不用加】

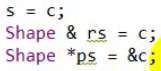
实现型：const【实现型就是说不管是在类里面还是在类外面都需要加上】

多态实现的前提：赋值兼容

赋值兼容是指在任何需要基类的地方都可以用公有派生类来替代，它是一种默认行为，不需要任何的显式转化步骤

赋值兼容规则：Shape是父类，Circle是子类，公有继承【Shape s,Circle c】

1. 派生类对象可以赋值给基类对象【s=c】
2. 派生类对象可以初始化基类的引用【Shape &rs=c】
3. 派生类对象的地址可以赋给指向基类对象的指针【Shape \*ps=&c】



多态形成的三要素

多态形成的条件：1、父类（既然是父类，那么肯定是有继承的）中有虚函数【虚函数：在函数前面加上virtual，在类里面写的时候要加virtual，在类外面就不用加virtual】

1. 子类override（覆写也就是重写）父类中的虚函数，【比如说基类中有一个函数virtual void draw()，那么子类中重写这个函数virtual void draw()，那么子类中的这个virtual可写可不写，但是最好写上，所以重写的函数也是虚函数，那么就意味着还可以被它的子类重写】
2. 通过已被子类对象赋值的基类指针【Shape \*pr=&r;】，调用共用接口【draw()就是共用接口】

区别多态和不是多态：

#include<iostream>

//这个不是多态

using namespace std;

class Shape{

public:

Shape(int x,int y)

:x(x),y(y){}

void draw()

{

cout<<x<<y<<endl;

}

protected:

int x,y;

};

class Rect:public Shape{

public:

Rect(int x,int y,int z)

:Shape(x,y),z(z){}

void draw()

{

cout<<x<<y<<z<<endl;

}

protected:

int z;

};

int main()

{

Rect r(1,2,3);

r.draw();//输出1，2，3

Shape \*pr=&r;

pr->draw();//输出1,2

return 0;

}

#include<iostream>

//这个才是多态

using namespace std;

class Shape{

public:

Shape(int x,int y)

:x(x),y(y){}

virtual void draw()

{

cout<<x<<y<<endl;

}

protected:

int x,y;

};

class Rect:public Shape{

public:

Rect(int x,int y,int z)

:Shape(x,y),z(z){}

virtual void draw()

{

cout<<x<<y<<z<<endl;

}

protected:

int z;

};

int main()

{

Rect r(1,2,3);

r.draw();//输出1，2，3

Shape \*pr=&r;

pr->draw();//输出1,2,3

return 0;}

重写成立的条件：函数和父类完全相同【函数名，函数返回类型、函数的参数要完全相同】，重写的函数加不加virtual都可以，最好是加上

纯虚函数

纯虚函数：virtual void func()=0，纯虚函数没有实现体【只有声明】

C++有开闭原则的规则：闭指的是高层业务的接口高度抽象，不要试图去修改它

开指的是新增的功能，就增加类来实现

含有纯虚函数的类称为抽象基类【上面的笔记有虚基类】，不能实例化【也就是不能定义对象】，但抽象类的作用是提供接口【作为一个接口】

#include<iostream>

using namespace std;

class interface{//抽象层模块

public://这个就是作为接口

virtual void readBook()=0;

};

class Book:public interface{

public://底层模块

void readBook()

{

cout<<"读书"<<endl;

}

};

class newspaper:public interface{//底层模块

public://新增的功能通过新增的类来实现

void readBook()

{

cout<<"读报纸"<<endl;

}

};

class Mother{

public:

void tellStory(interface \*b)//接口

//高层业务的接口高度抽象，不能动

//高层模块

{

b->readBook();

}

};

int main()

{

Book b;

newspaper n;

Mother m;

m.tellStory(&b);

虚析构函数在调用的时候，它会先把子类的虚函数执行一次，这样就是完整的析构，不然只是父类析构，子类并没有被析构

m.tellStory(&n);

return 0;

