|  |  |
| --- | --- |
| 实验序号 | 5 |
| 学号 |  |
| 姓名 |  |
| 成绩 |  |
| 截止日期 |  |
| 批阅 |  |

1. 问题说明

有一个高考数据文件（P05\_TestData\*\*\*.in）保存了某省的全部高考数据。每个考生的数据占一行，要求实现高考录取程序（实在搞不定可以用vector库，不过用了扣1分）。

【高考数据组织如下】：

准考证号 性别 总分 志愿1（学校编号） 志愿2（学校编号）

其中，性别0代表女性，1代表男性。

样例：

301000000 0 41 4 15

301000001 1 500 13 7

有一个学校录取计划文件(university.in)，每一行保存了学校的录取计划，每一行组织如下：

学校编号 学校名称 计划招生人数

样例：

1 北京大学 105

录取规则如下：

1.优先录取一志愿。

2.根据总分，总分高的优先录取。

3.如果总分相同，则优先考虑女生。

4.如果总分相同，性别也相同，则根据准考证号码，准考证小的优先。

5.每个考生只录取一次。如已经被某一所大学录取，不能再被录取。

6.所有学校的录取分为580。

【输出要求】

要求输出结果生成两个文件。一个文件保存每个大学录取的考生，一个文件保存每个考生的录取情况。

【文件一】

一个文件是每个大学所录取的考生（P05\_TestData\*\*\*\_school.out）。其格式如下：

学校代码（按代码大小顺序排序） 学校名称 计划录取人数 实际录取人数 录取考生准考证（按照准考证号码大小排序）

比如：

1 北京大学 105 2 301000002 301000398

2 清华大学 102 3 301000075 301000274 301002047

【文件二】

另一个输出文件是面向考生的(P05\_TestData\*\*\*\_student.out)，记录每位考生被哪所大学录取,它供考生查询使用，按准考证号大小排序输出。其格式如下：

准考证号（按大小排序） 性别 总分 录取状态（1录取0没录取） 录取学校代码（若没录取输出0）

比如：

301000004 0 745 1 7

301000146 1 723 1 11

二．数据结构说明

template<typename T>  
class vector {  
public:  
 // 默认构造函数  
 vector() : m\_size(0), m\_capacity(1), m\_data(new T[m\_capacity]) {}  
  
 // 构造函数  
 vector(int size) : m\_size(size), m\_capacity(size), m\_data(new T[m\_capacity]) {}  
  
  
 vector(int size, const T &val) : m\_size(size), m\_capacity(size), m\_data(new T[m\_capacity]) {  
 for (int i = 0; i < m\_size; i++) {  
 m\_data[i] = val;  
 }  
 }  
  
 // 拷贝构造函数  
 vector(const vector &other) : m\_size(other.m\_size), m\_capacity(other.m\_capacity), m\_data(new T[m\_capacity]) {  
 for (int i = 0; i < m\_size; i++) {  
 m\_data[i] = other.m\_data[i];  
 }  
 }  
  
 // 析构函数  
 ~vector() {  
 delete[] m\_data;  
 }  
  
 // 复制构造函数  
 vector &operator=(const vector &other) {  
 if (this != &other) {  
 delete[] m\_data;  
 m\_size = other.m\_size;  
 m\_capacity = other.m\_capacity;  
 m\_data = new T[m\_capacity];  
 for (int i = 0; i < m\_size; i++) {  
 m\_data[i] = other.m\_data[i];  
 }  
 }  
 return \*this;  
 }  
  
 // 返回数组大小  
 int size() const {  
 return m\_size;  
 }  
  
 // 返回数组容量  
 int capacity() const {  
 return m\_capacity;  
 }  
  
 // 判断数组是否为空  
 bool empty() const {  
 return m\_size == 0;  
 }  
  
 // 重载下标运算符  
 T &operator[](int index) {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // at函数 超出范围抛出异常  
 T &at(int index) {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
   
