第13章 拷贝控制

当定义一个类时，我们显示或隐式地指定在此类型的对象的拷贝、移动、赋值和销毁时做什么

一个类通过定义五种特殊的成员函数来控制这些操作：

- 拷贝构造函数
- 拷贝赋值运算符
- 移动构造函数
- 移动赋值运算符
- 析构函数

> 统称为 **拷贝控制操作**

拷贝和移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么；

拷贝和移动赋值函数定义将一个对象赋予同类型的另一个对象时做什么；

析构函数定义了当此对象销毁时做什么；

> 实际上，大多数类都应该定义上述函数
>
> 如果没显式定义，则编译器会自动为类定义缺失的操作

拷贝、赋值与销毁

拷贝构造函数

如果一个构造函数的**第一个参数必须是自身类类型的引用**，且**任何额外的参数都有默认值**，则此构造函数是拷贝构造函数。

> 为什么必须是引用类型？
>
> 因为如果不是引用类型，那么为了调用拷贝构造函数，我们必须拷贝参数；而为了拷贝参数，就必须调用拷贝构造参数。陷入死循环。

> 注意，因为任何额外的参数都有默认值，所有拷贝构造函数是不支持多个的

```
class test2{
public:
 test2() = default;
 test2(const test2& t) {
```

```
        a = t.a;
        b = t.b;
    }
    test2(const test2& t, int c = 2) {
        a = t.a;
        b = c;
    }
private:
    int a;
    int b;
};

int main(){
    test2 a1;
    a1.setA(1);
    a1.setB(2);
    test2 a2 = a1; // 因为不确定的匹配，所以导致错误
    return 0;
}
```

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

> 拷贝构造函数可能会被隐式使用，所以不能是 `explict` 的

- 对于默认构造的拷贝函数（合成拷贝构造函数），会将其参数的成员逐个拷贝到正在创建的对象中

  > 不包括 `static` 成员

  - 每个成员的类型决定了它如何拷贝
    - 类类型----使用拷贝构造函数来拷贝；
    - 内置类型----直接拷贝
    - 数组类型----逐元素拷贝
- 拷贝初始化通常使用拷贝构造函数完成

  > **C++11** 还可能使用移动构造函数完成

```
string dot(10,'.');      // 直接初始化 ---> 实际上是通过函数匹配来调用构造函数
string s2 = dot;         // 拷贝初始化
```

- 拷贝初始化的形式：
  - 使用 `=` 定义变量
  - 函数传递非引用类型参数
  - 函数返回非引用类型
  - 用花括号列表初始化一个数组中的元素或一个聚合类[1]中的成员

> insert或push会对其元素进行拷贝初始化；
>
> emplace进行直接初始化

```
class Complex{
public:
    double real, imag;
    Complex(double r,double i){
        real = r; imag = i;
    }
    Complex(Complex &c){
        real = c.real; imag = c.imag;
        cout<<"Copy Constructor called1"<<endl ;
    }
    Complex(const Complex & c){
        real = c.real; imag = c.imag;
        cout<<"Copy Constructor called2"<<endl ;
    }
};

int main(){
    Complex c1(1, 2);
    Complex c2(c1);  // 直接构造；  注意，如果没有第二个构造函数，那么此时也是调用拷贝初始化函数
    Complex c3 = c1;    // 拷贝初始化
    return 0;
}
```

