

# 计算机科学基础II

## 第六章 模板与数据结构

曹鹏

Email: [caopeng@seu.edu.cn](mailto:caopeng@seu.edu.cn)

Tel: 13851945861

# 本章提纲

---

- ◆ 模板的基本概念 (6.1, 6.4节)
- ◆ 线性表的基本概念 (6.1.2节)
- ◆ 排序算法 (6.2.2节)

# 函数重载的困惑

交换函数swap

```
void swap(int &a, int &b)
{ int tmp=a; a=b; b=tmp;}
```

```
void swap(double &a, double &b)
{double tmp=a; a=b; b=tmp;}
```

可以用同样的“形式”定义

- ◆ 参数类型不同
- ◆ 函数动作序列相同

```
void swap(Type &a, Type &b)
{Type tmp=a; a=b; b=tmp;}
```

# 函数模板

- ◆ 函数模板：具有相同**特征**的函数族
  - ◆ 函数名相同
  - ◆ 函数参数个数相同
  - ◆ 函数参数和返回值类型形式一致
  - ◆ 函数动作序列相同
- ◆ 适用于以下类型的函数：交换；排序；查找；.....
- ◆ 优点：避免对不同数据类型实现同一类型函数导致的重复劳动，避免导致的维护和调试开销

# 函数模板的声明和定义

//声明，必须有形参名，且与定义一致

```
template <typename T> void swap(T &a, T &b);
```

//typename也可以换成class

```
template <class T> void swap(T &a, T &b);
```

模板标识符

//定义

模板参数表

函数模板名

```
template <typename T> void swap(T &a, T &b)
{ T tmp=a; a=b; b=tmp; }
```

返回类型

形参表

# 模板参数表

```
template<typename T> void swap(T &a, T &b)  
{T tmp=a; a=b; b=tmp;}
```

```
template<typename T1, typename T2>  
void func(T1 a, T2 b, T2 c){...}
```

- ◆ 模板参数表：包括多个模板类型参数，逗号隔开
- ◆ 模板类型参数：关键字(typename)加模板参数名组成
- ◆ 模板参数名：任意合法标识符，如T, T1, T2
- ◆ 以下位置的实际数据类型可以被模板参数名置换：
  - ◆ 形参类型
  - ◆ 函数局部变量类型
  - ◆ 返回值类型

# 求最大值函数模板应用实例 【例6.1】

## ◆ 函数模板定义

```
template <typename Groap>
```

```
Groap max(const Groap *r_array, int size)
```

```
//Groap: 类型参数
```

```
//const标识指标所指数据不可修改, 防止误改实参
```

```
{
```

```
    Groap max_val=r_array[0];
```

```
    int i;
```

```
    for (i=1;i<size; ++i)
```

```
        if(r_array[i]>max_val) max_val=r_array[i];
```

```
    return max_val;
```

```
}
```



# 求最大值函数模板应用实例 【例6.1】

## ◆ 主函数定义

```
int ia[5]={10,7,14,3,25};  
double da[6]={10.2,7.1,14.5,3.2,25.6,16.8};  
string sa[5]={"上海","北京","沈阳","广州","武汉"};  
int main() {  
    int i=max(ia,5);  
    cout <<"整数最大值为: " <<i<<endl;  
    double d=max(da,6);  
    cout <<"实数最大值为: " <<d<<endl;  
    string s=max(sa,5);  
    cout <<"字典排序最大为: " <<s<<endl;  
    return 0;  
}
```

整数最大值为: 25  
实数最大值为: 25.6  
字典排序最大为: 武汉



# 模板函数

函数模板

实例化

模板函数

## ◆ 模板实例化

- ◆ 依据函数模板和函数模板实参完成函数定义的过程（即生成模板函数的过程）

## ◆ 实例化方式

- ◆ 隐式指定（模板实参推演）
- ◆ 显式指定

# 隐式指定

- ◆ 隐式指定（模板实参推演）
  - ◆ 编译器根据函数实参的数据类型，推断模板实参
  - ◆ 要求实参类型与模板参数类型精确匹配
  - ◆ 如果既匹配普通函数，又匹配模板函数，则优先匹配普通函数

# 隐式指定

```
void swap(char &a, char &b);  
template <typename T> void swap(T&a, T&b);
```

```
int    i1, i2, i3 ;  
double d1, d2;  
char   c1, c2;
```

```
swap(c1, c2); // 调用普通函数swap(char &, char&)
```

```
swap(i1, i2); //由实参i1, i2数据类型生成swap<int>模板函数定义
```

```
swap(d1, d2); //由实参d1, d2数据类型生成swap<double>模板函数定义
```

```
swap(i2, i3); //调用已定义的swap<int>模板函数
```

```
swap(i1, d1); //编译错误, 因为实参i1和d1数据类型不同
```

```
swap(i1, 3); //编译错误, 因为无法通过常数3初始化引用型形参
```

