前面讲了 C++继承并扩展 C 语言的传统类型转换方式，最后留下了一些关于指针和引用上的转换问题，没有做详细地讲述。C++相比于 C 是一门面向对象的语言，面向对象最大的特点之一就是具有“多态性(Polymorphism)”。

要想很好的使用多态性，就免不了要使用指针和引用，也免不了会碰到转换的问题，所以在这一篇，就把导师讲的以及在网上反复查阅了解的知识总结一下。

C++提供了四个转换运算符：

```
const_cast <new_type> (expression)
static_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
dynamic_cast <new_type> (expression)
```

它们有着相同的结构，看起来像是模板方法。这些方法就是提供给开发者用来进行指针和引用的转换的。

其实我很早就想写这篇内容的，自己不断地查看导师发来的资料，也在网上不停地看相关的知识，却一直迟迟不能完全理解 C++转换运算符的用法，倒是看了那些资料后先写了一篇传统转换方面的内容。虽然从字面上很好理解它们大致是什么作用，但是真正像使用起来，却用不知道他们具体的用途，只会不断的被编译器提醒 Error。所以如果出现理解不到位或错误的地方，还希望前人或来者能够指正。

在我看来这些标准运算符的作用就是对传统运算符的代替，以便做到统一。就像我们用 std::endl 来输出换行，而不是'\n'。我会用代码来说明相应的传统转换可以如何转换为这些标准运算符。当然，这是大致的理解，在标准运算符上，编译器肯定有做更多的处理，特别是 dynamic_cast 是不能用传统转换方式来完全实现的。

在这一篇文章里，我会先讲讲我对 const_cast 运算符的理解。

const_cast (expression)

const_cast 转换符是用来移除变量的 const 或 volatile 限定符。对于后者，我不是太清楚，因为它涉及到了多线程的设计，而我在这方面没有什么了解。所以我只说 const 方面的内容。

用 const_cast 来去除 const 限定

对于 const 变量，我们不能修改它的值，这是这个限定符最直接的表现。但是我们就是想违背它的限定希望修改其内容怎么办呢？

下边的代码显然是达不到目的的：

```
const int constant = 10;
int modifier = constant;
```

因为对 modifier 的修改并不会影响到 constant，这暗示了一点:const_cast 转换符也不该用在对象数据上，因为这样的转换得到的两个变量/对象并没有相关性。
只有用指针或者引用，让变量指向同一个地址才是解决方案，可惜下边的代码在 C++中也是编译不过的：

```
const int constant = 21;
int* modifier = &constant
// Error: invalid conversion from 'const int*' to 'int*'
```

（上边的代码在 C 中是可以编译的，最多会得到一个 warning，所在 C 中上一步就可以开始对 constant 里面的数据胡作非为了）
把 constant 交给非 const 的引用也是不行的。

```
const int constant = 21;
int& modifier = constant;
// Error: invalid initialization of reference of type 'int&' from
expression of type 'const int'
```

于是 const_cast 就出来消灭 const，以求引起程序世界的混乱。

下边的代码就顺利编译功过了：

```
const int constant = 21;
const int* const_p = &constant;
int* modifier = const_cast<int*>(const_p);
*modifier = 7;
```

传统转换方式实现 const_cast 运算符

我说过标：准转换运算符是可以用传统转换方式实现的。const_cast 实现原因就在于 C++对于指针的转换是任意的，它不会检查类型，任何指针之间都可以进行互相转换，因此 const_cast 就可以直接使用显示转换(int*)来代替：

```
const int constant = 21;
const int* const_p = &constant;
int* modifier = (int*)(const_p);
```

或者我们还可以把他们合成一个语句，跳过中间变量，用

```
const int constant = 21;
int* modifier = (int*)(&constant);
```

替代

```
const int constant = 21;
int* modifier = const_cast<int*>(&constant);
```

为何要去除 const 限定

从前面代码中已经看到，我们不能对 constant 进行修改，但是我们可以对 modifier 进行重新赋值。

但是，程序世界真的混乱了吗？我们真的通过 modifier 修改了 constant 的值了吗？修改 const 变量的数据真的是 C++去 const 的目的吗？

如果我们把结果打印出来：

```
cout << "constant: "<< constant <<endl;
cout << "const_p: "<< *const_p <<endl;
cout << "modifier: "<< *modifier <<endl;
/**
constant: 21
const_p: 7
modifier: 7
**/
```

constant 还是保留了它原来的值。

可是它们的确指向了同一个地址呀：

```
cout << "constant: "<< &constant <<endl;
cout << "const_p: "<< const_p <<endl;
cout << "modifier: "<< modifier <<endl;
/**
constant: 0x7fff5fbff72c
const_p: 0x7fff5fbff72c
modifier: 0x7fff5fbff72c
**/
```

这真是一件奇怪的事情，但是这是件好事：说明 C++里是 const，就是 const，外界千变万变，我就不变。不然真的会乱套了，const 也没有存在的意义了。IBM 的 C++指南称呼“*modifier = 7;”为“未定义行为（Undefined Behavior）”。所谓未定义，是说这个语句在标准 C++中没有明确的规定，由编译器来决定如何处理。

位运算的左移操作也可算一种未定义行为，因为我们不确定是逻辑左移，还是算数左移。

再比如下边的语句：v[i] = i++; 也是一种未定义行为，因为我们不知道是先做自增，还是先来找数组中的位置。

对于未定义行为，我们所能做的所要做的就是避免出现这样的语句。对于 const 数据我们更要这样保证：**绝对不对 const 数据进行重新赋值。**

如果我们不想修改 const 变量的值，那我们又为什么要去 const 呢？

原因是，我们可能调用了一个参数不是 const 的函数，而我们要传进去的实际参数确实 const 的，但是我们知道这个函数是不会对参数做修改的。于是我们就需要使用 const_cast 去除 const 限定，以便函数能够接受这个实际参数。

```cpp
#include <iostream>
using namespace std;

void Printer (int* val,string seperator = "\n")
{
    cout << val<< seperator;
}

int main(void)
{
    const int consatant = 20;
    //Printer(consatant);//Error: invalid conversion from 'int' to
'int*'
    Printer(const_cast<int *>(&consatant));

    return 0;
}
```

出现这种情况的原因，可能是我们所调用的方法是别人写的。还有一种我能想到的原因，是出现在 const 对象想调用自身的非 const 方法的时候，因为在类定义中，const 也可以作为函数重载的一个标示符。有机会，我会专门回顾一下我所知道 const 的用法，C++的 const 真的有太多可以说的了。