

第七章 模板与泛型编程

2020 年 10 月 26 日

一般人员只注重简单功能的简单完成。



一般人员只注重简单功能的简单完成。



学习目标

- ① 掌握模板的定义和基本使用方法，包括函数模板和类模板；
- ② 学会运用模板实现泛型编程；
- ③ 掌握常用排序算法和二分查找算法。

7.2 类模板

类似函数模板，可以定义一个类模板用来生成具有相同结构的一族类实例：

Array 类模板定义

```
template<typename T, size_t N>
class Array {
    T m_ele[N];
public:
    Array() {}
    Array(const std::initializer_list<T> &);
    T& operator[](size_t i);
    constexpr size_t size() { return N; }
};
```

说明

- 类型参数 `T` 和非类型参数 `N`，分别用来表示元素的类型和元素的数目
- `initializer_list` 类型是 C++11 标准库提供的新类型，支持具有相同类型但数量未知的列表类型

7.2.1 成员函数定义

与普通类相同，既可以在类的内部，也可以在类的外部定义类模板成员函数：

Array 类模板 构造函数 类外定义

```
template<typename T, size_t N>
Array<T, N>::Array(const std::initializer_list<T> &l):m_ele
{ T() }{
    size_t m = l.size() < N ? l.size() : N;
    for (size_t i = 0; i < m; ++i) {
        m_ele[i] = *(l.begin() + i);
    }
}
```

说明

- 必须以关键字 `template` 开始，后接与类模板相同的模板参数列表
- 紧随类名后面的参数列表代表一个实例化的实参列表，每个参数不需要 `typename` 或 `class` 说明符

7.2.1 成员函数定义

与普通类相同，既可以在类的内部，也可以在类的外部定义类模板成员函数：

Array 类模板 构造函数 类外定义

```
template<typename T, size_t N>
Array<T, N>::Array(const std::initializer_list<T> &l):m_ele
{ T() } {
    size_t m = l.size() < N ? l.size() : N;
    for (size_t i = 0; i < m; ++i) {
        m_ele[i] = *(l.begin() + i);
    }
}
```

说明

- m_ele 中的每一个元素用 T 类型的默认初始化方式初始化
- 将形参 l 中的元素依次复制
- l.begin 返回列表 l 中第一个元素的迭代器

7.2.1 成员函数定义

与普通类相同，既可以在类的内部，也可以在类的外部定义类模板成员函数：

Array 类模板 构造函数 类外定义

```
template<typename T, size_t N>
Array<T, N>::Array(const std::initializer_list<T> &l):m_ele
{
    {T()}{
        size_t m = l.size() < N ? l.size() : N;
        for (size_t i = 0; i < m; ++i) {
            m_ele[i] = *(l.begin() + i);
        }
    }
}
```

说明

- m_ele 中的每一个元素用 T 类型的默认初始化方式初始化
- 将形参 l 中的元素依次复制
- l.begin 返回列表 l 中第一个元素的迭代器

Array 类模板 [] 运算符函数 类外定义

```
template<typename T, size_t N>
T& Array<T, N>::operator[](size_t i) {
    return m_ele[i];
}
```

说明

类模板的下标运算符函数返回数组 m_ele 中第 i 个元素的引用

7.2.2 实例化类模板

当使用一个类模板时，我们需要显式提供模板参数信息，即模板实参列表：

实例化 Array 类模板

```
Array<char, 5> a; //创建一个Array<char, 5>类型对象 a  
Array<int, 5> b = {1,2,3}; //创建一个Array<int, 5>类型对象 b
```

说明

创建对象 b 时，将执行具有形参的构造函数，其形参 1 接受初始化列表 {1,2,3}，其余元素具有默认值 0

7.2.2 实例化类模板

当使用一个类模板时，我们需要显式提供模板参数信息，即模板实参列表：

实例化 Array 类模板

```
Array<char, 5> a; //创建一个Array<char, 5>类型对象 a  
Array<int, 5> b = {1,2,3}; //创建一个Array<int, 5>类型对象 b
```

