

第七章 模板与泛型编程

2019 年 9 月 10 日

目录

1 函数模板

- 定义函数模板
- 实例化函数模板
- 模板参数类型
- 类成员模板
- 可变参函数模板

2 类模板

- 成员函数定义
- 实例化类模板
- 默认模板参数

3 排序与查找

- 排序算法
- 二分查找算法

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。

7.1 函数模板

泛型编程

泛型编程是指一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。

模板是泛型编程的基础，它将**数据类型参数化**，为数据结构和算法的抽象提供**通用的代码**解决方案

7.1 函数模板

泛型编程

泛型编程是指一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。

模板是泛型编程的基础，它将**数据类型参数化**，为数据结构和算法的抽象提供**通用的代码解决方案**

请观察下面两组代码：

getMax 函数定义一

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

getMax 函数定义二

```
const string& getMax(const string &a, const string &  
    b){  
    return a>b ? a : b;  
}
```

7.1 函数模板

泛型编程

泛型编程是指一种采用与数据类型无关的方式编写代码的方法，是**代码重用**的重要手段。

模板是泛型编程的基础，它将**数据类型参数化**，为数据结构和算法的抽象提供**通用的代码解决方案**

请观察下面两组代码：

getMax 函数定义一

```
const int& getMax(const int &a, const int &b){  
    return a>b ? a : b;  
}
```

问题

这两个定义有什么相同点？

getMax 函数定义二

```
const string& getMax(const string &a, const string &  
    b){  
    return a>b ? a : b;  
}
```

7.1.1 定义函数模板

定义函数模板来实现一类函数的通用代码解决方案：

getMax 函数模板定义

```
template <typename T>
const T& getMax(const T &a, const T &b){
    return a>b ? a : b;
}
```

7.1.1 定义函数模板

定义函数模板来实现一类函数的通用代码解决方案:

getMax 函数模板定义

```
template <typename T>
const T& getMax(const T &a, const T &b){
    return a>b ? a : b;
}
```

说明

- 模板的定义以关键字 **template** 开始
- 模板参数列表放在一对**尖括号**里面
- 每一个参数前面用 **typename** 或 **class** 声明
- 列表含有多个模板参数则参数之间用**逗号**分开

7.1.1 定义函数模板

定义函数模板来实现一类函数的通用代码解决方案:

getMax 函数模板定义

```
template <typename T>
const T& getMax(const T &a, const T &b){
    return a>b ? a : b;
}
```

说明

- 模板的定义以关键字 **template** 开始
- 模板参数列表放在一对**尖括号**里面
- 每一个参数前面用 **typename** 或 **class** 声明
- 列表含有多个模板参数则参数之间用**逗号**分开

注意

模板的声明和定义应放在同一个头文件里

7.1.2 实例化函数模板 — 模板参数推断

实例化模板函数时，使用者需要提供具体的数据类型或值：

7.1.2 实例化函数模板 — 模板参数推断

实例化模板函数时，使用者需要提供具体的数据类型或值：

实例化方法一

```
cout << getMax(1.0, 2.5) << endl; // T被推断为double
```

生成如下函数实例

```
const double& getMax(const double &a, const double &b){  
    return a>b ? a : b;  
}
```

说明

编译器在**编译**的过程中，
利用实参来推断模板参数
的类型

7.1.2 实例化函数模板 — 模板参数推断

实例化模板函数时，使用者需要提供具体的数据类型或值：

实例化方法一

```
cout << getMax(1.0, 2.5) << endl; // T被推断为double
```

生成如下函数实例

```
const double& getMax(const double &a, const double &b){  
    return a>b ? a : b;  
}
```

说明

编译器在**编译**的过程中，利用实参来推断模板参数的类型

实例化方法二

```
cout << getMax<double>(1.0, 2.5) << endl; // 显式指明  
T为 double  
cout << getMax<string>("Hi ", "C++") << endl; // 显  
式指明T为 string
```