 // 重载下标运算符返回常引用  
 const T &operator[](int index) const {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // at函数 超出范围抛出异常 返回常引用  
 const T &at(int index) const {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // 返回数组首地址  
 T \*begin() {  
 return m\_data;  
 }  
  
 // 返回数组尾地址  
 T \*end() {  
 return m\_data + m\_size;  
 }  
  
 // 尾部插入元素  
 void push\_back(const T &val) {  
 if (m\_size == m\_capacity) {  
 reserve(m\_capacity \* 2);  
 }  
 m\_data[m\_size++] = val;  
 }  
  
 // 尾部删除元素  
 void pop\_back() {  
 if (m\_size > 0) {  
 m\_size--;  
 }  
 }  
  
 // 保持足够容量  
 void reserve(int newCapacity) {  
 // 如果新容量大于当前容量 那么重新分配内存  
 if (newCapacity > m\_capacity) {  
 T \*newData = new T[newCapacity]; // 创建新数组  
 for (int i = 0; i < m\_size; i++) { // 将原数组中的元素拷贝到新数组中  
 newData[i] = m\_data[i];  
 }  
 delete[] m\_data; // 释放原数组内存  
 m\_data = newData; // 将m\_data指向新数组  
 m\_capacity = newCapacity; // 更新容量  
 }  
 }  
  
 // 重新设置数组大小  
 void resize(int newSize) {  
 if (newSize > m\_capacity) { // 如果新大小大于当前容量 那么扩容  
 reserve(newSize);  
 }  
 for (int i = m\_size; i < newSize; i++) { // 将新元素初始化为默认值  
 m\_data[i] = T();  
 }  
 m\_size = newSize; // 更新大小  
 }  
  
private:  
 int m\_size; // 数组大小  
 int m\_capacity; // 数组容量  
 T \*m\_data; // 数组首地址  
};

三．算法说明

#include <iostream>  
#include <fstream>  
#include <chrono>  
#include <functional>  
#include <exception>  
  
namespace my {  
 // 二叉搜索树  
 template<typename T, typename Compare = std::less<T> >  
 // Compare 为比较函数对象，用于比较两个元素的大小  
 class BinaryTree {  
 private:  
 // 节点类  
 struct node {  
 T data; // 数据  
 node \*left; // 左子树  
 node \*right; // 右子树  
 node \*parent; // 父节点  
  
 // 构造函数  
 explicit node(const T &d, node \*l = NULL, node \*r = NULL, node \*p = NULL) : data(d), left(l), right(r),  
 parent(p) {}  
 };  
  
 node \*root; // 根节点  
 Compare cmp; // 比较函数对象  
  
 public:  
 // 空构造函数  
 // 将节点设置为空 并且传入比较器  
 BinaryTree() : root(NULL), cmp(Compare()) {}  
  
 // 析构函数  
 ~BinaryTree() {  
 clear();  
 }  
  
 // 清空树  
 void clear() {  
 clear(root);  
 }  
  
 // 清空当前节点下的树  
 void clear(node \*t) {  
 if (t == NULL) return;  
 clear(t->left);  
 clear(t->right);  
 delete t;  
 }  
  
 // 查找元素  
 node \*find(const T &x) const {  
 return find(x, root);  
 }  
  
 // 在当前节点下查找元素  
 node \*find(const T &x, node \*t) const {  
 if (t == NULL || t->data == x) return t; // 如果当前节点为空那么返回NULL或者当前节点的值等于x 返回当前节点  
 if (cmp(x, t->data)) return find(x, t->left); // 如果x小于当前节点的值 那么在左子树中查找  
 else return find(x, t->right); // 否则在右子树中查找  
 }  
  