# 显式指定

- ◆ 显式指定
  - ◆ 直接指定模板实参数数据类型
  - ◆ 确定数据类型后，可以对模板函数实参进行隐式的强制数据类型转换

# 显式指定

```
template <typename T> void swap(T&a, T&b);
```

```
int    i1, i2, i3;  
double d1, d2;
```

```
swap<int>(i1, i2);           //定义swap<int>模板函数
```

```
swap<double>(d1, d2);       //定义swap<double>模板函数
```

```
swap<int>(i3, i4);          //调用已定义swap<int>模板函数
```

```
swap<double>(i1, d1);       //编译错误, 无法将i1强制类型转换为  
                             double类型引用
```

```
swap<int>(i1, 3);           //编译错误, 因为无法通过常数3初始化引  
                             用型形参
```

# 矩阵转置/相乘函数模板应用实例 【例6.2】

## ◆ 函数模板声明 (2个模板类型参数)

```
template <typename T1,typename T2>  
void inverse(T1 *mat1, T2 *mat2, int a, int b);
```

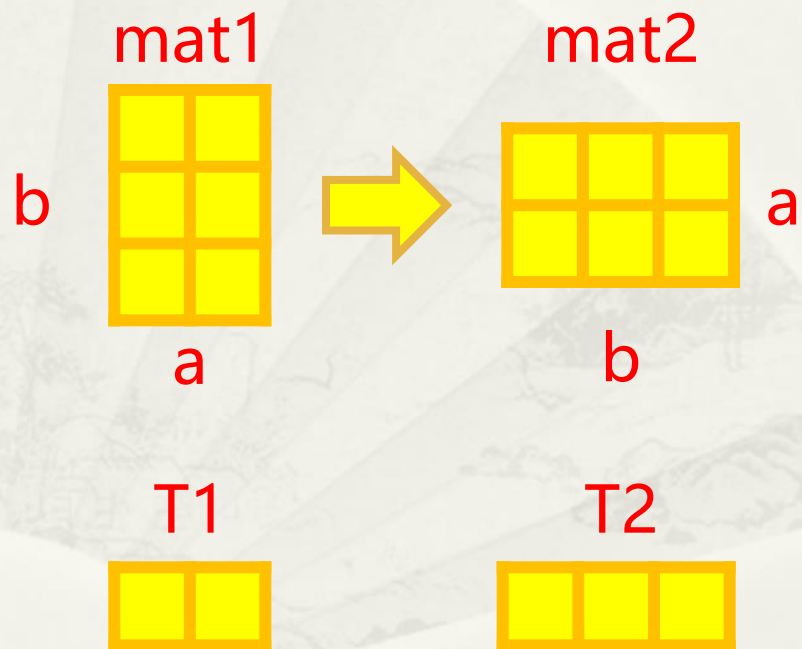
```
template <typename T1, typename T2>  
void multi(T1 *mat1, T2 *mat2,T2 *result, int a, int b, int c);
```

```
template <typename T>  
void output(T *mat, char*s, int a, int b);
```

# 矩阵转置/相乘函数模板应用实例 【例6.2】

## ◆ 矩阵转置函数模板定义

```
template <typename T1, typename T2>
void inverse(T1 *mat1, T2 *mat2, int a, int b){
    int i, j;
    for (i=0; i<b; i++)
        for (j=0; j<a; j++)
            mat2[j][i] = mat1[i][j];
    return;
}
```