说明

用户显式地指明模板参数的类型

7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为类类型时，需要为类对象添加模板使用到的相关操作：

7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为类类型时，需要为类对象添加模板使用到的相关操作：

示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为类类型时，需要为类对象添加模板使用到的相关操作：

示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

实例化代码

```
const Fraction& getMax(const Fraction &a, const  
    Fraction &b){  
    return a>b ? a : b;  
}
```

7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为类类型时，需要为类对象添加模板使用到的相关操作：

示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

问题

在编译上面代码时提示编译错误，原因可能是什么？

实例化代码

```
const Fraction& getMax(const Fraction &a, const  
    Fraction &b){  
    return a>b ? a : b;  
}
```


7.1.2 实例化函数模板 — 为类类型添加模板支持

当模板函数的实参为类类型时，需要为类对象添加模板使用到的相关操作：

示例代码

```
Fraction a(3,4),b(2,5);  
cout << getMax(a, b) << endl; // T为Fraction类型
```

实例化代码

```
const Fraction& getMax(const Fraction &a, const  
    Fraction &b){  
    return a>b ? a : b;  
}
```

问题

在编译上面代码时提示编译错误，原因可能是什么？

答案

在 getMax 模板内部用到了关系 > 运算，但 Fraction 类不支持关系 > 运算

7.1.2 实例化函数模板 — 为类类型添加模板支持

给 Fraction 类型添加关系 > 运算支持:

Fraction 类 关系 > 运算 声明及定义

```
class Fraction{
    // 将关系>运算声明为Fraction类的友元
    friend bool operator>(const Fraction &lhs, const
        Fraction &rhs);
    // 其它成员与之前一致
    ...
};

bool operator>(const Fraction &lhs, const Fraction &
    rhs){
    return lhs.m_numerator*rhs.m_denominator > lhs.
        m_denominator*rhs.m_numerator;
}
```

说明

根据运算符重载的原则将
关系运算符函数
operator> 作为
Fraction 类的辅助函数,
并将其声明为 Fraction
类的友元

7.1.3 模板参数类型

以下两组代码中，模板参数有什么区别？

foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

7.1.3 模板参数类型

以下两组代码中，模板参数有什么区别？

foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

类型参数

作为**类型说明符**，指定函数的返回值类型、形参类型以及函数体内对象的类型等

maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

7.1.3 模板参数类型

以下两组代码中，模板参数有什么区别？

foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

类型参数

作为**类型说明符**，指定函数的返回值类型、形参类型以及函数体内对象的类型等

maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

非类型参数

代表一个值，当编译器实例化该模板时必须要为其提供一个**常量表达式**

7.1.3 模板参数类型

以下两组代码中，模板参数有什么区别？

foo 函数定义

```
template <typename T, typename U>
T foo(const T &t, const U &u) {
    return T(t);
}
```

maxElem 函数定义

```
template<typename T, int size>
const T& maxElem(T (&arr)[size]) {
    T *p = &arr[0];
    for (auto i = 0; i < size; ++i)
        if (*p < arr[i])
            p = &arr[i];
    return *p;
}
```

类型参数

作为**类型说明符**，指定函数的返回值类型、形参类型以及函数体内对象的类型等

非类型参数

代表一个值，当编译器实例化该模板时必须要为其提供一个**常量表达式**

说明

maxElem 函数模板中的函数形参 arr 为一个指向含有 size 个 T 类型数据元素数组的引用

7.1.3 模板参数类型

调用 `maxElem` 函数：

`maxElem` 函数模板实例化

```
int arr[10] = {1,8,5,3};  
int x = maxElem(arr);  
// 或者显式调用 maxElem<int,10>(arr);
```

编译器将会生成如下版本的函数：

```
const int& maxElem(int (&arr)[10]);
```

7.1.3 模板参数类型

调用 `maxElem` 函数：

maxElem 函数模板实例化

```
int arr[10] = {1,8,5,3};  
int x = maxElem(arr);  
// 或者显式调用 maxElem<int,10>(arr);
```

