第13章 拷贝控制

1. **拷贝构造函数(copy constructor)** 和 **移动构造函数(move constructor)** 定义了当用同类型的另一个对象初始化本对象时做什么。
2. **拷贝赋值运算符(copy-assignment operator)** 和 **移动赋值运算符(move-assignment operator)** 定义了将一个对象赋予同类型的另一个对象时做什么。
3. **析构函数(destructor)** 定义了当此类型对象销毁时做什么。
4. 如果一个类没有定义这些拷贝控制成员，编译器会自动为它定义缺失的操作。

13.1 拷贝、赋值与销毁

13.1.1 拷贝构造函数

1. 如果一个构造函数的第一个参数是自身类类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数：

```
class Foo(){
public:
    Foo();            // 默认构造函数
    Foo(const Foo&); // 拷贝构造函数
    // ...
}
```

2. 如果我们没有为一个类定义拷贝构造函数，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。从给定对象中依次将每个非 `static` 成员拷贝到正在创建的对象中。
3. 每个成员的类型决定了它如何拷贝：对类类型的成员会使用其拷贝构造函数来拷贝，内置类型的成员则直接拷贝。

```
class Sales_data{
public:
    Sales_data(const Sales_data&);
    // ... ...

private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
}

// 自己定义，或让编译器合成：
Sales_data::Sales_data(const Sales_data& orig)
    : bookNo(orig.bookNo),          // 使用string的拷贝构造函数
    units_sold(orig.units_sold),    // 直接拷贝
    revenue(orig.revenue)           // 直接拷贝
    {   }
```

4. 当使用 **直接初始化** 时，实际上是要求编译器使用 **普通** 的函数匹配来选择与我们提供的参数最匹配的构造函数。
5. 当我们使用 **拷贝初始化** 时，要求编译器使用 **拷贝构造函数** 将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换。

```
// 直接初始化
string dots(10, '.');
string s(dots);

// 拷贝初始化
string s2 = dots;
string null_book = "9-9-9";
string nines = string(10, '9');
```

6. 拷贝初始化不仅在用 `=` 定义变量时会发生，在下列情况下也会发生：
   （1）将一个对象作为实参传递给一个非引用类型的形参
   （2）从一个返回类型为非引用类型的函数返回一个对象
   （3）用花括号列表初始化一个数组中的元素或一个聚合类中的成员。
7. 如果 **拷贝构造函数的参数** 不是 **引用类型**（而是值类型），则调用永远也不会成功——为了调用拷贝构造函数，我们必须拷贝它的实参，但为了拷贝实参，我们又需要调用拷贝构造函数，如此无限循环。

13.1.2 拷贝赋值运算符

1. 重载运算符(overloaded operator)的本质是函数，其名字由 `operator` 关键字后接表示要定义的运算符的符号组成 (比如 `operatro=`)。

2. 如果一个运算符是一个成员函数，其左侧运算对象就绑定到隐式的 `this` 参数。对于一个二元运算符，其右侧运算对象作为显示参数传递。
3. 为了与内置类型的赋值保持一直，赋值运算符通常应该返回一个指向其左侧运算对象的引用（连续拷贝时需要： `a = b = c` ）。
4. 标准库通常要求保存在容器中的类型要具有赋值运算符，且其返回值是左侧运算对象的引用。

```
class Sales_data{
public:
    // ... ...
   Sales_data& operator=(const Sales_data& rhs);

private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
}

Sales_data& Sales_data::operator=(const Sales_data& rhs)
{
    bookNo = rhs.bookNo;             // 调用string::operator=
    units_sold = rhs.units_sold;     // 使用内置类型的赋值
    revenue = rhs.revenue;
    return *this;                    // 返回左侧对象的引用
}
```

