

# 数据结构

Ch1 绪论

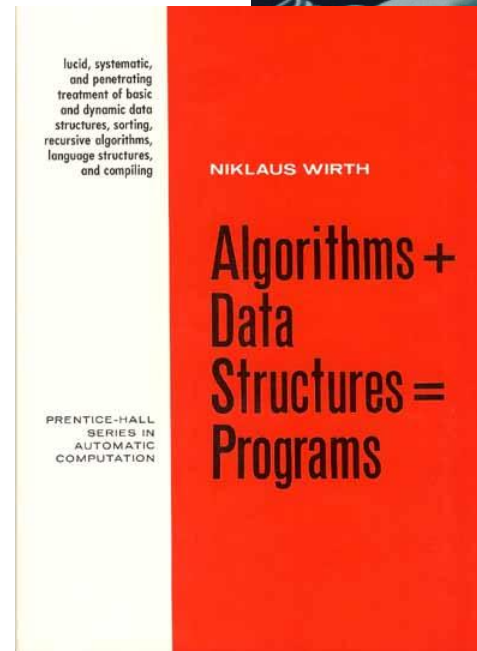
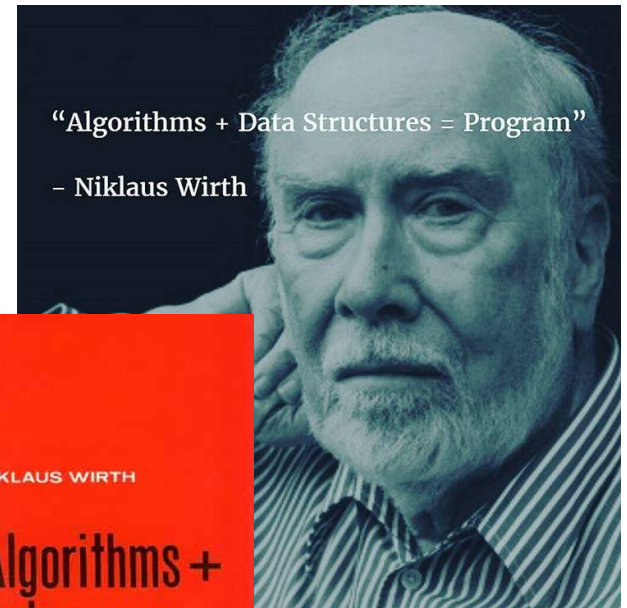
计算机学院（国家示范性软件学院）

# 第1章 绪论

- 1.1 引言
- 1.2 基本概念和术语
- 1.3 算法和算法分析
- 1.4 补充C/C++语言知识
- 1.5 本章知识点小结

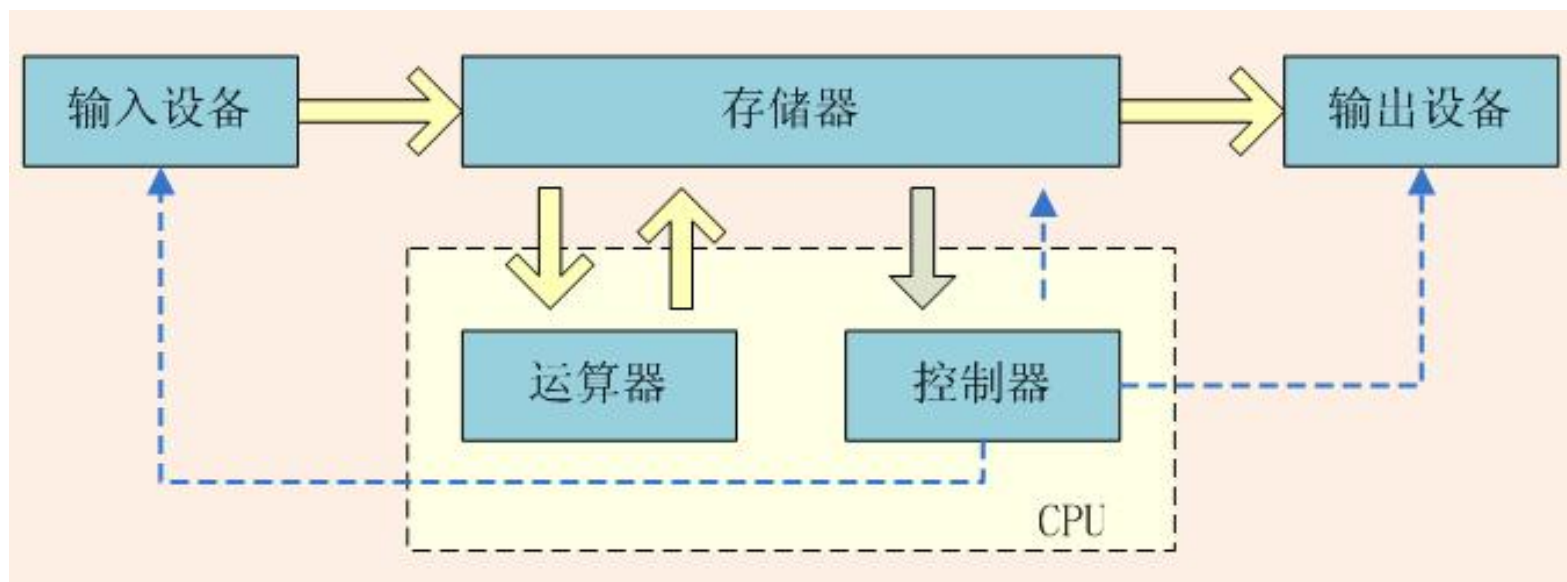
# 1.1 引言

- 程序 = 算法 + 数据结构 (Niklaus Wirth, PASCAL之父, 图灵奖得主)
- 数据结构关注数据在计算机中如何有效的组织起来
- 算法关注数据如何经过运算解决问题



