第11章　定义抽象数据类型

刘　卉

huiliu@fudan.edu.cn

前 言

- 类的作者可以控制对象行为的各个方面
  - 当类的对象被复制、赋值、销毁时，会发生什么?
  - 本章示例：创建一个vector的简化版Vec.

11.1 确定Vec类的接口

- 如何确定类的接口?
  - 用户会如何使用类来编写程序.
- 实现vector的一个操作子集

```
//1. construct a vector
vector<Student_info> vs;  //empty vector
vector<double> v(100);  //vector with 100 elements
```

```
//2. obtain the names of the types used by the vector
vector<Student_info>::const_iterator b, e;
vector<Student_info>::size_type i = 0;
/*3. use size and the index operator to look at each
  element in the vector */
for (i = 0; i != vs.size(); ++i) cout << vs[i].name();
/*4. return iterators positioned on the first and one
  past the last element */
b = vs.begin(); e = vs.end();
```

11.2 实现Vec类

- 定义一个模板类，使用它创建一系列实例类

```
template <typename T>
class Vec {
public:
   // interface
private:
   // implementation
};
```

- 保存什么数据？
  - Vec对象的元素：首元素指针，末尾元素下一个位置的指针.

```
template <typename T>
class Vec {
public:
   // interface
private:
   T* data;  // pointer of the first element in the Vec
   T* limit;  // one past the last element in the Vec
};
```

- 只有在Vec类模板被实例化时，模板参数的类型才会明确.

```
e.g.  Vec<int> v;  // T实例化为int
      Vec<string> s;  // T实例化为string
```

11.2.1 内存分配

- why not "new T[n]"?
  - new T[n]不仅分配内存空间，还会调用T的默认构造函数初始化每个元素.
  - 仅当T有默认构造函数时，用户才能创建Vec<T>对象.
  - 标准vector类没有这样的强制约束.

内存分配

- 标准库提供了一个分配内存的类allocator<T>
  - 允许分配未初始化的内存空间.
  - 编写私有成员函数管理内存——data和limit；而公有成员只能读取它们.
  - 公有成员需构造新的Vec对象或改变data/limit的值时→调用私有的内存管理函数来完成.

⚠ 策略：公有成员为用户提供接口，私有成员处理实现细节.

```
template <typename T> class Vec {
public:   …
private: …   // 假定以下私有成员函数均已实现
   // facilities for memory allocation
   std::allocator<T> alloc; //object to handle memory allocation
   // allocate and initialize the underlying array
   void create();
   void create(size_type, const T&);
   void create(const_iterator, const_iterator);
   // destroy the elements in the array and free the memory
   void uncreate();
   // support functions for push_back
   void grow();
   void unchecked_append(const T&);
};
```

11.2.2 构造函数

- 确保对象被正确初始化

```
template <typename T> class Vec {
public:
    Vec() { create(); }     // the default constructor
    explicit Vec(size_type n, const T& val=T())
    { create(n, val); }
     // remaining interface
private:
    T* data;
    T* limit;
};
```

返回时，data和limit都被设置为0

生成空的Vec对象（不包含任何元素）

带默认值的参数

```
explicit Vec(size_type n, const T& val=T()) {create(n, val);}
```

- create函数分配足够内存保存n个T类型对象，并为这些元素提供初值val.

e.g. `Vec<int> vi(10, 1)` //构造Vec对象vi，包含10个值为1的元素

explicit

- 只能用在带一个参数的构造函数定义中.
- 仅在用户明确调用该构造函数的地方，编译器才使用它.

```
Vec<int> vi(100);// ok, explicitly construct the Vec from an int
Vec<int> vi = 100;  // error: implicitly construct the Vec and
                    // copy 100 to vi
```

