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，这时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。

# 重载

## 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

## 区分重载(overload)，覆盖(Override)和隐藏(hide)

原文链接：<https://blog.csdn.net/jixingzhong/article/details/1858943>

重载overload，这个概念是大家熟知的。在同一可访问区内被声明的几个具有不同参数列的（参数的类型、个数、顺序不同）同名函数，程序会根据不同的参数列来确定具体调用哪个函数，这种机制就是重载。重载不关心函数的返回值类型，即返回类型不同无法构成重载。此外，C++ 中的const成员函数也可以构成overload。

总结一下重载的特征：

1、处在相同的空间中，即相同的范围内；

2、函数名相同；

3、参数不同，即参数个数不同，或相同位置的参数类型不同；

4、const成员函数可以和非const成员函数形成重载；

5、virtual关键字、返回类型对是否够成重载无任何影响。

覆盖override，是指派生类中存在重新定义的函数，其函数名、参数列、返回值类型必须同父类中的相对应被覆盖的函数严格一致，覆盖函数和被覆盖函数只有函数体（花括号中的部分）不同，当派生类对象调用子类中该同名函数时会自动调用子类中的覆盖版本，而不是父类中的被覆盖函数版本，这种机制就叫做覆盖，特征是：

1、不同的范围（分别位于派生类与基类）；

2、函数名字相同；

3、参数相同；

4、基类函数必须有virtual关键字。

针对上述两个概念，还有一个隐藏hide。所谓的隐藏，指的是派生类类型的对象、指针、引用访问基类和派生类都有的同名函数时，访问的是派生类的函数，即隐藏了基类的同名函数。隐藏规则的底层原因其实是Ｃ＋＋的名字解析过程。在继承机制下，派生类的类域被嵌套在基类的类域中。

派生类的名字解析过程如下：

1、首先在派生类类域中查找该名字。

2、如果第一步中没有成功查找到该名字，即在派生类的类域中无法对该名字进行解析，则编译器在外围基类类域对查找该名字的定义。

总结一下隐藏的特征：

1、如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。

2、如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

## 操作符重载

原文链接：https://blog.csdn.net/liitdar/article/details/80654324

1. 概述

1.1 what

operator 是C++的一个关键字，它和运算符（如=）一起使用，表示一个运算符重载函数，在理解时可将operator和运算符（如operator=）视为一个函数名。

使用operator重载运算符，是C++扩展运算符功能的方法。使用operator扩展运算符功能的原因如下：

使重载后的运算符的使用方法与重载前一致

扩展运算符的功能只能通过函数的方式实现（实际上，C++中各种“功能”都是由函数实现的）

1.2 why

对于C++提供的所有操作符，通常只支持对于基本数据类型和标准库中提供的类的操作，而对于用户自己定义的类，如果想要通过该操作符实现一些基本操作（比如比较大小，判断是否相等），就需要用户自己来定义关于这个操作符的具体实现了。

比如，我们要设计一个名为“person”的类，现在要判断person类的两个对象p1和p2是否一样大，我们设计的比较规则是按照其年龄来比较，那么，在设计person类的时候，就可以通过对操作符“==”进行重载，来使用操作符“==”对对象p1和p2进行比较了（根据前面的分析，实际上比较的内容应该是person类中的数据成员“age”）。

我们上面说的对操作符“==”进行重载，说是“重载”，是由于编译器在实现操作符“==”功能的时候，已经为我们提供了这个操作符对于一些基本数据类型的操作支持，只不过由于现在该操作符所操作的内容变成了我们自定义的数据类型（如class），而默认情况下，该操作符是不能对我们自定义的class类型进行操作的，所以，就需要我们通过重载该操作符，给出该操作符操作我们自定义的class类型的方法，从而达到使用该操作符对我们自定义的class类型进行运算的目的。

1.3 how

实现一个操作符重载的方式通常分为两种情况：

将操作符重载实现为类的成员函数；

操作符重载实现为非类的成员函数（即全局函数）。

1.3.1 将操作符重载实现为类的成员函数

在类体中声明（定义）需要重载的操作符，声明方式跟普通的成员函数一样，只不过操作符重载函数的名字是“关键字 operator +以及紧跟其后的一个C++预定义的操作符”，样式如下（person是我们定义的类）：

bool operator==(const person& ps)

{

if (this->age == ps.age)

{

return true;

}

return false;

}

示例代码（operator\_test2.cpp）如下：

#include <iostream>

using namespace std;

class person

{

private:

int age;

public:

person(int nAge)

{

this->age = nAge;

}

bool operator==(const person& ps)

{

if (this->age == ps.age)

{

return true;

}

return false;

}

};

int main()

{

person p1(10);

person p2(10);

if (p1 == p2)

{

cout << "p1 is equal with p2." << endl;

}

else

{

cout << "p1 is not equal with p2." << endl;

}

return 0;

}

编译并运行上述代码，结果如下：



通过上述结果能够知道：因为操作符重载函数“operator==”是person类的一个成员函数，所以对象p1、p2都可以调用该函数。其中的 if (p1 == p2) 语句，相当于对象p1调用函数“operator==”，把对象p2作为一个参数传递给该函数，从而实现了两个对象的比较。

1.3.2 操作符重载实现为非类的成员函数（即全局函数）

对于全局重载操作符，代表左操作数的参数必须被显式指定。

示例代码如下：

#include <iostream>

using namespace std;

class person

{

public:

int age;

};

// 左操作数的类型必须被显式指定

// 此处指定的类型为person类

bool operator==(person const& p1, person const& p2)

{

if (p1.age == p2.age)

{

return true;

}

else

{

return false;

}

}

int main()

{

person p1;

person p2;

p1.age = 18;

p2.age = 18;

if (p1 == p2)

{

cout << "p1 is equal with p2." << endl;

}

else

{

cout << "p1 is NOT equal with p2." << endl;

}

return 0;

}编译并运行上述代码，结果如下：



1.3.4 操作符重载的方式选择

可以根据以下因素，确定把一个操作符重载为类的成员函数还是全局函数：

如果一个重载操作符是类成员，那么只有当与它一起使用的左操作数是该类的对象时，该操作符才会被调用；而如果该操作符的左操作数确定为其他的类型，则操作符必须被重载为全局函数；

C++要求'='、'[]'、'()'、'->'操作符必须被定义为类的成员操作符，把这些操作符通过全局函数进行重载时会出现编译错误

如果有一个操作数是类类型（如string类），那么对于对称操作符（比如==操作符），最好通过全局函数的方式进行重载。

1.3.5 操作符重载的限制

实现操作符重载时，需要注意：

重载后操作符的操作数至少有一个是用户定义类型；

不能违反原来操作数的语法规则；

不能创建新的操作符；

不能重载的操作符包括（以空格分隔）：sizeof . .\* :: ?: RTTI类型运算符

=、()、[]、以及 ->操作符只能被类的成员函数重载

## 赋值运算符 = 重载

https://blog.csdn.net/liitdar/article/details/80656156

### 1. 概述

##### 1.1 why

首先介绍为什么要对赋值运算符“=”进行重载。某些情况下，当我们编写一个类的时候,，并不需要为该类重载“=”运算符，因为编译系统为每个类提供了默认的赋值运算符“=”，使用这个默认的赋值运算符操作类对象时，该运算符会把这个类的所有数据成员都进行一次赋值操作。例如有如下类：

class A

{

public:

int a;

int b;

int c;

};

那么对这个类的对象进行赋值时，使用默认的赋值运算符是没有问题的。示例代码（operator\_test4.cpp）如下：

#include <iostream>

using namespace std;

class ClassA

{

public:

int a;

int b;

int c;

};

int main()

{

ClassA obj1;

obj1.a = 1;

obj1.b = 2;

obj1.c = 3;

ClassA obj2;

obj2 = obj1;

cout << "obj2.a is: " << obj2.a << endl;

return 0;

}

编译并执行上述代码，结果如下：



从上述结果能够知道：通过使用系统默认的赋值运算符“=”，可以让对象obj2中的所有数据成员的值与对象obj1相同。这种情况下，编译系统提供的默认赋值运算符可以正常使用。

但是，在下面的示例中，使用编译系统默认提供的赋值运算符，就会出现问题了。示例代码（operator\_test5.cpp）如下：

#include <iostream>

#include <string.h>

using namespace std;

class ClassA

{

public:

ClassA()

{

}

ClassA(const char\* pszInputStr)

{

pszTestStr = new char[strlen(pszInputStr) + 1];

strncpy(pszTestStr, pszInputStr, strlen(pszInputStr) + 1);

}

virtual ~ClassA()

{

delete pszTestStr;

}

public:

char\* pszTestStr;

};

int main()

{

ClassA obj1("liitdar");

ClassA obj2;

obj2 = obj1;

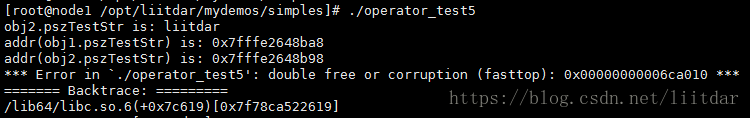
cout << "obj2.pszTestStr is: " << obj2.pszTestStr << endl;

cout << "addr(obj1.pszTestStr) is: " << &obj1.pszTestStr << endl;

cout << "addr(obj2.pszTestStr) is: " << &obj2.pszTestStr << endl;

return 0;

}编译并运行上述代码，结果如下：



上述错误信息说明：当obj1和obj2进行析构的时候，由于重复释放了一块内存，导致程序崩溃报错。在这种情况下，就需要我们重载赋值运算符“=”了。

### 2. 示例代码

##### 2.1 示例代码1

我们修改一下前面出错的代码示例，现编写一个包含赋值运算符重载函数的类，代码（operator\_test5.cpp）如下：

#include <iostream>

#include <string.h>

using namespace std;

class ClassA

{

public:

ClassA()

{

}

ClassA(const char\* pszInputStr)

{

pszTestStr = new char[strlen(pszInputStr) + 1];

strncpy(pszTestStr, pszInputStr, strlen(pszInputStr) + 1);

}

virtual ~ClassA()

{

delete pszTestStr;

}

// 赋值运算符重载函数

ClassA& operator=(const ClassA& cls)

{

// 避免自赋值

if (this != &cls)

{

// 避免内存泄露

if (pszTestStr != NULL)

{

delete pszTestStr;

pszTestStr = NULL;

}

pszTestStr = new char[strlen(cls.pszTestStr) + 1];

strncpy(pszTestStr, cls.pszTestStr, strlen(cls.pszTestStr) + 1);

}

return \*this;

}

public:

char\* pszTestStr;

};

int main()

{

ClassA obj1("liitdar");

ClassA obj2;

obj2 = obj1;

cout << "obj2.pszTestStr is: " << obj2.pszTestStr << endl;

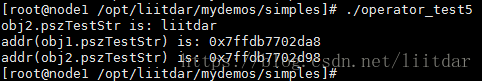
cout << "addr(obj1.pszTestStr) is: " << &obj1.pszTestStr << endl;

cout << "addr(obj2.pszTestStr) is: " << &obj2.pszTestStr << endl;

return 0;

}

编译并运行上述代码，结果如下：



通过上述结果能够看到，我们利用赋值运算符重载函数，解决了对象赋值的情况下，析构函数中过程中多次释放同一块内存的问题。

对于上述代码，有以下几点需要说明：

当为一个类的对象赋值（可以用本类对象为其赋值，也可以用其它类型的值为其赋值）时，该对象（如本例的obj2）会调用该类的赋值运算符重载函数，进行具体的赋值操作。如上述代码中的“obj2 = obj1;”语句，用obj1为obj2赋值，则会由obj2调用ClassA类的赋值运算符重载函数。

语句“ClassA obj2;

obj2 = obj1;“

和语句“ClassA obj2 = obj1;”在调用函数上是有区别的：前者第一句是对象obj2的声明及定义，调用类ClassA的无参构造函数，所以“obj2 = obj1;”一句是在对象obj2已经存在的情况下，用obj1来为obj2赋值，调用的是赋值运算符重载函数；而后者，是用obj1来初始化obj2，调用的是拷贝构造函数。关于拷贝构造函数的语句样式为“ClassA(const ClassA& cls)”，关于拷贝构造函数的内容，此处不进行详述。

当程序没有显式地提供一个以“本类或本类的引用”为参数的赋值运算符重载函数时，编译器会自动生成一个默认的赋值运算符重载函数。

##### 2.2 示例代码2

示例代码（operator\_test6.cpp）如下：

#include<iostream>

#include<string>

using namespace std;

class Data

{

private:

int data;

public:

// 构造函数

Data()

{

};

// 构造函数

Data(int \_data) :data(\_data)

{

cout << "This is constructor" << endl;

}

// 赋值运算符重载函数

Data& operator=(const int \_data)

{

cout << "This is operator=(int \_data)" << endl;

data = \_data;

return \*this;

}

};

int main()

{

// 调用构造函数

Data data1(1);

Data data2, data3;

// 调用赋值运算符重载函数

data2 = 1;

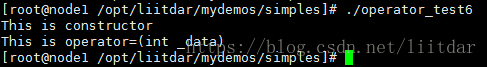
// 调用默认的赋值运算符重载函数

data3 = data2;

return 0;

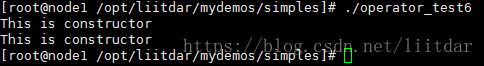
}

编译并执行上述代码，结果如下：



上述结果说明：“data2 = 1;”语句调用了我们提供的以int型参数（而非本类或本类的引用）为形参的赋值运算符重载函数，而“data3 = data2;”的成功执行，说明该语句调用了编译器提供的默认的赋值运算符重载函数。

如果将上述代码中赋值运算符重载函数去掉，重新编译执行，结果如下：



上述结果说明，当用一个非类A的值（如上面的int类型值）为类A的对象赋值时：

如果检测到构造函数和赋值运算符重载函数同时存在，则会调用赋值运算符重载函数；

如果检测到的构造函数，就会调用这个构造函数。

## 3. 总结

综合上述示例内容，我们可以知道针对以下情况，需要显式地提供赋值运算符重载函数（即自定义赋值运算符重载函数）：

用非类A类型的值为类A的对象赋值时（当然，这种情况下我们可以不提供相应的赋值运算符重载函数，而只提供相应的构造函数，如更改后的示例代码2）。

当用类A类型的值为类A的对象赋值，且类A的数据成员中含有指针的情况下，必须显式提供赋值运算符重载函数（如示例代码1）。

# 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 \*/

**注意：**

在C++程序编写过程中，能使用new和delete进行内存分配和删除操作的，就不要使用malloc和free。malloc /free主要为了兼容C，new和delete 完全可以取代malloc /free的。malloc /free的操作对象都是必须明确大小zhi的。而且不能用在动态类上。new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。当然从理论上说使用malloc申请的内存是可以通过delete释放的。不过一般不这样写的。而且也不能保证每个C++的运行时都能正常。

另外，**这两组操作符不要混用**，理论上没有强制规定这两组要分别搭配使用，对于简单的类对象或者其他简单变量等操作对象来说，这两组操作符错配使用没有什么影响。但是对于较复杂的类，比如构造和析构函数有自定义操作，或者new操作符有自定义的情况下，就很可能会出现错误，或者表面上没有出现错误，却会隐藏一些安全隐患并且不易被发现，对项目的安全性会有很大的影响。

**所以，从项目安全性和使用规则上来说，这两组操作符不要进行错配使用，堆上动态分配的内存一定要记得回收！！！**

## 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;

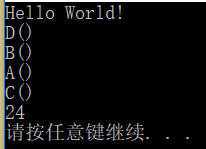
C c; //D, B, A ,C

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

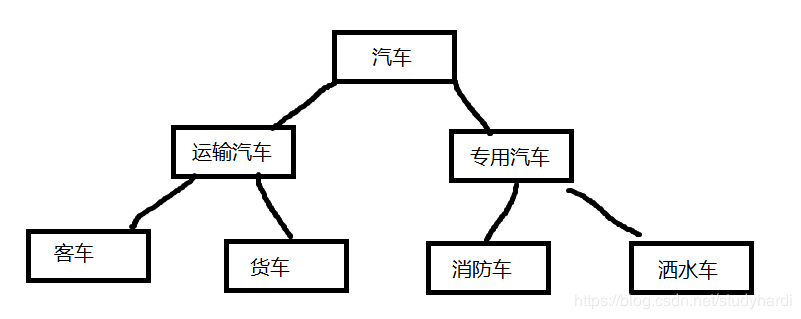
|  |  |  |  |
| --- | --- | --- | --- |
| 使用虚拟继承，类C继承A和B | 使用虚拟继承，去掉类A | 不使用虚拟继承，去掉类A | 不使用虚拟继承，类C继承A和B |
|  |  |  |  |

