类 进阶学习

拷贝与赋值

1. 类的拷贝控制操作：拷贝构造函数、拷贝赋值运算符、移动构造函数、移动赋值运算符和析构函数。这些操作若类没有定义，则编译器会采用默认的方式定义。
2. 拷贝构造函数：声明参数为const引用。**声明为引用是必须的**，因为为了调用拷贝构造函数，我们必须拷贝它的实参，为了拷贝实参，我们又需要调用拷贝构造函数，若参数不是引用，则会无限循环下去。
3. 拷贝初始化的发生情景：实参传递到非引用形参；函数返回非引用类型时；**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**
4. 在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象。即允许改写下面代码：

```
string str="ddd";
//改写为
string str("ddd");
```

   尽管如此，我们还是需要定义可以访问的拷贝/移动构造函数
5. 如果一个运算符是一个成员函数，则其左侧运算对象就绑定到隐式的this参数。
6. **赋值运算符通常返回一个指向其左侧运算对象的引用**。标准库通常要求保存在容器中的类型要有赋值运算符且返回值是左侧运算对象的引用。
7. 可以将拷贝构造函数定义为=default来显式地要求编译器生成合成的函数。当在类内使用=default时合成的函数将隐式地声明为内联的；当在类外定义使用default时则不是内联的。**只能对具有合成版本的成员函数使用=default**

析构函数

1. 构造函数初始化对象的**非static数据成员**；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. 析构函数没有返回值，也不接受参数，因此不能被重载。形式如下：

```
class Test{
    public:
        ~Test(){};
};
```

3. 构造函数中，成员的初始化是在函数体执行之前完成的（初始化列表），且**按照它们在类中出现的顺序进行初始化（而不是在列表中的顺序）**；析构函数则首先执行函数体，然后销毁成员，**成员按照初始化顺序的逆序销毁**。
4. 成员销毁时发生什么完全依赖于成员的类型。被销毁的成员需要执行自己类型的析构函数。不过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。因此，**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**。如下面代码：

```
class A{
    public:
        A():p(nullptr){ cout<<"构造A\n";}
        A(int a){p=&a;cout<<"构造A\n";}
        ~A(){cout<<"析构A\n";}
    private:
        int *p;
};

int main() {
    int a=5;
    int *p=&a;
    {
        A test(a);
    }
    cout<<a<<' '<<*p;
}
```

代码输出为：构造A \n 析构A 5 5
可见对象a并没有被销毁。
由于智能指针是类类型，具有析构函数，所以在析构阶段智能指针会被自动销毁。

5. **对于动态分配的对象，只有当对指向它的指针应用delete函数时才会被销毁**。如下代码：

```
class A{
    public:
        A():p(nullptr){
            cout << "构造A\n";
            p = new int(10);
        }
        ~A() { cout << "析构A\n"; }

            private:
                int *p;
};

int main(){
    {
        A* test = new A();
    }
}
```

输出是： 构造A。 没有输出析构A。 因为test是动态分配的 需要delete才能将其销毁。

6. 当指向一个对象的引用或者指针离开作用域时，析构函数不会执行。
7. 需要在析构函数中执行delete操作的类也需要自定义拷贝构造函数和赋值运算符，并且要采用new申请新的内存。**否则如果调用了编译器合成的拷贝构造函数或者赋值运算符，则可能会出现有多一个类对象的成员指向同一块内存，进而导致一块内存被多次释放的错误。**

```
class A{
    public:
        A():p(new int(5)){ cout<<"构造A "<<*p<<endl;}
        ~A() { cout<<*p<<" 析构A\n"; delete p; }

    private:
        int *p;
};

int main(){
    {
        A a;
        A b=a;  // 用 A b(a); 也一样
    }
}
/* 输出结果如下 :
构造A 5
5 析构A
1972752 析构A*/
```

可以看到拷贝构造函数只执行了一次，这是因为b=a执行的是编译器合成的赋值运算符。在析构函数里由于a和b中的指针指向的是同一块内存，因此在第二次释放之前p已经是一个没有指向对象的指针了，故输出垃圾值。

8. 需要自定义拷贝构造函数的类也需要自定义赋值运算符，反之亦然。

阻止拷贝

1. 删除函数是这样一种函数：我们虽然声明了它们，但不能以任何方式使用它们。这种函数的主要用途是禁止编译器自动合成拷贝控制成员。在函数的参数列表后面加上 “=delete” 可以声明。“=delete”必须出现在函数第一次声明的时候。且我们可以将任何函数指定为delete的。
2. 析构函数不能是删除成员。
3. 对于析构函数是删除的类的类型，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。但可以用new为该类的对象申请空间。
4. 如果类的某个成员函数的析构函数是删除的或不可访问的，**则类的合成析构函数也被定义为删除的，且类合成的拷贝构造函数也被定义为删除的**。本质的含义是：如果一个类有数据成员不能默认构造、拷贝、赋值或销毁，则对应的成员函数将被定义为删除的。

