

CS 1502H 程序设计思想与方法 (C++)

Principles and Methodology in Programming

Chapter 5. 数组-1

陈东尧

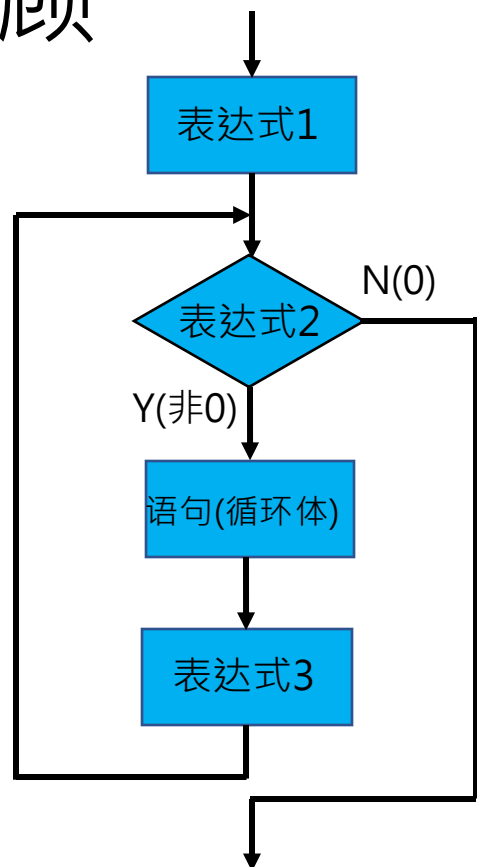
上海交通大学

2025年秋季

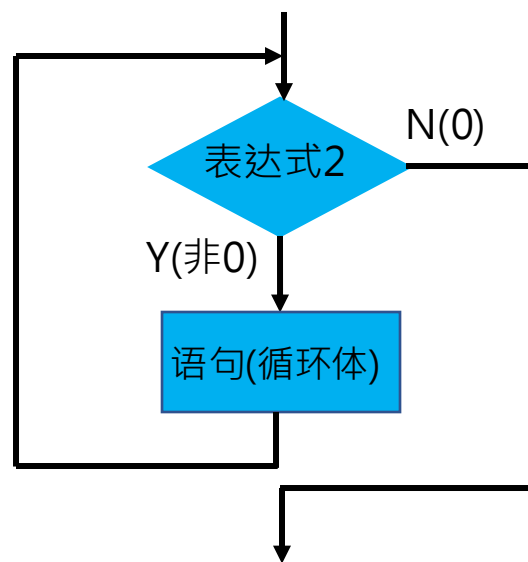




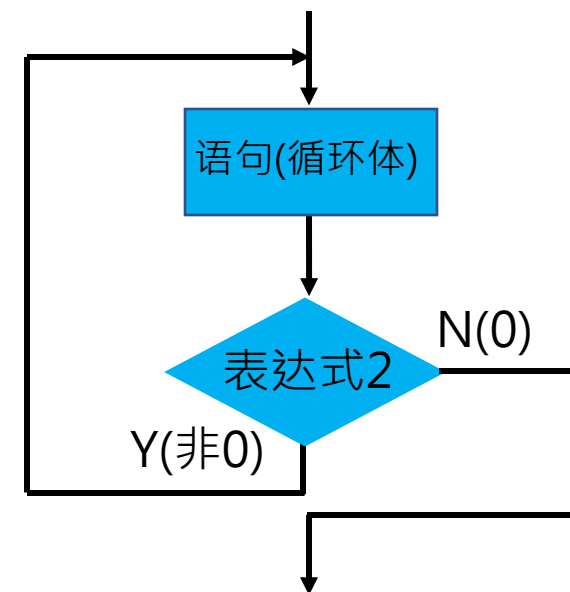
回顾



```
for(表达式1; 表达式2; 表达式3)
{
    语句;
}
```



```
while(表达式)
{
    语句;
}
```



```
do
{
    语句;
} while(表达式);
```