## 网上博客：

https://blog.csdn.net/studyhardi/article/details/90744785

### 1.为什么要使用继承

继承性是一个非常自然的概念，现实世界中的许多事物都是具有继承性的。人们一般用层次分类的方法来描述他们的关系。例如，下图就是一个简单的汽车分类图：



在这个分类树中建立了一个层次结构，最高一层是最普遍、最一般的，每一层都比它的前一层更具体，低层具有高层的特性，同时也有与高层的细微不同。例如，确定了一辆车是客车以后，没有必要再指出它可以进行运输，因为客车本身就是从运输汽车类中派生出来的。

##### 1.1 继承的概念

继承机制是面向对象程序设计使代码可以复用的最重要的手段，它允许程序员在保持原有的特性基础上进行扩展，增加功能，这样产生新的类，称作是派生类。继承呈现了面向对象程序设计的层析结构，体现了由简单到复杂的认知过程。继承是类设计层次的复用。

例如：

class Person

{

public:

void Print(){

cout<<"name:"<<\_name<<endl;

cout<<"age:"<<\_age<<endl;

}

protected:

string \_name = "Romeo"; //姓名

int \_age = 18; //年龄

};

/\*继承后父类的Person的成员（成员函数+变量）都会变成子类的一部分。这里

体现出了Student和Teacher复用了Person的成员。\*/

class Student: public Person

{

protected:

int \_stuid; //学号

};

class Teacher:public Person

{

protected:

int \_jobid; //工号

};

##### 1.2 继承的定义

class Student:public Person

{

public:

int \_stuid; //学号

char \_major; //专业

};

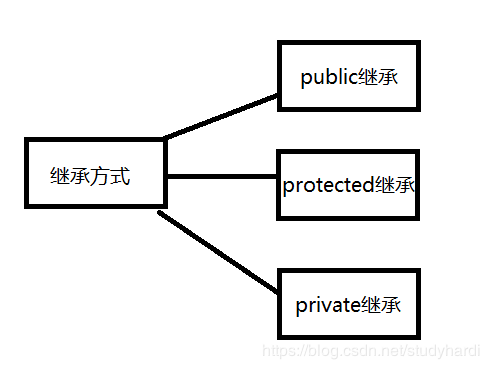
123456

Student称为 派生类；

第一行的public是继承方式；

Person称为基类。

以下是三种继承方式：



以下是继承基类成员访问方式的变化：



##### 总结：

1.基类private成员在派生类中无论以什么方式继承都是不可见的。这里的不可见是指基类的私有成员还是被继承到了派生类对象中，但是语法上限制派生类对象不管在类里面还是类外面都不能去访问它。

2.基类private成员在派生类中是不能被访问，如果基类成员不想在类外直接访问，但需要在派生类中能访问，就定义为protected。可见，保护成员限定符protected是因为继承才出现的。

3.表格里的访问方式都是取最小的“权限”。

4.使用关键字class时默认的继承方式是private，使用struct的默认继承方式是public，不过最好显示地写出继承方式。

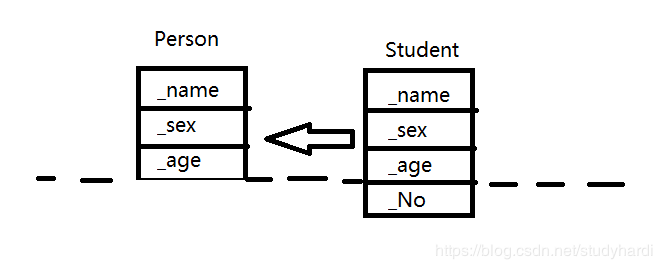
5.\*\*在实际运用中一般都使用的是public继承，几乎很少去使用protected/private继承，\*\*也不提倡去使用。因为protected/private继承下来的成员都只能在派生类的类里面使用，实际中的扩展维护性不强。

2.基类和派生类对象赋值转换

a. 派生类对象可以赋值给基类对象/基类的指针/基类的引用。这里有个形象的说法叫“切片”或者“切割”。寓意把派生类中父类那部分切来赋值过去。

b. 基类对象不能赋值给派生类对象。

c. 基类的指针可以通过强制类型转换赋值给派生类指针。但是必须是基类的指针指向派生类对象时才是安全的。 这里的基类如果是多态类型，可以使用dynamic \_cast来进行识别后进行安全转换。如下图所示：



示例代码：

class Person

{

protected:

string \_name; //姓名

string \_sex; //性别

int \_age; //年龄

};

class Student:public Person

{

public:

int \_No; //学号

};

void Test()

{

Student sobj;

//1.子类对象可以赋值给父类对象/指针/引用

Person pobj = sobj;

Person\* pp = &sobj;

Person& rp = sobj;

//2.基类对象不能赋值给派生类对象

sboj = pobj; //error

//3.基类的指针可以通过强制类型转换赋值给派生类指针

pp = &sobj;

Student\* ps1 = (Student\*)pp;

ps1->\_No = 10;

pp = &pobj;

Student\* ps2 = (Student\*)pp; //这种情况虽然可以，但会发生访问越界。

ps2->\_No = 10;

}

int main()

{

Test();

return 0;

}

### 3.继承中的作用域

（1） 在继承体系中基类和派生类都有独立的作用域。

（2） 子类和父类中有同名成员，子类成员将屏蔽父\*\*\*\*类对同名成员的直接访问，这种情况叫“隐藏”，也叫“重定义”（在子类成员函数中，可以使用 基类::基类成员 显示访问 ）

（3） 需要注意的是如果成员函数的隐藏，只需要函数名相同就构成了隐藏。

（4） 注意：在实际工程中，继承体系里最好不要定义同名成员。

示例1：

//Student的\_num和Person的\_num构成隐藏关系，可以看出这样的代码虽然能跑，但是非常容易混淆

class Person

{

protected:

string \_name = "小李子"; //姓名

int \_num = 111; //身份证号

};

class Student :public Person

{

public:

void Print(){

cout << "姓名： " << \_name << endl;

cout << "身份证： " << Person::\_num << endl;

cout << "学号： " << \_num << endl;

}

protected:

int \_num = 999; //学号

};

void Test()

{

Student s1;

s1.Print();

}

示例2：

//B中的fun和A中的fun不构成重载，因为不在同一作用域

//B中的fun和A中的fun构成隐藏，成员函数满足函数名相同就构成隐藏

class A

{

public:

void fun(){

cout << "fun()" << endl;

}

};

class B :public A

{

public:

void fun(int i){

A::fun();

cout << "fun(int i)-> " << i << endl;

}

};

void Test()

{

B b;

b.fun(10);

}

##### 辨析：

a. 成员函数被重载的特征：

（1）. 相同的范围（在同一个类中）；

（2）. 函数名字相同；

（3）. 参数不同；

（4）. virtual关键字可有可无。

b. 覆盖是指派生类函数覆盖基类函数。

（1）. 不同的范围（分别位于基类和派生类）；

（2）. 函数名字相同；

（3）. 参数相同；

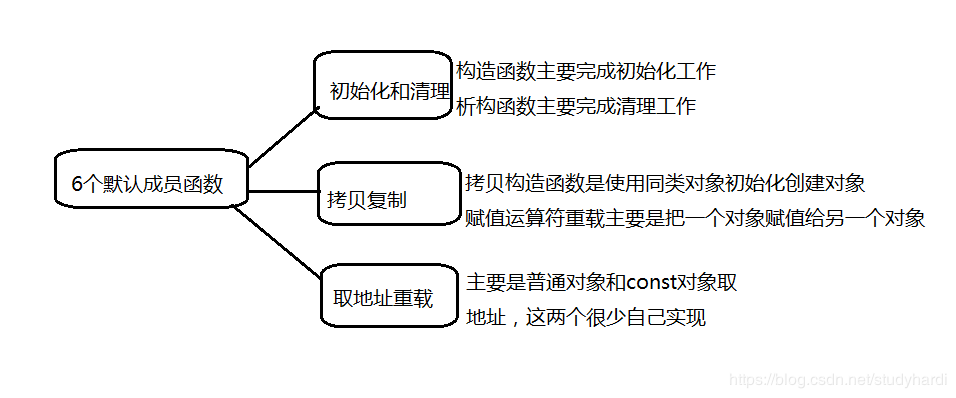
（4）. 基类函数必须有virtual关键字。

c. “隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）. 如果派生类的函数与基类函数同名，但是参数不同。此时，不论有无virtual, 基类的函数将被隐藏；

（2）. 如果派生类的函数与基类的函数同名，并且形参也相同，但是基类函数没有virtual，此时，基类的函数被隐藏。

### 4. 派生类的默认成员函数



a. 派生类的构造函数必须调用基类的构造函数初始化基类的那一部分初始化成员。如果基类没有默认的构造函数，则必须在派生类构造函数的初始化列表阶段显示调用；

b. 派生类的拷贝构造函数必须调用基类的拷贝构造函数完成基类的拷贝初始化；

c. 派生类的operator=必须要调用基类的operator=完成基类的赋值；

d. 派生类的析构函数会在被调用完成后自动调用基类的析构函数清理基类成员。因为这样才能保证派生类对象先清理，基类对象再清理的顺序；

e. 派生类对象初始化先调用基类构造函数再调用派生类构造函数；

f. 派生类对象析构清理先调用派生类的析构函数再调用基类的析构函数。

示例代码：

#include <iostream>

#include <string>

#include <stdlib.h>

using namespace std;

class Person

{

public:

Person(const char\* name = "Romeo")

:\_name(name){

cout << "Person()" << endl;

}

Person(const Person& p)

:\_name(p.\_name){

cout << "Person(const Person& p)" << endl;

}

Person& operator=(const Person& p){

cout << "Person& operator=(const Person& p)" << endl;

if (this != &p){

\_name = p.\_name;

}

return \*this;

}

~Person(){

cout << "~Person()" << endl;

}

protected:

string \_name; //姓名

};

class Student :public Person

{

public:

Student(const char\* name, int num)

:Person(name) //调用父类的构造函数完成从父类继承来的成员的初始化

, \_num(num)

{

cout << "Student()" << endl;

}

Student(const Student& s)

:Person(s) //调用父类的拷贝构造函数，其中也为“切片”操作，因为父类的拷贝构造函数的参数为引用

, \_num(s.\_num)

{

cout << "Student(const Student& s)" << endl;

}

Student& operator=(const Student& s){

cout << "Student& operator=(const Student& s)" << endl;

if (this != &s){

Person::operator=(s); //调用基类的

\_num = s.\_num;

}

return \*this;

}

~Student(){

//此处不需要显示调用父类的析构函数，编译器会自动调用

cout << "~Student()" << endl;

}

protected:

int \_num; //学号

};

void Test()

{

Student s1("jack", 18);

Student s2(s1);

Student s3("rose", 17);

s1 = s3;

}

int main()

{

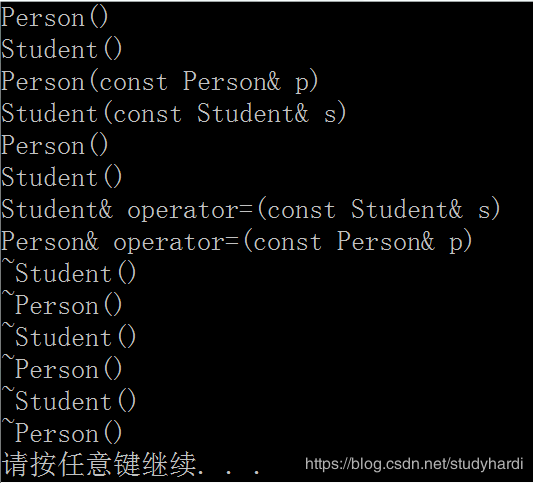
Test();

system("pause");

return 0;

}

结果：



### 5. 继承与友元

友元关系不能继承，也就是说基类友元不能访问子类私有和保护成员。

#include <iostream>

#include <string>

#include <stdlib.h>

using namespace std;

class Person

{

public:

friend void Display(const Person& p, const Student& s);

protected:

string \_name; //姓名

};

class Student :public Person

{

protected:

int \_stuNum; //学号

};

void Display(const Person& p, const Student& s)

{

cout << p.\_name << endl; //error

cout << s.\_stuNum << endl; //error

}

int main()

{

Person p;

Student s;

Display(p, s);

return 0;

}



### 6. 继承与静态成员

基类定义了static静态成员，则整个继承体系里只有一个这样的成员。无论派生出多少个子类，都有一个static成员实例。

示例代码：

#include <iostream>

#include <string>

#include <stdlib.h>

using namespace std;

class Person

{

public:

Person(){

++\_count;

}

protected:

string \_name; //姓名

public:

static int \_count; //统计人数

};

int Person::\_count = 0;

class Student :public Person

{

protected:

int \_stuNum; //学号

};

class Graduate : public Student

{

protected:

string \_seminarCourse; //研究科目

};

void TestPerson()

{

Student s1;

Student s2;

Student s3;

Graduate s4;

cout << "人数: " << Person::\_count << endl;

Student::\_count = 0;

cout << "人数: " << Person::\_count << endl;

}

int main()

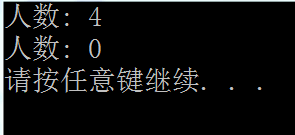
{

TestPerson();

system("pause");

return 0;

}



### 7.复杂的菱形继承和菱形虚拟继承

这里给大家贴了一个链接：

https://blog.csdn.net/audience\_fzn/article/details/81463275

### 8. 继承里的指针偏移问题

class Base1{

public:

int \_b1;

};

class Base2{

public:

int \_b2;

};

class Derive : public Base1, public Base2{

int \_d;

};

int main()

{

Derive d;

Base1\* p1 = &d;

Base2\* p2 = &d;

Derive\* p3 = &d;

cout << "p1 = " << p1 << endl;

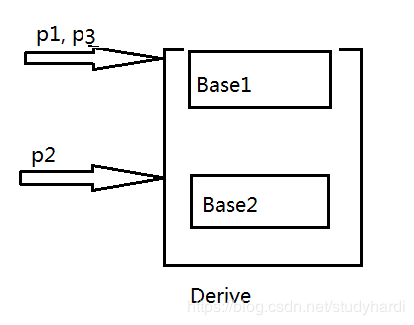
cout << "p2 = " << p2 << endl;

cout << "p3 = " << p3 << endl;

system("pause");

return 0;

}

多重继承时，子类指针转换为非第一继承的父类指针时，会发生地址偏移。这是因为每一个父类都会占用 4 个字节维护自己的虚函数表。

### 9．继承里的名称遮掩问题

<https://blog.csdn.net/qq_35987777/article/details/102894124>

C++中通过基类指针调用派生类中定义的方法&&C++继承中的名称遮掩动态绑定静态绑定C++继承中的名称遮掩dynamic\_cast<>动态转型的作用区分接口继承和实现继承

##### 动态绑定

动态类型指“目前所指对象的类型”；

动态绑定是指在执行期间（非编译期）判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

#include <iostream>

using namespace std;

class BaseA

{

public:

virtual void display(){

cout << "BaseA::display()" << endl;

}

virtual void display(int a){

cout<<"BaseA::display(a)" <<endl;

}

};

class DerivedA : public BaseA

{

public:

virtual void display(){

cout << "DerivedA::display()" << endl;

}

};

int main()

{

DerivedA t;

BaseA\* pt1=&t;

pt1->display();

DerivedA\* pt2=&t;

pt2->display();

//pt2->display(2);