5. 没有=delete声明之前，类是通过将其拷贝构造函数和拷贝赋值运算符声明为private来阻止拷贝的，但这样的话类的友元和其成员函数仍可以拷贝对象。

拷贝控制和资源管理

1. 若类中有成员是动态分配内存的，则类的赋值运算符通常组合了析构函数和构造函数的操作：要销毁左侧运算对象的资源，也要从右侧运算对象拷贝数据。（注意当运算符左右两边是同一对象时也要满足）
2. 自定义引用计数：将计数器保存在动态内存中，当创建一个对象时分配一个新的计数器；当拷贝或赋值对象时，拷贝指向计数器的指针。这样副本和原对象就都会指向相同的计数器了。（**也说明了，动态内存可以用来共享数据**）

```
class HasPtr{
    public:
        HasPtr(const string &str=string()):
            ps(new string(s)),i(0),use(new size_t(1)) {}
        HasPtr(const HasPtr &p):
            ps(p.ps),i(p.i),use(p.use) { ++*use;}
        HasPtr& operator=(const HasPtr &rhs){
            ++*rhs.use; //递增右侧对象的引用计数
            --*use;     //递减左侧对象的引用计数
            if(*use==0){
                delete ps;
                delete use;
            }
            ps=rhs.ps,i=rhs.i;
            use=rhs.use;
            return *this;   //返回本对象
        }
        ~HasPtr(){
            --*use;
            if(*use==0){
                delete ps;  //释放共享的string的内存
                delete use; //释放引用计数器内存
            }
        }
    private:
        string *ps;
        int i;
        size_t *use;    //记录有多少对象共享相同的string (ps)
};
```

设计动态内存管理的类

设计一个只用于保存string的vector————StrVec

1. 使用allocator来分配内存但不构造对象，当要构造对象时使用allocator的construct成员构造，当要删除对象时使用destroy成员来删除。
2. 每个StrVec有三个指针成员指向其元素所使用的内存：
    - elements：指向分配的内存中的首元素（begin）
    - first_free：指向最后一个实际元素之后的位置（end）
    - cap：指向分配的内存末尾之后的位置
3. 此外StrVec还有一个类型为allocator<string>的alloc静态成员，用于给每个StrVec对象分配StrVec的内存（针对性的）。
4. StrVec包含四个工具函数：
    - alloc_n_copy：分配内存并拷贝一个给定范围内的元素。
    - free：销毁构造的元素并释放内存。
    - chk_n_alloc：保证StrVec至少有容纳一个新元素的空间。如果没有空间添加新元素，则该函数会调用reallocate来分配更多的内存。
    - reallocate：在内存用完时为StrVec分配新内存。

```
class StrVec{
    public:
        StrVec():elements(nullptr),first_free(nullptr),cap(nullptr){}
        StrVec(const StrVec&);
        StrVec& operator=(const StrVec&);
        ~StrVec();
        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;
        void chk_n_alloc(){ //被push_back所用
            if(size()==capacity()) reallocate();
        }
        pair<string*,string*> alloc_n_copy(const string*,const string*);    //被拷贝构造函数、
        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
        string* elements;
        string* first_free;
        string* cap;
};
allocator<string> StrVec::alloc;    //静态成员必须在类外再次定义

void StrVec::push_back(const string& str){
    chk_n_alloc();  //确保有空间容纳新元素
    //传递给construct的第一个参数是指向allocate所分配的未构造的内存空间
    alloc.construct(first_free++,str);  //在first_free所在的内存构造一个对象
}

pair<string*,string*> StrVec::alloc_n_copy(const string* begin,const string* end){
    /*************************************************************
    该函数会分配足够的内存来保存给定范围的元素并将这些元素拷贝到新分配的内存中
    返回两个指针，一个指向新空间的开始位置，一个指向拷贝的尾后位置
    *************************************************************/
   auto data=alloc.allocate(end-begin);
   //uninitialized_copy是STL中的函数，作用是把[begin,end)范围内的元素拷贝指定的未初始化的内存中
   //返回值是最后一个元素拷贝到的目的地址的下一个地址——尾后地址
   return {data,uninitialized_copy(begin,end,data)};
}

void StrVec::free(){
    /*********************************************************************
    destroy函数的参数是一个指针，该函数对参数所指向的对象执行析构函数，销毁对象，但未释放内存
    deallocate(p,n)释放了从p指向的位置开始的保存了n个对象的内存
    在调用deallocate之前必须对每个在要释放的内存中创建的对象调用destroy
    *********************************************************************/
    if(elements!=nullptr){
        for(auto p=first_free;p!=elements;){
            alloc.destroy(--p); //p的内存还未释放，仍可递减
```

```
        }
        alloc.deallocate(elements,cap-elements);
    }
}

StrVec::StrVec(const StrVec &s){
    auto newdata=alloc_n_copy(s.begin(),s.end());
    elements=newdata.first;
    first_free=newdata.second;
    cap=newdata.second;
}

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

StrVec& StrVec::operator=(const StrVec &rhs){
    /*************************************************
    在释放已有元素之前调用alloc_n_copy使得可以正确处理自赋值
    *************************************************/
    auto data=alloc_n_copy(rhs.begin(),rhs.end());
    free();
    elements=data.first;
    first_free=data.second;
    cap=data.second;
    return *this;
}

void StrVec::reallocate(){
    /*************************************************
    当StrVec容量满时，扩大容量为原来的两倍
    当StrVec为空时，为其分配容纳一个元素的空间
    将原有的数据从旧内存移动到新内存时，为了减少字符串拷贝和销毁的开销
    采用新标准的移动构造函数来移动数据。
    调用移动构造函数的效果是：在新内存空间中构造的每个string都会从旧内存空间
    那接管内存的所有权，而不必将旧数据string进行拷贝。
    *************************************************/
   auto newcapacity=size()?2*size():1;
   auto newdata=alloc.allocate(newcapacity);
   //将数据从旧内存移动到新内存
   auto dest=newdata;   //指向新数组中下一个空闲的位置
   auto elem=elements;  //执行旧数组中下一个元素
   for(size_t i=0;i!=size();++i){
       //move(*elem)表示调用string的移动构造函数
       alloc.construct(dest++,move(*elem++));
   }
   free();  //旧数据移动完成后释放旧的内存空间
   elements=newdata;
   first_free=dest;
   cap=elements+newcapacity;
}
```