13.1.3 析构函数

1. 析构函数释放对象使用的资源，并销毁对象的非 `static` 数据成员。
2. 由于析构函数不接受参数，因此不能被重载。对于一个给定类，只会有唯一一个析构函数。
3. 析构过程中首先执行函数体，然后销毁成员。成员按初始化顺序的逆序销毁。
4. 析构函数体自身并不直接销毁成员，成员是在析构函数体之后隐含的析构阶段中被销毁的。
5. 在析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。
6. 销毁类类型的成员需要执行成员自己的析构函数。内置类型没有析构函数，因此销毁内置类型成员什么也不需要做。
7. 隐式销毁一个内置指针类型的成员不会 `delete` 它所指向的对象。（智能指针是类类型，所以在析构阶段被自动销毁时会根据情况 `delete` 所指的对象。）
8. 当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

13.1.4 三/五法则

1. 如果一个类需要自定义析构函数，几乎可以肯定它也需要自定义拷贝赋值运算符和拷贝构造函数。（例如动态分配对象的销毁带来的析构和拷贝问题）
2. 如果一个类需要一个拷贝构造函数，几乎可以肯定它也需要一个拷贝赋值运算符，反之亦然。
3. 然而无论是需要拷贝构造函数还是需要拷贝赋值运算符都不必然意味着也需要析构函数。

13.1.5 使用 `=default`

1. 通过将拷贝控制成员定义为 `=default` 来显示地要求编译器生成合成的版本。
2. 在类内用 `=default` 修饰成员的声明时，合成的函数将隐式地声明为内联的。
3. 如果不希望合成的成员是内联函数，应该只对成员的类外定义使用 `=default` 。
4. 只能对具有合成版本的成员函数使用 `=default` （即默认构造函数或拷贝构造函数）。

```
class Sales_data{
public:
    Sales_data() = default;          // 内联
    Sales_data(const Sales_data&);  // 外联
}
Sales_data::Sales_data(const Sales_data&) = default;
```

13.1.6 阻止拷贝

1. 新标准下，可以通过将拷贝构造函数和拷贝赋值运算符定义为 **删除的函数(deleted function)** 来阻止拷贝。
2. 与 `=default` 不同， `=delete` 必须出现在函数第一次声明的时候。编译器需要知道一个函数是删除的，以便禁止试图使用它的操作。
3. 一个默认的成员只影响为这个成员而生成的代码，因此 `=default` 直到编译器生成代码时才需要。
4. 与 `=default` 的另一个不同之处是，我们可以对任何函数指定 `=delete` ，相对的我们只能对编译器可以合成的默认构造函数或拷贝控制成员使用 `=default` 。

```
struct NoCopy{
    NoCopy() = default;             // 使用合成的默认构造函数
    NoCopy(const NoCopy&) = delete;              // 阻止拷贝
    NoCopy &operator=(const NoCopy&) = delete;  // 阻止赋值
    ~NoCopy() = default;               // 使用合成的析构函数
    // 其他成员... ...
};
```

5. 析构函数不能是删除的成员，否则无法销毁类型的对象了。

6. 对于删除了析构函数的类型，虽然不能直接定义这种类型的变量和成员，但可以动态分配这种类型的对象。但是不能释放这些对象。
7. 对于析构函数已删除的类型，不能定义该类型的变量或释放指向该类型动态分配对象的指针。

```
struct NoDtor{
    NoDtor() = default;      // 使用合成默认构造函数
    ~NoDtor() = delete;      // 不能销毁该类型的对象了
};
NoDtor nd;  // 错误：NoDtor的析构函数是删除的
NoDtor* p = new NoDtor();   // 正确：但我们不能delete p
delete p;   // 错误：NoDtor的析构函数是删除的
```

8. 本质上，当不可能拷贝、赋值或销毁类的成员时，类的合成拷贝控制成员就被定义为删除的。
9. 希望阻止拷贝的类应该使用 `=delete` 来定义它们自己的拷贝构造函数和拷贝赋值运算符，而不应该将它们声明为 `private` 的（旧标准时使用的方法）。

13.2 拷贝控制和资源管理

1. 类的行为像一个值，当拷贝一个像值的对象时，副本和原对象是完全独立的。改变副本不会对原对像有任何影响。
2. 行为像指针的类则共享状态。当拷贝时，副本和原对象使用相同的底层数据。改变副本也会改变原对象。