> - 因为拷贝初始化也是初始化函数，所以直接初始化也会考虑对其进行函数匹配；
>
> 而 使用 `=` 初始化对象，就表示一定是调用拷贝初始化函数

> - 因为拷贝初始化也是构造函数，所以自定义了拷贝构造函数，那么默认的构造函数也会消失，需要自己定义

拷贝赋值运算符

通过重载 `=` 运算符实现

> 重载运算符函数（14章）
>
> - 返回值是类的一个引用类型
> - 左侧对象默认绑定到隐式的this指针中

析构函数

析构函数是一个类的成员函数，名字由**波浪号+类名**构成。没有返回值，不接受参数

> 因为不接受参数，所以不支持重载。一个类只有一个析构函数

- 析构部分是隐式的。

  成员的销毁完全依赖于成员的类型

  - 类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，不需要操作

  > 隐式销毁内置指针不会delete它所指向的对象
  >
  > ```
  > class test{
  > public:
  > test(int new_a,int *new_p):a(new_a),p(new_p){};
  > int a;
  > int *p;
  > };
  > int main(){
  > int num = 1;
  > cout<<&num<<endl;        // 0x61fe1c
  > {
  >   test t1(1, &num);
  >   cout<<t1.p<<endl; // 0x61fe1c
  > }
  > cout<<num;           // 1 ，说明隐式销毁指针p时，num处在的内存并没有被释放
  > }
  > ```

- 析构函数的调用情况：

- 变量离开作用域
- 当一个对象被销毁，其成员被销毁
- 容器（库容器或数组）被销毁，其元素被销毁
- 动态分配的对象，delete指向该对象的指针时被销毁
- 对于临时对象，当创建它的完整表达式结束时被销毁；

> 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

```
int a = 1;
{
    int *p = &a;
}
cout<<a;        // 1
```

- 合成析构函数：函数一般为空就行

> 在一个析构函数中，首先执行函数体，然后再逆序销毁成员；
>
> **注意，并不是在函数体内实现销毁操作的**

三/五法则

> 自定义拷贝控制操作，必须3类一起定义。
>
> 而新标准下，则是5类一起定义

C++语言并不要求我们定义所有的类拷贝操作：可以只定义一个或两个，而不必定义所有。

当时，这些操作通常被看作一个整体，只需要定义其中一个操作而不需要其他操作的情况是很少见的

- 基本原则：
  1. 如果一个类需要自定义析构函数，那必须构造自定义拷贝赋值和构造函数；
  2. 如果一个类需要自定义拷贝构造函数，那必须构造自定义拷贝赋值函数

     > 反之，需要拷贝赋值运算符，那必然需要拷贝构造函数

- **宗旨**：**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**

阻止拷贝

如果某种类不想要拷贝或赋值操作，不定义对应的函数是没用的

> 即使不显示的定义，编译器也会隐式的定义

这个时候，我们需要定义 **删除的函数** 来组织拷贝

- 删除的函数：我们虽然定义了它们，但是不能以任何方式使用它们

```
struct NoCopy{
    NoCopy() = default;
    NoCopy(const NoCopy&) = delete;      // 组织拷贝初始化
    NoCopy &operator = (const NoCopy&) = delete;     // 阻止赋值
}
```

- `=delete` 必须出现在函数的**第一次声明**的时候；

  并且可以对任何函数指定（可以用来引导函数匹配过程）

  > `=default` 必须对编译器可以合成的默认构造函数或拷贝控制成员使用

- 析构函数是不能删除的成员
- 如果一个类有删除的或不可访问的析构函数，那么其默认和拷贝构造函数会被定义为删除的。
- 当不可能拷贝、赋值和销毁类的成员时，类的合成拷贝控制成员就被编译器定义为删除的

拷贝控制和资源管理

拷贝控制的类型

- 类的拷贝行为类似一个值：意味着拷贝之后的副本和原对象时完全独立的，该变副本不会对原对象有影响
- 类的拷贝行为类似一个指针：意味着副本和原对象使用相同的底层数据，改变副本也会改不原对象】

> 两者的区别，主要类内指针拷贝的操作

分析背景：

- `HasPtr` 类中两个数据成员：一个int，一个string指针

行为像值的类

拷贝构造函数

```
HasPtr(const HasPtr &p): ps(new string(*p.ps)), i(p.i) { };          //
注意对于指针的操作
```