}

如果子类中没有重写纯虚函数的话，那么子类也是抽象基类，也不能实例化

凡是含有虚函数的类，其析构函数也要虚一下，不然析构对象的时候不是完整的析构【在析构(delete)父类指针的时候，才会调用子类的析构（父类A，子类B，A \*a=new B,delete a【析构函数有虚的话，子类才会析构，不然只是父类析构而已】）】【构造函数不能是virtual

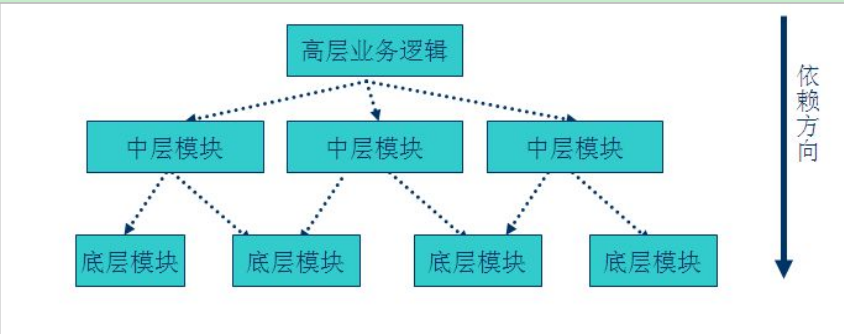
】

依赖倒置

虚函数的一些限制：1、只有类的成员函数才能声明为虚函数【因为虚函数在继承关系中才使用到，所以只有类的成员函数才能声明为虚函数，普通函数是不可以的】

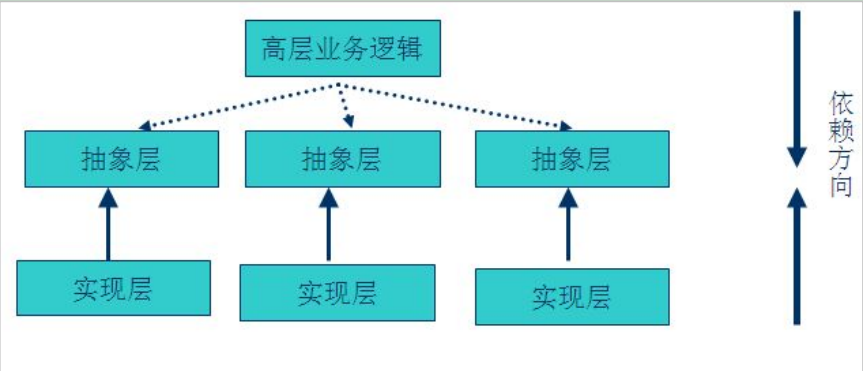
1. 静态成员函数不能声明为虚函数【因为虚函数是用在（把子类的对象赋给父类的指针【多态的时候】），而静态成员函数不受子类对象的限制，就可以访问，所以不可以声明为虚函数】【静态成员函数属于类而不是属于对象】
2. 内联函数不能声明为虚函数【因为内联函数的作用是用来替换的，所以不能声明为虚函数】
3. 构造函数不能声明为虚函数【因为在构造的时候，对象还没有完成创建，所以不可以声明为虚函数】
4. 析构函数可以声明为虚函数，并且通常是虚函数【因为析构函数是虚函数的话，父类子类才会都析构，不然只是父类析构而已】

依赖倒置的定义：高层模块不应该依赖于底层模块，二者都应该依赖于抽象，抽象不应该依赖细节，细节应该依赖于抽象



这个就是依赖倒置，二者都依赖于抽象，抽象不应该依赖于细节（高层和实现称为细节），细节应该依赖于抽象

这个是传统的依赖，高层模块依赖于中层模块，中层模块依赖于底层模块



采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本 Mother 类与

Book 类直接耦合时，Mother 类必须等 Book 类编码完成后才可以进行编码，因为 Mother 类依赖于 Book 类。修改后的程序则可以同时开工，互不影响，因为 Mother 与 Book 类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的 TDD 开发模式就是依赖倒置原则最成功的应用。

