C++学习笔记

# 一、C++与C语言的区别

## 形式上:

总体上来说，C语言是C++语言的一个子集。C++语言包含了C语言的全部词法和语法的内容，同时它增加了面向对象程序设计语言所必备的内容。C为面向过程的语言，C++为面向对象的语言。

C++语言源程序中比C语言多一种数据类型----**-类**，也就是多一种变量-----**对象**。

1、C++之类的语言是面向对象的。使用这样的语言编程被称为面向对象编程（Object-Oriented Programming,OOP。它可以让计算机程序员将一个面向对象的设计实现称为一个可工作的软件系统

2、C之类的语言是过程性（Procedural），所以编程总是面向动作的。

3、在C语言中，编程的单位是函数（function）,而在C++中，编程的单位是类（class），类最终要实例化为对象。C++的类包含了实现操作的函数和实现属性的数据。

4、**C语言的程序员把重点放在了函数的编写上，程序员将完成某任务的动作组织成函数，再由函数组成程序**。数据在C语言中的确很重要，但是其中的滚点确是数据主要为支持函数执行的动作而存在的。系统规格说明书中的动词可以帮助C语言程序员确定用于实现系统的一组函数集

5、**C++语言的程序员将重点放在创建自己的用户自定义类型上，这些用户自定义类型称为类，每个类都包含数据和函数集，这些函数操作数据并向客户提供服务，类的数据称为数据成员，类的函数成分称为成员函数**，在其他面向对象程序设计语言中称为方法，系统规格说明书中的名词有助于C++程序员确定实现系统所需的用来生成对象的一组类。

### 内容上：

### ****1.引入命名空间****

一个中大型软件往往由多名程序员共同开发，会使用大量的变量和函数，不可避免地会出现变量或函数的命名冲突。当所有人的代码都测试通过，没有问题时，将它们结合到一起就有可能会出现命名冲突。为了解决合作开发时的命名冲突问题，C++ 引入了命名空间（Namespace）的概念。

命名空间将全局作用域分成不同的部分；不同命名空间中的标识符可以同名而不会发生冲突；命名空间可以相互嵌套；全局作用域也叫默认命名空间。

使用using编译指令可以省略每次引用名称空间中的名称时必须加上的名称空间名，可以比较方便地引用名称空间中的名称。例如 “using namespace std;”如果不使用using，每次使用cout等std名称空间中的名称时，就必须写成“ std::cout<<” xx” ”, 使用using语句可以省去“ std:: “前缀。这是一种偷懒的做法，在大型项目中是一个潜在的问题。更好的方法是只使用using声明所使用的名称，而不是直接声明整个名称空间，例如要使用cout，先只声明“ using std：：cout; ”。

### 2.实用性加强

C语言中的变量都必须在作用域开始的位置定义！C++中更强调语言的“实用性”，所有的变量都可以在需要使用时再定义。例如： for (int i = 0; i < 10; i++)。

### 3.register关键字

register关键字请求“编译器”将局部变量存储于寄存器中。

C语言中无法取得register变量地址。

在C++中依然支持register关键字。

C++编译器有自己的优化方式，不使用register也可能做优化。

C++中可以取得register变量的地址。

C++编译器发现程序中需要取register变量的地址时，register对变量的声明变得无效。

### 4.变量监测加强

在C语言中重复定义多个全局变量是合法的，这些同名的全局变量最终都会被链接全局数据区的同一个地址空间上。

在C语言中只有一个全局作用域，C语言中所有的全局标识符共享同一个作用域，标识符之间可能发生冲突 。

在C++中**不允许**定义多个同名的全局变量。

### 5.struct类型加强

C语言的struct定义了一组变量的集合，C编译器并不认为这是一种新的类型。

C++中的struct是一个新类型的定义声明。

C语言中 struct 定义了一组数据的集合，而不是一种新的数据类型。所以在定义变量的时候需要在前面加上 struct 关键字进行修饰。

C++中 struct 定义了一种新的数据类型，可以直接用来定义变量。

### 6.c++中所有变量和函数都必须要有数据类型

**c语言：**

**//1 函数可以没有返回参数类型**

f()

{

printf ("hello world\n");

}

**//2 函数参数可以没有数据类型**

g(i)

{

return i;

}

int main()

{

**// 3 没有参数的函数可以接收任意个参数**

f(1,2,3,4,5);

g(12,15);

return 0;

}

**c++加强：**

**在C语言中：**

int f( )；表示返回值为int，接受任意参数的函数。

int f(void)；表示返回值为int的无参函数。

**在C++中：**

int f( );和int f(void)具有相同的意义，都表示返回值为int的无参函数。

**C++更加强调类型，任意的程序元素都必须显示指明类型**

### 7.新增bool类型关键字

**C++中的布尔类型**

C++在C语言的基本类型系统之上增加了bool。

C++中的bool可取的值只有true和false。

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

true代表真值，编译器内部用1来表示；false代表非真值，编译器内部用0来表示。

bool类型只有true（非0）和false（0）两个值。

C++编译器会在赋值时将非0值转换为true，0值转换为false。

### 8.三目运算符的加强

1）C语言返回变量的值，C++语言是返回变量本身。

C语言中的三目运算符返回的是变量值，不能作为左值使用。

C++中的三目运算符可直接返回变量本身，因此可以出现在程序的任何地方。

2）注意：三目运算符可能返回的值中如果有一个是常量值，则不能作为左值使用

(a < b ?1 : b )= 30; 这里1为常量，不能作为左值使用。

3）C语言如何支持类似C++的特性呢？

当左值的条件：要有内存空间；C++编译器帮助程序员取了一个地址而已。

# 二、C++学习零散知识笔记

#### 1、C++ 11初始化

C++ 11初始化时，可以将大括号初始化用于单值变量。采用这种方式时，可以使用等号（=），也可以不使用：

int emus{7}; //set emus to 5

int rheas={12}; //set rheas to 12

其次，大括号内可以不包含任何东西，在这种情形下，变量将被初始化为零：

int rocs={}; //set rocs to 0

int psychics{}; //set psychics to 0

第三，这有助于更好地规范类型转换错误。

#### 2、选择数据类型

在选择数据类型时，注意选择数据类型，尤其是需要考虑内存空间和不同系统移植时的数据类型的长度可能会发生变化。请记住：节省一点就是赢得一点。

char最常被用作处理字符，但是也可以将它用作比int更小的整型。

#### 3、表示特殊字符

C++标准除了支持一个基本的源字符集（即可以用来编写源代码的字符集），还有一种表示特殊字符的机制，它独立于任何特定的键盘，实用的是通用字符名。

通用字符名的用法类似于转义序列，通用字符名可以以 \u 或 \U 打头。\u 后面是8个十六进制位，\U后边是16个十六进制位。这些位表示的是自负的ISO10646码点（ISO10646是一种正在指定的国际标准，为大量的字符提供了数值编码）。比如ö的ISO10646码点为00F6，而â的码点为00E2.

#### 4、wchar\_t（宽字符类型）

8位char可以表示基本字符集，另一种各类型wchar\_t（宽字符类型）可以表示扩展字符集。这是一种整数类型，它有足够的空间，可以表示系统使用的最大扩展字符集。

cin 和 cout 将输入和输出看作是char 流，因此不适用于处理wchar\_t类型。Iostream头文件的最新版本提供了作用相似的工具----wcin 和wcout，可以用来处理wchar\_t流。另外，可以通过加上前缀L来只是宽字符常量和宽字符串：

wchar\_t bob=L’p’; //a wide-character constant

wcout<< L”tall”<<endl; //outputting a wide-character string

C++11新增了类型char16\_t和char32\_t，两者都是无符号的，前者长16位。后者长32位。C++11使用前缀u来表示char16\_t字符常量和字符串常量，如u’C’和u”be good”；并使用前缀U表示char32\_t常量。

类型char16\_t与\u00F6形式的通用字符名匹配，而类型char32\_t与\U0000222B形式的通用字符名匹配:

char16\_t ch1=u’q’; // basic character in 16-bit form

char32\_t ch2=U’\U0000222B’; // universal character name in 32-bit form

#### 5、const限定符

关键字const叫做限定符，因为它限定了声明的含义。一种常见的做法是将名称首字母大写，以提醒某名称是个常量，比如Month。另一种约定是将整个名称大写，使用#define创建常量时通常使用这种约定。

如果在声明常量时没有提供值，则该常量的值将是不确定的，且无法修改。

#### 6、强制类型转换

C++中强制转换的通用格式如下：

*(typeName) value* // converts value to typeName type

*typeName (value)*  // converts value to typeName type

C++还引入了4个强制类型转换运算符，对它们的使用更加严格。其中static\_cast<>可以将一种数值类型转换为另一种数值类型。例如可以像下面这样将thorn转换为long类型：

static\_cast<long> （thorn）；

#### 7、字符串输入

##### getline()和get()

getline()和get()都读取一行，直到到达换行符。然而，随后getline()将丢弃换行符，而get()将换行符保留在输入序列中。

getline()函数读取整行，通过回车键输入的换行符来确定输入结尾，要调用这种方法，可以使用cin.getline()，**该函数有两个参数，第一个存储输入行的数组名称，第二个参数是要读取的字符数**，例如：” cin.getline(name,20); ”.

cin.get ()也是两个参数，如cin.get (name，20)，和cin.getline()接受的参数相同，解释参数的方式也相同，而且都读到行尾，但是cin.get()将保留换行符，如果两次连续调用该函数，第二个函数将读取第一个函数读取后剩余的行尾那个换行符，不能读取下一次的输入。所以要将第一次的cin.get ()接收后剩余的换行符用其他方法接收掉。可以用不带任何参数的cin.get ()，其可以读取且只读取一个字符。

##### string类

使用string类头文件中必须包含头文件string。String类位于名称空间std中，因此必须提供一条using指令，或者使用std：：string来引用它。

#### 8、结构

##### 定义

C++中，比如已经定义了结构类型：

struct inflatable

{

char name[20];

float volume;

double price;

