@阿西拜-南昌

拷贝控制

C++ Primer第五版　　第13章

q

一个类通过定义五种特殊的成员函数来控制对象的拷贝、移动、赋值和销毁操作。

- 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

这些操作称为：拷贝控制操作

如果一个类没有定义这些操作，编译器会自动合成缺失的操作

拷贝、赋值与销毁

拷贝构造函数

const 类 &

```
//拷贝构造函数：第一个参数是自身类型的引用，额外参数都有默认值
class Foo {
pblic:
      Foo();
      Foo(const Foo&); //拷贝构造函数
      //...
};   Foo f(Foo f) {return  f;} 可以 =  ()  return 使用
```

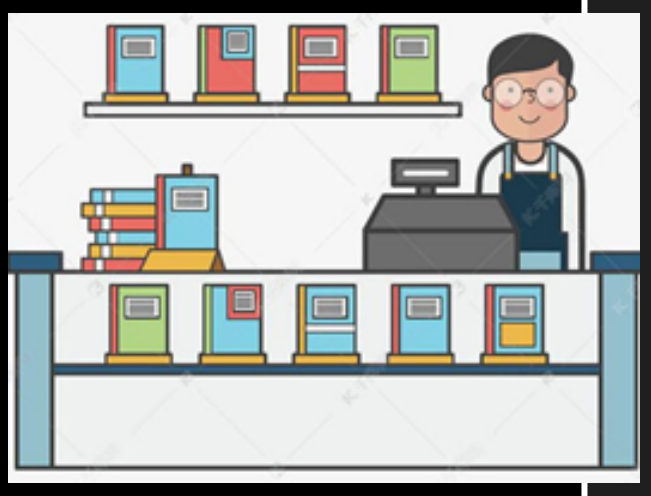
& 是为了防止拷贝构造函数陷入死循环

拷贝构造函数通常不应该是explicit的

如果加上就会报错 explict 抑制隐式转换

合成拷贝构造函数

```
//Sales_data类的合成拷贝构造函数等价于：
class Sales_data{
public:
      //其他成员与构造函数的定义，如前
      //与合成的拷贝构造函数等价的拷贝构造函数的声明
      Sales_data(const Sales_data&);
private:
      std::string bookNo;
      int units_sold = 0;
      double revenue = 0.0;
};
                  合成的拷贝构造函数
Sales_data::Sales_data(const Sales_data &orig):
            bookNo(orig.bookNo),//使用string的拷贝构造函数
            unitis_sold(orig.units_sold),
            revenue(orig.revenue)
{ }
```



拷贝初始化


```
string dots(10,'.'); //直接初始化
string s(dots); //直接初始化
string s2 = dots; //拷贝初始化
string null_book = "9-999-99999-9"; //拷贝初始化
string nines = string(100,'9'); //拷贝初始化
```

拷贝构造函数用来初始化非引用类类型参数，所以自己的参数必须是引用类型

可以隐式类型转换

拷贝初始化的限制

```
explicit vector (size_type n);
vector (const vector& x);
```

```
vector<int> v1(10); //正确：直接初始化
vector<int> v2 = 10; //错误：接受大小的构造含是explicit的
void f(vector<int>); //f的参数进行拷贝初始化
f(10);  //错误：不能用一个explicit的构造函数来拷贝一个实参
f(vector<int>(10)); //正确：从一个int直接构造一个临时vector
```

编译可以绕过拷贝构造函数

```
string null_book = "9-999-99999-9";//拷贝初始化
//改写为
string null_book("9-999-99999-9"); //编译器略过了拷贝构造函数
```

拷贝赋值运算符

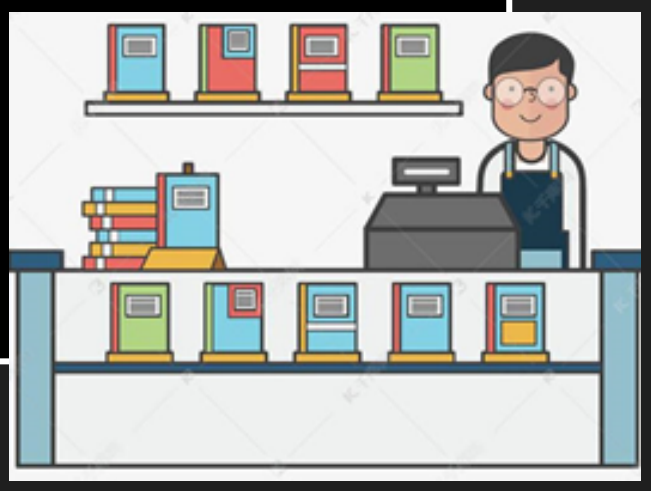
```
Sales_data trans, accum;
trans = accum; //使用Sales_data的拷贝赋值运算符

//拷贝赋值运算符接受一个与其所在类相同类型的参数
class Foo{
public:
	Foo& operator=(const Foo&); //赋值运算符
}
```