13.2.1 行为像值的类

1. 对于一个赋值运算符来说，即使是将一个对象赋予它自身，也要能正确工作。一个好的方法是在销毁左侧运算对象资源之前拷贝右侧运算对象。

```
class HasPtr{
public:
    // 对ps指向的string，每个HasPtr对象都有自己的拷贝
    HasPtr(const HasPtr& p)
        : ps(new std::string(*p.ps)), i(p.i) {}
    HasPtr& operator=(const HasPtr&);
    ~HasPtr() { delete ps; }

private:
    std::string *ps;    // 动态分配内存
    int i;
}

HasPtr& HasPtr::operator=(const HasPtr& rhs)
{
    // 错误：如果rhs和*this是同一个对象
    // 我们就将从已释放的内存中拷贝数据！
    delete ps;  // 释放对象指向的string
    ps = new string(*rhs.ps);

    // 正确：
    auto newp = new string(*rhs.ps);    // 拷贝底层string
    delete ps;                          // 释放旧内存
    i = rhs.i;
    return *this;
}
```

13.2.2 定义行为像指针的类

1. 类比 `shared_ptr`，我们为了直接管理资源，需要自己设计引用计数。
2. 将计数器保存在动态内存中。

```
class HasPtr{
public:
    // 拷贝构造函数拷贝所有三个数据成员，并递增计数器
    HasPtr(const HasPtr& p)
        : ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& Operator=(const HasPtr&);
    ~HasPtr();

private:
    std::string *ps;
    int i;
    std::size_t *use;   // 用来记录有多少个对象共享*ps的成员
}

HasPtr::~HasPtr()
{
    // 如果引用计数变为0，释放成员的动态内存
    if(--*use = 0)
    {
        delete ps;
        delete use;
    }
}

HasPtr& HasPtr::operator=(const HasPtr& rhs)
{
    // 先递增rhs中的计数然后再递减左侧运算对象中的计数，避免自赋值
    ++*rhs.use;
    if(--*use == 0)
    {
        delete ps;
        delete use;
    }

    // 逐成员拷贝
    ps = rhs.ps;
    i = rhs.i;
    use = rhs.use;
    return *this;   // 返回本(左侧)对象
}
```

13.3 交换操作

1. 编写 `HasPtr` 类自己的 `swap` 函数（通过交换成员变量指针来优化代码）。

2. 与拷贝控制成员不同， `swap` 并不是必要的。但是，对于分配了资源的类，定义 `swap` 可能是一种很重要的优化手段。

```
class HasPtr{
    // 为了访问private成员，将函数定义为friend
    friend void swap(HasPtr&, HasPtr&);
    // 其他成员...
};

// 为了优化代码将其声明为内联函数
inline void swap(HasPtr& lhs, HasPtr& rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps);    // 交换指针，而不是string数据
    swap(lhs.i, rhs.i);      // 交换int成员
}

// 假定有一个Foo类，它有一个类型为HasPtr的成员h
// 为Foo编写一个swap函数
void swap(Foo& lhs, Foo& rhs)
{
    // 错误：这个函数使用了标准库版本的swap，而不是HasPtr版本
    std::swap(lhs.h, rhs.h);

    // 正确：使用HasPtr版本的swap
    swap(lhs.h, rhs.h);

    // 交换类型Foo的其他成员
}
```

3. 定义 `swap` 的类通常用 `swap` 来定义它们的赋值运算符。使用一种名为 **拷贝并交换（copy and swap）** 的技术，将左侧运算对象与 **右侧运算对象的一个副本** 进行交换。
4. 使用拷贝和交换的赋值运算符自动就是异常安全的，且能正确处理自赋值。

```
// rhs是值传递
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // 交换左侧运算对象和局部变量rhs的内容
    swap(*this, rhs);   // rhs现在指向本对象曾经使用的内存
    return *this;       // rhs被销毁，从而delete了rhs中的指针
}
```

13.4 拷贝控制示例

1. 虽然通常来说分配资源的类更需要拷贝控制，但资源管理并不是一个类需要定义自己的拷贝控制成员的唯一原因。一些类也需要拷贝控制成员的帮助来进行薄记工作或其他操作。
2. **拷贝赋值运算符** 通常执行 **拷贝构造函数** 和 **析构函数** 中也要做的工作。这种情况下，公共的工作应该放在 `private` 的工具函数中完成。