回顾





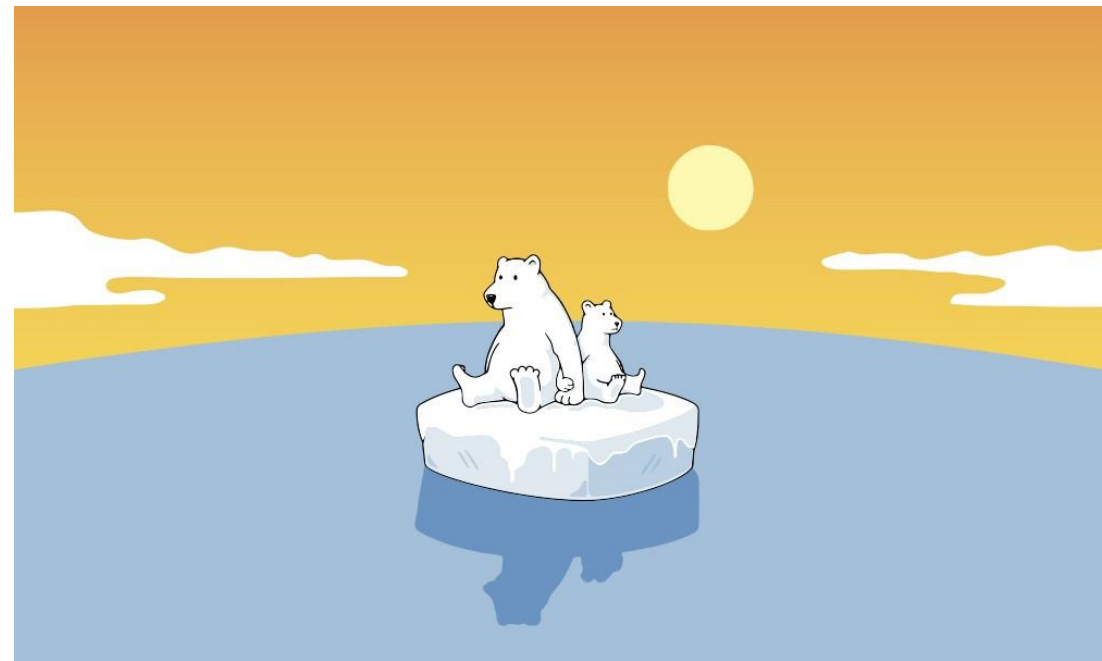
什么是数组

- 数组是一种**数据结构**
 - 数据结构用来保存数据



什么是数组

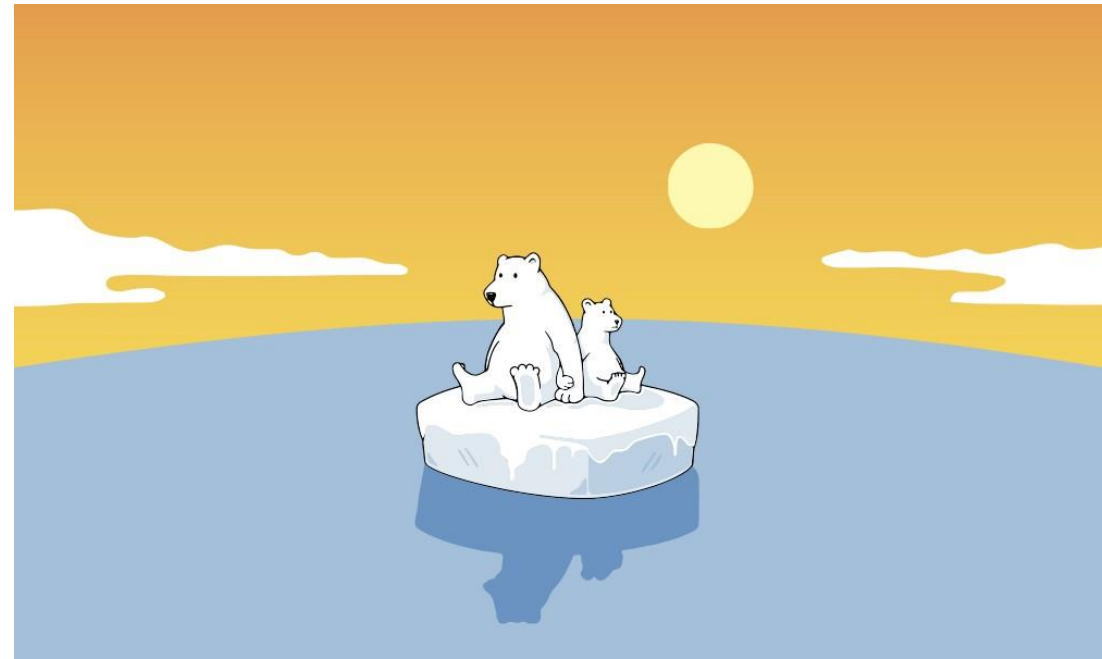
- 有时，我们需要存储一批**同类型的数据**
- 例如记录最近的气温





什么是数组

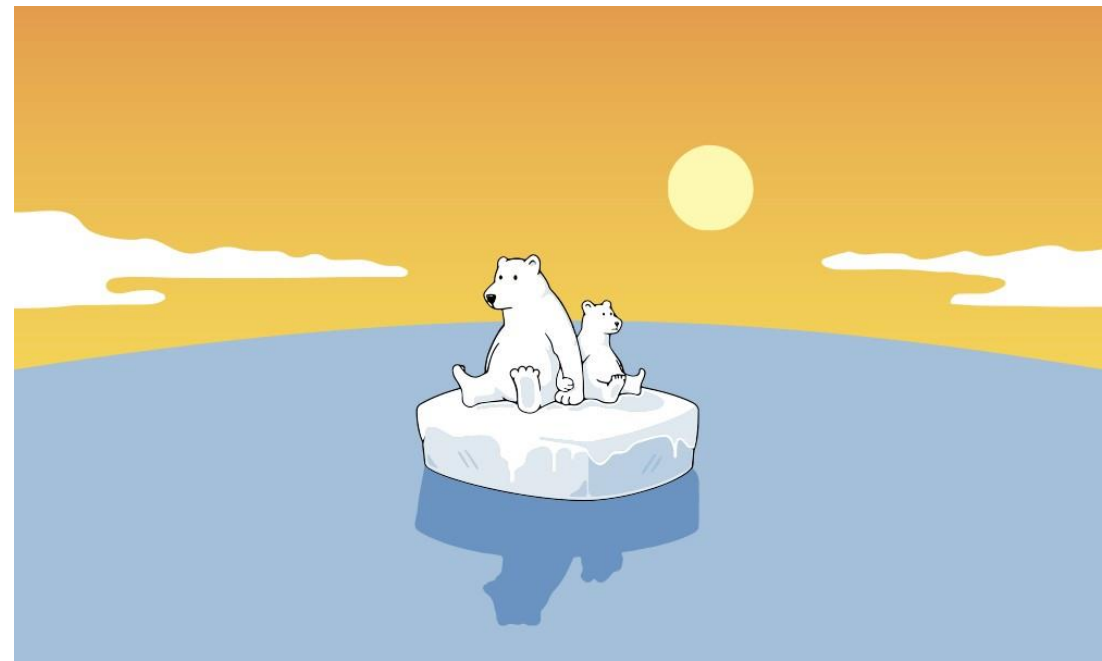
- 有时，我们需要存储一批**同类型的数据**
- 例如记录最近的气温
 - 38.2
 - 37.1
 - 39.2
 - 40.0
 - 41.9
 - 39.6
 - 40.1
 - ...





什么是数组

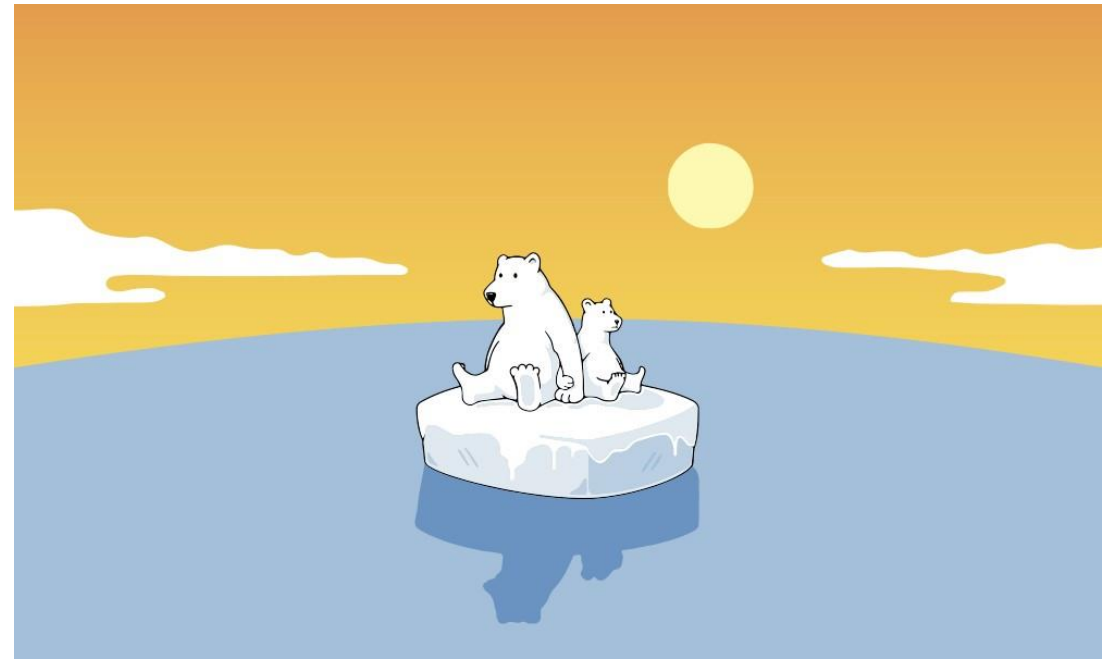
- 有时，我们需要存储一批**同类型的数据**
- 例如记录最近的气温
 - 顺序记录过于冗长，我们是否可以这样