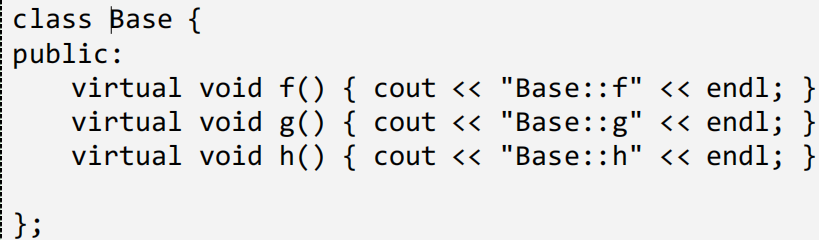
耦合度：相互之间的调用关系【耦合度高指的是相互之间的调用关系紧密】我们应该要低耦合高内聚

虚函数浅浅析

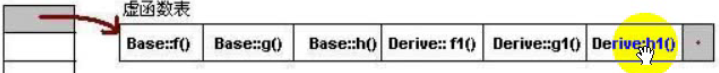
空的结构体或者是空的类，它的大小sizeof是1，因为有一个占位符

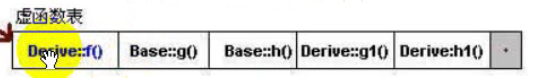
C++的多态是通过一张虚函数表来实现的【假设是下面这样的，然后这样的类的大小是4，因为需要有一个指针来指向这张虚函数表】这个表示Base类的，然后再有一个它的子类，子类中是完全新的虚函数，没有重写父类的，那么子类的虚函数就接在父类的虚函数表的后面。那么如果子类中的虚函数是重写父类的话，那么虚函数表中就会覆盖原来父类的虚函数所在的位置

我们可以看到，我们可以通过强行把&b 转成 int \*，取得虚函数表的地址，然后，再次取址就可以得到第一个虚函数的地址了，也就是 Base::f()



这是子类没有重写的情况





实现多态的根本原因：Base \*b = new Derive();b->f();因为子类重写了父类的虚函数，那么子类的虚函数就会排在虚函数表的最前面，所以调用的就是子类重写的虚函数，这就是实现了多态

虚函数表的最后多加了一个结点，这是虚函数表的结束

结点，就像字符串的结束符“/0”一样，其标志了虚函数表的结束。这个结束标志的值在

不同的编译器下是不同的。

这是子类有重写的情况



这是一个char \*里面的数组元素

二级指针char \*\*p，\*p是char \*的地址，\*\*p是“abcde”的地址

评价多态：1、实现了动态绑定 2、牺牲了一些空间和效率

为什么虚函数必须是类的成员函数？

因为虚函数的出现就是为了实现多态，所以在类外定义是没有意义的，所以必须是类的成员函数

为什么类的静态成员函数不能为虚函数？

因为类的静态成员函数属于类而不属于对象，如果被定义为虚函数，那么它就是动态绑定的，也就是说在派生类中可以被重写，这与它的定义是相互矛盾的

#include<iostream>

using namespace std;

class A{

public:

A()

{

this->func();//这个this就是b

}

~A()

{

this->func();

}

virtual void func()

{

cout<<"aaaaa"<<endl;

}

};

class B:public A{

public:

virtual void func()

{

cout<<"bbbbb"<<endl;

}

};

int main()

{

B b;//这个会输出两个aaaaa，第一个aaaaa是因为父类先构造，b的对象还没构造完成，所以调用父类的构造。第二个aaaaa是因为子类先析构，析构完了再析构父类的

return 0;