编译器将会生成如下版本的函数：

```
const int& maxElem(int (&arr)[10]);
```

问题

还有什么传递数组参数的方式？

7.1.3 模板参数类型

调用 `maxElem` 函数：

maxElem 函数模板实例化

```
int arr[10] = {1,8,5,3};  
int x = maxElem(arr);  
// 或者显式调用 maxElem<int,10>(arr);
```

编译器将会生成如下版本的函数：

```
const int& maxElem(int (&arr)[10]);
```

问题

还有什么传递数组参数的方式？

答案

还可以通过指针传递数组首地址的方式

7.1.3 模板参数类型 — 模板重载与特化

如果前面定义的 `getMax` 函数模板在调用过程中的实参为指针类型：

getMax 函数调用—

```
int a = 1, b = 2;  
getMax(&a, &b);
```

说明

需要返回两个指针所指向的对象的比较结果

getMax 定义—

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}
```

7.1.3 模板参数类型 — 模板重载与特化

如果前面定义的 `getMax` 函数模板在调用过程中的实参为指针类型：

getMax 函数调用—

```
int a = 1, b = 2;  
getMax(&a, &b);
```

说明

需要返回两个指针所指向的对象的比较结果

getMax 定义—

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}
```

问题

`getMax` 函数模板的定义还能否满足要求？

7.1.3 模板参数类型 — 模板重载与特化

如果前面定义的 `getMax` 函数模板在调用过程中的实参为指针类型：

getMax 函数调用—

```
int a = 1, b = 2;  
getMax(&a, &b);
```

说明

需要返回两个指针所指向的对象的比较结果

getMax 定义—

```
template <typename T>  
const T& getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}
```

问题

`getMax` 函数模板的定义还能否满足要求？

答案

不能。编译器推演出的参数 `T` 为 `int*`，函数体里面的操作变成了两个指针对象的比较

7.1.3 模板参数类型 — 模板重载与特化

为此，需要**重载**一个 `getMax` 模板函数：

getMax 函数调用二

```
int a = 1, b = 2;  
getMax(&a, &b);
```

getMax 函数模板重载

```
template <typename T>  
T* const & getMax( T* const &a, T* const &b){  
    return *a>*b ? a : b;  
}
```

说明

模板实参 `T` 的类型为 `int`，`*a` 和 `*b` 指向的是 `int` 对象，函数体里面的操作是两个 `int` 对象的比较

7.1.3 模板参数类型 — 模板重载与特化

进一步，如果调用的实参是指向字符串常量的指针：

getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

说明

需要返回指向字符串值较大的字符指针

getMax 函数定义二

```
template <typename T>  
const T & getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}  
  
template <typename T>  
T* const & getMax( T* const &a, T* const &b){  
    return *a>*b ? a : b;  
}
```

7.1.3 模板参数类型 — 模板重载与特化

进一步，如果调用的实参是指向字符串常量的指针：

getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

说明

需要返回指向字符串值较大的字符指针

getMax 函数定义二

```
template <typename T>  
const T & getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}  
  
template <typename T>  
T* const & getMax(T* const &a, T* const &b){  
    return *a > *b ? a : b;  
}
```

问题

现有的两个 getMax 函数的定义还能否满足要求？

7.1.3 模板参数类型 — 模板重载与特化

进一步，如果调用的实参是指向字符串常量的指针：

getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

说明

需要返回指向字符串值较大的字符指针

getMax 函数定义二

```
template <typename T>  
const T & getMax(const T &a, const T &b){  
    return a > b ? a : b;  
}  
  
template <typename T>  
T* const & getMax(T* const &a, T* const &b){  
    return *a > *b ? a : b;  
}
```

问题

现有的两个 getMax 函数的定义还能否满足要求？

答案

不能。*a 和 *b 指向的是单个字符，函数体里面的操作变成了两个字符的比较

7.1.3 模板参数类型 — 模板重载与特化

为此, 需要**特例化**一个 getMax 模板函数:

getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

说明

模板参数列表为空, 表明将显式提供所有模板实参

getMax 函数模板特化

```
template <>  
const char* const & getMax(const char* const &a,  
    const char* const &b){  
    return strcmp(a,b) > 0 ? a : b;  
}
```