11.2.3 类型定义

- 标准模板类的惯例
  - 提供给用户使用的类型名→隐藏类的实现细节.
  - 用typedef语句定义const/非const迭代器类型，以及表示Vec大小的类型.

```
typedef T* iterator; //使用指针作为Vec的迭代器类型
typedef const T* const_iterator;
typedef size_t size_type; //使用size_t作为size_type的底层类型
```

- 标准库容器还要求有：value_type类型，reference和const_reference类型，difference_type类型

```
typedef T value_type; // Vec元素的类型
typedef std::ptrdiff_t difference_type; // 指针差值类型
typedef T& reference; // 引用类型
typedef const T& const_reference; // 常引用类型
```

- 使用自定义的类型名→代码易读 && 便于修改.

```
template <typename T> class Vec {
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    typedef T& reference;
    typedef const T& const_reference;
    typedef std::ptrdiff_t difference_type;
    Vec() {create();}
    explicit Vec(size_type n, const T& t=T()){create(n, t); }
    // remaining interface
private:
    iterator data;   // changed
    iterator limit;  // changed
};
```

类型定义

构造函数

数据成员

11.2.4 大小和索引

- size成员函数

```
size_type size() const { return limit - data; }
```

  - limit-data的类型是ptrdiff_t，可自动转换为size_t类型.
  - size函数为const成员函数→可获得Vec类const对象的大小.

重载操作符[]

- 函数名：operator[]

```
T& operator[](size_type i) { return data[i]; }
const T& operator[](size_type i) const { return data[i]; }
```

  - 索引操作符找到底层数组的对应元素，返回该元素的引用. 两个版本：

1）非const成员函数→非const对象调用，返回元素本身，可作为左值.

```
const T& operator[](size_type i) const { return data[i]; }
```

2）const成员函数→非const/const对象均可调用，返回元素本身的常引用，只读.

```
Vec<int> vi(10, 0);
for(Vec<int>::size_type i = 0; i != 10; ++i)
      cout << vi[i] << endl;  // vi[i]: 只读
// vi.operator[](i)⇔vi[i]
vi[9] = 10;  // vi[9]: 调用T& operator[](size_type i)
```

- 返回引用→避免复制容器中的元素.
- 每个成员函数都带有一个隐式参数：它们所操作的对象.

⚠ 索引操作符[]必须重载为类的成员函数.

11.2.5 返回迭代器的操作

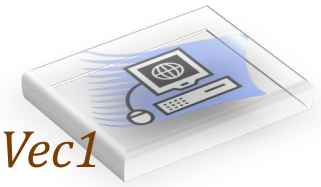
- begin和end函数

```
iterator begin() { return data; }
const_iterator begin() const { return data; }
iterator end() { return limit; }
const_iterator end() const { return limit; }
```

  - 分别提供了两个版本:const版本返回const_iterator，通过它可以读取容器元素但不允许修改.

11.3 Copy Control

- 如果没有定义对象被创建、复制、赋值、销毁等操作，编译器就会合成这些操作的定义.
  - 默认构造函数，拷贝构造函数，赋值运算符，析构函数.
- C++是唯一一种为程序员提供了这种级别控制的语言.



11.3.1 Copy constructor

1. 显式拷贝构造

- 用一个对象初始化另一个对象

```
vector<Student_info> vs;   // 默认构造
vector<Student_info> v2 = vs;    // copy vs into v2
// vector<Student_info> v2(vs);
```

2．隐式拷贝构造

1）实参→形参：把对象的值传递给函数

```
vector<double> vd;
double d;
d = median(vd); // copy vd into vec in median
```

```
double median(vector<double> vec)
```

2）函数返回某个对象的值

```
string line;
vector<string> words = split(line);
/* copy the return from split into words */
```

```
vector<string> split(const string& s)
```

Copy constructor

```
Vec(const Vec& v);  //why not Vec<T>?
```

- 成员函数，函数名与类名相同，无返回值.
- 形参：同类型对象的常引用.
  - 在类的生存空间中，可以省略模板参数(隐式包含).
- 功能：把新对象初始化为已有同类型对象的副本
- 具体做什么?
  - 把已有对象的每个数据成员，复制到新对象的每个数据成员中.

