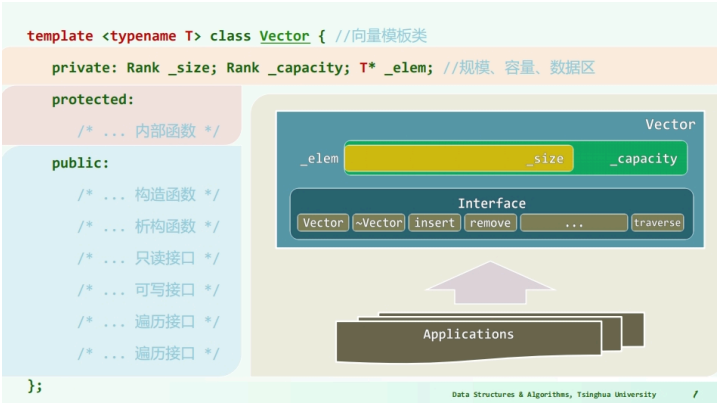
第1章 向量

本章主要介绍向量的以下几个方面：
构造和析构
访问
增删查排

In [1]:
```
#include <iostream>
using namespace std;
```



我们定义的数据结构是按照邓俊辉老师的DSACPP来的，
但是只注重数据的结构，以及算法的实现，对于数据和接口
对外的可读性，对下的继承性，不作任何要求，因此，我们
后面定义的所有成员变量和成员函数，全部为公有属性

In [2]:
```
template <class T>
class Vector{
    public:
        // 构造和析构
        int size; int capacity; T* data;                // 数据和属性
        Vector(int s);                                  // 构造函数1
        Vector(const T* A, int lo, int hi);             // 构造函数2
        Vector(const Vector<T>& A, int lo, int hi);     // 构造函数3
        ~Vector(){delete [] data;}                     // 析构函数

        // 访问
        T& operator[](int r){
            return data[r];
        }

        // 扩容和缩容
        void expand();
        void shrink();

        // 插入和删除
        int insert(int r, T e);
        int remove(int r);

        // 查找和排序
        int search(T e, int lo, int hi);
        void merge(int lo, int mi, int hi);
        void mergeSort(int lo, int hi);

        // 补充：
        bool empty(){return !size;}
};
```

一、Vector的定义和构造

给定向量的大小创建一个空向量
首先输入参数就是一个整数s，然后我们设定
这个向量的大小size就是s，容量设定为2倍的s，
然后开辟出一个容量大小的空间给data

In [3]:
```
template <class T>
Vector<T>::Vector(int s){
    size = s;
    if (s == 0)
        capacity = 100;
```

```
    else
        capacity = s << 1;
    data = new T[capacity];
}
```

In [4]:
```
auto v1 = Vector<int>(20);
cout << v1.size << endl;
cout << v1.capacity << endl;
```

```
20
40
```

根据数组构造一个向量

我们的输入参数就是一个数组,以及数组的区间,

然后把这个数组拷贝到向量的data中,并更新向量的size和capacity属性

In [5]:
```
template <class T>
Vector<T>::Vector(const T* A, int lo, int hi){
    size = hi - lo;
    capacity = size << 1;
    data = new T[capacity];
    for(int i = 0; i < size; i++){
        data[i] = A[lo + i];
    }
}
```

In [6]:
```
int a[4] = {1, 2, 3, 4};
auto v2 = Vector<int>(a,0,4);
```

In [7]:
```
cout << v2.size << endl;
cout << v2.capacity << endl;
cout << v2.data[0] << endl;
```

```
4
8
1
```

从一个向量的指定区间构造另一个向量

In [8]:
```
template <class T>
Vector<T>::Vector(const Vector<T>& A, int lo, int hi){
    size = A.size;
    capacity = A.capacity;
    data = new T[capacity];
    for(int i = 0; i < size; i++){
        data[i] = A.data[i];
    }
}
```

In [9]:
```
auto v3 = Vector<int>(v2,0,v2.size);
```

In [10]:
```
cout << v3.size << endl;
cout << v3.capacity << endl;
cout << v3.data[0] << endl;
```

```
4
8
1
```

访问接口

这里重置运算符[]进行循秩访问，就是循下标访问，并支持修改元素，功能与数组相同，

另外这个可以拆分为两个函数分别实现，分别为get和put，这里就不这样做了。

由于xeus-cling不支持运算符在类外面定义，所以我们直接在上面定义好了，下面直接举例使用

In [11]:
```
v3[0] = 100;
cout << v3.data[0] << endl;
```

```
100
```

二、动态空间管理

由于插入删除算法会使得向量的大小发生变化，频繁的插入或者删除可能导致向量

空间不够，所以这里先介绍扩容与缩容算法

扩容算法

判断当前向量的元素数量size是否小于容量，如果不小于，则扩容两倍

In [12]:
```
template <class T>
void Vector<T>::expand(){
    if (size < capacity) return;
    T *oldData = data;
```

```
    data = new T[capacity <<= 1];
    for (int i = 0; i < size; i++)
        data[i] = oldData[i];
    delete [] oldData;
}
```