说明

T 被推断为 const char*, a 和 b 分别为一个指向 const char 对象的 const 指针的引用, 函数是对两个字符串值的比较

7.1.3 模板参数类型 — 模板重载与特化

为此，需要**特例化**一个 getMax 模板函数：

getMax 函数调用三

```
const char *a = "Hi", *b = "C++";  
cout << getMax(a, b) << endl;
```

getMax 函数模板特化

```
template <>  
const char* const & getMax(const char* const &a,  
    const char* const &b){  
    return strcmp(a,b) > 0 ? a : b;  
}
```

说明

模板参数列表为空，表明将显式提供所有模板实参

说明

T 被推断为 const char*，a 和 b 分别为一个指向 const char 对象的 const 指针的引用，函数是对两个字符串值的比较

注意

一个特例化的函数模板本质上是一个实例，而非函数名的一个重载版本

7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

Swap 函数模板定义

```
template<typename T>
void Swap(T &a, T &b) {
    T c(a); // 复制构造对象 c
    a = b;
    b = c;
}
```

说明

需要构造一个辅助的局部对象 *c*，才能完成 *a* 和 *b* 的交换

7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

Swap 函数模板定义

```
template<typename T>
void Swap(T &a, T &b) {
    T c(a); // 复制构造对象 c
    a = b;
    b = c;
}
```

说明

需要构造一个辅助的局部对象 *c*，才能完成 *a* 和 *b* 的交换

如果 *T* 是 `int`，可以利用模板特化做出优化：

7.1.3 模板参数类型 — 模板重载与特化

还可以通过模板特化改善算法：

Swap 函数模板定义

```
template<typename T>
void Swap(T &a, T &b) {
    T c(a); // 复制构造对象 c
    a = b;
    b = c;
}
```

说明

需要构造一个辅助的局部对象 *c*，才能完成 *a* 和 *b* 的交换

如果 *T* 是 *int*，可以利用模板特化做出优化：

Swap 函数模板特化

```
template<>
void Swap(int &a, int &b)
{
    a ^= b;
    b ^= a;
    a ^= b;
}
```

说明

利用异或操作完成两个整数的交换，没有创建辅助对象，没有产生构造和析构行为，提高了执行效率

7.1.4 类成员模板

类的成员函数也可以定义为函数模板：

类 X 定义

```
class X{  
    void * m_p = nullptr;  
public:  
    template <typename T>  
    void reset(T *t) { m_p = t; }  
};
```

说明

成员函数 `reset` 定义为一个函数模板，接受不同类型的指针实参

7.1.4 类成员模板

类的成员函数也可以定义为函数模板：

类 X 定义

```
class X{  
    void * m_p = nullptr;  
public:  
    template <typename T>  
    void reset(T *t) { m_p = t; }  
};
```

reset 函数调用

```
int i = 0;  
double d = 0;  
X x;  
x.reset(&i); // 或者 x.reset<int>(&i);  
x.reset(&d); // 或者 x.reset<double>(&d);
```

说明

成员函数 `reset` 定义为一个函数模板，接受不同类型的指针实参

说明

- 第一条 `reset` 函数调用中 `T` 被推断为 `int` 类型，`m_p` 存放整型对象 `i` 的地址
- 第二条 `reset` 函数调用中 `T` 被推断为 `double` 类型，`m_p` 存放 `double` 类对象 `d` 的地址

7.1.5 可变参函数模板

C++11 新标准允许我们使用**数目可变**的模板参数：

foo 函数定义

```
template<typename... Args >
void foo(Args... args) {
    // 打印参数包args中参数的个数
    cout << sizeof...(args) << endl;
}
```

说明

- 可变数目的参数称为**参数包**，用省略号 “...” 表示，可包含 0 到任意个模板参数
- foo 函数的形参 args 为模板参数包类型，接受可变数目的实参