}

再次使用inflatable定义结构体变量的时候，类比于C语言，可以省略struct字符，如下：

inflatable hat； //hat是一个inflatable类型的结构体变量。而在C中，需要写成：struct inflatable hat；

##### 初始化

初始化和C语言中初始化可以一样，也可以省略“=”。例如：

inflatable hat={“Dapha”, 0.12, 0.25};

inflatable hat {“Dapha”, 0.12, 0.25}; 只在C++中允许，C中不允许。

而“inflatable hat {};”语句导致hat.valume和hat.price都将被置为0，hat.name 每个字节都被设置为0。

也可以用string对象作为成员，比如先声明了string类型，则可以将“char name[20];”换为“string name;”

##### 结构数组

可以这样定义一组结构数组：inflatable gifts[100]； 这样每个元素gifts[i]都是inflatable的对象，具有inflatable的结构类型。并且可以在定义时对结构数组进行初始化，类似于字符数组的初始化。

#### 9、共用体

共用体（union）是一种数据格式，它也能存储不同的数据类型，其数据结构和“结构”一样。只是只能同时存储其中的一种类型。如下例子：

Union one4all

{

Int int\_val;

Long long\_val;

Double double\_val;

};

可以使用one4all来存储以上三种类型变量，条件是在不同的时间进行。如果先存储了一个int类型变量，如果再输入一个long类型变量，则之前的int类型变量将丢失。由于每次只是存储一个值，所以它必须有足够的空间来存储最大的成员。所以共用体的长度为其最大成员的长度。

#### 10 函数传参的三种方式

参考博客原文：https://blog.csdn.net/qq\_39539470/article/details/79519156

函数传参有三种传参方式：传值、传址、传引用**。**

1、按值传递

（1）形参和实参各占一个独立的存储空间。

（2）形参的存储空间是函数被调用时才分配的，调用开始，系统为形参开辟一个临时的存储区，然后将各实参传递给形参，这是形参就得到了实参的值。

#include<stdio.h>

void swap1(int x, int y)//定义中的x,y变量是swap函数的两个形参

{

int tmp;

tmp = x;

x = y;

y = tmp;

printf("x=%d,y=%d\n", x, y);

}

int main()

{

int a = 2;

int b = 3;

swap1(a, b);//a,b变量为swap函数的实际参数

printf("a=%d,b=%d", a, b);

return 0;

}

输出结果为：x=3,y=2; a=2,b=3

代码分析：你也许会疑惑，a和b的值明明在函数swap中交换，为什么输出值却没变呢？

那是因为你还没理解形参和实参的关系。

为了解释这个问题，我先给出如下代码：

int a=2;

int x;

x=a;

x=x+3;

最终的结果是不是a=2，x=5

这段代码我们需要理解a虽然赋值给x，但是a的值并没有改变，对x的任何修改都不会改变a的值。

而代码中值传递方式如下：

main函数中调用了swap函数;swap中包含了以下代码

int x = a;

int y = b;/\*前两行代码是函数的隐含操作不可写在代码之中，

是为了方便大家理解我自己加上的\*/

int tmp;

tmp = x;

x = y;

y = tmp;

printf("x=%d,y=%d\n", x, y);

以上的分析中我们可以看出函数是通过赋值把a,b赋给x,y，这是一个隐含操作，我们不能把它显式的写出来，进行函数中变量的值进行交换时只是形参x,y的交换，并没有对实参进行真正交换，所以a,b值不变。

2、地址传递

地址传递与值传递的不同在于，它把实参的存储地址传送给形参，使得形参指针和实参指针指向同一块地址。因此，被调用函数中对形参指针所指向的地址中内容的任何改变都会影响到实参。

示例代码：

void swap2(int \*px, int \*py)

{

int tmp;

tmp = \*px;

\*px = \*py;

\*py = tmp;

printf("px=%d,py=%d\n", \*px, \*py);

}

int main()

{

int a = 2;

int b = 3;

swap2(&a, &b);/\*调用了swap函数，同样也有隐含动作px=&a;py=&b;\*/

printf("a=%d,b=%d", a, b);

return 0;

}

运行结果为\*px=3,py=2 a=3,b=2;

代码分析：有了两行隐含赋值操作，我们可以清晰的看出指针px,\*py是对变量a,b的值操作。函数里面对a和b的值进行了交换。这就是传址。

3、引用传递

引用传递是以引用为参数，则既可以使得对形参的任何操作都能改变相应数据，又使函数调用方便。引用传递是在形参调用前加入引用运算符“&”。引用为实参的别名，和实参是同一个变量，则他们的值也相同，该引用改变则它的实参也改变。

代码示例

#include<stdio.h>

void swap3(int &x, int &y)

{

int tmp = x;

x = y;

y = tmp;

printf("x=%d,y=%d\n", x, y);

}

int main()

{

int a = 2;

int b = 3;

swap3(a, b);//调用方式与传值一样

printf("a=%d,b=%d", a, b);

system("pause");

return 0;

}

输出结果：x=3,y=2; a=3,b=2;

代码分析：我们看到该代码只与传值中swap函数定义不同，swap3中参数都加了取地址符号&，有了这个函数会将a,b分别替代了x,y，这样函数里面操作就是a,b本身了。

4、对于值传递和引用传递的比较：

在C语言中，并没有引用的概念，引用是C++的概念。

在C++的函数参数中，有两种形式，其中 TYPE & var的形式，称为传引用方式； TYPE var的形式，称为传值。

二者的区别为，当传引用时，实际传到函数中的形参，是实际参数的一个引用，而不是仅传递值到函数中。具体的表现有以下几条：

1 传引用时，形参和实参是同一个变量，即使用相同的内存空间，二者有相同的地址。而传值时二者地址不同；

2 传引用时，由于没有新建变量，所以对于类对象参数，不会产生构造和析构。而如果是传值调用，调用时会进行构造，退出函数时会进行析构；

3 由于传引用使用的是原本实参的地址，所以对引用参数值的修改，会在退出函数后体现在主调函数中，而传值调用对参数的修改不会影响到主调函数。

### 11、int和string类型相互转换

#### 一、int转换成string

##### Ⅰ、to\_string函数

c++11标准增加了全局函数std::to\_string:

需要包含头文件 <string>

string to\_string (int val);

string to\_string (long val);

string to\_string (long long val);

string to\_string (unsigned val);

string to\_string (unsigned long val);

string to\_string (unsigned long long val);

string to\_string (float val);

string to\_string (double val);

string to\_string (long double val);

例如：

int n = 123;

string s = to\_string(n); //s就变成了字符串“123”

##### Ⅱ、借助字符串流

该方法麻烦！

　　标准库定义了三种类型字符串流：istringstream,ostringstream,stringstream，看名字就知道这几种类型和iostream中的几个非常类似，分别可以读、写以及读和写string类型，它们也确实是从iostream类型派生而来的。要使用它们需要包含#include <sstream>头文件。

除了从iostream继承来的操作

　　1.sstream类型定义了一个有string形参的构造函数，即：  stringstream stream(s); 创建了存储s副本的stringstream对象,s为string类型对象

　　2.定义了名为str的成员，用来读取或设置stringstream对象所操纵的string值：stream.str(); 返回stream中存储的string类型对象stream.str(s); 将string类型的s复制给stream，返回void

**Example：**

int aa = 30;

stringstream ss;

ss<<aa;

string s1 = ss.str();

cout<<s1<<endl; // 30

#### 二、string转换成int

##### Ⅰ、采用标准库中atoi函数

对于其他类型也都有相应的标准库函数，比如浮点型atof(),long型atol()等等

Example：

**std::string str = "123";**

**int n = atoi(str.c\_str());**

**cout<<n; //123**

##### Ⅱ、采用sstream头文件中定义的字符串流对象来实现转换

istringstream is("12"); //构造输入字符串流，流的内容初始化为“12”的字符串

int i;

is >> i; //从is流中读入一个int整数存入i中

# Const

Const指针可以接收const和非const数据或指针，但是非const指针只能接收非const数据或者指针（除非用强制类型转换），所以应该尽可能的使用const，将指针参数声明为指向常量（const）数据的指针。

好处：1 可以避免由于无意间修改数据而导致的变成错误

2 使用const使得函数能够处理const和非const参数，否则只能接收非const数据

如果条件允许，则应将指针形参声明为指向const的指针。

Const位置不同意义不同：

const 在 \* 左侧，则修饰的是指针指向的常量：

char const \* p = &a; 和 const char \* p = &a; 是一样的，const都在 \* 左侧，修饰指针P指向的常量a，即a的值不可修改。

const 在 \* 右侧，则修饰的是指针本身：

char\* const p = &a; p的值，即p指向的地址不能修改，而a的值可以修改。

例子：

Int nake=5;

Int sloth=3;

Const int \*ps=&sloth; //ps指针（非const指针）指向const常量，可以将新地址赋予ps指针，但是不能修改ps指向的常量，例如 ps=&nake; 是合法语句，但是 \*ps=5； 是非法语句；

Int \* const finger=&sloth; //finger为const指针，不能改变finger指向的地址，可以改变finger指向的地址处存储的数值，例如，语句 finger=&nake； 是非法的，而 \*finger=5；是合法的。

Const int \* const stick=&nake； //这时stick和\*stick都是const类型，所以stick只能指向nake，也不能通过stick来修改nake的值，所以，

Stick=&sloth; 语句和 \* stick=3; 语句都是非法的。

# 函数重载

C++的**参数名**不能相同，但是**函数名**却可以相同，在同一个文件内，可以使用同一个函数名表示不同的函数功能，其关键点就是同名函数的参数一定不能相同，否则就是同一个函数了，也就是虽然函数名相同，却可以通过判断所给参数的不同，来调用不同函数实现不同功能。如下例子：虽然都是MAX函数，由于参数不同，调用时就会调用不同函数。

#include <iostream>

**u**[**sin**](http://c.biancheng.net/ref/sin.html)**g** **namespace** std;

void Max(int a, int b)

{

cout << "Max 1" << endl;

}

void Max(double a, double b)

{

cout << "Max 2" << endl;

}

void Max(double a, double b, double c)

{

cout << "Max 3" << endl;

}

int main()

{

Max(3, 4); //调用 int Max(int, int)

Max(2.4, 6.0); //调用 double Max(doubleA double)