下面代码逐个输出对象 b 的每一个元素：

```
for (int i = 0; i < b.size(); ++i)  
    cout << b[i] << " ";
```

输出结果为：1 2 3 0 0

说明

创建对象 b 时，将执行具有形参的构造函数，其形参 1 接受初始化列表 {1,2,3}，其余元素具有默认值 0

7.2.3 默认模板参数

函数参数可以具有默认值，模板参数同样也可以有默认值：

Array 类模板定义二

```
template<typename T = int, size_t N = 10>
class Array {
    // 其它成员保持不变
};
```

说明

- 类模板参数 T 具有默认类型
int
- 类模板参数 N 具有默认值
10

7.2.3 默认模板参数

函数参数可以具有默认值，模板参数同样也可以有默认值：

Array 类模板定义二

```
template<typename T = int, size_t N = 10>
class Array {
    // 其它成员保持不变
};
```

实例化 Array 类模板二

```
Array<> a = { 'A' };
cout << a.size() << " " << a[0] << endl;
```

输出结果为：10 65

说明

- 类模板参数 T 具有默认类型 int
- 类模板参数 N 具有默认值 10

说明

- a 的元素数目为默认值 10
- a[0] 的类型为 int，字符 'A' 转换为 65

7.2.3 默认模板参数

为函数模板参数提供默认值：

Array 类模板定义三

```
template<typename T, size_t N = 10>
class Array {
    // 其它成员保持不变
public:
    template<typename F = Less<T>>
    void sort(F f = F());
};
```

说明

- 新增了一个成员函数模板 `sort`，用来对数组进行排序
- `sort` 的函数模板参数 `F` 具有默认值 `Less<T>`

7.2.3 默认模板参数

为函数模板参数提供默认值:

Array 类模板定义三

```
template<typename T, size_t N = 10>
class Array {
    // 其它成员保持不变
public:
    template<typename F = Less<T>>
    void sort(F f = F());
};
```

Less 类模板定义

```
template<typename T>
struct Less{
    bool operator()(const T &a, const T &b) {
        return a < b;
    }
};
```

说明

- 新增了一个成员函数模板 `sort`, 用来对数组进行排序
- `sort` 的函数模板参数 `F` 具有默认值 `Less<T>`

说明

类模板 `Less<T>` 具有一个模板参数 `T`, 且只有一个函数调用运算符, 该成员函数带有两个形参, 用来比较两个形参的大小, 返回值类型为 `bool`

7.2.3 默认模板参数

和类模板参数一样，函数模板参数也可以有默认值：

Array 类模板定义三

```
template<typename T, size_t N = 10>
class Array {
    // 其它成员保持不变
public:
    template<typename F = Less<T>>
    void sort(F f = F());
};
```

说明

- sort 的函数参数 f 也有默认值，即 F 类的一个函数对象，代表默认比较方式为 Less

7.2.3 默认模板参数

和类模板参数一样，函数模板参数也可以有默认值：

Array 类模板定义三

```
template<typename T, size_t N = 10>
class Array {
    // 其它成员保持不变
public:
    template<typename F = Less<T>>
    void sort(F f = F());
};
```

说明

- sort 的函数参数 f 也有默认值，即 F 类的一个函数对象，代表默认比较方式为 Less

问题

理清 函数参数和模板参数的概念

7.3.1 排序算法

排序是数据处理的最基本任务，目的是按照某种规则将一组无序数据重新排列，使之有序。

7.3.1 排序算法

排序是数据处理的最基本任务，目的是按照某种规则将一组无序数据重新排列，使之有序。

Array 类模板定义四

```
template<typename T, size_t N>
class Array {
public:
    template<typename F = Less<T> >
    void selectionSort(F f = F()); //选择排序
    template<typename F = Less<T> >
    void insertionSort(F f = F()); //插入排序
    template<typename F = Less<T> >
    void bubbleSort(F f = F());    //冒泡排序
private:
    void swap(int i, int j){
        T t = m_ele[i];
        m_ele[i] = m_ele[j];
        m_ele[j] = t;
    }
};
```