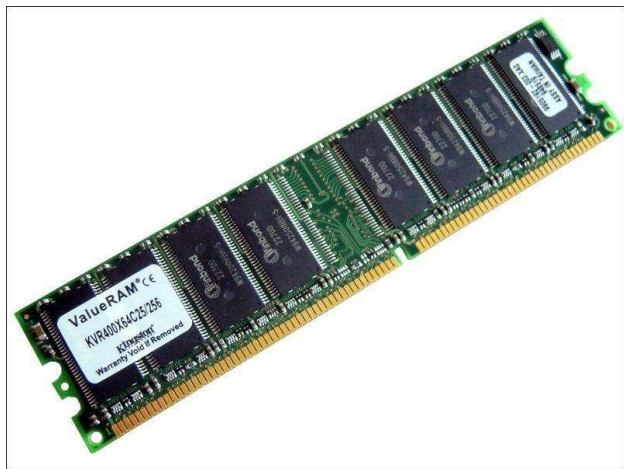
# 计算机中如何表示/存储和操作数据？

- 冯.诺依曼体系结构是现代计算机的基础，计算机由控制器、运算器、存储器、输入设备、输出设备五部分组成



# 计算机中如何表示/存储和操作数据？

- 按照与CPU的接近程度，存储器分为内存存储器与外存储器，简称内存与外存



内存

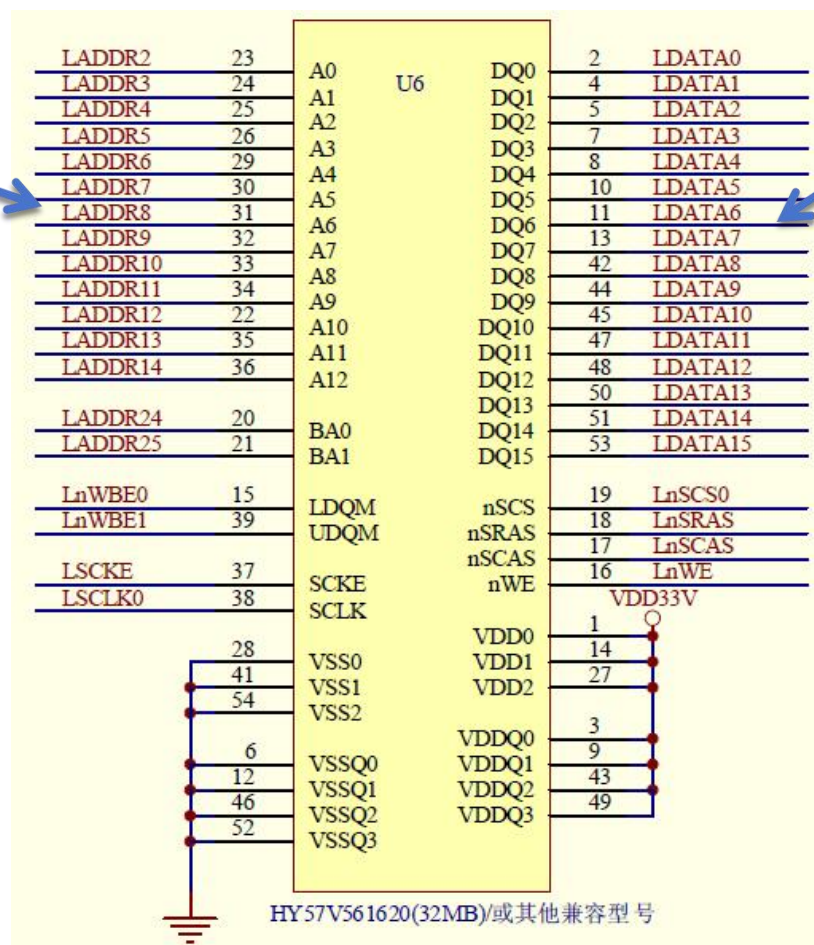


外存

# 计算机中如何表示/存储和操作数据？



地址总线



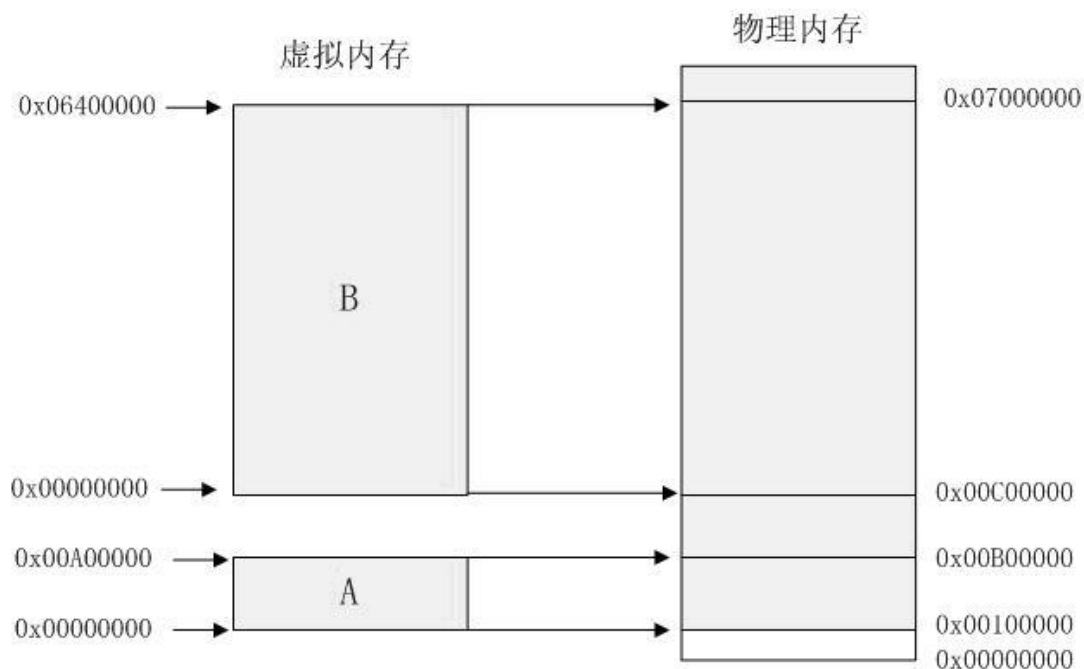
数据总线

# 计算机中如何表示/存储和操作数据？

操作系统为每一个进程提供了一个假象，好像每个进程都在独占的使用主存。每个进程看到的存储器都是一致的，称之为**虚拟地址空间**。

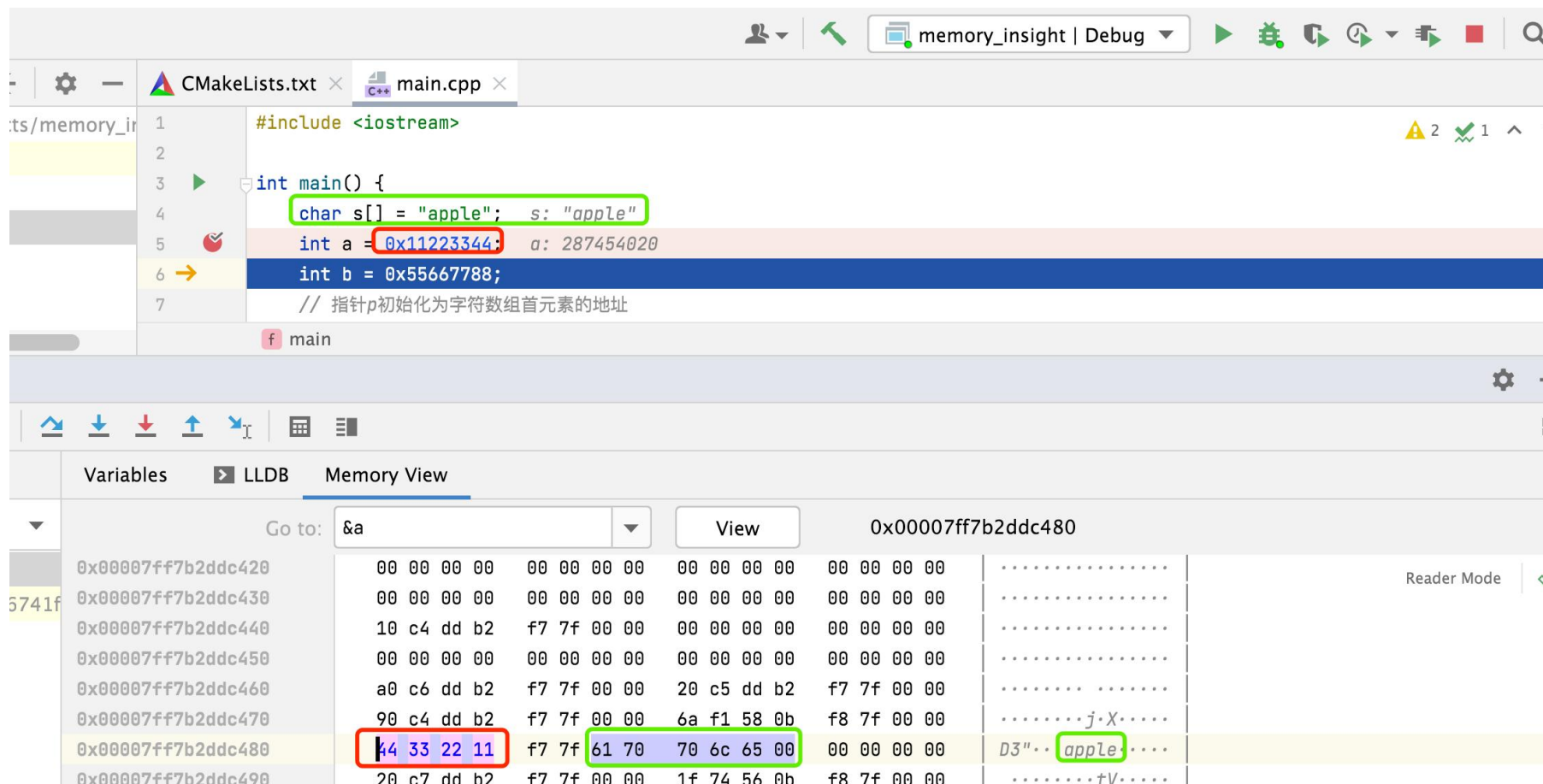
**32位系统虚拟地址空间为4GB**

**如0x00CC0000, 8位16进制数**





# 计算机中如何表示/存储和操作数据？



The screenshot displays a C++ IDE with a code editor and a memory view window. The code editor shows a C++ program with a character array 's' initialized to "apple". The memory view window shows the memory address 0x00007ff7b2ddc480, which contains the string "apple".

```
#include <iostream>

int main() {
    char s[] = "apple";    s: "apple"
    int a = 0x11223344;    a: 287454020
    int b = 0x55667788;
    // 指针p初始化为字符数组首元素的地址
}
```