- 当数据成员的类型是指针时，

  e.g. Vec类的data和limit成员

  - 仅复制指针会使新老对象指向相同的底层数据.
  - 标准vector类复制时，不会共享底层存储，而是互相独立.
  - 复制Vec对象时，需分配新的空间，然后复制内容.

```
public:
   Vec(const Vec& v) { create(v.begin(), v.end()); }
  // as before
```

11.3.2 Assignment

- 控制赋值操作符'='的行为
  - 定义了把同类对象赋值给另一个对象时的操作.
  - 参数类型：类本身的const引用(与copy constructor一样)
  - 必须重载为类的成员函数
  - 返回值：左操作数的引用→连续赋值运算

```
public:
  Vec& operator=(const Vec&); //why not Vec<T>?
  // as before
```

Assignment is not initialization

- Assignment *vs* Copy constructor
  - 相同：为每个数据成员赋值.
    - 对于指针类型的数据成员，仅复制指针会使新老对象指向相同的底层数据.
  - 不同：赋值操作先删除左操作数的已有数据，用右操作数的值来替换.
- 避免自我赋值——使用this指针
  - 只在成员函数内部有效，指向所操作对象本身.

- operator=放在类定义之外实现

```
template <class T>
Vec<T>& Vec<T>::operator=(const Vec& rhs)
{
    // check for self-assignment
    if (&rhs != this) {
        // free the array in the left-hand side
        uncreate();
    // copy elements from the right-hand to the left-hand side
        create(rhs.begin(), rhs.end());
    }
    return *this;// e.g. v1=v2, 与v1.operator=(v2)等价, 返回v1
}                // 支持连续赋值, e.g. v1=v2=v3
```

在模板类的定义之外实现成员函数

```
public:
    Vec& operator=(const Vec&);
    // as before
```

函数名

```
template <class T>
Vec<T>& Vec<T>::operator=(const Vec& rhs) {...}
```

- 返回值类型是Vec<T>&而不是函数声明中的Vec&.
  - 在类的生存空间中，可以省略模板参数(隐式包含).
  - 在类的-------之外，必须显式说明.
- 一旦指定它是Vec<T>的成员，就不再需要使用模板限定词了→参数类型是const Vec&

正确地处理自我赋值非常重要！

```
if (&rhs != this) {
    uncreate();  // free the array in the left-hand side
    create(rhs.begin(), rhs.end());
}
```

- 如果不进行检测，发生自我赋值时：

1）uncreate函数在释放左操作数所占空间的同时，也释放了右操作数的空间.

2）create函数将无从复制.

- 返回*this
  - 返回当前对象的引用：必须确保在函数返回时，引用的对象依然存在.
  - 返回局部对象的引用将导致灾难.

Assignment is not initialization

- 理解赋值和初始化之间的区别——学好C++的关键之一
  - 使用'='为一个变量提供初值时，调用copy constructor.
  - 在赋值表达式中使用'='时，调用operator=.
  - 赋值总会删除先前的值，而初始化则不会.
  - 初始化会创建一个新的对象，同时为这个对象提供值.

何时发生初始化

1）变量声明.

2）函数入口，实参向形参传递值.

3）函数返回一个值的时候.

4）构造函数的初始化列表.

```
Student_info::Student_info():
midterm(0), final(0) { }
```

```
e.g.   string url_ch = "~;/?:@=&$-_.+!*'(),"   // initialization
/*构造函数string(const char* cp)直接从字符串常量构造url_ch; 或者先
   构造一个临时string对象, 再调用string(const string&)构造url_ch */
       string spaces(url_ch.size(), ' ') ;     // initialization
       string y;     // initialization
       y = url_ch;   // assignment
```

Assignment is not initialization

```
e.g. vector<string> split(const string&); //function declaration
     vector<string> v; //initialization, calling constructor
     v = split(line); /* on exit, both initialization of the
   return value and assignment to v */
```

- 返回时，需要两步：

1）调用copy constructor把返回值复制到一个临时对象中；

2）赋值操作符再把临时对象的值赋给左操作数.