Max(1.2, 3.4, 5); //调用 double Max(double, double, double)

Max(1, 2, 3); //调用 double Max(double, double, double)

Max(3, 1.5); //编译出错：二义性

**return** 0;

}

使用条件：仅当函数基本上执行相同任务，但是使用不同参数时，才使用函数重载

# 函数模板

函数模板就是指通用函数，也就是使用泛型来定义函数，而不是具体的某一种类型，而泛型可以使用具体的类型（如int double等）来代替。通过将类型作为参数传递给模板，可使便以其声称该类型的函数。

有了模板，可以只写一个 Swap 模板，编译器会根据 Swap 模板自动生成多个 Sawp 函数，用以交换不同类型变量的值。

在 C++ 中，模板分为函数模板和类模板两种。函数模板是用于生成函数的，类模板则是用于生成类的。

函数模板的写法如下：

template <class 类型参数1, class类型参数2, ...>

返回值类型  模板名(形参表)

{

    函数体

}

其中的template和class以及尖括号是必须的 ，class 关键字也可以用 typename 关键字替换，例如：

template <typename 类型参数1, typename 类型参数2, ...>

函数模板看上去就像一个函数。例如 Swap 模板的写法如下：

[复制纯文本复制](http://c.biancheng.net/view/315.html)

template <class T>

void Swap(T & x, T & y)

{

T tmp = x;

x = y;

y = tmp;

}

template <class T>

void Swap(T & x, T & y)

{

T tmp = x;

x = y;

y = tmp;

}

T 是类型参数，代表类型。编译器由模板自动生成函数时，会用具体的类型名对模板中所有的类型参数进行替换，其他部分则原封不动地保留。同一个类型参数只能替换为同一种类型。编译器在编译到调用函数模板的语句时，会根据实参的类型判断该如何替换模板中的类型参数。

例如下面的程序：

[复制纯文本复制](http://c.biancheng.net/view/315.html)

#include <iostream>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

template<class T>

void Swap(T & x, T & y)

{

T tmp = x;

x = y;

y = tmp;

}

int main()

{

int n = 1, m = 2;

Swap(n, m); //编译器自动生成 void Swap (int &, int &)函数

double f = 1.2, g = 2.3;

Swap(f, g); //编译器自动生成 void Swap (double &, double &)函数

return 0;

}

#include <iostream>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

template<class T>

void Swap(T & x, T & y)

{

T tmp = x;

x = y;

y = tmp;

}

int main()

{

int n = 1, m = 2;

Swap(n, m); //编译器自动生成 void Swap (int &, int &)函数

double f = 1.2, g = 2.3;

Swap(f, g); //编译器自动生成 void Swap (double &, double &)函数

return 0;

}

编译器在编译到Swap(n, m);时找不到函数 Swap 的定义，但是发现实参 n、m 都是 int 类型的，用 int 类型替换 Swap 模板中的 T 能得到下面的函数：

[复制纯文本复制](http://c.biancheng.net/view/315.html)

void Swap (int & x, int & y)

{

int tmp = x;

x = y;

y = tmp;

}

void Swap (int & x, int & y)

{

int tmp = x;

x = y;

y = tmp;

}

该函数可以匹配Swap(n, m);这条语句。于是编译器就自动用 int 替换 Swap 模板中的 T，生成上面的 Swap 函数，将该 Swap 函数的源代码加入程序中一起编译，并且将Swap(n, m);编译成对自动生成的 Swap 函数的调用。

同理，编译器在编译到Swap(f, g);时会用 double 替换 Swap 模板中的 T，自动生成以下 Swap 函数：

void Swap(double & x, double & y)

{

double tmp = x;

x = y;

y = tmp;

}

void Swap(double & x, double & y)

{

double tmp = x;

x = y;

y = tmp;

}

然后再将Swap(f, g);编译成对该 Swap 函数的调用。

# C++ 类

## 类

类是一种将抽象转换为用户定义类型的C++工具，它将数据表示和操纵数据的方法组合成一个整洁的包。

### 类定义：

类的定义由class标识符、类名、以及类结构中的private和public属性的成员组成，如下定义所示：

#ifndef STOCK\_H\_

#define STOCK\_H\_

#include <iostream>

class Stock { //class 是类标识符，Stock是类型名

private: //私有属性的成员

std::string company;

long shares;

double shar\_val;

double total\_val;

void set\_tot() { total\_val = shares \* shar\_val; }

public: //公有属性的成员

Stock(); //默认构造函数

Stock(const std::string &co, long n = 0, double pr = 0.0); //自定义构造函数

~Stock(); //析构函数

void buy(long num, double price); //类成员函数

void sell(long num, double price);

void update(double price);

void show();

};

#endif // !STOCK\_H\_

### 构造函数

构造是一类特殊的函数，因为一个类中大部分数据是隐藏的，不能老是通过调用公共函数的方法来进行初始化。构造函数没有返回值，构造函数的名字和类的名字完全[一模一样](https://www.baidu.com/s?wd=%E4%B8%80%E6%A8%A1%E4%B8%80%E6%A0%B7&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)（大小写也相同），所以看到一个类声明中没有返回值，那么说明这个函数是构造函数，构造函数必须放在public下面。

作用：赋初值,初始化对象的数据成员,由编译器帮我们调用。

特点：①函数名和类名一样。②没有返回值。③支持有参/无参。④可以重载。

调用时机：在类的对象创建时刻,编译器帮我们调用构造函数。

Stock stock1("nanosmart", 12, 20.0); //隐式的调用构造函数

Stock stock1 = Stock("nanosmart", 12, 20.0); //显式的调用构造函数

Stock \*p1 = new Stock("nanosmart", 12, 20.0); //使用指针和new方法调用构造函数

Stock stock1; //这样会调用默认构造函数，这样会创建stock1对象，并且根据默认构造函数来//对stock1进行初始化，如果该构造函数是这样类型的：Stock :: Stock(){};

//的，将不初始化stock1对象

### 构造函数初始化

构造函数的一项重要功能是对成员变量进行初始化，为了达到这个目的，可以在构造函数的函数体中对成员变量一一赋值，还可以采用初始化列表。

[C++](http://c.biancheng.net/cplus/)构造函数的初始化列表使得代码更加简洁，请看下面的例子：

#include <iostream>

u[sin](http://c.biancheng.net/ref/sin.html)g namespace std;

class Student{

private:

char \*m\_name;

int m\_age;

float m\_score;

public:

Student(char \*name, int age, float score);

void show();

};

//采用初始化列表

Student::Student(char \*name, int age, float score): m\_name(name), m\_age(age), m\_score(score){

//TODO:

}

void Student::show(){

cout<<m\_name<<"的年龄是"<<m\_age<<"，成绩是"<<m\_score<<endl;

}

int main(){

Student stu("小明", 15, 92.5f);

stu.show();

Student \*pstu = new Student("李华", 16, 96);

pstu -> show();

return 0;

}

运行结果：

小明的年龄是15，成绩是92.5

李华的年龄是16，成绩是96

如本例所示，定义构造函数时并没有在函数体中对成员变量一一赋值，其函数体为空（当然也可以有其他语句），而是在函数首部与函数体之间添加了一个冒号:，后面紧跟m\_name(name), m\_age(age), m\_score(score)语句，这个语句的意思相当于函数体内部的m\_name = name; m\_age = age; m\_score = score;语句，也是赋值的意思。

使用构造函数初始化列表并没有效率上的优势，仅仅是书写方便，尤其是成员变量较多时，这种写法非常简单明了。

初始化列表可以用于全部成员变量，也可以只用于部分成员变量。下面的示例只对 m\_name 使用初始化列表，其他成员变量还是一一赋值：

Student::Student(char \*name, int age, float score): m\_name(name){

m\_age = age;

m\_score = score;

}

**注意，成员变量的初始化顺序与初始化列表中列出的变量的顺序无关，它只与成员变量在类中声明的顺序有关。**请看代码：

#include <iostream>

using namespace std;

class Demo{

private:

int m\_a;

int m\_b;

public:

Demo(int b);

void show();

};

Demo::Demo(int b): m\_b(b), m\_a(m\_b){ }

void Demo::show(){ cout<<m\_a<<", "<<m\_b<<endl; }

int main(){

Demo obj(100);

obj.show();

return 0;

}

运行结果：

2130567168, 100

在初始化列表中，我们将 m\_b 放在了 m\_a 的前面，看起来是先给 m\_b 赋值，再给 m\_a 赋值，其实不然！成员变量的赋值顺序由它们在类中的声明顺序决定，在 Demo 类中，我们先声明的 m\_a，再声明的 m\_b，所以构造函数和下面的代码等价：

Demo::Demo(int b): m\_b(b), m\_a(m\_b){

m\_a = m\_b;

m\_b = b;

}

给 m\_a 赋值时，m\_b 还未被初始化，它的值是不确定的，所以输出的 m\_a 的值是一个奇怪的数字；给 m\_a 赋值完成后才给 m\_b 赋值，此时 m\_b 的值才是 100。

obj 在栈上分配内存，成员变量的初始值是不确定的。

### 初始化 const 成员变量

构造函数初始化列表还有一个很重要的作用，那就是初始化 const 成员变量。初始化 const 成员变量的唯一方法就是使用初始化列表。例如 VS/VC 不支持变长数组（数组长度不能是变量），我们自己定义了一个 VLA 类，用于模拟变长数组，请看下面的代码：

class VLA{

private:

const int m\_len;

int \*m\_arr;

public:

VLA(int len);

};

//必须使用初始化列表来初始化 m\_len

VLA::VLA(int len): m\_len(len){

m\_arr = new int[len];

}

VLA 类包含了两个成员变量，m\_len 和 m\_arr [指针](http://c.biancheng.net/c/80/)，需要注意的是 m\_len 加了 const 修饰，只能使用初始化列表的方式赋值，如果写作下面的形式是错误的：

class VLA{

private:

const int m\_len;

int \*m\_arr;

public:

VLA(int len);

};

VLA::VLA(int len){

m\_len = len;

m\_arr = new int[len];

