69 Casting in C++

尽管之前实际上使用过类型转换，但没有讨论过这一点，这是一种需要联系并通过经验来学习的东西，而不是别人告诉你它是如何运作的，在这个主题上只有理论而没有实践对你的帮助不会多大。

1. 什么是casting

这里的*casting*（转换）是指类型转换，或者说是必须在C++可用类型系统中进行的类型转换。

C++是一门强类型语言，意味着存在一个类型系统，而且类型是强制的。（见66课：类型双关）
如果我把某物设为int，那就不能突然把它当做double或者float，反过来也一样。我必须坚持原有的类型，除非有一个简单的**隐式转换**（见40课：隐式和显式），这意味着C++知道如何在这两种类型之间转换，并且没有数据损失，这就是隐式转换；或者是有一个**显示转换**（见66课：类型双关），告诉C++你需要把这个类型转换成目标类型，本章将正是介绍**强制转换**的含义，并了解如何使用它。

2. 类型转换

我们执行类型转换的方法，特别是显式类型转换，分为两种方式（个人理解），有C语言风格和C++风格的。

C语言风格类型转换

```C++
int a = 5;
double value = a; // 这是一种隐式转换，很容易做到而且没有数据丢失

double value = 5.25;
int a = value;  // 这也被视为是隐式转换，因为不需要具体说明要把这个类型转换成哪个类型

// 要想显式转换，可以像下行一样

int a = (int)value; // 当然这种情况下它可以隐式地做到，但是我们正在讨论强制转换，所以先这样写
```

举一个更好的例子：

```C++
int main()
{
    double value = 5.25;
    double a = value + 5.3;
    std::cout << a << std::endl;    // 显然，这里输出10.55

    std::cin.get();
}
```

如果我们将value转换成`int`类型，那就会省去小数部分，最后就变成 $5 + 5.3 = 10.3$ 了：

```C++
double a = int(value) + 5.3;  // 10.3 而不是 10.55
```

这里使用的技术叫做**C语言风格类型转换**，因为它在圆括号中指定了要强制转换的类型，然后是我们要强制转换的变量。
还可以整个括起来，截取整体的整数部分：

```C++
double a = int(value + 5.3);  // 10
```

C++风格类型转换

1. static_cast

用C++的方式做这个，要使用C++的`cast`：

```C++
double s = static_cast<int>(value) + 5.3;
```

C++风格的转换有多种，一个是`static_cast`，还有`reinterpret_cast`、`dynamic_cast`、`const_cast`，共这四种主要的cast。它们并不能做任何C风格类型转换做不到的事情，这并不是添加新功能，只是添加了一些*syntax sugar*。

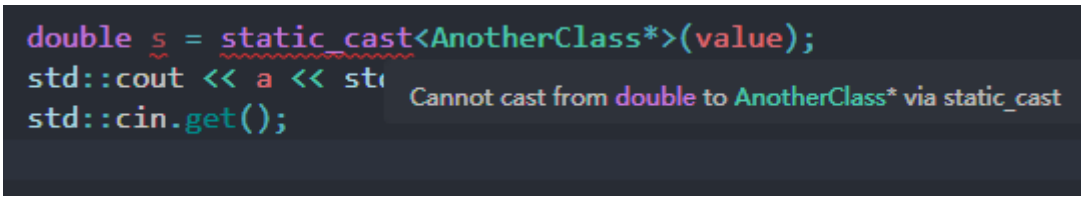
比如`dynamic_cast`，它会实际执行一个检查，如果转换不成功返回NULL，所以这做了额外的事情，会降低运行速度。但在大多数情况下，C++风格类型转换并不做额外的事情，它们只是一些代码中的英文单词。
例如`static_cast`，意思是静态类型转换，在静态类型转换的情况下，还会做一些其它的编译时检查，检查这种转换是否可能。
`reinterpret_cast`也是一样，就像是把我们说过的**类型双关**用英语表达出来一样，意思就是我要把这段内存重新解释成其它东西.
`const_cast`，移除或者添加变量的const限定。