对象移动

1. 右值引用：必须绑定到右值的引用，通过“&&”获得。右值引用有一个重要的性质，就是**只能绑定到一个将要销毁的对象**。可以将右值引用看作是给临时变量取了别名，只要临时变量的右值引用存在，该临时变量就不会被销毁，延长了临时变量的生命周期。
2. 对于常规的引用，称为**左值引用**，我们不能将其绑定到要求转换的表达式、字面常量或者是返回右值的表达式。右值引用的绑定特性则相反，可以将一个右值引用绑定到上述的表达式上，但不能将一个右值引用直接绑定到一个左值上。
   看下面的例子：

```
int i=1;
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 &r1=i*42;        //错误，i*42是右值
const int &r2=i*42; //正确，可以将一个const的引用绑定到一个右值上
int &&rr1=i*42;      //正确，右值引用可以绑定右值
```

   上面代码中的**常量左值引用**是一个“万能”的引用，它可以绑定非常量左值、常量左值和右值，但就是绑定后不能**通过该引用**（直接改变其绑定的非常量对象的话还是可以的）更改指向的对象。
3. 左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。因此**右值引用所引用的对象常常是将要被销毁的对象**，该对象没有其他用户
4. **不能将一个右值引用绑定到一个右值引用类型的变量上，因为变量是左值**。

```
int &&rr1=5;         //正确，字面常量是右值
int &&rr2=rr1;       //错误，表达式rr1是左值
```

5. move函数：用于**获得绑定到左值上的右值引用**。定义在头文件utility中。

```
int &&rr1=std::move(rr);
```

   调用了move函数之后意味着承诺除了对rr赋值或者销毁rr外，我们将不再使用它。调用了move之后rr被称为移后源对象。可以看作move是转移了资源的控制权。但是一些基本类型没有对应的移动构造函数，所以使用了move也不可避免的要拷贝。
6. 自定义移动构造函数：

```
//移动构造函数的参数是右值引用
StrVec::StrVec(StrVec &&s) noexcept:
    elements(s.elements),first_free(s.first_free),cap(s.cap){
        s.elements=s.first_free=s.cap=nullptr;
    }
```

   上面的noexpect表示此函数不抛出任何异常（必须在类函数的声明和定义中都指明）。该函数不分配任何新内存，而是**接管给定的StrVec中的内存，在接管内存之后将给定对象中的指针都置为nullptr**。

7. 当右值引用与模板结合时，如下：

```
template<typename T>
void func(T&& arg){}
```

则T&&既可能是左值引用又可能是右值引用，看传递的参数是左值还是右值。未传递参数前，&&是一个未定义的引用类型，称为universal references（通用引用），必须被初始化。&&与auto结合的话也是一样的效果。所以，**只有当发送自动类型推断（模板或者auto）时&&才是一个universal references。

8. 移动赋值运算符：

```
StrVec& StrVec::operator=(StrVec &&rhs) noexcept {
    //检测是否为自赋值，因为rhs可能是move函数的返回结果
    if(this!=&rhs){
        free();
        elements=rhs.elements;
        first_free=rhs.first_free;
        cap=rhs.cap;
        rhs.elements=rhs.first_free=rhs.cap=nullptr;
    }
    return *this;
}
```

9. 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数据成员都可以移动时，编译器才会为该类合成移动构造函数或移动赋值运算符。
10. 新标准库定义了**移动迭代器**，通过改变给定迭代器的解引用运算符实现。移动带带去解引用得到的是一个右值引用。采用make_move_iterator(iterator)将一个普通迭代器iterator转为一个移动迭代器。其他操作与普通迭代器相同。
11. 强制左侧运算对象（即this指向的对象）是一个左值：方式与定义const成员函数相同，即在参数列表后放置一个**引用限定符**。const限定符和引用限定符都只能用于非static的成员函数，且**必须同时出现在函数的声明和定义中**。同样的，使用引用限定符也可以区分重载版本。若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& operator=(cinst Foo&) &;
};
```