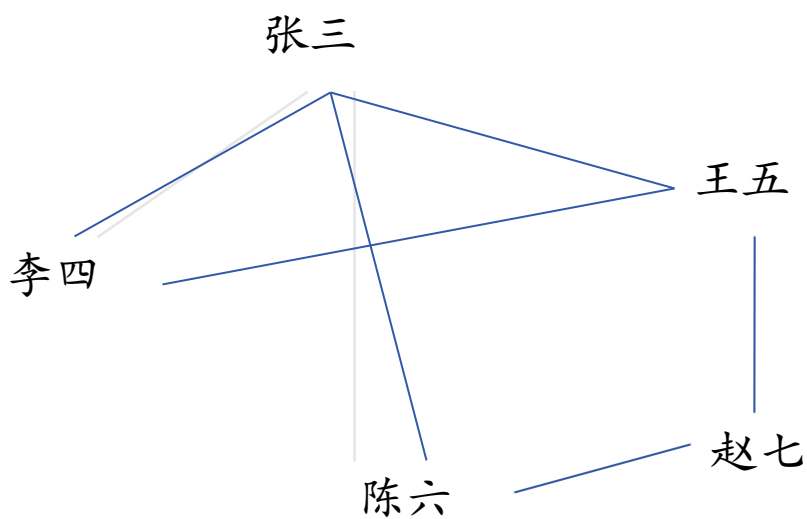
Memory View (Address: 0x00007ff7b2ddc480):

Address	Hex	ASCII
0x00007ff7b2ddc420	00 00 00 00	
0x00007ff7b2ddc430	00 00 00 00	
0x00007ff7b2ddc440	10 c4 dd b2	
0x00007ff7b2ddc450	00 00 00 00	
0x00007ff7b2ddc460	a0 c6 dd b2	
0x00007ff7b2ddc470	90 c4 dd b2	
0x00007ff7b2ddc480	44 33 22 11	apple
0x00007ff7b2ddc490	20 c7 dd b2	

C++ IDE: Clion

<https://www.jetbrains.com/clion/>





存储社会关系图的示例

存储地址

0x2040

张三

0x3118

0x3060

0x1596

...

0x3060

王五

.....

0x3118

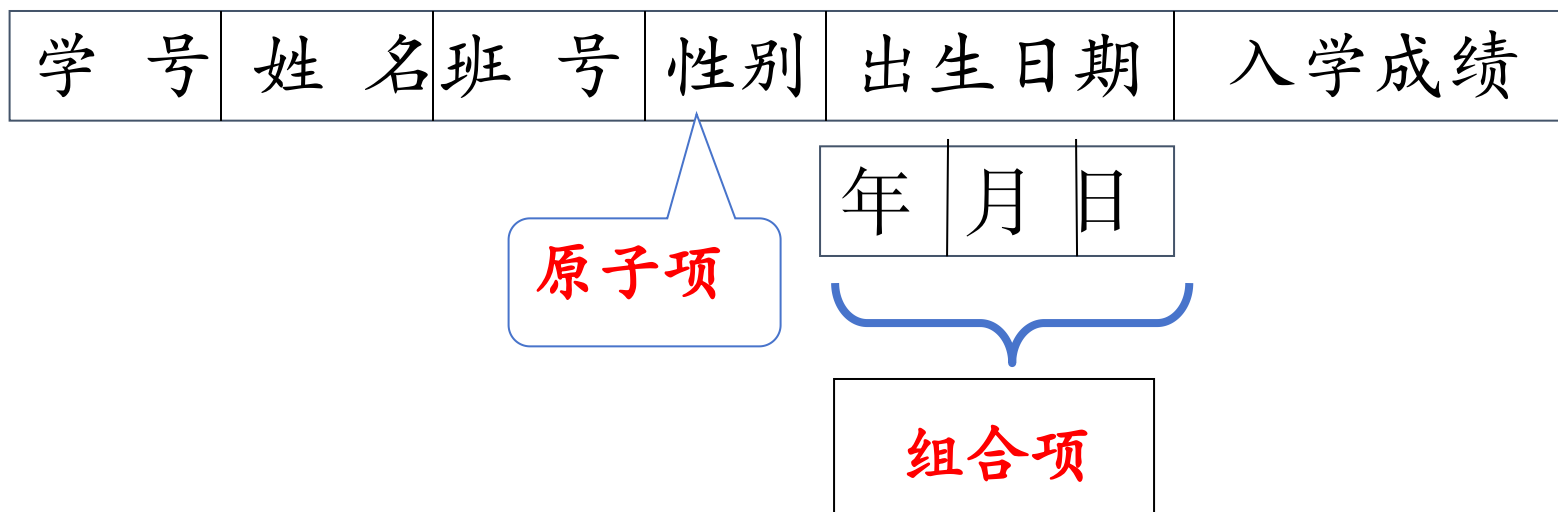
李四

... ..

张三的社会  
关系 (好友  
的存储地址)

## 1.2 基本概念和术语

- **数据 (data)**: 信息的载体, 被计算机识别、存储加工处理的对象
- **数据元素 (data element)**: 数据的基本单位, 是数据集合中的个体。一个数据元素可由若干个**数据项**组成。
- **数据项(data item)**: 具有独立含义的标识单位, 是数据不可分割的最小单位。



**数据对象(data object)**是**性质相同的数据元素**的**集合**，是数据的一个子集。

数据对象可以是有限的，也可以是无限制的。

例：整数的数据对象是 $\{\dots-3, -2, -1, 0, 1, 2, 3, \dots\}$

英文字符类型的数据对象是 $\{A, B, C, D, E, F, \dots\}$

学生数据对象

学号 姓名 班号 性别 出生日期 入学成绩

001	刘影	01	女	20040417	623
002	李恒	01	男	20031211	679
003	陈诚	02	男	20040910	638
...	...	...	...	...	...