合成拷贝赋值运算符

系统帮我们合成的

```
//等价于合成拷贝赋值运算符
Sales_data& Sales_data::operator=(const Sales_data &rhs)
{
	bookNo=rhs.bookNo;//调用string::operator=
	units_sold=rhs.units_sold;//使用内置的int赋值
	revenue=rhs.revenue;//使用内置的double赋值
	return *this; //返回一个此对象的引用
}
```



与构造函数一样 都是对非 static数据成员进行操作的 @阿西拜-南昌

析构函数：不接收参数，不允许重载

它 没有返回值 也不接收参数

```
//构造函数初始化对象的非static数据成员
//析构函数释放对象的使用资源，并销毁对象的非static数据成员
class Foo{
    ~Foo();//析构函数
    //...
};
```

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

一个对象被销毁（离开作用域 ...），就会自动调用其构造函数

下面代码段定义了四个Sales_data对象：

```
{//新作用域
    //p和p2指向动态分配的对象
    Sales_data *p = new Sales_data;          //p是一个内置指针
    auto p2 = make_shared<Sales_data>(); //p2是一个shared_ptr
    Sales_data item(*p);                 //拷贝构造函数将*p拷贝到item中
    vector<Sales_data> vec;              //局部对象
    vec.push_back(*p2);                  //拷贝p2指向的对象
    delete p;                            //对p指向的对象指向析构函数
}//退出局部作用域；对item、p2和vec调用析构函数
//销毁p2会递减其引用计数；如果引用计数变为0，对象被释放
//销毁vec会销毁它的元素
```

当指向一个对象的引用或指针离开作用域时，不会执行析构函数

所以需要自己去delete回收

合成析构函数

智能指针是一个类 它具有自己的析构函数.

```
//等价于Sales_data的合成析构函数
class Sales_data{
public:
    //成员会被自动销毁，除此之外不需要做其他事情
    ~Sales_data(){ }
    //在析构函数体执行完毕后，成员会被自动销毁

    //其他成语的定义，如前
};
```

编译器默认生成的

需要析构函数的类也需要拷贝和赋值操作

```
class HasPtr{          回收指针
public:
      HasPtr(const std::string &s = std::string()):
            ps(new std::string(s)),i(0){ }

       ~HasPtr(){ delete ps; }
        //错误：HasPtr需要一个拷贝构造函数和一个拷贝赋值运算符

        //其他成员的定义，如前
};

HasPtr f(HasPtr hp) //HasPtr是传值参数，所以将被拷贝
{
      HasPtr ret = hp; //拷贝给定的HasPtr
      //处理ret
      return ret; //ret和hp被销毁  会调用俩次析构  delete 俩次  程序报错
}

HasPtr p("some values");
f(p);//f结束时，p.ps指向的内存被释放
HasPtr q(p);  //现在p和q都指向无效内存
```

需要拷贝操作的类也需要赋值操作，反之亦然：

- 考虑一个类为每个对象分配一个唯一的ID
  - 需要自定义拷贝构造函数，和赋值运算操作符
  - 不需要自定义一个析构函函数

使用=default可以显示地要求编译器生成合成的版本

```
class Sales_data{
public:
      //拷贝控制成员；使用default
      Sales_data() = default;
      Sales_data(const Sales_data&) = default;
      Sales_data& operator = (const Sales_data &);
      ~Sales_data() = default;
//其他成员的定义，如前
};
Sales_data& Sales_data::operator=(const Sales_data&) = default;//非内联
//我们只能对具有合成版本的成员函数使用=default
```

阻止拷贝　不能够delete 析构函数

例如，iostream类阻止拷贝，以免多个对象写入或读取相同的IO缓冲

```
//=delete通知编译器，我们不希望定义这些成员
//删除的函数（deleted function）
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; //阻止拷贝
        ~NoCopy() = default; //使用合成的析构函数
        //其他成员
};
```

