关于C++中的类型转换操作符

内容简介：

本文对四种标准C++的类型转换符：static\_cast、dynamic\_cast、reinterpret\_cast、和const\_cast进行了介绍，通过本文应当能够理解这四个类型转换操作符的含义。

四种标准C++的类型转换符：static\_cast、dynamic\_cast、reinterpret\_cast、和const\_cast。

dynamic\_cast:动态类型转换，一般用在父类和子类指针或应用的互相转化;

static\_cast:静态类型转换，一般是普通数据类型转换(如int m=static\_cast<int>(3.14));

reinterpret\_cast:重新解释类型转换，很像c的一般类型转换操作;

const\_cast:常量类型转换，是把cosnt或volatile属性去掉。

下面将依次对它们进行相对详细地介绍。

主要内容：

一、static\_cast

二、dynamic\_cast

三、reinterpret\_cast

四、const\_cast

五、其它

一、static\_cast

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

支持子类指针到父类指针的转换，并根据实际情况调整指针的值，反过来也支持，但会给出编译警告，它作用最类似C风格的“强制转换”，一般来说可认为它是安全的。

用法：static\_cast < type-id > ( expression )

[功能]

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。

[描述]

主要如下几种用法：

(a)用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。

进行上行转换（把派生类的指针或引用转换成基类表示是安全的；

进行下行转换（把基类指针或引用转换成派生类表示时，由于没动态类型检查，所以是不安全的。

(b)用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。

(c)把空指针转换成目标类型的空指针。

(d)把任何类型的表达式转换成void类型。

注意：static\_cast不能转换掉expression的const、volitale、或者\_\_unaligned属性。

[举例]

这里，关于static\_cast的使用举例，通过与reinterpret\_cast的例子进行对比，容易理解，所以参见后面reinterpret\_cast的使用举例部分中对static\_cast的使用方法。

二、dynamic\_cast

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

用法：dynamic\_cast < type-id > ( expression )

[功能]

该运算符把expression转换成type-id类型的对象，Type-id必须是类的指针、类的引用或者void \*。

[描述]

支持子类指针到父类指针的转换，并根据实际情况调整指针的值，和static\_cast不同，反过来它就不支持了，会导致编译错误，这种转换是最安全的转换。如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。

dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。

[举例]

1)在类层次间进行转换

代码如下：

class B{

public:

       int m\_iNum;

       virtual void foo();

};

class D:public B{

    public:

       char \*m\_szName[100];

};

void func(B \*pb){

    D \*pd1 = static\_cast<D \*>(pb);

    D \*pd2 = dynamic\_cast<D \*>(pb);

}

这里可见，使用dynamic\_cast进行转换，如果出现了把指向父类对象的指针，转换成了子类的指针的时候，就会返回空值。

在上面的代码段中，如果pb指向一个D类型的对象，pd1和pd2是一样的，并且对这两个指针执行D类型的任何操作都是安全的；但是，如果pb指向的是一个B类型的对象，那么pd1将是一个指向该对象的指针，对它进行D类型的操作将是不安全的（如访问m\_szName），而pd2将是一个空指针。

另外要注意：B要有虚函数，否则会编译出错；static\_cast则没有这个限制。这是由于运行时类型检查需要运行时类型信息，而这个信息存储在类的虚函数表（关于虚函数表的概念，详细可见<Inside c++ object model>）中，只有定义了虚函数的类才有虚函数表，没有定义虚函数的类是没有虚函数表的。

2)类之间的交叉转换

代码如下：

class A{

public:

        int m\_iNum;

        virtual void f(){}

};

class B:public A{

};

class D:public A{

};

void foo(){

    B \*pb = new B;

    pb->m\_iNum = 100;

    D \*pd1 = static\_cast<D \*>(pb);    //compile error???实践好象没有编译错误

    D \*pd2 = dynamic\_cast<D \*>(pb); //pd2 is NULL

    delete pb;