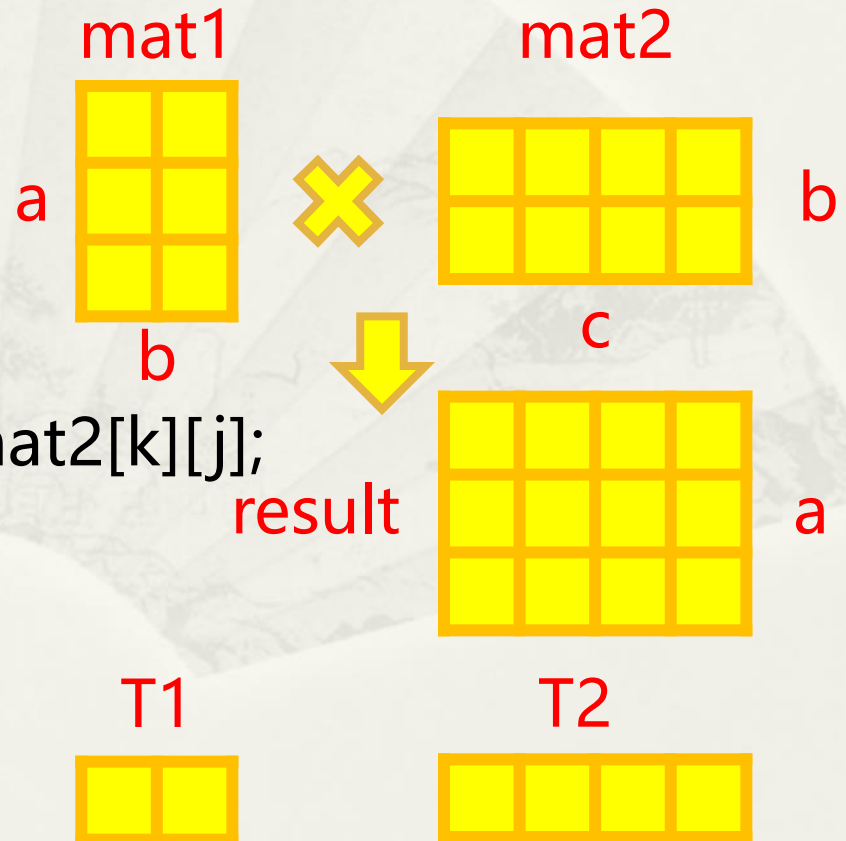




# 矩阵转置/相乘函数模板应用实例 【例6.2】

## ◆ 矩阵相乘函数定义

```
template <typename T1, typename T2>
void multi(T1 *mat1, T2 *mat2, T2 *result, int a, int b, int c)
{
    int i, j, k;
    for (i=0; i<a; i++) {
        for (j=0; j<c; j++) {
            result[i][j] = 0;
            for (k=0; k<b; k++)
                result[i][j] += mat1[i][k]*mat2[k][j];
        }
    }
    return;
}
```



# 矩阵转置/相乘函数模板应用实例 【例6.2】

## ◆ 矩阵输出函数模板定义

```
template <typename T>
void output(T *mat, char *s, int a, int b){
    int i,j;
    cout<<s<<endl;
    for (i=0;i<a;i++){
        for (j=0;j<b;j++){
            cout<<setw(6)<<mat[i][j];
            cout<<endl;
        }
    }
    return;
}
```

# 矩阵转置/相乘函数模板应用实例 【例6.2】

## ◆ 主函数定义

```
int main(){
    int middle[6][3], result[6][4];
    int matrix1[3][6]={8,10,12,23,1,3,5,7,9,2,4,6,34,45,56,2,4,6};
    int matrix2[3][4]={3,2,1,0,-1,-2,9,8,7,6,5,4};
    char *s1="result ", *s2="middle";
    inverse(matrix1,middle,6,3);
    //显式: inverse<int[6],int[3]> (matrix1,middle,6,3);
    multi(middle,matrix2,result,6,3,4);
    //显式: multi <int[3],int[4]> (middle,matrix2,result,6,3,4);
    output(matrix1, "matrix1", 3, 6);
    output(middle, s2, 6, 3);
    output(matrix2, "matrix2", 3, 4);
    output(result, s1, 6, 4);
    return 0;
```

# 函数模板和函数重载的异同

	函数模板	函数重载
形式	<code>template &lt;typename T&gt; void swap(T &amp;a, T &amp;b);</code>	<code>void func(int a); void func(int a, int b); void func(int a, char c);</code>
函数名	相同	相同
函数参数	个数相同、类型不同 但形式统一	个数不同，或 个数相同、类型不同
函数体	动作序列一致， 只有一种形式的代码	各自代码
重用	重用函数名 重用参数的对应形式 (将数据类型作为参数) 重用同一份代码	仅仅重用函数名

# 类模板

- ◆ 类模板：具有相同特征类族
  - ◆ 类名相同
  - ◆ 成员构成相同

# 类模板的定义

模板标识符

模板参数表

```
template <typename T, int size>
```

```
class CArray
```

类模板名

```
{  
    T data[size];  
    int MaxSize;
```

```
public:
```

```
    void setdata(int pos, T &d);  
    T getdata(int pos);
```

```
};
```

//类模板外定义的成员函数必须是函数模板

```
template <typename T, int size>
```

```
void CArray<T, size>::setdata(int pos, T &d){....}
```

```
template <typename T, int size>
```

```
T CArray<T, size>::getdata(int pos){....}
```

类体

模板参数名表

# 模板参数表

```
template <typename T , int size > class CArray  
{.....};
```

- ◆ 模板参数表：包括多个模板类型参数，逗号隔开
  - ◆ 类型参数（如T）：关键字(typename)加模板参数名组成，代表某个实际的数据类型
  - ◆ 非类型参数（如size）：数据类型，代表某个类型的常量



# 模板类

类模板

实例化

模板类

## ◆ 模板实例化

◆ 依据类模板和类模板实参完成类定义的过程（即生成类的过程）

## ◆ 实例化方式

- ◆ 不存在隐式指定
- ◆ 必须显式指定

# 显式指定

## ◆方式1：类模板→模板类→类对象

**typedef** 类模板名 <类模板实在参数表> 模板类名;

例如: **typedef** CArray<**char**, 10> string;  
string mystr;

## ◆方式2：类模板→类对象

类模板名 <类模板实在参数表> 类对象名;

例如: CArray<**char**, 10> mystr;