与=default不同，=delete必须出现在函数第一次声明的时候

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; //我们不能销毁NoDTor类型的对象
};

NoDtor nd; //错误
NoDtor *P = new NoDtor(); //正确
delete p; //错误
```

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。

在新标准发布之前，阻止拷贝时通过声明为private来实现的：

```
class PrivateCopy{
        //无访问说明符；接下来的成员默认为private的；
        //拷贝控制成员是private的，因此普通用户代码无法访问
        PrivateCopy(const PrivateCopy&);
        PrivateCopy &operator = (const PrivateCopy&);
        //其他成员
public:
        PrivateCopy() = default; //使用合成的默认构造函数
        ~PrivateCopy();  //用户可以定义此类型的对象，但无法拷贝他们
};
```

友元和成员函数仍旧可以拷贝对象

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

拷贝控制和资源管理

两种选择：定义拷贝操作，使类的行为看起来像一个值或者像一个指针

行为像值的类

对于管理的资源，每个对象都应该拥有一份自己的拷贝

```
class HasPtr{
public:
    HasPtr(const std::string &s = std::string()):
                        ps(new std::string(s)),i(0){ }
    //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;
    int i;
};

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps);//拷贝底层string
    delete ps; //释放就内存
    ps =  newp;  //从右侧运算对象拷贝数据到本对象
    i = rhs.i;
    return *this; //返回本对象
}
```

new一个新的 与原先的无关

- 如果将一个对象赋予它自己，赋值运算符必须能正确工作
- 大多数赋值运算符组合了析构函数和拷贝构造函数的工作

当成员中有指针的时候

这三步顺序不能错

删除this.p 的内存

下面的赋值运算符是错误的！

```
HasPtr& HasPtr::operator=(const HasPtr &rhs){
    delete ps; //释放对象指向的string
    //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！
    ps = new string(*rhs.ps);
    i = rhs.i;
    return *this;
}
```

定义行为像指针的类

我们这里不使用shared_ptr，而是设计自己的引用计数

```
class HasPtr{
public:
    //构造函数分配新的string和新的计数器，将计数器置为1
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)),i(0),use(new std::size_t(1)){ }
    //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()
{                    * (this.use)
    if(--*use==0){          //如果引用计数变为0
        delete ps;          //释放string内存
        delete use;         //释放计数器内存
    }
}

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++*rhs.use;  //递增右侧运算对象的引用计数
    if(--*use == 0){ //然后递减本对象的引用计数
        delete ps;
        delete use;          如果为0 就虚构
    }
    ps = rhs.ps;
    i = rhs.i;
    use = rhs.use;
    return * this;
}
```

交换操作


```
class HasPtr{
public:
      HasPtr(const std::string &s = std::string()):
                                    ps(new std::string(s)),i(0){ }
      //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;
      int i;
};
```

假设需要交换两个HasPtr对象,v1和v2

```
HasPtr temp =  v1;  //创v1的值的一个零时副本
v1 = v2;
v2 = temp;
```

我们希望交换指针，而不是分配string的新副本

```
string *temp = v1.ps; //为v1.ps中的指针创建一个副本
v1.ps = v2.ps;
v2.ps = temp;
```

编写我们自己的swap函数

```
class HasPtr{
      friend void swap(HasPtr&,HasPtr&);
      //其他成员定义...
};
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,rihs.i);       //交换int成员
}
```

swap函数应该调用swap，而不是std::swap

```
//假定类Foo有HasPtr的成员h
//下面的代码能够正常运行，但性能...
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);
    //交换类型Foo的其他成员
}