}

这里，可见，如果出现了交叉转换的情况那么dynamic\_cast将会返回空值。

在函数foo中，使用static\_cast进行转换是不被允许的，将在编译时出错；而使用 dynamic\_cast的转换则是允许的，结果是空指针。

三、reinterpret\_cast

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

用法：reinterpret\_cast<type-id> (expression)

[功能]

它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。type-id必须是一个指针、引用、算术类型、函数指针或者成员指针。

[描述]

reinterpret\_cast是C++里的强制类型转换符，支持任何转换，但仅仅是如它的名字所描述的“重解释”而已。也就是说：操作符修改了操作数类型,但仅仅是重新解释了给出的对象的比特模型而没有进行二进制转换。例如：

int \*n= new int;

double \*d=reinterpret\_cast<double\*> (n);

在进行计算以后, d包含无用值.这是因为reinterpret\_cast仅仅是复制n的比特位到d, 没有进行必要的分析。

reinterpret\_cast是为了映射到一个完全不同类型的意思，这个关键词在我们需要把类型映射回原有类型时用到它。我们映射到的类型仅仅是为了故弄玄虚和其他目的，这是所有映射中最危险的(C++编程思想中的原话)。将static\_cast和reinterpret\_cast对比一下进行解释，比较容易理解：static\_cast 和 reinterpret\_cast 操作符修改了操作数类型，但是reinterpret\_cast 仅仅是重新解释了给出的对象的比特模型而没有进行二进制转换。例如：

int n=9;

double d=static\_cast<double>(n);

上面的例子中, 我们将一个变量从int转换到double。这些类型的二进制表达式是不同的,所以将整数9转换到双精度整数9，static\_cast需要正确地为双精度整数d补足比特位。其结果为 9.0。而reinterpret\_cast 的行为却不同:

int n=9;

double d=reinterpret\_cast<double & >(n);

这里, 和static\_cast不同，在进行计算以后, d包含无用值。这是因为reinterpret\_cast仅仅是复制n的比特位到d, 没有进行必要的分析.

因此, 需要谨慎使用 reinterpret\_cast。

[举例]

这个例子，将static\_cast和reinterpret\_cast对比进行测试，具体的输出参见其中的注释。

  1 #include <iostream>

  2 using std::cout;

  3 using std::endl;

  4 class CBaseX

  5 {

  6     public:

  7         int x;

  8         CBaseX() { x = 10; }

  9         void foo() { printf("CBaseX::foo() x=%d/n", x); }

10 };

11 class CBaseY

12 {

13     public:

14         int y;

15         int\* py;

16         CBaseY() { y = 20; py = &y; }

17         void bar() { printf("CBaseY::bar() y=%d, \*py=%d/n", y, \*py);}

18 };

19 class CDerived : public CBaseX, public CBaseY

20 {

21     public:

22         int z;

23 };

24

25 int main(int argc, char \*argv[])

26 {

27     float f = 12.3;

28     float\* pf = &f;

29

30     //基本类型的转换

31     cout<<"=================Basic Cast================="<<endl;

32     //======static cast<>的使用:

33     int n = static\_cast<int>(f); //成功编译

34     cout<<"n is :"<<n<<endl;//n = 12

35     //int\* pn = static\_cast<int\*>(pf);//编译错误，指向的类型是无关的,不能将指针指向无关的类型

36     void\* pv = static\_cast<void\*>(pf);//编译成功

37     int\* pn2 = static\_cast<int\*>(pv);//成功编译, 但是 \*pn2是无意义的内存（rubbish）

38     cout<<"pf is:"<<pf<<",pv is:"<<pv<<",pn2 is:"<<pn2<<endl;//三者值一样

39     cout<<"\*pf is:"<<\*pf<<",\*pn2 is:"<<\*pn2<<endl;//pf=12.3,pn2是无用值，注意无法使用"\*pv"因为编译错。

40

41     //======reinterpret\_cast<>的使用:

42     //int i = reinterpret\_cast<int>(f);//编译错误，类型‘float’到类型‘int’的转换无效.