}

### 析构函数

析构函数是在对象过期时，程序自动调用的一个特殊的成员函数，它将完成清理任务。

~Stock(); //析构函数

作用：用于释放资源。

特点：①和类名一样,不过得在前面加上~。②无参数,无返回值。③因为无参数,无返回值,所以不可以重载。④尽量不要自己调用析构函数,但是在某些需要的时候再调用。

调用时机：快退出函数的时候,编译器帮我们调用。

## 对象数组

如果有需要，可以同时创建多个对象，可以利用创建对象的形式创建一组对象数组，数组中每个对象都是该数组的一个组成成员，都是单独的类对象。

Stock mystuff[4]; //创建一组4个组员的对象数组。

# C++动态内存

C++ 程序中的内存分为两个部分：

**栈：**在函数内部声明的所有变量都将占用栈内存。

**堆：**这是程序中未使用的内存，在程序运行时可用于动态分配内存。

New 运算符可以给指定类型的变量在运行时分配堆内的内存，在该变量作用结束后，动态内存就没有用了，可以使用delate运算符删除之前由new分配的动态内存。语法如下：

double\* pvalue = NULL; // 初始化为 null 的指针

pvalue = new double; // 为变量请求内存

delete pvalue; // 释放 pvalue 所指向的内存

## delate数组区别

int \*a = new int[10];

delete a;

delete [] a;

例子：

class A {

private: char \*m\_cBuffer;

int m\_nLen;

public: A(){

m\_cBuffer = new char[m\_nLen];

}

~A() {

delete [] m\_cBuffer;

}

};

A \*a = new A[10];

delete a; // 仅释放了a指针指向的全部内存空间（比如如果a只是指向a[0]，那么只是将a[0]的空间全部释放），但是只调用了a[0]对象的析构函数 剩下的从a[1]到a[9]这9个用户自行分配的m\_cBuffer对应内存空间将不能释放，从而造成内存泄漏。

delete [] a; // 调用使用类对象的析构函数释放用户自己分配内存空间并且释放了a指针指向的全部内存空间。

## New 和 malloc的区别 ，delate和free的区别

new 的功能是在堆区新建一个对象，并返回该对象的指针。所谓的**【新建对象】**的意思就是，将调用该类的构造函数，因为如果不构造的话，就不能称之为一个对象。

malloc 只是机械的分配一块内存，如果用 mallco 在堆区创建一个对象的话，是不会调用构造函数的。严格说来用 malloc 不能算是新建了一个对象，只能说是分配了一块与该类对象匹配的内存而已，然后强行把它解释为【这是一个对象】，按这个逻辑来，也不存在构造函数什么事。

用 delete 去释放一个堆区的对象，会调用该对象的析构函数。

用 free 去释放一个堆区的对象，不会调用该对象的析构函数。

例子：

#include <iostream>

#include <malloc.h>

class TEST {

private:

int num1;

int num2;

public: TEST() {

num1 = 10;

num2 = 20;

}

void Print() {

std::cout << num1 << " " << num2 << std::endl;

}

};

int main(void) {

TEST \* pObj1 = (TEST \*)malloc(sizeof(TEST)); // 用malloc()函数在堆区分配一块内存空间，然后用强制类型转换将该块内存空间解释为是一个TEST类对象，这不会调用TEST的默认构造函数

pObj1->Print();

TEST \* pObj2 = new TEST; // 用new在堆区创建一个TEST类的对象，这会调用TEST类的默认构造函数

pObj2->Print(); return 0;

}

/\* 运行结果：

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

-842150451 -842150451 |

10 20 |

请按任意键继续. . . |

----------------------------- 我们可以看到pObj1所指的对象中，字段num1与num2都是垃圾值 而pObj2所指的对象中，字段num1与num2显然是经过了构造后的值 \*/

## Malloc calloc realloc new

1) malloc 函数： void \*malloc(unsigned int size)

     在内存的动态分配区域中分配一个长度为size的连续空间，如果分配成功，则返回所分配内存空间的首地址，否则返回NULL，申请的内存不会进行初始化。

2）calloc 函数： void \*calloc(unsigned int num, unsigned int size)

     按照所给的数据个数和数据类型所占字节数，分配一个 num \* size 连续的空间。

    calloc申请内存空间后，会自动初始化内存空间为 0，但是malloc不会进行初始化，其内存空间存储的是一些随机数据。

3）realloc 函数： void \*realloc(void \*ptr, unsigned int size)

    动态分配一个长度为size的内存空间，并把内存空间的首地址赋值给ptr，把ptr内存空间调整为size。

    申请的内存空间不会进行初始化。

4）new是动态分配内存的运算符，自动计算需要分配的空间，在分配类类型的内存空间时，同时调用类的构造函数，对内存空间进行初始化，即完成类的初始化工作。动态分配内置类型是否自动初始化取决于变量定义的位置，在函数体外定义的变量都初始化为0，在函数体内定义的内置类型变量都不进行初始化。

# C++继承

### 封装

所谓封装就是将某些东西包装盒隐藏起来，让外界无法直接使用，只能通过某些特定的方式才能访问。封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是通过外部接口以及特定的访问权限来使用类的成员。

例如，下例中时一个动物类，我们把数据和功能封装起来。

class Animal

{

public: //这就是公共成员,外部的接口

void SetAnimalName(string strname);

void ShowAnimalName();

private: //这是私有成员，外部是无法访问到的

string m\_strName;

};

我们可以通过封装使一部分成员充当类与外部的接口，而将其它的成员隐藏起来，这样就限制了外部对成员的访问，也使不同类之间的相互影响度降低。

优点： 1）隔离变化；2）便于使用； 3）提高重用性； 4）提高安全性

缺点： 1）如果封装太多，影响效率； 2）使用者不能知道代码具体实现。

## 概念

面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行时间的效果。

当创建一个类时，不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为**基类**，新建的类称为**派生类**。

继承代表了this **is a** that关系。例如，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

命名方式：class derived-class: access-specifier base-class

其中，访问修饰符 access-specifier 是 **public、protected** 或 **private** 其中的一个，base-class 是之前定义过的某个类的名称。如果未使用访问修饰符 access-specifier，则默认为 private。

例子：

#include <iostream>

using namespace std;

// 基类 class Shape {

public: void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected: int width;

int height;

};

// 派生类 class Rectangle: public Shape {

public: int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// 输出对象的面积 cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

改程序运行结果为：Total area: 35

## 访问控制和继承

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private。

我们可以根据访问权限总结出不同的访问类型，如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **访问** | **public** | **protected** | **private** |
| 同一个类 | yes | yes | yes |
| 派生类 | yes | yes | no |
| 外部的类 | yes | no | no |

一个派生类继承了所有的基类方法，但下列情况除外：

1：基类的构造函数、析构函数和拷贝构造函数。

2：基类的重载运算符。

3：基类的友元函数。

## 继承类型

当一个类派生自基类，该基类可以被继承为 **public、protected** 或 **private** 几种类型。继承类型是通过上面讲解的访问修饰符 access-specifier 来指定的。

我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。当使用不同类型的继承时，遵循以下几个规则：

**1、公有继承（public）：**当一个类派生自**公有**基类时，基类的**公有**成员也是派生类的**公有**成员，基类的**保护**成员也是派生类的**保护**成员，基类的**私有**成员不能直接被派生类访问，但是可以通过调用基类的**公有**和**保护**成员来访问。

**2、保护继承（protected）：** 当一个类派生自**保护**基类时，基类的**公有**和**保护**成员将成为派生类的**保护**成员。

**3、私有继承（private）：**当一个类派生自**私有**基类时，基类的**公有**和**保护**成员将成为派生类的**私有**成员。

## 多继承

多继承即一个子类可以有多个父类，它继承了多个父类的特性。

C++ 类可以从多个类继承成员，语法如下：

class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…

{

<派生类类体>

};

其中，访问修饰符继承方式是 **public、protected** 或 **private** 其中的一个，用来修饰每个基类，各个基类之间用逗号分隔，如上所示。

实例：

#include <iostream>

using namespace std;

// 基类 Shape

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// 基类 PaintCost

class PaintCost

{

public:

int getCost(int area)

{

return area \* 70;

}

};

// 派生类

class Rectangle : public Shape, public PaintCost

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// 输出对象的面积

cout << "Total area: " << Rect.getArea() << endl;

// 输出总花费

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Total area: 35

Total paint cost: $2450

## 环状继承

,A->D, B->D, C->(A，B)，例如：

class D{......};

class B: public D{......};

class A: public D{......};

class C: public B, public A{.....};

这个继承会使D创建两个对象,要解决上面问题就要用**虚拟继承格式**

格式：**class 类名: virtual 继承方式 父类名**

class D{......};

class B: virtual public D{......};

class A: virtual public D{......};

class C: public B, public A{.....};

虚继承--（在创建对象的时候会创建一个虚表）在创建父类对象的时候

A:virtual public D

B:virtual public D

**实例：**

#include <iostream>

using namespace std;

//基类

class D

{

public:

D() { cout << "D()" << endl; }

~D() { cout << "~D()" << endl; }

protected:

int d;

};

class B :virtual public D

{

public:

B() { cout << "B()" << endl; }

~B() { cout << "~B()" << endl; }

protected:

int b;

};

class A :virtual public D

{

public:

A() { cout << "A()" << endl; }

~A() { cout << "~A()" << endl; }

protected:

int a;

};

class C :public B, public A

{

public:

C() { cout << "C()" << endl; }

~C() { cout << "~C()" << endl; }

protected:

int c;

};

int main()

{

cout << "Hello World!" << endl;

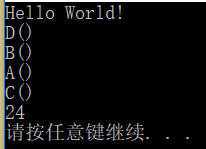
C c; //D, B, A ,C

cout << sizeof(c) << endl;

return 0;

}

Win10 64位 运行结果



下边三个图是在Win10 64位系统（1）使用虚拟继承，类C继承A和B 、（2）使用虚拟继承，去掉类A、（3）不使用虚拟继承，去掉类A、（4）不使用虚拟继承，类C继承A和B 三种情况下的VS运行结果。