13.5 动态内存管理类

1. 某些类需要在运行时分配可变大小的内存空间，这种类通常可以使用标准库容器来保存它们的数据。但是这一策略并不是对每个类都适用，某些类需要自己进行内存分配，这些类一般来说必须定义自己的拷贝控制成员来管理所分配的内存。
2. 类 `strVec` 将模仿 `vector<string>` 的功能。我们将使用一个 `allocator` 来获取原始内存（未构造）。在需要添加新元素时用 `allocator` 的 `construct` 成员在原始内存中创建对象。在需要删除一个元素时，我们将使用 `destroy` 成员来销毁元素。
3. 使用 `string` 的 **移动构造函数** 来避免在重新分配内存空间时因拷贝 `string` 所带来的分配和释放的额外开销， `strVec` 的性能会好得多。

13.6 对象移动

1. 在重新分配内存的过程中，从旧内存将元素拷贝到新内存是不必要的，更好的方式是移动元素。
2. 使用移动而非拷贝的另一个原因源于IO类或 `unique_ptr` 这样的类。这些类都包含不能被共享的资源（如指针或IO缓冲）。因此这些类型的对象不能拷贝但可以移动。
3. 在旧C++标准中，没有直接的方法移动对象。因此，即使不必拷贝对象的情况下，也不得不拷贝。
4. 标准库容器、 `string` 和 `shared_ptr` 类既支持移动也支持拷贝。IO类和 `unique_ptr` 类可以移动但是不能拷贝。
5. 我理解的 **对象移动的本质** 其实就是对象内存地址所有权的转移。之所以需要分出左/右值，就是由于左值对象的所有权不唯一，如果移动的是左值对象，接下来再使用左值对象的值时会出错。而右值对象的所有权唯一，当移动的是右值对象时，就应该确保接下来不会有任何地方再次调用这个右值对象（这也就是为什么要遵守即使使用 `move` 强行将左值转成了右值也不能再调用移后源对象的规定）。

13.6.1 右值引用

1. 为了支持移动操作，新标准引入了**右值引用（rvalue reference）**——必须绑定到右值的引用。通过 && 而不是 & 来获得右值引用。
2. 我们不能将**左值引用**绑定到要求转换的表达式、字面常量或返回右值的表达式。**右值引用**则相反，我们可以将其绑定在这类表达式上。但是不能将一个右值引用直接绑定到一个左值上：

```
int i = 42;
int& r = i;              // 正确：r左值引用i
int&& rr = i;            // 错误：不能将一个右值引用绑定到一个左值上
int& r2 = i * 42;        // 错误：i*42是一个右值
const int& r3 = i * 42; // 正确：可以将一个const的左值引用绑定到一个右值上
int&& rr2 = i * 42;      // 正确：将右值引用绑定到右值上
```

3. 返回左值引用的函数，连同赋值、下标、解引用和前置递增/递减运算符，都是返回左值的表达式的例子。
4. 返回非引用类型的函数，连同算数、关系、位以及后置递增/递减运算符，都生成右值。
5. 右值引用要么是字面值常量，要么是在表达式求值过程中创建的将要被销毁的临时对象。因此，右值引用的代码可以自由地接管所引用的对象的资源。
6. 变量表达式都是左值，我们不能将一个右值引用绑定到一个右值引用类型的变量上：

```
int&& rr1 = 42;      // 正确：字面常量42是右值，rr1是右值引用类型
int&& rr2 = rr1;     // 错误：虽然rr1是右值引用类型，但是rr1是左值

// 变量(rr1)是左值，因此不能将一个右值引用(rr2)直接绑定到一个变量(rr1)上，
// 即使这个变量(rr1)是右值引用类型也不行。
```

7. 虽然不能将一个右值引用直接绑定到一个左值上，但是可以通过调用 `move` 标准库函数显示地将一个左值转换为对应的右值引用类型。

```
int&& rr3 = std::move(rr1); // 正确

// 调用move就意味着对于rr1：可以销毁这个移后源对象，也可以赋予它新值，
// 但是不能使用这个移后源对象的值。
```