Assignment is not initialization

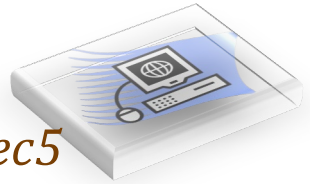
- 初始化和赋值会引起不同操作
  - 构造函数：控制对象的初始化.
  - operator=：控制对象的赋值操作.

11.3.4 析构函数(destructor)

- 控制对象被销毁时发生的操作
  - 函数名：~类名
  - 不带任何参数，没有返回值.
  - 完成对象被删除时的清理工作：释放构造函数分配的所有资源.

```
public:
    ~Vec() { uncreate(); }//与operator=删除左操作数原先值的行为类似
    // as before
```

  - 析构顺序：后定义的对象先析构.



11.3.5 默认操作

- 默认版本定义为递归操作
  - 以对象本身初始化时采用的方式，递归地初始化每个数据成员

1）成员的类型是类→调用该类的copy constructor、赋值操作符或析构函数.

2）成员的类型是内置类型→直接进行拷贝或赋值. `Date d = d1;`

3）内置类型的析构函数不做任何操作.

  - 销毁指针时，不释放指针所指的内存空间——内存泄漏.

默认构造函数

- 如果明确定义了任何一个构造函数，编译器就不再合成默认构造函数.
- 良好的编程习惯：为每个类提供一个默认构造函数.

11.3.6 "三者缺一不可"原则

- 管理资源的类，需注意复制操作
  - 默认的copy constructor不能满足要求.
  - 如果类在构造函数中分配资源⇨复制时，需要复制资源⇨析构时，需要释放资源.
  - 如果一个类需要析构函数，那么它也需要copy constructor和赋值操作符.

- 为了控制类T对象(需要分配资源)每次复制的独立性，需提供如下操作：

```
T::T() one or more constructors, perhaps with
  arguments
T::~T() the destructor
T::T(const T&) the copy constructor
T::operator=(const T&) the assignment operator
```

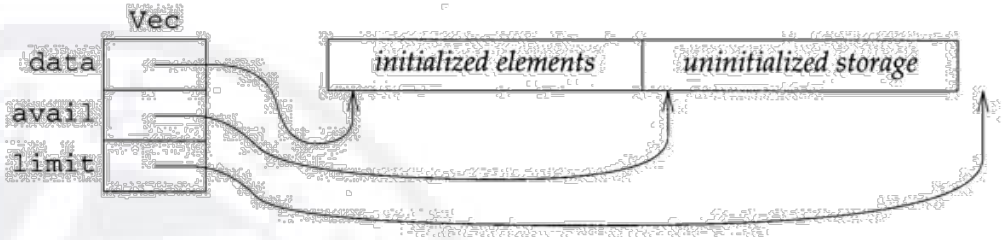
  - 类T对象被(隐式/显式)创建、复制、赋值或销毁时，编译器就会调用相应操作.

11.4 动态Vec对象

- 如何在Vec对象的末尾添加元素？
  - 实现push_back操作
  - 添加一个元素，申请一次空间——效率低.
  - 分配比实际需要更多的内存空间.

    e.g. 分配当前使用空间两倍的内存空间.
  - 用完预分配的空间之后，再分配更多内存.