7.1.5 可变参数函数模板

C++11 新标准允许我们使用**数目可变**的模板参数：

foo 函数定义

```
template<typename... Args >
void foo(Args... args) {
    // 打印参数包args中参数的个数
    cout << sizeof...(args) << endl;
}
```

foo 函数调用

```
foo(); // 输出: 0
foo(1,1.5); // 输出: 2
foo(1,1.5,"C++"); // 输出: 3
```

说明

- 可变数目的参数称为**参数包**，用省略号 “...” 表示，可包含 0 到任意个模板参数
- foo 函数的形参 args 为模板参数包类型，接受可变数目的实参

7.1.5 可变参函数模板 — 包展开

可以通过**递归**的方式展开函数模板参数包：

print 函数定义

```
template<typename T, typename... Args>
ostream& print(ostream &os, const T &t, const Args
    &... rest) {
    os << t << " "; // 打印第一个参数
    return print(os, rest...); // 递归调用
}

template<typename T>
ostream& print(ostream &os, const T &t) {
    return os << t; // 打印最后一个参数
}
```

说明

- 第一次处理参数包中的第一个参数，然后用剩余参数调用自身
- 当参数包里面只剩下一个参数时，非可变参模板与可变参模板都匹配，但是非可变参模板更特例化，编译器首选非可变参数版本

7.1.5 可变参函数模板 — 包展开

可以通过**递归**的方式展开函数模板参数包：

print 函数定义

```
template<typename T, typename... Args>
ostream& print(ostream &os, const T &t, const Args
    &... rest) {
    os << t << " "; // 打印第一个参数
    return print(os, rest...); // 递归调用
}

template<typename T>
ostream& print(ostream &os, const T &t) {
    return os << t; // 打印最后一个参数
}
```

print 函数调用

```
print(cout, 1, 2.5, "C++") << endl; // 输出1 2.5 C++
```

说明

- 第一次处理参数包中的第一个参数，然后用剩余参数调用自身
- 当参数包里面只剩下一个参数时，非可变参模板与可变参模板都匹配，但是非可变参模板更特例化，编译器首选非可变参数版本

7.1.5 可变参函数模板 — 包展开

可以通过**递归**的方式展开函数模板参数包：

print 函数定义

```
template<typename T, typename... Args>
ostream& print(ostream &os, const T &t, const Args
    &... rest) {
    os << t << " "; // 打印第一个参数
    return print(os, rest...); // 递归调用
}

template<typename T>
ostream& print(ostream &os, const T &t) {
    return os << t; // 打印最后一个参数
}
```

print 函数调用

```
print(cout, 1, 2.5, "C++") << endl; // 输出1 2.5 C++
```

说明

- 第一次处理参数包中的第一个参数，然后用剩余参数调用自身
- 当参数包里面只剩下一个参数时，非可变参模板与可变参模板都匹配，但是非可变参模板更特例化，编译器首选非可变参数版本

问题

print 函数的形参是左值引用，如果是右值引用呢？

4.1 引用—右值引用

右值引用定义:

必须绑定到右值的引用 (只能绑定到一个将要销毁的对象), 通过 `&&` 实现。

右值引用语法格式为:

```
int i = 0;  
int &&rr1 = i+1; //正确: rr1 为右值引用, 绑定到一个临时对象  
int &&rr2 = i;   //错误: rr2 为右值引用, 不能绑定到左值对象
```