//错误，DerivedA中的函数display()在“名称遮掩规则”下遮掩了BasedA中的函数display(int a)

system("pause");

return 0;

}

结果：



virtual函数是动态绑定，动态绑定时通过指针调用的是该指针指向的对象中的成员函数。

virtual函数是动态绑定，而成员函数缺省参数却是静态绑定

##### 静态绑定

静态类型指在程序声明时所采取的类型，

静态绑定是指在程序编译过程中，把函数（方法或者过程）调用与响应调用所需的代码结合的过程称之为静态绑定。

#include <iostream>

using namespace std;

class BaseA

{

public:

void display(){

cout << "BaseA::display()" << endl;

}

void display(int a){

cout<<"BaseA::display(a)" <<endl;

}

};

class DerivedA : public BaseA

{

public:

void display(){

cout << "DerivedA::display()" << endl;

}

};

int main()

{

DerivedA t;

BaseA\* pt1=&t;

pt1->display();//BasedA::display()

DerivedA\* pt2=&t;

pt2->display();//DerivedA::display()

//pt2->display(2);

//错误，DerivedA中的函数display()在“名称遮掩规则”下遮掩了BasedA中的函数display(int a)

system("pause");

return 0;

}

结果：



这个示例只是为了说明静态绑定时，通过指针调用的是该指针声明时的类型对应的类对象中的成员函数。

但是必须强调一下：不要在派生类中重新定义基类的非虚函数

从面向对象的角度来说，公有继承是一种Is-a的关系。其表明每一个派生类对象都可以被当做基类对象来处理。基类的每一个接口和成员变量派生类也有。如果重定义了非虚函数，会导致派生类对象将不再是基类对象（当通过指向派生类对象的指针访问时）

很少情况下，需要重定义非虚函数。一个特例是为了解决私有继承中的名称遮掩问题。在私有继承中，基类的公有函数在派生类中都是私有，如果派生类想要继承基类的某个接口，可以使用所谓的转交函数（forwarding function）。即定义一个public的与基类中那个接口同名的函数，这个函数一般是inline,函数体实现部分仅调用基类的对应函数即可。

关于C++继承中的名称遮掩问题，下面细讲一下：

##### C++继承中的名称遮掩

示例：

#include <iostream>

using namespace std;

class BaseA

{

public:

void display() {

cout << "BaseA::display()" << endl;

}

void display(int a) {

cout << "BaseA::display(a)" << endl;

}

};

class DerivedA : public BaseA

{

public:

void display() {

cout << "DerivedA::display()" << endl;

}

};

int main()

{

DerivedA t;

BaseA\* pt1 = &t;

pt1->display();//BasedA::display()

pt1->display(2);//静态绑定，所以还能通过基类指针去调用派生类中的display(int a)函数

DerivedA\* pt2 = &t;

pt2->display();//DerivedA::display()

//pt2->display(2);

//错误，DerivedA中的函数display()在“名称遮掩规则”下遮掩了BasedA中的函数display(int a)

system("pause");

return 0;

}

结果：



上面的代码展示了，DerivedA类“没有能够成功继承”BaseA中的display(int a)函数。打上引号的原因是，我们使用基类指针指向派生类时，又能够成功调用。

从名称查找的观点出发，BaseA::display(int a)被DerivedA::display()函数遮蔽了。

编译器拒绝这样的继承是为了防止在程序库或应用框架内建立新的derived class 时附带地从疏远的base classes 继承重载函数。

如果你想从基类BasedA中继承重载了的display()和display(int a)函数，同时又想在派生类DerivedA中重新定义display()函数时不遮掩display(int a)函数名，

可以通过在派生类DerivedA中用using声明基类BasedA中的函数名：

#include <iostream>

using namespace std;

class BaseA

{

public:

void display() {

cout << "BaseA::display()" << endl;

}

void display(int a) {

cout << "BaseA::display(a)" << endl;

}

};

class DerivedA : public BaseA

{

public:

using BaseA::display;//让BaseA class内名为dislay的所有东西在DerivedA中都可见(并且public)

void display() {

cout << "DerivedA::display()" << endl;

}

};

int main()

{

DerivedA t;

BaseA\* pt1 = &t;

pt1->display();//BasedA::display()

pt1->display(2);//静态绑定，所以还能通过基类指针去调用派生类中的display(int a)函数

DerivedA\* pt2 = &t;

pt2->display();//DerivedA::display()

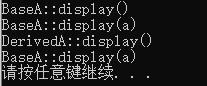
pt2->display(2);//正确， DerivedA类中使用using BaseA::display，避免了名称遮掩

system("pause");

return 0;

}

结果：



##### 总结：

derived classes 内的名称会遮掩base classes 内的名称；如果继承base class并加上重载函数，而又希望在derive中重新定义或覆写其中一部分，那必须手动使用using 声明那些覆写的函数，否则某些希望继承的名称会被遮掩。

但有时候，比如在private继承时(表示的是一种根据基类实现派生类的关系，is-implemented-in-terms-of)，基类的公有函数在派生类中都是私有。如果派生类不想继承base class的所有函数接口。而只想要继承基类的某一个接口，可以使用所谓的转交函数（forwarding function）。

##### 转交函数（forwarding function）

以上面的例子为例，DerviedA只想继承BaseA::display()函数，

#include <iostream>

using namespace std;

class BaseA

{

public:

virtual void display() {

cout << "BaseA::display()" << endl;

}

virtual void display(int a) {

cout << "BaseA::display(a)" << endl;

}

};

class DerivedA : private BaseA

{

public:

//using BaseA::display;//让BaseA class内名为dislay的所有东西在DerivedA中都可见(并且public)

virtual void display() {

BaseA::display();

}

};

int main()

{

DerivedA t;

t.display();

//t.display(2);//错误

system("pause");

return 0;

}

结果：

在这里插入图片描述

##### dynamic\_cast<>动态转型的作用

dynamic\_cast:将指向base class objects的pointers或references转型为指向derived(或sibling base)clasee objects的pointers或references，

#include <iostream>

using namespace std;

class BaseA {

public:

virtual void display() = 0;

};

class DerivedA : public BaseA {

public:

virtual void display() {

cout << "DerivedA::display()" << endl;

}

void displaymore() {

cout << "DerivedA::displaymore()" << endl;

}

};

int main() {

BaseA \*pt = new DerivedA;

pt->display();

//pt->displaymore();//error: ‘class BaseA’ has no member named ‘display()’

dynamic\_cast<DerivedA\*>(pt)->displaymore();

system("pause");

return 0;

}

结果：



##### 区分接口继承和实现继承

在设计类的继承关系时，

有时候你希望derived classes只继承成员函数的接口（即声明）；

有时候你希望derived classes同时继承成员函数的接口和实现，但又希望能够覆写它们所继承的实现；

有时候你希望derived classes同时继承成员函数的接口和实现，并且不允许覆写任何东西；

声明一个pure virtual 函数的目的是为了让derived classes只继承成员函数的接口

声明impure virtual函数(非纯虚函数）是为了让derived classes同时继承成员函数的接口和缺省实现

声明non-virtual 函数的目的是为了让derived classes同时继承成员函数的接口和一份强制性实现；

## 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。

## C++虚函数详解

https://blog.csdn.net/weixin\_43329614/article/details/89103574

前言

C++的特性使得我们可以使用函数继承的方法快速实现开发，而为了满足多态与泛型编程这一性质，C++允许用户使用虚函数 (virtual function) 来完成 运行时决议 这一操作，这与一般的 编译时决定 有着本质的区别。

虚函数表实现原理

虚函数的实现是由两个部分组成的，虚函数指针与虚函数表。

虚函数指针

虚函数指针 (virtual function pointer) 从本质上来说就只是一个指向函数的指针，与普通的指针并无区别。它指向用户所定义的虚函数，具体是在子类里的实现，当子类调用虚函数的时候，实际上是通过调用该虚函数指针从而找到接口。

虚函数指针是确实存在的数据类型，在一个被实例化的对象中，它总是被存放在该对象的地址首位，这种做法的目的是为了保证运行的快速性。与对象的成员不同，虚函数指针对外部是完全不可见的，除非通过直接访问地址的做法或者在DEBUG模式中，否则它是不可见的也不能被外界调用。

只有拥有虚函数的类才会拥有虚函数指针，每一个虚函数也都会对应一个虚函数指针。所以拥有虚函数的类的所有对象都会因为虚函数产生额外的开销，并且也会在一定程度上降低程序速度。与JAVA不同，C++将是否使用虚函数这一权利交给了开发者，所以开发者应该谨慎的使用。

虚函数表

上文已经提到，每个类的实例化对象都会拥有虚函数指针并且都排列在对象的地址首部。而它们也都是按照一定的顺序组织起来的，从而构成了一种表状结构，称为虚函数表 (virtual table) 。

我们先来规定一个基类

class Base

{

public:

virtual void f(){cout<<"Base::f"<<endl;}

virtual void g(){cout<<"Base::g"<<endl;}

virtual void h(){cout<<"Base::h"<<endl;}

};

12345678

首先对于基类Base它的虚函数表记录的只有自己定义的虚函数

接下来我们来看看子类的情况

class Derived:public Base

{

public:

virtual void f(){cout<<"Derived::f"<<endl;}

virtual void g1(){cout<<"Derived::g1"<<endl;}

virtual void h1(){cout<<"Derived::h1"<<endl;}

}

1234567

·一般覆盖继承

首先是最常见的继承，子类Derived对基类的虚函数进行覆盖继承，在这个例子中仅设计了一个函数继承的情况以此推广情况。

那么此时情况是这样的：

首先基函数的表项仍然保留，而得到正确继承的虚函数其指针将会被覆盖，而子类自己的虚函数将跟在表后。

而当多重继承的时候，表项将会增多，顺序会体现为继承的顺序，并且子函数自己的虚函数将跟在第一个表项后。

C++中一个类是公用一张虚函数表的，基类有基类的虚函数表，子类是子类的虚函数表，这极大的节省了内存

同名覆盖原则与const修饰符

如果继续深入下去的话我们将会碰见一个有趣的状况

class Base

{

public:

virtual void func()const

{

cout << "Base!" << endl;

}

};

class Derived :public Base

{

public:

virtual void func()

{

cout << "Derived!" << endl;

}

};

void show(Base& b)

{

b.func();

}

Base base;

Derived derived;

int main()

{

show(base);

show(derived);

base.func();

derived.func();

return 0;

}

1234567891011121314151617181920212223242526272829303132

在上述程序中我们将Base类中的虚函数base定义为const类型，我们知道const后缀的目的是为了限定该函数不对类内成员做出修改。然后我们分别声明base与derived并且通过show函数调用它们的func函数，子类传参给父类也是非常正常的一个操作，但是结果可能却令人不解：

Base!

Base!

Base!

Derived!

1234

这里有一个很大的问题，因为当我们将derived传过去的时候并没有调用derived的虚函数！也就是说虚函数不再是多态的了。

但是的话我们只需要简单的修改任意一项：将line4结尾的const限定符去掉或者将Derived的func1后加上const便可以使一切正常。这是为什么呢？

很多其他的博客将其解释为是const符号作用的原因，但实际上这样的解释并不正确。正确的原因是:

虚函数的声明与定义要求非常严格，只有在子函数中的虚函数与父函数一模一样的时候（包括限定符）才会被认为是真正的虚函数，不然的话就只能是重载。这被称为虚函数定义的同名覆盖原则，意思是只有名称完全一样时才能完成虚函数的定义。

因此在上述的例子中，将Derived类型的子类传入show函数时，实际上类型转化为了Base，由于此时虚函数并未完成定义，Derived的func()此时仅仅是属于Derived自己的虚函数，所以在show中b并不能调用，而调用的是Base内的func。而当没有发生类型转换的时候，Base类型与Derived类型就会各自调用自己的func函数。

————————————————

版权声明：本文为CSDN博主「Whitesad\_」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/weixin\_43329614/article/details/89103574

## 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>

指针和引用形式上很好区别，但是他们似乎有相同的功能,都能够直接引用对象，对其进行直接的操作。但是什么时候使用指针？什么时候使用引用呢？这两者很容易混淆,在此我详细介绍一下指针和引用，力争将最真实的一面展现给大家。如果我喷得不够好，希望嘴下留情、手下留命，还请指点一二；如果感觉还不错，请大家鼓掌。

       1、指针和引用的定义

       在深入介绍之前我们首先来看一下指针和引用的定义、指针和引用的区别，然后分别针对指针和引用展开讨论，深入细节为何有这些差异。

       ● 指针的权威定义：

In a declaration T D where D has the form

\* cv-qualifier-seqopt D1

And the type of the identifier in the declaration T D1 is "derived-declarator-type-list T", then the type of the identifier of D is "derived-declarator-type-list cv-qualifier-seq pointer to T". The cv-qualifiers apply to the pointer and not to the object pointer to.

——摘自《ANSI C++ Standard》

       注：可能有些读者并不明白cv-qualifier-seq

       CV-qualifiers(CV限定符)

       CV-qualifiers有三种：const-qualifier(const限定符)、volatile-qualifier(volatile限定符)、以及const-volatile-qualifier(const-volatile限定符)。

       const类对象的非静态、非mutable、以及非引用数据成员是const-qualified；

       volatile类对象的非静态、非引用数据成员是volatile-qualified；

       const-volatile类对象的非静态、非引用数据成员是const-volatile-qualified。

       当CV-qualifiers用于限定数组类型时，实际上是数组成员被该CV-qualifiers限定，而非该数组类型。

       复合类型并不因其成员被CV-qualifier限定而被该CV-qualifier限定，也就是说，即使复合类型的成员有CV-qualifier限定，该复合类型也不是CV-qualified对象。

       ● 引用的权威定义：

In a declaration T D where D has the form

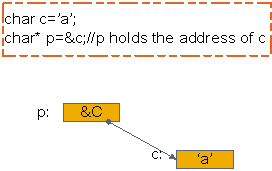
& D1

And the type of the identifier in the declaration T D1 is "derived-declarator-type-list T", then the type of the identifier of D is "derived-declarator-type-list cv-qualifier-seq reference to T". Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef or a template type argument, in which case the cv-qualifiers are ignored.