 // 判断是否包含元素  
 bool contain(const T &x) const {  
 return find(x) != NULL;  
 }  
  
 // 插入元素  
 bool insert(const T &x) {  
 if (root == NULL) { // 如果根节点为空 那么直接插入  
 root = new node(x); // 创建新节点  
 return true; // 创建新节点返回true  
 }  
 node \*t = root; // 否则从根节点开始查找  
 while (true) {  
 if (t->data == x) return false; // 如果有原来节点 那么返回false  
 if (cmp(x, t->data)) { // 如果x小于当前节点的值  
 if (t->left == NULL) { // 如果左子树为空  
 t->left = new node(x, NULL, NULL, t); // 创建新节点  
 return true;  
 }  
 t = t->left; // 否则继续在左子树中查找  
 } else {  
 if (t->right == NULL) { // 如果右子树为空  
 t->right = new node(x, NULL, NULL, t); // 创建新节点  
 return true;  
 }  
 t = t->right; // 否则继续在右子树中查找  
 }  
 }  
 }  
  
  
 bool remove(const T &x) {  
 node \*t = find(x);  
 if (t == NULL) return false;  
 if (t->left != NULL && t->right != NULL) {  
 node \*tmp = t->right;  
 while (tmp->left != NULL) tmp = tmp->left; //  
 t->data = tmp->data;  
 t = tmp;  
 }  
 node \*child = t->left != NULL ? t->left : t->right;  
 if (t == root) {  
 root = child;  
 if (root != NULL) root->parent = NULL;  
 } else {  
 node \*parent = t->parent;  
 if (parent->left == t) parent->left = child;  
 else parent->right = child;  
 if (child != NULL) child->parent = parent;  
 }  
 delete t;  
 return true;  
 }  
 };  
  
 // 求C风格字符串长度  
 size\_t strlen(const char \*s) {  
 size\_t len = 0;  
 while (s[len] != '\0') {  
 len++;  
 }  
 return len;  
 }  
  
 // 拷贝C风格字符串  
 void strcpy(char \*dest, const char \*src) {  
 size\_t i = 0;  
 while (src[i] != '\0') {  
 dest[i] = src[i];  
 i++;  
 }  
 dest[i] = '\0';  
 }  
  
 // 判断是否为空白字符  
 bool isspace(char c) {  
 return c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\f' || c == '\v';  
 }  
  
 // 求最小值  
 template<typename T>  
 T min(const T &a, const T &b) {  
 return a < b ? a : b;  
 }  
  
 // 求最大值  
 template<typename T>  
 T max(const T &a, const T &b) {  
 return a > b ? a : b;  
 }  
  
 // 字符串类  
 class string {  
 public:  
  
 // 默认构造函数 构造一个不包含任何字符的字符串  
 string() : m\_data(new char[1]), m\_size(0), m\_capacity(1) {  
 m\_data[0] = '\0';  
 }  
  
 // 构造函数 从C风格字符串中构造字符串  
 explicit string(const char \*str) : m\_data(new char[my::strlen(str) + 1]), m\_size(my::strlen(str)),  
 m\_capacity(my::strlen(str) + 1) {  
 my::strcpy(m\_data, str);  
 }  
  
 // 拷贝构造函数  
 string(const string &other) : m\_data(new char[other.m\_capacity]), m\_size(other.m\_size),  
 m\_capacity(other.m\_capacity) {  
 my::strcpy(m\_data, other.m\_data);  
 }  
  
 // 析构函数  
 ~string() {  
 delete[] m\_data;  
 }  
  
 // 清空字符串  
 void erase() {  
 delete[] m\_data;  
 m\_data = new char[1];  
 m\_data[0] = '\0';  
 m\_size = 0;  
 m\_capacity = 1;  
 }  
  
 // 追加C风格字符串  
 void append(const char \*str, size\_t n) {  
 if (m\_size + n + 1 > m\_capacity) {  
 reserve(m\_size + n + 1);  
 }  
 my::strcpy(m\_data + m\_size, str);  
 m\_size += n;  
 m\_data[m\_size] = '\0';  
 }  
  
