今天是剑指offer第二期

本期关键词：C++ C语言 类型转换，附带解释切片和虚函数的链接

如果有什么问题或者建议，看到我哪里写的模糊或者错误，欢迎和我讨论共同进步！

**跪求指导和建议和讨论orz**



问题：

C++中，有哪4种与类型转换相关的关键字？各有什么特点？应该在什么场合下使用？ (p23)

## static\_case(静态转换)

## reinterpret\_cast(不相关类型的转换)

## const\_cast

## dynamic\_cast(动态转换)

（我是这样记的，静态常量需要动态地解释。

静态（static）

常量(const)

需要动态地(dynamic)

解释(重新解释reinterpret)

）

详细回答之前先抛出一个问题： C语言的强制类型转换可以随意转换我们想要的类型，不就够了吗？为什么还需要C++的四个新的类型转换符呢？（同时在C++环境中，原先的C-Style的类型转换仍旧可以使用。）

1. 新的类型转换控制符很好的控制类型转换的过程,允许控制各种类型不同的转换.（更灵活，更通用）

2. C++的类型转换控制符能告诉程序员或读者我们这个转换的目的是什么,我们只要看一下代码的类型转换控制符,就能明白我们想要达到什么样的目的.（一目了然）

以及。。看着更高级、更专业、更能突出咱们程序员的专业水准对不对？（误）



下面详细对比和说明一下C语言类型转换和C++的4种类型转换

会有很多例子，方便大家理解。

  。。。。。

C语言：有隐式类型的转换和强制类型的转换.

隐式类型转换：相关类型的转换，即意义相近的类型.

强制类型转换：不相关的类型的转换，比如：指针和整型

例如 ：

int i = 20;

double d = i;//隐式类型的转换.将int转换为double

int\* p = (int\*)i;//强制类型的转换.int 转换成int\*

C++：以下是C++的类型转换的四种类型。他们都是类模板

## static\_case(静态转换)

用法：static\_cast <typeid> (expression)

    说明：该运算符把expression转换为typeid类型，但没有运行时类型检查来确保转换的安全性。

    用途：

    a) 用于类层次结构中基类和派生类之间指针或者引用的转换。

        up-casting (把派生类的指针或引用转换成基类的指针或者引用表示)是安全的；

        down-casting(把基类指针或引用转换成子类的指针或者引用)是不安全的。

（注1：基类和派生类指针引用见本题末）

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

    c) 可以把空指针转换成目标类型的空指针(null pointer)。

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。

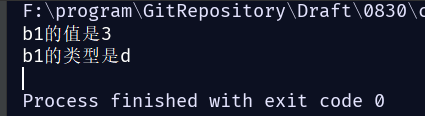
int i = 3;

double b1 = static\_cast<double>(i);

cout<<"b1的值是"<<b1<<endl;

cout<<"b1的类型是"<< typeid(b1).name()<<endl;

运行结果：（IDE是Clion, d代表double类型，注意包含头文件#include<typeinfo>）



不相关类型是不可以转换的,比如下面:

int\* p = static\_cast<int\*>(i); //i 是整型,转为int\*是不可以的

整形和指针属于两种完全不同的类型,所以是不可以通过static\_cast转换.

## reinterpret\_cast(不相关类型的转换)

 reinterpret\_cast

     用法：reinterpret\_cast <typeid>(expression)

     说明：转换一个指针为其他类型的指针，也允许将一个指针转换为整数类型，反之亦然。这个操作符能够在非相关的类型之间进行

        转换。操作结果只是简单的从一个指针到别的指针的值的二进制拷贝，在类型之间指向的内容不做任何类型的检查和转换。这

        是一个强制转换。使用时有很大的风险，慎用之。

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

在上面的例子中,不相关类型使用static\_cast是不可以转换的.

例如：

int\* p = static\_cast<int\*>(i);

但是可以通过reinterpret\_cast来进行转换：

int\* p = reinterpret\_cast<int\*>(i);

不仅仅只有变量类型可以转换,函数也可以:

typedef void(\*Fun)(); //函数指针

int Test(int i)

{

cout<<"Test"<<endl<<i<<endl;

return 0;

}

void fun()

{

Fun f = reinterpret\_cast<Fun>(Test);

f();

}

但是不建议使用reinterpert\_cast来进行函数指针的强转,**完全可以用C语言的强转来实现.**

## const\_cast

 用法：const\_cast<typeid>(expression)

     说明：这个类型操纵传递对象的const属性，或者是设置或者是移除。如：

           Class C{…}

           const C\* a = new C;

           C\* b = const\_cast<C\*>(a);

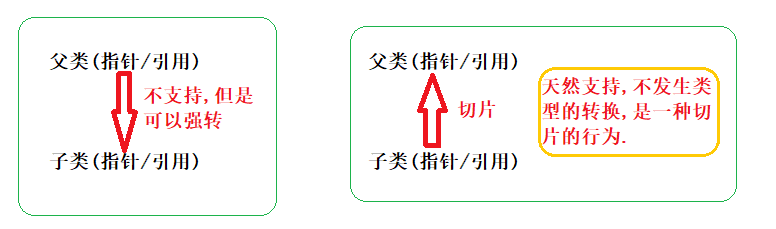
           如果将上面的const\_cast转换成其他任何其他的转换，编译都不能通过，出错的信息大致如下：

           “…cannot convert from 'const class C \*' to 'class C \*'”。

## dynmic\_cast(动态转换)

 用法：dynamic\_cast <typeid> (expression)

**目的:用于讲一个父类的指针/引用转化为子类的指针/引用.在前面的学习中,我们知道,把一个子类的指针或引用转化为父类的指针/引用,这是可以很自然的转换,会发生切片的行为（切片参见注2）.**   
这种类型的转换在C语言中是不支持的,因为在C语言中还没有出现多态.