说明

成员 swap 函数用来交换两个元素的位置，它仅在 Array 类内部使用，因此它的访问属性为 private

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面



7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面



7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

-5	4	8	1	5
----	---	---	---	---

后四位中1最小

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

-5	1	8	4	5
----	---	---	---	---

1换到第二位

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

-5	1	8	4	5
----	---	---	---	---

1换到第二位

-5	1	8	4	5
----	---	---	---	---

后三位中4最小

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

-5	1	8	4	5
----	---	---	---	---

1换到第二位

-5	1	4	8	5
----	---	---	---	---

4换到第三位

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面



7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

-5	1	8	4	5
----	---	---	---	---

1换到第二位

-5	1	4	8	5
----	---	---	---	---

4换到第三位

-5	1	4	5	8
----	---	---	---	---

5换到第四位

7.3.1 排序算法 — 选择排序

每次在待排序元素中选择最小的一个，换放到已排序数列后面

5	4	8	1	-5
---	---	---	---	----

排序前

-5	4	8	1	5
----	---	---	---	---

-5换到第一位

-5	1	8	4	5
----	---	---	---	---

1换到第二位

-5	1	4	8	5
----	---	---	---	---

4换到第三位

-5	1	4	5	8
----	---	---	---	---

5换到第四位

-5	1	4	5	8
----	---	---	---	---

最后一位不变，完成排序

7.3.1 排序算法 — 选择排序

选择排序算法的实现如下：

Array 成员函数 selectionSort 定义

```
template<typename T, size_t N>
template<typename F >
void Array<T, N>::selectionSort(F f) {
    for (int i = 0; i < N - 1; ++i){
        int min = i;        // 记录待排序数据中最小元素位置
        for (int j = i + 1; j < N; ++j) {
            if (f(m_ele[j], m_ele[min]))
                min = j;    //更新最小元素位置
        }
        swap(i, min);      //把最小元素放到位置i
    }
}
```

说明

if 语句里的条件表达式将调用函数对象 f (Less<T>), 检查第一个实参对象是否小于第二个实参对象

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中



7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中



排序前



前一位为有序数列



1是第一个待插入元素

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中



排序前



前一位为有序数列



$1 < 5$, 往前移一位

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
1	5	4	-5	8
1	5	4	-5	8

前一位为有序数列

1到第一位，停止移动

4是第二个待插入元素

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

$4 < 5$, 往前移一位

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

4>1, 停止移动

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

4>1, 停止移动

1	4	5	-5	8
---	---	---	----	---

-5是第三个待插入元素

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

4>1, 停止移动

-5	1	4	5	8
----	---	---	---	---

$-5 < 1 < 4 < 5$ ，往前移三位

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

4>1, 停止移动

-5	1	4	5	8
----	---	---	---	---

-5到第一位，停止移动

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

4>1, 停止移动

-5	1	4	5	8
----	---	---	---	---

-5到第一位，停止移动

-5	1	4	5	8
----	---	---	---	---

8是第四个待插入元素

7.3.1 排序算法 — 插入排序

将待排序的元素逐个插入已经排好序的元素序列中

5	1	4	-5	8
---	---	---	----	---

排序前

5	1	4	-5	8
---	---	---	----	---

前一位为有序数列

1	5	4	-5	8
---	---	---	----	---

1到第一位，停止移动

1	4	5	-5	8
---	---	---	----	---

4>1, 停止移动

-5	1	4	5	8
----	---	---	---	---

-5到第一位，停止移动

-5	1	4	5	8
----	---	---	---	---

8>5, 停止移动，完成排序

7.3.1 排序算法 — 插入排序

插入排序算法的实现如下：

Array 成员函数 insertionSort 定义

```
template<typename T, size_t N>
template<typename F >
void Array<T, N>::insertionSort(F f) {
    for (int i = 1, j; i < N; ++i) {
        T t = m_ele[i];           //待插入元素
        for (j = i; j > 0; --j) { //查找插入位置
            if (f(m_ele[j - 1], t))
                break;
            m_ele[j] = m_ele[j - 1]; //逐个向后移动元素
        }
        m_ele[j] = t;              //将待插入元素放到正确位置
    }
}
```