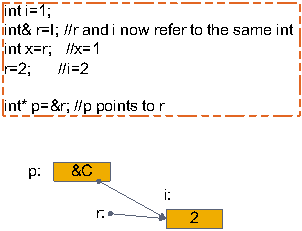
——摘自《ANSI C++ Standard》

       上面这些定义初看有些难懂，如果是这样的话，那说明你对[C++](http://www.jizhuomi.com/catalog.asp?tags=C%EF%BC%8B%EF%BC%8B)还不够熟悉，你还有很长的路要走。下面用通俗易懂的话来概述一下：

       ● 指针-对于一个类型T，T\*就是指向T的指针类型，也即一个T\*类型的变量能够保存一个T对象的地址，而类型T是可以加一些限定词的，如const、volatile等等。见下图，所示指针的含义：

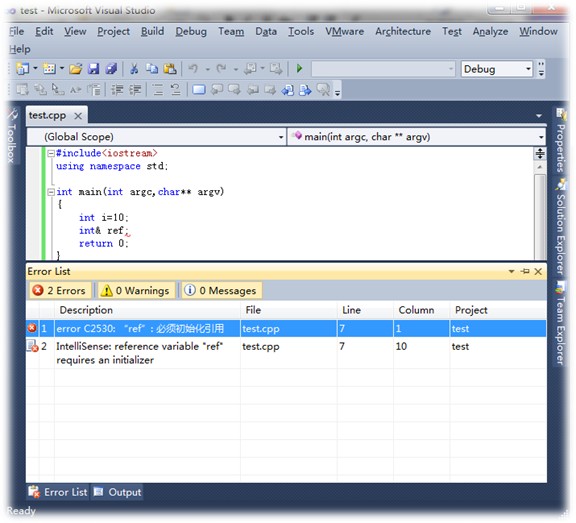


       ● 引用-引用是一个对象的别名，主要用于函数参数和返回值类型，符号X&表示X类型的引用。见下图，所示引用的含义：



       2、指针和引用的区别

       ● 首先，引用不可以为空，但指针可以为空。前面也说过了引用是对象的别名，引用为空——对象都不存在，怎么可能有别名！故定义一个引用的时候，必须初始化。因此如果你有一个变量是用于指向另一个对象，但是它可能为空，这时你应该使用指针；如果变量总是指向一个对象，i.e.，你的设计不允许变量为空，这时你应该使用引用。如下图中，如果定义一个引用变量，不初始化的话连编译都通不过（编译时错误）：



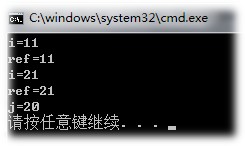
       ● 而声明指针是可以不指向任何对象，也正是因为这个原因，使用指针之前必须做判空操作，而引用就不必。

       ● 其次，引用不可以改变指向，对一个对象"至死不渝"；但是指针可以改变指向，而指向其它对象。说明：虽然引用不可以改变指向，但是可以改变初始化对象的内容。例如就++操作而言，对引用的操作直接反应到所指向的对象，而不是改变指向；而对指针的操作，会使指针指向下一个对象，而不是改变所指对象的内容。见下面的代码：

**C++代码**

1. **#include<iostream>**
2. **using** **namespace** std;
4. **int** main(**int** argc,**char**\*\* argv)
5. {
6. **int** i=10;
7. **int**& ref=i;
8. ref++;
10. cout<<"i="<<i<<endl;
11. cout<<"ref="<<ref<<endl;
13. **int** j=20;
14. ref=j;
15. ref++;
17. cout<<"i="<<i<<endl;
18. cout<<"ref="<<ref<<endl;
19. cout<<"j="<<j<<endl;
20. **return** 0;
21. }

       对ref的++操作是直接反应到所指变量之上，对引用变量ref重新赋值"ref=j"，并不会改变ref的指向，它仍然指向的是i，而不是j。理所当然，这时对ref进行++操作不会影响到j。而这些换做是指针的话，情况大不相同，请自行实验。输出结果如下：



       ● 再次，引用的大小是所指向的变量的大小，因为引用只是一个别名而已；指针是指针本身的大小，4个字节。见下图所示：

C++指针和引用的区别终极详解

       ● 从上面也可以看出：引用比指针使用起来形式上更漂亮，使用引用指向的内容时可以之间用引用变量名，而不像指针一样要使用\*；定义引用的时候也不用像指针一样使用&取址。

       ● 最后，引用比指针更安全。由于不存在空引用，并且引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用，因此引用很安全。对于指针来说，它可以随时指向别的对象，并且可以不被初始化，或为NULL，所以不安全。const 指针虽然不能改变指向，但仍然存在空指针，并且有可能产生野指针（即多个指针指向一块内存，free掉一个指针之后，别的指针就成了野指针）。

       总而言之，言而总之——它们的这些差别都可以归结为"指针指向一块内存，它的内容是所指内存的地址；而引用则是某块内存的别名，引用不改变指向。"

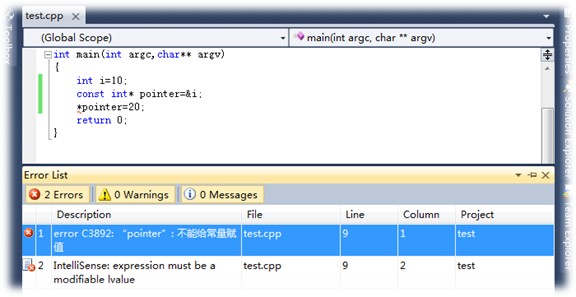
       3、特别之处const

       在这里我为什么要提到const关键字呢？因为const对指针和引用的限定是有差别的，下面听我一一到来。

       ● 常量指针VS常量引用

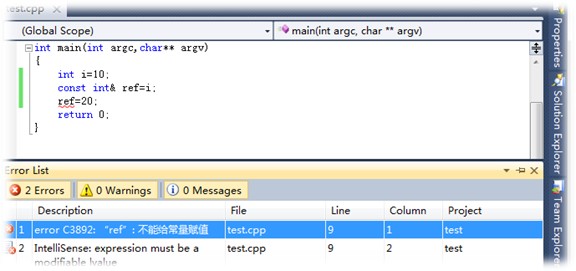
       常量指针：指向常量的指针，在指针定义语句的类型前加const，表示指向的对象是常量。

       定义指向常量的指针只限制指针的间接访问操作，而不能规定指针指向的值本身的操作规定性。



       常量指针定义"const int\* pointer=&a"告诉编译器，\*pointer是常量，不能将\*pointer作为左值进行操作。

       常量引用：指向常量的引用，在引用定义语句的类型前加const，表示指向的对象是常量。也跟指针一样不能利用引用对指向的变量进行重新赋值操作。



       ● 指针常量VS引用常量

       在指针定义语句的指针名前加const，表示指针本身是常量。在定义引用常量时必须初始化！而这是引用天生具来的属性，不用再引用指针定义语句的引用名前加const。

       指针常量定义"int\* const pointer=&b"告诉编译器，pointer是常量，不能作为左值进行操作，但是允许修改间接访问值，即\*pointer可以修改。

C++指针和引用的区别终极详解

       ● 常量指针常量VS常量引用常量

       常量指针常量：指向常量的指针常量，可以定义一个指向常量的指针常量，它必须在定义时初始化。常量指针常量定义"const int\* const pointer=&c"告诉编译器，pointer和\*pointer都是常量，他们都不能作为左值进行操作。

       而就不存在所谓的"常量引用常量"，因为跟上面讲的一样引用变量就是引用常量。C++不区分变量的const引用和const变量的引用。程序决不能给引用本身重新赋值，使他指向另一个变量，因此引用总是const的。如果对引用应用关键字const，起作用就是使其目标称为const变量。即没有：Const double const& a=1；只有const double& a=1；

       总结：有一个规则可以很好的区分const是修饰指针，还是修饰指针指向的数据——画一条垂直穿过指针声明的星号（\*），如果const出现在线的左边，指针指向的数据为常量；如果const出现在右边，指针本身为常量。而引用本身与天俱来就是常量，即不可以改变指向。

       4、指针和引用的实现

       我们利用下面一段简单的代码来深入分析指针和引用：

**C++代码**

1. **#include<iostream>**
2. **using** **namespace** std;
4. **int** main(**int** argc, **char**\*\* argv)
5. {
6. **int** i=1;
7. **int**& ref=i;
8. **int** x=ref;
10. cout<<"x is "<<x<<endl;
12. ref=2;
13. **int**\* p=&i;
15. cout<<"ref = "<<ref<<", i = "<<i<<endl;
16. }

       上面的代码用g++ test.c编译之后，然后反汇编objdump -d a.out，得到main函数的一段汇编代码如下：

**C++代码**

1. 08048714 <main>:
2. 8048714: 55　　　　push %ebp
3. 8048715: 89 e5　　　mov %esp,%ebp
4. 8048717: 83 e4 f0 and $0xfffffff0,%esp//为main函数的参数argc、argv保留位置
5. 804871a: 56 push %esi
6. 804871b: 53 push %ebx
7. 804871c: 83 ec 28 sub $0x28,%esp
8. 804871f: c7 44 24 1c 01 00 00 movl $0x1,0x1c(%esp) //将0x1存到esp寄存器中，即int i=1
9. 8048726: 00
10. 8048727: 8d 44 24 1c lea 0x1c(%esp),%eax// esp寄存器里的变量i的地址传给eax
11. 804872b: 89 44 24 18 mov %eax,0x18(%esp)//将寄存器eax中的内容（i的地址）传给寄存器中的变量ref，即int& ref=i
12. 804872f: 8b 44 24 18 mov 0x18(%esp),%eax//将寄存器esp中的ref传给eax，即i的地址
13. 8048733: 8b 00 mov (%eax),%eax//以寄存器eax中的值作为地址，取出值给eax 8048735: 89 44 24 14 mov %eax,0x14(%esp) //将寄存器eax中的值传给寄存器esp中的x，即x=ref
14. 8048739: c7 44 24 04 00 89 04 movl $0x8048900,0x4(%esp)
15. 8048740: 08
16. 8048741: c7 04 24 40 a0 04 08 movl $0x804a040,(%esp)
17. 8048748: e8 cb fe ff ff call 8048618 <\_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc@plt>
18. 804874d: 8b 54 24 14 mov 0x14(%esp),%edx
19. 8048751: 89 54 24 04 mov %edx,0x4(%esp)
20. 8048755: 89 04 24 mov %eax,(%esp)
21. 8048758: e8 5b fe ff ff call 80485b8 <\_ZNSolsEi@plt>
22. 804875d: c7 44 24 04 38 86 04 movl $0x8048638,0x4(%esp)
23. 8048764: 08
24. 8048765: 89 04 24 mov %eax,(%esp)
25. 8048768: e8 bb fe ff ff call 8048628 <\_ZNSolsEPFRSoS\_E@plt>//从8048739~8048768这些行就是执行"cout<<"x is "<<x<<endl;"
26. 804876d: 8b 44 24 18 mov 0x18(%esp),%eax//将寄存器esp中的ref传到eax中
27. 8048771: c7 00 02 00 00 00 movl $0x2,(%eax) //将0x2存到eax寄存器中
28. 8048777: 8d 44 24 1c lea 0x1c(%esp),%eax// esp寄存器里的变量i的地址传给eax
29. 804877b: 89 44 24 10 mov %eax,0x10(%esp) //将寄存器eax中的内容（即i的地址）传到寄存器esp中的p
30. 804877f: 8b 5c 24 1c mov 0x1c(%esp),%ebx
31. 8048783: 8b 44 24 18 mov 0x18(%esp),%eax
32. 8048787: 8b 30 mov (%eax),%esi
33. 8048789: c7 44 24 04 06 89 04 movl $0x8048906,0x4(%esp)
34. 8048790: 08
35. 8048791: c7 04 24 40 a0 04 08 movl $0x804a040,(%esp)
36. 8048798: e8 7b fe ff ff call 8048618 <\_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc@plt>
37. 804879d: 89 74 24 04 mov %esi,0x4(%esp)
38. 80487a1: 89 04 24 mov %eax,(%esp)
39. 80487a4: e8 0f fe ff ff call 80485b8 <\_ZNSolsEi@plt>
40. 80487a9: c7 44 24 04 0d 89 04 movl $0x804890d,0x4(%esp)
41. 80487b0: 08
42. 80487b1: 89 04 24 mov %eax,(%esp)
43. 80487b4: e8 5f fe ff ff call 8048618 <\_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc@plt>
44. 80487b9: 89 5c 24 04 mov %ebx,0x4(%esp)
45. 80487bd: 89 04 24 mov %eax,(%esp)
46. 80487c0: e8 f3 fd ff ff call 80485b8 <\_ZNSolsEi@plt>
47. 80487c5: c7 44 24 04 38 86 04 movl $0x8048638,0x4(%esp)
48. 80487cc: 08
49. 80487cd: 89 04 24 mov %eax,(%esp)
50. 80487d0: e8 53 fe ff ff call 8048628 <\_ZNSolsEPFRSoS\_E@plt>//这些行就是执行"cout<<"ref = "<<ref<<", i = "<<i<<endl;"
51. 80487d5: b8 00 00 00 00 mov $0x0,%eax
52. 80487da: 83 c4 28 add $0x28,%esp
53. 80487dd: 5b pop %ebx
54. 80487de: 5e pop %esi
55. 80487df: 89 ec mov %ebp,%esp
56. 80487e1: 5d pop %ebp
57. 80487e2: c3 ret

       从汇编代码可以看出实际上指针和引用在编译器中的实现是一样的：

       ● 引用 int& ref=i;

       8048727: 8d 44 24 1c lea 0x1c(%esp),%eax// esp寄存器里的变量i的地址传给eax

       804872b: 89 44 24 18 mov %eax,0x18(%esp)//将寄存器eax中的内容（i的地址）传给寄存器中的变量ref，即int& ref=i

       ● 指针 int\* p=&i;

       8048777: 8d 44 24 1c lea 0x1c(%esp),%eax// esp寄存器里的变量i的地址传给eax

       804877b: 89 44 24 10 mov %eax,0x10(%esp) //将寄存器eax中的内容（即i的地址）传到寄存器esp中的p

       虽然指针和引用最终在编译中的实现是一样的，但是引用的形式大大方便了使用也更安全。有人说："引用只是一个别名，不会占内存空间？"通过这个事实我们可以揭穿这个谎言！实际上引用也是占内存空间的。

       5、指针传递和引用传递

       为了更好的理解指针和引用，我们下面来介绍一下指针传递和引用传递。当指针和引用作为函数的函数是如何传值的呢？（下面这一段引用了C++中引用传递与指针传递区别（进一步整理））

       ● 指针传递参数本质上是值传递的方式，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，即在栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。

       ● 引用传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

       引用传递和指针传递是不同的，虽然它们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针地址，它将影响不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量，那就得使用指向指针的指针，或者指针引用。

# [C++ 各种关键字](https://www.cnblogs.com/schips/p/12309999.html)

## 命名空间 namespace

源地址：<https://baijiahao.baidu.com/s?id=1662580430712018597&wfr=spider&for=pc>

编写程序过程中，名称（name）可以是符号常量、变量、函数、结构、枚举、类和对象等等。工程越大，名称互相冲突性的可能性越大。另外使用多个厂商的类库时，也可能导致名称冲突。为了避免，在大规模程序的设计中，以及在程序员使用各种各样的 C++ 库时，这些标识符的命名发生冲突，标准 C++ 引入关键字 namespace（命名空间/名字空间/名称空间），可以更好地控制标识符的作用域。

例如，我们在 C 语言中，通过 static 可以限制名字只在当前编译单元内可见，在 C++ 中我们通过 namespace 来控制对名字的访问。

##### 1. namespace 语法

1. namespace 中可定义常量、变量、函数、结构体、枚举、类等

2. namespace 只能在全局定义。

3. namespace 支持嵌套定义。

4. namespace 是开放的，可随时添加新的成员。

5. namespace 关键字可以为已有空间名字增加别名

6. 无名命名空间意味着命名空间中的符号只能在本文件中访问，相当于给符号增加了 static 修饰。

1.1 namespace 可包含的成员



1.2 namespace 只能在全局定义



1.3 namespace 支持嵌套定义

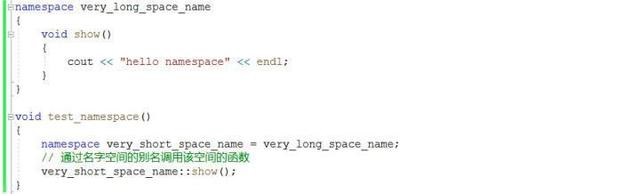


名字空间 my\_space 中可以嵌套定义子名字空间 my\_sub\_space.

1.4 namespace 是开放的，随时可添加成员



1.5 namespace 为已有空间名字创建别名



1.6 namespace 匿名名字空间

无名命名空间意味着命名空间中的符号只能在本文件中访问，相当于给符号增加了 static 修饰，只能在当前文件内访问。



##### 2. 使用 namespace 中符号

使用名字空间中定义的符号有三种方式：

1. 直接通过 namespace 作用域访问

2. using 声明指定某个符号在某个作用域下可见

3. using 编译指令指定名字空间中所有符号在在某个作用域下可见

2.1 直接访问



2.2 using 声明

using 声明指定某个符号在某个作用域下可见。例如：



2.3 using 编译指令

using 编译指令指定名字空间中所有符号在在某个作用域下可见。



## This指针

http://m.blog.chinaunix.net/uid-21411227-id-1826942.html

##### 1. this指针的用处:

　　一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。 　　例如，调用date.SetMonth(9) <===> SetMonth(&date, 9)，this帮助完成了这一转换 .