# 模板与类参数

应用方式	独立的函数模板, 模板实参指定为类	类模板的成员函数
形式	<pre>template &lt;typename T&gt; void Sort(T *arr, int n)</pre>	<pre>template &lt;typename T, int size&gt; class CArray {     T arr[size];     int maxsize; public:     Sort(); };</pre>
操作方式	<ul style="list-style-type: none"><li>◆实例化为模板函数</li><li>◆类模板是参数</li><li>◆通过类模板公有成员函数访问私有数据成员</li></ul>	<ul style="list-style-type: none"><li>◆实例化为模板类</li><li>◆成员函数是函数模板</li><li>◆直接访问私有数据成员</li></ul>
编程思想	面向过程	面向对象

# 模板

- ◆函数和类参数化的自动设计
- ◆通过模板可以实现类型参数化，即把数据类型定义为参数，从而实现了真正的代码可重用性
- ◆实现代码重用机制的一种工具，使得代码不受数据类型的限制

# 线性表的概念

## ◆一种常见的数据结构

## ◆一种含有 $n(\geq 0)$ 个同类结点的有限序列

- ◆均匀性：各个结点具有相同的数据类型

- ◆有序性：各个结点之间的相对位置是线性的

  - ◆唯一的“第一个”和“最后一个”结点（表头，表尾）

  - ◆直接前驱：每个结点有且只有一个，除了“第一个”

  - ◆直接后继：每个结点有且只有一个，除了“最后一个”

  - ◆相邻结点间具有直接前驱和直接后继的关系

# 线性表的分类

## ◆访问方式

- ◆直接随机访问：顺序表（数组）
- ◆间接顺序访问：链表（参见7.2节）
- ◆双端单向访问：队列（参见7.3节）
- ◆单端双向访问：栈（参见7.3节）

# 顺序表的概念

- ◆ 顺序表：直接随机访问的线性表
  - ◆ 元素在内存中顺序排列
  - ◆ 通过下标访问
  - ◆ 访问元素的时间开销与表的长度无关



# 顺序表的属性

- ◆数据表（数组）：“大开小用”
- ◆表头位置（通常是0）
- ◆表尾位置（可以任意位置）
- ◆表长 = 表尾位置 - 表头位置 + 1
- ◆最大表长=数组大小

# 顺序表的操作

---

- ◆ 表长计算与表空满判断
- ◆ 下标操作
- ◆ 查找元素位置
- ◆ 插入元素
- ◆ 删除元素

# 顺序表的实现

- ◆有属性、有操作→封装成类
- ◆对于不同数据类型的线性表，属性、操作都是相同的→类模板的方式
  - ◆类模板类型参数：数组数据类型
  - ◆类模板非类型参数：数组大小，顺序表最大长度

# 顺序表类模板【例6.3】

## 类模板实现顺序表（属性）

```
template < typename T, int size >
class SeqList
{
private:
    T m_list[size]; // 存放顺序表的数组
    int m_max_size; // 最大可容纳元素个数
    int m_last;     // 表尾元素位置
```

# 顺序表类模板【例6.3】

## 类模板实现顺序表（操作）

public:

```
SeqList( ) // 构造函数，初始化为空表  
{ m_last= -1; m_max_size=size;}
```

```
int Length( ) const // 计算表长度  
{ return m_last+1; }
```

```
bool IsEmpty( ) const // 判断表是否空  
{ return m_last==-1; }
```

```
bool IsFull( ) const // 判断表是否满  
{ return m_last==m_max_size-1; }
```

```
T Get(int i) //取第i个结点，注意判断i值合理性  
{ return i<0||i>last?NULL:m_list[i];}
```

# 顺序表类模板【例6.3】

## 类模板实现顺序表（操作）

public:

```
T& operator[] ( int i );           // 重载下标运算符[]  
int Find( T & x ) const;           // 寻找x在表中位置（下标）  
bool IsIn(T &x) const;             // 判断x是否在表中  
int Next( T & x ) const;           // 寻找x的后继位置（下标）  
int Prior( T & x ) const;          // 寻找x的前驱位置（下标）  
bool Insert( T & x, int i );        // x插入到列表中的i位置（下标）  
bool Remove( T & x );              // 删除x  
};
```

# 顺序表类模板【例6.3】

## 下标操作

```
template <typename T,int size>
T& SeqList<T,size>::operator[](int i)
{
    if( i<0 || i>=m_max_size || i>m_last )
    {
        cout<<"下标出界！"<<endl;
        exit(1); //系统函数，强行退出进程
                //一般0表示正常退出，其他表示异常退出
    }

    return m_list [i];
}
```



# 顺序表类模板【例6.3】

## 查找元素位置

```
template <typename T, int size>
int seqlist<T,size>::Find(T & x)const
{
    int i=0;
    while(i<=m_last && m_slist[i]!=x) i++; //从前到后，逐个
    比对
    if (i>m_last) return -1;                //到表尾仍未找到，返回-1
    else return i;                          //找到，返回下标
}
```

## 顺序表类模板【例6.3】

### 判断元素是否在表中

```
template <typename T, int size>
bool seqlist<T,size>::IsIn(T & x)const
{
    int i=0;
    bool found=0;

    while(i<=m_last && !found)
        if (m_list[i]!=x) i++;
        else found=1;
    return found;
}
```