- 改变数组的记录方式
  - 需要两个"末尾"指针.




```
private:
    //as before, pointer to the first element in the Vec
    iterator data;
    // pointer to (one past) the last constructed element
    iterator avail;
    //now points to (one past) the last available element
    iterator limit;
```

```
public:
    size_type size() const { return avail - data; }  // changed
    iterator end() { return avail; }  // changed
    const_iterator end() const { return avail; }  // changed
    void push_back(const T& t) {
        if (avail == limit)    // get space if needed
            grow();
        unchecked_append(t);   // append the new element
    }
   // rest of the class interface and implementation as before
```

11.5 灵活的内存管理

- 不使用new和delete的原因
  - new不仅分配内存，还会初始化→多了限制和开销.
- 类allocator<T>———<memory>
  - 分配保存T类对象的整块内存，并不初始化.
  - 返回一个指针，指向这块内存的首元素.
  - 程序员确定哪些空间保存构造的对象，哪些空间尚未初始化.

相关函数

```
template<class T> class allocator {
public:
   T* allocate(size_t);  //分配指定类型但未初始化的内存
   void deallocate(T*, size_t);  //释放未初始化的内存
   void construct(T*, const T&) ; //在未初始化空间中构造一个对象
   void destroy(T*); //销毁参数所指对象，恢复到未初始化状态
   // ...
};
//初始化allocate分配的原始内存空间
template<class In, class For> For uninitialized_copy(In, In,
   For);
template<class For, class T> void uninitialized_fill(For, For,
   const T&);
```

- 用allocator类的对象为Vec类分配/释放内存空间→为Vec<T>添加一个allocator类的成员alloc.

```
private:
    // facilities for memory allocation
    std::allocator<T> alloc; // object to handle memory allocation
    // allocate and initialize the underlying array
    void create();
    void create(size_type, const T&);
    void create(const_iterator, const_iterator);
    // destroy the elements in the array and free the memory
    void uncreate();
    // support functions for push_back
    void grow();
    void unchecked_append(const T&);
```

实现进行内存分配的函数

- 类不变式
  - 一个有效的Vec对象需满足：

1）如果对象中有元素，data指向首元素；否则data为零；

2）data <= avail <= limit;

3）区间[data, avail)中的元素被构造；

4）区间[avail, limit)中的元素没有被构造.

  - 只要构造类的对象，就需要建立这个类不变式.

- 构造对象时满足这4个条件，确保没有任何成员函数违背该类不变式→确保不变式总是为真.
- 使不变式为假的唯一方式是改变data/avail/limit的值→公有函数不会使不变式为假.

- 各种版本的create函数
  - 负责内存分配，初始化内存中的元素，并设置各指针的值.
  - 在运行create函数后，指针limit和avail总是相等的→满足类不变式.

- uncreate函数
  - 从后往前逆序销毁Vec的所有元素，并释放该对象使用的所有空间.
- grow函数
  - 重新分配双倍空间，释放原来空间.
  - 分配足够的空间，使其至少可多保存一个元素.
  - 特殊情况：当前Vec对象为空.
- unchecked_append
  - 在有效元素的下一个位置创建一个元素.

小 结

- 模板类

```
template <class type-parameter [, class type-parameter]... >
class class-name { ... } ;
```

  - 所有type-parameter都可用在模板中需要它们的地方.
  - 在类的生存空间中，不使用限定词就可使用该模板类;
  - 在类的生存空间外，class-name必须使用type-parameter限定.
  - 在创建模板类对象时，用户指定实际类型.

- 拷贝控制
  - 在创建(且拷贝)对象时，调用构造函数.
  - 包含赋值操作的表达式则调用赋值操作符.
  - 当对象退出生存空间，或被明确销毁时，调用析构函数.
  - 在构造函数中分配资源的类，需要定义copy constructor、operator=和destructor.
  - 赋值操作符：检查自我赋值，返回左操作数的引用.
- 合成操作
  - 类的每个合成操作都会递归使用数据成员的相应操作.

- 重载操作符operator op
  - 如果操作符函数是类成员→它的左操作数(或仅有的操作数)就是调用它的对象.
  - 索引操作符和赋值操作符必须定义为类的成员.