4.1 引用—右值引用

右值引用功能

- 程序员可以**操纵右值对象**，尤其是临时对象
- 可以通过右值引用**获取**即将消亡的右值对象的**资源**

思考：

以下代码会出现什么情况？为什么？

```
int &&rr1 = 10 + 1; //rr1 为右值引用
int &&rr3 = rr1;
```

4.1 引用—右值引用

右值引用功能

- 程序员可以**操纵右值对象**，尤其是临时对象
- 可以通过右值引用**获取**即将消亡的右值对象的**资源**

思考：

以下代码会出现什么情况？为什么？

```
int &&rr1 = 10 + 1; //rr1 为右值引用  
int &&rr3 = rr1;
```

编译器报错：**rr1** 为左值，**rr3** 不能绑定到左值对象

4.1 引用—右值引用

将左值显式转换成右值

```
int &&rr3 = std::move(rr1); //将rr1 转换成右值
```

有时候有些左值对象具有“临时性”，可以像右值一样使用。如函数形参

通用引用

右值引用声明 `&&` 与类型推导结合，变成一种通用引用类型，可与右值或左值绑定：

```
int i = 0;  
auto &&rr1 = 10; //rr1 为右值引用  
auto &&rr2 = i;  //rr2 为左值引用
```

7.1.5 可变参函数模板 — 转发参数包

两个函数的形参都是右值引用，forwardValue 函数调用报错，为什么？

forwardValue 函数及 rvalue 函数定义

```
void rvalue(int &&val){}

template<typename T>
void forwardValue(T &&val) {
    rvalue(val);
}
```

forwardValue 函数调用

```
forwardValue(42); // 错误
```

7.1.5 可变参函数模板 — 转发参数包

两个函数的形参都是右值引用，forwardValue 函数调用报错，为什么？

forwardValue 函数及 rvalue 函数定义

```
void rvalue(int &&val){}

template<typename T>
void forwardValue(T &&val) {
    rvalue(val);
}
```

forwardValue 函数调用

```
forwardValue(42); // 错误
```

forwardValue 函数调用细节

```
T &&val = 42; // 等同于 auto &&val = 42
rvalue(val);
```

答案

- 右值引用声明 && 与类型推导结合可以与右值绑定，所以 val 为右值引用
- 右值引用 val 引用右值 42，但 val 本身是左值
- rvalue 函数只接受右值形参，但 val 是左值

7.1.5 可变参函数模板 — 转发参数包

两个函数的形参都是右值引用，forwardValue 函数调用报错，为什么？

forwardValue 函数及 rvalue 函数定义

```
void rvalue(int &&val){}

template<typename T>
void forwardValue(T &&val) {
    rvalue(val);
}
```

forwardValue 函数调用

```
forwardValue(42); // 错误
```

forwardValue 函数调用细节

```
T &&val = 42; // 等同于 auto &&val = 42
rvalue(val);
```

答案

- 右值引用声明 && 与类型推导结合可以与右值绑定，所以 val 为右值引用
- 右值引用 val 引用右值 42，但 val 本身是左值
- rvalue 函数只接受右值形参，但 val 是左值

问题

可以让传入 rvalue 函数的实参保持原属性吗？

7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

`std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

`std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

`forwardValue` 函数定义二

```
template<typename T>
void forwardValue(T &&val) {
    rvalue(std::forward<T>(val));
}
```

`forwardValue` 函数调用二

```
forwardValue(42); // 正确
int a = 42;
forwardValue(a); // 正确
```

7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

`std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

`forwardValue` 函数定义二

```
template<typename T>
void forwardValue(T &&val) {
    rvalue(std::forward<T>(val));
}
```

`forwardValue` 函数调用二

```
forwardValue(42); // 正确
int a = 42;
forwardValue(a); // 正确
```

说明

- 当传入 `forwardValue` 的实参为右值，`T` 被推断为非引用类型，`forward<T>` 返回右值引用
- 当传入 `forwardValue` 的实参为左值，`T` 被推断为左值引用类型，此时 `forward<T>` 将返回左值引用

7.1.5 可变参函数模板 — 转发参数包

在 C++11 新标准下可以利用 `std::forward` 函数实现：

`std::forward` 转发左值描述性声明

```
template<typename T>
T&& forward( typename std::remove_reference<T>::type& t );
```

`forwardValue` 函数定义二

```
template<typename T>
void forwardValue(T &&val) {
    rvalue(std::forward<T>(val));
}
```

`forwardValue` 函数调用二

```
forwardValue(42); // 正确
int a = 42;
forwardValue(a); // 正确
```

