

## 第八章 强制类型转换

C++提供了隐式类型转换，所谓隐式类型转换，是指不需要用户干预，编译器默认进行的类型转换行为（很多时候用户可能都不知道到底进行了哪些转换）。例如：

```
int nValue = 8;
double dValue = 10.7;
// nValue会被自动转换为double类型，用转换的结果再与dValue相加
double dSum = nValue + dValue;
```

但是很多时候我们希望在表达式中明确指定将一种类型转换为另一种类型，这种转换方式我们称之为显示类型转换。隐式类型转换一般是由编译器进行转换操作，显示类型转换是由程序员写明要转换成的目标类型。显示类型转换又被称为强制类型转换。

C++提供了四种强制类型转换操作符：**static\_cast**、**const\_cast**、**dynamic\_cast**、**reinterpret\_cast**

### 8.1 const\_cast

const\_cast是一种C++运算符，主要是用来去除复合类型中const和volatile属性(没有真正去除)

变量本身的const属性是不能去除的，要想修改变量的值，一般是去除指针（或引用）的const属性，再进行间接修改。

用法:const\_cast <type> (expression)

通过const\_cast运算符，也只能**将const type \*转换为type \***，**将const type&转换为type&**。

也就是说源类型和目标类型除了const属性不同，其他地方完全相同。

我们先来看这样一段代码：

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

int main () {
    const int data = 100;
    int *pp = (int *)&data;
    *pp = 300;
    cout << "data = " << data << "\t地址 : " << &data << endl << endl ;
    cout << " pp = " << *pp << "\t地址 : " << pp << endl << endl ;

    int *p = const_cast<int*>( &data ) ;
    cout << "data = " << data << "\t地址 : " << &data << endl << endl ;
    cout << " p = " << *p << "\t地址 : " << p << endl << endl ;
```

```

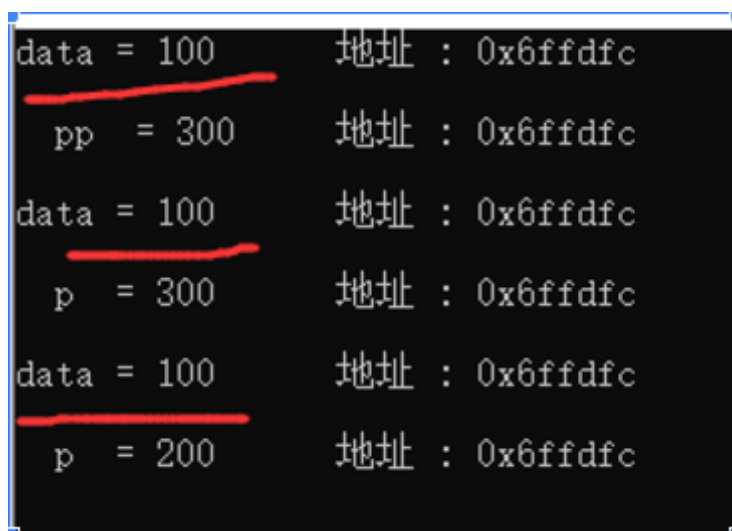
    *p = 200 ;

    cout << "data = " << data << "\t地址 : " << &data << endl << endl ;
    cout << "    p  = " << *p << "\t地址 : " << p << endl << endl ;

    return 0 ;
}

```

运行结果如下：



```

data = 100      地址 : 0x6ffdfc
pp  = 300      地址 : 0x6ffdfc
data = 100      地址 : 0x6ffdfc
p  = 300      地址 : 0x6ffdfc
data = 100      地址 : 0x6ffdfc
p  = 200      地址 : 0x6ffdfc

```

很奇怪？ data 的地址是 0x6ffdfc, p 指向的地址也是 0x6ffdfc, 但是修改 p 之后, 同一个地址上的内容却不相同。

可能是 const 的问题？ const 的机制，就是在编译期间，用一个常量代替了 data。这种方式叫做**常量折叠**。

**常量折叠与编译器的工作原理有关，是编译器的一种编译优化。**在编译器进行语法分析的时候，将常量表达式计算求值，并用求得的值来替换表达式，放入常量表。所以在上面的例子中，编译器在优化的过程中，会把碰到的data（为const常量）全部以内容100替换掉，跟宏的替换有点类似。

**常量折叠只对原生类型起作用，对我们自定义的类型，是不会起作用的**

```

#include <iostream>
#include <stdio.h>

using namespace std;

class Test {
public:
    int a;
    Test() {
        a = 10;
    };
};

int main()
{

```

```

const Test b;
Test *b1 = const_cast<Test *>(&b);
cout << "b = " << b.a << endl;
b1->a = 100;
cout << "b = " << b.a << ", *b1 = " << b1->a << endl;

return 0;
}

```

函数的形参为非const，当我们传递一个const修饰的变量时可以通过const\_cast去除该变量的const属性。

```

#include <iostream>
using namespace std;

const int *f(int *t)
{
    int *p = new int;
    *p = 100;
    return p;
}

int main () {
    int x = 1;
    int *p1 = const_cast<int *>(f(&x));
    *p1 = 1000;

    const int *y = new int(10);
    //int *p2 = const_cast<int *>(f(y));//[Note] in passing argument 1 of 'const int* f(i
    int *p2 = const_cast<int *>(f(const_cast<int *>(y)));
    cout << *y <<endl;
    // *y = 100; //[Error] assignment of read-only location '* y'
}

```

const\_cast也可以去除函数返回值的const属性

## 8.2 static\_cast

1、static\_cast 主要用于内置数据类型之间的相互转换