数据元素

## 1.2 基本概念和术语

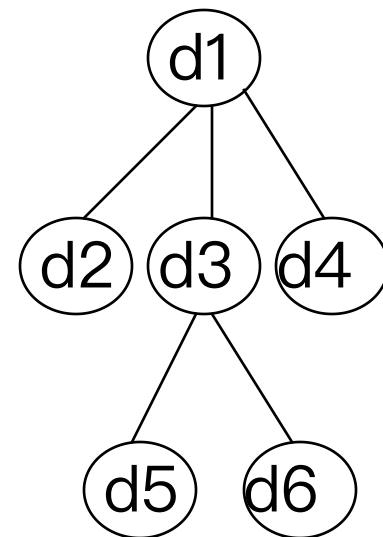
- **数据结构 (data structure)** : 相互之间存在着一种或多种**特定关系的数据元素的集合**。
- **结构 (structure)** : 数据元素之间的关系, 可以看做是从具体问题抽象出来的数学模型, 与计算机无关, 与数据的存储无关, 也叫做**逻辑结构**。

逻辑结构可用一个二元组表示：

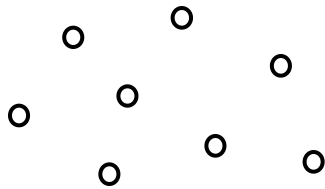
$$\text{Data\_Structure} = (D, S)$$

D—数据元素的有穷集合， S—D上关系的有穷集合。

**【例】** 设有数据结构  $B = (D, S)$  ,  
其中  $D = \{d1, d2, d3, d4, d5, d6\}$ ,  
 $S = \{ \langle d1, d2 \rangle, \langle d1, d3 \rangle,$   
 $\quad \langle d1, d4 \rangle, \langle d3, d5 \rangle,$   
 $\quad \langle d3, d6 \rangle \}$ ,  
试画出其逻辑结构图。



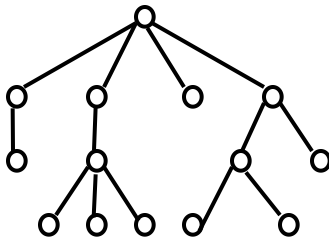
根据数据元素间关系的不同特性，数据的逻辑结构  
常分为以下四类：



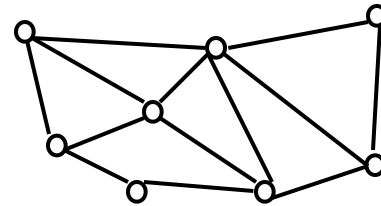
集合结构  
(数据元素同属于一个集合)



线性结构  
(数据元素之间存在一对一关系)



树形结构  
(数据元素之间存在一对多关系)



图形结构  
(数据元素之间存在多对多关系)

**存储结构**（也叫**物理结构**）：指数据结构在计算机中的映象表示。

## 数据元素的映象

每个数据元素的映象称为**结点** (node)

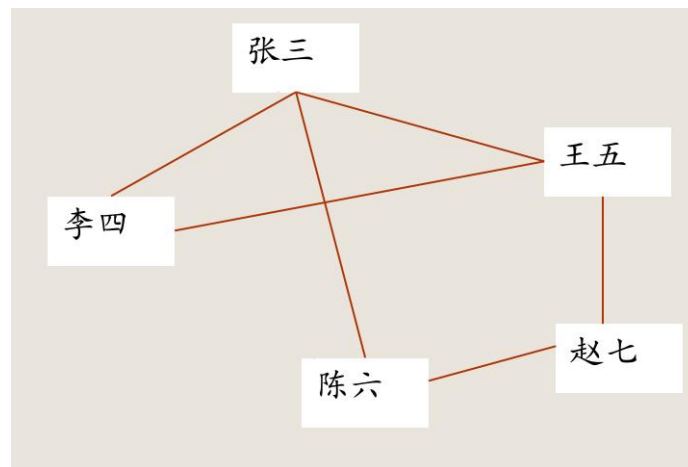
每个数据项的映象称为**数据域** (data field)

## 关系的映象

最常用的两种基本方法。

**顺序**：以相对的存储位置表示关系，(逻辑相邻<-->物理相邻)

**链式**：以附加信息(指针)表示关系



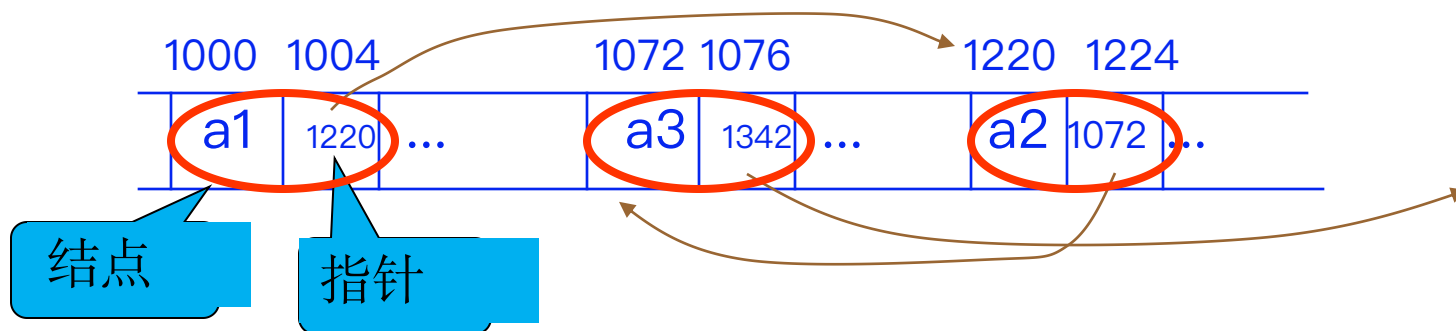


## 数据存储方式的两种常用结构

(1) **顺序存储**: 数据元素依次放在连续的存储单元中。



(2) **链式存储**: 在存储结点中增加若干指针域, 记录后继或者相关结点的地址 (指针)。



**运算（操作）**：在数据逻辑结构上定义的一组数据被使用的方式，其具体实现要在存储结构上进行。

几种常用的运算有：

- |           |           |
|-----------|-----------|
| (1)建立数据结构 | (6)检索*    |
| (2)清除数据结构 | (7)更新     |
| (3)插入数据元素 | (8)判空和判满* |
| (4)删除数据元素 | (9)求长*    |
| (5)排序     |           |

\*操作为**引用型操作**，即数据值不发生变化；  
其它为**加工型操作**。

## 1.2 基本概念和术语

面向用户，概念层

方面 层次	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	存储结构	算法
评价	不同数据结构的比较及算法分析	

面向计算机，实现层

## 1.2 基本概念和术语

- **数据类型** (data type): 最早在程序设计语言中表示变量所具有的数据种类, 是一个**值的集合**和定义在这个值集上的**操作**的总称。
- 例1、在C语言中
  - 数据类型: 基本类型和构造类型
  - 基本类型: 整型、浮点型、字符型
  - 构造类型: 数组、结构、联合、指针、枚举型、自定义
- 例2: 在python语言中除布尔、整数、浮点外还有**字符串、列表、元组、集合、字典**等类型

# 1.2 基本概念和术语

**抽象数据类型 ADT** ( Abstract Data Type ) :

数据类型概念的引伸。指一个**数学模型**以及在其上定义的**操作集合**，与计算机无关。实际上就是对该**数据结构的定义**。它定义了一个数据的**逻辑结构**以及在此结构上的一组**算法**。

三元组表示：  $(D, S, P)$

- D: 数据对象
- S: D上的关系集
- P: D的基本操作集

# 1.2 基本概念和术语

## 抽象数据类型的描述方法

**ADT 抽象数据类型名 {**

**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**

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〉

# 1.2 基本概念和术语

## 基本操作的描述方法

例：将线性表L的第i个元素的值赋给e

GetElem(L,i,&e)

### 参数表

- **赋值参数** 只为操作提供输入值。
- **引用参数** 除可提供输入值外，还将返回该参数值在操作后的变化结果

**初始条件：** 描述了操作执行之前数据结构和参数应满足的条件，若不满足，则操作失败，并返回相应出错信息。

**操作结果：** 说明了操作正常完成之后，数据结构的变化状况和应返回的结果。



# 1.3 算法和算法分析

**算法**：是对特定问题求解步骤的一种描述，是指令的有限序列，其中每一条指令表示一个或多个操作。

算法具有以下**五个特性**：

- (1) **有穷性** 一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。
- (2) **确定性** 算法中每一条指令必须有确切的含义。不存在二义性。相同的输入必然有相同的输出。
- (3) **可行性** 一个算法是可行的。即算法描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