//正确的swap函数
void swap(Foo &lhs, Foo &rhs)
{
    using std::swap;
    swap(lhs.h,rhs.h); //使用HasPtr版本的swap
    //交换类型Foo的其他成员
}
```

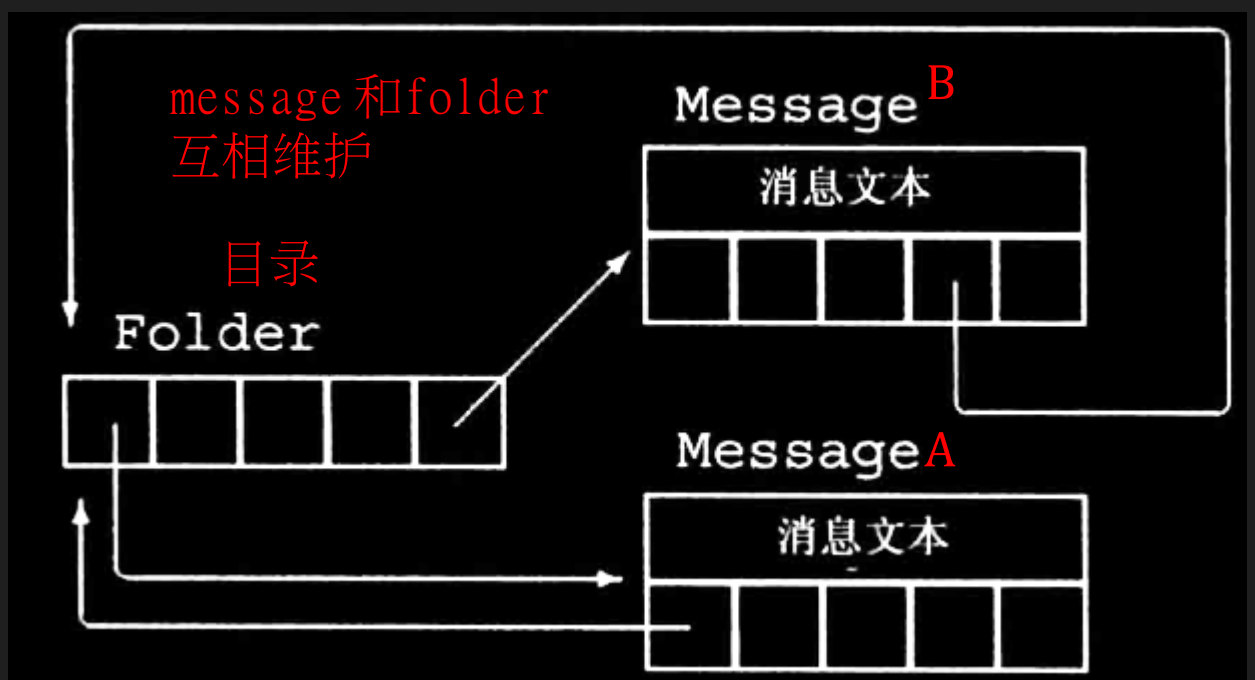
如果有专属的版本就用专属的
没有就用标准库的

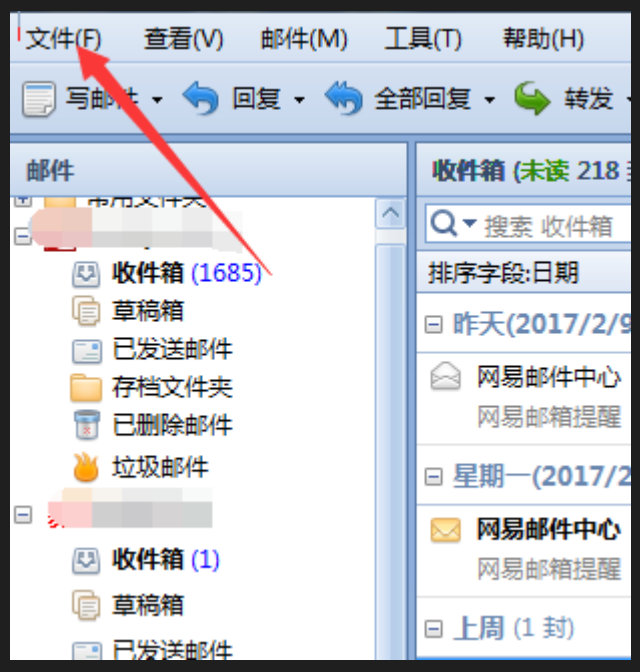
在赋值运算符中使用swap

```
//注意rhs是按值传递的，意味着HasPtr的拷贝构造函数
//将右侧运算对象中的string拷贝到rhs
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
}
```

拷贝控制示例

- 资源管理并不是定义自己的拷贝控制成员的唯一原因
- 一些类需要拷贝控制成员的帮助来进行簿记工作或其他操作





邮件系统示例：Message 和 Folder 类的设计

假设Folder类包含名为addMsg和remMsg的成员，分别完成添加和删除消息