什么是数组

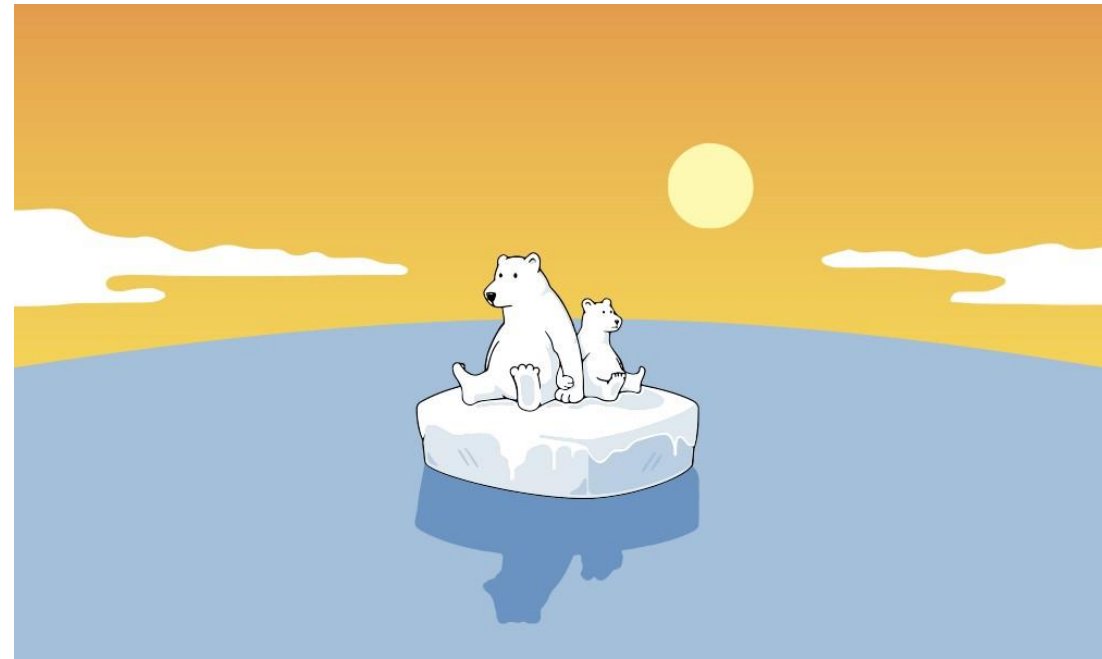
- 有时，我们需要存储一批**同类型的数据**
- 例如记录最近的气温
- 数组记录：`[38.2, 37.1, 39.2, 40.0, 41.9, 39.6, 40.1]`
- 还可以解锁很多有用的功能，比如：





什么是数组

- 有时，我们需要存储一批**同类型的数据**
- 例如记录最近的气温
- 数组记录：[38.2, 37.1, 39.2, 40.0, 41.9, 39.6, 40.1]
- 还可以解锁很多有用的功能，比如：
 - 最高温是多少？
 - → 41.9
 - 平均气温？
 - → 39.4
 - 40度以上（包含）的天数？
 - → 3
 - 与往年相比？





本章概览

- 一维数组
- 排序和查找
- 二维数组
- 字符串



数组定义

- 数组是保存一组同类元素的数据类型， 有两个特征
 - 数组元素是**有序**的
 - 数组元素是**同类**的
- 定义数组要定义三个基本内容 `Type    name [array_size] ;`
 - 数组元素的类型
 - 数组名字
 - 数组的大小



数组定义

- 格式：

其中，元素个数必须是常量。如：

```
int myarray[10];
```

但 `int n=10;`

`int myarray[n];` 是错误的

- 可以将元素个数定义为一个常量。如：

```
#define NumOfElement 10
```

```
int myarray[NumOfElement]; 相当于 int myarray[10];
```



数组定义：初始化

- 定义数组时可以对数组初始化

```
float x[5] = { -1.1, 0.2, 33.0, 4.4, 5.05 };
```

- 初始化表的长度短于要被初始化的数组元素数目，那么**剩余元素被初始化为0**。
- 带有初始化的数组可以不定义数组规模，编译器根据初值个数决定数组的大小

```
int a[]={1,2,3,4,5}; 则默认数组大小为5
```



数组元素

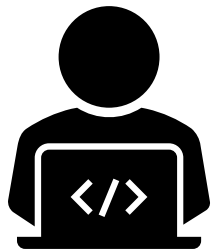
- 数组元素的使用是通过数组名及元素的序号来指定，如iArray[2]。
 - 当数组的大小为n时，元素的序号为0 到 n-1
 - 元素的序号称为下标。下标可为整数、整型变量或结果为整型的任意表达式

```
int iArray[10], idx;  
for (idx = 0; idx <= 9; ++idx)  
    cin >> iArray[idx] ;  
for ( idx = 0; idx <= 9; ++idx)  
    cout << iArray[idx];
```



数组的应用

- 统计：找出最重的羊是哪一只？



代码演示

Chapter5-trivia-sheep.cpp

```
#define NUM 5
int main(){
    double sheep[NUM], maxWeight=0;
    int i, maxNum;

    for (i=0; i<NUM; ++i) {
        cout << "请输入第"<< i <<"只羊的重量";
        cin >> sheep[i];
    }

    for (i=0; i<NUM; ++i) {
        if (sheep[i]>maxWeight) {
            maxWeight = sheep[i];
            maxNum = i;
        }
    }
    cout << "最重的羊是第" << maxNum << "只"<< endl;
    cout << "它的重量是" << maxWeight << endl;
    return 0;
}
```