**通过结果可以看出，虽然2和3输出一样，但是占用内存却不一样，这是由于虚拟继承和普通继承对内存使用情况不同造成的。**使用虚拟继承时，继承类和被继承类使用各自的虚函数表指针，普通继承时，两个类使用的是同一个虚函数表指针（被继承类的虚函数表指针）。具体分析参见以下链接：

https://blog.csdn.net/u011841639/article/details/39156347

https://www.oschina.net/translate/cpp-virtual-inheritance?print

|  |  |  |  |
| --- | --- | --- | --- |
| 使用虚拟继承，类C继承A和B | 使用虚拟继承，去掉类A | 不使用虚拟继承，去掉类A | 不使用虚拟继承，类C继承A和B |
|  |  |  |  |

## C++多态

**多态**按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

例子：

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0)

{

width = a;

height = b;

}

int area()

{

cout << "Parent class area :" << endl;

return 0;

}

};

class Rectangle : public Shape {

public:

Rectangle(int a = 0, int b = 0) :Shape(a, b) { }

int area()

{

cout << "Rectangle class area :" << endl;

return (width \* height);

}

};

class Triangle : public Shape {

public:

Triangle(int a = 0, int b = 0) :Shape(a, b) { }

int area()

{

cout << "Triangle class area :" << endl;

return (width \* height / 2);

}

};

// 程序的主函数

int main()

{

Shape \*shape;

Rectangle rec(10, 7);

Triangle tri(10, 5);

// 存储矩形的地址

shape = &rec;

// 调用矩形的求面积函数 area

shape->area();

// 存储三角形的地址

shape = &tri;

// 调用三角形的求面积函数 area

shape->area();

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Parent class area

Parent class area

导致错误输出的原因是，调用函数 area() 被编译器设置为基类中的版本，这就是所谓的**静态多态**，或**静态链接** - 函数调用在程序执行前就准备好了。有时候这也被称为**早绑定**，因为 area() 函数在程序编译期间就已经设置好了。

但现在，让我们对程序稍作修改，在 Shape 类中，area() 的声明前放置关键字 **virtual**，如下所示：

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0)

{

width = a;

height = b;

}

virtual int area()

{

cout << "Parent class area :" << endl;

return 0;

}

};

修改后，当编译和执行前面的实例代码时，它会产生以下结果：

Rectangle class area

Triangle class area

此时，编译器看的是指针的内容，而不是它的类型。因此，由于 tri 和 rec 类的对象的地址存储在 \*shape 中，所以会调用各自的 area() 函数。

形成多态必须具备三个条件：

1、必须存在继承关系；

2、继承关系必须有同名虚函数（**其中虚函数是在基类中使用关键字Virtual声明的函数，在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数**）；

3、存在基类类型的指针或者引用，通过该指针或引用调用虚函数；

正如上述例子，每个子类都有一个函数 area() 的独立实现。这就是**多态**的一般使用方式。有了多态，您可以有多个不同的类，都带有同一个名称但具有不同实现的函数，函数的参数甚至可以是相同的。

## 虚函数

**虚函数** 是在基类中使用关键字 **virtual** 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。

我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为**动态链接**，或**后期绑定**。正如C++多态中的例子所示。

## 纯虚函数

纯虚函数声明如下： **virtual void funtion1()=0;** 纯虚函数一定没有定义，纯虚函数用来规范派生类的行为，即接口。包含纯虚函数的类是抽象类，抽象类不能定义实例，但可以声明指向实现该抽象类的具体类的指针或引用。

例子：

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0)

{

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

“= 0”告诉编译器，函数没有主体，上面的虚函数是**纯虚函数**。

纯虚函数是为了我们要在基类中定义虚函数，以便在派生类中重新定义该函数更好地适用于对象，但是在基类中又不能对虚函数给出有意义的实现，这个时候就会用到纯虚函数。

## 问题：构造和析构函数可以是虚函数吗？

1. 为什么构造函数不能为虚函数？

    虚函数的调用需要虚函数表指针，而该指针存放在对象的内容空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数——构造函数了。

2. 为什么析构函数可以为虚函数，如果不设为虚函数可能会存在什么问题？

    首先析构函数可以为虚函数，而且当要使用基类指针或引用调用子类时，最好将基类的析构函数声明为虚函数，否则可以存在内存泄露的问题。

    举例说明：

    子类B继承自基类A；A \*p = new B; delete p;

    1） 此时，如果类A的析构函数不是虚函数，那么delete p；将会仅仅调用A的析构函数，只释放了B对象中的A部分，而派生出的新的部分未释放掉。

    2） 如果类A的析构函数是虚函数，delete p; 将会先调用B的析构函数，再调用A的析构函数，释放B对象的所有空间。

    补充： B \*p = new B; delete p;时也是先调用B的析构函数，再调用A的析构函数。

C++类有继承时，析构函数必须为虚函数。如果不是虚函数，则使用时可能存在内在泄漏的问题。

假设我们有这样一种继承关系：

SubClass 继承 BaseClass

如果我们以这种方式创建对象：

SubClass\* pObj = new SubClass();

delete pObj;12

不管析构函数是否是虚函数(即是否加virtual关键词)，delete时基类和子类都会被释放；

如果我们以这种方式创建对象：

BaseClass\* pObj = new SubClass();

delete pObj;12

若析构函数是虚函数(即加上virtual关键词)，delete时基类和子类都会被释放；

若析构函数不是虚函数(即不加virtual关键词)，delete时只释放基类，不释放子类；

# 友元

如果一个类A有自己的私有成员函数t()或私有成员变量t，而另外的类B或者函数B想访问t()或t，就需要在类A中声明：B为A的友元函数或者友元类（意思就是A说：“B是我的朋友，他可以访问我的私有成员”）。

## 友元函数

在定义一个类的时候，可以把一些函数（包括全局函数和其他类的成员函数）声明为“友元”，这样那些函数就成为该类的友元函数，在友元函数内部就可以访问该类对象的私有成员了。

将全局函数声明为友元的写法如下：

friend  返回值类型  函数名(参数表);

将其他类的成员函数声明为友元的写法如下：

friend  返回值类型  其他类的类名::成员函数名(参数表);

但是，不能把其他类的私有成员函数声明为友元。

例子：

#include<iostream>

using namespace std;

class CCar; //提前声明CCar类，以便后面的CDriver类使用

class CDriver

{

public:

void ModifyCar(CCar\* pCar); //改装汽车

};

class CCar

{

private:

int price;

friend int MostExpensiveCar(CCar cars[], int total); //声明友元

friend void CDriver::ModifyCar(CCar\* pCar); //声明友元

};

void CDriver::ModifyCar(CCar\* pCar)

{

pCar->price += 1000; //汽车改装后价值增加

}

int MostExpensiveCar(CCar cars[], int total) //求最贵气车的价格

{

int tmpMax = -1;

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

if (cars[i].price > tmpMax)

tmpMax = cars[i].price;

return tmpMax;

}

int main()

{

return 0;

}

第 3 行声明了 CCar 类，CCar 类的定义在后面。之所以要提前声明，是因为 CDriver 类的定义中用到了 CCar 类型（第7行），而此时 CCar 类还没有定义，编译会报错。

不要第 3 行，而把 CCar 类的定义写在 CDriver 类的前面，是解决不了这个问题的，因为 CCar 类中也用到了 CDriver 类型（第14行），把 CCar 类的定义写在前面会导致第 14 行的 CDriver 因没有定义而报错。C++ 为此提供的解决办法是：可以简单地将一个类的名字提前声明，写法如下：

class  类名;

尽管可以提前声明，但是在一个类的定义出现之前，仍然不能有任何会导致该类对象被生成的语句。但使用该类的[指针](http://c.biancheng.net/c/80/)或引用是没有问题的。

第 13 行将全局函数 MostExpensiveCar 声明为 CCar 类的友元，因此在第 24 行可以访问 cars[i] 的私有成员 price。同理，第 14 行将 CDriver 类的 ModifyCar 成员函数声明为友元，因此在第 18 行可以访问 pCar 指针所指向的对象的私有成员变量 price。

## 友元类

一个类 A 可以将另一个类 B 声明为自己的友元，类 B 的所有成员函数就都可以访问类 A 对象的私有成员。在类定义中声明友元类的写法如下：

friend  class  类名;

例子：

class CCar

{

private:

int price;

friend class CDriver; //声明 CDriver 为友元类

};

class CDriver

{

public:

CCar myCar;

void ModifyCar() //改装汽车

{

myCar.price += 1000; //因CDriver是CCar的友元类，故此处可以访问其私有成员

}

};

int main()

{

return 0;

}

第 5 行将 CDriver 声明为 CCar 的友元类。这条语句本来就是在声明 CDriver 是一个类，所以 CCar 类定义前面就不用声明 CDriver 类了。第 5 行使得 CDriver 类的所有成员函数都能访问 CCar 对象的私有成员。如果没有第 5 行，第 13 行对 myCar 私有成员 price 的访问就会导致编译错误。

一般来说，类 A 将类 B 声明为友元类，则类 B 最好从逻辑上和类 A 有比较接近的关系。例如上面的例子，CDriver 代表司机，CCar 代表车，司机拥有车，所以 CDriver 类和 CCar 类从逻辑上来讲关系比较密切，把 CDriver 类声明为 CCar 类的友元比较合理。

友元关系在类之间不能传递，即类 A 是类 B 的友元，类 B 是类 C 的友元，并不能导出类 A 是类 C 的友元。“咱俩是朋友，所以你的朋友就是我的朋友”这句话在 C++ 的友元关系上不成立。

# C++接口（抽象类）

接口描述了类的行为和功能，而不需要完成类的特定实现。

C++ 接口是使用**抽象类**来实现的，抽象类与数据抽象互不混淆，数据抽象是一个把实现细节与相关的数据分离开的概念。

如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。

设计**抽象类**（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，它只能作为**接口**使用。如果试图实例化一个抽象类的对象，会导致编译错误。

因此，如果一个 ABC 的子类需要被实例化，则必须实现每个虚函数，这也意味着 C++ 支持使用 ABC 声明接口。如果没有在派生类中重写纯虚函数，就尝试实例化该类的对象，会导致编译错误。

可用于实例化对象的类被称为**具体类**。

例子：

#include <iostream>

using namespace std;

// 基类

class Shape

{

public:

// 提供接口框架的纯虚函数

virtual int getArea() = 0;