 // 复制构造函数  
 string &operator=(const string &other) {  
 if (this != &other) {  
 char \*temp = new char[other.m\_capacity];  
 my::strcpy(temp, other.m\_data);  
 delete[] m\_data;  
 m\_data = temp;  
 m\_size = other.m\_size;  
 m\_capacity = other.m\_capacity;  
 }  
 return \*this;  
 }  
  
 // 追加字符串  
 string operator+(const string &other) const {  
 string newString;  
 newString.m\_size = m\_size + other.m\_size;  
 newString.m\_capacity = newString.m\_size + 1;  
 newString.m\_data = new char[newString.m\_capacity];  
 my::strcpy(newString.m\_data, m\_data);  
 my::strcpy(newString.m\_data + m\_size, other.m\_data);  
 return newString;  
 }  
  
 // 散列仿函数  
 class hash {  
 public:  
 size\_t operator()(const string &str) const {  
 size\_t hash = 0;  
 for (size\_t i = 0; i < str.size(); ++i) {  
 hash = hash \* 131 + str[i];  
 }  
 return hash;  
 }  
 };  
  
 // 相等运算符  
 bool operator==(const string &other) const {  
 if (m\_size != other.m\_size) { // 如果长度不相等 那么直接返回false  
 return false;  
 }  
 for (size\_t i = 0; i < m\_size; ++i) { // 否则逐个比较字符  
 if (m\_data[i] != other.m\_data[i]) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 bool operator!=(const string &other) const {  
 return !(\*this == other);  
 }  
  
 // 小于运算符  
 bool operator<(const string &other) const {  
 size\_t minSize = min(m\_size, other.m\_size); // 求最小长度  
 for (size\_t i = 0; i < minSize; ++i) {  
 if (m\_data[i] < other.m\_data[i]) { // 逐个比较字符  
 return true;  
 } else if (m\_data[i] > other.m\_data[i]) {  
 return false;  
 }  
 }  
 return m\_size < other.m\_size; // 如果前面的字符都相等 那么长度小的字符串小  
 }  
  
 // operator >  
 bool operator>(const string &other) const {  
 return other < \*this;  
 }  
  
 // operator <=  
 bool operator<=(const string &other) const {  
 return !(other < \*this);  
 }  
  
 // operator >=  
 bool operator>=(const string &other) const {  
 return !(\*this < other);  
 }  
  
 // 取出字符串中的字符  
 char &operator[](size\_t index) {  
 return m\_data[index];  
 }  
  
 // 取出字符串中的字符  
 const char &operator[](size\_t index) const {  
 return m\_data[index];  
 }  
  
 // 返回字符串长度  
 size\_t size() const {  
 return m\_size;  
 }  
  
 // 返回字符串容量  
 size\_t capacity() const {  
 return m\_capacity;  
 }  
  
 // 扩容  
 void reserve(size\_t new\_capacity) {  
 if (new\_capacity > m\_capacity) {  
 char \*temp = new char[new\_capacity];  
 my::strcpy(temp, m\_data);  
 delete[] m\_data;  
 m\_data = temp;  
 m\_capacity = new\_capacity;  
 }  
 }  
  
 // 重新设置字符串长度  
 void resize(size\_t new\_size) {  
 if (new\_size > m\_capacity) {  
 reserve(new\_size);  
 }  
 for (size\_t i = m\_size; i < new\_size; ++i) {  
 m\_data[i] = '\0';  
 }  
 m\_size = new\_size;  
 }  
  
 // 在字符串末尾追加字符  
 void push\_back(char c) {  
 if (m\_size + 1 > m\_capacity) {  
 reserve(m\_capacity \* 2);  
 }  
 m\_data[m\_size] = c;  
 m\_data[m\_size + 1] = '\0';  
 ++m\_size;  
 }  
  
 // 返回C风格字符串  
 const char \*c\_str() const {  
 return m\_data;  
 }  
  
 // C++流输出运算符  
 friend std::ostream &operator<<(std::ostream &os, const string &str) {  
 os << str.m\_data;  
 return os;  
 }  
  