数组的应用

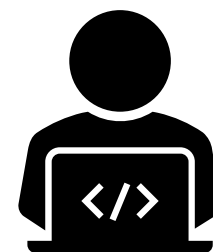
- 从终端输入一串字符，统计字符串中个字母出现的次数。
- 解决方法：
 - 方法一：用26个整型变量计数26个字母，对输入字符串中的每一字符用switch语句分别计数。
 - 方法二：用一个26个元素的数组，如num[26]，表示计数。num[0]存放a的个数，num[1]存放b的个数…。这样对每一个字符不必用switch，而只需用一个简单的计算：
++num[toupper(ch) - 'A'];

库函数toupper用来处理统一大小写

```
int main(){
int count[26] = {0};
char ch;

ch = toupper(cin.get()); // 统一大写
while (ch>='A' && ch <='Z') {
    ++count[ch-'A'];
    ch = toupper( cin.get());
}

for (int i=0; i< 26; ++i) cout << count[i] << '\t';
return 0;
}
```



代码演示

Chapter5-trivia-word.cpp



注：C++ toupper

- 转大写

```
#include <iostream>
```

```
#include <cctype>
```

```
using namespace std;
```

```
int main() {
```

```
    // convert 'a' to uppercase
```

```
    char ch = toupper('a');
```

```
    cout << ch;
```

```
    return 0;
```

```
}
```

```
// Output: A
```



数组在内存中

- 数组名中存放的是该数组的**起始地址**
 - 不能直接对数组名进行赋值。如：`myArray=30` 是错的。
- 定义数组就是定义了一块连续空间
 - 空间的大小等于元素数*每个元素所占的空间大小。
 - 数组元素按序存放在这块空间中。

内存地址用16进制（Hexadecimal）表示

地址	0xff0	0xff4	0xff8
数据	1	2	3



数组在内存中

- 如：`int myArray[5]={3,1,4,27,9}`;占用了20个字节，因为每个整型数占四个字节。如给`myArray[3]`赋值为27，如果这块空间的起始地址为100，那么在内存中的情况是：

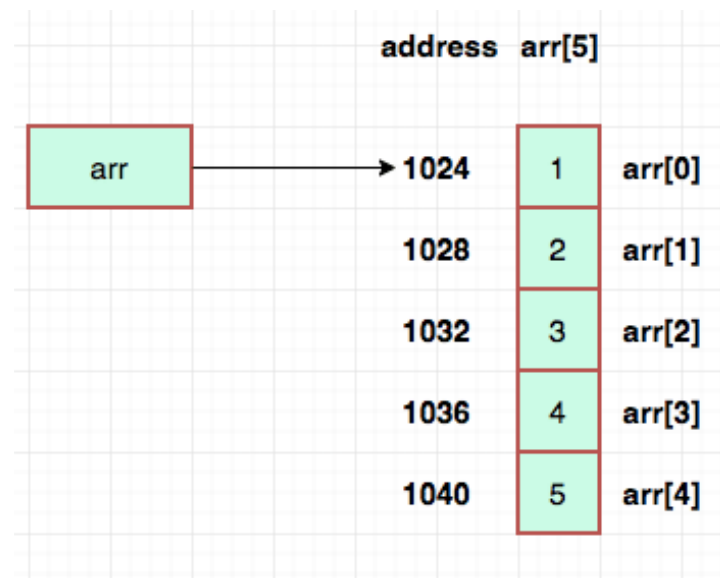
3	1	4	27	9
100 103	104 107	108 111	112 115	116 119

- 当你引用变量`myArray[idx]`时，系统计算它的地址 $100+idx*4$ ，对该地址的内容进行操作。



数组在内存中

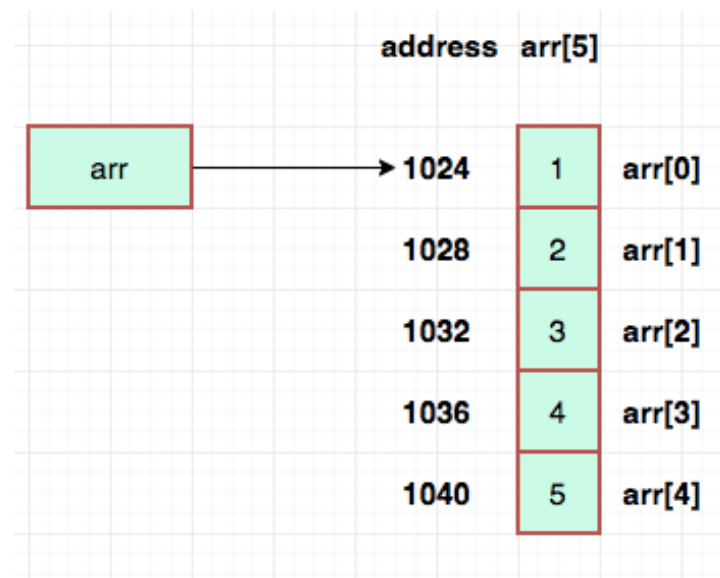
- 为何需要提前声明数组的大小？
- 程序如何访问数组元素？





思考：为何数组下标从0开始？

- 历史原因
 - C语言数组下标是从0开始->Java也是->JavaScript也是
 - 降低学习成本
- 减少CPU指令计算
 - 下标从0开始： $\text{arr}(i) = \text{base_address} + i * \text{type_size}$
 - 下标从1开始： $\text{arr}(i) = \text{base_address} + (i-1) * \text{type_size}$
 - 可见，从0开始减少了一次减法运算





数组下标越界问题

- C/C++语言不检查数组下标的超界。如定义数组 `int myarray[10]`; 合法的下标范围是0 – 9，但如果你引用`myarray[10]`，系统不会报错。
 - 如数组`myarray` 的起始地址是1000，当引用`myarray[10]`时，系统对1040号内存进行操作。而1040可能是另一个变量的地址
- 解决方法：由程序员自己控制。在对下标变量进行操作前，先检查下标的合法性。