}

为什么构造函数不能为虚函数：

因为如果构造函数为虚函数的话，它将在执行期间被构造，而执行期则需要对象已

经建立，构造函数所完成的工作就是为了建立合适的对象，因此在没有构建好的对象上

不可能执行多态（虚函数的目的就在于实现多态性）的工作。在继承体系中，构造的顺

序就是从基类到派生类，其目的就在于确保对象能够成功地构建。构造函数同时承担着

虚函数表的建立，如果它本身都是虚函数的话，如何确保对象的构建成功呢？

注意：当基类的构造函数内部有虚函数时，会出现什么情况呢？结果是在构造函数

中，虚函数机制不起作用了，调用虚函数如同调用一般的成员函数一样。当基类的析构

函数内部有虚函数时，又如何工作呢？与构造函数相同，只有“局部”的版本被调用。但

是，行为相同，原因是不一样的。构造函数只能调用“局部”版本，是因为调用时还没有

派生类版本的信息。析构函数则是因为派生类版本的信息已经不可靠了。我们知道，析

构函数的调用顺序与构造函数相反，是从派生类的析构函数到基类的析构函数。当某个

类的析构函数被调用时，其派生类的析构函数已经被调用了，相应的数据也已被丢失，

如果再调用虚函数的派生类的版本，就相当于对一些不可靠的数据进行操作，这是非常

危险的。因此，在析构函数中，虚函数机制也是不起作用的。

函数模板和类模板

模板：templates

泛型：是指具有在多种数据类型【比如交换函数的重载，原本是int、double这种类型不一样的参数都要重载一次交换函数，但是使用了模板之后，就不用重载了，适用于多种数据类型】上皆可操作的含义（泛型编程的代表作品STL）

模板分为函数模板（交换函数）【函数模板不是函数，函数模板要经过实例化【函数模板会自己实例化Swap<int>(a,b)，但是类模板就需要自己实例化，否则编译不通过】才是函数。函数模板只适用于函数参数个数、类型一样，且函数体一样的情况】、类模板（栈的实现、vector[vector<int> a[10]）【类模板是类的抽象，类是类模板的实例化，类模板实例化【类要自己实例化，函数不需要，函数会自动实例化】之后才变为类】【如果类的成员函数要在类外面实现的话，就要在每个函数前面加上template<typename T>，声明自己是类模板的成员，并且在类的名字后面加上<T>，（比如template<typename T>bool Stack<T>::isEmpty()）否则编译不通过】

//函数模板

#include<iostream>

using namespace std;

template <typename T>//这个关键字可以是typename或者class

void Swap(T &a,T &b)//注意：只有同类型的才可以使用这个模板

{//这个函数模板，函数模板不是函数，函数模板要经过实例化才是函数

T t=a;//函数模板只适用于函数参数类型和个数相同，且函数体相同

a=b;

b=t;

}

int main()

{

int a=3,b=4;

Swap(a,b);

cout<<a<<b<<endl;

double aa=3.3,bb=4.4;

Swap(aa,bb);

cout<<aa<<bb<<endl;

string aaa="china",bbb="great";

Swap(aaa,bbb);

cout<<aaa<<bbb<<endl;

return 0;

}

//类模板

#include<iostream>

using namespace std;

template<typename T>

class Stack{

public:

Stack(int size)

{

data=new T[size];

top=0;

this->size=size;

}

~Stack(){}

void push(T ch)

{

data[top++]=ch;

}

T pop()

{

cout<<data[--top]<<endl;

}

bool isEmpty()

{

return top==0;

}

bool isFull()

{

return top==size;

}

private:

T \*data;

int top;

int size;

};

int main()

{

Stack<char>s(100);//这个就是类的实例化

if(!s.isFull())

{

s.push('a');

}

s.pop();

return 0;

}

文件操作从C到C++

用C++实现File \*fp=fopen(“aa.txt”,”w);

#include<iostream>

using namespace std;

class Stream{

public:

Stream(char \*a,char \*m)

{

fp=fopen(a,m);

if(NULL==fp)

{

cout<<"error"<<endl;

}

}

void func(char \*a,char \*m)

{

fp=fopen(a,m);

if(NULL==fp)

{

cout<<"error"<<endl;

}

}

void put(char ch)

{

fputc(ch,fp);

}

void close()

{

fclose(fp);