7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换

5	1	4	-5	8
---	---	---	----	---

排序前

1	5	4	-5	8
---	---	---	----	---

交换

7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换

5	1	4	-5	8
---	---	---	----	---

排序前

1	5	4	-5	8
---	---	---	----	---

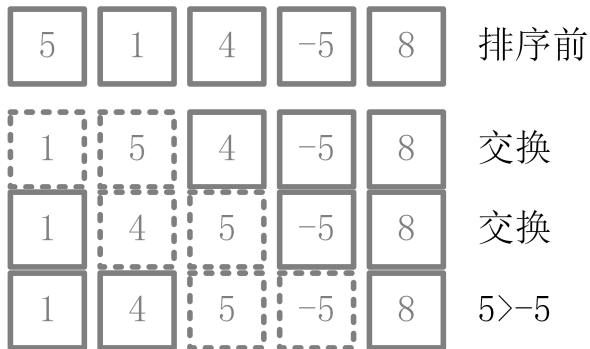
交换

1	4	5	-5	8
---	---	---	----	---

交换

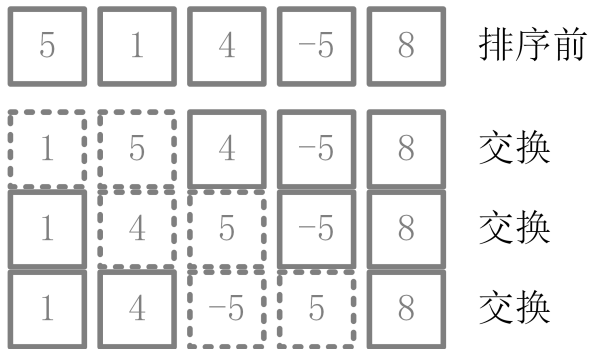
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



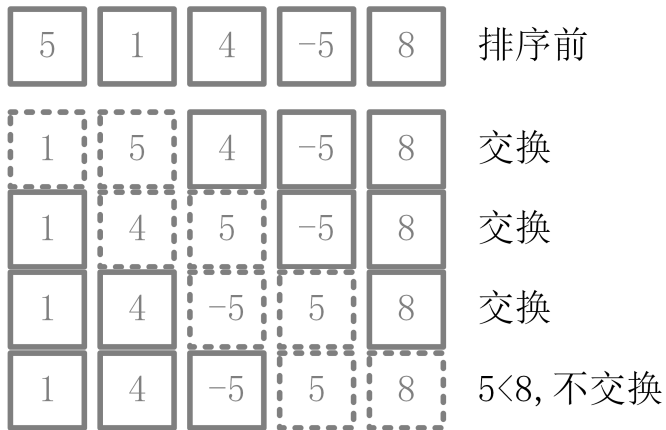
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



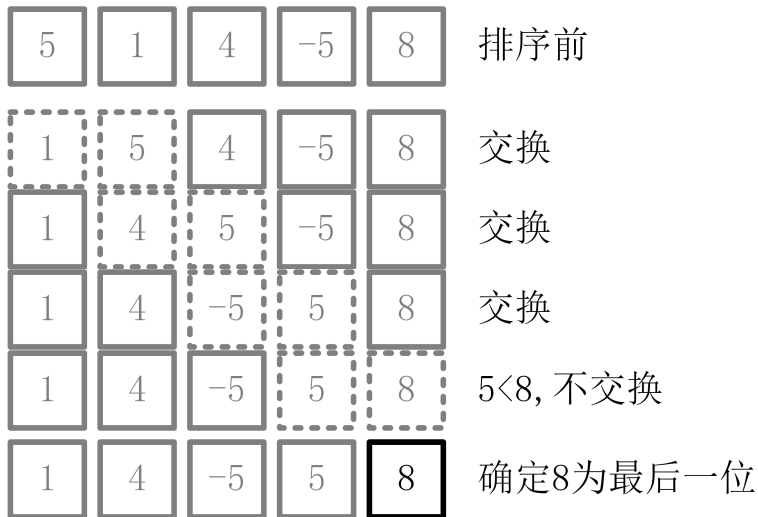
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