# 1.3 算法和算法分析

算法具有以下五个特性（续）：

(4) **输入** 一个算法有零个或多个输入，这些输入取自于某个特定的对象集合。

(5) **输出** 一个算法有一个或多个输出，这些输出是同输入有着某些特定关系的量。

# 1.3 算法和算法分析

算法与程序：十分相似，又有区别。

程序不一定满足有穷性，例如，操作系统启动后进入事件处理循环。

程序中的指令必须是机器可执行的，而算法无此限制。

算法代表了对问题的解，程序是算法在计算机上特定的实现。

算法可以使用不同的方法来描述，如利于理解的自然语言、流程图、N-S图等。为了解决理解和执行之间的矛盾，常使用伪码语言描述。伪码语言忽略程序设计语言中一些严格的语法规则和描述细节，比程序设计语言更易描述和理解。

# 1.3 算法和算法分析

评价一个好的算法有以下几个标准：

- (1) **正确性(Correctness)** 算法应满足具体问题的需求。
- (2) **可读性(Readability)** 算法应该好读。以有利于阅读者对程序的理解。
- (3) **健壮性(Robustness)** 算法应具有容错处理。当输入非法数据时，算法应对其作出反应，而不是产生莫名其妙的输出结果。
- (4) **效率与存储量需求** 效率指的是算法执行的时间；存储量需求指算法执行过程中所需要的最大存储空间。一般，这两者与问题的规模有关。
- (5) 支持分布式和并行处理的算法在大数据场景下更有优势。

# 1.3 算法和算法分析

求解 $1+2+3+\dots+100$

# 1.3 算法和算法分析

求解 $1+2+3+\dots+100$

```
int i, sum =0, n=100;  
for(i=1;i<=n;i++)  
{  
    sum=sum+i;  
}
```

算法1

```
int i, sum=0, n=100;  
sum =(1+n)*n/2;
```

算法2

# 1.3 算法和算法分析

## 算法的性能分析与度量

一般从**算法的计算时间**与**所需存储空间**来评价一个算法的优劣

其方法通常有两种：

**事后统计**：计算机内部进行执行时间和实际占用空间的统计。

问题：必须先运行依据算法编制的程序；依赖软硬件环境，容易掩盖算法本身的优劣；一般没有实际价值。

**事前分析**：求出该算法的一个时间/空间界限函数。



# 1.3 算法和算法分析

与计算时间相关的因素有：

- 算法选用何种策略；
- 问题的规模；
- 程序设计的语言；（一般情况下，语言的级别越低效率越高）
- 编译程序所产生的机器代码的质量；
- 机器执行指令的速度；

撇开软硬件等有关因素，可以认为一个特定算法“运行工作量”的大小，只依赖于问题的规模（通常用 $n$ 表示），或者说，它是问题规模的函数。

# 1.3 算法和算法分析

时间复杂度:

- 一个算法执行所耗费的时间是所有语句执行时间之和, 用  $T(n)$  表示

$$T(n) = \sum_{i \in \{\text{语句集合}\}} t_i * c_i = \sum_{i \in \{\text{语句集合}\}} c_i$$

时间频度:

- 一个算法中的原操作执行次数称为语句频度或时间频度, 用  $c$  表示;

# 1.3 算法和算法分析

算法1:

```
int i, sum =0, n=100;      //执行1次
for(i=1;i<=n;i++)          //执行n+1次
{
    sum=sum+i;              //执行n次
}
```

总共执行?       $2n+2$ 次

# 1.3 算法和算法分析

算法2:

```
int i, sum=0, n=100; //执行1次
```

```
sum =(1+n)*n/2; //执行1次
```

总共执行2次

## 1.3 算法和算法分析

算法3:

```
int i, j, sum =0, n=100; //执行1次
for(i=1;i<=n;i++)        //执行n+1次
{
    for(j=1;j<=n;j++)    // 执行(n+1)*n次
    {
        sum=sum+j;      //执行n*n次
    }
}
```

总:  $1+n+1+(n+1)*n+n*n=2n^2+2n+2$

# 1.3 算法和算法分析

- 通常一个算法不一定会在所有情况下都优于或者次于另一个算法。

次数	$4n+8$	$n$	$2n^2+1$	$n^2$
$n=1$	12	1	3	1
$n=2$	16	2	9	4
$n=3$	20	3	19	9
$n=10$	48	10	201	100
$n=100$	408	100	20001	10000

$12 > 3$

$408 < 20001$

# 1.3 算法和算法分析

- 通常一个算法不一定会在所有情况下都优于或者次于另一个算法。

次数	$2n^2$	$3n+1$	$2n^2+3n+1$
$n=1$	2	4	6
$n=2$	8	7	15
$n=5$	50	16	66
$n=10$	200	31	231
$n=100$	20000	301	20301

n越大, 差异越小



# 1.3 算法和算法分析

## 2、时间复杂度

- 若有某个辅助函数  $f(n)$ , 使得当  $n$  趋近于无穷大时,  $T(n)/f(n)$  的极限值为不等于零的常数, 则称  $f(n)$  是  $T(n)$  的同数量级函数 (记作  $T(n) = O(f(n))$ ), 称  $O(f(n))$  为算法的渐近时间复杂度, 简称时间复杂度
- 例如, 若  $T(n) = n(n+1)/2$ , 则有  $1/2 \leq T(n)/n^2 \leq 1$ , 故它的时间复杂度为  $O(n^2)$ , 即  $T(n)$  与  $n^2$  数量级相同
- 一般  $T(n)$  增长最慢的算法为最优算法

## 1.3 算法和算法分析

例1 分析以下程序段的时间复杂度

```
for (i=1;i<n;i++) {  
    y=y+1;           //n-1  
    for (j=0; j<=(2*n); j++)  
        x++;         //(n-1)(2n+1)=2n2-n-1  
}
```

该程序段的时间复杂度:

$$T(n) = 2n^2 - n - 1 = O(n^2)$$

## 1.3 算法和算法分析

例2、 for(i=1, i<=n;++i){

for(j=1;j<=n;++j){

c[i][j]=0;

for(k=1;k<=n;++k){

c[i][j]+=a[i][k]\*b[k][j];

}

}

}

由于是一个三重循环，每个循环从1到n，

则总次数为： $n \times n \times n = n^3$

时间复杂度为 $T(n) = O(n^3)$

## 1.3 算法和算法分析

例3、  $\{++x; s=0;\}$

将 $x$ 自增看成是基本操作，则语句频度为 1，即时间复杂度为  $O(1)$

如果将 $s=0$ 也看成是基本操作，则语句频度为 2，其时间复杂度仍为  $O(1)$ ，即常量阶。

## 1.3 算法和算法分析

例4、for(i=1;i<=n;++i) {

    ++X;S+=X;

}

语句频度为：n 其时间复杂度为：O(n)

即时间复杂度为线性阶。

## 1.3 算法和算法分析

例5、for(i=1;i<=n;++i)

for(j=1;j<=n;++j)

{++x;s+=x;}

语句频度为：  $n^2$

其时间复杂度为：  $O(n^2)$

即时间复杂度为平方阶。

# 1.3 算法和算法分析

- 以下六种计算算法时间的多项式是最常用的。其关系为：

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$$

- 指数时间的关系为：

$$O(2^n) < O(n!) < O(n^n)$$

- 当 $n$ 取得很大时，指数时间算法和多项式时间算法在所需时间上非常悬殊。因此，只要有人能将现有指数时间算法中的任何一个算法化简为多项式时间算法，那就取得了一个伟大的成就。

# 最大子序列和问题

- 问题描述

求取数组中**最大连续子序列和**，例如给定数组为 $A=\{1, 3, -2, 4, -5\}$ ，则最大连续子序列和为6，即 $1+3+(-2)+4=6$ 。（为简化起见，假设最大连续子序列和不少于0）

