左值与右值

简单定义

- 左值（lvalue, locator value）
  - **左值表示一个占据内存中可识别位置的一个对象，更进一步地，可以对左值取地址**
  - int a = 10;
int *p = &a;
int **q = &p;
    - a、p、q 都是非常经典的左值（locator value），我们可以通过标识符 a、p、q 取出内存地址中对应的对象。
- 右值（rvalue）
  - **判断右值的一个简单方法就是能不能对变量或者是表达式取地址，如果不能，那它就是右值**
  - int foo() {return 10;}
x + 1;
    - 函数返回的对象（非引用、非指针）是一种典型的右值，这些对象在函数返回之后会被立即销毁，也就不存在说取地址这样的操作了。如 foo() = 20; 是一个典型的错误
    - 加减乘除等基本算数表达式也是右值，因为 x + 1 的结果保存在临时寄存器中，并不会输出到内存，因为没有办法对这个结果取地址，所以它们也是左值
- example
  - int a;
a = 4;
    - 如果我们在函数中执行该语句的话，变量 a 将会在栈帧中开辟一个 4 字节的内存空间，其值未定义。所以，a 为左值，能够取其地址。
    - 赋值语句中左操作数必须是一个左值。这很好理解，赋值操作本质上就是对内存进行更新，我们必须要找得到内存地址才能进行更新
  - a + 1 = 4;
foo() = 10;
    - a + 1 以及 foo() 返回的值均为右值，它们都是一个临时的值，在表达式结束时其生命周期将结束

左值与右值的转换

- 左值和右值在不同的表达式中有不同的定义
  - 加减乘除等简单运算符需要两个右值，并返回一个右值
  - int a = 10, b = 20;
int c = a + b;
    - 在第一行中，a、b 都是左值
    - 但在第二行中，由于要执行加法，所以会将左值隐式地转换成右值，其结果也是一个右值，保存在临时寄存器中，而后写入 c 的内存之中
- 解引用操作符 *
  - 解引用操作符作用于指针，取出指针指向的内存内容，该结果可以作为左值使用
  - int *p = &a;
*(p + 1) = 20;
    - p + 1 的结果是一个右值，但是 *(p + 1) 的结果为左值

作用于右值，返回左值

- 取地址操作符 &
  - 取地址操作符 & 自然不用多说，一定是作用在左值上，这也是我们前面定义左值使用的方式
  - int array[] = {1, 2, 3};
int *p = &array[2];
int *q = &(array + 1); // wrong!

作用于左值，返回右值

左值引用

- 引用类型又称之为"左值引用"，顾名思义，引用只能引用一个左值，而不能是右值
  - string& s = string("Hello"); // 不能引用右值
- **但是，凡事都有例外。常量引用可以引用一个右值**
  - const string& s = string("Hello");
  - void foo(string& s) {/*do something*/}

foo("Hello"); // Wrong，一个右值无法转换成左值引用
  - void foo(const string& s) {/*do something*/}

foo("Hello"); // Right，一个右值可以转换成常量引用
- int a = 10;
const int b = 20;

const int& c = a; // 引用非常量左值
const int& d = b; // 引用常量左值
const int& e = 10 + 20; // 引用右值
  - 正常来说，10 + 20 这一右值会在表示结束时被销毁，但是常量引用延长了其生命周期

右值引用

- 目的
  - 右值引用是 C++11 的新特性，主要解决 "移动语义"（move semantics） 以及 "完美转发"（Perfect forwarding）
- 行为
  - 右值引用只能作用在右值上，例如运算表达式、返回返回的值类型等
    - int&& s = 10;
int&& p = 10 * 1024;

int q = 225;
int&& k = q; //wrong!

int foo();
int&& f = foo();
  - **右值引用本身是一个左值**
    - 听起来会有点儿难以理解，但是右值引用是一个变量，而变量一般都是左值
    - int&& s = 1024;
int&& q = s; // wrong! 此时 s 为左值
  - **右值引用由于引用的是右值，而右值又是临时的、随时可能被销毁的对象。因此，在使用右值引用的地方我们可以随意的接管所引用对象的资源，而不用担心内存泄漏、数据被更改的情况**
- 移动语义
move semantics
  - 在一般的拷贝复制复制函数中，我们通常会使用 临时对象+swap 的方式来实现异常安全以及数据安全

```
class Buz {
private:
    int *m_ptr;

public:
    Buz(int *ptr = 0) : m_ptr(ptr) {}

    ~Buz() {delete m_ptr;}

    Buz(const Buz& other);

    Buz& operator= (const Buz& rhs) {
        Buz temp = Buz(rhs);
        using std::swap;
        swap(this->m_ptr, temp.m_ptr);
        return *this;
    }
};
```

在上述代码中，我们首先使用 rhs 通过拷贝构造函数构造出了临时对象 temp，然后交换 this 和 temp 中的数据。当函数返回时，temp 对象将被自动调用其析构函数并销毁，从而释放掉原来 this 的数据。这样的写法是异常安全的

  - Buz buz;
buz = Buz();
    - buz = Buz(); 这一简单的赋值背后其实发生了许多的事情
    - 首先，创建 Buz 对象，然后将这个临时的对象赋值给 buz，此时将会调用拷贝赋值，而在拷贝赋值又会生成一个临时的对象，如此一来，Buz 对象相当于被创建了 2 次。
  - 在有了右值引用以后，我们就可以使用移动的方式来编写移动赋值函数，直接将右值中的数据搬到自己这边儿来

```
Buz& operator= (Buz&& rhs) {
    using std::swap;
    swap(this->ptr, rhs.ptr);
    return *this;
}
```

    - buz = Buz(); 中的 Buz() 是一个右值，所以该赋值语句将调用移动赋值函数
  - **移动语义利用的基本事实就是右值本身就是临时的、随时可能被销毁的，那么我们从右值中"窃取"数据就不会有任何副作用。当我们窃取完数据以后，甚至可以将一些我们需要销毁的数据"挂在"该右值上，右值销毁时带着这些数据一并销毁**
- 我们可以通过 std::move() 函数将一个左值强制类型转成一个右值引用，注意 std::move() 本身不具有移动语义，它只是一个类型转换函数而已，内部使用 static_cast 实现