拷贝赋值函数

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

- 赋值操作会销毁左侧运算对象的资源；
- 赋值操作必须支持自身赋值

> 所以这里先拷贝右侧对象而不是先删除左侧对象，是为了保证如果两者是同一个对象，先删除再访问相同内存是错误操作。
>
> 所以好的操作是，先拷贝右侧的对象、

析构函数

```
~HasPtr() { delete ps;}          // 必须显示删除指针
```

> 隐式的析构函数销毁内置指针不会删除它所指向的对象

行为像指针的类

最好的方法是使用 `shared_ptr` 来管理类中的资源

- 拷贝（或赋值）一个shared_ptr会拷贝（或赋值）shared_ptr所指向的指针

交换操作

管理资源的类通常还定义一个名为 `swap` 的函数

- 对于重排元素的算法，在交换两函数是会调用swap
- 如果一个类定义了自己的swap,那么算法将使用类自定义的版本；

  否则，将使用标准库定义的swap

> 自定义的swap可以实现指针交换，而不用多余的内存分配

如果一个类成员有自己类型特定的swap版本，我们应该调用swap而不是std::swap

```
void swap(Foo &lhs, Foo &rhs) {
    using std::swap;            // 这里声明命名空间，是防止如果没有特定版本，也能使用标准库版本，不至于不能运行
    swap(lhs.h, rhs.h);         // 假设lhs.h类型有其特定的swap版本
}
```

在赋值运算符中使用swap

- 定义swap的类通常使用swap来定义它们的赋值运算符。使用一种名为 **拷贝并交换** 的技术，将左侧对象与右侧对象的一个副本进行交换

```
HasPtr& HasPtr::operator = (HasPtr rhs) {   // 参数值传递
    swap(*this, ths);       // rhs内部成语现在指向this曾经使用过的内存
    return *this;           // rhs被销毁，从而delete了rhs的指针
}
```

- 因为在改变右侧对象之前拷贝了右侧对象保证了自赋值的正确性

拷贝控制示例(练习)

动态内存管理类

> 可能是STL底层实现vector的思想

背景：设计一个简化版本的vector---StrVec

```
/**
 * 实现 vector<string>
 */
class StrVec{
public:
    // 默认初始化
    StrVec():elements_(nullptr), cap_(nullptr), first_free_(nullptr) { }
    // 拷贝初始化
    StrVec(const StrVec&);
    // 赋值初始化
```

```
    StrVec& operator = (const StrVec &);
    // 析构函数
    ~StrVec();
    // vector操作实现
    void push_back(const string&);
    size_t size() const {
        return first_free_ - elements_;
    }
    size_t capacity() const {
        return cap_ - elements_;
    }
    string *begin() const {
        return elements_;
    }
    string *end() const {
        return first_free_;
    }

private:
    static allocator<string> alloc_;     // 分配元素
    string *elements_;       // 指向数组首元素
    string *cap_;            // 指向数组尾后元素
    string *first_free_;     // 指向数组第一个空闲位置的指针
    // 工具函数
    void CheckMem() {
        if (size() == capacity()) ReAllocate();
    }
    void ReAllocate();
    void free();
    pair<string *, string *> AllocNCopy(const string *, const string
*);
};

void StrVec::push_back(const string &s) {
    CheckMem();
    alloc_.construct(first_free_++, s);
}

void StrVec::ReAllocate() {
    // 根据vector特性，会生成当前两倍大小内存
    auto new_capacity = size() ? size()*2 : 1;
```

```
51      auto new_data = alloc_.allocate(new_capacity);
52      // 复制内容
53      auto dest = new_data;
54      auto elem = elements_;
55      for (size_t i = 0; i != size(); ++i)
56          alloc_.construct(dest++, move(*elem++));        // 调用move返回的结果会令construct使用string的移动构造函数
57      free();         // 如果移动构造函数是将内存分配给新的目录，那么还能释放原对象的内存吗？
58      elements_ = new_data;
59      first_free_ = dest;
60      cap_ = elements_ + new_capacity;
61  }
62
63  void StrVec::free() {
64      if (elements_) {
65          for (auto p = first_free_; p != elements_;)
66              alloc_.destroy(--p);
67          alloc_.deallocate(elements_, cap_ - elements_);
68      }
69  }
70
71  pair<string *, string *> StrVec::AllocNCopy(const string *b, const string *e) {
72      auto data = alloc_.allocate(e - b);
73      return {data, uninitialized_copy(b, e, data)};
74  }
75
76  StrVec::StrVec(const StrVec &s) {
77      auto newdata = AllocNCopy(s.begin(), s.end());
78      elements_ = newdata.first;
79      first_free_ = cap_ = newdata.second;
80  }
81
82  StrVec &StrVec::operator=(const StrVec &s) {
83      auto data = AllocNCopy(s.begin(), s.end());
84      free();
85      elements_ = data.first;
86      first_free_ = cap_ = data.second;
87      return *this;
88  }
```

```
89
90 StrVec::~StrVec() {
91     free();
92 }
```