##### 2. this指针的使用:

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return \*this；另外一种情况是当参数与成员变量名相同时，如this->n = n （不能写成n = n）。

##### 3. this指针程序示例:

this指针存在于类的成员函数中,指向被调用函数所在的类实例的地址。 　　根据以下程序来说明this指针

#include

class Point 　　{ 　　int x, y;

public:

Point(int a, int b) { x=a; y=b;}

void MovePoint( int a, int b){ x+=a; y+=b;}

void print(){ cout<<"x="<<x<<"y="<<y<<=""></x<<"y="<<y<

};

void main( ) 　　{

Point point1( 10,10);

point1.MovePoint(2,2);

point1.print( );

}

当对象point1调用MovePoint(2,2)函数时，即将point1对象的地址传递给了this指针。

MovePoint函数的原型应该是 void MovePoint( Point \*this, int a, int b);第一个参数是指向该类对象的一个指针，我们在定义成员函数时没看见是因为这个参数在类中是隐含的。这样point1的地址传递给了this，所以在MovePoint函数中便显式的写成：

void MovePoint(int a, int b) { this->x +=a; this-> y+= b;} 　　即可以知道，point1调用该函数后，也就是point1的数据成员被调用并更新了值。 　　即该函数过程可写成 point1.x+= a; point1. y + = b;

##### 4. 关于this指针的一个经典回答:

当你进入一个房子后，

你可以看见桌子、椅子、地板等，

但是房子你是看不到全貌了。

对于一个类的实例来说，

你可以看到它的成员函数、成员变量，

但是实例本身呢？

this是一个指针，它时时刻刻指向你这个实例本身

##### 5. 类的this指针有以下特点：

###### （1）this只能在成员函数中使用。

全局函数、静态函数都不能使用this.

实际上，成员函数默认第一个参数为T \* const this。

如：

class A

{

public:

int func(int p)

{

}

};

**其中，func的原型在编译器看来应该是：**

**int func(A \* const this,int p);**

###### （2）由此可见，this在成员函数的开始前构造，在成员函数的结束后清除。

这个生命周期同任何一个函数的参数是一样的，没有任何区别。

当调用一个类的成员函数时，编译器将类的指针作为函数的this参数传递进去。如：

A a;

a.func(10);

此处，编译器将会编译成：

A::func(&a,10);

看起来和静态函数没差别，对吗？**不过，区别还是有的。编译器通常会对this指针做一些优化，**因此，this指针的传递效率比较高--如VC通常是通过ecx寄存器传递this参数的。

###### （3）几个this指针的易混问题。

##### A. this指针是什么时候创建的？

this在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果class或者struct里面没有方法的话，它们是没有构造函数的，只能当做C的struct使用。采用 TYPE xx的方式定义的话，在栈里分配内存，这时候this指针的值就是这块内存的地址。采用new的方式 创建对象的话，在堆里分配内存，new操作符通过eax返回分配 的地址，然后设置给指针变量。之后去调 用构造函数（如果有构造函数的话），这时将这个内存块的地址传给ecx，之后构造函数里面怎么处理请 看上面的回答。

##### B. this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。在汇编级 别里面，一个值只会以3种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在内 存中，它们并不是和高级语言变量对应的。

##### C. this指针是如何传递类中的函数的？绑定？还是在函数参数的首参数就是this指针？那么，this指针 又是如何找到“类实例后函数的”？

大多数编译器通过ecx寄存器传递this指针。事实上，这也是一个潜规则。一般来说，不同编译器都会遵从一致的传参规则，否则不同编译器产生的obj就无法匹配了。

在call之前，编译器会把对应的对象地址放到eax中。this是通过函数参数的首参来传递的。this指针在调用之前生成，至于“类实例后函数”，没有这个说法。类在实例化时，只分配类中的变量空间，并没有为函数分配空间。自从类的函数定义完成后，它就在那儿，不会跑的。

##### D. this指针是如何访问类中的变量的？

如果不是类，而是结构体的话，那么，如何通过结构指针来访问结构中的变量呢？如果你明白这一点的话，就很容易理解这个问题了。

**在C++中 ,类和结构是只有一个区别的：类的成员默认是private，而结构是public。**

**this是类的指针，如果换成结构，那this就是结构的指针了。**

##### E. 我们只有获得一个对象后，才能通过对象使用this指针。

如果我们知道一个对象this指针的位置，可以直接使用吗？

this指针只有在成员函数中才有定义。因此，你获得一个对象后，也不能通过对象使用this指针。所以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

##### F. 每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？

普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。只有虚函数才会被放到函数表中。但是，即使是虚函数，如果编译器能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用，而是会直接调用该函数。

## Const

#### 文章1：

<https://blog.csdn.net/Eric_Jo/article/details/4138548?utm_medium=distribute.pc_relevant_right.none-task-blog-BlogCommendFromBaidu-5.channel_param_right&depth_1-utm_source=distribute.pc_relevant_right.none-task-blog-BlogCommendFromBaidu-5.channel_param_right>

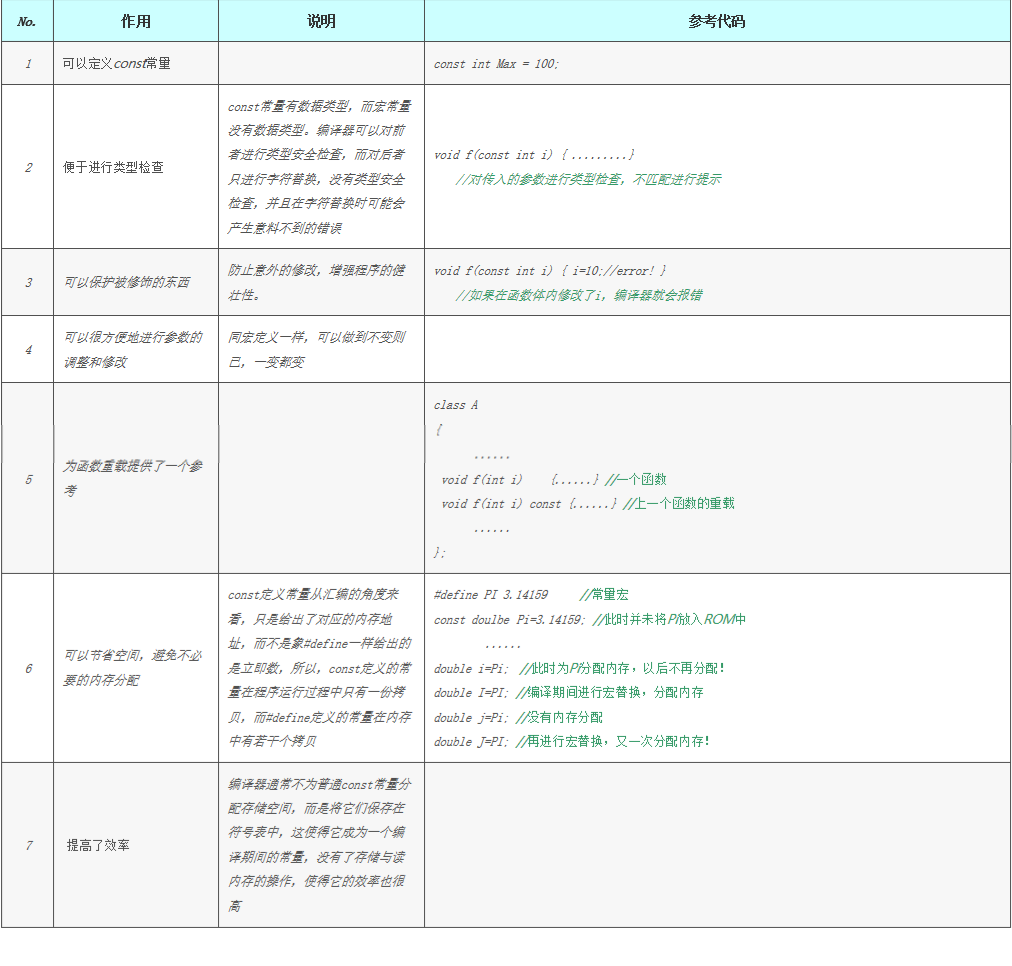
C++中的const关键字的用法非常灵活，而使用const将大大改善程序的健壮性，本人根据各方面查到的资料进行总结如下，期望对朋友们有所帮助。

Const

是C++中常用的类型修饰符,常类型是指使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的。

###### 一、Const作用

如下表所示：



###### 二、Const的使用

1、定义常量(1)const修饰变量，以下两种定义形式在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。

TYPE const ValueName = value;

const TYPE ValueName = value;

(2)将const改为外部连接,作用于扩大至全局,编译时会分配内存,并且可以不进行初始化,仅仅作为声明,编译器认为在程序其他地方进行了定义.

extend const int ValueName = value;

2、指针使用CONST(1)指针本身是常量不可变 char\* const pContent;

(2)指针所指向的内容是常量不可变 const char \*pContent;

(3)两者都不可变 const char\* const pContent;

(4)还有其中区别方法，沿着\*号划一条线：如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

###### 3、函数中使用CONST

(1)const修饰函数参数a.传递过来的参数在函数内不可以改变(无意义，因为Var本身就是形参)

void function(const int Var);

b.参数指针所指内容为常量不可变

void function(const char\* Var);

c.参数指针本身为常量不可变(也无意义，因为char\*

Var也是形参)

void function(char\* const Var);

d.参数为引用，为了增加效率同时防止修改。修饰引用参数时：

void function(const Class& Var);

//引用参数在函数内不可以改变

void function(const TYPE& Var);

//引用参数在函数内为常量不可变

这样的一个const引用传递和最普通的函数按值传递的效果是一模一样的,他禁止对引用的对象的一切修改,唯一不同的是按值传递会先建立一个类对象的副本,

然后传递过去,而它直接传递地址,所以这种传递比按值传递更有效.另外只有引用的const传递可以传递一个临时对象,因为临时对象都是const属性,

且是不可见的,他短时间存在一个局部域中,所以不能使用指针,只有引用的const传递能够捕捉到这个家伙.

(2)const修饰函数返回值

const修饰函数返回值其实用的并不是很多，它的含义和const修饰普通变量以及指针的含义基本相同。 a.const int fun1() //这个其实无意义，因为参数返回本身就是赋值。 b. const int \* fun2() //调用时

const int \*pValue = fun2();

//我们可以把fun2()看作成一个变量，即指针内容不可变。 c.int\* const fun3() //调用时

int \* const pValue = fun2();

//我们可以把fun2()看作成一个变量，即指针本身不可变。

一般情况下，函数的返回值为某个对象时，如果将其声明为const时，多用于操作符的重载。通常，不建议用const修饰函数的返回值类型为某个对象或对某个对象引用的情况。原因如下：如果返回值为某个对象为const（const

A test = A 实例）或某个对象的引用为const（const A& test = A实例） ，则返回值具有const属性，则返回实例只能访问类A中的公有（保护）数据成员和const成员函数，并且不允许对其进行赋值操作，这在一般情况下很少用到。

4、类相关CONST

(1)const修饰成员变量

const修饰类的成员函数，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。

class A {

…

const int nValue; //成员常量不能被修改 …

A(int x): nValue(x) { } ; //只能在初始化列表中赋值 }

(2)const修饰成员函数

const修饰类的成员函数，则该成员函数不能修改类中任何非const成员函数。一般写在函数的最后来修饰。 class A {

… void function()const; //常成员函数,

它不改变对象的成员变量.

//也不能调用类中任何非const成员函数。}

对于const类对象/指针/引用， 只能调用类的const成员函数，因此，const修饰成员函数的最重要作用就是限制对于const对象的使用。

a. const成员函数不被允许修改它所在对象的任何一个数据成员。

b. const成员函数能够访问对象的const成员，而其他成员函数不可以。

(3)const修饰类对象/对象指针/对象引用

· const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。

· const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。例如：

class AAA

{

void func1();

void func2() const;

}

const AAA aObj;

aObj.func1(); ×

aObj.func2(); 正确

const AAA\* aObj = new AAA();

aObj-> func1(); ×

aObj-> func2(); 正确

三、将Const类型转化为非Const类型的方法

采用const\_cast

进行转换。

用法：const\_cast <type\_id> (expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。

· 常量指针被转化成非常量指针，并且仍然指向原来的对象；

· 常量引用被转换成非常量引用，并且仍然指向原来的对象；

· 常量对象被转换成非常量对象。

四、使用const的一些建议

· 要大胆的使用const，这将给你带来无尽的益处，但前提是你必须搞清楚原委；

· 要避免最一般的赋值操作错误，如将const变量赋值，具体可见思考题；

· 在参数中使用const应该使用引用或指针，而不是一般的对象实例，原因同上；

· const在成员函数中的三种用法（参数、返回值、函数）要很好的使用；

· 不要轻易的将函数的返回值类型定为const;

· 除了重载操作符外一般不要将返回值类型定为对某个对象的const引用;

· 任何不会修改数据成员的函数都应该声明为const

类型。

五、补充重要说明

· 类内部的常量限制：使用这种类内部的初始化语法的时候，常量必须是被一个常量表达式

初始化的整型或枚举类型，而且必须是static和const形式。

· 如何初始化类内部的常量：一种方法就是static

和 const

并用，在外部初始化，例如：

class A { public: A() {} private: static const int i; file://注意必须是静态的！

}；

const int A::i=3;另一个很常见的方法就是初始化列表： class A { public: A(int

i=0):test(i) {} private: const int i; }； 还有一种方式就是在外部初始化，

· 如果在非const成员函数中，this指针只是一个类类型的；如果在const成员函数中，

this指针是一个const类类型的；如果在volatile成员函数中,this指针就是一个

volatile类类型的。

· new返回的指针必须是const类型的。

#### 文章2：

原文链接：https://blog.csdn.net/hustli/article/details/19342

1、什么是const? 常类型是指使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的。（当然，我们可以偷梁换柱进行更新：）

2、为什么引入const？　 const 推出的初始目的，正是为了取代预编译指令，消除它的缺点，同时继承它的优点。

3、cons有什么主要的作用？ （1）可以定义const常量，具有不可变性。

例如：

const int Max=100;

int Array[Max];

（2）便于进行类型检查，使编译器对处理内容有更多了解，消除了一些隐患。 例如：

void f(const int i) { .........}

编译器就会知道i是一个常量，不允许修改； （3）可以避免意义模糊的数字出现，同样可以很方便地进行参数的调整和修改。

同宏定义一样，可以做到不变则已，一变都变！如（1）中，如果想修改Max的内容，只需要：const int Max=you want;即可！ （4）可以保护被修饰的东西，防止意外的修改，增强程序的健壮性。

还是上面的例子，如果在函数体内修改了i，编译器就会报错；

例如：

void f(const int i) { i=10;//error! } (5) 为函数重载提供了一个参考。

class A

{

......

void f(int i) {......} file://一个函数

void f(int i) const {......} file://上一个函数的重载

......

}; (6) 可以节省空间，避免不必要的内存分配。

例如：

#define PI 3.14159

file://常量宏

const doulbe Pi=3.14159; file://此时并未将Pi放入ROM中

......

double i=Pi;

file://此时为Pi分配内存，以后不再分配！

double I=PI;

file://编译期间进行宏替换，分配内存

double j=Pi;

file://没有内存分配

double J=PI;

file://再进行宏替换，又一次分配内存！

const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝。 （7） 提高了效率。

编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。

3、如何使用const？

（1）修饰一般常量

一般常量是指简单类型的常量。这种常量在定义时，修饰符const可以用在类型说明符前，也可以用在类型说明符后。

例如：

int const x=2;　　或　　const int x=2;

（2）修饰常数组

定义或说明一个常数组可采用如下格式：

int const a[5]={1, 2, 3, 4, 5};

const int a[5]={1, 2, 3, 4, 5};

（3）修饰常对象

常对象是指对象常量，定义格式如下：

class A;

const A a;

A const a;

定义常对象时，同样要进行初始化，并且该对象不能再被更新，修饰符const可以放在类名后面，也可以放在类名前面。

（4）修饰常指针

const int \*A; file://const修饰指向的对象，A可变，A指向的对象不可变

int const \*A; 　 file://const修饰指向的对象，A可变，A指向的对象不可变

int \*const A; 　 file://const修饰指针A，A不可变，A指向的对象可变

const int \*const A; file://指针A和A指向的对象都不可变

（5）修饰常引用

使用const修饰符也可以说明引用，被说明的引用为常引用，该引用所引用的对象不能被更新。其定义格式如下：

const double & v;

（6）修饰函数的常参数

const修饰符也可以修饰函数的传递参数，格式如下：

void Fun(const int Var);

告诉编译器Var在函数体中的无法改变，从而防止了使用者的一些无意的或错误的修改。

（7）修饰函数的返回值：

const修饰符也可以修饰函数的返回值，是返回值不可被改变，格式如下：

const int Fun1();

const MyClass Fun2();

（8）修饰类的成员函数：

const修饰符也可以修饰类的成员函数，格式如下：

class ClassName {

public:

　 　int Fun() const;

　 .....