//从前到后，逐个比对  
//没找到，找下一个  
//找到，标注返回下标

# 插入操作



# 顺序表类模板【例6.3】

## 插入操作

```
template <typename T,int size>
bool SeqList<T,size>::Insert(T & x, int i){
    int j;
    if ( i<0 || i>m_last+1 || m_last==m_max_size-1 )
        return false; //不能插入
    else{
        m_last++;
        for(j=m_last; j>i; j--)
            m_list[j]= m_list [j-1]; //从后向前, 依次后移
        m_list [i]=x;
        return true;
    }
}
```

# 删除操作



# 顺序表类模板【例6.3】

## 删除操作

```
template <typename T, int size>
bool seqlist<T,size>::Remove(T & x){
    int i, j;

    i=Find(x);           //先找到x的位置i
    if(i>=0){           //x在表中
        m_last--;
        for(j=i; j<=m_last; j++)
            m_slist[j]=m_slist[j+1]; //从前向后, 依次前移
        return true;
    }
    return false;       //x不在表中
}
```

## 顺序表类模板【例6.3】

### 查找前驱/后继元素位置

```
template <typename T, int size>
int seqlist<T,size>::Next(T & x){
    int i=Find(x);                //寻找x位置i
    if(i>=0 && i<m_last) return i+1; //x后继位置
    else return -1;               //x不在表中, 或在表尾
}

template <typename T, int size>
int seqlist<T,size>::Prior(T & x){
    int i=Find(x);                //寻找x位置i
    if(i>0 && i<=m_last) return i-1; //x前驱位置
    else return -1;               //x不在表中, 或在表头
}
```

# 顺序表类模板 【例6.3】

```
#include <iostream>
using namespace std;
int main(){
    seqlist <int,100> seqlisti;           //顺序表对象seqlisti的元素为整型
    int i,j,k,a[10]={2,3,5,7,11,13,17,19,23,29};
    for(j=0;j<10;j++){
        if (!seqlisti.Insert(a[j],j)){//把素数写入
            cout<<"数据太多表放不下了!"<<endl;
            break;
        }
    }
    j=seqlisti.Length();
    for(i=0;i<j;i++) cout<<seqlisti.Get(i)<<' '; //2 3 5 7 11 13 17 19 23 29
    cout << endl ;
    for(j=0;j<10;j++) seqlisti[j]=0;           //采用下标运算符运算
    for(j=0;j<10;j++) cout<<seqlisti[j]<<' '; //0 0 0 0 0 0 0 0 0 0
    cout<<endl;
    for(j=0;j<10;j++) seqlisti[j]=a[j];
    seqlisti[10]=31;                           //实验能否增加元素
    for(j=0;j<11;j++) cout<<seqlisti[j]<<' '; //2 3 5 7 11 13 17 19 23 29 31
    cout<<endl;
```



## 顺序表类模板 【例6.3】

```
k=7;
if (seqlisti.IsIn(k)) cout<<"素数7在顺序表中"<< endl; //素数7在顺序表中
//因形参为引用, 所以实参不可用整数常量7
else cout <<"素数7不在顺序表中"<<endl;
k=17;
if (seqlisti.Remove (k)) cout<<"删除素数17"<<endl; //删除素数17
else cout<<"找不到素数17, 无法删除";
j=seqlisti.Length( );
for (i=0;i<j;i++) cout<<seqlisti.Get(i)<<' '; //2 3 5 7 11 13 19 23 29 31
cout<<endl;
if (seqlisti.Insert(k,j-1)){ // 把素数17装回去,成功则打印
    j=seqlisti.Length ( );
    for (i=0;i<j;i++) cout<<seqlisti.Get(i)<<' ';
    cout<<endl; //2 3 5 7 11 13 19 23 29 17 31
}
```

## 顺序表类模板 【例6.3】

```
cout<<"打印17后一个素数: "<<seqlisti.Get(seqlisti.Next(k))<<endl; //31
cout<<"打印17前一个素数: "<<seqlisti.Get(seqlisti.Prior(k))<<endl; //29
cout<<"素数17在表中位置（下标）为: "<<seqlisti.Find(k)<<endl; //9
if(seqlisti.IsEmpty( )) cout<<"表是空的"<<endl;
else cout<<"表不空" <<endl; //表不空
if (seqlisti.IsFull()) cout<<"表是满的"<<endl;
else cout<<"表也不满" <<endl; //表也不满
if (seqlisti.IsIn(k)) cout<<"素数17在表中" <<endl; //素数17在表中
return 0;
}
```