8. 使用 `move` 的代码应该使用 `std::move` 而不是 `move` 。这样做可以避免潜在的名字冲突。

13.6.2 移动构造函数和移动赋值运算符

1. 为了让类支持移动操作，需要为其定义**移动构造函数**和**移动赋值运算符**。
2. 除了完成资源移动，**移动构造函数**还必须确保移后源对象能够被销毁。移后源对象必须不再指向被移动的资源-此时这些资源的所有权已经归属新创建的对象。

```
StrVec::StrVec(StrVec&& s) noexcept
    // 成员初始化器接管s中的资源
    : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // 令s进入这样的状态-对其运行析构函数是安全的。
    s.elements = s.first_free = s.cap = nullptr;

    // StrVec的析构函数在first_free上调用deallocate。
    // 如果忘记了改变s.first_free,
    // 则销毁移后源对象就会释放掉我们刚刚移动的内存。
}

// noexcept 通知标准库此构造函数不抛出任何异常。
```

3. **移动赋值运算符**执行析构函数和移动构造函数相同的工作，不抛出异常，且要正确处理自赋值情况
4. 移动赋值运算符去检查自赋值情况有些奇怪，因为移动赋值运算符需要右侧运算符对象的一个右值。之所以检查的原因是此右值可能是 `move` 调用的返回结果。
5. 在移动操作之后，移后源对象必须保持有效的、可析构的状态，但用户不能对其值进行任何假设。
6. 只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非 `static` 数据成员都能移动构造或移动赋值时，**编译器**才会为它合成移动构造函数或移动赋值运算符。
7. 定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作。否则，这些成员默认地被定义为删除的。
8. **更新“三/五法则”**：一般来说，如果一个类定义了任何一个拷贝操作，它就应该定义所有的五个操作。某些类必须定义拷贝构造函数、拷贝赋值运算符和析构函数才能正确工作。这些类通常拥有一个资源，而拷贝成员必须拷贝此资源。拷贝一个资源会导致一些额外的开销。在这种拷贝并非必要的情况下，定义了移动构造函数和移动赋值运算符的类就避免此问题。
9. 一个**移动迭代器**通过**改变**给定迭代器的**解引用运算符的行为**来适配此迭代器，移动迭代器的解引用运算符生成一个右值引用。
10. 通过调用标准库的 `make_move_iterator` 函数将一个普通迭代器转换为一个移动迭代器。
11. 不要随意使用移动操作，由于一个移后源对象具有不确定的状态，对其调用 `std::move` 是危险的，当我们调用它时，必须绝对确认移后源对象没有其他用户。
12. 在移动构造函数和移动赋值运算符这些类实现代码之外的地方，只有当你确信需要进行移动操作且移动操作是安全的，才可以使用 `std::move` 。

13.6.3 右值引用和成员函数

1. 区分移动和拷贝的重载函数通常有一个版本接受一个 `const T&` ，而另一个版本接受一个 `T&&` 。

```
class StrVec {
public:
    void push_back(const std::string&); // 拷贝：绑定到任意类型的T
    void push_back(std::string&&);      // 移动：只能绑定到类型T的可修改的右值
    // ...
}

StrVec vec;
string s = "some string...";
vec.push_back(s);                // 拷贝
vec.push_back(std::move(s));     // 移动
vec.push_back("done");           // 移动
```

2. **引用限定符**：用来指出一个非 `static` 成员函数可以用于左值或右值的符号。限定符 `&` 和 `&&` 应该放在参数列表之前或者 `const` 限定符之后（如果有的话）。被 `&` 限定的函数只能用于左值；被 `&&` 限定的函数只能用于右值。

```
class string{
    void func1() &;
    void func2() &&;
    void func3() const &;
};

void string::func1() & {}
void string::func2() && {}
void string::func3() const & {}

string a = "a" , b = "b";
a.func1();                   // 左值调用成员函数
auto n = (s1 + s2).func2(); // 右值调用成员函数
```

3. 综合 `const` 和引用限定符可以区分重载版本。
4. 如果一个成员函数有引用限定符，则具有相同参数列表的所有版本都必须有引用限定符。