**拷贝，赋值，移动构造函数**

class Book

{

// 这里我们定义的swap函数，这样在标准库容器中swap时将使用我们的swap函数而不是标准库的

friend void swap(Book &left, Book &right);

public:

// 默认构造函数

Book() = default;

// 拷贝构造函数，如果我们没有定义拷贝构造函数，编译器会帮我们定义

// 编译器生成的拷贝，其成员采用拷贝的形式生成

Book(const Book &book)

{

\_bookNo = book.\_bookNo;

}

// 移动构造函数，接受一个右值，使用移动构造函数应确保移动的对象不在被使用

Book(Book &&book){

\_bookNo = book.\_bookNo;

book.\_bookNo = ""; // 将传进来的值设为空

}

// 如果我们的类不支持拷贝，则应该

// Book(const Book &book) = delete;

// 拷贝赋值操作，如果我们没有定义，编译器会帮我们定义

// 可选：后面的 & 是说明当前对象必须是个左值，才能使用这个操作

Book &operator=(const Book &book) &

{

\_bookNo = book.\_bookNo;

return \*this;

}

// 后面的 && 是说明当前对象必须是个右值，才能使用这个操作

Book &operator=(const Book &book) &&

{

\_bookNo = book.\_bookNo;

return \*this;

}

// 移动赋值操作，使用移动赋值函数应确保移动的对象不在被使用

Book &operator=(Book &&book)

{

\_bookNo = book.\_bookNo;

book.\_bookNo = "";

return \*this;

}

// // 这里有趣的是 Book book 是值参数，所以在生成book的时候会根据当前类的构造函数生成

// // 如果转入的是左值会调用拷贝构造函数

// // 如果转入的是右值会使用移动构造函数

// // 这样移动拷贝赋值合为一个

// Book &operator=(Book book)

// {

// swap(\*this, book);

// return \*this;

// }

// 如果我们的类不支持拷贝赋值，则应该

// Book& operator=(const Book& book) = delete;

// 析构函数，在对象被销毁时会被调用，同理没有定义编译器也会帮我们定义

~Book()

{

}

private:

std::string \_bookNo;

};

void swap(Book &left, Book &right){

// 之所以这样写是希望如果\_bookNo有定义swap，则调用\_bookNo的swap，否则调用标准库的swap

// 这是一个优先级的问题，在查找swap时总是先查找参数类型完全匹配的

using std::swap;

swap(left.\_bookNo, right.\_bookNo);

}

**左值与右值**

int main()

{

Book book1;

Book book2(book1); // 调用Book的拷贝函数生成book2

// 右值：常量，表达式 为右值

// 左值：变量 为左值

// 右值引用：右值的引用

// 右值可以赋值给左值，但左值不能赋值给右值

int a = 42; // a是左值，42是右值

int &b = a; // b是左值引用

int &&c = 42; // c是右值引用

// int &d = 42; // 错误，左值引用不能引用右值

int &&e = a \* 2; // 表达式是右值引用

// int &&f = a; // 错误右值引用不能引用左值

int &&g = move(a); // 正确，标准库move将a转为右值

Book book3(move(book2)); // 移动boo2到book3，移动后，不在使用book2

cout << "enter key" << endl;

cin.get();

return 0;

}