43     //成功编译, 但是 \*pn 实际上是无意义的内存,和 \*pn2一样

44     int\* pi = reinterpret\_cast<int\*>(pf);

45     cout<<"pf is:"<<pf<<",pi is:"<<pi<<endl;//值一样

46     cout<<"\*pf is:"<<\*pf<<",\*pi is:"<<\*pi<<endl;//pi是无用值，和pn2一样。

47

48

49     //对象类型的转换

50     cout<<"=================Class Cast================="<<endl;

51     CBaseX cx;

52     CBaseY cy;

53     CDerived cd;

54

55     CDerived\* pD = &cd;

56     CBaseX \*pX = &cx;

57     CBaseY \*pY = &cy;

58     cout<<"CDerived\* pD = "<<pD<<endl;

59

60     //======static\_cast<>的使用:

61     CBaseY\* pY1 = pD;  //隐式static\_cast转换

62     //不一样是因为多继承，pD还要前移动以便也指向CBaseX.

63     cout<<"CDerived\* pD = "<<pD<<",CBaseY\* pY1 = "<<pY1<<endl;//pY1=pD+4!!!!!!

64

65     //CDerived\* pD1 = pY1;//编译错误,类型 ‘CBaseY\*’ 到类型 ‘CDerived\*’ 的转换无效

66     CDerived\* pD1 = static\_cast<CDerived\*>(pY1);//成功编译

67     cout<<"CDerived\* pD1 = "<<pD1<<endl;//现在 pD1 = pD

68

69     //pX = static\_cast<CBaseX\*>(pY);//编译错误，从类型 ‘CBaseY\*’ 到类型 ‘CBaseX\*’ 中的 static\_cast 无效。

70     pD1 = static\_cast<CDerived\*>(pY);//竟然可以编译通过!!!!!!

71     cout<<"CDerived\* pD1 = "<<pD1<<",CBaseY \*pY = "<<pY<<endl;//现在 pD1 = pY-4

72     //======reinterpret\_cast<>的使用:

73     CBaseY\* pY2 = reinterpret\_cast<CBaseY\*>(pD);// 成功编译, 但是 pY2 不是 CBaseY\*

74     cout<<"CDerived\* pD = "<<pD<<",CBaseY\* pY2 = "<<pY2<<endl;//pY2=pD!!!!!!

75

76     //======通过void的转换注意：

77     CBaseY\* ppY = pD;

78     cout<<"CDerived\* pD = "<<pD<<",CBaseY\* ppY = "<<ppY<<endl;//ppY = pD + 4

79

80     void\* ppV1 = ppY; //成功编译

81     cout<<"CBaseY\* ppY = "<<ppY<<",void\* ppV1 = "<<ppV1<<endl;//ppV1 = ppY

82

83     //CDerived\* ppD2 = ppV1;//编译错误,类型‘void\*’ 到类型 ‘CDerived\*’的转换无效

84     CDerived\* ppD2 = static\_cast<CDerived\*>(ppV1);

85     cout<<"CDerived\* ppD2 = "<<ppD2<<endl;//ppD2 = ppY, 但是我们预期 ppD2 = ppY - 4 = pD

86     //ppD2->bar();//系统崩溃,段错误

87     return 0;

88 }

这里，需要注意的地方是:

\*第63行中基类指针pY1被赋予子类指针pD后，pY1=pD+4而不是pD，因为pD是多继承，pD还要前移动以便也指向CBaseX.内存布局大致如下：

       +CDerived------------------+

       |   +CBase X--------+      |\

       |   |  int x        |      | 4 bytes

       |   +---------------+      |/

       |                          |

       |   +CBase Y--------+      |

       |   |  int y,\*py    |      |

       |   +---------------+      |

       +--------------------------+

\*第69行和70行的可以将父类指针用static\_cast强制转换成子类指针，但是两个无关的类的指针之间却不能转换。

\*第74行中使用reinterpret\_cast将子类指针强制转换赋给父类指针后，却没有像static\_cast那样将父类指针位置调整以指向正确的对象位置，这样导致虽然两者值是一样的，但是父指针所指向的内容却不是父对象了。