```
class Message {
friend class Folder;
public:
    //folders被隐式的初始化为空集合
    explicit Message(const std::string &str = ""):contents(str){ }
    //拷贝控制成员，用来管理指向本Message的指针
    Message(const Message&);                    //拷贝构造函数
    Message& operator = (const Message&);       //拷贝赋值运算符
    ~Message();                                 //析构函数
    //从给定Folder集合中添加/删除本Message
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents;                       //实际消息文本
    std::set<Folder*> folders;                  //包含本Message的Folder
    //拷贝构造函数、拷贝赋值运算符和析构函数所使用的工具函数
    void add_to_Folders(const Message&);
    //从folders中的每个Folder中删除本Message
    void remove_from_Folders();
}
```

默认构造函数　也可是一个接收string的

save和remove成员

```
void Message::save(Folder &f)
{
	folders.insert(&f); //将给定Folder添加到我们的Folder列表中
	f.addMsg(this); //将本Message添加到f的Message集合中
}

void Message::remov(Folder &f)
{
	folders.erase(&f); //将给定Folder从我们的Folder列表中删除
	f.remMsg(this); //将本Message从f的Message集合中删除
}
```

Message的类拷贝控制成员

```
//将本Message添加到指向m的Folder中
void Message::add_to_Folders(const Message &m)
{
	for(auto f:m.folders) //对每个包含m的Folder
		f->addMsg(this); //向该Folder添加一个指向本Message的指针
}

Message::Message(const Message &m):contents(m.contents),folders(m.folders)
{
	add_to_Folders(m); //将本消息添加到指向m的Folder中
}
```

Message的析构函数

```
//从对应的Folder中删除本Message
void Message::remove_from_Folders()
{
	for(auto f:folders)	//对folders中每个指针
		f ->remMsg(this); //从该Folder中删除本Message
}

Message::~Message()
{
	remove_from_Folders();
}
```

Message的拷贝赋值运算符

```
Message& Message::operator=(const Message &rhs)
{
    //通过先删除指针再插入它们来处理自赋值的情况
    remove_from_Folders();          //更新已有Folder
    contents = rhs.contents;        //从rhs拷贝消息内容
    folders = rhs.folders;          //从rhs拷贝Folder指针
    add_to_Folders(rhs);            //将本Message添加到哪些Folder中
    return *this;
}
```

Message的swap函数

```
void swap(Message &lhs, Message &rhs)
{
    using std::swap; //是个好习惯
    //将每个消息的指针从它（原来）所在的Folder中删除
    for(auto f:lhs.folders)
        f->remMesg(&lhs);
    for(auto f:rhs.folders)
        f->remMesg(&rhs);
    //交换contents和Folder指针set
    swap(lhs.folders, rhs.folders); //使用swap(set&,set&)
    swap(lhs.contents,rihs.contents); //swap(string&,string&)
    //将每个Message的指针添加到它的（新）Folder中
    for(auto f:lhs.folders)
        f->addMsg(&lhs);
    for(auto f:rhs.folders)
        f->addMsg(&rhs);
}
```

lhs 和 rhs 已经互换了

动态内存管理内

- 实现标准库vector类的一个简化版本：strVec
- 每个StrVec有三个指针成员指向其元素所使用的内存：


```
//类vector类内存分配策略的简化实现
class StrVec{
public:
    //allocator成员进行默认初始化
    StrVec():elements(nullptr),first_free(nullptr),cap(nullptr){ }
    StrVec(const StrVec&);
    StrVec &operator=(const StrVec&);
    ~StrVec();
    void pubsh_back(const std::string&); //拷贝元素
    size_t size() const { return first_free - elements; }
    size_t capacity() cosnt { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
      //...
private:                    分配空间的
    static  std::allocator<std::string> alloc; //分配元素
    //被添加元素的函数所使用
    void chk_n_alloc(){ if(size() == capacity()) reallocate(); }
    //被拷贝构造函数、赋值运算符和析构函数所使用
    std::par<std::string*,std::string*> alloc_n_copy
        (const std::string*,const std::string*);
    void free();                //销毁元素并释放内存
    void reallocate();          //获得更多内存并拷贝已有元素
    std::string *elements;              //指向数组首元素的指针
    std::string *first_free;            //指向数组第一个空闲元素的指针
    std::string *cap;                   //指向数组尾后位置的指针
};
```

使用construct

```
void StrVec::push_back(const string& s){
        chk_n_alloc();                          //确保有空间容纳新元素
        alloc.construct(first_free++,s); //在first_free指向的元素中构建s的副本
}
```

alloc_n_copy成员

