面向对象程序语言C++
类和模板

周雅倩
zhouyaqian@fudan.edu.cn
2016/4/28

定义抽象数据类型

- 模板类可以通过使用模板功能来生成：

```
template <class type_parameter,...>
class class-name{
};
```

- 上面的语句创建了一个名叫class-name的模板类，这个类与给定的参数类型有关。
- 所有的type-parameter名字都可以用在模板中需要这些类型的地方。

模板类

- 在类的生存空间中，不使用限定词就可以使用这个模板类
- 在类的生存空间外，class-name必须使用type-parameter来限定：
  - template < class T>
  - Vec<T> & Vec<T>::operator=(const Vec&){...}
- 在创建模板类型的对象时，用户可以指定实际的类型

Vec类

- Vector类的使用：
- //construct a vector

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

- //obtain the names of the types used by the vector

```
vector<Student_info>::const_iterator b,e;
vector<double>::size_type i=0;
```

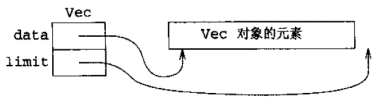
- //use size and the index operator to look at each element in the vector

```
for(i=0;i!=vs.size();++i)
    cout << vs[i].name();
```

- //return iterators positioned on the first and on past the last element

```
b=vs.begin(); e=vs.end();
```


实现Vec类




```
template <class T> class Vec{
    public:
        //接口
    private:
        T* data;//Vec中的首元素
        T* limit;//Vec中的末元素
};

Vec<int> v;
Vec<string> vs;
```


内存分配

- 方式一：new T[n];
  - 要求：T有一个默认构造函数
  - 标准vector类没有这样的强制约束
- 方式二：使用标准库提供的内存管理类

new和delete

- C++语言引入对象动态创建和释放运算符new和delete，可让程序实现动态创建对象和回收对象占用的内存和资源。
- 如以下代码：

```
T *ptr;
ptr = new T;
...
delete ptr;
```


空间的申请

- 若T不是类，是某种普通的数据类型，则其中代码“ptr = new T”实现动态申请一块堆空间，与用以下代码有同样的功能：

```
ptr = (T*)malloc(sizeof(T));
```

- 若类型T是类，则代码“ptr = new T”将自动调用类的构造函数，创建对象，为对象分配堆空间和其它资源，并为对象设置初值。
- 用new运算符动态分配堆空间还可指定初值。如：

```
double *dPtr = new double(3.1415926);
```

  - 将分配的对象初始化为3.1415926。

数组空间的申请与释放

- 也可以用new动态分配一个数组。如：

```
int *arrayPtr;
arrayPtr = new int[100];
```

- 要释放为该数组分配的堆空间，需用以下形式的代码：

```
delete [ ]arrayPtr;
```


new,delete Vs. malloc, free

	C++	c
申请空间	new 若是用于一般变量，则除分配时可设置初值外，分配堆空间与malloc()相同； 若是用于创建对象，new会调用构造函数，动态创建对象，对象创建和内存分配是同时进行的。	malloc 只是按要求的字节数分配堆空间
释放空间	会调用类的析构函数，析构对象和回收堆空间。	只做回收堆空间的工作

3个构造函数

- 默认构造函数
- 指定元素个数的构造函数
- 指定元素个数和初始值的构造函数

```
Vec(){create();}
//第二个形参带默认参数
explicit Vec(std::size_t n, const T& Val=T()){create(n,Val);}
```


关键词explicit

- 只在带有一个参数的构造函数的定义中有意义
- 当我们说一个构造函数是explicit,表明只有在用户明确地调用这个构造函数的地方，编译器才能使用这个构造函数,否则就无法使用它
- Vec<int> vi(100);//OK
- Vec<int> vi=100;//ERROR

类型定义

- 提供一个用户可以使用的类型名，这样可以 隐藏实现 这个类 的细节。

```
template  <class  T> class  Vec{
    public:
        typedef  T* iterator;
        typedef  const  T* const_iterator;
        typedef  size_t  size_type;
        typedef  T value_type;
        typedef  ptrdiff_t  difference_type;
        typedef  T&  reference;
        typedef  const  T&  const_reference;

        Vec(){creat();}
        explicit  Vec(std::size_t  n, const  T& Val=T()){creat(n, val );}
    private:
        iterator  data;
        iterator  limit;
};
```


索引和大小

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


重载操作符

- 重载操作符是通过名叫operator　op的函数来定义的，其中op是需要定义的操作符。
  - 操作符的种类在一定程度上决定了对应函数应该带有多少个参数。
  - 这个操作符的操作数中，必须至少有一个参数的类型和该类相同。
- 操作符函数可以是成员函数，也可以是非成员函数。
  - 当一个操作符函数是一个类的成员时，它的左操作数或者它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是类的成员。

返回迭代器的操作

- 关于begin和end操作，都 提供了两个版本，都是根据Vec是否为const来重载的。

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

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


复制控制

- 一般来说，类可以控制对象在创建、复制、赋值和销毁时的所有行为。
  - 在创建或复制对象时，就会调用构造函数；
  - 包含赋值操作的表达式会调用赋值操作符；
  - 当对象退出生存空间，或被明确地销毁时，就会自动调用析构函数。