7.3.1 排序算法 — 冒泡排序

冒泡排序算法的实现如下：

Array 成员函数 selectionSort 定义

```
template<typename T, size_t N>
template<typename F >
void Array<T, N>::bubbleSort(F f){
    for (int i = N - 1; i >= 0; --i){
        for (int j = 0; j <= i - 1; ++j){
            if (f(m_ele[j + 1], m_ele[j]))
                swap(j, j + 1); //相邻元素交换
        }
    }
}
```

7.3.1 排序算法 — 快速排序

快速排序

快速排序是冒泡排序的改进，在排序过程中数据移动少。

7.3.1 排序算法 — 快速排序

快速排序

快速排序是冒泡排序的改进，在排序过程中数据移动少。

快速排序的基本思想

基本思想：划分和分治递归。

- ① 划分：将整个数组划分为两个部分，第一部分所有值小于基准值 (key)，第二部分所有值大于基准值 (key)。(基准值的选择是随机的，一般选择待排数组的第一个元素)。
- ② 分治递归：第一步将数组划分为两部分后，两部分内部还不是有序的，再分别对两部分递归地进行快速排序，最终得到一个完整的有序数组

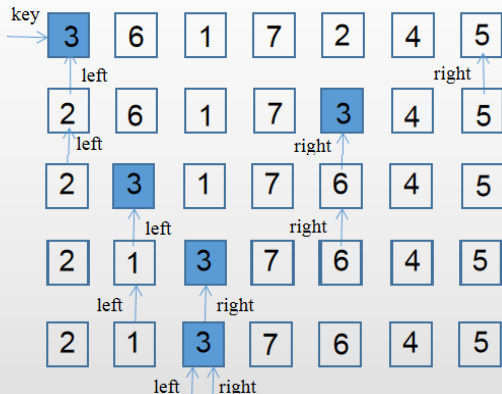
7.3.1 快速排序

快速排序的流程

- (1) left、right 指针（索引）分别指向待排数组的首、尾。
- (2) left 指针向后遍历，right 指针向前遍历。
- (3) 当 right 指针指向元素小于基准值（key）时，right 指针元素便赋值给 left 指针元素，完成转移。
- (4) 当 left 指针指向元素大于基准值（key）时，left 指针元素便赋值给 right 指针元素，完成转移。
- (5) 最终当 left=right 时，遍历结束。
- (6) 以基准值（key）为界限，把数组分成两部分，分别对这两部分进行快速排序（显然这是一个递归的过程）