}；

这样，在调用函数Fun时就不能修改类里面的数据

（9）在另一连接文件中引用const常量

extern const int i; file://正确的引用

extern const int j=10; file://错误！常量不可以被再次赋值

另外，还要注意，常量必须初始化！

例如：

const int i=5;

4、几点值得讨论的地方：

（1）const究竟意味着什么？

说了这么多，你认为const意味着什么？一种修饰符？接口抽象？一种新类型？

也许都是，在Stroustup最初引入这个关键字时，只是为对象放入ROM做出了一种可能，对于const对象，C++既允许对其进行静态初始化，也允许对他进行动态初始化。理想的const对象应该在其构造函数完成之前都是可写的，在析构函数执行开始后也都是可写的，换句话说，const对象具有从构造函数完成到析构函数执行之前的不变性，如果违反了这条规则，结果都是未定义的！虽然我们把const放入ROM中，但这并不能够保证const的任何形式的堕落，我们后面会给出具体的办法。无论const对象被放入ROM中，还是通过存储保护机制加以保护，都只能保证，对于用户而言这个对象没有改变。换句话说，废料收集器（我们以后会详细讨论，这就一笔带过）或数据库系统对一个const的修改没有任何问题。

（2）位元const V.S. 抽象const?

对于关键字const的解释有好几种方式，最常见的就是位元const 和 抽象const。下面我们看一个例子：

class A

{

public:

......

A f(const A& a);

......

};

如果采用抽象const进行解释，那就是f函数不会去改变所引用对象的抽象值，如果采用位元const进行解释，那就成了f函数不会去改变所引用对象的任何位元。

我们可以看到位元解释正是c++对const问题的定义，const成员函数不被允许修改它所在对象的任何一个数据成员。

为什么这样呢？因为使用位元const有2个好处：

最大的好处是可以很容易地检测到违反位元const规定的事件：编译器只用去寻找有没有对数据成员的赋值就可以了。另外，如果我们采用了位元const，那么，对于一些比较简单的const对象，我们就可以把它安全的放入ROM中，对于一些程序而言，这无疑是一个很重要的优化方式。（关于优化处理，我们到时候专门进行讨论）

当然，位元const也有缺点，要不然，抽象const也就没有产生的必要了。

首先，位元const的抽象性比抽象const的级别更低！实际上，大家都知道，一个库接口的抽象性级别越低，使用这个库就越困难。

其次，使用位元const的库接口会暴露库的一些实现细节，而这往往会带来一些负面效应。所以，在库接口和程序实现细节上，我们都应该采用抽象const。

有时，我们可能希望对const做出一些其它的解释，那么，就要注意了，目前，大多数对const的解释都是类型不安全的，这里我们就不举例子了，你可以自己考虑一下，总之，我们尽量避免对const的重新解释。

（3）放在类内部的常量有什么限制？

看看下面这个例子：

class A

{

private:

const int c3 = 7;

// ??? static int c4 = 7;

// ??? static const float c5 = 7; // ???

...... };

你认为上面的3句对吗？呵呵，都不对！使用这种类内部的初始化语法的时候，常量必须是被一个常量表达式初始化的整型或枚举类型，而且必须是static和const形式。这显然是一个很严重的限制！

那么，我们的标准委员会为什么做这样的规定呢？一般来说，类在一个头文件中被声明，而头文件被包含到许多互相调用的单元去。但是，为了避免复杂的编译器规则，C++要求每一个对象只有一个单独的定义。如果C++允许在类内部定义一个和对象一样占据内存的实体的话，这种规则就被破坏了。 （4）如何初始化类内部的常量？

一种方法就是static 和 const 并用，在内部初始化，如上面的例子；

另一个很常见的方法就是初始化列表：

class A

{

public:

A(int i=0):test(i) {}

private:

const int i;

}；

还有一种方式就是在外部初始化，例如：

class A

{

public:

A() {}

private:

static const int i; file://注意必须是静态的！

}；

const int A::i=3;

（5）常量与数组的组合有什么特殊吗？

我们给出下面的代码：

const int size[3]={10,20,50};

int array[size[2]];

有什么问题吗？对了，编译通不过！为什么呢？

const可以用于集合，但编译器不能把一个集合存放在它的符号表里，所以必须分配内存。在这种情况下，const意味着“不能改变的一块存储”。然而，其值在编译时不能被使用，因为编译器在编译时不需要知道存储的内容。自然，作为数组的大小就不行了：）

你再看看下面的例子：

class A

{

public:

A(int i=0):test[2]({1,2}) {} file://你认为行吗？

private:

const int test[2];

}；

vc6下编译通不过，为什么呢？

关于这个问题，前些时间，njboy问我是怎么回事？我反问他：“你认为呢？”他想了想，给出了一下解释，大家可以看看：我们知道编译器堆初始化列表的操作是在构造函数之内，显式调用可用代码之前，初始化的次序依据数据声明的次序。初始化时机应该没有什么问题，那么就只有是编译器对数组做了什么手脚！其实做什么手脚，我也不知道，我只好对他进行猜测：编译器搜索到test发现是一个非静态的数组，于是，为他分配内存空间，这里需要注意了，它应该是一下分配完，并非先分配test[0],然后利用初始化列表初始化，再分配test[1],这就导致数组的初始化实际上是赋值！然而，常量不允许赋值，所以无法通过。

呵呵，看了这一段冠冕堂皇的话，真让我笑死了！njboy别怪我揭你短呀：）我对此的解释是这样的：C++标准有一个规定，不允许无序对象在类内部初始化，数组显然是一个无序的，所以这样的初始化是错误的！对于他，只能在类的外部进行初始化，如果想让它通过，只需要声明为静态的，然后初始化。

这里我们看到，常量与数组的组合没有什么特殊！一切都是数组惹的祸！ （6）this指针是不是const类型的？

this指针是一个很重要的概念，那该如何理解她呢？也许这个话题太大了，那我们缩小一些：this指针是个什么类型的？这要看具体情况：如果在非const成员函数中，this指针只是一个类类型的；如果在const成员函数中，this指针是一个const类类型的；如果在volatile成员函数中,this指针就是一个volatile类类型的。

（7）const到底是不是一个重载的参考对象？

先看一下下面的例子：

class A

{

......

void f(int i) {......} file://一个函数

void f(int i) const {......} file://上一个函数的重载

......

};

上面是重载是没有问题的了，那么下面的呢？

class A

{

......

void f(int i) {......} file://一个函数

void f(const int i) {......} file://？？？？？

......

};

这个是错误的，编译通不过。那么是不是说明内部参数的const不予重载呢？再看下面的例子：

class A

{

......

void f(int& ) {......} file://一个函数

void f(const int& ) {......} file://？？？？？

......

};

这个程序是正确的，看来上面的结论是错误的。为什么会这样呢？这要涉及到接口的透明度问题。按值传递时，对用户而言，这是透明的，用户不知道函数对形参做了什么手脚，在这种情况下进行重载是没有意义的，所以规定不能重载！当指针或引用被引入时，用户就会对函数的操作有了一定的了解，不再是透明的了，这时重载是有意义的，所以规定可以重载。

（8）什么情况下为const分配内存？

以下是我想到的可能情况，当然，有的编译器进行了优化，可能不分配内存。

A、作为非静态的类成员时；

B、用于集合时；

C、被取地址时；

D、在main函数体内部通过函数来获得值时；

E、const的 class或struct有用户定义的构造函数、析构函数或基类时；。

F、当const的长度比计算机字长还长时；

G、参数中的const；

H、使用了extern时。

不知道还有没有其他情况，欢迎高手指点：）

（9）临时变量到底是不是常量？

很多情况下，编译器必须建立临时对象。像其他任何对象一样，它们需要存储空间而且必须被构造和删除。区别是我们从来看不到编译器负责决定它们的去留以及它们存在的细节。对于C++标准草案而言：临时对象自动地成为常量。因为我们通常接触不到临时对象，不能使用与之相关的信息，所以告诉临时对象做一些改变有可能会出错。当然，这与编译器有关，例如：vc6、vc7都对此作了扩展，所以，用临时对象做左值，编译器并没有报错。

（10）与static搭配会不会有问题？

假设有一个类：

class A

{

public:

......

static void f() const { ......}

......

};

我们发现编译器会报错，因为在这种情况下static不能够与const共存！

为什么呢？因为static没有this指针，但是const修饰this指针，所以... （11）如何修改常量？

有时候我们却不得不对类内的数据进行修改，但是我们的接口却被声明了const，那该怎么处理呢？我对这个问题的看法如下：

1）标准用法：mutable

class A

{

public:

A(int i=0):test(i)

{ }

void SetValue(int i)const { test=i; }

private:

mutable int test; file://这里处理！

}；

2）强制转换：const\_cast

class A

{

public:

A(int i=0):test(i)

{ }

void SetValue(int i)const

{ const\_cast <int>(test)=i; }//这里处理！

private:

int test;

}；

3）灵活的指针：int\*

class A

{

public:

A(int i=0):test(i)

{ }

void SetValue(int i)const

{ \*test=i; }

private:

int\* test; file://这里处理！

}；

4）未定义的处理

class A

{

public:

A(int i=0):test(i)

{ }

void SetValue(int i)const

{ int \*p=(int\*)&test; \*p=i; }//这里处理！

private:

int test;

}；

注意，这里虽然说可以这样修改，但结果是未定义的，避免使用！

5）内部处理：this指针

class A

{

public:

A(int i=0):test(i)

{ }

void SetValue(int i)const

{ ((A\*)this)->test=i; }//这里处理！

private:

int test;

}；

6）最另类的处理：空间布局

class A

{

public:

A(int i=0):test(i),c('a') { }

private:

char c;

const int test;

};

int main()

{

A a(3);

A\* pa=&a;

char\* p=(char\*)pa;

int\* pi=(int\*)(p+4）；//利用边缘调整

\*pi=5;

file://此处改变了test的值！

return 0;

}

虽然我给出了6中方法，但是我只是想说明如何更改，但出了第一种用法之外，另外5种用法，我们并不提倡，不要因为我这么写了，你就这么用，否则，我真是要误人子弟了：）

（12）最后我们来讨论一下常量对象的动态创建。

既然编译器可以动态初始化常量，就自然可以动态创建，例如：

const int\* pi=new const int(10);

这里要注意2点：

1）const对象必须被初始化！所以(10)是不能够少的。

2）new返回的指针必须是const类型的。

那么我们可不可以动态创建一个数组呢？

答案是否定的，因为new内置类型的数组，不能被初始化。

这里我们忽视了数组是类类型的，同样对于类内部数组初始化我们也做出了这样的忽视，因为这涉及到数组的问题，我们以后再讨论。

## override 关键字的使用

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：
2. 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。
3. 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* void DKnuth() override; *//无法通过编译，常量性不一致，并非重载*
22. void Print() override; *//无法通过编译，非虚函数重载*
23. };

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; virtual void vmendd() override;
6. virtual void test(int x) override;
7. virtual void splle() override;
8. };

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
2. 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)); }

}

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只适合函数体内代码简单的函数使用，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

}

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);

}

// 定义文件

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

### 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){};

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。

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。

## explicit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

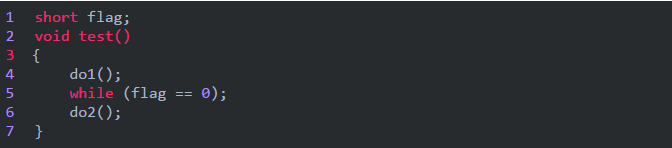
f3

## volatile关键字详解 C/C++

#### 博客1：https://blog.csdn.net/weixin\_44363885/article/details/92838607

##### 一、volatile介绍

volatile提醒编译器它后面所定义的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有volatile关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。下面举例说明。在DSP开发中，经常需要等待某个事件的触发，所以经常会写出这样的程序：

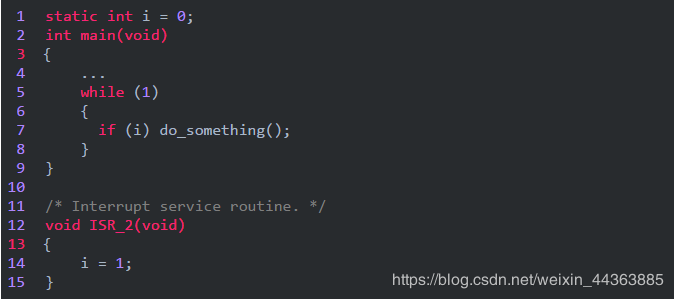


这段程序等待内存变量flag的值变为1(怀疑此处是0,有点疑问,)之后才运行do2()。变量flag的值由别的程序更改，这个程序可能是某个硬件中断服务程序。例如：如果某个按钮按下的话，就会对DSP产生中断，在按键中断程序中修改flag为1，这样上面的程序就能够得以继续运行。但是，编译器并不知道flag的值会被别的程序修改，因此在它进行优化的时候，可能会把flag的值先读入某个寄存器，然后等待那个寄存器变为1。如果不幸进行了这样的优化，那么while循环就变成了死循环，因为寄存器的内容不可能被中断服务程序修改。为了让程序每次都读取真正flag变量的值，就需要定义为如下形式：

在这里插入图片描述

需要注意的是，没有volatile也可能能正常运行，但是可能修改了编译器的优化级别之后就又不能正常运行了。因此经常会出现debug版本正常，但是release版本却不能正常的问题。所以为了安全起见，只要是等待别的程序修改某个变量的话，就加上volatile关键字。

volatile的本意是“易变的”,由于访问寄存器的速度要快过RAM，所以编译器一般都会作减少存取外部RAM的优化。比如：



程序的本意是希望ISR\_2中断产生时，在main当中调用do\_something函数，但是，由于编译器判断在main函数里面没有修改过i，因此可能只执行一次对从i到某寄存器的读操作，然后每次if判断都只使用这个寄存器里面的“i副本”，导致do\_something永远也不会被调用。如果变量加上volatile修饰，则编译器保证对此变量的读写操作都不会被优化（肯定执行）。此例中i也应该如此说明。

一般说来，volatile用在如下的几个地方：

1、中断服务程序中修改的供其它程序检测的变量需要加volatile；

2、多任务环境下各任务间共享的标志应该加volatile；

3、存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能由不同意义；

另外，以上这几种情况经常还要同时考虑数据的完整性（相互关联的几个标志读了一半被打断了重写），在1中可以通过关中断来实现，2中可以禁止任务调度，3中则只能依靠硬件的良好设计了。

##### ****二、volatile 的含义****

volatile总是与优化有关，编译器有一种技术叫做数据流分析，分析程序中的变量在哪里赋值、在哪里使用、在哪里失效，分析结果可以用于常量合并，常量传播等优化，进一步可以死代码消除。但有时这些优化不是程序所需要的，这时可以用volatile关键字禁止做这些优化，volatile的字面含义是易变的，它有下面的作用：

**1** 不会在两个操作之间把volatile变量缓存在寄存器中。在多任务、中断、甚至setjmp环境下，变量可能被其他的程序改变，编译器自己无法知道，volatile就是告诉编译器这种情况。

**2** 不做常量合并、常量传播等优化，所以像下面的代码：

在这里插入图片描述

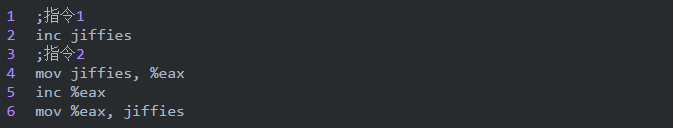
if的条件不会当作无条件真。

**3** 对volatile变量的读写不会被优化掉。如果你对一个变量赋值但后面没用到，编译器常常可以省略那个赋值操作，然而对Memory Mapped IO的处理是不能这样优化的。

前面有人说volatile可以保证对内存操作的原子性，这种说法不大准确，其一，x86需要LOCK前缀才能在SMP下保证原子性，其二，RISC根本不能对内存直接运算，要保证原子性得用别的方法，如atomic\_inc。