# 线性表的操作算法

---

- ◆查找元素：从前向后，逐个比对
- ◆插入元素：从后向前，逐个后移
- ◆删除元素：从前向后，逐个前移

# 排序算法

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

# 排序算法

---

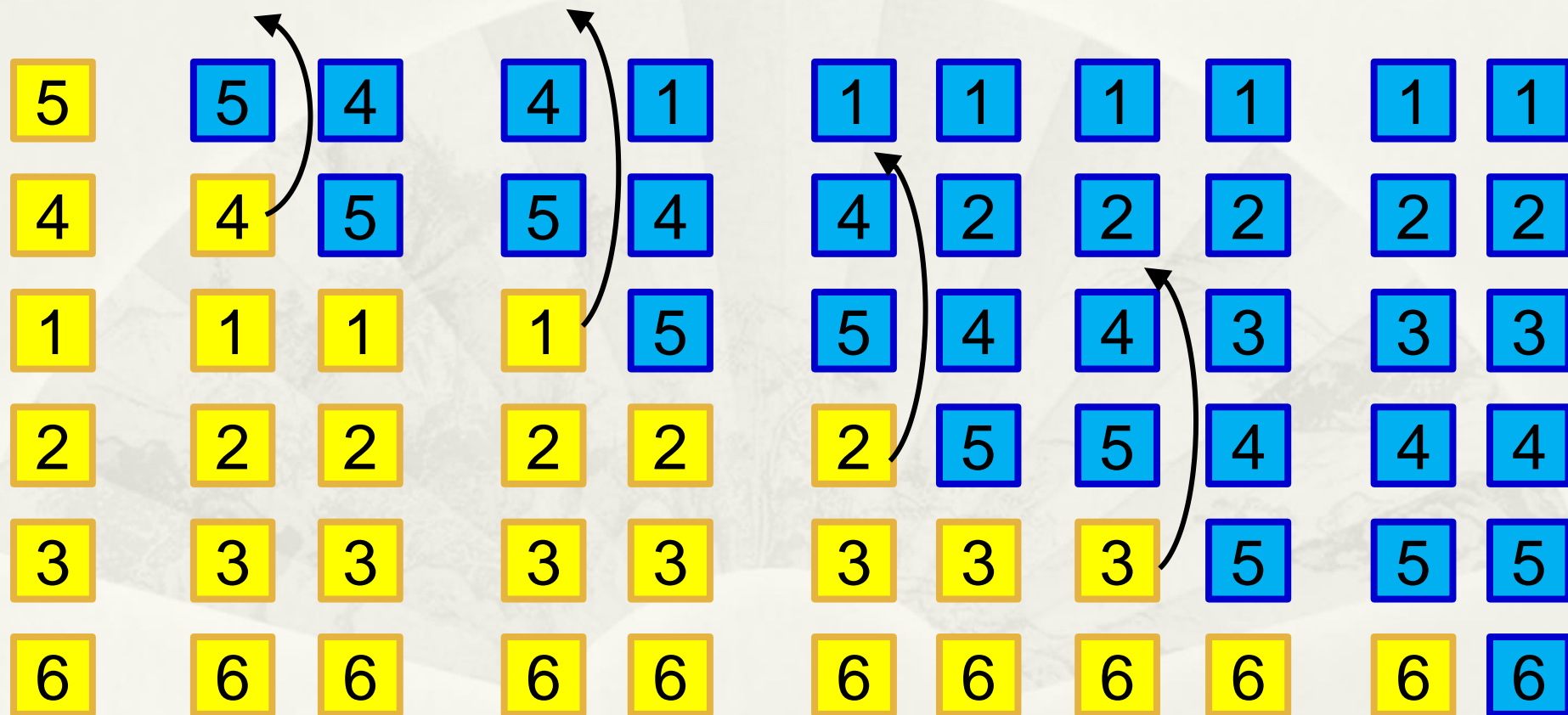
- ◆ 插入排序（直接插入排序）
- ◆ 交换排序（冒泡排序）
- ◆ 选择排序（直接选择排序）

# 直接插入排序

- ◆ 基本思想（以升序为例）
  - ◆  $N$ 个元素需要 $N-1$ 轮排序
  - ◆ 第 $i$  ( $1 \leq i \leq N-1$ )轮执行前，前 $i$ 个元素是已排序的（第1轮执行前，下标为0的元素默认已排序）
  - ◆ 第 $i$ 轮执行时，将下标 $i+1$ 元素插入前 $i$ 个已排序的元素
    - ◆ 从后( $i-1$ )向前(0)，一旦发现比下标 $i$ 元素小的，就插在后面
    - ◆ 如果前 $i-1$ 个元素里所有的都比下标 $i$ 元素大，就插在最前面
  - ◆ 第 $i$ 轮执行后，前 $i+1$ 个元素是已排序的，即下标0到 $i$ 的元素已排序

# 直接插入排序

初始序列 第1轮 第2轮 第3轮 第4轮 第5轮



未排序



待插入



已排序



# 直接插入排序

```
template <typename T, int size>
void Orderedlist<T,size>::InsertSort(){ //升序
    T temp; //暂存待插入数据
    int i, j;
    for (i=1;i<=last;i++){ //注意从i=1开始, 只做last轮
        temp=slist[i]; //第i轮插入slist[i], 先暂存到temp
        j=i; //j是插入位置, 初始值是i
        while (j>0&& temp<slist[j-1]){ //从后向前找到插入位置j
            slist[j]=slist[j-1]; //如果不是即将j-1数据后移
            j--; //依次往前
        }
        slist[j]=temp; //插在位置j
    }
}
```