}

private:

FILE \*fp;

};

int main()

{

Stream s("aa.txt","w");

s.put('s');

s.close();

return 0;

}

构造函数不能是虚函数，析构函数通常要是虚函数，父类的构造函数和析构函数调用公共接口时，不会发生多态

几乎总是在头文件中放模板的声明和定义

C语言：FILE \*fp=fopen(“a.txt”,”w”); C++：fstream fp(“a.txt”,ios::out|ios::trunc);

文件操作

cin和cout不是函数，是对象【scanf和printf是函数】

IO对象不可复制和赋值，因为它的拷贝构造和运算符重载都是私有的【fstream f1,f2;f1=f2或者fstream f3(f2)】

IO对象是缓冲的

刷缓冲的几种情况：1，程序正常结束，作为 main 函数结束的一部分，将清空所有缓冲区。

2，缓冲区满，则会刷缓冲。

3，endl, flush 也会刷缓冲。cout<<”hi”<<endl;cout<<”hi”<<flush

4，在每次输出操作执行完后，用 unitbuf 操作符设置流的内部状态，从而清空缓冲区cout<<”hi”<<unitbuf

流对象的输入（>>）和输出（<<）都是重载的【fstream fs(“a.txt”,ios::in|ios::out|ios::trunc)，fs<<12<<endl；这是输入，fs>>a;cout<<a<<endl；这是输出】

文件存取方式：1、顺序存取，常用于文本文件

2、随机存取，常用于二进制文件

cout格式cinl输入

iomanip：流算子

强制显示小数点后面的数：cout<<setiosflags(ios::showpoint)<<2.0<<endl;输出2.0000

强制显示数的符号：cout<<setiosflags(ios::showpos)<<2<<endl;输出+2

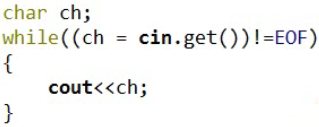
cout<<resetiosflags(ios::showpos)<<2<<endl;恢复原来的样子re

使用科学记数法输出：cout<<setiosflags(ios::scientific)<<103541354<<endl;

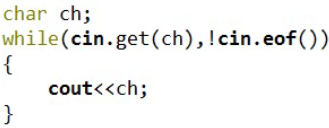
设置有效数字（并且自动实现四舍五入）：cout<<setprecision(2)<<12.6<<endl;输出13

cout的成员函数：ostream put(char)【cout.put(‘a’).put(‘b’);输出ab，它可以实现连等式，因为返回ostream】

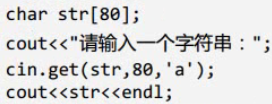
cin的成员函数：1、char get()【原本cin输入字符串的时候遇到空格、table、回车就会停止输入，这个get()都可以继续输入的，那么下面的循环的终止条件是！=EOF，按Ctrl+d或者Ctrl+z就是输入EOF】



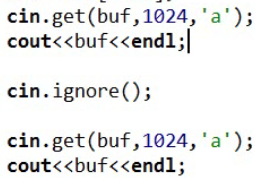
2、iostream get(ch)【原本cin输入字符串的时候遇到空格、table、回车就会停止输入，这个get()都可以继续输入的，那么下面的循环的终止条件是！=EOF，按Ctrl+d或者Ctrl+z就是输入EOF】【成功返回非0，失败返回0】【因为是iostream，所以可以实现连等式】



3、iostream cin.get(字符数组,字符个数n,终止字符)或者iostream cin.get(字符指针,字符个数n,终止字符）【原本我们输入字符串的时候遇到空格就会停止，那么这个不但不会停止输入，而且还可以自己指定终止字符】【最多读n-1个字符，遇到终止符既不吸收也不跳过】