对于jiffies，它已经声明为volatile变量，我认为直接用jiffies++就可以了，没必要用那种复杂的形式，因为那样也不能保证原子性。

你可能不知道在Pentium及后续CPU中，下面两组指令作用相同，但一条指令反而不如三条指令快。



##### ****三、编译器优化 → C关键字volatile → memory破坏描述符****

memory比较特殊，可能是内嵌汇编中最难懂部分。为解释清楚它，先介绍一下编译器的优化知识，再看C关键字volatile。最后去看该描述符。

**1、编译器优化介绍**

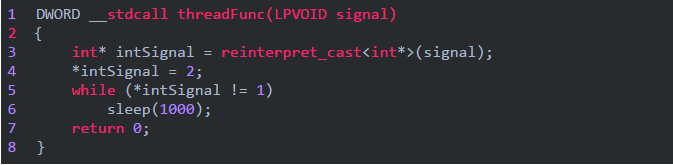
内存访问速度远不及CPU处理速度，为提高机器整体性能，在硬件上引入硬件高速缓存Cache，加速对内存的访问。另外在现代CPU中指令的执行并不一定严格按照顺序执行，没有相关性的指令可以乱序执行，以充分利用CPU的指令流水线，提高执行速度。以上是硬件级别的优化。再看软件一级的优化：一种是在编写代码时由程序员优化，另一种是由编译器进行优化。编译器优化常用的方法有：将内存变量缓存到寄存器；调整指令顺序充分利用CPU指令流水线，常见的是重新排序读写指令。对常规内存进行优化的时候，这些优化是透明的，而且效率很好。由编译器优化或者硬件重新排序引起的问题的解决办法是在从硬件（或者其他处理器）的角度看必须以特定顺序执行的操作之间设置内存屏障（memory barrier），linux 提供了一个宏解决编译器的执行顺序问题。

在这里插入图片描述

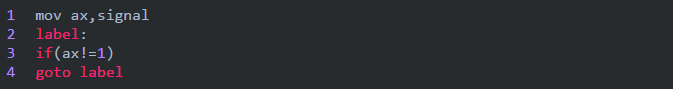
这个函数通知编译器插入一个内存屏障，但对硬件无效，编译后的代码会把当前CPU寄存器中的所有修改过的数值存入内存，需要这些数据的时候再重新从内存中读出。

**2、C语言关键字volatile**

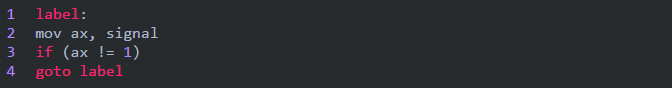
C语言关键字volatile（注意它是用来修饰变量而不是上面介绍的volatile）表明某个变量的值可能在外部被改变，因此对这些变量的存取不能缓存到寄存器，每次使用时需要重新存取。该关键字在多线程环境下经常使用，因为在编写多线程的程序时，同一个变量可能被多个线程修改，而程序通过该变量同步各个线程，例如：



该线程启动时将**intSignal**置为**2**，然后循环等待直到intSignal为1时退出。显然intSignal的值必须在外部被改变，否则该线程不会退出。但是实际运行的时候该线程却不会退出，即使在外部将它的值改为1，看一下对应的伪汇编代码就明白了：



对于C编译器来说，它并不知道这个值会被其他线程修改。自然就把它cache在寄存器里面。记住，C 编译器是没有线程概念的！这时候就需要用到volatile。volatile 的本意是指：这个值可能会在当前线程外部被改变。也就是说，我们要在**threadFunc**中的**intSignal**前面加上volatile关键字，这时候，编译器知道该变量的值会在外部改变，因此每次访问该变量时会重新读取，所作的循环变为如下面伪码所示：



**3、Memory**

有了上面的知识就不难理解Memory修改描述符了，Memory描述符告知GCC：

1）不要将该段内嵌汇编指令与前面的指令重新排序；也就是在执行内嵌汇编代码之前，它前面的指令都执行完毕。

2）不要将变量缓存到寄存器，因为这段代码可能会用到内存变量，而这些内存变量会以不可预知的方式发生改变，因此GCC插入必要的代码先将缓存到寄存器的变量值写回内存，如果后面又访问这些变量，需要重新访问内存。

如果汇编指令修改了内存，但是GCC 本身却察觉不到，因为在输出部分没有描述，此时就需要在修改描述部分增加“memory”，告诉GCC 内存已经被修改，GCC 得知这个信息后，就会在这段指令之前，插入必要的指令将前面因为优化Cache 到寄存器中的变量值先写回内存，如果以后又要使用这些变量再重新读取。

使用“volatile”也可以达到这个目的，但是我们在每个变量前增加该关键字，不如使用“memory”方便。

#### 博客2：<https://blog.csdn.net/wenqiang1208/article/details/71117818>

##### 1、什么是volatile

这是在MSDN中对关键字“volatile”的说明：

The volatile keyword is a type qualifier used to declare that an object can be modified in the program by something other than statements, such as the operating system, the hardware, or a concurrently executing thread.

volatile关键字是一种限定符用来声明一个对象在程序中可以被语句外的东西修改,比如操作系统、硬件或并发执行线程。

遇到该关键字，编译器不再对该变量的代码进行优化，不再从寄存器中读取变量的值，而是直接从它所在的内存中读取值，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

下面写个代码测试一下volatile关键字

注：VC6.0中一般调试模式没有进行代码优化，所以这个关键字的作用看不出来。下面通过插入汇编代码，测试有无 volatile 关键字，对程序最终代码的影响：

#include <stdio.h>

void main()

{

int i = 10;

int a = i;

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

// 下面汇编语句的作用就是改变内存中 i 的值

// 但是又不让编译器知道

\_\_asm {

mov dword ptr [ebp-4], 20h

}

int b = i;

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

}

12345678910111213141516171819

结果

在 Debug 版本模式运行程序：

i = 10 i = 32

在Release版本模式下运行程序：

i = 10 i = 10

上述 输出的结果明显表明，Release 模式下，编译器对代码进行了优化，第二次没有输出正确的 i 值。下面，我们把 i 的声明加上 volatile 关键字（在这里不再贴代码，就是上面的代码i前面加上volatile）

结果显示：在Debug和Release版本模式下运行程序

i = 10 i =32

这就是表示volatile关键字发挥了作用

##### 2、volatile在哪儿使用

一般说来，volatile用在如下的几个地方：

(1)、中断服务程序中修改的供其它程序检测的变量需要加volatile；

(2)、多任务环境下各任务间共享的标志应该加volatile；

(3)、存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能有不同意义；

另外，以上这几种情况经常还要同时考虑数据的完整性（相互关联的几个标志读了一半被打断了重写），在1中可以通过关中断来实现，2 中可以禁止任务调度，3中则只能依靠硬件的良好设计了。

##### 3、多线程下的volatile

当两个线程都要用到某一个变量且该变量的值会被改变时，应该用volatile声明，该关键字的作用是防止优化编译器把变量从内存装入CPU寄存器中。如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。volatile的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值，如下：

volatile BOOL bStop = FALSE; // (1) 在一个线程中： while( !bStop ) { ... } bStop = FALSE; return; 12345

//(2) 在另外一个线程中，要终止上面的线程循环： bStop = TRUE; while( bStop ); //等待上面的线程终止，123

如果bStop不使用volatile申明，那么这个循环将是一个死循环，因为bStop已经读取到了寄存器中，寄存器中bStop的值永远不会变成FALSE，加上volatile，程序在执行时，每次均从内存中读出bStop的值，就不会死循环了。

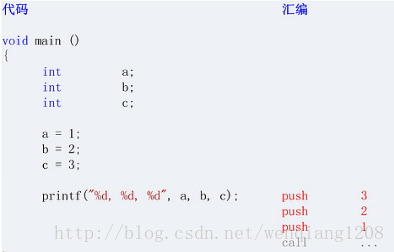
##### 4、volatile的特性

(1)易变性

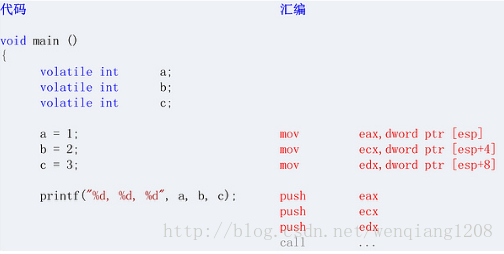
所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的volatile变量的寄存器内容，而是重新从内存中读取。volatile的这个特性，相信也是大部分朋友所了解的特性。

(2)不可优化

测试非volatile变量



测试volatile变量



volatile告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

(3)顺序性

C/C++ Volatile关键词前面提到的两个特性，让Volatile经常被解读为一个为多线程而生的关键词：一个全局变量，会被多线程同时访问/修改，那么线程内部，就不能假设此变量的不变性，并且基于此假设，来做一些程序设计。当然，这样的假设，本身并没有什么问题，多线程编程，并发访问/修改的全局变量，通常都会建议加上Volatile关键词修饰，来防止C/C++编译器进行不必要的优化。但是，很多时候，C/C++ Volatile关键词，在多线程环境下，会被赋予更多的功能，从而导致问题的出现。

以下测试在Linux 系统下，centos6.5

测试用例1:非volatile 变量



注意：全局变量A，B均为非volatile变量。通过gcc O2优化进行编译，你可以惊奇的发现，A，B两个变量的赋值顺序被调换了！！！在对应的汇编代码中，B = 0语句先被执行，然后才是A = B + 1语句被执行。

在这里，我先简单的介绍一下C/C++编译器最基本优化原理：保证一段程序的输出，在优化前后无变化。将此原理应用到上面，可以发现，虽然gcc优化了A，B变量的赋值顺序，但是foo()函数的执行结果，优化前后没有发生任何变化，仍旧是A = 1；B = 0。因此这么做是可行的。

测试用例2：一个volatile变量



此测试，相对于测试用例五，最大的区别在于，变量B被声明为volatile变量。通过查看对应的汇编代码，B仍旧被提前到A之前赋值，Volatile变量B，并未阻止编译器优化的发生，编译后仍旧发生了乱序现象。

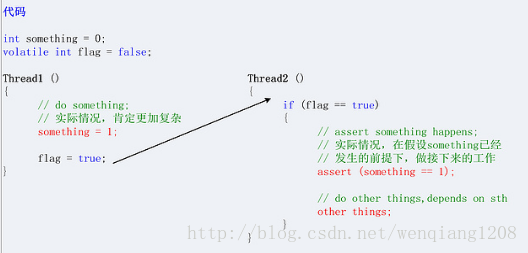
如此看来，C/C++ Volatile变量，与非Volatile变量之间的操作，是可能被编译器交换顺序的。在多线程下，如此使用volatile，会产生很严重的问题。

测试用例3：两个volatile变量同时将A，B两个变量都声明为volatile变量，再来看看对应的汇编。奇迹发生了，A，B赋值乱序的现象消失。此时的汇编代码，与用户代码顺序高度一直，先赋值变量A，然后赋值变量B。

如此看来，C/C++ Volatile变量间的操作，是不会被编译器交换顺序的。

下面看看在多线程下volatile的顺序性

下面这段伪代码，声明另一个Volatile的flag变量。一个线程(Thread1)在完成一些操作后，会修改这个变量。而另外一个线程(Thread2)，则不断读取这个flag变量，由于flag变量被声明了volatile属性，因此编译器在编译时，并不会每次都从寄存器中读取此变量，同时也不会通过各种激进的优化，直接将if (flag == true)改写为if (false == true)。



这只要flag变量在Thread1中被修改，Thread2中就会读取到这个变化，进入if条件判断，然后进入if内部进行处理。在if条件的内部，由于flag == true，那么假设Thread1中的something操作一定已经完成了，在基于这个假设的基础上，继续进行下面的other things操作。

但是实际情况中，不一定能保证flag == true时 ，something == 1，

在测试用例2中，C/C++ Volatile变量与非Volatile变量间的操作顺序，有可能被编译器交换。因此，上面多线程操作的伪代码，在实际运行的过程中，就有可能变成下面的顺序：

所以当flag == true时 ，something == 1，

其实，针对这个多线程的应用，真正正确的做法，是构建一个happens-before语义。进入if语句中，先assert(somthing == 1)，确保发生的情况下，执行otherthings。

小结：

C/C++ Volatile关键词的第三个特性：”顺序性”，能够保证Volatile变量间的顺序性，编译器不会进行乱序优化。Volatile变量与非Volatile变量的顺序，编译器不保证顺序，可能会进行乱序优化。同时，C/C++ Volatile关键词，并不能用于构建happens-before语义，因此在进行多线程程序设计时，要小心使用volatile，不要掉入volatile变量的使用陷阱之中

## static 静态成员和静态成员函数

https://www.cnblogs.com/arxive/p/11747943.html

与静态数据成员一样，静态成员函数是类的一部分，而不是对象的一部分。如果要在类外调用公用的静态成员函数，要用类名和域运算符"∷"。如Box∷volume( );实际上也允许通过对象名调用静态成员函数，如a.volume( );但这并不意味着此函数是属于对象a的，而只是用class a的类型而已。静态成员函数的作用是为了能处理静态数据成员。

可以说，两者的根本区别：非静态成员函数有this指针，静态成员函数并不属于某一对象，它与任何对象都无关，静态成员函数没有this指针。由此决定了静态成员函数不能访问本类中的非静态成员。在C++程序中，静态成员函数主要用来访问静态数据成员，而不访问非静态成员。

假如在一个静态成员函数中有以下语句：

cout<<height<<endl;   //若height已声明为static，则引用本类中的静态成员，合法

cout<<width<<endl;    //若width是非静态数据成员，不合法

但是，并不是绝对不能引用本类中的非静态成员，只是不能进行默认访问，因为无法知道应该去找哪个对象。如果一定要引用本类的非静态成员，应该加对象名和成员运算符“.”。如

cout<<a.width<<endl;            //引用本类对象a中的非静态成员

假设a已定义为Box类对象，且在当前作用域内有效，则此语句合法。

例:

运行结果为

　#include <iostream>

using namespace std;

class Student //定义Student类

{

public:

Student(int n, int a, float s) :num(n), age(a), score(s) { } //定义构造函数

void total();

static float average(); //声明静态成员函数

private:

int num;

int age;

float score;

static float sum; //静态数据成员

static int count; //静态数据成员

};

void Student::total() //定义非静态成员函数

{

sum += score; //累加总分

count++; //累计已统计的人数

}

float Student ::average() //定义静态成员函数

{

return(sum / count);

}

float Student::sum = 0; //对静态数据成员初始化

int Student::count = 0; //对静态数据成员初始化

int main()

{

Student stud[3] = { //定义对象数组并初始化

Student(1001,18,6.5),

Student(1002,19,6.3),

Student(1005,20,6.3) };

Student a(9,9,9); //对象a是为了证明可以调用静态函数 average

int n;

cout << "please input the number of students : ";

cin >> n; //输入需要求前面多少名学生的平均成绩

for (int i = 0; i < n; i++) //调用3次total函数

stud[i].total();

//调用静态成员函数

cout << "the average score of" << n << " students is " <<Student:: average() << endl;

//cout << "the average score of" << n << " students is " << a.average() << endl;

//这两种调用方式结果是一样的。

return 0;

}



说明：

(1) 在主函数中定义了stud对象数组，为了使程序简练，只定义它含3个元素，分别存放3个学生的数据。程序的作用是先求用户指定的n名学生的总分，然后求平均成绩(n由用户输入)。

(2) 在Student类中定义了两个静态数据成员sum(总分)和count(累计需要统计的学生人数)，这是由于这两个数据成员的值是需要进行累加的，它们并不是只属于某一个对象元素，而是由各对象元素共享的，可以看出： 它们的值是在不断变化的，而且无论对哪个对象元素而言，都是相同的，而且始终不释放内存空间。