使用场景:   
1.dynamic只能用于含有虚函数的类.   
2.dynamic在进行转换时先会检查是否会转换成功,能成功则转,不成功就会返回0;

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

void fun(AA\* ptr)

{

ptr->F();

BB\* b = (BB\*)ptr; //AA是父类(ptr)，BB是子类（b）

cout<<"a:"<<b->a<<endl;

cout<<"b:"<<b->b<<endl; //如果是将父类转换为子类时,由于父类中没有变量b,因此就会是随机值.

}

int main()

{

AA a;

a.a = 1;

BB b;

b.a = 10;

b.b = 100;

fun(&a);

fun(&b);

}

为了解决上面的问题,同时打印父类和子类的成员变量.所以此时就可以使用dynamic\_cast来进行转换了.

void fun(AA\* ptr)

{

ptr->F();

BB\* b = dynamic\_cast<BB\*>(ptr); //先会检查是否转换成功,如果转换失败,就会返回0;

if(b)

{

cout<<"转换成功"<<endl;

cout<<"a:"<<b->a<<endl;

cout<<"b:"<<b->b<<endl;

}

else

{

cout<<"转换失败"<<endl;

cout<<"a:"<<ptr->a<<endl;

}

}

如果此时在BB类中不重写F()函数.但是父类AA()类有虚函数,有虚表.BB类继承了AA自然也会有虚表,所以在子类中不重写虚函数也可以用来转换.但是如果在父类中没有虚函数,那么就会直接编译不通过. (虚函数参见注3)

下面的代码是四种casting方法的典型用法示例：

     #include <iostream>

     using namespace std;

     class Base

     {

     public:

         int \_base;

         virtual void printinfo()

         {

              cout << \_base << endl;

         }

     };

     class Derived : public Base

     {

     public:

         int \_derived;

         virtual void printinfo()

         {

              cout << \_derived << endl;

         }

     };

     int main(void)

     {

         Base    b1;

         Derived d1;

         int    aInt    = 10;

         long   aLong   = 11;

         float  aFloat  = 11.11f;

         double aDouble = 12.12;

         Derived\* pd = static\_cast<Derived\*>(&b1);       // down-casting 不安全

         Base\*    pb = static\_cast<Base\*>(&d1);          // up-casting   安全

         Derived& d = static\_cast<Derived&>(b1);         // down-casting 不安全

         Base& b = static\_cast<Base&>(d1);               // up-casting   安全

         aInt = static\_cast<int>(aFloat);                // 基本数据类型转换

         void\* sth = static\_cast<void\*>(&aDouble);       // 将double指针类型转换成void指针类型

         double\* bDouble = static\_cast<double\*>(sth);    // 将void指针类型转换成double指针类型

         cout << \*bDouble << endl;

         Base\* pb1 = dynamic\_cast<Base\*>(&d1);

         //Derived\* pd1 = dynamic\_cast<Derived\*>(&b1);   // 编译时有warning，运行时出错

         int bInt = reinterpret\_cast<int>(pb1);          // 将地址或指针转换成整数

         cout << bInt << endl;

         pb1 = reinterpret\_cast<Base\*>(bInt);            // 将整数转换成地址或指针

         int\* cInt = reinterpret\_cast<int\*>(&aFloat);    // 这个转换的结果会出乎意料

         cout << (int)\*cInt << endl;

         const Base\* bBase = new Base();

         Base\* cBase = const\_cast<Base\*>(bBase);

         //Base\* dBase = dynamic\_cast<Base\*>(bBase);     // 不能通过编译

         //Base\* eBase = static\_cast<Base\*>(bBase);      // 不能通过编译

         //Base\* fBase = reinterpret\_cast<Base\*>(bBase); // 不能通过编译

         return 0;

    }

注1：基类和派生类之间的引用和指针关系：

因为：基类如果定义的指针，指针中存放的数据都是基类中存在的，而派生类继承了这些subobject；但是如果派生类定义指针，这个指针所指向的对象不仅仅有基类的subobject，还有自己的一部分定义的数据，用这个指针去访问基类就可能会访问到一些根本不存在的成员。引用也是同理！

详见博客<https://blog.csdn.net/qq_34269988/article/details/88879830>）

注2：一般来说，就是向上类型转化过程中，[派生类](https://www.baidu.com/s?wd=%E6%B4%BE%E7%94%9F%E7%B1%BB&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)被截成只有基类相同的部分。详见<https://blog.csdn.net/Yinghuhu333333/article/details/80722389#commentBox>

注3虚函数

虚函数的作用主要是实现了多态的机制。关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”

说白了就是试图使用不变的代码来实现可变的算法

<https://blog.csdn.net/lyztyycode/article/details/81326699>推荐，比较清楚直白

以上还参见：<https://blog.csdn.net/Yinghuhu333333/article/details/80722389#commentBox>

<https://bbs.csdn.net/topics/390819632>的代码

<https://blog.csdn.net/xtzmm1215/article/details/46475565> 的原因



好了今天的分享时间就到这里了，欢迎批评欢迎建议。

**我也是搬运工，知识共享，跪求大佬纠错，表情包随便取。**

**只在公众号(Darring的象牙塔)和微博（@张丹颖呀）不同时发布。**