说明

- 当传入 `forwardValue` 的实参为右值，`T` 被推断为非引用类型，`forward<T>` 返回右值引用
- 当传入 `forwardValue` 的实参为左值，`T` 被推断为左值引用类型，此时 `forward<T>` 将返回左值引用

注意

在 C++11 新标准下，
`&& &` 和 `& &&` 折叠为 `&`

7.1.5 可变参函数模板 — 转发参数包

可以组合使用可变参模板与 `forward` 函数，实现**参数包完美转发**：

7.1.5 可变参函数模板 — 转发参数包

可以组合使用可变参模板与 `forward` 函数，实现**参数包完美转发**：

fun 函数定义二和 foo 函数定义二

```
void foo(const string &s, int &&i) {  
    cout << s << i << endl;  
}  
  
template<typename... Args>  
void fun(Args&&... args) {  
    foo(std::forward<Args>(args)...);  
}
```

说明

`std::forward<Args>(args)...`

相当于：

`std::forward< T_i >(t_i)`

其中 T_i 为参数包中第 i 个参数 t_i 的类型

7.1.5 可变参函数模板 — 转发参数包

可以组合使用可变参模板与 `forward` 函数，实现**参数包完美转发**：

fun 函数定义二和 foo 函数定义二

```
void foo(const string &s, int &&i) {  
    cout << s << i << endl;  
}  
  
template<typename... Args>  
void fun(Args&&... args) {  
    foo(std::forward<Args>(args)...);  
}
```

说明

`std::forward<Args>(args)...`

相当于：

`std::forward< T_i >(t_i)`

其中 T_i 为参数包中第 i 个参数 t_i 的类型

当进行如下调用：

fun 函数调用

```
fun("abc", 42);
```

`foo` 函数的实参将扩展为：

```
std::forward<const char * >("abc"), std::forward<int>(42)
```

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 (T()) 初始化
- 将形参 l 中的元素依次复制到类模板数组成员 m_ele 中相应的位置
- 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 (T()) 初始化
- 将形参 l 中的元素依次复制到类模板数组成员 m_ele 中相应的位置
- 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
};
```

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

实例化 Array 类模板二

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

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

输出结果为：10 65

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>
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

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>
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>`

Less 类模板定义

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

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、模板参数、类模板参数、函数模板参数 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。下面将给 `Array` 模板类增加三种最常用的排序算法：选择排序、插入排序和冒泡排序

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。下面将给 `Array` 模板类增加三种最常用的排序算法：选择排序、插入排序和冒泡排序

Array 类模板定义四

```
template<typename T, size_t N>
class Array {
    //其它成员保持不变
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
----	---	---	---	---

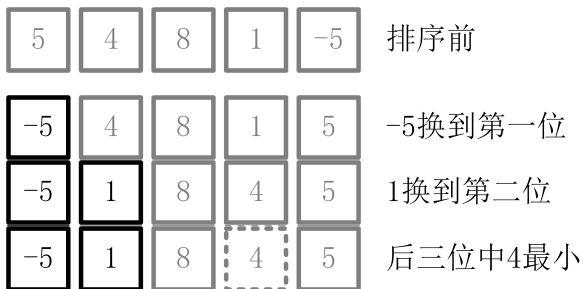
-5换到第一位

-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



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



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
---	---	---	----	---

前一位为有序数列

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

1到第一位，停止移动

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

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

前一位为有序数列

1到第一位，停止移动

4>1, 停止移动

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

前一位为有序数列

1到第一位，停止移动

4>1, 停止移动

$-5 < 1 < 4 < 5$ ，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
1	5	4	-5	8
1	4	5	-5	8
-5	1	4	5	8

前一位为有序数列

1到第一位，停止移动

4>1, 停止移动

-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

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

 排序前

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



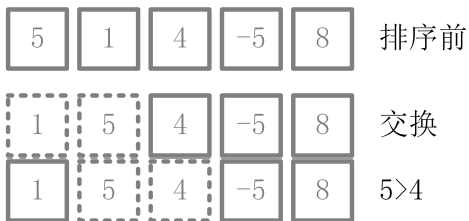
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