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// 派生类

class Rectangle : public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

class Triangle : public Shape

{

public:

int getArea()

{

return (width \* height) / 2;

}

};

int main(void)

{

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

// 输出对象的面积

cout << "Total Rectangle area: " << Rect.getArea() << endl;

Tri.setWidth(5);

Tri.setHeight(7);

// 输出对象的面积

cout << "Total Triangle area: " << Tri.getArea() << endl;

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Total Rectangle area: 35

Total Triangle area: 17

在该例子中，class Shape定义了一个接口getArea()，两个派生类通过实例化对象来调用实现该接口函数。

# C++异常处理

异常是程序在执行期间产生的问题。C++ 异常是指在程序运行时发生的特殊情况，比如尝试除以零的操作。

异常提供了一种转移程序控制权的方式。C++ 异常处理涉及到三个关键字：**try、catch、throw**。

**throw:** 当问题出现时，程序会抛出一个异常。这是通过使用 **throw** 关键字来完成的。

**catch:** 在您想要处理问题的地方，通过异常处理程序捕获异常。**catch** 关键字用于捕获异常。

**try:** **try** 块中的代码标识将被激活的特定异常。它后面通常跟着一个或多个 catch 块。

如果有一个块抛出一个异常，捕获异常的方法会使用 **try** 和 **catch** 关键字。try 块中放置可能抛出异常的代码，try 块中的代码被称为保护代码。使用 try/catch 语句的语法如下所示：

try {

// 保护代码

}

catch( ExceptionName e1 ) {

// catch 块

}

catch( ExceptionName e2 ) {

// catch 块

}

catch( ExceptionName eN ) {

// catch 块

}

如果 **try** 块在不同的情境下会抛出不同的异常，这个时候可以尝试罗列多个 **catch** 语句，用于捕获不同类型的异常。

# 指针和引用的区别

参考：<https://blog.csdn.net/superwangxinrui/article/details/80565594>

# [C++ 11 ：override 关键字的使用](https://www.cnblogs.com/schips/p/12309999.html)

https://www.cnblogs.com/schips/p/12309999.html

## override 关键字

作用：在成员函数声明或定义中， override 确保该函数为虚函数并覆写来自基类的虚函数。  
位置：函数调用运算符之后，函数体或纯虚函数标识 “= 0” 之前。

使用以后有以下好处:  
1．可以当注释用,方便阅读．  
2．告诉阅读你代码的人，这是方法的复写．  
3．编译器可以给你验证 override 对应的方法名是否是你父类中所有的，如果没有则报错．

## override 使用举例

如果你想重写父类的方法，比如toString()方法：

正确的是：

public String toString() override {

...

}

假如不小心把方法名写错了而没写 override ,这时编译器是可以编译通过的，因为编译器以为这个方法是你的子类中自己增加的方法。如：

// 注意这里的小写方法，实际上是错误的。

public String tostring() {...}

相反，如果你很机智，在知道自己要重写父类的方法，加上了 override 标签后，编译器会检查出重写方法错误，会保证你重写父类方法的正确性。  
　　总结：在重写方法时，最好加上这个 override 这个关键字 以 加强代码规范。

https://blog.csdn.net/luoshao\_/article/details/82288146

1. override关键字作用：  
   如果派生类在虚函数声明时使用了override描述符，那么该函数必须重载其基类中的同名函数，否则代码将无法通过编译。
2. 举例子说明
4. struct Base
5. {
6. virtual void Turing() = 0;
7. virtual void Dijkstra() = 0;
8. virtual void VNeumann(int g) = 0;
9. virtual void DKnuth() const;
10. void Print();
11. };
12. struct DerivedMid: public Base
13. {
14. *// void VNeumann(double g);*
15. *//接口被隔离了，曾想多一个版本的VNeumann函数*
16. };
17. struct DerivedTop : public DerivedMid
18. {
19. void Turing() override;
20. void Dikjstra() override; *//无法通过编译，拼写错误，并非重载*
21. void VNeumann(double g) override; *//无法通过编译，参数不一致，并非重载*
22. void DKnuth() override; *//无法通过编译，常量性不一致，并非重载*
23. void Print() override; *//无法通过编译，非虚函数重载*
24. };



上面的1-5个重载函数编译过程中，除了返回值不同的infor会报错以外，其他函数都不会有问题，但是在类实例化的时候会提示是抽象类，因为他们都没有真正实现重载C

1. class C: public testoverride
2. {
3. public:
4. virtual void show() override;
5. virtual void infor() override;
6. virtual void vmendd() override;
7. virtual void test(int x) override;
8. virtual void splle() override;
9. };

添加了override以后，会在编译器override修饰符则可以保证编译器辅助地做一些检查，上面的情况无法通过编译

1. 结论  
   如果派生类里面是像重载虚函数 就加上关键字override 这样编译器可以辅助检查是不是正确重载，如果没加这个关键字 也没什么严重的error 只是少了编译器检查的安全性

# Inline 关键字

https://blog.csdn.net/kaige2111/article/details/51280355

### 1. 引入inline关键字的原因

在c/c++中，为了解决一些频繁调用的小函数大量消耗栈空间（栈内存）的问题，特别的引入了inline修饰符，表示为内联函数。

栈空间就是指放置程序的局部数据（也就是函数内数据）的内存空间。

在系统下，栈空间是有限的，假如频繁大量的使用就会造成因栈空间不足而导致程序出错的问题，如，函数的死循环递归调用的最终结果就是导致栈内存空间枯竭。

下面我们来看一个例子：

#include <stdio.h>

//函数定义为inline即:内联函数

inline char\* dbtest(int a) {

return (i % 2 > 0) ? "奇" : "偶";

}

int main()

{

int i = 0;

for (i=1; i < 100; i++) {

printf("i:%d 奇偶性:%s /n", i, dbtest(i));

}

}12345678910111213

上面的例子就是标准的内联函数的用法，使用inline修饰带来的好处我们表面看不出来，其实，在内部的工作就是在每个for循环的内部任何调用dbtest(i)的地方都换成了(i%2>0)?”奇”:”偶”，这样就避免了频繁调用函数对栈内存重复开辟所带来的消耗。

### 2. inline使用限制

inline的使用是有所限制的，inline只适合涵数体内代码简单的涵数使用，不能包含复杂的结构控制语句例如while、switch，并且不能内联函数本身不能是直接递归函数（即，自己内部还调用自己的函数）。

### 3. inline仅是一个对编译器的建议

inline函数仅仅是一个对编译器的建议，所以最后能否真正内联，看编译器的意思，它如果认为函数不复杂，能在调用点展开，就会真正内联，并不是说声明了内联就会内联，声明内联只是一个建议而已。

### 4. 建议：inline函数的定义放在头文件中

其次，因为内联函数要在调用点展开，所以编译器必须随处可见内联函数的定义，要不然就成了非内联函数的调用了。所以，这要求每个调用了内联函数的文件都出现了该内联函数的定义。

因此，将内联函数的定义放在头文件里实现是合适的，省却你为每个文件实现一次的麻烦。

声明跟定义要一致：如果在每个文件里都实现一次该内联函数的话，那么，最好保证每个定义都是一样的，否则，将会引起未定义的行为。如果不是每个文件里的定义都一样，那么，编译器展开的是哪一个，那要看具体的编译器而定。所以，最好将内联函数定义放在头文件中。

### 5. 类中的成员函数与inline

定义在类中的成员函数缺省都是内联的，如果在类定义时就在类内给出函数定义，那当然最好。如果在类中未给出成员函数定义，而又想内联该函数的话，那在类外要加上inline，否则就认为不是内联的。

例如，

class A

{

public:void Foo(int x, int y) { } // 自动地成为内联函数

}1234

将成员函数的定义体放在类声明之中虽然能带来书写上的方便，但不是一种良好的编程风格，上例应该改成：

// 头文件

class A

{

public:

void Foo(int x, int y);

}123456

// 定义文件

inline void A::Foo(int x, int y){} 12

### 6. inline 是一种“用于实现的关键字”

关键字inline 必须与函数定义体放在一起才能使函数成为内联，仅将inline 放在函数声明前面不起任何作用。

如下风格的函数Foo 不能成为内联函数：

inline void Foo(int x, int y); // inline 仅与函数声明放在一起

void Foo(int x, int y){}123

而如下风格的函数Foo 则成为内联函数：

void Foo(int x, int y);

inline void Foo(int x, int y) {} // inline 与函数定义体放在一起123

所以说，inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了inline 关键字，但我认为inline不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。

### 7. 慎用inline

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。

如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

以下情况不宜使用内联：

（1）如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。

（2）如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中）。

8.总结

内联函数并不是一个增强性能的灵丹妙药。只有当函数非常短小的时候它才能得到我们想要的效果；但是，如果函数并不是很短而且在很多地方都被调用的话，那么将会使得可执行体的体积增大。 最令人烦恼的还是当编译器拒绝内联的时候。在老的实现中，结果很不尽人意，虽然在新的实现中有很大的改善，但是仍然还是不那么完善的。一些编译器能够足够的聪明来指出哪些函数可以内联哪些不能，但是大多数编译器就不那么聪明了，因此这就需要我们的经验来判断。如果内联函数不能增强性能，就避免使用它！

————————————————

版权声明：本文为CSDN博主「进步小二郎」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/kaige2111/java/article/details/51280355>

面试题：

链接：<https://www.nowcoder.com/questionTerminal/0cd6af2fd4374df597b49e09302b1a5a>  
来源：牛客网  
  
关于c++的inline关键字,以下说法正确的是()

A 使用inline关键字的函数会被编译器在调用处展开

B 头文件中可以包含inline函数的声明

C 可以在同一个项目的不同源文件内定义函数名相同但实现不同的inline函数

D 定义在Class声明内的成员函数默认是inline函数

E 优先使用Class声明内定义的inline函数

F 优先使用Class实现的内inline函数的实现

答案说是D,D是明显正确的，但是B的说法也不是不可以吧。A明显错了，C上文说了也明显错了。EF没有优先关系。

# Inflict关键字

https://blog.csdn.net/Alisa\_xf/article/details/89674370

转自：<https://blog.csdn.net/tianmingdyx/article/details/79823470>

### 1 莫名其妙的explicit

     在很多C++代码中，定义类的构造函数的时候，往往会在前面加一个explicit关键字。先来看cppreference上面对它的解释：

The explicit specifier specifies that a constructor or conversion function (since C++11) doesn't allow implicit conversions or copy-initialization. It may only appear within the decl-specifier-seq of the declaration of such a function within its class definition.