7.3.1 快速排序

快速排序的举例



无序数组，第一个元素为基准值

right向左遍历，找出小于基准值的第一个元素2，交换right和left指向的元素

left向右遍历，找出大于基准值的第一个元素6，交换left和right指向的元素

right继续向左遍历，找出小于基准值的第一个元素1，交换right和left指向的元素

left继续向右遍历，此时left=right，以基准值为界，分两部分，继续进行快速排序

很棒！你听说过快速排序吗？



7.3.1 快速排序

快速排序的举例续



7.3.1 快速排序

问你问题要快速回答



课堂小测试

对下列序列进行快速排序，都以第一个元素为基准进行第一次划分，则在
该次划分过程中，需要移动元素次数最多的序列是（ ）

- (A) 1, 3, 5, 7, 9 (B) 9, 7, 5, 3, 1
(c) 5, 1, 3, 7, 9 (D) 5, 7, 9, 3, 1

7.3.1 排序算法 — 快速排序

快速排序算法的实现如下：

函数 quickSort 定义

```
template<typename T, size_t N>
template<typename F >
void Array<T, N>::quickSort(int left, int right, F f) {
    ^^Iif (left < right){
        ^^I^^Iint i = left, j = right;
        ^^I^^IT x = m_ele[left];
        ^^I^^Iwhile (i < j){
            ^^I^^I^^Iwhile (i < j && f(x,m_ele[j])) j--; // 从右向左找第一个小于x的数
            ^^I^^I^^Iif (i < j)^^Im_ele[i++] = m_ele[j];
            ^^I^^I^^Iwhile (i < j && f(m_ele[i],x))^^Ii++; // 从左向右找第一个大于等于x的数
            ^^I^^I^^Iif (i < j)^^Im_ele[j--] = m_ele[i];
            ^^I^^I}
            ^^I^^Im_ele[i] = x;
            ^^I^^IquickSort(left, i - 1, f);           //左半部分排序
            ^^I^^IquickSort(i + 1, right, f);         //右半部分排序
            ^^I}
    }
```

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之

-5	1	5	12	16
----	---	---	----	----

 在有序数列中查找16

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为5， $5 < 16$

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



在右半序列继续查找

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为12, $12 < 16$

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



在右半序列继续查找

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



中点位置为5， $5 > 0$

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



在左半序列继续查找

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



中点位置为-5， $-5 < 0$

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



在右半序列继续查找

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



中点位置为1， $1 > 0$

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



在左半序列继续查找

7.3.2 二分查找算法

又称折半查找，在有序序列中使用，其基本思想为分而治之



在有序数列中查找16



中点位置为16。返回此位置



在有序数列中查找0



左半序列为空，查找失败

7.3.2 二分查找算法

二分查找算法的实现如下：

Array 成员函数 `binarySearch` 定义

```
template<typename T, size_t N>
int Array<T, N>::binarySearch(const T &value, int left, int
right) {
    while (left <= right) {
        int middle = (left + right) / 2; //计算中点位置
        if (m_ele[middle] == value)
            return middle;
        else if (m_ele[middle] > value)
            right = middle - 1;           //修改right
        else
            left = middle + 1;           //修改left
    }
    return -1;                           //查找失败
}
```

说明

- value 小于中点位置元素，将 right 设为 middle-1
- value 大于中点位置元素，将 left 设为 middle+1
- 查找失败则返回-1

7.3.2 二分查找算法

二分查找算法的实现如下：

Array 成员函数 `binarySearch` 定义

```
template<typename T, size_t N>
int Array<T, N>::binarySearch(const T &value, int left, int
right) {
    while (left <= right) {
        int middle = (left + right) / 2; //计算中点位置
        if (m_ele[middle] == value)
            return middle;
        else if (m_ele[middle] > value)
            right = middle - 1;           //修改right
        else
            left = middle + 1;           //修改left
    }
    return -1;                           //查找失败
}
```

说明

- value 小于中点位置元素，将 right 设为 middle-1
- value 大于中点位置元素，将 left 设为 middle+1
- 查找失败则返回-1

问题

查找 4 返回时，left 和 right 的值是多少？

7.3.2 二分查找算法

二分查找算法的实现如下：

Array 成员函数 `binarySearch` 定义

```
template<typename T, size_t N>
int Array<T, N>::binarySearch(const T &value, int left, int
right) {
    while (left <= right) {
        int middle = (left + right) / 2; //计算中点位置
        if (m_ele[middle] == value)
            return middle;
        else if (m_ele[middle] > value)
            right = middle - 1;           //修改right
        else
            left = middle + 1;            //修改left
    }
    return -1;                           //查找失败
}
```

说明

- value 小于中点位置元素，将 right 设为 middle-1
- value 大于中点位置元素，将 left 设为 middle+1
- 查找失败则返回-1

问题

查找 4 返回时，left 和 right 的值是多少？

答案

left 为 2, right 为 1

7.3.2 改进插入排序算法效率

问题

如何改进插入排序算法的效率？

本章结束

作业

- ① 在线提交系统：第七章作业
- ② 检查日期：待定