# 交换排序

---

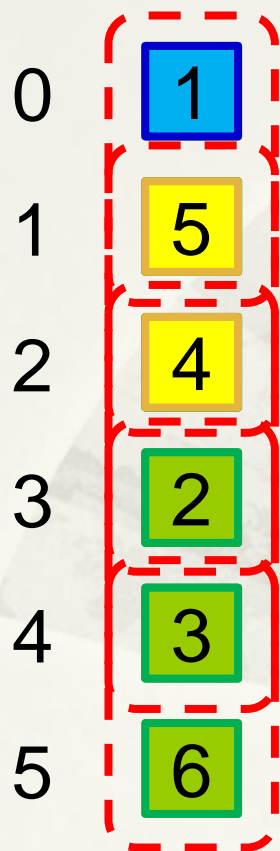
- ◆ 基本思想（以升序为例）
  - ◆ 元素两两比较，如逆序则交换
  - ◆ 发现所有元素按序即可停止（能充分利用原有序列的部分有序性， $N$ 个元素不一定需要 $N$ 轮排序）

# 冒泡排序

- ◆ 冒泡排序基本思想（以升序为例）
  - ◆ 一种典型的交换排序算法，以 $N$ 个元素为例
  - ◆ 第1轮：从尾( $N-1$ )至首( $0$ )，两两比较 $N-1$ 次，如逆序则交换，最终将最小的交换至表首( $0$ )
  - ◆ 第2轮：从尾( $N-1$ )至下标 $1$ ，两两比较 $N-2$ 次，如逆序则交换，最终将次小的交换至下标 $1$
  - ◆ 以此类推，如某轮未发生交换，则说明已全部排好序，排序结束

# 冒泡排序

第1轮：从尾( $N-1=5$ )至首( $0$ )，两两比较 $N-1=5$ 次



未排序



待交换



已排序

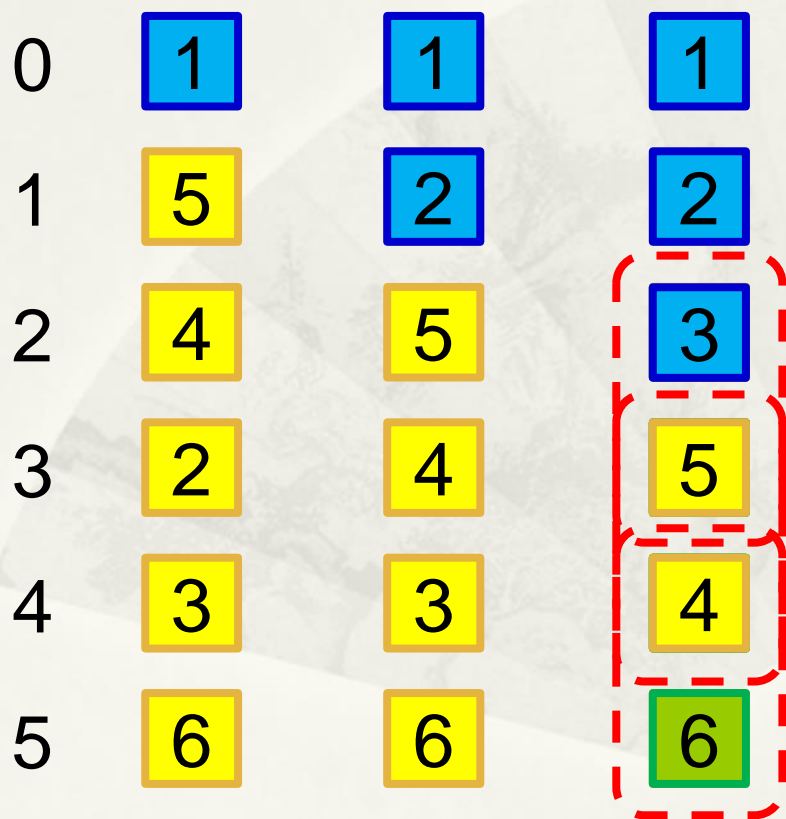
# 冒泡排序

第2轮：从尾( $N-1=5$ )至1，两两比较 $N-2=4$ 次



# 冒泡排序

第3轮：从尾( $N-1=5$ )至2，两两比较 $N-3=3$ 次



未排序



待交换



已排序

# 冒泡排序

第4轮：从尾( $N-1=5$ )至3，两两比较 $N-4=2$ 次

0	<div>1</div>	<div>1</div>	<div>1</div>	<div>1</div>
1	<div>5</div>	<div>2</div>	<div>2</div>	<div>2</div>
2	<div>4</div>	<div>5</div>	<div>3</div>	<div>3</div>
3	<div>2</div>	<div>4</div>	<div>5</div>	<div>4</div>
4	<div>3</div>	<div>3</div>	<div>4</div>	<div>5</div>
5	<div>6</div>	<div>6</div>	<div>6</div>	<div>6</div>