复制控制-2

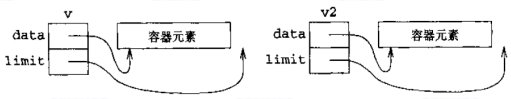
- 在构造函数中分配资源的类，几乎都必须定义复制构造函数、赋值操作符和析构函数。
- 当我们编写一个赋值操作符时，我们要特别检查自我赋值的情况。
- 为了与内置的赋值操作符保持一致，我们应该养成一个好的习惯：返回左操作数的一个引用。

复制构造函数的应用

- 把一个对象的值传递到一个函数中
  vector<int> vi;
  double d;
  d=median(vi);
- 从一个函数返回一个对象的值
  string line;
  vector<string> words=split(line);
- 通过一个对象来初始化另一个对象
  vector<Student_info> vs;
  vector<Student_info> v2=vs;

复制构造函数

```
template <class T> class Vec{
    public:
        ...
        Vec(){creat();}
        explicit Vec(std::size_t n,const T& Val=T()){creat(n,val);}
        Vec(const Vec&v){create(v.begin(),v.end());}
        ...
    private:
        iterator data;
        iterator limit;
};
```



赋值

- 一个类可以定义几种赋值操作符函数。
- 带有一个类本身的const引用的版本通常被叫做“赋值操作符”

```
template <class T>
Vec<T> & Vec<T>::operator=(const Vec &rhs)
{
    if(&rhs != this){
        uncreate();
        create(rhs.begin(),rhs.end());
    }
    return *this;
}
```


赋值不是初始化

- 当使用=来为一个变量提供一个初始值时，调用的是复制构造函数；
  Vec<double> hw;
  ……
  Vec<double> fail = hw;
- 当在赋值表达式中使用它时，调用的是operator=。
  Vec<double> hw, fail;
  ……
  fail = hw;

析构函数

- 析构函数的任务是完成对象被删除时的所有清理工作：一般指的是释放构造函数分配的所有资源，比如内存空间。

```
template <class T>
class Vec{
    public:
        ~Vec(){uncreate();}
        ...
};
```


默认操作

- 有一些类，并没有显式的定义一个复制构造函数、赋值操作符或者一个析构函数。编译器会生成这些未定义操作的默认版。
- 默认版的函数会定义为递归操作：根据元素类型的适当规则，复制、赋值或者销毁数据元素。

合成的操作

- 如果一个类没有定义构造函数，编译系统就会合成默认构造函数
- 如果这个类没有明确的定义复制构造函数、赋值操作符以及析构函数的话，编译器也会合成这些操作。
- 合成的操作是递归的定义的：每个合成的操作都会递归地使用这个类的数据成员的相应操作。

三者缺一不可的规则

- 管理资源（比如内存）的类，需要密切注意复制操作。
  - 一般来说，默认构造函数不能够满足这样的类。
- 如果一个类需要一个析构函数，那么它也需要一个复制构造函数和一个赋值操作符。

动态Vec对象

```
template <class T>
class Vec{
    public:
        size_type size() const{return avail-data;}
        iterator end(){return avail;}
        const_iterator end()const{return avail;}
        void push_back(const T& val){
                if(avail==limit)//若空间不够
                        grow();
                unchecked_append(val);
        }
    private:
        T* data;
        T* avail;
        T* limit;
};
```

create函数

```
#include <memory>
template <class T>
void Vec<T>::create(size_type n, const T&val)
{
    data=alloc.allocate(n,NULL);
    avail=limit=data+n;
    std::uninitialized_fill(data,limit,val);
}
```

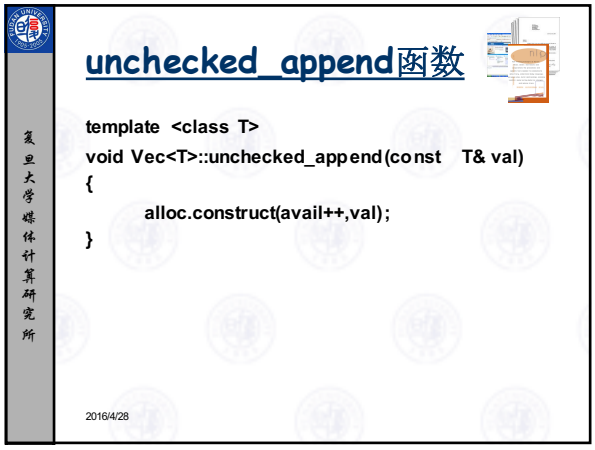

uncreate函数

```
template <class T>
void Vec<T>::uncreate()
{
        if(data){
                iterator it=avail;
                while(it!=data)
                        alloc.destroy(--it);
                alloc.deallocate(data,limit-data);
        }
        data=avail=limit=0;
}
```


grow函数

```
template <class T>
void Vec<T>::grow()
{
        size_type new_size=1;
        if(limit!=data) new_size=2*(limit-data);
        iterator new_data = alloc.allocate(new_size,0);
        iterator new_avail = uninitialized_copy(data,avail,new_data);

        uncreate();
        data=new_data;
        avail=new_avail;
        limit=data+new_size;
}
```


复旦大学媒体计算研究所
unchecked_append函数
template <class T>
void Vec<T>::unchecked_append(const T& val)
{
alloc.construct(avail++,val);
}
2016/4/28