- `ReAllocate()`

  在重新分配内存的过程中移动元素而不是拷贝元素

  > 因为string的行为类值，每个string对构成它的所有字符都会保存自己一份副本。拷贝一个string必须为这些字符分配空间，而销毁一个string必须释放所占用的内存。

  该函数只需要一个string，进行拷贝操作是多余的，希望能避免分配和释放string的额外开销

  - 移动构造函数：将资源从给定对象**“移动”而不是拷贝**到正在创建的对象
  - `move()`函数
    - 定义在`<utility>`中
    - 调用move来表示希望使用string的移动构造函数

      > 如果漏掉，则使用拷贝构造函数

- `free()`：

  destory()会运行string的析构函数

  deallocate()释放StrVec对象分配的资源

  > `???` 在重分配内存中使用free()不会将原对象的内存销毁，那么新对象的**移动怎么理解呢**？？

对象移动(C++11)（难点）

- 对象拷贝后就立即销毁，这个时候拷贝是不必要的。
- IO类或unique_ptr这样的类，包括不能共享的资源（指针或IO缓冲），这些对象不能拷贝但能移动

> 标准库、string和shared_ptr类支持移动和拷贝；
>
> IO类和unique_ptr可以移动但不能拷贝

右值引用——必须要绑定的右值的引用

- 通过 `&&` 来获得右值引用

```
int &&i = 43;
int &j = 43;    // 错误
```

- 只能绑定到一个将要销毁的对象；

  即，可以将右值引用绑定到左值不能绑定的对象上，但不能绑定到一个左值上

  > ◦ 左值不能绑定到要求转换的表达式、字面值常量或返回右值的表达式

  ◦ 左值持久、右值短暂

  ◦ 因为右值引用指向将要被销毁的对象，所以我们可以从绑定的右值引用的对象“窃取”状态

  > 实际上是将所有权转让出去了，这样就不用作额外的拷贝操作，直接转移

- `std::move()`：将一个左值声明为右值使用

  > 需要声明命名空间，不然容易调用其他函数

```
std::string s =" hell world";
auto left = std::move(s);
cout<<s;        // 输出为空
s = "s";
cout<<s;        // 输出s

-----------------------------------;

    int p = 1;
    auto &&p2 = std::move(p);
    std::cout<<p;       // ????  为什们没有影响，正常输出 1
```

  ◦ 移动以后(调用移动函数以后)，原来变量s指代的值的所有权**转移**到了left，这个变量s被改变成了“空值”状态；

  可以销毁s，或赋予它新值，但不能够使用它的值

什么是move？理解C++ Value categories，move， move in Rust

移动构造函数(细看)

- 第一个参数是该类类型的右值引用，任何其他的参数都要有默认实参
- 要确保移动源对象处于 **销毁它是无害** 的这样一个状态

```
StrVec::StrVec(StrVec &&s) noexcpet : elements(s.elements),
first_free(s.first_free), cap(s.cp) {
s.elements = s.first_free = s.cap = nullptr;
}
```

- 移动构造函数不分配任何新内存；它先将传递参数的内存地址空间接管，后将内部所有指针设置为nullptr，并且在原地址上进行新对象的构造，最后调用原对象的的析构函数，这样做既不会产生额外的拷贝开销，也不会给新对象分配内存空间

> 将rhs指针置空原因：
>
> 如果不为空，则源对象销毁时，析构函数在first_free上调用deallocate，会释放掉我们刚移动内存

> **如何理解呢?**
>
> 我直接用左值引用，然后进行指针的赋值，再销毁参数对象，感觉也没有进行拷贝操作呀?