```
pair<string*,string*> StrVec::allc_n_copy(const string *b,const string *e){
        //分配空间保存给定范围中的元素，分配的空间恰好容纳给定的元素
        auto data = alloc.allocate(e-b);
        //初始化并返回一个pair，该pair由data和unitialized_copy的返回值构成
        return {data,uninitialized_copy(b,e,data)};
}
```

free成员

```
void StrVec::free(){
        //不能传递给deallocate一个空指针，如果elements为0，函数什么也不做
        if(elements){
                for(auto p = first_free; p!=elements; /*空*/)
                        alloc.destroy(--p); //逆序销毁救援时
                alloc.deallocate(elements,cap-elements);
        }
}
```

拷贝控制成员

```
StrVec::StrVec(const StrVec &s){
        //调用alloc_n_copy分配空间以容纳与s中一样多的元素
        auto newdata = alloc_n_copy(s.begin(),s.end());
        elements = newdata.first;
        first_free = cap =newdata.second; //分配的空间恰好容纳给定的元素
}

StrVec::~StrVec(){ free(); }

StrVec &StrVec::operator = (const StrVec &rhs){
        //调用alloc_n_copy分配内存，大小与rhs中元素占用空间一样多
        auto data = alloc_n_copy(rhs.begin(),rhs.end());
        free();
        elements = data.first;
        first_free = cap = data.second;
        return *this;
}
```

在重新分配内存的过程中移动而不是拷贝元素


```
void StrVec::reallocate()
{
      //我们将分配当前大小两边的内存空间
      auto newcapacity = size()?2*size():1;
      //分配新内存
      auto newdata = alloc.allocate(newcapacity);
      //将数据从旧内存移动到新内存
      auto dest = newdata;                  //指向新数据中下一个空闲位置
      auto elem = elements;                 //指向旧数据中下一个元素
      for(size_t i = 0; i != size(); ++i)
            alloc.construct(dest++,std::move(*eleme++));
      free();  //一旦我们移动完元素就释放旧内存空间
      elements = newdata;
      first_free = dest;
      cap = elements + newcapacity;
}
```

对象移动

- 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类可以移动但不能拷贝

右值引用

- 通过&&而不是&来获得右值引用
- 只能绑定到一个将要销毁的对象

左值持久：对象的身份
右值短暂：对象的值

```
int i = 42;
int &r = i;                 //正确：r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：我们将一个const的引用绑定到一个右值上
int &&rr2 = i*42;           //正确：将rr2 绑定到乘法结果上
```

变量是左值，即使这个变量是右值引用

```
int &&rr1 = 42; //正确：字面常量时右值  检
int &&rr2 = rr1; //错误：表达式rr1是左值
```

标准库move函数

```
//move告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它
int &&rr3 = std::move(rr1); //ok      偷
```

移动构造函数和移动赋值运算符

类似对应的拷贝操作，但它们从给定对象“窃取”资源而不是拷贝资源

不抛出异常的移动构造函数和移动赋值运算符必须标记为noexcept

相当于捡枪

```
//移动构造函数
StrVec::StrVec(strVec &&s) noexcept //移动操作不应抛出任何异常
    //成员初始化器接管s中的资源
    :elements(s.elements),first_free(s.first_free),cap(s.cap)
{                                   拷贝三个指针
    //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;
}   把原有的指针虚构掉  防止影响this指针
```

拷贝：不影响原容器，move会影响原容器，无法复原
移动本身就是存在风险的
所以需要noexcept去告诉编译器

移动构造函数不分配新内存

最终，移后源对象会被销毁，如果我们忘记了改变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

移动赋值运算符


```
StrVec &StrVec::operator = (StrVec &&rhs) noexcept
{
    //直接检测自赋值
    if(this != &rhs){
        free(); //释放已有元素
        elements = rhs.elements; //从rhs接管资源
        first_free = rhs.first_free;
        cap = rhs.cap;
        //将rhs置于可析构状态
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
    return *this;
}
```

移动操作后，移后源对象必须保持有效、可析构的状态。

合成的移动操作

只有当一个类没有定义任何自己版本的拷贝控制成员，且他们的所有数据成员都能移动构造或移动赋值时，编译器才会为他们合成移动构造函数或移动赋值运算符

```
//编译器会为X和hasX合成移动操作
struct X{
    int i; //内置类型可以移动
    std::string s; //string定义了自己的移动操作
};
struct hasX{
    X mem; //X有合成的移动操作
};
X x, x2 = std::move(x);                 //使用合成的移动构造函数
hasX hx, hx2 = std::move(hx);           //使用合成的移动构造函数
```

假定Y是一个类，定义了拷贝构造函数但未定义移动构造函数