算法		1	2	3	4
时间		$O(N^3)$	$O(N^2)$	$O(N\log N)$	$O(N)$
规模	$N=10$	0.0010	0.00045	0.00066	0.00034
	$N=100$	0.4702	0.01112	0.00486	0.00063
	$N=1000$	448.77	1.1233	0.05843	0.00333
	$N=10000$	NA	111.13	0.68631	0.03042
	$N=100000$	NA	NA	8.0113	0.29832

几种算法的运行时间（秒）



# 算法1： 穷举法

```
int MaxSubSequenceSum(const int A[], int N){  
    int ThisSum, MaxSum, i, j, k;  
  
    MaxSum = 0;  
    for(i=0; i < N; i++){    //i为子序列起点，遍历所有的N个可能  
        for(j=i; j < N; j++){ //j为子序列终点，遍历所有的i~N个可能  
            ThisSum = 0;  
            for(k=i; k <= j; k++){ //计算从i到j的子序列和  
                ThisSum += A[k];  
                if(ThisSum > MaxSum){  
                    MaxSum = ThisSum;  
                }  
            }  
        }  
    }  
    return MaxSum;  
}
```

# 算法2： 穷举法--改进

```
int MaxSubSequenceSum(const int A[], int N){  
    int ThisSum, MaxSum, i, j, k;  
  
    MaxSum = 0;  
    for(i=0; i < N; i++){ //以i为子序列起点，遍历所有的N个可能  
        ThisSum = 0;  
        for(j=i; j < N; j++){ //j为子序列终点，遍历所有的i~N个可能  
            ThisSum += A[j]; //计算从i到j的子序列和时利用已经计算  
                             //出的从i到j-1的子序列和  
            if(ThisSum > MaxSum){  
                MaxSum = ThisSum;  
            }  
        }  
    }  
    return MaxSum;  
}
```

例如：A={1, 3, -2, 4, -5}

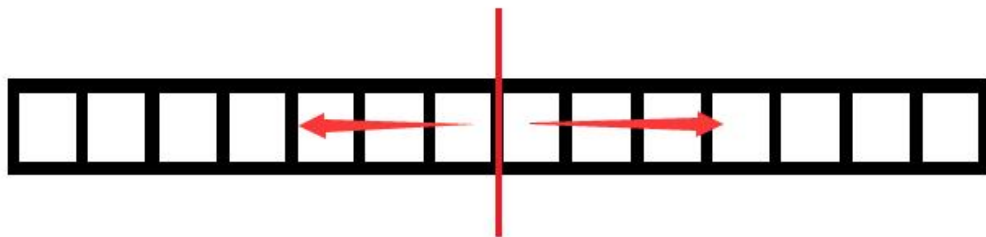
计算1、3、-2三个元素和时，利用已经计算出的1、3两个元素的和。

# 算法3：递归-分治法

可以把整个序列平均分成左右两部分，答案则会在以下三种情况中：

- 1、所求序列完全包含在左半部分的序列中。
- 2、所求序列完全包含在右半部分的序列中。
- 3、所求序列刚好横跨分割点，即左右序列各占一部分。

以分割点为起点向左的**最大连续序列和**+以分割点为起点向右的**最大连续序列和**



最大子序列和为以上三种情况取**max**

例如：{1, 3, | -2, 4}

左半部分最大和：4

右半部分最大和：4

跨分割点： $(3+1)+(-2+4)=6$

则最大子序列和为

$\max\{4,4,6\}=6$

(为简化起见，假设最大连续子序列和不少于0)

## 算法4：完美的联机算法

解析：

假设 $a[i]$ 为负数，则 $a[i]$ 不可能为此子序列的起始，同理，若 $a[i]$ 到 $a[j]$ 的子序列为负，则 $a[i]$ 到 $a[j]$ 不可能为子序列的起始，则可以从 $a[j+1]$ 开始推进。

```
int MaxSubSequenceSum(const int A[ ], int N){  
    int ThisSum, MaxSum, j;  
  
    ThisSum = MaxSum = 0;  
    for (j= 0; j < N; j++)  
    {  
        ThisSum += A[j];  
  
        if (ThisSum > MaxSum)  
        {  
            MaxSum = ThisSum;  
        }  
        else if(ThisSum < 0){  
            ThisSum = 0;  
        }  
    }  
  
    return MaxSum;  
}
```

- 有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。例如冒泡排序：

初始状态：

3	6	4	2	11	10	5
---	---	---	---	----	----	---

第1趟排序：

3	4	2	6	10	5	11
---	---	---	---	----	---	----

 (比较6次，11沉到未排序序列尾部)

第2趟排序：

3	2	4	6	5	10	11
---	---	---	---	---	----	----

 (比较5次，10沉到未排序序列尾部)

第3趟排序：

2	3	4	5	6	10	11
---	---	---	---	---	----	----

 (比较4次，6沉到未排序序列尾部)

第4趟排序：

2	3	4	5	6	10	11
---	---	---	---	---	----	----

 (比较3次，5沉到未排序序列尾部)

第5趟排序：

2	3	4	5	6	10	11
---	---	---	---	---	----	----

 (比较2次，4沉到未排序序列尾部)

第6趟排序：

2	3	4	5	6	10	11
---	---	---	---	---	----	----

 (比较1次，3沉到未排序序列尾部)

- 有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。例如：

```
void bubble-sort(int a[], int n)
{
    for(i=0,change=true; i<n-1 && change; i++)
    {
        change=false;
        for(j=0; j<n-i-1; j++){
            if (a[j]>a[j+1]) {
                a[j]  $\longleftrightarrow$  a[j+1];
                change=true }
        }
    }
}
```

最好情况：2, 3, 4, 5, 6, 10, 11

只需一趟操作，比较 $n-1$ 次，交换：0次

最好时间复杂度是 $O(n)$

最坏情况：11,10,6,5,4,3,2,1

需要进行 $n-1$ 趟排序，比较次数： $n(n-1)/2$ , 交换次数： $3n(n-1)/2$

最坏时间复杂度是  $O(n^2)$

平均情况：N个元素组成的输入集可能有 $n!$ 中排列情况，若各种情况等概率，则冒泡排序的平均时间复杂度： $O(n^2)$

- 算法的存储空间需求
- 空间复杂度:算法所需存储空间的度量, 记作:
- $S(n)=O(f(n))$
- 其中n为问题的规模(或大小)



## 算法的存储空间

- 输入数据所占空间
- 程序本身所占空间
- 辅助变量所占空间

可分为：

- 固定部分：程序代码，常量，简单变量，定长的结构变量；
- 可变部分：与问题规模有关的存储空间
- 当问题规模较大时，可变部分可能会远大于固定部分，所以一般讨论算法的渐近空间复杂度，分析方法和时间复杂度类似。

# 1.4 C语言相关知识

在本课程中，数据的存储结构是用C语言的数据类型描述（定义）的，主要用到下列数据类型：

- 数组
- 指针
- 结构
- 结构指针

# 1.4 C语言相关知识

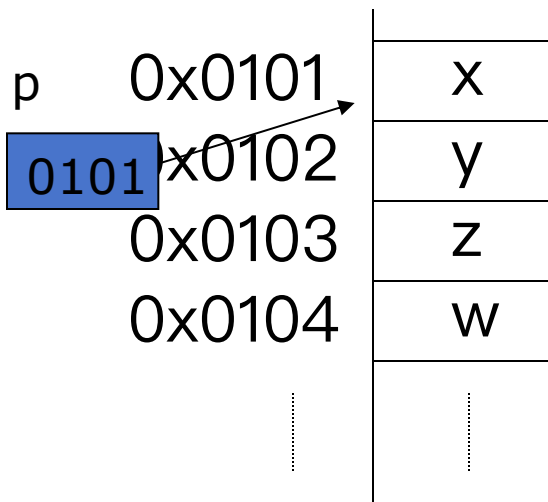
## 1 数组

```
char  a[100];
```

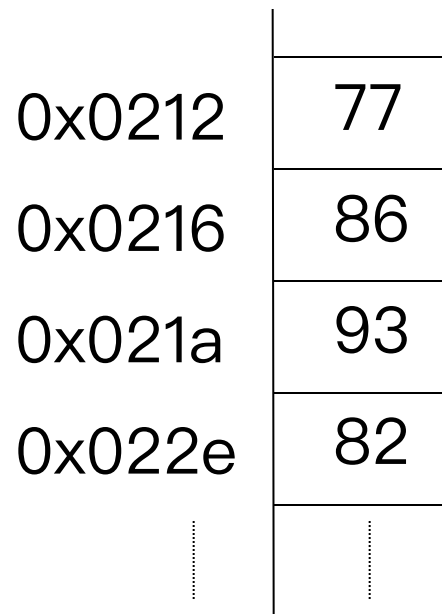
```
int    b[100];
```

数组a中每个元素占用一个字节空间，数组b中每个元素占用四个字节空间。

# 数组



```
char * p; char ch;  
p = 0x0101;  
ch = *p;  
p++;  
ch = *p;
```



```
int * p; int num;  
p = 0x0212;  
num = *p;  
p++;  
num = *p;
```

# 结构定义的一般形式

```
struct 结构名 {
```

```
    类型    变量名;
```

```
    类型    变量名;
```

```
    .....  
};
```

结构类型变量（结构变量）由一组类型可以不同的数据元素组成

```
struct 结构名 结构变量;
```

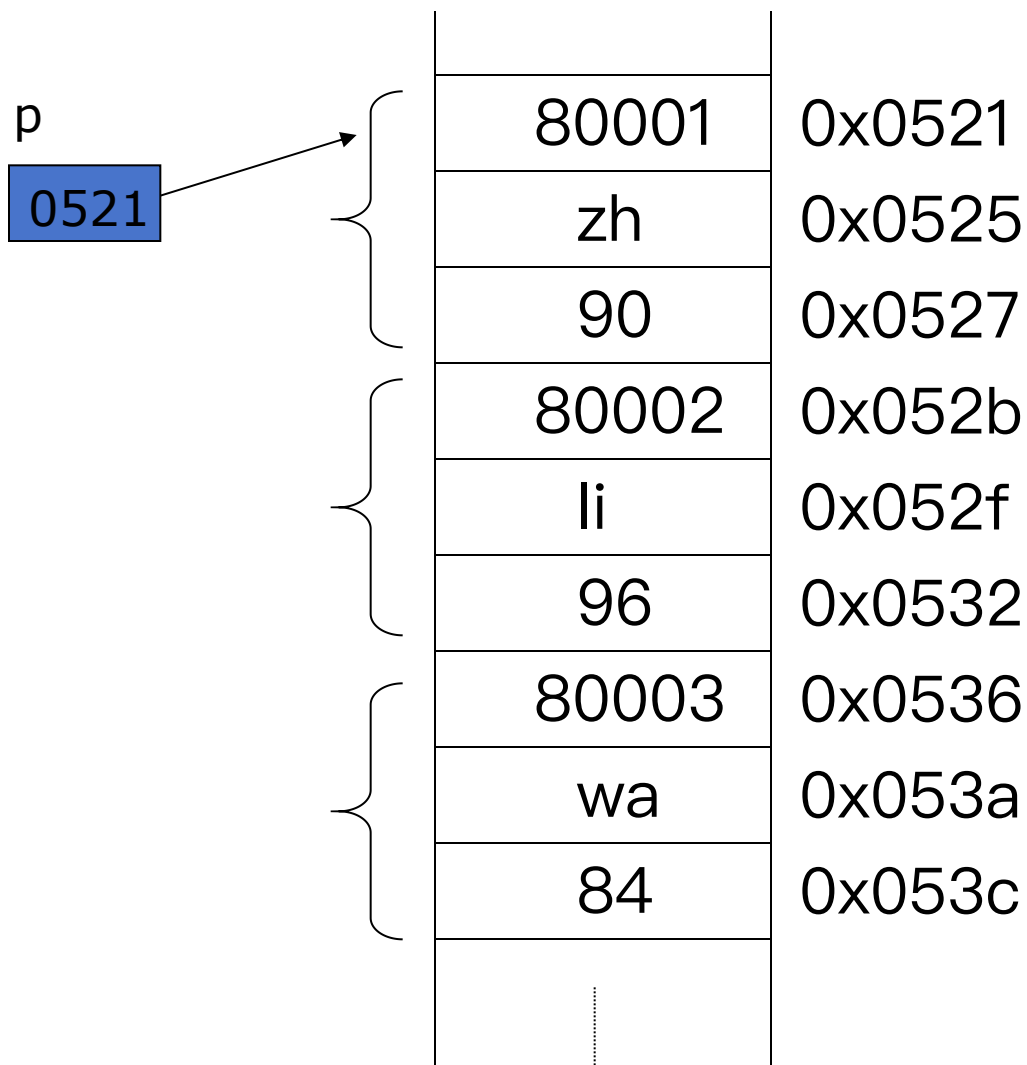
# 结构

{	80001
	zh
	90
{	80002
	li
	96
{	80003
	wa
	84
	⋮

```
struct record {  
    int    number;  
    char   name[2];  
    int    score;  
}
```

```
struct record people[100];
```

# 结构指针



```
struct record {  
    int    number;  
    char  name[2];  
    int    score;  
}
```

```
struct record  
* p;
```

```
p = 0x0521;  
x = (*p).score;  
p++;  
x = (*p).score;
```

```
y = p->score;
```

```
p = 0x0536;  
int i;  
i=*((int *)p)
```

# typedef (定义别名) :

```
typedef float real;
```

区分类型定义和变量定义

```
typedef struct xxx {  
    char  number;  
    char  name;  
    char  score;  
} study;
```

```
struct xxx a;  
study b;
```

```
study people[100];  
study *p;
```



# 结构类型定义

*typedef 结构定义 结构类型名;*

*结构类型名 结构变量名;*

例 一本书可以用有2个成员（数据域）的结构变量存储。

*typedef struct {*

*int no;*

*char title[40];*

*}BookType;*

*BookType book1;*

# 结构指针类型

指针类型变量（指针变量）用于存储变量地址（或称指向该变量）  
(只介绍本课程用到的指向结构变量指针类型)

*typedef 结构定义 \*指针的类型名;*

*指针的类型名 指针变量名*

- 例

```
typedef struct {  
    int no;  
    char title;  
}*BookPtr;  
BookPtr pbook;
```