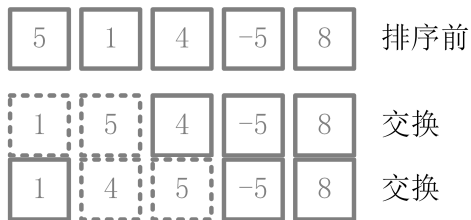
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



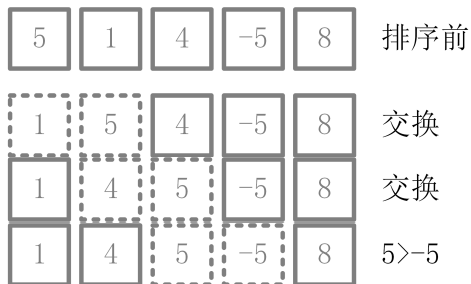
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



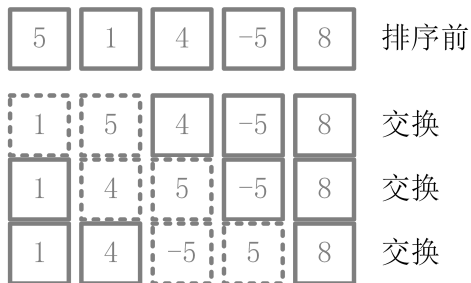
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



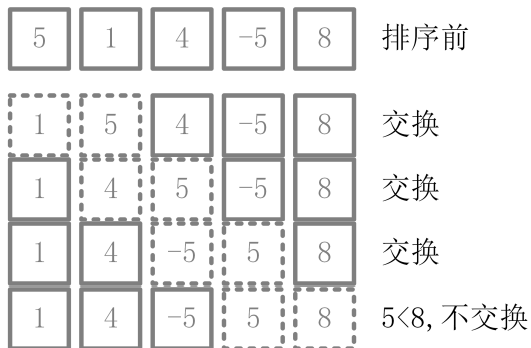
7.3.1 排序算法 — 冒泡排序

不断比较相邻的两个元素，如果发现逆序则交换



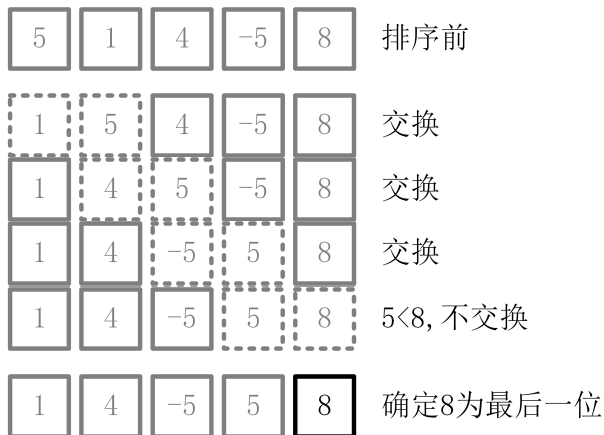
7.3.1 排序算法 — 冒泡排序

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
        }
    }
}
```

7.3.2 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

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

 在有序数列中查找16

-5	1	5	12	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

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

在有序数列中查找16

-5	1	5	12	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

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

在有序数列中查找16

-5	1	5	12	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



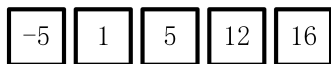
在有序数列中查找0



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

- 如果 value 小于中点位置 (middle) 元素，则将 right 设为 middle-1
- 如果 value 大于中点位置元素，则将 left 设为 middle+1
- 如果查找失败则返回-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

- 如果 value 小于中点位置 (middle) 元素，则将 right 设为 middle-1
- 如果 value 大于中点位置元素，则将 left 设为 middle+1
- 如果查找失败则返回-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

- 如果 value 小于中点位置 (middle) 元素，则将 right 设为 middle-1
- 如果 value 大于中点位置元素，则将 left 设为 middle+1
- 如果查找失败则返回-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

本章结束