__缩容算法__
判断当前向量的元素数量size是否小于容量的1/4，如果小于，则缩容一半

In [13]:
```
template <class T>
void Vector<T>::shrink(){
    if (size > capacity >> 2) return;
    T *oldData = data;
    data = new T[capacity >>= 1];
    for (int i = 0; i < size; i++)
        data[i] = oldData[i];
    delete [] oldData;
}
```

三、元素的插入和删除

__插入算法__
在下标r的位置插入元素e，返回值：插入的位置r
这里返回的位置r不清楚后面用于什么用途，暂时先不管

In [14]:
```
template <class T>
int Vector<T>::insert(int r, T e){
    expand();
    for (int i = size; i > r; i--)
        data[i] = data[i-1];
    data[r] = e;
    size++;
    return r;
}
```

In [15]:
```
v3.insert(2,90);
cout << v3[2];
```

```
90
```

__删除算法__
删除下标为r的元素，并返回其值，
在选择排序中，选出前缀中最大的元素，删除后需要获得其值，并插入后缀序列中
注： 这里实现的是删除单个元素的算法，如果删除某个区间元素，调用此算法一个一个删，
每次都会搬运一堆元素，效率低下。
课本上实现的是删除区间的算法，由于这里仅介绍删除逻辑，只给出最简单的实现，

In [16]:
```
template <class T>
int Vector<T>::remove(int r){
    shrink();
    T e = data[r];
    for (int i = r; i < size-1; i++)
        data[i] = data[i+1];
    size--;
    return e;
}
```

In [17]:
```
v3.remove(2);
cout << v3[2];
```

```
3
```

四、查找

查找接口分为两种，一种是无序向量的查找find，一种是有序向量的查找search，
find的实现较为简单，就是挨个元素遍历过去，这种我们就不实现了，因此，我们
后面各种数据结构实现的查找接口都是search
返回值： 所查找元素的位置

__二分查找算法__
由于元素已经排好序了，我们看向量中间的元素是不是我们要找的，如果不是，
则要找的向量要么在左边，要么在右边，这样递归找下去直到找到,如果找不到，返回-1

In [18]:
```
template <class T>
int Vector<T>::search(T e, int lo, int hi){
    while (lo < hi) {
        int mi = (lo + hi) >> 1;
        if (e < data[mi]) hi = mi;
```

```
        else if (e > data[mi]) lo = mi + 1;
        else return mi;
    }
    return -1;
}
```

In [19]:
```
int b[] = {1, 2, 3, 4, 5, 6};
auto v4 = Vector<int>(b,0,5);
v4.search(2,0,6)
```

Out[19]: 1

五、排序

前面涉及的排序算法主要为：选择排序，插入排序，归并排序
选择排序：假定后缀序列有序，前缀序列无序，从前缀序列选择最大元素，插入后缀序列第一个，然后删除前缀序列中的那个元素；
插入排序：假定前缀序列有序，后缀序列无序，把后缀序列第一个删除弹出，并插入前缀序列相应的位置；
归并排序：将一串序列拆分成两个，假设这两串子序列分别有序，然后合并成一大串有序的序列。
由于插入和选择排序涉及到元素的插入和删除操作，这个对于向量来说需要不停搬运，时间复杂度太大，因此向量这里我们只介绍归并排序

In [20]:
```
template <class T>
void Vector<T>::merge(int lo, int mi, int hi){
    int len1 = mi - lo;
    T* temp = new T[len1];  // 备份前缀序列
    for (int k = 0; k < len1; k++) temp[k] = data[lo + k];

    for (int i= 0, j = mi, k = lo; i < len1 && j < hi;){
        if (temp[i] < data[j]) {data[k] = temp[i];i++;}
        else {data[k] = data[j];j++;}
        k++;
    }

    delete [] temp;
}

template <class T>
void Vector<T>::mergeSort(int lo, int hi) {
    if (hi - lo == 1) return; // 单元素向量，自然有序
    int mi = (lo + hi) >> 1;
    mergeSort(lo,mi);
    mergeSort(mi,hi);
    merge(lo,mi,hi);
}
```

In [21]:
```
int c[] = {2, 5, 1, 4, 6, 3};
auto v5 = Vector<int>(c,0,5);
v5.mergeSort(0,5);
```

In [22]:
```
for (int i = 0; i < v5.size; i++)
    cout << v5[i] << " ";
```

```
1 2 4 5 6
```

In [ ]: