



# 电子信息学院



武汉大学  
Wuhan University

## 算法与数据结构

### (基于现代C++的方法及实践)

#### ALGORITHM & DATA STRUCTURE IN MODERN C++

## 第一章 绪论


王文伟 Wang Wenwei, Dr.-Ing.

Tel: 189-71562600

Email: [wwwang@aliyun.com](mailto:wwwang@aliyun.com)


课程QQ群: 骆迦EIS数据结构与算法, 668792335

电子信息学院	Table of Contents	
本章位置	第1章 绪论	<p>本章是整个课程的绪论，将讨论数据结构与算法分析中重要的基本概念，如数据类型、数据结构、算法等，以及介绍算法分析的基本方法。</p>
	第2章 C++编程基础	
	第3章 遍历、迭代与递归	
	第4章 字符串	
	第5章 排序算法	
	第6章 线性表	
	第7章 栈与队列	
	第8章 数组和广义表	
	第9章 树和二叉树	
	第10章 图	
	第11章 查找算法	
TPL	第一章 绪论	2


电子信息学院	Table of Contents	武汉大学 Wuhan University	
<hr/>			
1.0 简介			
1.1 数据结构的基本概念			
1.2 算法与算法分析			
>>> C#编程基础 与数据集类型			
<hr/>			
TPL	第一章 绪论		3


## 1.0 Introduction

- ◆ 计算机应用领域迅速扩展，计算机技术日益重要。
- ◆ **软件设计**是计算机科学多个领域的核心和主体任务。
- ◆ 软件设计时要考虑的首要问题是数据的表示、组织和处理方法。**数据结构设计和算法设计是软件系统设计的核心。**
- ◆ “数据结构+算法=程序”。数据结构与算法探讨如何构建描述现实世界实体的计算模型并实现其操作。

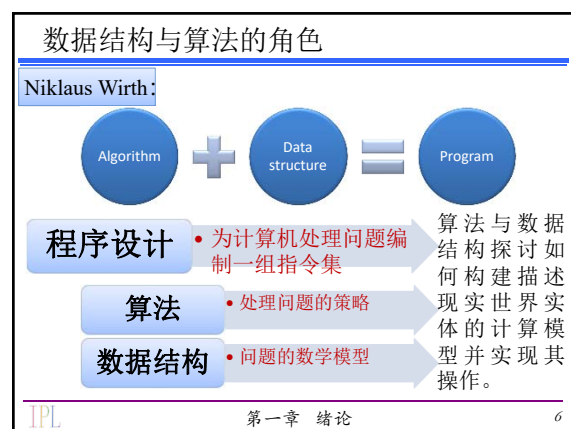
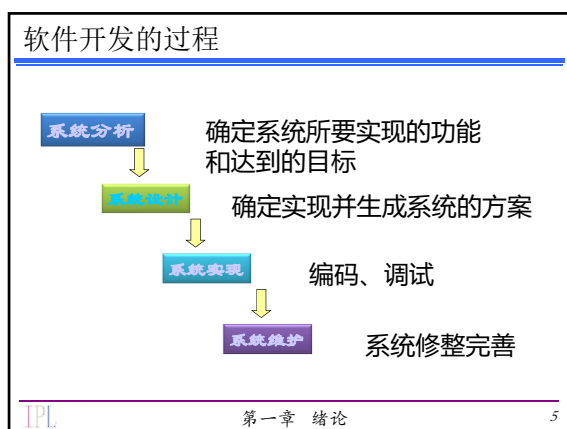


Niklaus Wirth:





第一章 绪论



## 1.1 数据结构的基本概念

- ◆ 数据结构是一门讨论“描述现实世界实体的数学模型及其操作在计算机中如何表示和实现”的学科。

### 1.1.1 数据类型与数据结构

#### 1.1.2 数据的逻辑结构

#### 1.1.3 数据的存储结构

#### 1.1.4 数据的操作

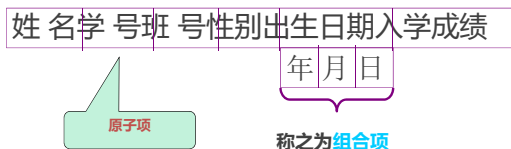
## 1.1.1 数据类型与数据结构

- ◆ **数据 (data)** : 数据是计算机程序的处理对象, 它可以是任何能输入到计算机的符号集合 (数据是以多种形式呈现的信息); 例如描述客观事物数量特征的**数值**数据以及名称特性的**字符**数据。
- ◆ **数据元素 (data element)** : 数据的基本单位是数据元素, 它是表示一个事物的一组数据, 有时又称为**数据结点**。通常在逻辑上作为一个整体进行考量和处理。
- ◆ **数据项 (data item/field)** : 一个数据元素可能分成若干成分, 构成数据元素的某个成分称作数据元素的数据项。数据项是数据元素的基本组成单位, 有时又称为**数据域 (data field)**。

## 数据元素与数据项 (以Student为例)

描述一个学生的**数据元素**由多个域构成, 其中每个域称为一个“**数据项**”。

**数据项是数据结构中讨论的最小单位。**



## 高级程序语言中的数据类型

在用高级程序语言编写的程序中, 必须对程序中出现的每个变量、常量或表达式, **明确说明它们所属的数据类型**。主流是强类型语言

例如, C语言中的**基本数据类型**有:

**字符型** char、**整型** int 和 **实型**(浮点型 float和双精度型 double)

## 数据类型与数据结构的意义示例

- ◆ **int类型**  
定义: 数据对象, 如: 3, 5, 7  
运算:  $3 + 7 = 10$  (方便运算, 高效, 易验证)
- ◆ **DateTime类型**  
定义: dt1, dt2  
运算:  $dt1 - dt2 = dt3$  (易处理, 高效, 易验证)
- ◆ **某数据结构 (类型)**  
定义: 数据对象, 如: ds1, ds2  
运算:  $ds1 \cap ds2 = ds3$  (易处理, 高效, 易验证)

## 数据类型的定义

- ◆ **数据类型 (data type)** 是一个“**值**”的集合和定义在此集合上的“**一组操作**”的总称。它定义了数据的性质、取值范围以及对数据所能进行的各种操作。例如, C++语言中整数类型 int32的值域是 $\{-2^{31}, \dots, -2, -1, 0, 1, 2, \dots, 2^{31}-1\}$ , 对这些值进行的操作包括加、减、乘、除、求模、相等或不等比较操作或运算等。

对程序员而言, 各种语言中的整数类型基本上是一样的, 因为它们的数学特性相同。

从这个意义上可称“**整数**”是一个抽象数据类型。

## 数据类型与抽象数据类型

高级程序设计语言都提供了一些**基本数据类型**，这些基本数据类型在数据处理程序中应用得最为频繁，但是它们往往不能满足程序设计中的所有需求，这时可以利用基本类型设计出各种复杂的**复合数据类型**，称为**自定义数据类型**。自定义数据类型要声明一个“值”的集合和定义在此集合上的“一组操作”。

计算机指令代码本身也是数据，可以为不同的代码片段（函数或过程）划分类型。一些高级编程语言中引入**委托**或**函数对象**等类型，用以表达特定的函数数据类型。

**数据类型**、**变量**及其可能的**值**构成程序设计的三个基本要素，变量是一块命名的内存空间，可以也只能存储由数据类型规定的值。

TPL

第一章 绪论

13

## 数据类型 => 抽象数据类型

为了描述更广泛的数据实体，数据结构和算法描述中更多地使用**抽象数据类型**（Abstract Data Type, ADT）：概念意义上的类型和这个类型上的逻辑操作集合。

狭义的**数据类型**指的是程序设计语言支持的数据类型；而**抽象数据类型**是数据与算法在较高层次的描述中用到的概念，是指一个概念意义上的类型和这个类型上的逻辑操作集合。最终演变为在常规数据类型支持下**用户新设计的高层次数据类型**。抛开细节的数据类型

TPL

第一章 绪论

14

## 抽象数据类型

(Abstract Data Type 简称**ADT**)

是指一个**数学模型**以及定义在该模型上的一组操作。

抽象数据类型有两个重要特征：

**“数据抽象” 和 “数据封装”**

TPL

第一章 绪论

15

## 数据抽象与数据封装

用ADT描述程序处理的实体时，强调的是数据的**本质特征**、**其所能完成的功能**以及它和**外部世界的接口**（即**外界使用它的方法**）。 - **数据抽象**

将数据实体的**外部特性**和其**内部实现**细节分离，并且对外部用户隐藏其内部实现细节。 - **数据封装**

TPL

第一章 绪论

16

## 抽象数据类型的描述方法

**ADT = (D, S, P)**

其中: **D** 是数据对象,

**S** 是 **D** 上的关系集,

**P** 是对 **D** 的基本操作集。

**数据对象**是个特定的数据子集，(一起)用来表示实体的特性。

TPL

第一章 绪论

17

## 抽象数据类型的描述方法(II)

**ADT 抽象数据类型名称{**

**数据对象：**〈数据对象的定义〉

**数据关系：**〈数据关系的定义〉

**基本操作：**〈基本操作的定义〉

**} ADT 抽象数据类型名称**

其中基本**操作**的定义格式为：

**基本操作名 (参数表)**

**初始条件：**〈初始条件描述〉

**操作结果：**〈操作结果描述〉

TPL

第一章 绪论

18

## 数据结构

粗略地说，**数据结构 (data structure)** 是指数据的不同成分之间所存在的某种关系特性，数据结构与**(抽象) 数据类型**是等价的。

狭义地说，对一个**数据 (元素) 的集合**来说，如果在数据元素之间存在一种或多种特定的关系，则称为**数据结构 (data structure)**。因此，“结构”就是指数据元素之间存在的**关系**。

数据结构可以看成是关于数据集合的数据类型，是一种特殊的抽象数据类型ADT。

TPL

第一章 绪论

19

## 数据结构的形式定义

数据结构是关于数据集合的特殊的（抽象）数据类型，它关注**三个方面的内容**：数据元素的特性、数据元素之间的关系以及由这些数据元素组成的数据集合所允许进行的操作。

**数据结构**是一个三元组

$\text{Data\_Structures} = (D, S, P)$

其中 **D** 是**数据元素的有限集**，

**S** 是 **D** 上**关系的有限集**。

**P** 是**数据集合所允许进行的操作**

TPL

第一章 绪论

20

## 例1.1：学生和学生信息表

学生类（型）**Student**定义  
(类型种类：**struct**类型，类型名称：**Student**)

```
struct Student{
    int id;
    string name;
    float age;
    double score;
}
```

TPL

第一章 绪论

21

## 学生信息表

学号	姓名	性别	年龄	成绩
200518001	王兵	男	18	85
200518002	李霞	女	19	92
200518003	张飞	男	19	78

学生信息表

学生信息表是由**Student**类型的数据元素组成的、能够进行特定操作的**数据集合**，即学生信息表是一种特定类型的**数据结构**。

TPL

第一章 绪论

22

学生信息表(类型种类：**class**类型，类型名称：**StudentInfoTable**)

```
class StudentInfoTable{
    vector<Student> studentList;
    //学生信息表内部存储块
    int insert(const Student& st);
    //将新学生添加到表的结尾处
    bool contains(const Student& st) const;
    //确定某个学生是否在表中
    void sort(); //对表中元素进行排序
}
```

逻辑结构：一个班级的学生按学号排列就是**顺序关系**；按班长、组长、组员排列具有**层次关系**；

存储结构：不同学生既可按顺序存储于一个数组(**顺序结构**)，也可分散存储在不同位置但保持链接(**链式结构**)；

## 数据结构的两个方面(层次)

**逻辑结构** 是对数据元素之间的逻辑关系的描述，它可以用一个数据元素的集合和定义在此集合上的若干关系来表示。**侧重于数据集合的抽象特性**

**物理结构** 是数据结构在计算机中的表示和实现，故又称“**存储结构**”。数据的存储结构依赖于计算机，它是逻辑结构在计算机中的实现。

TPL

第一章 绪论

24

### 1.1.2 数据的逻辑结构

数据的逻辑结构侧重于数据集合的抽象特性，它描述集合中数据元素之间的逻辑关系。按照数据集合中数据元素之间存在的不同特性的逻辑关系，数据结构可以分为三种基本类型：

**线性结构**



1 to 1

**树形结构**



1 to n

**图状结构**



n to n

**集合结构**



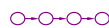
TPL

第一章 绪论

25

### 定义

◆ **线性结构**：每个数据元素只有一个前驱数据元素和一个后继数据元素，**逻辑上的顺序关系**。数组是最基本的具有线性结构的集合，其他常用的线性结构有**线性表、栈、队列**等



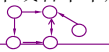
1 to 1

◆ **树结构**：每个数据元素只有一个前驱数据元素，可有零个或若干个后继数据元素，**层次关系**。



1 to n

◆ **图结构**：每个数据元素可有零个或若干个前驱数据元素，零个或若干个后继数据元素，**网状关系**。



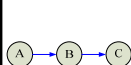
n to n

TPL

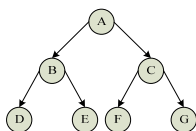
第一章 绪论

26

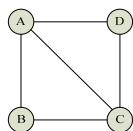
### 图示法表示数据的逻辑结构



(a) 线性结构



(b) 树结构



(c) 图结构

圆圈表示一个数据元素（数据结点），圆圈中的字符或数字表示数据元素的标记或数据元素的值，连线则表示数据元素间的逻辑关系。

TPL

第一章 绪论

27

### 1.1.3 数据的存储结构

◆ 数据的逻辑结构是软件设计人员从逻辑关系的角度观察和描述数据，而为了在计算机中实现对数据的操作，还需要按某种方式在计算机中表示和存储这些数据，这指的是数据的存储结构。

◆ 数据集合在计算机中的存储表示方式称为数据的存储结构，也称为物理结构。有**两种基本的存储结构**：

➢ 顺序存储结构

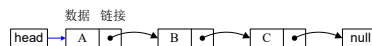
➢ 链式存储结构

用不同方式组合这两种基本存储结构，可以产生复杂的存储结构

◆ 数据的逻辑结构独立于计算机，而存储结构则依赖于计算机。

下标 数据

0	A
1	B
2	C



(a) 顺序存储结构

(b) 链式存储结构

TPL

第一章 绪论

28

### 顺序存储结构

◆ 顺序存储结构将数据集合的元素存储在一块**地址连续的内存空间**中，并且逻辑上相邻的元素在物理上也相邻。

**以 x 和 y 之间相对的存储位置表示顺序关系**

**例如**：y 的存储位置和 x 的存储位置（之间的关系）取决于它们之间的逻辑位置。



顺序存储结构中只包含元素本身的信息。典型的顺序结构是**数组**

TPL

第一章 绪论

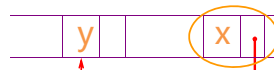
29

### 链式存储结构

链式存储结构使用称为**链结点（link node）**的扩展类型存储各个数据元素，结点由**数据元素域**和指向其他结点的**指针域**组成，链式存储结构使用指针把相互关联的结点链接起来。逻辑上相邻的数据元素在物理上不一定相邻，数据间的逻辑关系表现在结点的链接关系上。

以附加信息（指针）表示后继关系

结点类型的定义



```
struct Node{
    int item;
    //数据域，存放结点值
    Node* next;
    //指针域，指向后继结点
}
```

TPL

第一章 绪论

30

## 数据存储结构的选择

- ◆ 在软件设计时，既要用正确的逻辑结构描述要解决的问题，还应选择一种合适的存储结构，使得所实现的程序在以下两方面的综合性能最佳：数据操作所花费的时间和程序所占用的存储空间。
- ◆ 例如，对于线性数据集合的存储，可以按下面两种情况分别处理：
  - 当不需要频繁插入和删除时，可以采用顺序存储结构，此时占用的存储空间少，访问数据的时间效率高。
  - 当插入和删除操作很频繁时，需要采用链式存储结构。此时虽然占用的存储空间较多，但操作的时间效率高。这种方案以存储空间为代价换取了较高的时间效率。

## 1.1.4 数据的操作

- ◆ 对数据结构对象进行的某种处理称作**数据的操作**。
- ◆ 每种特定的逻辑结构都有一个自身的操作集合，不同的逻辑结构有不同的操作。线性结构的操作：
  - 访问集合中某数据元素，更新数据元素值(**get/set**)。
  - 统计集合中数据元素的个数(**count**)。
  - 插入数据元素：在数据结构中增加新的结点(**insert**)。
  - 删除数据元素：将指定元素从数据结构中删除(**remove**)。
  - 查找：在数据结构中查找满足一定条件的数据元素。插入、删除、更新操作都包括一个查找操作，以确定需要插入、删除、更新数据元素的确切位置(**search**)。
  - 排序：在线性结构中数据元素数目不变的情况下，将数据元素按某种指定的顺序重新排列(**sort**)。

## 数据操作的具体实现与存储结构有关

- ◆ 数据的操作一般是根据数据的逻辑结构定义的，但即使是同一种逻辑结构，操作的具体实现（算法）可能与数据的存储结构有关。
- ◆ 例如对于线性表，选择顺序存储结构还是链式存储结构对于插入或删除操作，都会造成不同的实现方式。

## 1.2 算法与算法分析

### 1.2.1 算法

### 1.2.2 算法设计

### 1.2.3 算法效率分析

### 1.2.1 算法

#### 1. 算法定义

《The Art of Computer Programming》

D.Knuth

算法 (Algorithm) 是对特定问题求解过程的一种描述，是解决该问题的一个确定的、有限的操作序列。

一个算法必须满足以下五个重要特性：

1. 确定性
2. 可行性
3. 有穷性
4. 有输入
5. 有输出

### 算法的特性

**确定性** 对于每种情况下所应执行的操作，在算法中都有确切的规定，算法的執行者或阅读者都能明确其含义，并且在任何条件下，算法都只有一条执行路径。

**可行性** 算法中的所有操作都必须足够基本，都可以通过已经实现的基本操作运算有限次予以实现。

**有穷性** 对于任意一组合法输入值，在执行有穷步骤之后一定能结束。算法中的每个步骤都能在有限时间内完成。



## 算法的特性(II)

**有输入** 算法有零个或多个输入数据，即算法的加工对象。有些输入量需要在算法执行过程中输入，而有的算法表面上可以没有输入，实际上输入已被嵌入算法之中。

**有输出** 算法有一个或多个输出数据，它们是算法进行信息加工后得到的结果，与“输入”数据之间有确定关系，这种关系体现算法的功能。

## 2. 计算模型

◆ 指算法实现技术的模型。讨论的计算模型适合于常用计算机，具有两方面的基本特性：

1) 采用**通用单处理器**，在同一时间执行一条**指令**，并且执行的指令是确定的，程序以指令一条接一条地执行的方式实现算法。`int x = 9; x = x + 1;`

2) 随机存储机 (random access machine, RAM) 模型，处理器可以**随机访问**存储器。

◆ 单条指令完成的任务很基本，但计算机能**精确、高速**完成基本指令，并能不厌其烦地重复做简单的事情。

◆ 高级编程语言中的程序语句最终可化解为采用二进制形式表示的数据和指令；另一方面，完成某任务的若干语句可以抽象为单语句。

◆ 简单指令和计算机求解的问题之间存在巨大的鸿沟，**算法**是架接二者的桥梁。随着硬件性能的不断增长以及算法越来越丰富，计算机能用来解决越来越复杂的问题。

## 3. 算法的描述方式

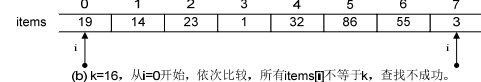
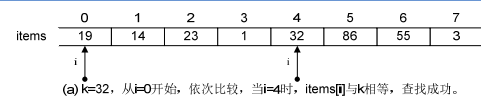
◆ 算法过程可用**文字、流程图、高级程序设计语言**或类同于高级程序设计语言的**伪码**描述。

◆ 无论哪种描述形式，都要体现出算法是由**语义明确的操作步骤组成的有限操作序列**，它精确地描述出怎样从给定的输入信息逐步得到要求的输出信息。

◆ 算法的执行者和阅读者都能明确其含义。

### 【例1.3】线性表的顺序查找算法

在线性表中，按**关键字**进行**顺序查找** (sequential search) 的算法思路为：对于给定值**k**，从线性表的一端开始，依次与每个元素的关键字进行比较，如果存在关键字与**k**相同的数据元素，则查找**成功**；否则查找**不成功**。



## 4. 算法与数据结构的关系

◆ 数据的**逻辑结构**、**存储结构**以及对数据所进行的**操作**三者是相互依存的。

➢ 在研究一种数据结构时，总是离不开研究对这种数据结构所能进行的各种操作，因为，这些操作从不同角度体现了这种数据结构的某种性质，只有通过研究这些操作的算法，才能更清楚地理解这种数据结构的性质。

➢ 反之，每种算法都是建立在特定的数据结构上的。数据结构和算法之间存在着本质的联系，失去一方，另一方就没有意义。

(1) 不同的逻辑结构需采用不同的算法。

每种特定的逻辑结构都有一个自身的操作集合，在不同的逻辑结构中插入和删除元素的操作算法是不同的。在后续各章中，讨论不同的数据逻辑结构，均需探讨基本操作算法的不同。

(2) 同样的逻辑结构因为存储结构的不同而采用不同的算法。

线性表可以用**顺序存储结构**或**链式存储结构**实现，不同存储结构的线性表上的插入、删除、排序等算法是不同的。例如，冒泡排序、折半插入排序等算法适用于顺序存储结构的线性表；适用于链式存储结构线性表的排序算法有直接插入排序、简单选择排序等。

(3) 同样的逻辑结构和存储结构，因为要解决问题的要求不同而采用不同的算法。

【例1.4】大规模线性表的**分块查找**（blocking search）算法

**顺序查找**算法适合于数据量较小的线性表，如学生成绩表。一部按字母顺序排序的**字典**也是一个顺序存储的线性表，具有与学生成绩表相同的逻辑结构和存储结构，但数据量较大，采用顺序查找算法的效率会很低，此时可以采用分块查找算法。

#### 【例1.4】字典的分块查找算法

◆ 字典是由首字母相同、大小不等的若干块（block）所组成的，为使查找方便，每部字典都设计了一个**索引表**，指出每个字母对应单词的起始页码。

◆ 字典分块查找算法的基本思想：将所有单词排序后存放在数组dict中，并为字典设计一个**索引表index**，index的每个数据元素由两部分组成：首字母和下标，它们分别对应于单词的首字母和以该字母为首字母的单词在dict数组中的起始下标。

◆ 使用分块查找算法，在字典dict中查找给定的单词token，必须分两步进行：

1. 根据token的首字母，查找索引表index，确定token应该在dict中的哪一块。
2. 在相应数据块中，使用顺序查找算法查找token，得到查找成功与否的信息。

#### 1.2.2 算法设计的要求

正确性

可读性

健壮性

高时间效率

高空间效率

#### 算法的正确性（Correctness）

- ◆ **首先**，算法应确切地满足具体问题的需求，这是算法设计的基本目标。
- ◆ **其次**，对算法是否“正确”的理解可以有以下四个层次：
  - 不含语法错误；
  - 对于某几组输入数据能够得出满足要求的结果；
  - **程序对于精心选择的、典型、苛刻且带有刁难性的几组输入数据能够得出满足要求的结果；**
  - 程序对于一切合法的输入数据都能得出满足要求的结果；

#### 算法的可读性（Readability）

- ◆ 算法既是为了计算机执行，也是为了人的**阅读与交流**。因此算法应该**易于人的理解**，这既有利于程序的调试和维护，也有利于算法的交流和移植。
- ◆ 相反，晦涩难读的程序易于隐藏较多错误而且难以调试；
- ◆ 算法的可读性主要体现在两方面：一是被描述算法中的类名、对象名、方法名等的**命名要见名知意**；二是要有足够多的清晰**注释**。



## 算法的健壮性 (Robustness)

- ◆ 当输入的数据非法时，算法应当恰当地作出反应或进行相应处理（**错误或异常处理**），而不是产生莫名奇妙的输出结果。
- ◆ **错误或异常处理**：处理出错的方法不应是中断程序的执行，而应是返回一个表示**错误或错误性质的值**，以便在更高的抽象层次上进行处理。

## 高效性 (Efficiency)

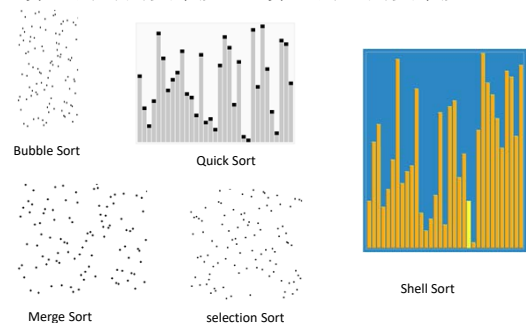
- ◆ 算法的执行时间应满足问题的需求。执行时间短的算法称为**高时间效率**的算法。
- ◆ 算法在执行时一般要求额外的内存空间。内存要求低的算法称为**高空间效率**的算法。
- ◆ 算法应满足高时间效率与低存储量需求的目标，对于同一个问题，如果有多个算法可供选择，应尽可能选择执行时间短和内存要求低的算法。
- ◆ 但算法的高时间效率和高空间效率通常是矛盾的，在很多情况下，首先考虑算法的时间效率目标。

## 通用性和可复用性

- ◆ 在算法设计实践中，人们往往也希望所设计的算法具有通用性和可复用性，降低或排除设计中的重复工作。软件工程领域流传的一句经典名言“不要重复发明车轮”体现了对算法设计可复用性的倡导。
- ◆ 在各种现代高级程序设计语言中的循环结构、函数编程、结构化编程、面向对象编程，以及泛型编程等都可以说是对这一倡导的响应，并在不同程度上对算法设计的可复用予以支持。

## 1. 2. 3 算法效率分析

### 1. 算法的时间复杂度 2. 算法的空间复杂度



## 1. 算法的时间复杂度 (time complexity)

算法的执行时间等于所有语句执行时间的总和，它取决于**控制结构**和**原操作**两者的综合效果。通常选取一种对于所研究的问题来说是基本操作的原操作，以该**基本操作**重复执行的次数作为算法的**时间度量**。