 // C++流输入运算符  
 friend std::istream &operator>>(std::istream &is, string &str) {  
 char c = '\0'; // 将str中的数据清空  
 while (is.get(c) && my::isspace(c)); // 跳过空白字符  
 if (is) { // 如果输入流正常  
 str.m\_size = 0; // 将str中的数据清空  
 do {  
 if (str.m\_size + 1 > str.m\_capacity) { // 如果空间不足 那么扩容  
 str.reserve(str.m\_capacity \* 2);  
 }  
 str.m\_data[str.m\_size] = c;  
 ++str.m\_size;  
 } while (is.get(c) && !my::isspace(c)); // 读取非空白字符  
 str.m\_data[str.m\_size] = '\0'; // 在末尾添加'\0'  
 if (is) is.unget(); // 如果输入流正常 那么将最后读取的字符放回输入流  
 else is.clear(is.rdstate() & ~std::ios\_base::failbit); // 否则清除错误标志  
 }  
 return is;  
 }  
  
 private:  
 char \*m\_data; // 字符串数据  
 size\_t m\_size; // 字符串长度  
 size\_t m\_capacity; // 字符串容量  
 };  
  
 template<typename T>  
 void swap(T &a, T &b) {  
 T tmp = a;  
 a = b;  
 b = tmp;  
 }  
  
 // 划分函数  
 template<class RandomIt, class Compare = std::less<> >  
 RandomIt partition(RandomIt first, RandomIt last, Compare comp) {  
 RandomIt pivot = last - 1; // 枢轴  
 RandomIt i = first; // i指向小于枢轴的最后一个元素  
 for (RandomIt j = first; j < pivot; ++j) { //  
 if (comp(\*j, \*pivot)) { // 如果j指向的元素小于枢轴  
 my::swap(\*i, \*j); // 交换i和j指向的元素  
 ++i; // i向后移动一位  
 }  
 }  
 my::swap(\*i, \*pivot); // 将枢轴放到正确的位置  
 return i;  
 }  
  
 // 快速排序  
 template<class RandomIt, class Compare = std::less<> >  
 // Compare 为比较函数对象，用于比较两个元素的大小  
 void quickSort(RandomIt first, RandomIt last, Compare comp) {  
 if (first < last) { // 如果first < last 那么继续排序  
 RandomIt pivot = my::partition(first, last, comp); // 求枢轴  
 quickSort(first, pivot, comp); // 对左半部分排序  
 quickSort(pivot + 1, last, comp); // 对右半部分排序  
 }  
 }  
  
  
 // 自动扩容的数组  
 template<typename T>  
 class vector {  
 public:  
 // 默认构造函数  
 vector() : m\_size(0), m\_capacity(1), m\_data(new T[m\_capacity]) {}  
  
 // 构造函数  
 vector(int size) : m\_size(size), m\_capacity(size), m\_data(new T[m\_capacity]) {}  
  
  
 vector(int size, const T &val) : m\_size(size), m\_capacity(size), m\_data(new T[m\_capacity]) {  
 for (int i = 0; i < m\_size; i++) {  
 m\_data[i] = val;  
 }  
 }  
  
 // 拷贝构造函数  
 vector(const vector &other) : m\_size(other.m\_size), m\_capacity(other.m\_capacity), m\_data(new T[m\_capacity]) {  
 for (int i = 0; i < m\_size; i++) {  
 m\_data[i] = other.m\_data[i];  
 }  
 }  
  
 // 析构函数  
 ~vector() {  
 delete[] m\_data;  
 }  
  
 // 复制构造函数  
 vector &operator=(const vector &other) {  
 if (this != &other) {  
 delete[] m\_data;  
 m\_size = other.m\_size;  
 m\_capacity = other.m\_capacity;  
 m\_data = new T[m\_capacity];  
 for (int i = 0; i < m\_size; i++) {  
 m\_data[i] = other.m\_data[i];  
 }  
 }  
 return \*this;  
 }  
  