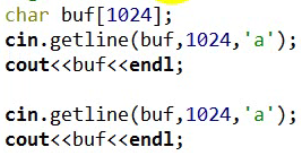


像这样子，一开始没有cin.ignore()的时候，我们输入bbbbbbabbbbbbba,它第一个的输出是bbbbbb，第二个的输出是输出一个换行，因为第二个一开始就碰到了a，那么第二次要如何跳过这个a呢，用cin.ignore()，就可以跳过，那么这样第二个的输出就会使bbbbbbb【最多读n-1个字符，遇到终止符既不吸收也不跳过】



4、cin.getline(字符数组或字符指针,字符个数,[终止字符])【这里有默认的终止符是空格，这个也可以自己修改】

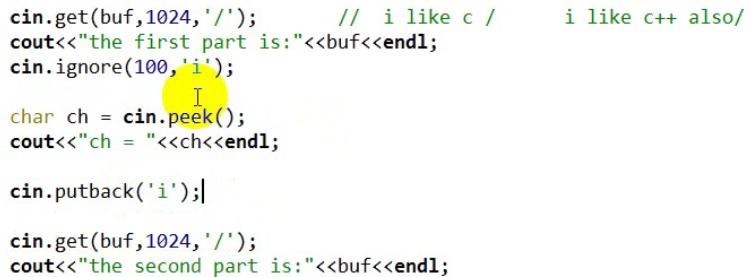
最多读n-1个字符，遇到终止符它可以跳过，所以就不用cin.ignore()【对比上面那个】



5、cin.ignore(n,终止字符)，跳过n个字符或者遇到了终止符【终止符是会跳过去然后停止】就不跳了，默认是跳过一个字符cin.ignore()这个是默认跳过一个字符

6、cin.peek()，查看当前的指针

7、cin.putback(‘i’)，把i插在当前指针前一个处



Ignore（）一直跳到i并且把i也跳过了，用peek（）查看当前指针是i后面的空格，用putback(‘i’)在当前空格前面插入i，相当于把原本的i补了回来

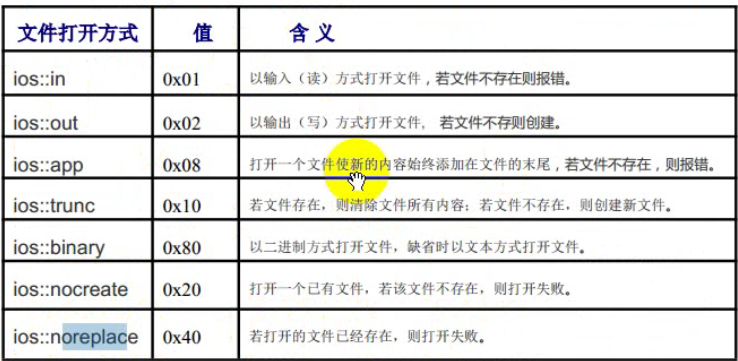
eof()判断文件的结束标志，未到文件尾返回 false，到文件尾返回 true。