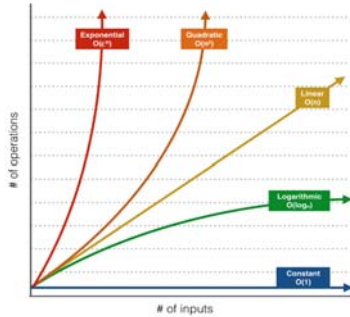
算法重复执行**原操作**的次数是算法所处理的**数据个数 $n$ 的函数 $f(n)$** ，其渐进特性称为该**算法的时间复杂度**，表示为 **$T(n)=O(f(n))$** 。on Order of  $f(n)$

$O(1)$ 表示时间是一个常数，不依赖于 $n$ ； $O(n)$ 表示时间与 $n$ 成正比，是线性关系， $O(n^2)$ 、 $O(n^3)$ 、 $O(2^n)$ 分别称为平方阶、立方阶和指数阶； $O(\log_2 n)$ 为对数阶。

## 时间复杂度随问题规模 $n$ 变化情况的比较

时间复杂度	$n=8(2^3)$	$N=10$	$n=100$	$n=1000$
$O(1)$	1	1	1	1
$O(\log_2 n)$	3	3.322	6.644	9.966
$O(n)$	8	10	100	1000
$O(n \log_2 n)$	24	33.22	664.4	9966
$O(n^2)$	64	100	10 000	$10^6$

## Big-O Complexity Plot



## 【例1.5】分析算法片段的时间复杂度

- ◆ 时间复杂度为 $O(1)$ 的简单语句。

```
sum = 0;
```

- ◆ 时间复杂度为 $O(n)$ 的单重循环。

```
int n = 100, sum = 0;
for(int i=0; i<n; i++)
    sum += a[i];
```

## 【例1.5】分析算法片段的时间复杂度（2）

- ◆ 时间复杂度为 $O(n^2)$ 的二重循环。

```
int n = 100;
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
        cout << i*j;
```

二重循环中的循环体语句  
被执行 $n \times n$ 次

```
int n = 100;
for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
        cout << i*j;
```

二重循环的执行次数为

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

## 【例1.5】分析算法片段的时间复杂度（3）

- ◆ 时间复杂度为 $O(n \log_2 n)$ 的二重循环。

```
int n = 64;
for(int i=1; i<=n; i*=2)
    for(int j=1; j<=n; j++)
        cout << i*j;
```

- ◆ 外层循环每执行一次， $i$ 就乘以2，直至 $i > n$ 停止，外层循环执行 $\log_2 n$ 次。内层循环执行次数恒为 $n$ 。则时间复杂度为 $O(n \log_2 n)$ 。

## 【例1.5】分析算法片段的时间复杂度（4）

- ◆ 时间复杂度为 $O(n)$ 的二重循环。

```
int n = 64;
for(int i=1; i<=n; i*=2)
    for(int j=1; j<=i; j++)
        cout << i*j;
```

- ◆ 外层循环执行 $\log_2 n$ 次。内层循环执行 $i$ 次，随着外层循环的增长而成倍递增。此时时间复杂度为 $O(n)$ 。

$$\sum_{i=1}^{\log_2 n} 2^i = O(n)$$

## 2. 算法的空间复杂度

算法的执行除了需要存储空间来寄存本身所用指令、变量和输入数据外，也需要一些对数据进行操作的工作单元和存储一些为实现算法所需的辅助空间。与算法的时间复杂度概念类似，算法在执行时**所需辅助内存空间大小与待处理的数据量之间的关系**称为算法的**空间复杂度**（space complexity），也用 $O(f(n))$ 的形式表示。

在冒泡排序过程中，需要一个辅助存储空间来交换两个数据元素，这与序列的长度无关，故冒泡排序算法的空间复杂度为 $O(1)$ 。归并排序算法在运行过程中需要与存储原数据序列的空间相同大小的辅助空间，所以它的空间复杂度为 $O(n)$ 。

## 现代C++简介

- ◆ 用单独的课件归纳C++快速入门的要点。
- ◆ 用单独的课件介绍C++的数据集合类型。
- ◆ 用一个多媒体教程学习C++程序设计基础。

- ◆ C++的精髓在面向对象编程OOP。
- ◆ OOP技术较好地解决“软件复用”的问题。
- ◆ 深入理解OOP就会拥有站在巨人肩上的阶梯。

## 面向对象技术对数据结构与算法描述的优势

- ◆ **面向过程**的程序设计以分离方式描述数据和对数据的操作，所设计的代码往往具有重用性差、可移植性差、数据维护困难等缺点。
- ◆ 数据的逻辑结构、存储结构以及对数据所进行的操作三者实际上是相互依存、互为一体的，所以**用封装、继承和多态等OOP编程特性能够更深入地刻画数据结构**，并且带来好的软件可复用性。

## 举例：OOP视角看string类

- ◆ 为处理字符串数据声明为**string**类，而串连接、串比较等操作则声明为该类的方法，**string**类的**设计者**用面向对象思想设计这个类并实现其中的方法。此时数据的描述和对数据的操作都**封装**在同一个**以类string为单位的模块**中，因此增强了代码的重用性、可移植性，使数据与代码易于维护。
- ◆ **string**类的**使用者**，只需要知道该类对外的接口（即类中的公共方法和属性）即可方便地应用“字符串”这样一种数据结构。
- ◆ 类似的例子很多，如：vector，map，ofstream