 // 返回数组大小  
 int size() const {  
 return m\_size;  
 }  
  
 // 返回数组容量  
 int capacity() const {  
 return m\_capacity;  
 }  
  
 // 判断数组是否为空  
 bool empty() const {  
 return m\_size == 0;  
 }  
  
 // 重载下标运算符  
 T &operator[](int index) {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // at函数 超出范围抛出异常  
 T &at(int index) {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // 重载下标运算符返回常引用  
 const T &operator[](int index) const {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // at函数 超出范围抛出异常 返回常引用  
 const T &at(int index) const {  
 if (index < 0 || index >= m\_size) {  
 throw std::out\_of\_range("Index out of range");  
 }  
 return m\_data[index];  
 }  
  
 // 返回数组首地址  
 T \*begin() {  
 return m\_data;  
 }  
  
 // 返回数组尾地址  
 T \*end() {  
 return m\_data + m\_size;  
 }  
  
 // 尾部插入元素  
 void push\_back(const T &val) {  
 if (m\_size == m\_capacity) {  
 reserve(m\_capacity \* 2);  
 }  
 m\_data[m\_size++] = val;  
 }  
  
 // 尾部删除元素  
 void pop\_back() {  
 if (m\_size > 0) {  
 m\_size--;  
 }  
 }  
  
 // 保持足够容量  
 void reserve(int newCapacity) {  
 // 如果新容量大于当前容量 那么重新分配内存  
 if (newCapacity > m\_capacity) {  
 T \*newData = new T[newCapacity]; // 创建新数组  
 for (int i = 0; i < m\_size; i++) { // 将原数组中的元素拷贝到新数组中  
 newData[i] = m\_data[i];  
 }  
 delete[] m\_data; // 释放原数组内存  
 m\_data = newData; // 将m\_data指向新数组  
 m\_capacity = newCapacity; // 更新容量  
 }  
 }  
  
 // 重新设置数组大小  
 void resize(int newSize) {  
 if (newSize > m\_capacity) { // 如果新大小大于当前容量 那么扩容  
 reserve(newSize);  
 }  
 for (int i = m\_size; i < newSize; i++) { // 将新元素初始化为默认值  
 m\_data[i] = T();  
 }  
 m\_size = newSize; // 更新大小  
 }  
  
 private:  
 int m\_size; // 数组大小  
 int m\_capacity; // 数组容量  
 T \*m\_data; // 数组首地址  
 };  
}  
  
struct University {  
 int id;  
 std::string name;  
 int capacity;  
 int get{0};  
 std::vector<int> stu;  
  
 University() = default;  
  
 University(int id, const std::string &name, int capacity) : id(id), name(name), capacity(capacity) {}  
  
};  
  
struct Student {  
 int id;  
 int gender;  
 int score;  
 int w1, w2;  
 int get{0};  
  
  
 Student() = default;  
  
 Student(int id, int gender, int score, int w1, int w2) : id(id), gender(gender), score(score), w1(w1), w2(w2) {}  
  
  
};  
  
int main() {  
 auto begin = std::chrono::steady\_clock::now();  
  
 std::ios::sync\_with\_stdio(false), std::cin.tie(nullptr), std::cout.tie(nullptr);  
  
  
 std::ifstream in1("university.in");  
 std::ifstream in2("P05\_TestData500000.in");  
 std::ofstream out1("P05\_500000ch.out");  
 std::ofstream out2("P05\_500000stu.out");  
 if (!in1.is\_open() || !in2.is\_open() || !out1.is\_open() || !out2.is\_open()) {  
 std::cout << "Error opening files" << std::endl;  
 return 1;  
 }  
  