本章概览

- 一维数组
- 排序和查找
- 二维数组
- 字符串

0.879862851 0.47525426 0.479606838 0.956114994 0.141993619 0.121802787 0.447390024
0.472592204 0.441201661 0.12883539 0.402319203 0.915838027 0.602200468 0.796172281
0.94003884 0.973399931 0.885465995 0.019759123 0.263015293 0.466611472 0.036628551
0.043703272 0.002468304 0.184442252 0.954127887 0.662974082 0.557175037 0.145870011
0.751640678 0.497293297 0.82599317 0.078648667 0.026854127 0.630121055 0.625142869
0.869486333 0.493730961 0.898507747 0.856918124 0.397633504 0.476803915 0.310901559
0.967041401 0.155187776 0.600015619 0.228216723 0.856263742 0.306318165 0.26301231
0.742222718 0.123391277 0.98979350 0.109456697 0.485834312 0.444266721 0.33994936
0.009110596 0.458533220 0.754188397 0.261222264 0.704687535 0.686216465 0.330631104
0.080688363 0.558414322 0.332876335 0.333176308 0.238411518 0.582042593 0.177425115
0.618095941 0.870330114 0.317109126 0.791749961 0.902223828 0.578939156 0.161880241
0.523922695 0.245170668 0.433632433 0.510581513 0.59932295 0.954803631 0.353413157
0.625259029 0.492813059 0.905918758 0.512446852 0.856013718 0.522147056 0.354622147
0.211844292 0.562581171 0.527106926 0.350487206 0.225925156 0.685085439 0.010780608
0.884696342 0.103655492 0.510903614 0.063478837 0.886784169 0.011573384 0.184155971
0.3045916 0.775125123 0.81467203 0.187015657 0.524926728 0.584617998 0.662566639
0.394940838 0.079670036 0.088427801 0.463354361 0.686123267 0.313500398 0.077839555
0.203582194 0.699670888 0.377824462 0.648848254 0.384478909 0.788299803 0.776278051

从下列数字中找到“0.414950917”。如果找到，
请输出它的位置。如果找不到，则输出“Not
found!”



顺序查找 (Linear Search)

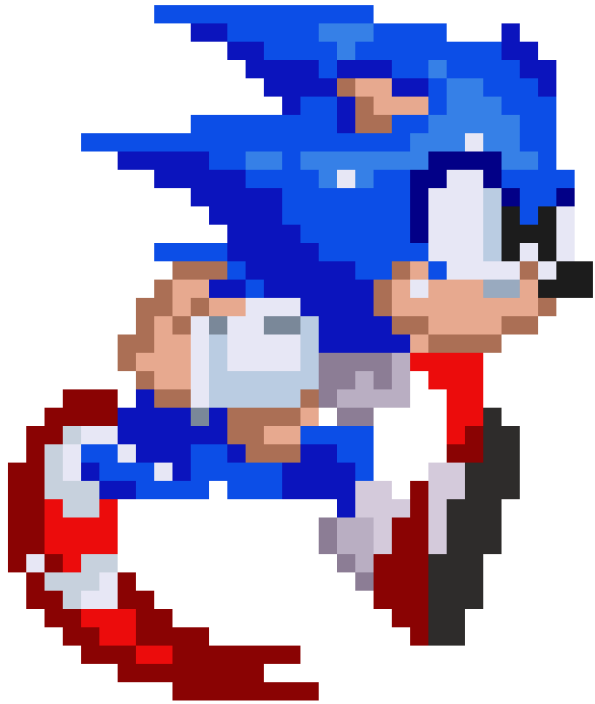
- 被查找的数存放在一个数组中
- 从数组的第一个元素开始, **依次往下比较, 直到找到要找的元素为止。**
- 如在一整数数组中查找元素x的存储位置。找到则输出k, 找不到则输出“找不到”

```
int main()
{ int k, x;
  int array[ ] = { 2, 3, 1, 7, 5, 8, 9,
0, 4, 6};
  cout << "请输入要查找的元素值: ";
  cin >> x;
  for (k = 0; k < 10; ++k){
    if (x == array[k])
      { cout << k; break;}
  }
  if (k == 10) cout << "not found";
  return 0;
}
```



顺序查找 (Linear Search)

- 这一方法是否可能更快？



Credit: Sega, Inc.



二分查找 (Binary Search)





二分查找 (Binary Search)

- 数组元素已按某一顺序排序，如数字的大小顺序、字母的字母序等。以下讨论中都假设是按升序排序。
- 例子：寻找47

0	4	7	10	14	23	45	47	53
---	---	---	----	----	----	----	----	----

0	4	7	10	14	23	45	47	53
---	---	---	----	----	----	----	----	----



二分查找详细过程 ✨ ✨ ✨ ✨ ✨

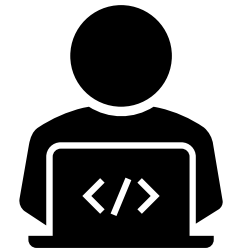
Step 1: 确定上下界 : lh, rh

Step 2: 找出中间元素的位置 : $mid = (lh + rh) / 2$

Step 3: 比较中间元素与欲查找的元素 key :

- 如 key 等于中间元素, 则 mid 就是要查找的元素的位置 ;
- 如 $key >$ 中间元素, 则从 lh - mid 的这些元素不可能是要查找的元素, 修正查找范围为 $lh = mid + 1$ 到 rh ;
- 如 $key <$ 中间元素, 则从 mid - rh 的这些元素不可能是要查找的元素, 修正查找范围为 lh 到 $rh = mid - 1$;
- 如 $lh > rh$, 则要查找的元素不存在。

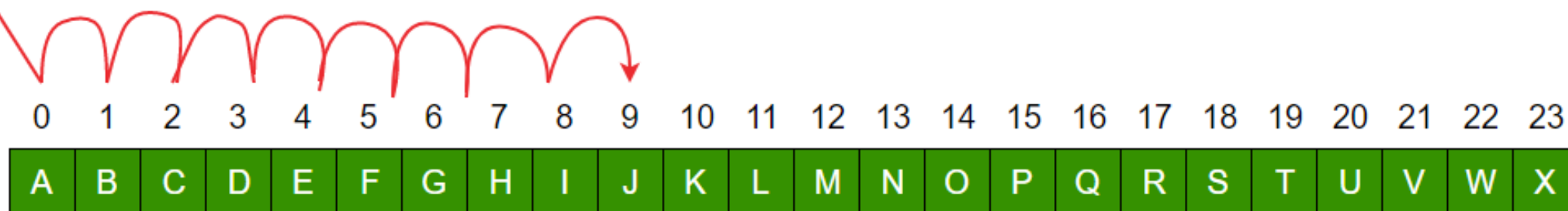
```
int main() {  
    int lh, rh, mid, x;  
    int array[ ]={0,1,2,3,4,5,6,7,8,9};  
    cout <<"请输入要查找的数据: ";  
    cin >> x;  
    lh = 0; rh = 9;  
    while ( lh <= rh ) {  
        mid = ( lh + rh ) / 2;  
        if ( x== array[mid] ) {  
            cout << x << "的位置是: " << mid << endl;  
            break;  
        }  
        if ( x < array[mid]) rh = mid - 1;  
        else lh = mid + 1;  
    }  
    if (lh > rh) cout << "没有找到" << endl;  
    return 0;  
}
```