    大体意思是说，用explicit修饰构造函数的作用是禁止隐式转换或复制初始化。那什么是隐式转换和复制初始化呢？这要从类的构造函数和初始化过程说起。

### 2 构造函数和初始化过程

     在C++中，变量的初始化有两种方式：直接初始化（用（）运算符，如int a(1);）和复制初始化（用=运算符，如int a = 1;）。在编译和代码运行过程中，这两种初始化是有区别的。先来看一个例子：

 // 实例2

#include<iostream>

using namespace std;

class A

{

public:

A(){ cout << "f1" << endl;} // 无参数构造函数，记为f1

A(int temp){ cout << "f2" << endl; } // 单参数构造函数，记为f2

A(const A& temp){ cout << "f3" << endl;} // 复制构造函数，记为f3

private:

int a;

};

int main()

{

A a1; // 调用f1

A a2(1); // 调用f2

A a3(a2); // 调用f3

cout << "------" << endl;

A a4 = 1; // 隐式转换

cout << "------" << endl;

A a5 = a2; // 拷贝初始化

return 0;

}

    实例1代码可以编译通过并运行，运行结果如下：（gcc6.3）

 f1

f2

f3

------

f2

------

f3

    从运行结果可以看出，在直接初始化（即main函数的第二三句代码）过程中，会根据（）中的数据的类型去调用不同构造函数，其中就包括复制构造函数。A a4 = 1和A a5 = a2这两句代码是复制初始化。但是其中的A a4 = 1这句代码令人困惑，为什么一个数字可以初始化一个类对象呢？其实这个地方发生了隐式转换。

### 3 隐式转换

     看一下cppreference中对隐式转换的解释：

 Implicit conversions are performed whenever an expression of some type T1 is used in context that does not accept that type, but accepts some other type T2; in particular:

\* when the expression is used as the argument when calling a function that is declared with T2 as parameter;

\* when the expression is used as an operand with an operator that expects T2;

\* when initializing a new object of type T2, including return statement in a function returning T2;

\* when the expression is used in a switch statement (T2 is integral type);

\* when the expression is used in an if statement or a loop (T2 is bool).

     其中高亮的那一条，指的就是我们在实例1中遇到的A a4 = 1情况。通过代码运行的结果我们可以看出，上面这两句实际上被编译器隐式的转换成了下面的过程。（具体的过程可能跟编译器有关，有资料说A a4 = 1会先调用f2，构造出一个临时变量temp，再调用f3将temp复制给a4，带这里实际测试结果是只调用的f2）

 A a4(1);

A a5(a2);

    虽然上面的代码是可以正常运行的，但是像A a4 = 1这样的代码的可读性非常的差，因此，为了避免出现这种情况，我们可以使用explicit关键字来修饰构造函数，从而禁止隐式转换和复制初始化。

### 4 使用explicit关键字

 将实例1的代码做如下的修改：

 // 实例2

#include<iostream>

using namespace std;

class A

{

public:

A(){ cout << "f1" << endl;} // 无参数构造函数，记为f1

explicit A(int temp){ cout << "f2" << endl; } // 单参数构造函数，记为f2

explicit A(const A& temp){ cout << "f3" << endl;} // 复制构造函数，记为f3

private:

int a;

};

int main()

{

A a1; // 调用f1

A a2(1); // 调用f2

A a3(a2); // 调用f3

cout << "------" << endl;

A a4 = 1; // 隐式转换

cout << "------" << endl;

A a5 = a2; // 拷贝初始化

return 0;

}

    在单参数构造函数和拷贝构造函数之前加上explicit关键字，然后再编译代码。这时，编译不能通过，并会报出如下的错误：

 prog.cpp: In function ‘int main()’:

prog.cpp:22:12: error: conversion from ‘int’ to non-scalar type ‘A’ requested

A a4 = 1; // 隐式转换

^

prog.cpp:24:12: error: no matching function for call to ‘A::A(A&)’

A a5 = a2; // 拷贝初始化

 这就是因为explicit关键字禁止了隐式转换和复制初始化，因此编译不能通过。注释掉A a4 = 1和A a5 = a2这两句代码就可以编译通过了。 通过使用explicit关键字，可以强制性的要求类的使用者按照规范的方式编写代码，提高代码的可读性，并降低发生错误的风险。

### 5 注意

 复制构造函数又被叫做拷贝构造函数（copy constructor），复制初始化也叫做拷贝初始化。

拷贝构造函数和拷贝初始化不要混淆。拷贝初始化过程要调用拷贝构造函数，但拷贝构造函数也可以用在直接初始化过程。

explicit关键字只对单形参的构造函数起作用，对于多形参的构造函数，不会出现隐式转换的问题。 《C++ Primer》中有指出“可以用单个实参来调用的构造函数定义了从形参类型到该类型的一个隐式转换。”

隐式转换不仅会出现在初始化过程中，另一个很常见的情况是以类类型为形参的函数的参数传递过程，比如下面的代码，如果加上explicit关键字，是不能编译通过的。

// 实例3

#include<iostream>

using namespace std;

class A

{

public:

A(){ cout << "f1" << endl;} // 无参数构造函数，记为f1

A(int temp){ cout << "f2" << endl; } // 单参数构造函数，记为f2

A(const A& temp){ cout << "f3" << endl;} // 复制构造函数，记为f3

int x;

};

A fun(A a){a.x = 1; return a;}

int main()

{

A ret;

cout << "----------" << endl;

ret = fun(1);

return 0;

}

运行结果：

 f1

----------

f2

f3

# C++拷贝构造函数详解

原文链接：https://blog.csdn.net/lwbeyond/article/details/6202256

##### 一. 什么是拷贝构造函数

首先对于普通类型的对象来说，它们之间的复制是很简单的，例如：

int a = 100;

int b = a; 而类对象与普通对象不同，类对象内部结构一般较为复杂，存在各种成员变量。

下面看一个类对象拷贝的简单例子。

#include <iostream>

using namespace std;

class CExample {

private:

　int a;

public:

//构造函数

　CExample(int b)

　{ a = b;}

//一般函数

　void Show ()

　{

cout<<a<<endl;

}

};

int main()

{

　CExample A(100);

　CExample B = A; //注意这里的对象初始化要调用拷贝构造函数，而非赋值

　 B.Show ();

　return 0;

}

运行程序，屏幕输出100。从以上代码的运行结果可以看出，系统为对象 B 分配了内存并完成了与对象 A 的复制过程。就类对象而言，相同类型的类对象是通过拷贝构造函数来完成整个复制过程的。

下面举例说明拷贝构造函数的工作过程。

#include <iostream>

using namespace std;

class CExample {

private:

int a;

public:

//构造函数

CExample(int b)

{ a = b;}

//拷贝构造函数

CExample(const CExample& C)

{

a = C.a;

}

//一般函数

void Show ()

{

cout<<a<<endl;

}

};

int main()

{

CExample A(100);

CExample B = A; // CExample B(A); 也是一样的

B.Show ();

return 0;

} CExample(const CExample& C)　就是我们自定义的拷贝构造函数。可见，拷贝构造函数是一种特殊的构造函数，函数的名称必须和类名称一致，它必须的一个参数是本类型的一个引用变量。

##### 二. 拷贝构造函数的调用时机

在C++中，下面三种对象需要调用拷贝构造函数！1. 对象以值传递的方式传入函数参数

class CExample

{

private:

int a;

public:

//构造函数

CExample(int b)

{

a = b;

cout<<"creat: "<<a<<endl;

}

//拷贝构造

CExample(const CExample& C)

{

a = C.a;

cout<<"copy"<<endl;

}

//析构函数

~CExample()

{

cout<< "delete: "<<a<<endl;

}

void Show ()

{

cout<<a<<endl;

}

};

//全局函数，传入的是对象

void g\_Fun(CExample C)

{

cout<<"test"<<endl;

}

int main()

{

CExample test(1);

//传入对象

g\_Fun(test);

return 0;

}调用g\_Fun()时，会产生以下几个重要步骤：(1).test对象传入形参时，会先会产生一个临时变量，就叫 C 吧。(2).然后调用拷贝构造函数把test的值给C。 整个这两个步骤有点像：CExample C(test);(3).等g\_Fun()执行完后, 析构掉 C 对象。

2. 对象以值传递的方式从函数返回

class CExample

{

private:

int a;

public:

//构造函数

CExample(int b)

{

a = b;

}

//拷贝构造

CExample(const CExample& C)

{

a = C.a;

cout<<"copy"<<endl;

}

void Show ()

{

cout<<a<<endl;

}

};

//全局函数

CExample g\_Fun()

{

CExample temp(0);

return temp;

}

int main()

{

g\_Fun();

return 0;

}当g\_Fun()函数执行到return时，会产生以下几个重要步骤：(1). 先会产生一个临时变量，就叫XXXX吧。(2). 然后调用拷贝构造函数把temp的值给XXXX。整个这两个步骤有点像：CExample XXXX(temp);(3). 在函数执行到最后先析构temp局部变量。(4). 等g\_Fun()执行完后再析构掉XXXX对象。

3. 对象需要通过另外一个对象进行初始化；CExample A(100);

CExample B = A;

// CExample B(A);

后两句都会调用拷贝构造函数。

##### 三. 浅拷贝和深拷贝

###### 1. 默认拷贝构造函数

很多时候在我们都不知道拷贝构造函数的情况下，传递对象给函数参数或者函数返回对象都能很好的进行，这是因为编译器会给我们自动产生一个拷贝构造函数，这就是“默认拷贝构造函数”，这个构造函数很简单，仅仅使用“老对象”的数据成员的值对“新对象”的数据成员一一进行赋值，它一般具有以下形式：

Rect::Rect(const Rect& r)

{

width = r.width;

height = r.height;