读写文件



这边可以是fstream fs()在括号里面输入open函数的括号里面的，也可以用fs.open()函数打开文件，二者是一样的





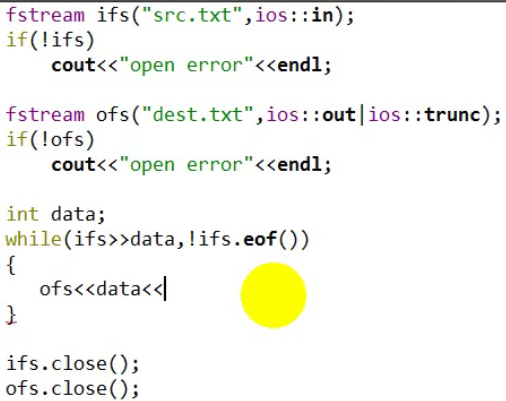
mode的方式

如果未指明以二进制方式打开文件，则默认是以文本方式打开文件。

if(cin)和while(cin)都是合法的【operator void \*() const;将流对象转化为void \*（隐式转化）】

If(!cin)和while(!cin)都是合法的【bool operator!() const;将流对象转化为bool型（隐式转化）】





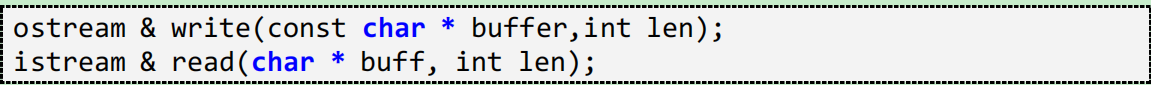
流对象，文件打开成功返回1，不成功返回0

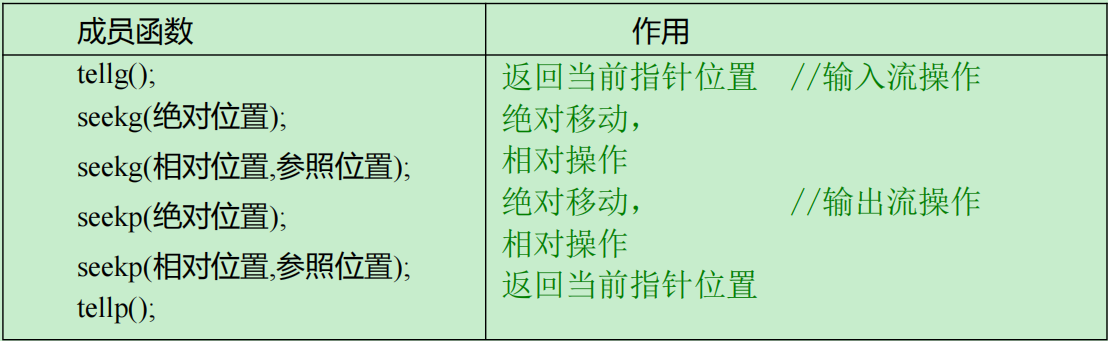
ofs<<data叫写入

Ifs>>data叫读出

读写文件二进制

二进制的读和写，记得要强转一下类型





ofs.seekg(0,ios::beg)：就是说我们写入结束的时候，指针在最后面，那么要读出的时候，要把指针移到前面

异常

try catch是不可以分开的，中间不能有任何东西

异常的特点：当一个函数设置异常的时候，被另一个函数调用，那么这时候异常仍然有效【异常可以跨越函数】

#include<iostream>

using namespace std;

int divide(int x,int y)

{

if(y==0)

{//异常处理，遇到异常的时候要

//抛出来

throw(10);

}

}

int main()

{//尝试着出现异常

try{

divide(4,0);

}catch(int e)//对应throw(10)

{//尝试了就要抓住异常，

//并且可以输出抛出来的异常

cout<<e<<endl;

}

catch(double e)//对应throw(10.1)

{

cout<<e<<endl;

}

catch(...){

cout<”不知道什么类型”<<endl;