```

double dValue = 12.12;
float fValue = 3.14; //从“double”到“float”截断
int nDValue = static_cast<int>(dValue); // 12
int nFValue = static_cast<int>(fValue); // 3

```

2、也可以转换自定义类型。如果涉及到类，static\_cast只能在有相互联系（继承）的类型间进行转换，且不一定包含虚函数

```

class A
{
};
class B : public A
{
    public:
        void f();
};
class C
{
};

int main()
{
    A *pA = new A;
    B *pB = static_cast<B*>(pA); // 编译不会报错，B类继承于A类
    pB = new B;
    pA = static_cast<A*>(pB); // 编译不会报错，B类继承于A类
    // 编译报错，C类与A类没有任何关系。error C2440: “static_cast”: 无法从“A *”转换为“C *”
    C *pC = static_cast<C*>(pA);
    A *p1;
    B b2;
    p1 = &b2;
    static_cast<B *>(p1)->f();
}

```

## 8.3 dynamic\_cast

dynamic\_cast是四个强制类型转换操作符中最特殊的一个，它支持运行时识别指针或引用。

dynamic\_cast用于类继承层次间的指针或引用转换。主要还是用于执行“安全的向下转型（safe downcasting）”，也即是基类对象的指针或引用转换为同一继承层次的其他指针或引用。

至于“向上转型”（即派生类指针或引用类型转换为其基类类型），本身就是安全的，尽管可以使用dynamic\_cast进行转换，但这是没必要的，普通的转换已经可以达到目的。

“向下转型”的前提条件：被转换对象必须是多态类型（必须公有继承自其他类，或者有虚函数）

```

#include <iostream>
using namespace std;

class Base
{
    public:
        Base(){};
        virtual void Show(){cout<<"This is Base calss";}
};
class Derived:public Base
{
    public:
        Derived(){};
}

```

```

    void Show(){cout<<"This is Derived class";}
};

int main()
{
    Base *base ;
    Derived *der = new Derived;
    //base = dynamic_cast<Base*>(der); //正确，但不必要。
    base = der; //先上转换总是安全的
    base->Show();
}

```

“向下转型” 有两种情况。

- 一种是基类指针所指对象是派生类类型的，这种转换是安全的；
- 另一种是基类指针所指对象为基类类型，在这种情况下dynamic\_cast在运行时做检查，转换失败，返回结果为0；

```

#include <iostream>
using namespace std;

class Base
{
public:
    Base(){};
    virtual void Show(){cout<<"This is Base calss" << endl;}
};

class Derived:public Base
{
public:
    Derived(){};
    void Show(){cout<<"This is Derived class" << endl;}
    void d_f()
    {cout << "d_f" << endl;}
};

class Derived1:public Base
{
public:
    Derived1(){};
    void Show(){cout<<"This is Derived1 class" << endl;}
    void d_f()
    {cout << "d1_f" << endl;}
};

void f(Base *p)
{
    Derived *d;
    if (d = dynamic_cast<Derived *>(p))
    {
        d->d_f();
    }
}

```

```

        d->Show();
    }
}

int main()
{
    Derived d1;
    Derived1 d2;
    f(&d2);

    Base *p = new Derived;
    f(p);

    Base *p1 = new Base;
    f(p1);
    #if 0
    Derived d;
    Base *n = new Derived;

```

## 8.4 reinterpret\_cast

1、reinterpret\_cast 用于进行各种不同类型的指针之间、不同类型的引用之间以及指针和能容纳指针的整数类型之间的转换。转换时，执行的是逐个比特复制的操作。

2、这种转换提供了很强的灵活性，但转换的安全性只能由程序员的细心来保证了。例如，程序员执意要把一个 int\* 指针、函数指针或其他类型的指针转换成 string\* 类型的指针也是可以的，至于以后用转换后的指针调用 string 类的成员函数引发错误，程序员也只能自行承担查找错误的烦琐工作：（C++ 标准不允许将函数指针转换成对象指针，但有些编译器，如 Visual Studio 2010，则支持这种转换）

```

#include <iostream>
using namespace std;
class A
{
public:
    int i;
    int j;
    A(int n):i(n),j(n) { }
};

int main()
{
    A a(100);
    int &r = reinterpret_cast<int&>(a); //强行让 r 引用 a
    r = 200; //把 a.i 变成了 200
    cout << a.i << "," << a.j << endl; // 输出 200,100
    int n = 300;
    A *pa = reinterpret_cast<A*> ( & n); //强行让 pa 指向 n
    pa->i = 400; // n 变成 400
    pa->j = 500; //此条语句不安全，很可能导致程序崩溃
    cout << n << endl; // 输出 400

```

```
long long la = 0x12345678abcdLL;
pa = reinterpret_cast<A*>(la); //la太长，只取低32位0x5678abcd拷贝给pa
unsigned int u = reinterpret_cast<unsigned int>(pa); //pa逐个比特拷贝到u
cout << hex << u << endl; //输出 5678abcd
typedef void (* PF1) (int);
typedef int (* PF2) (int,char *);
PF1 pf1; PF2 pf2;
pf2 = reinterpret_cast<PF2>(pf1); //两个不同类型的函数指针之间可以互相转换
}
```