(3) total是公有的成员函数，其作用是将一个学生的成绩累加到sum中。公有的成员函数可以引用本对象中的一般数据成员(非静态数据成员)，也可以引用类中的静态数据成员。score是非静态数据成员，sum和count是静态数据成员。

(4) average是静态成员函数，它可以直接引用私有的静态数据成员(不必加类名或对象名)，函数返回成绩的平均值。

(5) 在main函数中，引用total函数要加对象名(今用对象数组元素名)，引用静态成员函数average函数要用类名或对象名。

## function 和 bind

### C++ 之std::function()及 std::bind() 学习总结

本文链接：<https://blog.csdn.net/p942005405/article/details/84760715>

版权

**1. std::function介绍**

类模版std::function是一种通用、多态的函数封装。std::function的实例可以对任何可以调用的目标实体进行存储、复制、和调用操作，这些目标实体包括普通函数、Lambda表达式、函数指针、以及其它函数对象等。std::function对象是对C++中现有的可调用实体的一种类型安全的包裹（我们知道像函数指针这类可调用实体，是类型不安全的）。

通常std::function是一个函数对象类，它包装其它任意的函数对象，被包装的函数对象具有类型为T1, …,TN的N个参数，并且返回一个可转换到R类型的值。std::function使用 模板转换构造函数接收被包装的函数对象；特别是，闭包类型可以隐式地转换为std::function。std::function统一和简化了相同类型可调用实体的使用方式，使得编码变得更简单。

最简单的理解就是：

通过std::function对C++中各种可调用实体（普通函数、Lambda表达式、函数指针、以及其它函数对象等）的封装，形成一个新的可调用的std::function对象；让我们不再纠结那么多的可调用实体。一切变的简单粗暴。

**2. std::function的原型：**

1. template< class R, class... Args >
2. class function<R(Args...)>

R是返回值类型，Args是函数的参数类型，实例一个std::function对象很简单，就是将可调用对象的返回值类型和参数类型作为模板参数传递给std::function模板类。比如：

std::function<void()> f1;

std::function<int (int , int)> f2;

**3. std::function的用法**

std::function包含于头文件 #include<functional>中，可将各种可调用实体进行封装统一，包括

* 普通函数
* lambda表达式
* 函数指针
* 仿函数(functor 重载括号运算符实现)
* 类成员函数
* 静态成员函数

下面通过几个例子说明其用法

1. #include <iostream>
2. #include <functional>
4. using namespace std;
6. std::function<bool(int, int)> fun;
7. //普通函数
8. bool compare\_com(int a, int b)
9. {
10. return a > b;
11. }
12. //lambda表达式
13. auto compare\_lambda = [](int a, int b){ return a > b;};
14. //仿函数
15. class compare\_class
16. {
17. public:
18. bool operator()(int a, int b)
19. {
20. return a > b;
21. }
22. };
23. //类成员函数
24. class compare
25. {
26. public:
27. bool compare\_member(int a, int b)
28. {
29. return a > b;
30. }
31. static bool compare\_static\_member(int a, int b)
32. {
33. return a > b;
34. }
35. };
36. int main()
37. {
38. bool result;
39. fun = compare\_com;
40. result = fun(10, 1);
41. cout << "普通函数输出, result is " << result << endl;
43. fun = compare\_lambda;
44. result = fun(10, 1);
45. cout << "lambda表达式输出, result is " << result << endl;
47. fun = compare\_class();
48. result = fun(10, 1);
49. cout << "仿函数输出, result is " << result << endl;
51. fun = compare::compare\_static\_member;
52. result = fun(10, 1);
53. cout << "类静态成员函数输出, result is " << result << endl;
55. 类普通成员函数比较特殊，需要使用bind函数，并且需要实例化对象，成员函数要加取地址符
56. compare temp;
57. fun = std::bind(&compare::compare\_member, temp, std::placeholders::\_1, std::placeholders::\_2);
58. result = fun(10, 1);
59. cout << "类普通成员函数输出, result is " << result << endl;
60. }

**4. std::function使用注意事项:**

可见std::function的使用其实是很简单的，只要创建一个模板类对象，并传入相应的模板参数就可以存储任何具有相同返回值和参数的可调用对象，在调用的时候直接将std::function对象加上（）或加如参数就可以调用存储在其中的可调用实体。

* 关于可调用实体转换为std::function对象需要遵守以下两条原则：
  + 转换后的std::function对象的参数能转换为可调用实体的参数；
  + 可调用实体的返回值能转换为std::function对象的返回值。
* std::function对象最大的用处就是在实现函数回调，使用者需要注意，它不能被用来检查相等或者不相等，但是可以与NULL或者nullptr进行比较。
* 需要注意的是创建的std::function对象中存储的可调用实体不能为空，若对空的std::function进行调用将抛出 std::bad\_function\_异常。

**5. std::bind 介绍**

std::bind函数将可调用对象(用法中所述6类)和可调用对象的参数进行绑定，返回新的可调用对象(std::function类型，参数列表可能改变)，返回的新的std::function可调用对象的参数列表根据bind函数实参中std::placeholders::\_x从小到大对应的参数确定。

下面通过代码会更容易使用和理解:

1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. public:
6. void fun\_3(int k,int m)
7. {
8. cout<<k<<" "<<m<<endl;
9. }
10. };
12. void fun(int x,int y,int z)
13. {
14. cout<<x<<" "<<y<<" "<<z<<endl;
15. }
17. void fun\_2(int &a,int &b)
18. {
19. a++;
20. b++;
21. cout<<a<<" "<<b<<endl;
22. }
24. int main(int argc, const char \* argv[])
25. {
26. auto f1 = std::bind(fun,1,2,3); //表示绑定函数 fun 的第一，二，三个参数值为： 1 2 3
27. f1(); //print:1 2 3
29. auto f2 = std::bind(fun, placeholders::\_1,placeholders::\_2,3);
30. //表示绑定函数 fun 的第三个参数为 3，而fun 的第一，二个参数分别有调用 f2 的第一，二个参数指定
31. f2(1,2);//print:1 2 3
33. auto f3 = std::bind(fun,placeholders::\_2,placeholders::\_1,3);
34. //表示绑定函数 fun 的第三个参数为 3，而fun 的第一，二个参数分别有调用 f3 的第二，一个参数指定
35. //注意： f2 和 f3 的区别。
36. f3(1,2);//print:2 1 3

39. int n = 2;
40. int m = 3;
42. auto f4 = std::bind(fun\_2, n,placeholders::\_1);
43. f4(m); //print:3 4
45. cout<<m<<endl;//print:4 说明：bind对于不事先绑定的参数，通过std::placeholders传递的参数是通过引用传递的
46. cout<<n<<endl;//print:2 说明：bind对于预先绑定的函数参数是通过值传递的

49. A a;
50. auto f5 = std::bind(&A::fun\_3, a,placeholders::\_1,placeholders::\_2);
51. f5(10,20);//print:10 20
53. std::function<void(int,int)> fc = std::bind(&A::fun\_3, a,std::placeholders::\_1,std::placeholders::\_2);
54. fc(10,20);//print:10 20
56. return 0;
57. }

**6. 参考链接：**

https://blog.csdn.net/hailong0715/article/details/54890403

https://blog.csdn.net/liukang325/article/details/53668046

https://blog.csdn.net/xiaoyink/article/details/79348806

<https://www.cnblogs.com/diegodu/p/6180350.html>

# 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;

}

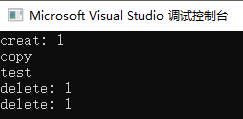
调用g\_Fun()时，会产生以下几个重要步骤：

(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。

运行结果：



###### 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;

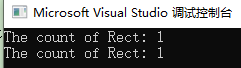
cout<<"The count of Rect: "<<Rect::getCount()<<endl;

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;

}

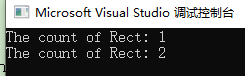
private:

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;

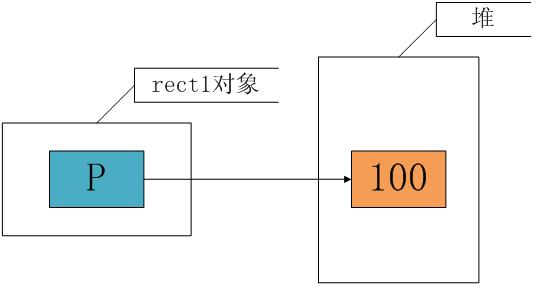
Rect rect2(rect1); // 复制对象

return 0;

}

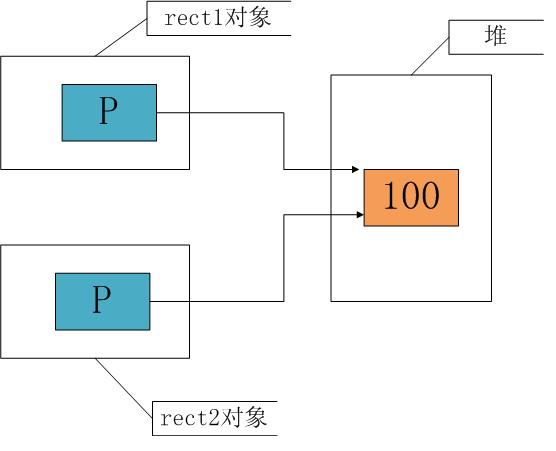
在这段代码运行结束之前，会出现一个运行错误。原因就在于在进行对象复制时，对于动态分配的内容没有进行正确的操作。我们来分析一下：

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:

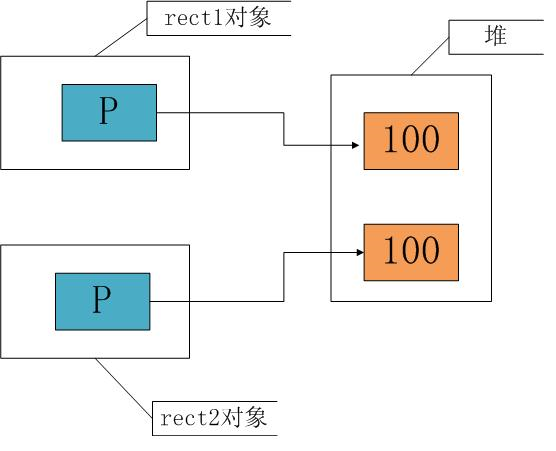
int width;

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。

# C++ 指针

### 指针函数与函数指针

#### 指针函数

指针函数是一个函数，只不过指针函数返回的类型是某一类型的指针。

**格式：**

类型名\* 函数名(函数参数列表)

**使用：**

/\*

\* 指针函数，返回int\* 指针变量

\*/

int\* add(int a, int b)

{

int \*p;

int c = a + b;

p = &c;

return p;

}

int main()

{

int\* p;

p = add(1, 4);

printf("%d\n", \*p);

getchar();

return 1;

}

#### 函数指针

函数指针是指向函数的指针变量。所以函数指针其实是指针变量，只不过该指针变量指向函数。

**格式：**

类型名 (\*指针变量名) (函数参数列表)

**函数指针的用途：**

1. a.调用函数
2. b.做函数的参数

**调用函数使用：**

/\*

\* 函数指针，调用函数

\*/

int fun1(int a, int b)

{

return a + b;

}

int main()

{

int(\*p)(int, int);

p = fun1;

printf("%d\n", p(1, 5)); //结果为 6

getchar();

return 1;

}

**函数参数使用：**

/\*

\* 函数指针，最为参数

\*/

int fun2(int a, int b)

{

return a + b;

}

int fun3(int a, int b, int(\*f)(int, int))

{

return f(a, b);

}

int main()

{

printf("%d\n", fun3(1, 5, fun2));

getchar();

return 1;

}

函数指针是和函数名的作用类似，都起到链接函数地址的作用。定义时要注意格式。

# 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);

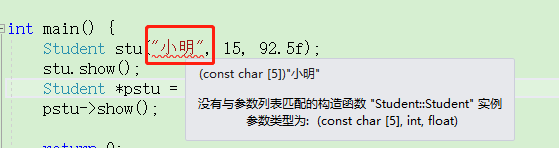
stu.show();

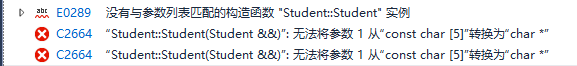
Student \*pstu = new Student("李华", 16, 96);

pstu->show();

return 0;

}

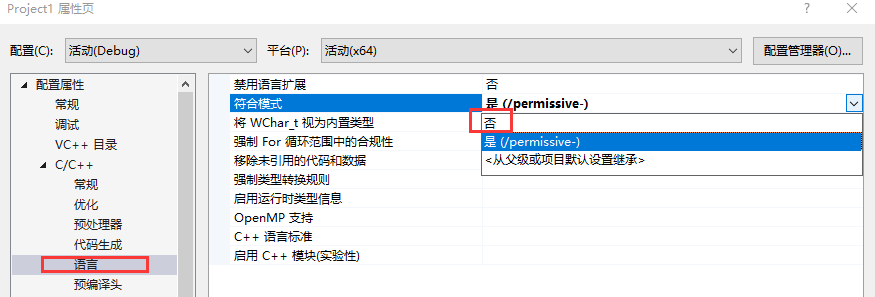
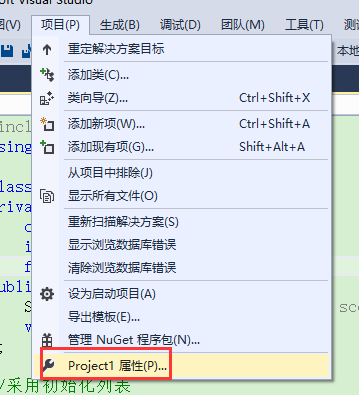




意思很明确const char\* 和char \*类型不匹配，个人理解为在创建stu对象的时候，系统将“小明”认定为是一个const类型的字符数组，所以跟student类中的char\* 类型相匹配。

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";

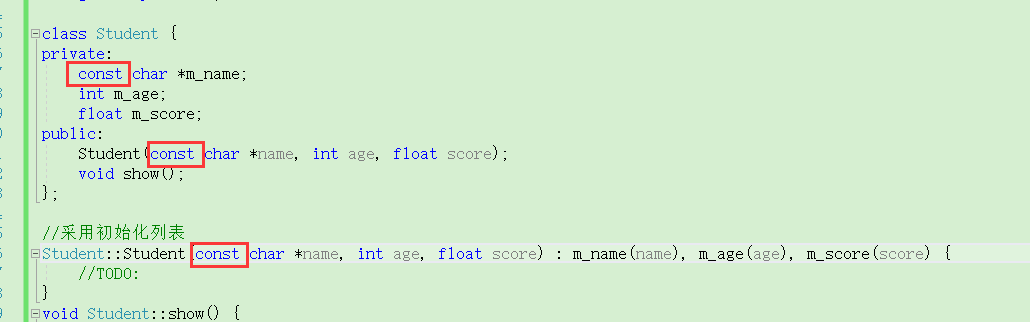
##### 3、先用字符数组进行存储，然后再使用指针：

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。



## switch 的case语句下有初始化变量提示错误：note: jump bypasses variable initialization

##### 原因：

例如：

*switch* (k) {

*case* x:

int a;

a = 2;

*//或者* *int* *a* *=* *2;*

*break*;

*case* y:

a = 5;

*default*:

*break*;

}

主要为了防止在case x:中定义的一个变量a在case y：中进行调用，由于a在switch中是全局可见的，所以是能在Y中调用的。但是如果在switch的过程中，跳过了case x，进入到了case y，那么在case y 中就调用了没有定义过的a,就会出现错误。

所以C++是不允许这么干的，要么你把a的作用域限制在case x的情况内，要么就把a 的定义放在switch之前，以避免这种错误发生。

##### 解决方案：

1、将case x的内容用大括号括起来，将a的作用域限制在case x内：

*switch* (k) {

*case* x:

{

int a;

a = 2;

}

*break*;

*case* y:

a = 5;

*default*:

*break*;

}

2、将a的定义放在switch之前。

int a;

*switch* (k) {

*case* x:

a = 2;

*break*;

*case* y:

a = 5;

*default*:

*break*;

}