# 数组类型

数组的**类型定义**和**变量定义**

*typedef 数组元素类型名 数组类型名[常量表达式];*

*typedef int Scores[30];*

数组类型变量（数组变量）由一组类型相同的数据元素组成

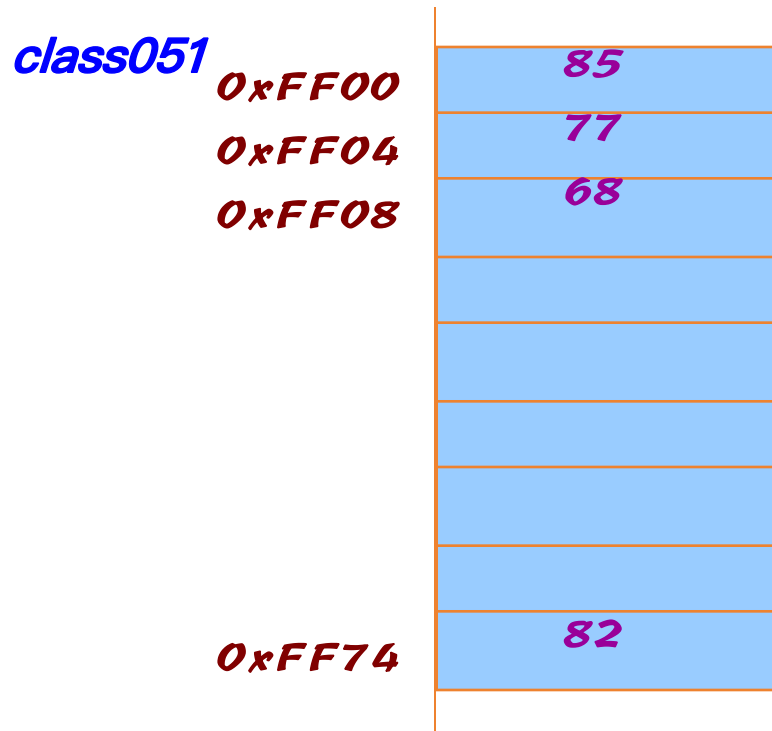
*数组类型名 数组变量名;*

例 某班30个学生的数学成绩，可以用有30个分量的整型数组变量存储。

*typedef int Scores[30];*

*Scores class051;*

- 该数组在内存中的存储示意图



# 参数传递

- 值传递
- 指针/地址传递
- 引用传递

# 值传递

```
1. void Exchg1(int x, int y)
2. {
3.     int tmp;
4.     tmp = x;
5.     x = y;
6.     y = tmp;
7.     printf("x = %d, y = %d\n", x, y);
8. }
9. main()
10. {
11.     int a = 4, b = 6;
12.     Exchg1(a, b);
13.     printf("a = %d, b = %d\n", a, b);
14.     return(0);
15. }
```

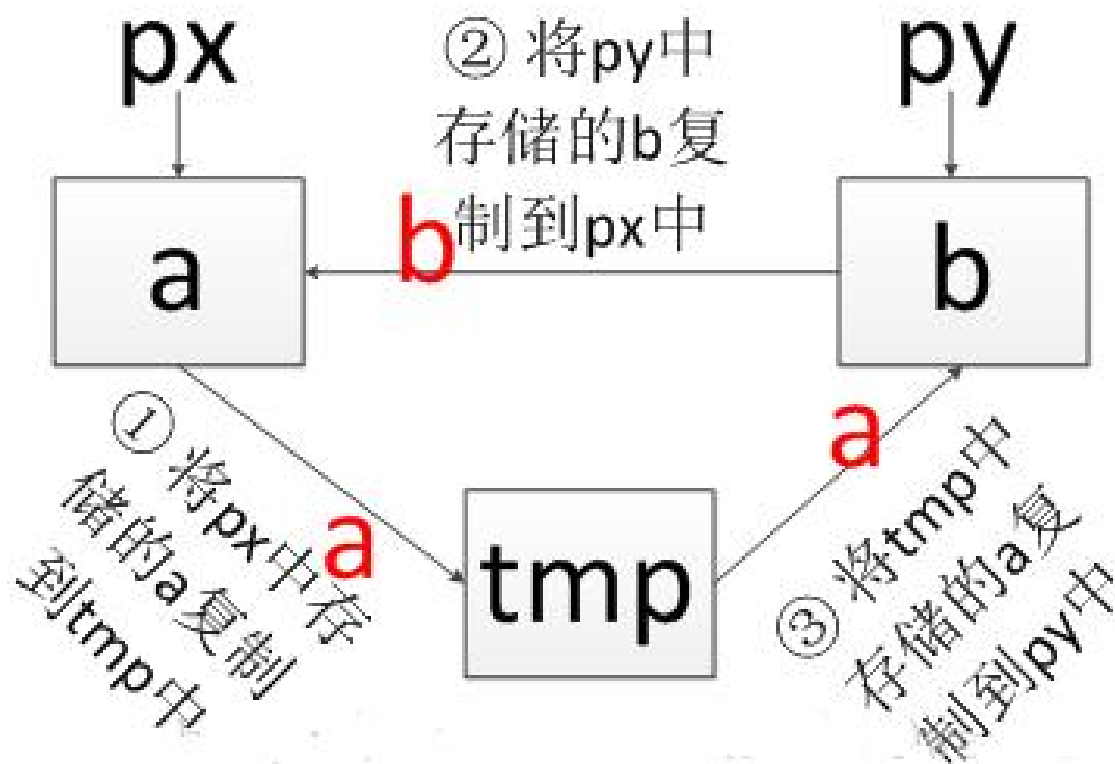
函数在调用时是隐含地把参数a,b的值分别赋值给了x,y。之后在函数体内一直是**对形参x,y**进行操作。并没有对a,b进行任何操作。

# 指针/地址传递

```
1. void Exchg2(int *px, int *py)
2. {
3.     int tmp = *px;
4.     *px = *py;
5.     *py = tmp;
6.     printf("*px = %d, *py = %d.\n", *px, *py);
7. }
8. main()
9. {
10.    int a = 4, b = 6;
11.    Exchg2(&a, &b);
12.    printf("a = %d, b = %d.\n", a, b);
13.    return(0);
14. }
```

函数在调用时是隐含地把参数a,b的地址分别传递给了指针px,py。之后在函数体内一直是对指针px,py进行操作。也就是对a,b的地址进行的操作。

# 指针/地址传递





# 引用传递

```
1. void Exchg3(int &x, int &y)
2. {
3.     int tmp = x;
4.     x = y;
5.     y = tmp;
6.     printf("x= %d,y = %d\n", x, y);
7. }
8. main()
9. {
10.    int a = 4,b =6;
11.    Exchg3(a, b);
12.    printf("a= %d, b = %d\n", a, b);
13.    return(0);
14. }
```

x、y前都有一个“&”。有了这个，调用Exchg3时函数会将a、b 分别代替了x、y了，我们称：**x、y分别引用了a、b变量**。这样函数里操作的其实就是实参a、b本身了，因此函数的值可在函数里被修改

# C++ 命名空间

<https://www.runoob.com/cplusplus/cpp-namespaces.html>

随着项目规模的扩大，可能在不同的库中可能存在名字相同的函数、类、变量等。

为了解决这一问题，C++引入了命名空间（namespace）的概念，它可作为附加信息来区分不同库中相同名称的函数、类、变量等。

```
#include <iostream>
using namespace std;

// 第一个命名空间
namespace first_space{
    void func(){
        cout << "Inside first_space" << endl;
    }
}

// 第二个命名空间
namespace second_space{
    void func(){
        cout << "Inside second_space" << endl;
    }
}

int main ()
{

    // 调用第一个命名空间中的函数
    first_space::func();

    // 调用第二个命名空间中的函数
    second_space::func();

    return 0;
}
```

# C++ 标准输入输出流 (cin/cout)

<https://www.runoob.com/cplusplus/cpp-basic-input-output.html>

```
#include<iostream>
using namespace std;
int main(){
    int x;
    float y;
    cout<<"Please input an int number:"<<endl;
    cin>>x;
    cout<<"The int number is x= "<<x<<endl;
    cout<<"Please input a float number:"<<endl;
    cin>>y;
    cout<<"The float number is y= "<<y<<endl;
    return 0;
}
```

运行结果如下 (✓ 表示按回车键) :

Please input an int number:

8✓

The int number is x= 8

Please input a float number:

7.4✓

The float number is y= 7.4

C++ 编译器根据要输入/输出值的数据类型，选择合适的流提取/流插入运算符来提取值/输出值。

# 1.5 本章知识点小结

- 数据结构
- 数据类型
- 抽象数据类型
- 数据结构的存储结构
- 算法的概念及特性
- 算法的设计要求
- 算法描述
- 算法的时间复杂度
- 算法的空间复杂度