代码演示

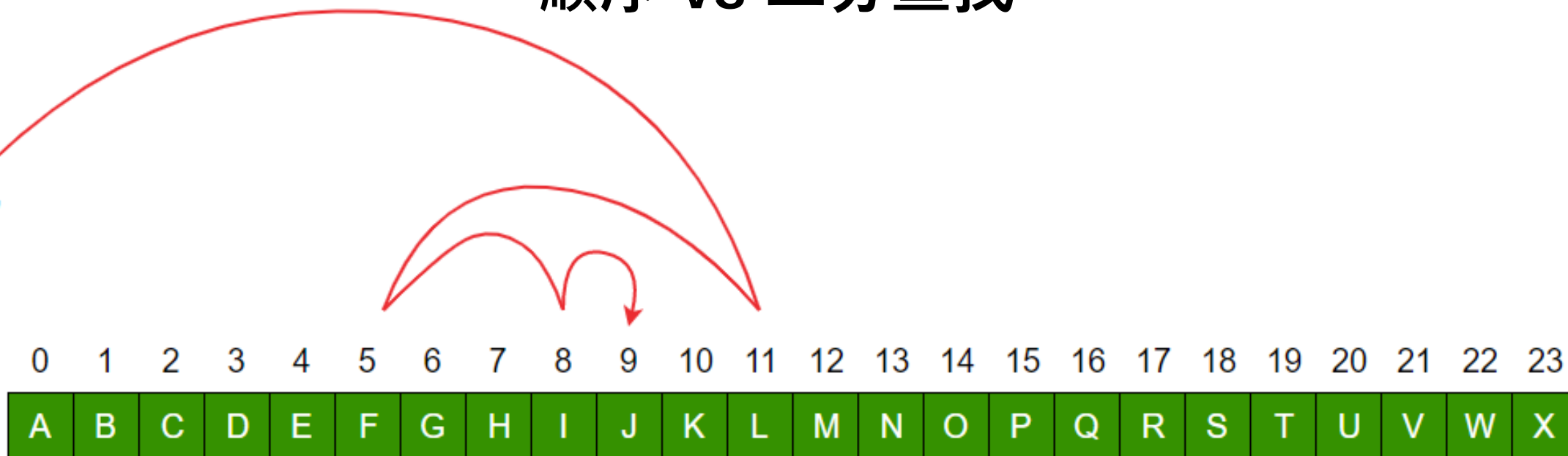
chapter5-binarySearch.cpp

Find 'J'



顺序 vs 二分查找

Find 'J'





算法效率分析

- 平均时间性能

- 顺序搜索：

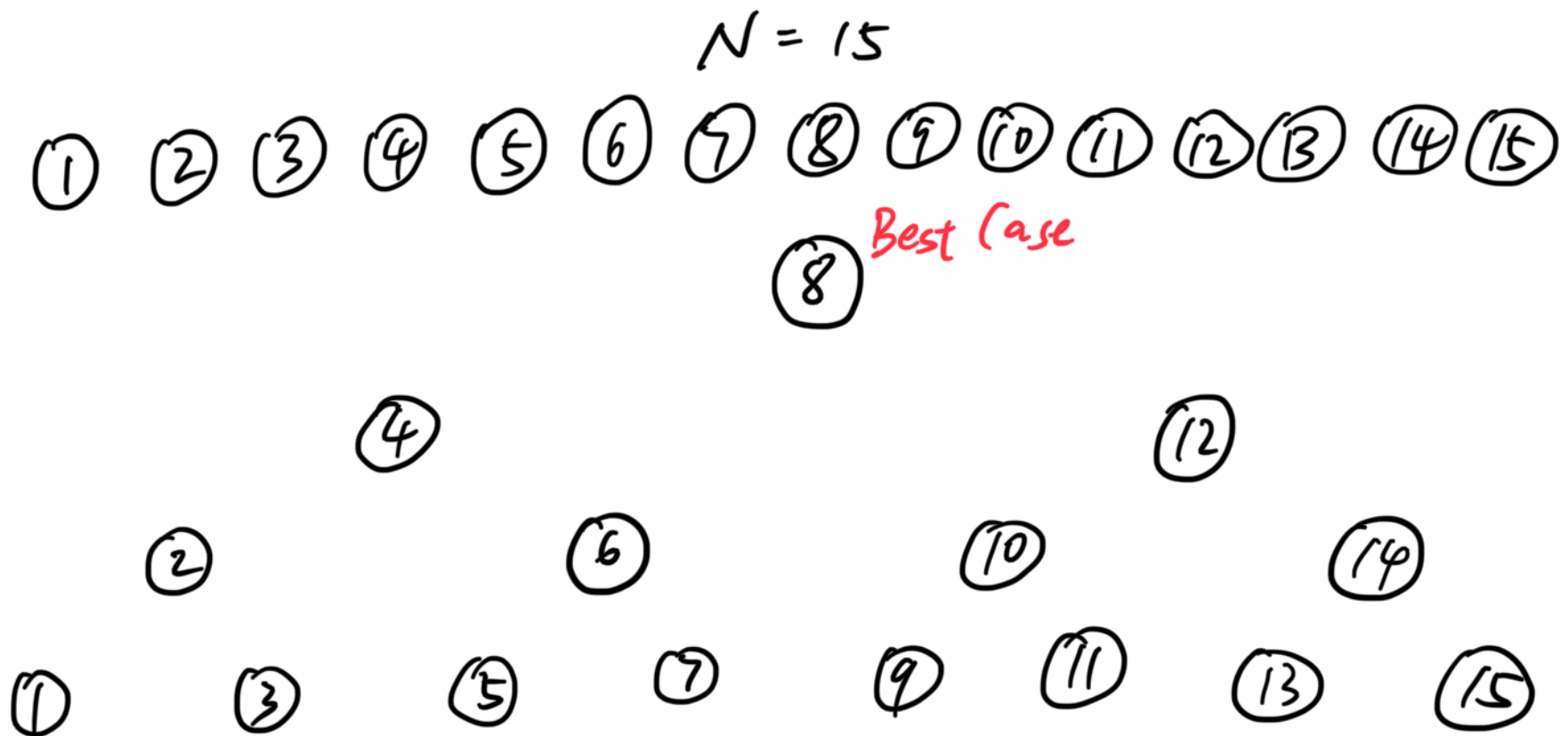
$$(1 + 2 + 3 + \dots + n) / n = (n + 1) / 2$$

