C 风格（C-style）强制转型如下：

(T) exdivssion // cast exdivssion to be of type T

函数风格（Function-style）强制转型使用这样的语法：

T(exdivssion) // cast exdivssion to be of type T

这两种形式之间没有本质上的不同，它纯粹就是一个把括号放在哪的问题。我把这两种形式称为旧风格（old-style）的强制转型。

使用标准C++的类型转换符：**static\_cast**、dynamic\_cast、reintepret\_cast、和const\_cast。

3.1 **static\_cast**  
用法：**static\_cast** < type-id > ( exdivssion )   
该运算符把exdivssion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。它主要有如下几种用法：  
①用于类层次结构中基类和子类之间指针或引用的转换。  
　　进行上行转换（把子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子类表示）时，由于没有动态类型检查，所以是不安全的。  
②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。  
③把空指针转换成目标类型的空指针。  
④把任何类型的表达式转换成void类型。

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

3.2 dynamic\_cast  
用法：dynamic\_cast < type-id > ( exdivssion )  
该运算符把exdivssion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 \*；  
如果type-id是类指针类型，那么exdivssion也必须是一个指针，如果type-id是一个引用，那么exdivssion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。

Cpp代码 [复制代码](http://www.iteye.com/topic/372303) [[收藏代码http://www.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. class B
2. {
3. public:
4. **int** m\_iNum;
5. virtual void foo();
6. };
8. class D : public B
9. {
10. public:
11. **char** \*m\_szName[100];
12. };
14. void func(B \*pb)
15. {
16. D \*pd1 = static\_cast(pb);
17. D \*pd2 = dynamic\_cast(pb);
18. }

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(pb);

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

}

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）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。

另外，dynamic\_cast还支持交叉转换（cross cast）。如下代码所示。

Cpp代码 [复制代码](http://www.iteye.com/topic/372303) [[收藏代码http://www.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. class A
2. {
3. public:
4. **int** m\_iNum;
5. virtual void f(){}
6. };
8. class B : public A
9. {
10. };
12. class D : public A
13. {
14. };
16. void foo()
17. {
18. B \*pb = new B;
19. pb->m\_iNum = 100;
21. D \*pd1 = static\_cast(pb); //compile error
22. D \*pd2 = dynamic\_cast(pb); //pd2 is NULL
23. delete pb;
24. }

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(pb); //compile error

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

delete pb;

}

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。

3.3 reintepret\_cast  
用法：reintepret\_cast (exdivssion)  
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。  
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）。

该运算符的用法比较多。

3.4 const\_cast   
用法：const\_cast (exdivssion)  
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exdivssion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。

Voiatile和const类试。举如下一例：

Cpp代码 [复制代码](http://www.iteye.com/topic/372303) [[收藏代码http://www.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. class B
2. {
3. public:
4. **int** m\_iNum;
5. }
6. void foo()
7. {
8. const B b1;
9. b1.m\_iNum = 100; //comile error
10. B b2 = const\_cast(b1);
11. b2. m\_iNum = 200; //fine
12. }

class B

{

public:

int m\_iNum;

}

void foo()

{

const B b1;

b1.m\_iNum = 100; //comile error

B b2 = const\_cast(b1);

b2. m\_iNum = 200; //fine

}

**上面的代码编译时会报错，因为b1是一个常量对象，不能对它进行改变；  
使用const\_cast把它转换成一个常量对象，就可以对它的数据成员任意改变。注意：b1和b2是两个不同的对象。**

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

== **dynamic\_cast** .vs. **static\_cast**   
== ===========================================

Cpp代码 [复制代码](http://www.iteye.com/topic/372303) [[收藏代码http://www.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. class B { ... };
2. class D : public B { ... };
4. void f(B\* pb)
5. {
7. D\* pd1 = dynamic\_cast(pb);
9. D\* pd2 = static\_cast(pb);
10. }

class B { ... };

class D : public B { ... };

void f(B\* pb)

{

D\* pd1 = dynamic\_cast(pb);

D\* pd2 = static\_cast(pb);

}

If pb really points to an object of type D, then pd1 and pd2 will get the same value. They will also get the same value if pb == 0.

If pb points to an object of type B and not to the complete D class, then dynamic\_cast will know enough to return zero. However, **static\_cast** relies on the programmer’s assertion that pb points to an object of type D and simply returns a pointer to that supposed D object.

即**dynamic\_cast**可用于继承体系中的向下转型，即将基类指针转换为派生类指针，比**static\_cast**更严格更安全。**dynamic\_cast**在执行效率上比**static\_cast**要差一些，但**static\_cast**在更宽上范围内可以完成映射，这种不加限制的映射伴随着不安全性。**static\_cast**覆盖的变换类型除类层次的静态导航以外，还包括无映射变换、窄化变换(这种变换会导致对象切片,丢失信息)、用VOID\*的强制变换、隐式类型变换等...   
  
  
== ===========================================  
== **static\_cast** .vs. **reintepret\_cast**   
== ================================================

**reintepret\_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。(这句话是C++编程思想中的原话)

**static\_cast** 和 **reintepret\_cast** 操作符修改了操作数类型。它们不是互逆的； **static\_cast** 在编译时使用类型信息执行转换，在转换执行必要的检测(诸如指针越界计算, 类型检查). 其操作数相对是安全的。另一方面；**reintepret\_cast** 仅仅是重新解释了给出的对象的比特(byte)模型而没有进行二进制转换， 例子如下：

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。 要将整数 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reintepret\_cast 的行为却不同:

int n=9;

double d=**reintepret\_cast** (n);

这次, 结果有所不同. 在进行计算以后, d 包含无用值. 这是因为 **reintepret\_cast** 仅仅是复制 n 的比特位到 d, 没有进行必要的分析.

因此, 你需要谨慎使用 **reintepret\_cast**.