}

//这里的catch如果找不到，系统会调用terminate，终止程序

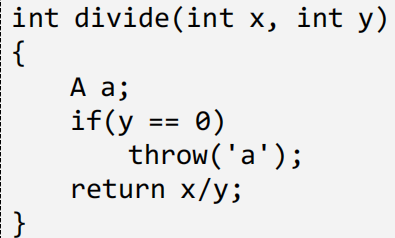
}

抛出异常类型声明：void func() throw(A,B,C,D)；：声明能抛出A、B、C、D类型的异常

void func()；：可以抛出任何类型的异常

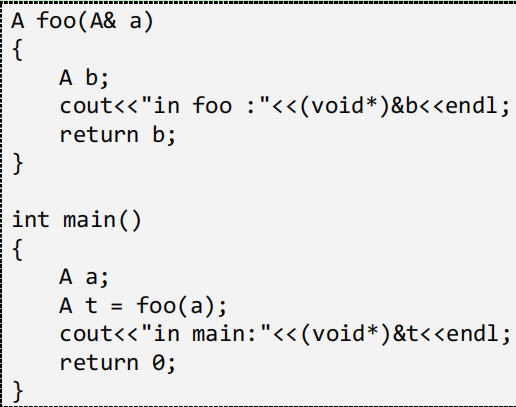
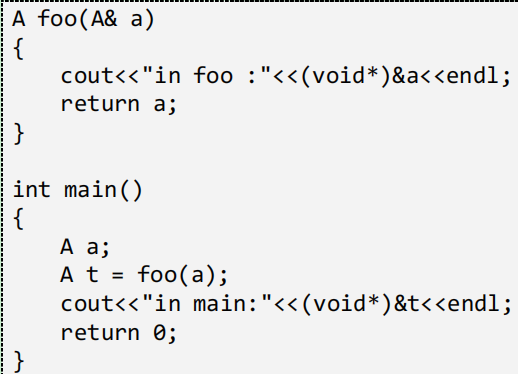
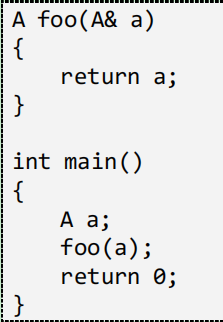
void func() throw()；：不抛出异常

栈自旋：就是说在异常抛出前面【throw】有定义类的对象的时候，它会自己调用析构函数，并且析构函数的顺序和构造函数的顺序相反【堆不会有自旋，也就是说堆不会析构】



异常如果是抛出对象的话【A a;（构造）throw(a);（拷贝构造）】会构造两次，析构两次，先析构一次，抛出异常后再析构一次

看书的补充



两次构造，两次析构

一次构造，一次拷贝，两次析构

一次构造，一次拷贝，两次析构

创建对象数组的时候，不能指定数组元素的个数和部分初始化，会报错【称为栈对象数组】

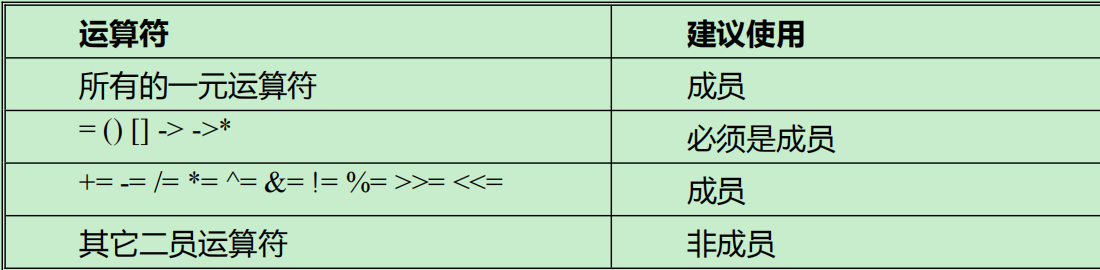


创建对象指针的时候，可以指定数组元素的大小，但指定的大小一定要初始化，不然报错【称为堆对象数组】



不管是栈对象数组还是堆对象数组，定义的时候没有初始化，就会自动调用无参构造器，或者手动调用带参数的构造器，所以在重载或者是默认参数的时候，一定要把无参构造器包含进来，不然还得自己手动调用

extern”C”：为什么会有这个东西呢？因为C++为了实现重载，所以采用的命名倾轧，因为这一种机制的原因，C++对于C的兼容，所以有了这个东西



像这种，它虽然在类B里面new的，但是本质还是类A

#include<iostream>

#include<memory>

using namespace std;

class A{

public:

A()

{

cout<<"Astructor"<<endl;

}

~A()

{

cout<<"Adestructor"<<endl;

}

void func()

{

cout<<"haha"<<endl;

}

};

class B{

public:

B(A \*pp)

:p(pp)

{

cout<<"Bstructor"<<endl;

}

~B()

{

cout<<"Bdestructor"<<endl;

}

A\* operator->()

{

return p;

}

A& operator\*()

{

return \*p;

}

private:

A \*p;

};

int main()

{

A \*p=new A;

B b(new A);

(\*b).func();

b->func();

return 0;

}