未排序



待交换



已排序

# 冒泡排序

第5轮：从尾( $N-1=5$ )至4，两两比较 $N-5=1$ 次



# 冒泡排序

```
template <typename T, int size>
void Orderedlist<T,size>::BubbleSort(){
    bool noswap; //为真时表示本轮未发生交换，可结束排序
    int i, j; T temp;
    for (i=0;i<last;i++){ //最多做last-1轮
        noswap=true; //默认为真，发生交换后置为假
        for (j=last;j>i;j--){ //从底至首，两两比较
            if (slist[j]<slist[j-1]){ //发现逆序对
                temp=slist[j];slist[j]=slist[j-1];slist[j-1]=temp; //交换
                noswap=false; //标记发生交换
            }
        }
        if (noswap) break; //本轮未发生交换，结束排序
    }
}
```



# 选择排序

- ◆ 基本思想（以升序为例）
- ◆  $N$ 个元素需要 $N-1$ 轮排序
- ◆ 步骤1：从未排序的元素中选出最小元素
- ◆ 步骤2：插在已排序的所有元素之后
- ◆ 重复步骤1和2，直至所有排序完成

原来后面那个  
元素放哪儿？



与这个最小元素交换位置



直接选择排序


# 直接选择排序

- ◆ 基本思想（以升序为例）
  - ◆  $N$ 个元素需要 $N-1$ 轮排序
  - ◆ 步骤1：从未排序的元素中选出最小元素
  - ◆ 步骤2：将其与已排序的所有元素后的那个交换位置
  - ◆ 重复步骤1和2，直至所有排序完成

# 直接选择排序

第1轮：从尾( $N-1=5$ )至0，选出最小元素

0	1
1	5
2	4
3	2
4	3
5	6

 未排序  最小元素  未排序首元素  已排序

# 直接选择排序

第2轮：从尾( $N-1=5$ )至1，选出最小元素

0	<div>1</div>	<div>1</div>
1	<div>5</div>	<div>2</div>
2	<div>4</div>	<div>4</div>
3	<div>2</div>	<div>5</div>
4	<div>3</div>	<div>3</div>
5	<div>6</div>	<div>6</div>

未排序

最小元素

未排序首元素

已排序

# 直接选择排序

第3轮：从尾( $N-1=5$ )至2，选出最小元素

0	1	1	1
1	5	2	2
2	4	4	3
3	2	5	5
4	3	3	4
5	6	6	6

 未排序  最小元素  未排序首元素  已排序

# 直接选择排序

第4轮：从尾( $N-1=5$ )至3，选出最小元素

0	1	1	1	1
1	5	2	2	2
2	4	4	3	3
3	2	5	5	4
4	3	3	4	5
5	6	6	6	6

 未排序  最小元素  未排序首元素  已排序

# 直接选择排序

第5轮：从尾( $N-1=5$ )至4，选出最小元素

0	1	1	1	1	1
1	5	2	2	2	2
2	4	4	3	3	3
3	2	5	5	4	4
4	3	3	4	5	5
5	6	6	6	6	6

 未排序  最小元素  未排序首元素  已排序

# 直接选择排序

```
template <typename T, int size>
void Orderedlist<T>::SelectSort(){
    int i,j,k; T temp;
    for(i=0;i<last;i++){ //做last-1轮
        k=i; //最小元素下标, 初始值i
        temp=slist[i]; //最小元素, 初始值slist[i]
        for (j=i;j<=last;j++) //找出真正最小元素
            if (slist[j]<temp){ //发现比temp小的, 替换之
                k=j;
                temp=slist[j];
            }
        if(k!=i) //最小元素(k)与未排序首元素(i)交换
            temp=slist[i]; slist[i]=slist[k]; slist[k]=temp;
    }
}
```



# 本章小结

- ◆ 函数模板 (6.1.1节)
  - ◆ 定义格式
  - ◆ 实例化方式 (函数模板→模板函数)：显式指定；隐式指定 (模板实参推导)
- ◆ 类模板 (6.1.2节)
  - ◆ 定义格式
  - ◆ 模板参数：类型参数；非类型参数
  - ◆ 实例化方式 (类模板→模板类)：显式指定
- ◆ 顺序表 (6.1.2 节)
  - ◆ 属于线性表，另外还有链表、队、栈
  - ◆ 线性表特征：均匀性；有序性
  - ◆ 顺序表特征：直接随机访问
    - ◆ 访问操作的时间：与大小无关
    - ◆ 修改 (增加/删除) 操作的时间：与大小有关
  - ◆ 采用类模板实现的原因
  - ◆ 类模板成员变量：数组；最大长度；当前长度 (表尾位置)
  - ◆ 类模板成员函数：
    - ◆ 常函数：计算长度；判断空满；查找 (前驱/后继)
    - ◆ 修改顺序表：插入元素；删除元素

# 本章小结

## ◆排序算法（6.2.2节）

### ◆插入排序

#### ◆直接插入排序

#### ◆对半插入排序

### ◆交换排序（冒泡排序）

### ◆选择排序（直接选择排序）

---



End