## 通过实例快速入门C++编程

- ◆ 编程语言本身内容都相对简单，但掌握编程却比较繁杂。原因：1）面对的问题非常广泛；2）作为程序运行基础的系统非常繁杂；3）编程思想相对抽象。
- ◆ 通过实例快速体验现代C++的便捷和面向对象编程的威力，开始构建站在巨人肩上的阶梯。

```
#include <iostream> #include <string> #include <vector>
using namespace std;
int main (int argc, char* argv[]) {
    vector<string> words{ "Hello", "C++", "World", "!" };
    for (const auto& word : words) {
        cout << word << " ";
    } cout << endl;
    if (argc == 3)
        cout << "你输入的命令参数是：" << argv[1] << "和" << argv[2] << endl;
    return 0;
}
```

## 实例中的几个要点（编程要素）

1. C++具有与C/C#/Java等语言相似的词法、语法；
2. **main 函数**：C++ 程序控制从main函数开始和结束。程序任务体现：在 main函数中创建对象和执行其他函数。
3. 类：封装相关方面的数据和操作，类的设计和使用是编程的中心问题。
  - 使用：用到的类iostream、string、vector，类库中有大量可用的类。**找到合适的对象做正确的事情**；
  - 设计：自定义类，**为现实世界实体和问题建立合适的模型**。
4. 输入和输出：cout，cin，<<，>>
5. 注释：单行注释//，多行注释/\* \*/
6. 编译compile：cl hw.cpp => hw.exe
7. 执行execute：D> hw.exe ¶

## 类与对象是面向对象编程的灵魂

OOP

**类class**：描述所属的一类对象的蓝图  
**对象object**：根据蓝图生成的具体实例instance  
先定义类，再创建类的对象（实例），然后用它完成任务。  
设计者class A {...} 客户端A o; 或A\* p=new A;

- ◆ **对象是类的实例**，属于某个已知的类。**对象声明**的格式为：<类名> <对象名>，例：**stack<int> s;**
- ◆ 或用指针配合**new**操作符**创建**新的对象，并为之分配内存。它调用类的构造函数来初始化对象。  
**<类名>\* po = new <类名>(<参数列表>);**  
**stack<string>\* ps = new stack(20);**
- ◆ 通过对象引用类的**public成员变量**：  
**<对象名>.<成员名>** s.Count 或 ps->Count
- ◆ 通过对象调用**public成员方法**：  
**<对象名>.<方法名> (<参数列表>)** ps->push\_back("World")

## Object Oriented: “all are objects!”

```
// 狭义下变量指标量类型、对象指复合类型
void main() {
    int a = 10;    float pi = 3.14;
    string s( "Hello World!");
    Student* st = new Student(172001, "Zhang San");
    int ia[] = {0,1,2,3};
    string sa[] = {"C#", "says", "Hello", "World", "!"};
    Student sta[] = {{172001, "Zhang San"}, {...}}
                  {172010, "Li Si"}}
}
```

## IO操作和文件系统-认识 IO类

```
ifstream  infile("Input.txt");
ofstream  outfile("Output.txt");
auto pfbuf = infile.rdbuf();
outfile << pfbuf;
//cout << pfbuf;
infile.close();
outfile.close();
```

- ◆ 注释: 1) 类: `ifstream` ; `ofstream`
- 2) `using namespace std;`
- 3) `#include <fstream> #include <iostream>`

## 提前看【例】利用栈进行数制转换

$$N = a_n \times d^n + a_{n-1} \times d^{n-1} + \dots + a_1 \times d^1 + a_0$$

数制转换就是要确定序列

$$\{a_0, a_1, a_2, \dots, a_n\} \quad N = (N / d) \times d + N \% d$$

例如:  $(2468)_{10}$  转换成  $(4644)_8$  的运算过程如下:

	N	N/8	N%8	
计算 顺序 ↓	2468	308	4	输出 顺序 ↑
	308	38	4	
	38	4	6	
	4	0	4	

## 利用栈进行数制转换

```
void d2o(int n) {
    // 对于输入的任意一个非负十进制整数, 打印输出与其等值的八进制数
    stack<int> s;
    cout << "十进制数:" << n << " -> 八进制:";
    while (n != 0) {
        s.push(n % 8); // "余数"入栈
        n = n / 8;      // 非零"商"继续运算
    }
    // 和"求余"所得相逆的顺序输出八进制的各位数
    while (!s.empty()) {
        cout << s.top();    s.pop();
    }
    cout << endl;
}
```

要点: 1. 找到合适的对象`stack`; 2. 构建实例  
3. `#include <stack>` `using namespace std;`

## 泛型(泛型函数和泛型类)

- ◆ 泛型编程具备可重用性、类型安全和效率等方面的优点, 这是非泛型编程所不具备的。C++ 使用模板 (template) 支持泛型编程, 包括函数模板和类模板。
- ◆ C++ 语言中泛型的优越性在下面的一段例子中应能较好的显示出来。对于同样的运算逻辑 (例子中是交换两个变量的内容), 但仅是数据的类型不一样, 原始编程模式可能就需要定义一堆相似的函数; 而应用泛型编程, 程序员可以编写出与类型无关的代码, 即仅需定义一个模板函数 (例子中是 `myswap`)。