```
struct hasY{
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; //hasY将有一个删除的移动构造函数
};
hasY hy, hy2 = std::move(hy); //错误：移动构造函数是删除的
```

移动右值，拷贝左值……

```
//移动构造函数、拷贝构造函数同时存在，编译器使用普通的函数匹配规则
//赋值操作的情况也类似
StrVec v1, v2;
v1 =  v2;                               //v2是左值；使用拷贝赋值
StrVec getVec(istream &);  //getVec返回一个右值
v2 = getVec(cin);                  //getVec(cin)是一个右值；使用移动赋值
```

如果没有移动构造函数，右值也被拷贝

```
class Foo{
public:
      Foo() = default;
      Foo(const Foo&); //拷贝构造函数
      //其他成员定义，但Foo未定义移动构造函数
};
Foo x;
Foo y(x);                       //拷贝构造函数；x是一个左值
Foo z(std::move(x));            //拷贝构造函数，因为未定义移动构造函数
```

如果有拷贝构造函数，但没有定义移动构造函数，那么这个类就没有移动构造函数

const & == &&

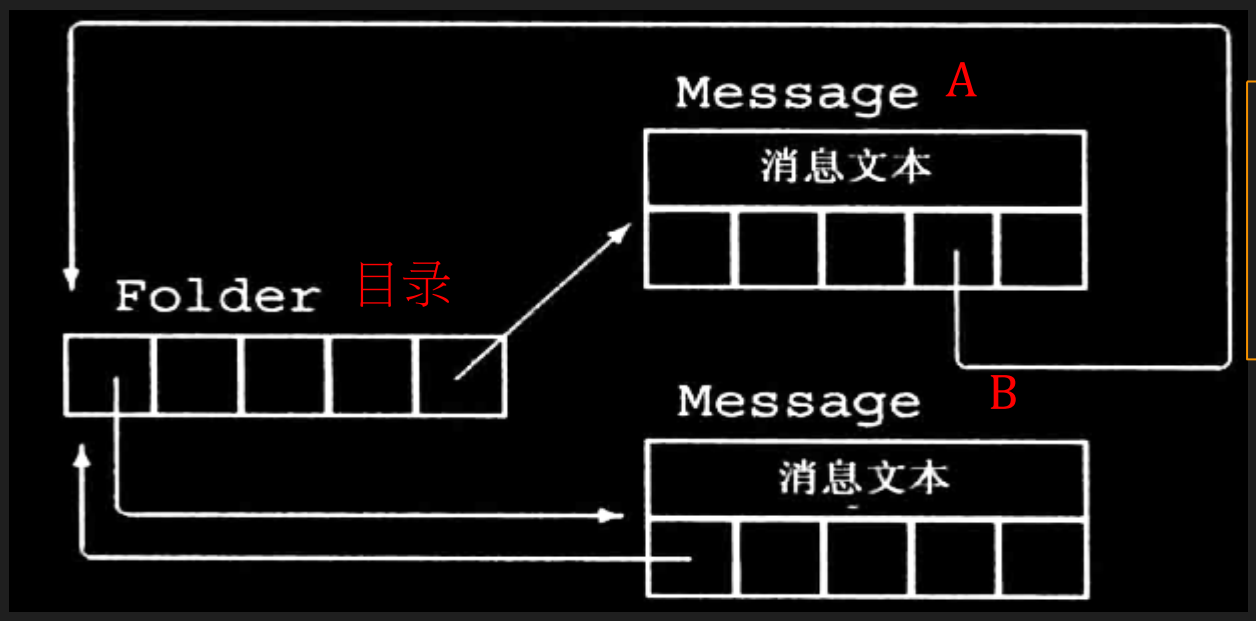
拷贝并交换赋值运算符和移动操作

```
class HasPtr{
public:
      //添加的移动构造函数
      HasPtr(HasPtr &&p) noexcept: ps(p.ps),i(p.i){p.ps = 0;}
      //赋值运算符即是移动赋值运算符，也是拷贝赋值运算符
      HasPtr& operator=(HasPtr rhs)
            { swap(*this,rhs); return *this;}