 my::vector<University> us;  
 {  
 int id, capacity;  
 std::string name;  
 while (in1 >> id >> name >> capacity) {  
 us.push\_back(University(id, name, capacity));  
 }  
 }  
 my::vector<Student> ss;  
 {  
 int id, gender, score, w1, w2;  
 while (in2 >> id >> gender >> score >> w1 >> w2) {  
 ss.push\_back(Student(id, gender, score, w1, w2));  
 }  
 }  
  
 my::quickSort(ss.begin(), ss.end(),  
 [](const Student &a, const Student &b) {  
 if (a.score != b.score) return a.score > b.score;  
 if (a.gender != b.gender) return a.gender < b.gender;  
 return a.id < b.id;  
 });  
  
 for (auto &s: ss) {  
 int school = s.w1;  
 if (us.at(school - 1).get < us.at(school - 1).capacity && s.score >= 580) {  
 us.at(school - 1).get++;  
 us.at(school - 1).stu.emplace\_back(s.id);  
 s.get = school;  
 continue;  
 }  
 }  
  
  
 for (auto &s: ss) {  
 if (s.get != 0) continue;  
 int school = s.w2;  
 if (us.at(school - 1).get < us.at(school - 1).capacity && s.score >= 580) {  
 us.at(school - 1).get++;  
 us.at(school - 1).stu.emplace\_back(s.id);  
 s.get = school;  
 continue;  
 }  
 }  
  
  
 for (auto &i: us) {  
 out1 << i.id << " " << i.name << " " << i.capacity << " " << i.get << " ";  
 my::quickSort(i.stu.begin(), i.stu.end(),  
 [](auto a, auto b) {  
 return a < b;  
 });  
 for (auto j: i.stu) {  
 out1 << j << " ";  
 }  
 out1 << "\n";  
 }  
  
 my::quickSort(ss.begin(), ss.end(),  
 [](const Student &a, const Student &b) {  
 return a.id < b.id;  
 });  
 for (auto &i: ss)  
 out2 << i.id << " " << i.gender << " " << i.score << " " << (i.get != 0) << " " << i.get << "\n";  
  
 auto end = std::chrono::steady\_clock::now();  
 std::cout << "Time: " << std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin).count() << "ms"  
 << std::endl;  
 return 0;  
}

四．结果说明

程序运行时间：

|  |  |  |
| --- | --- | --- |
| P05\_TestData500.in | P05\_TestData5000.in | *P05\_TestData500000.in* |
| 1ms | 6ms | 1177ms |

程序运行结果：

|  |  |
| --- | --- |
| P05\_TestData500.in | 100% |
| P05\_TestData5000.in | 100% |
| P05\_TestData500000.in | 100% |

五．结论和总结

1. 该程序是一个高考录取程序，它读取高考考生数据文件和学校录取计划文件，并根据一定的录取规则为考生分配大学录取结果。

2. 考生数据和学校录取计划数据都是以文本文件的形式进行存储和读取的。程序通过文件输入输出实现了数据的读取和结果的保存。

3. 程序使用了自定义的向量（vector）数据结构来存储考生和学校的数据。这个自定义向量类实现了基本的向量操作，如添加元素、删除元素、大小调整等。这种数据结构类似于C++标准库中的`std::vector`，但是在本程序中使用自定义的向量类。

4. 录取规则包括了优先录取第一志愿、根据总分和性别进行排序，以及其他细节规则。这些规则都被程序严格实现，确保了录取的公平性和合理性。

5. 录取结果生成了两个输出文件，一个是每个大学所录取的考生的文件，另一个是面向考生的录取情况文件。这两个文件提供了方便查询和查看录取结果的途径。

6. 程序中还包括了快速排序算法的实现，用于对考生进行排序，以满足录取规则中的总分和性别排序要求。

总的来说，该程序实现了一个完整的高考录取系统，包括数据读取、录取规则的实现、结果生成和存储，以及排序算法的应用。通过文件输入输出和自定义数据结构，程序成功处理了大规模高考数据和录取计划，为每个考生分配了录取结果，确保了录取的公平性和合理性。