- `noexcept` ——移动操作(移动构造、移动赋值函数）通常不会抛出任何异常

> 告知标准库，减少标准库为处理异常所做的额外的工作

  - 必须在类头文件的声明和定义中都指定noexcept
  - 移动操作通常不抛出异常，但抛出异常也是允许的；

    标准库容器能对异常发生时其自身的行为提供保障；

    拿vector.push_back()为例，标准库确保发生异常时，vector自身不会变化。
    如果是使用移动构造函数，如果在移动中途抛出异常，因为旧空间的移动源元素已经改变，而新空间中未构造的元素可能尚不存在，所以无法保证vector不变；
    如果是使用拷贝函数，因为拷贝过程中，源对象始终不变，所以即使出现异常，也能保持不变
    所以，除非vector知道元素类型的移动构造函数不会抛出异常，否则在重新分配内存的过程中，它就必须使用拷贝构造函数而不是移动构造函数。
    如果希望vector重新分配内存的这类情况下对我们自定义类型的对象进行移动而不是拷贝，则必须要声明noexcept

移动赋值运算符

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept {
    if (this != &rhs) { // 检查自赋值，相同则不用处理
        free();
        elements = rhs.elements;
        first_free = rhs.first.free;
        cap = rhs.cap;
        // 将rhs置于可析构的状态
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
}
```

移后源对象必须可析构

> 在移动操作后，移后源对象必须保持有效的、可析构的状态，但是用户不能对其值进行任何假设

- 对象有效是指可以安全地为其赋予新值，或可以安全地使用而不依赖其当前值
- 移动操作对移后源对象中留下的值没有任何要求。程序不应依赖于移后源对象的数据

合成的移动操作

- 只有当一个类中没有定义 **任何** 自己版本的拷贝控制成员，且类的每个非static数据成员都可以移动时，

  编译器才会为它合成移动构造函数或移动赋值函数

  - 编译器可以移动内置类型的成员
  - 如果成员是一个类类型，且该类有对象的移动操作，则编译器也能移动这个成员
- 如果类定义了一个移动构造/赋值函数，则该类合成的拷贝构造函数和拷贝赋值函数会被定义为删除的

移动操作永远不是隐式定义为删除的函数

> **暂略**

移动构造函数和拷贝构造函数按普通的函数匹配规则调用

- 左值拷贝、右值被移动
- 如果没有移动构造函数，拷贝构造函数也可以替代其操作

  > 仍然可以理解为拷贝函数的重载

可以为成员函数同时提供拷贝和移动的b版本

```
void push_back(const X &);
void push_back(X &&);

v.push_back(s);      // 拷贝版本
v.push_back("123"); // 移动版本
```

移动迭代器

- 移动迭代器的解引用返回生成一个右值引用
- 调用标准库函数 `make_move_iterator` 函数将一个普通迭代器转换为一个移动迭代器
    - 原迭代器的所有操作在移动迭代器中都照常运算

> 标准库不保证哪些算法适用于移动迭代器，哪些不适用。
>
> 由于移动一个对象可能会销毁原对象，只有在你确信算法在为一个元素赋值后或传递该函数后不再访问，才可使用
>
> 不要随意使用移动操作

右值和左值引用成员函数

```
std::string s = "123";
std::string s2 = "4";
s + s2 = "123";      // 新旧标准库都允许向一个右值赋值
```

- 我们可以通过 在参数列表后放置一个引用限定符 来强制左侧对象是一个左值

    ```
    Foo &operator=(const Foo&) & {
        ...
    }
    ```

    - `&` 和 `&&` 分别指出this指向一个左值或右值
    - 引用限定符只能用于（非static）成员函数，且必须同时出现在函数的声明和定义中
- 同时用于const和引用限定，则引用限定符必须跟在const限定符后面

    ```
    Foo someMem() const &;
    ```

- 引用限定符可以区分重载版本

> 如果一个成员函数有引用限定符，那么和具有相同参数列表的所有同名版本都必须有引用限定符

---

1. == struct ↩