      //其他成员函数的定义...
}

//假定hp和hp2都是HasPtr对象
hp =  hp2;                      //hp2是一个左值；hp2通过拷贝构造函数来拷贝
hp = std::move(hp2);            //移动构造函数移动hp2
```

Message类的移动操作



通过定义移动操作，Message类可以使用string和set的移动操作来避免拷贝contents和folders成员的额外开销

移动构造函数和移动赋值运算符都需要更新Folder指针

```
//从本Message移动Folder指针
void Message::move_Folders(Message *m)
{
	folders = std::move(m->folders); //使用set的移动赋值运算符
	for( auto f:folders){
		f->remMsg(m);	//从Folder中删除旧Message
		f->addMsg(this);  //将本Message添加到Folder中
	}
	m->folders.clear();	//确保销毁m是无害的
}
```

Message的移动构造函数调用move来移动contents，并默认初始化自己的folders成员

```
Message::Message(Message &&m): contents(std::move(m.contents))
{
	move_Folders(&m); //移动folders并更新Folder指针
}
```

移动赋值运算符直接检测自赋值情况

```
Message& Message::operator=(Message &&rhs)
{
	if(this!=&rhs){
		remove_from_Folders();
		contents = std::move(rhs.contents); //移动赋值运算符
		move_Folders(&rhs); //重置Folders指向本Message
	}
	return *this;
}
```

移动迭代器

- 移动迭代器的解引用运算符生成一个右值引用
- make_move_iterator函数将一个普通迭代器转换为移动迭代器

```
void StrVec::reallocate()
{
    //分配大小两倍于当前规模的内存空间
    auto newcapacity = size() ? 2*size() : 1;
    auto first = alloc.allocate(newcapacity);
    //移动元素
    auto last = uninitialized_copy(make_move_iterator(begin()),
                make_move_iterator(end()),first);
    free(); //释放旧空间
    elements = first;
    first_free = last;
    cap = elements + newcapacity;
}
```

右值引用和成员函数

```
class StrVec{
public:
    void push_back(const std::string&);      //拷贝
    void push_back(std::string&&);           //移动
    //其他成员定义，如前
};

void StrVec::push_back(const string&s)
{
    chk_n_alloc(); //确保有空间容纳新元素
    //在first_free指向的元素中构造s的一个副本
    alloc.construct(first_free++,s);
}

void StrVec::push_back(string &&s)
{
    chk_n_alloc();//如果需要的话为StrVec重新分配内存
    alloc.construct(first_free++,std::move(s));
}

StrVec vec; //空Strvec
string s = "some string or another";
vec.push_back(s); //拷贝
vec.push_back("done"); //移动
```

右值和左值引用成员函数，引用限定符可以是&或&&　　@阿西拜-南昌

```
string s1 = "a value", s2 = "another";
auto n = (s1 + s2).find('a');

s1 + s2 = "wow!";

//阻止对一个右值进行赋值
//引用限定符
class Foo {
public:
	Foo &operator=(const Foo&) &; //只能向可修改的左值赋值
	//...
};
Foo &Foo::operator=(const Foo &rhs) &
{
	//...
	return *this;
}
```

```
Foo &retFoo();	//返回一个引用；retFoo调用是一个左值
Foo retVal();	//返回一个值；retVal调用是一个右值
Foo i,j;	//i和j是左值
i=j;	//正确
retFoo() = j;	//正确
retVal() = j;	//错误
i = retVal();	//正确

//与const一起使用必须在const限定符之后
class Foo{
public:
	Foo someMem() & const;	//错误：const限定符必须在前
	Foo anotherMem() const &;	//正确
};
```

```
class Foo{
public:
	Foo sorted() &&; //可用于可改变的右值
	Foo sorted() const &; //可用于任何类型的Foo
	//Foo的其他成员的定义
private:
	vector<int> data;
};

//本对象为右值，因此可以原址排序
Foo Foo::sorted() &&
{
	sort(data.begin(),data.end());
	return *this;
}
//本对象是const或是一个左值，哪种情况我们都不能对其进行原址排序
Foo Foo::sorted() const &{
	Foo ret(*this); //拷贝一个副本
	sort(ret.data.begin(),ret.data.end()); //排序副本
	return ret;
}

retVal().sorted(); //&&
retFool().sorted(); //&
```

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

```
class Foo{
public:
	Foo sorted() &&;
	Foo sorted() const; //错误：必须加上引用限定符

	using Comp = bool(const int&,const int&);
	Foo sorted(Comp*); //正确：不同的参数列表
	Foo sorted(Comp*) const; //正确：两个版本都没有引用限定符
};
```