强制类型转换

C 风格的强制类型转换

- 在 C 语言中，强制类型转换是通过 "(目标类型)变量名" 或者是 "目标类型(变量名)" 所实现的。例如 int(3.14) 或者 (int)3.14
- (int)3.14 这样的方式常称之为 C 风格的类型转换，而 int(3.14) 则被称之为函数风格的强制类型转换。这两种方式在其它语言中也比较常见

static_cast

- 基本使用 —— static_cast<目标类型>(expression) —— C++ 中的提供的类型转换均以类模板的方式给出
- static_cast 即为静态类型转换，在编译时进行类型检查以及类型转换，其作用主要有以下 4 个方面
- 用途
  - ❶ 相关类型间的转换
    - 最典型的例子就是整型和实数类型之间的转换，double -> int，int -> double，float -> int 等等
    - double pi = 3.14;
int k = static_cast<int>(pi);
  - ❷ 子类对象向父类对象的转换
    - 子类对象向父类对象相当于对子类对象进行了一个"切片"，只会保留父类中存在的信息
    - Base b = static_cast<Base>(derived_ins);
  - ❸ void * 与其它类型指针之间的转换
    - operator new 就是使用 static_cast 将 void * 转换成指向具体对象的指针的
    - void *memory = operator new(sizeof(Buz));
buz = static_cast<Buz *>(memory);
  - ❹ 左值与右值之间的转换
    - std::move() 方法将一个左值转换为右值，其内部就是使用 static_cast 所实现的
    - 实际上 std::forward 内部也是使用 static_cast + 引用折叠所实现的

dynamic_cast

- 基本使用 —— dynamic_cast<目标类型>(expression)
- **dynamic_cast 主要用于父类和子类之间的类型转换，能够将父类指针或者是引用安全的转换成派生类的指针或者引用，这个过程发生在运行时。并且，dynamic_cast 正确工作的前提是父类中存在至少一个虚函数**
- 也就是说，dynamic_cast 主要用在多态类类型，有时候我们需要将基类指针特例化，就可以使用该类模板进行类型转换
- 转换目标
  - 指针 —— 当我们的转换目标为指针时，也就是子类指针，若转换失败，则返回空指针 (nullptr)
  - 引用 —— 当我们的转换目标为引用时，也就是子类引用，若转换失败，则会直接抛出 std::bad_cast 异常，因为没有空引用之说

const_cast

- 基本使用 —— const_cast<目标类型>(expression)
- **const_cast 的主要作用就是添加或去除指针或者是引用的 const 属性，但不可以是值类型**
- 该类型转换绝大部分出现在函数重载中，其余场合使用 const_cast 去除 const 属性就是一个很危险的行为
- 示例

```
int a = 1024;
const int *ptr = &a;

int *ktr = const_cast<int *>(ptr);
*ktr = 2048;

cout << *ktr << endl;
```

  - 这段代码能够正确的编译和运行，但是强烈建议不要这么做，很容易引发系统漏洞
- 用途
  - 函数重载
    - 比如说有这样的一个函数，接收两个 stirng，返回长度较长的那个

```
string &longer_string(string& a, string& b) {
    return a.size() > b.size() ? a : b;
}
```

    - 上一版的函数有一个局限性，一方面是不能接收 const 类型的实参，另一方面则不能接收右值

```
const string &longer_string(const string& a, const string& b) {
    return a.size() > b.size() ? a : b;
}
```

      - 这样就没问题了，const 和非 const 字符串都能使用
    - 但是上面的改进版本也有问题，因为不管实参是不是带有 const 属性的，我们的 longer_string 都将其隐式地转换成常量引用了

```
string &longer_string(string& a, string& b) {
    auto &r = longer_string(const_cast<const string&>(a),
                            const_cast<const string&>(b));
    return const_cast<string&>(r);
}
```

      - 这样一来我们就不再需要在 longer_string 的非常量引用的重载函数中重复书写逻辑，而只是进行一些类型转换
    - 最后一个函数版本我们首先利用 const_cast 为形参添加 const 属性，得到的结果再去除 const 属性，行为安全
- 因此，对于一个非常量的指针或者引用来说，我们可以先将其转换成 const 指针或引用，然后做一些事情。事情一做完，再把它们的 const 属性 "撸掉"，大家就当无事发生过
- 另外，const_cast 只对"底层" const 有用，也就是 const T *，对于顶层 const ，也就是 T * const 是没有用的

reinterpret_cast

- 基本使用 —— reinterpret_cast<目标类型>(expression)
- reinterpret_cast 主要用于处理无关类型间的转换，也就是说两个转换类型之间可以没有任何关系，想怎么转都行
- 常见用途
  - 将一种类型的指针转换成另一种类型的指针，按照转换后的内容重新解释内存中的内容

```
int i = 68;
int *ptr = &i;

char *ktr = reinterpret_cast<char *>(ptr);

cout << *ktr << endl;
```

    - 这里我们将 int 指针转换成了 char 类型的指针，也就是说，原本 ptr 指针读取 4 字节的内容，而 ktr 则只会读取变量 i 的第一个字节
  - 将一个整型转换成指针，或者将一个指针转换成整型