所以为什么要搞这么多 cast？

是因为除了可能收到上面说的那些编译时的检查外，还可以方便我们从代码库中搜索它们。如果我想看到我的类型转换都在哪儿，也许我有性能问题而不想用`dynamic_cast`，我可以直接搜索这个词，如果用的是C语言风格的cast，就很难去搜索它，所以它对程序员的阅读和编写代码都有帮助。
而且它也能帮助我们减少在尝试强制转换时，可能意外犯下的错误，比如类型不兼容。

```C++
class AnotherClass
{
public:
    AnotherClass(){}
    ~AnotherClass(){}
};

int main()
{
    double value = 5.25;
    double a = int(value) + 5.3;

    double s = static_cast<AnotherClass*>(value);
    std::cout << a << std::endl;
    std::cin.get();
}
```

可以看到这是一个无效的类型转换



对于类型双关，可以用`interpret_cast`，这里我们将value指针处的数据重新解释为AnotherClass实例的数据：

```C++
double s = reinterpret_cast<AnotherClass*>(&value);
```

但是关键是用`static_cast`增加了编译时检查，如果你用C风格的转换它会默认地做`interpret_cast`会做的事情。

实际学习这些的最好方法就是实践，尝试构建一个使用这些cast的示例。

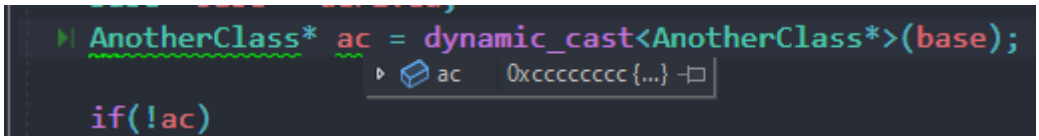
2. dynamic_cast

```C++
class Base
{
public:
    Base() {}
    virtual ~Base(){}
};

class Derived : public Base
{
public:
    Derived() {}
    ~Derived() {}
};

class AnotherClass : public Base
{
public:
    AnotherClass(){}
    ~AnotherClass(){}
};

int main()
{
    Derived* derived = new Derived();
    Base* base = derived;    // 将derived转换成Base类型

    std::cin.get();
}
```

有一个问题：这里我有一个Base指针，它是一个Derived类的实例呢还是AnotherClass类的实例呢？它们两个都是从Base类拓展来的。

```cpp
AnotherClass* ac = dynamic_cast<AnotherClass*>(base);
```

如果我用`dynamic_cast`，它不仅是问这个问题，而且它会尝试转换，如果转换失败还会做一些事情。

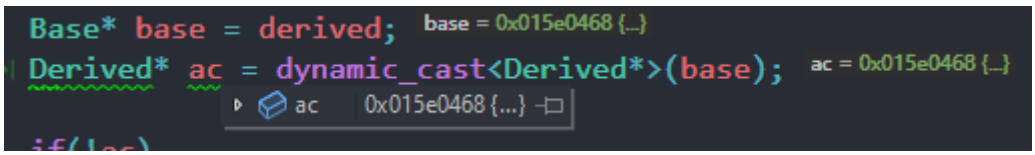


这里的`dynamic_cast`试图将基类指针`base`转换为`AnotherClass`指针。然而，原始的`base`指针实际上是指向一个`Derived`对象的，而非`AnotherClass`对象。这两个类（`Derived`和`AnotherClass`）都是从`Base`类派生的，但它们是完全不同的类，无法互相转换。

`dynamic_cast`在运行时执行类型检查。如果类型转换是安全的，那么它就执行转换，否则它返回null。在这个例子中，因为`base`指针实际上是指向`Derived`对象，而不是`AnotherClass`对象，所以`dynamic_cast`判断这个转换是不安全的，于是返回null。

这种类型检查的能力是`dynamic_cast`的一个重要特性，它可以在运行时保护我们免于执行不安全的类型转换。这就是为什么`ac`是null的原因。

改为Derived类后，这个指针就是有效的了，因为这个转换是成功的。



这与运行时类型信息*RTTI*(runtime type information)紧密联系。

你需要知道这些类型转换操作符对你来说是一种简化类型转换的方法，可能会让转换更可靠，因为它会做编译时检查，而`dynamic_cast`会做运行时检查。通过使用这样的强制转换操作符，你会得到更坚实的代码。