\*第76行之后使用void将子类转换成父类再转回子类，却无法使用了。

因为任何指针可以被转换到void\*，而void\*可以被向后转换到任何指针（对于static\_cast<> 和 reinterpret\_cast<>转换都可以这样做），如果没有小心处理的话错误可能发生。一旦我们已经转换指针为void\*，我们就不能轻易将其转换回原类所以使用void转换的时候一定要小心。在上面的例子中，从一个void\* 返回CDerived\*的唯一方法是将其转换为CBaseY\*然后再转换为CDerived\*。但是如果我们不能确定它是CBaseY\* 还是 CDerived\*，这时我们不得不用dynamic\_cast<> 或typeid[2](dynamic\_cast<>需要类成为多态，即包括“虚”函数，并因此而不能成为void\*)。

[其它]

dynamic\_cast<>，从另一方面来说，可以防止一个泛型CBaseY\* 被转换到CDerived\*。

四、const\_cast

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

用法：const\_cast<type\_id> (expression)

[功能]

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。

[描述]

const\_cast剥离一个对象的const属性，允许对常量进行修改。

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。

Voiatile和const类似。参见后面的例子可以了解更多信息。

[举例]

给出的源代码如下：

  1 #include <iostream>

  2 using std::cout;

  3 using std::endl;

  4

  5 class CTest

  6 {

  7     public:

  8         CTest(int i){m\_val = i;cout<<"construction"<<m\_val<<endl;}

  9         ~CTest(){cout<<"destructionn"<<endl;}

10         void SelfAdd(){m\_val++;};

11         int m\_val;

12 };

13

14 int main(int argc, char \*argv[])

15 {

16     const int ic = 100;

17     //int cc = const\_cast<int>(ic);//编译错误

18     int cc = const\_cast<int&>(ic);

19     cout<<cc<<endl;//输出100

20     //const\_cast<int &>(&ic)=200;//编译错误，从类型 ‘const int\*’ 到类型 ‘int&’ 中的 const\_cast 无效

21     const\_cast<int &>(ic)=200;

22     cout<<ic<<endl;//输出100

23     cout<<\*(&ic)<<endl;//输出100

24     //int \*pc = &ic;//编译错误，从类型 ‘const int\*’ 到类型 ‘int\*’ 的转换无效

25     const int \*pc=&ic;

26     //const\_cast<int &>(pc)=200;//编译错误，从类型 ‘const int\*\*’ 到类型 ‘int\*’ 中的 const\_cast 无效

27     const\_cast<int &>(ic)=200;

28     //printf("%d,%d/n", ic, \*pc);

29     cout<<ic<<','<<\*pc<<endl;//100,200

30     //int \*ppc = const\_cast<int\*>(ic);//编译错误

31     int \*ppc = const\_cast<int\*>(&ic);

32     \*ppc = 300;

33     cout<<ic<<','<<\*ppc<<endl;//100,300

34

35     const CTest test(1000);

36     CTest test2(1050);

37     //test = test2;//编译错误，无法给常量赋值

38     const\_cast<CTest &>(test)= test2;

39     cout<<test.m\_val<<endl;//输出1050

40 }

这里，结果输出参见每行代码相应的注释。根据结果可知：凡是对结构体或类进行这个转换，都是成功的，但对char，short等基本类型的转换，通过直接打印变量显示其值都是不成功的，但是通过指针却能显示出修改之后的值。

通过对代码进行反汇编，可知，虽然本身我们没使用优化，但系统还是对ic这个const进行了预编译般的替换，将它替换成“64h”（十六进制的64就是十进制的100），这肯定不是一般用户想要的结果，如果它不是一个C++的规范，应该算是个C++的bug吧。

[其他]

注意，

(1)操作对象

const\_cast操作的对象必须是pointer, reference, nor a pointer-to-data-member type，如下代码是错误的：

const int a = 5;

int aa = const\_cast<int>(a);