最好情况

最差情况



二分查找时间性能分析





二分查找时间性能分析

$$N = 15$$

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15)



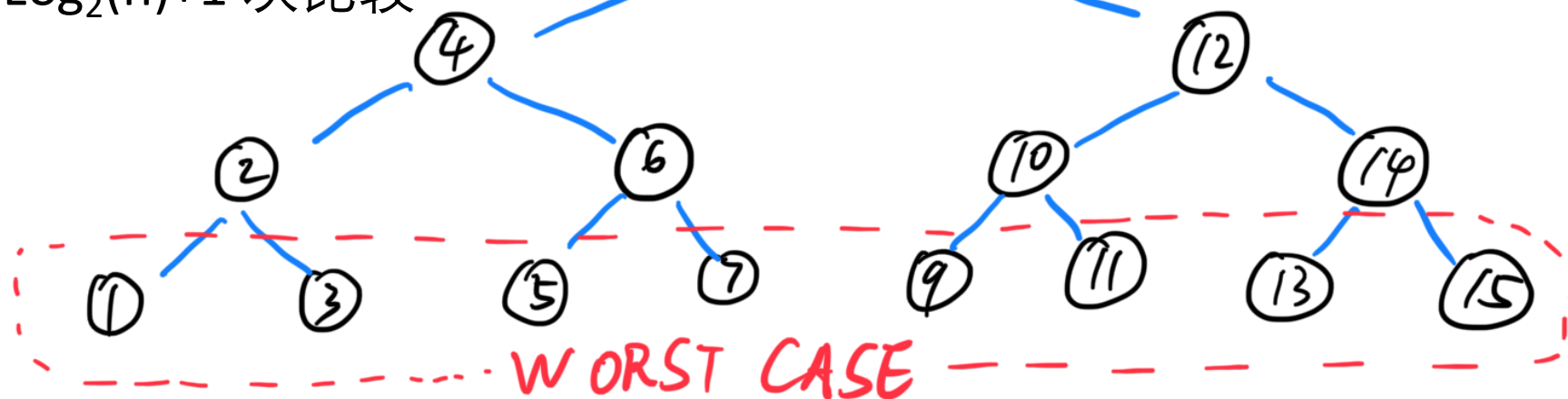
二分查找时间性能分析

$N = 15$

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮

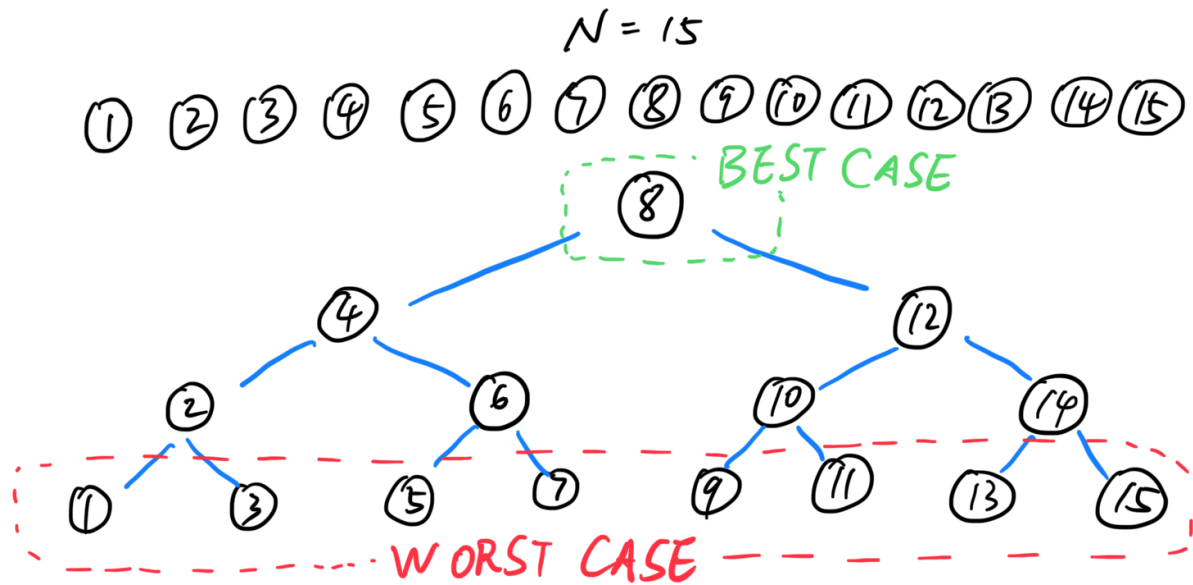
最坏情况的时间性能：
需进行 $\log_2(n)+1$ 次比较

BEST CASE

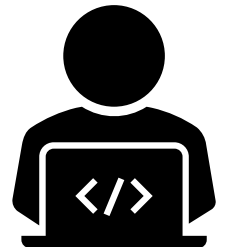




二分查找时间性能分析



二叉树的深度 $\lfloor \log_2 N + 1 \rfloor$



代码演示

chapter5-search-time-
compare.cpp



思考：二分查找的缺陷？

- 前提条件：需要已排序的数组

Sorting Algorithms

排序算法

(🔥)





选择排序法 (Selection Sort)

- 选择排序法：
 - 在所有元素中找到最小的元素放在数组的第0个位置
 - 在剩余元素中找出最小的放在第1个位置
 - ...
 - 以此类推，直到所有元素都放在适当的位置

