Report

Report of template class Vector

一.模板类的实现

私有成员变量

1.obj_size:存储元素个数
2.obj_capacity:实际申请空间大小
3.objects:一级指针，指向申请空间的首地址

私有成员函数

1.void releasemem():为析构函数提供释放内存空间的手段
2.void warning(int error):用于抛出异常，如提示下标越界或者数组为空

公有成员函数

1.explicit Vector(int init_size=0):普通构造函数，初始元素个数为0,通过explicit声明单参数显式调用
2.explicit Vector(const Vector& rhs):深复制构造函数,通过explicit声明单参数显式调用
3.const Vector& operator=(const Vector& rhs):重载赋值运算符函数，申请新空间并释放旧空间
4.~Vector():析构函数，终结变量声明周期并释放相应内存空间,调用releasemem()
5.size(),capacity():分别返回obj_size,obj_capacity
6.bool is_empty():判断obj_size是否为0
7.obj& operator [] (int obj_index) const:函数调用运算符，obj_index>=obj_size时调用warning()抛出异常
8.void printVector():输出对象0~size()-1存储的值
9.void resize(int new_size):当new_size>obj_capacity时调用reserve()重构一个obj_capacity=2*new_size+1的新数组并将旧元素复制回去，若new_size>obj_size则为obj_size赋新值
10.void reserve(int new_capacity):若new_capacity>=obj_size则创建一个temporary指针指向当前objects指向的地址并为对象申请新规模空间并将旧元素复制回去，最后释放temporary
11.const obj& back() const:返回对象存储的最后一个元素
12.void push_back(const obj& new_element):若obj_size==obj_capacity则reserve一个新空间，为对象添加一个新元素，同时obj_size增加1
13.void pop_back():令obj_size减1，相当于忽略对象最末尾的元素

迭代器

typedef obj* iterator;
typedef const obj* const_iterator;
1.const_iterator begin() const:返回存储的第一个元素的地址
2.const_iterator end() const:返回存储的最后一个元素的地址

二.Vector类基本测试

具体见main.cpp，测试点全部通过会有相关说明。测试表明不会发生内存泄漏。

三.SPARE_CAPACITY(以下简称s)的作用

obj_size为实际存储个数，obj_capacity为实际申请空间大小，s=obj_capacity-obj_size,显然是为了防止频繁push新元素导致不停reserve新空间带来的时间上的浪费而提前预留的空闲空间。

四.s对各类成员函数的影响

实际上s只对obj_capacity的值与push_back()有一定影响。其中对于push_back(),由于提前预留了一定空间以便于push新元素的时候不用reserve新空间，故属于空间换时间的方式。尤其当size变化范围较大的时候，申请新空间、复制旧元素会很浪费时间，故这时s(SPARE_CAPACITY)就显得十分有用。

五.对s最优大小的理论分析

注：size等可能分布在1~N中.

步骤一

首先我们通过**均摊分析法**来计算push_back()的时间复杂度。
首先抛开s，考虑初始大小为1、倍增因子为m的数组空间，现要求size为n，那么需要调用reserve()的次数为t=logm(n),第i次reserve会复制m^i数量的元素，因此所有reserve()花费总时间为：

$$\sum_{i=1}^{t} m^i \approx \frac{n*m}{m-1}$$

因为共n次操作，平均下来每push一次花费的时间为：

$$\frac{m}{m-1}$$

，即常量时间（在我们这m几乎就是2）.

步骤二

下面来计算对于分配范围为[1,N]的size需要花费的总时间。设分配单位大小空间的相对时间为常数a；复制单个元素的相对时间为常数b；根据步骤一，我们可以设push_back一个新元素的相对时间为常数c。至于obj_size,obj_capacity等值之间比较的操作时间复杂度都是O(1)，不考虑在内。
考虑初始分配空间大小为S{0}=s，现要求size为n。由S{k}=2*S{k-1}+1易得S{k}+1=2^k(s+1),现要找出p和q,其中q=p+1,使得：

$$2^p * (s+1) < n+1 <= 2^q * (s+1)$$

解得p=

$$p < \log_2(\frac{n+1}{s+1}) <= q = p+1$$

$$p+1=\lceil \log_2(\frac{n+1}{s+1})\rceil$$

即共需调用reserve()函数(p+1)次。总共新增元素的次数为：

$$S0+S1+...+Sp+n=(2^0+2^1+...2^p)*(s+1)-(p+1)+n$$

化简得可近似为:

$$2*n-s-(p+1)=2*n-s-\lceil \log_2(\frac{n+1}{s+1})\rceil$$

其中复制次数和push次数分别为:

$$n-s-\lceil \log_2(\frac{n+1}{s+1})\rceil$$

$$n$$

又分配单位空间次数可近似为:

$$S0+S1+...+Sp+1\approx 2*n-s-\lceil \log_2(\frac{n+1}{s+1})\rceil$$

故对于size为n花费的相对总时间为:

$$time(n)=(a+b)*(2*n-s-\lceil \log_2(\frac{n+1}{s+1})\rceil)+c*n$$

又因为size在[1,N]均匀分布，不妨考虑对n累加起来的总时间减去与s无关的常量得到的值:

$$T(s)=\frac{\sum_{n=1}^{N} time(n)-uncorrelation}{a+b}\approx -N*s-\sum_{n=1}^{N}\lceil \log_2(\frac{n+1}{s+1})\rceil$$

若只考虑时间上达到最优，那么只需要求T最小时s所取的值，对T求s的偏导:

$$T'=[\frac{1}{(s+1)*ln2}-1]*N$$

可见T随着s的增大先递增后递减，又考虑到上述讨论基于s<=N的事实，并且显然不会延拓到s>N的情况，因此哪怕不是为了节省空间我们也仍然至多选取s为N。s的另一个选择是1，我们分别代入1和N进T并比较:

$$T(1)-T(N)=N^2-N-\sum_{n=1}^{N}\log_2(\frac{N+1}{n+1})=(N-N)+(N-\log_2(\frac{N+1}{1+1}))+...+(N-\log_2(\frac{N+1}{N-1+1}))>0$$

可见s=**N**是最优解.

六.对s最优大小的实验测试

注:实验测试代码在main.cpp中，可以make然后bash run进行测试，测试首先是基本测试，然后将提示输入N和spare_capacity的值并输出相对时间大小，推荐对同一个N测试多组并进行比较。

```
int t,N,count = 1;
   clock_t start,end;
   cout<<"Now testing spare_capacity, recommend "<<endl<<"input";
   cout<<":N/10,N/5,N/4,N/3,N/2,6N/10,7N/10,8N/10,9N/10,N."<<endl;
   cout<<"input 0 to stop, 1 to try again:";
   while (cin>>t && t) {// input 0 if want to stop the experiment.
       cout<<count++<<"th try, please input N and spare_capacity:";
       cin>>N>>spare_capacity;
       start = clock();
       Vector<int> testvec;
       for (int n = 1;n < N;n++) {
           for (int i = 0;i < n;i++) {
               testvec.push_back(i);
           }
       }
           end = clock();
           cout<<"while N="<<N<<", spare_capacity="<<spare_capacity;
           cout<<":relatively caused time="<<end - start<<endl;
            cout<<"input 0 to stop, 1 to try again:";
   }
   cout<<"test ends."<<endl;
```

（上为相关代码展示）

实验一：选取N为1000

SPARE_CAPACITY	1	150	250	333	500	600	700	800	900	1000
relative time	3000	2275	875	3310	2400	2300	3000	2400	1900	800

实验二：选取N为100

SPARE_CAPACITY	1	15	25	33	50	60	70	80	90	100
relative time	18	19	14	15	13	15	15	14	13	13

实验三：选取N为10000

SPARE_CAPACITY	1	1500	2500	3333	5000	6000	7000	8000	9000
relative time	400000	470000	440000	350000	430000	460000	410000	400000	40000

结论与思考

由三个实验我们大致可以看出s=N时的确能优化时间，但是效果不甚明显，甚至三个实验都出现了当s=N/4时总相对时间出现了不正常的减少。其实前面一个问题不难解释，编译器会对代码进行优化，同时根据代码写法和编译器类型的不同，都会对实际运行时间产生一定的影响。在工程与应用中，让类的使用者自己输入SPARE_CAPACIYTY的值肯定是不现实的，故一般取为16或者干脆不使用SPARE_CAPACITY，这样虽然降低了效率，但提高了所谓的封装性。