而使用引用的方式,如下却是正确的：

const int a = 5;

int aa = const\_cast<int&>(a);

(2)可能的误解

可能上面的描述误解的地方，根据参考资料中的一个评论，说：const\_cast只能修改变量的常引用的const属性，和变量的常指针的const属性，还有对象的const属性。要想改变常量本身的值是不可能的，也就是说，你改变的是引用的const属性，而不是常量本身的const属性。估计 const int ic = 100; 定义的时候就已经将这个基础类型对象放入常量符号表里面了，永远不会改变它的值。

五、其它

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

做为一个对前面所说的四种类型转换操作符的补充，对它们之间的区别大致进行说明一下，如下：

1，static\_cast和dynamic\_cast的对比：

1)static\_cast在编译期间发现错误。

对于基本类型，

它不允许将一种类型的指针指向另一种类型,所以如下代码是错误的：

float f = 12.3;

float\* pf = &f;

int\* pn = static\_cast<int\*>(pf);//编译错误，指向的类型是无关的,不能将指针指向无关的类型

对于复合类型（例如类，

它允许转换子对象地址赋值给父指针，也允许转换父对象地址赋值给子指针，但是不允许两个无关的类之间的转换，所以如下是错误的：

CBaseX \*pX = &cx;

CBaseY \*pY = &cy;

pX = static\_cast<CBaseX\*>(pY);//编译错误，从类型 ‘CBaseY\*’ 到类型 ‘CBaseX\*’ 中的 static\_cast 无效。

2)dynamic\_cast在运行期间发生错误，它只允许它允许转换子对象地址赋值给父指针，其它情况都返回空。

例如:

    B \*pb = new B;

    D \*pd = dynamic\_cast<D \*>(pb); //pd is NULL

    delete pb;

2，static\_cast，dynamic\_cast和reinterpret\_cast之间的对比：

1)static\_cast和dynamic\_cast可以执行指针到指针的转换，或实例本身到实例本身的转换，但不能在实例和指针之间转换。static\_cast只能提供编译时的类型安全，而dynamic\_cast可以提供运行时类型安全。举个例子：

class a；class b:a；class c。

上面个类a是基类，b继承a，c和a,b没有关系。假设有一个函数void function(a&a);现在有一个对象是b的实例b，一个c的实例c。function(static\_cast<a&>(b)可以通过而function(static<a&>(c))不能通过编译，因为在编译的时候编译器已经知道c和a的类型不符，因此static\_cast可以保证安全。

2)reinterpret\_cast可以转换任意一个32bit整数，包括所有的指针和整数。可以把任何整数转成指针，也可以把任何指针转成整数，以及把指针转化为任意类型的指针，威力最为强大！但不能将非32bit的实例转成指针。总之，只要是32bit的东东，怎么转都行！对于刚刚说的例子，下面我们骗一下编译器，先把c转成类型a

b& ref\_b = reinterpret\_cast<b&>c;

这样，function(static\_cast<a&>(ref\_b))就通过了！因为从编译器的角度来看，在编译时并不能知道ref\_b实际上是c！而function(dynamic\_cast<a&>(ref\_b))编译时也能过，但在运行时就失败了，因为dynamic\_cast在运行时检查了ref\_b的实际类型，这样怎么也骗不过去了。

在应用多态编程时，当我们无法确定传过来的对象的实际类型时使用dynamic\_cast，如果能保证对象的实际类型，用static\_cast就可以了。至于reinterpret\_cast很象c语言那样的暴力转换。

参考资料：

http://zhidao.baidu.com/question/81318972.html

http://blog.csdn.net/guogangj/article/details/1545119

http://zhidao.baidu.com/question/212970514.html

http://blog.csdn.net/deyili/article/details/5354242

以上是从网上搜集的，以及根据自己的理解对C++中四种操作符号的总结，如有不准确的地方，感谢读者的告知。另外文件的图示内容由于格式转换所以不太准确，可以存成文本格式的。^\_^

作者：QuietHeart

Email：quiet\_heart000@126.com

日期：2011年7月12日