5 3 4 1 2



选择排序伪代码

5 3 4 1 2

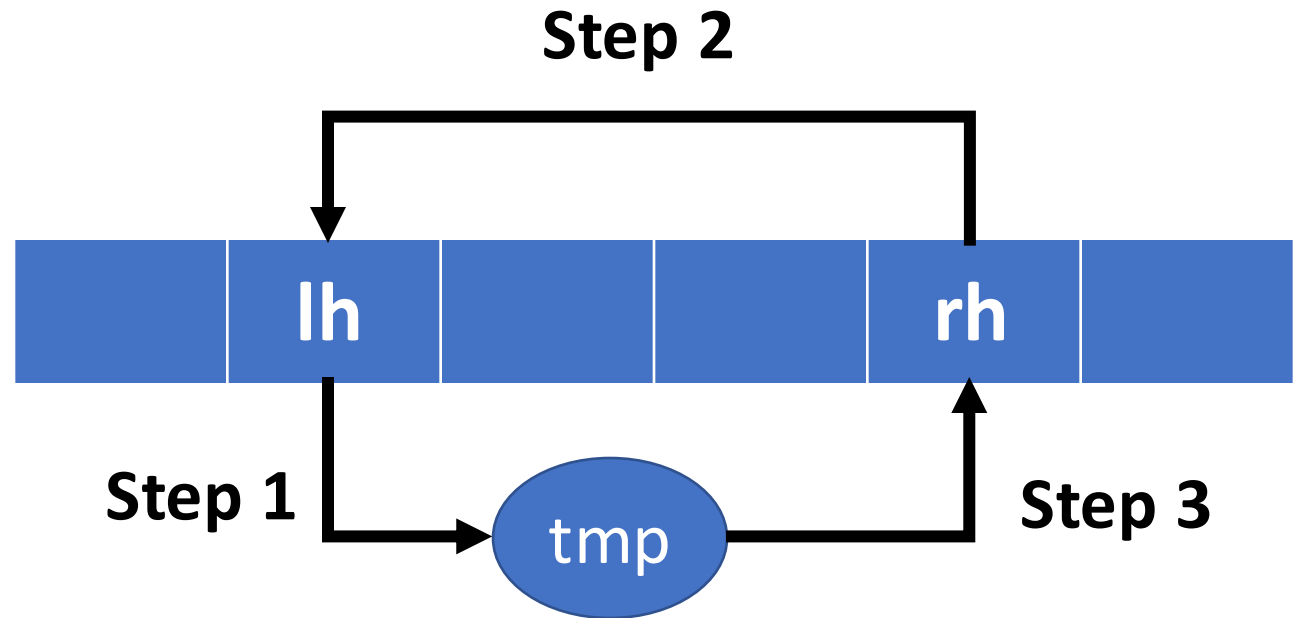
```
int lh, rh, array;  
输入要排序的元素, 存入array;  
for (lh = 0; lh < n; lh++)  
    {在array的从lh到n-1的元素之间找出最小元素, 记录其位置rh;  
      交换下标 lh和 rh中的值;} // 升序排列  
输出排好序的元素;
```



如何将数组中两个元素互换

~~array[lh] = array[rh];
array[rh] = array[lh];~~

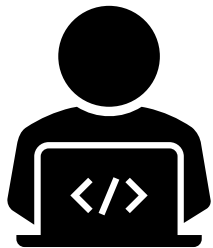
**tmp = array[lh];
array[lh] = array[rh];
array[rh] = tmp;**





选择排序算法的代码实现

```
int lh, rh, array;  
输入要排序的元素, 存入array;  
for (lh = 0; lh < n; lh++)  
    {在array的从lh到n-1的元素之间  
    找出最小元素, 记录其位置rh;  
    交换下标 lh和 rh中的值;}  
输出排好序的元素;
```



代码演示

chapter5-selectionSort.cpp

```
int main( )  
{ int lh, rh, k, tmp;  
  int array[ ] = {2, 5, 1, 9, 10, 0, 4, 8, 7, 6};  
  for (lh = 0; lh < 10; lh++)  
  {  
    rh = lh;  
    for (k = lh; k < 10; ++k)  
      if ( array[k] < array[rh] ) rh = k;  
    tmp = array[lh];  
    array[lh] = array[rh];  
    array[rh] = tmp;  
  }  
  for (lh = 0; lh < 10; ++lh) cout << array[lh] << ' ';  
  return 0;  
}
```



选择排序的效率

- 对n个元素的排序来说，找出第一个元素要比较n次，找出第二个元素比较n-1次，...，找出第n个元素比较一次。因此，总的比较次数为：

$$1 + 2 + 3 + \dots + n = n(n + 1) / 2$$

则称时间复杂度为 $O(n^2)$



章节回顾

- 数组两大特性：同类型，内存中有序排放
- 数组的初始化、下标越界
- 二分查找算法
 - 二分查找的思路
 - 要求**熟练掌握**并能**快速实现**
- 排序算法
 - 选择排序

冒泡排序法 (Bubble Sort)





冒泡排序法

- 对数组元素进行扫描。第一遍扫描冒出一个最大的气泡，放入最后一个位置。然后对剩余元素再进行第二次冒泡，冒出最大的泡放入倒数第二个位置，依次执行到最后一个元素



冒泡排序法

- 对数组元素进行扫描。第一遍扫描冒出一个最大的气泡，放入最后一个位置。然后对剩余元素再进行第二次冒泡，冒出最大的泡放入倒数第二个位置，依次执行到最后一个元素
- 如何放置：比较相邻的两个元素，如果大的在前小的在后，就交换这两个元素。这样经过从头到尾的检查，最大的一个就被交换到最后了。**每一遍都会排好一个元素**

Bubble sort

Array

6	3	0	5	1
---	---	---	---	---



冒泡排序算法

Bubble sort

Array

6

3

0

5

1

- 如果在一次起泡中没有发生交换，则表示数据都已排好序，不需要再进行起泡

For ($i=1$; $i<n$; $++i$)

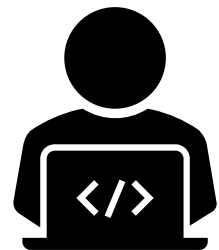
从元素0到元素 $n-i$ 进行冒泡，最大的泡放入元素 $n-i$;



冒泡排序算法的代码实现

```
int main()
{
    int a[ ] = { 0, 3, 5, 1, 8, 7, 9, 4, 2, 10, 6};
    int i, j, tmp, n = 11;
    bool flag;
    for (i=1; i<n; ++i) { //控制循环次数
        flag = false;
        for (j=0; j<n-i; ++j) // 第i次结束时，最后i个已经排好
            if (a[j+1] < a[j]){
                tmp = a[j]; a[j] = a[j+1];
                a[j+1] = tmp;
                flag = true;
            }
        if (!flag) break; /* 一趟冒泡中没有发生交换，排序结束 */
    }

    for (i=0; i<n; ++i) cout << a[i] << ' ';
    return 0;
}
```



代码演示

chapter5-
bubbleSort.cpp



冒泡排序的复杂度分析

- 对n个元素的排序来说，找出第一个元素要比较n次，找出第二个元素比较n-1次，...，找出第n个元素比较一次。因此，总的比较次数为：

$$n + n-1 + n-2 + \dots + 1 = n(n+1)/2$$

时间复杂度也为 $O(n^2)$

chapter5-sort-time-
compare.cpp



其他查找和排序算法

- 查找算法集合：<https://www.geeksforgeeks.org/searching-algorithms/>
- 排序算法集合：<https://www.geeksforgeeks.org/sorting-algorithms/>