## 泛型方法

```
using namespace std;
void swapint(int& x, int& y) {int t = x; x = y; y = t;}
void swapdouble(double& x, double& y) {double t=x; x=y; y=t;}
template<typename T>
void myswap(T& x, T& y) {T t = x; x = y; y = t;}
int main() {
    int a=3,b=7; cout<<"a="<<a<<"\tb="<<b<<endl;
    swapint(a, b);cout<<"after swap"<<endl;
    // 无泛型机制的年代
    cout<<"a="<<a<<"\tb="<<b<<endl;
    double ad=3.5,bd=7.5;cout<<"ad="<<ad<<"\tbd="<<bd;
    swapdouble(ad, bd);cout<<"after swap"<<endl;
    cout<<"ad="<<ad<<"\tbd="<<bd<<endl;
    // 应用泛型机制的年代
    cout<<"a="<<a<<"\tb="<<b<<endl;
    myswap(a, b); cout<<"after swap"<<endl;
    cout<<"ad="<<ad<<"\tbd="<<bd<<endl;}
```

## 泛型类应用举例：构造特定类型的列表

```
vector<int> * pa = new vector<int> ();
// 声明并构造int型数列表
pa->push_back(86); pa->push_back(100);
// 向列表中添加整型元素
vector<int> nums {0,1,2,3};
vector<string> s; // 声明并构造字符串列表
s.push_back("Hello"); s.push_back("C++11");
vector<Student> st; // 声明并构造学生列表
st.push_back(Student(8001, "王兵", 92));
```

## 构造特定类型的列表

```
vector<Student> sl {{3016, "张超", 89}, {3053, "马飞", 80}, {3041, "刘羽", 96},
{3025, "赵备", 79}, {3039, "关云", 85}};
sl.insert(begin(sl)+2, Student(3000, "马超", 95));
for(const auto& item: sl)
    cout << item;
```

集合初始化

基于范围的循环  
与可迭代集合

类定义举例: class Student /struct Student

```
class Student{
private:
    int _studentID; string _name; double _score;
    // 私有成员变量
public:
    // 带参数的构造函数
    Student(int id, const char* name, double score):
        _name(name) { _studentID = id; _score = score; }
    // copy constructor 拷贝构造函数
    Student(const Student& s): _name(s._name) {
        _studentID = s._studentID; _score = s._score; }
    int ID() const { return _studentID; }
    int& ID() { _studentID; }
} // 后面对Student类型重载>>和<运算符
```

## 使用类

```
void main() {
    // 用new操作符创建对象
    Student* ps1= new Student(172001, "Zhang San");
    Student s2(*ps1); s2.ID()= 172100;
    cout << "Student #1: " << *ps1;
    cout << "Student #2: " << s2;
}
```

输出结果:

Student #1: 172001-Zhang San

Student #2: 172100-Zhang San

类定义举例: class Complex

```
class Complex{
private:
    double _rp = 0.0; // 复数的实部
    double _ip = 0.0; // 复数的虚部
    static double eps = 0.0; // 缺省精度
public:
    Complex(double r=0, double i=0): _rp(r), _ip(i) { }
    double real() const { return _rp; }
    double& real() { return _rp; }
    double imag() const { return _ip; }
    double& imag() { return _ip; }
    ..... // 实现复数操作的其他相关方法, 如加减乘除、
    指数对数、三角函数等。
}
```

## 使用（类库中的）类

```
#include <complex>
#include <iostream>
using namespace std;
void main() {
    complex c1(3.0, 4.0); // 创建对象并初始化
    complex c2(5.0, 0.0);
    complex c3 = c1 + c2;
    cout << "the sum is: " << c3;
}
```

输出结果:

the sum is: (8.00, 4.00)

## 程序命令行参数的处理 (char\*数组argv)

重回  
旧石  
器时  
代

- ◆ 编译源程序: D>cl cmd\_arg.cpp[回车]
- ◆ 运行程序: D>cmd\_arg 3 7[回车]

```
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[]) {
    if (argc == 3) {
        cout << "你输入的命令行参数是: " << argv[1]
              << "和" << argv[2] << endl;
        cout << argv[1] << " + " << argv[2] << " = "
              << stoi(argv[1]) + stoi(argv[2]) << endl;
    }
    return 0;
} //可见main函数通过argv字符串数组接受命令行参数, 提供程序运行灵活性
```

TPL

第一章 绪论

81

## 通过实验1 熟悉现代C++ (数组和类的定义)

### ◆ 实验目的:

理解C++的基本概念及其基本操作, 重点是数组的处理和新类型(class/struct)的定义。

### ◆ 题意:

1. 定义和初始化数组, 在数组中查找特定数据, 对数组中的数据进行排序。
2. 应用复数类模板complex, 实现复数的基本操作。

TPL

第一章 绪论

80

## 本章学习要点

- ◆ 熟悉各名词、术语的含义, 掌握基本概念。
- ◆ 理解算法五个要素的确切含义。
- ◆ 掌握计算语句频度和估算算法时间复杂度的方法。

TPL

第一章 绪论

81

## 几点要求

- ◆ 课前预习, 了解本堂课内容;
- ◆ 课后复习, 在理解教学内容的基础上做练习题;
- ◆ 独立完成作业;
- ◆ 加强编程实验: 重要的话重复多遍  
you haven't really learned something well  
until you've taught it to a computer. (Don  
Knuth)

TPL

第一章 绪论

82