}

当然，以上代码不用我们编写，编译器会为我们自动生成。但是如果认为这样就可以解决对象的复制问题，那就错了，让我们来考虑以下一段代码：class Rect

{

public:

Rect() // 构造函数，计数器加1

{

count++;

}

~Rect() // 析构函数，计数器减1

{

count--;

}

static int getCount() // 返回计数器的值

{

return count;

}

private:

int width;

int height;

static int count; // 一静态成员做为计数器

};

int Rect::count = 0; // 初始化计数器

int main()

{

Rect rect1;

cout<<"The count of Rect: "<<Rect::getCount()<<endl;

Rect rect2(rect1); // 使用rect1复制rect2，此时应该有两个对象

cout<<"The count of Rect: "<<Rect::getCount()<<endl;

return 0;

}

　　这段代码对前面的类，加入了一个静态成员，目的是进行计数。在主函数中，首先创建对象rect1，输出此时的对象个数，然后使用rect1复制出对象rect2，再输出此时的对象个数，按照理解，此时应该有两个对象存在，但实际程序运行时，输出的都是1，反应出只有1个对象。此外，在销毁对象时，由于会调用销毁两个对象，类的析构函数会调用两次，此时的计数器将变为负数。

说白了，就是拷贝构造函数没有处理静态数据成员。

出现这些问题最根本就在于在复制对象时，计数器没有递增，我们重新编写拷贝构造函数，如下：class Rect

{

public:

Rect() // 构造函数，计数器加1

{

count++;

}

Rect(const Rect& r) // 拷贝构造函数

{

width = r.width;

height = r.height;

count++; // 计数器加1

}

~Rect() // 析构函数，计数器减1

{

count--;

}

static int getCount() // 返回计数器的值

{

return count;

}

private:

int width;

int height;

static int count; // 一静态成员做为计数器

};

###### 2. 浅拷贝

所谓浅拷贝，指的是在对象复制时，只对对象中的数据成员进行简单的赋值，默认拷贝构造函数执行的也是浅拷贝。大多情况下“浅拷贝”已经能很好地工作了，但是一旦对象存在了动态成员，那么浅拷贝就会出问题了，让我们考虑如下一段代码：class Rect

{

public:

Rect() // 构造函数，p指向堆中分配的一空间

{

p = new int(100);

}

~Rect() // 析构函数，释放动态分配的空间

{

if(p != NULL)

{

delete p;

}

}

private:

int width;

int height;

int \*p; // 一指针成员

};

int main()

{

Rect rect1;

Rect rect2(rect1); // 复制对象

return 0;

}

在这段代码运行结束之前，会出现一个运行错误。原因就在于在进行对象复制时，对于动态分配的内容没有进行正确的操作。我们来分析一下：

在运行定义rect1对象后，由于在构造函数中有一个动态分配的语句，因此执行后的内存情况大致如下：

在使用rect1复制rect2时，由于执行的是浅拷贝，只是将成员的值进行赋值，这时

rect1.p = rect2.p，也即这两个指针指向了堆里的同一个空间，如下图所示：

当然，这不是我们所期望的结果，在销毁对象时，两个对象的析构函数将对同一个内存空间释放两次，这就是错误出现的原因。我们需要的不是两个p有相同的值，而是两个p指向的空间有相同的值，解决办法就是使用“深拷贝”。

###### 3. 深拷贝

在“深拷贝”的情况下，对于对象中动态成员，就不能仅仅简单地赋值了，而应该重新动态分配空间，如上面的例子就应该按照如下的方式进行处理：class Rect

{

public:

Rect() // 构造函数，p指向堆中分配的一空间

{

p = new int(100);

}

Rect(const Rect& r)

{

width = r.width;

height = r.height;

p = new int; // 为新对象重新动态分配空间

\*p = \*(r.p);

}

~Rect() // 析构函数，释放动态分配的空间

{

if(p != NULL)

{

delete p;

}

}

private:

int width;

int height;

int \*p; // 一指针成员

};

此时，在完成对象的复制后，内存的一个大致情况如下：

此时rect1的p和rect2的p各自指向一段内存空间，但它们指向的空间具有相同的内容，这就是所谓的“深拷贝”。

###### 4. 防止默认拷贝发生

通过对对象复制的分析，我们发现对象的复制大多在进行“值传递”时发生，这里有一个小技巧可以防止按值传递——声明一个私有拷贝构造函数。甚至不必去定义这个拷贝构造函数，这样因为拷贝构造函数是私有的，如果用户试图按值传递或函数返回该类对象，将得到一个编译错误，从而可以避免按值传递或返回对象。

// 防止按值传递

class CExample

{

private:

int a;

public:

//构造函数

CExample(int b)

{

a = b;

cout<<"creat: "<<a<<endl;

}

private:

//拷贝构造，只是声明

CExample(const CExample& C);

public:

~CExample()

{

cout<< "delete: "<<a<<endl;

}

void Show ()

{

cout<<a<<endl;

}

};

//全局函数

void g\_Fun(CExample C)

{

cout<<"test"<<endl;

}

int main()

{

CExample test(1);

//g\_Fun(test); 按值传递将出错

return 0;

}

##### 四. 拷贝构造函数的几个细节

1. 拷贝构造函数里能调用private成员变量吗?

解答：这个问题是在网上见的，当时一下子有点晕。其时从名子我们就知道拷贝构造函数其时就是一个特殊的构造函数，操作的还是自己类的成员变量，所以不受private的限制。

2. 以下函数哪个是拷贝构造函数,为什么?

X::X(const X&);

X::X(X);

X::X(X&, int a=1);

X::X(X&, int a=1, int b=2); 解答：对于一个类X, 如果一个构造函数的第一个参数是下列之一:

a) X&

b) const X&

c) volatile X&

d) const volatile X&

且没有其他参数或其他参数都有默认值,那么这个函数是拷贝构造函数.

X::X(const X&); //是拷贝构造函数

X::X(X&, int=1); //是拷贝构造函数

X::X(X&, int a=1, int b=2); //当然也是拷贝构造函数

3. 一个类中可以存在多于一个的拷贝构造函数吗?

解答：类中可以存在超过一个拷贝构造函数。

class X {

public:

X(const X&); // const 的拷贝构造

X(X&); // 非const的拷贝构造

};

注意,如果一个类中只存在一个参数为 X& 的拷贝构造函数,那么就不能使用const X或volatile X的对象实行拷贝初始化.

class X {

public:

X();

X(X&);

};

const X cx;

X x = cx; // error

如果一个类中没有定义拷贝构造函数,那么编译器会自动产生一个默认的拷贝构造函数。

这个默认的参数可能为 X::X(const X&)或

X::X(X&),由编译器根据上下文决定选择哪一个。

# 常见error和解决办法

### [E0144"const char \*" 类型的值不能用于初始化 "char \*" 类型的实体的三种解决方法](https://www.cnblogs.com/xiximayou/p/12121287.html)

例如：在以下程序源代码中会出现该提示：

#include <iostream>

#include <string>

using namespace std;

class Student {

private:

char \*m\_name;

int m\_age;

float m\_score;

public:

Student(char \*name, int age, float score);

void show();

};

//采用初始化列表

Student::Student(char \*name, int age, float score) : m\_name(name), m\_age(age), m\_score(score) {

//TODO:

}

void Student::show() {

cout << m\_name << "的年龄是" << m\_age << "，成绩是" << m\_score << endl;

}

int main() {

Student stu("小明", 15, 92.5f);

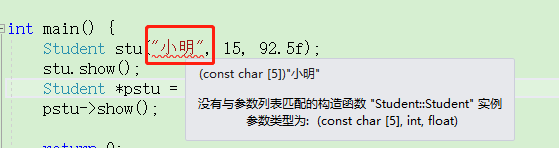
stu.show();

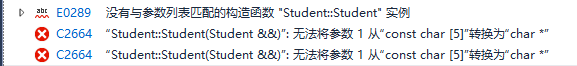
Student \*pstu = new Student("李华", 16, 96);

pstu->show();

return 0;

}

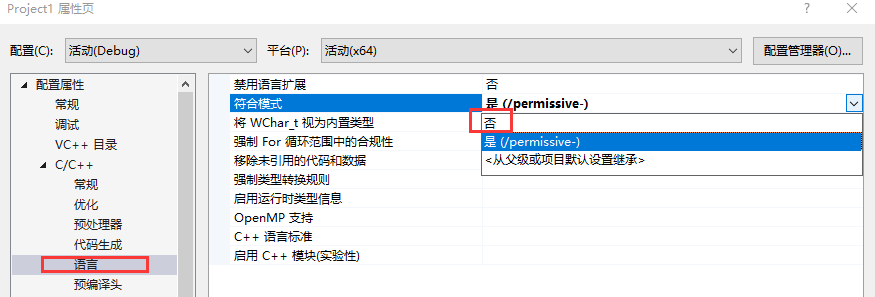
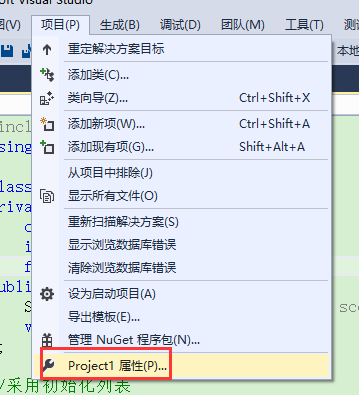




意思很明确const char\* 和char \*类型不匹配，个人理解为在创建stu对象的时候，系统将“小明”认定为是一个const类型的字符数组，所以跟student类中的char\* 类型相匹配。

解决的方法有以下几种：

##### 1、在visual studio中，在项目上右键，在弹出菜单上选择“属性”,选择“C/C++”,然后在“语言”中，把“符合模式”改为“否”就好了。



##### 2、使用强制类型转换：

//有问题代码char \*str2 = "hello world";

//正确的代码char \*str2 = (char \*)"hello world";

##### 3、先用字符数组进行存储，然后再使用指针：

char str1[] = "hello world";

char\* str2 = str1;

##### 4、将char类型定义改为 const char 类型

可以在以下地方添加const 关键字，来解决该问题。但是记得初始化const成员变量的时候，只能用初始化列表。

