

计算机导论与程序设计 [CS006001]

段江涛

机电工程学院



2022 年 11 月

Outlines

- I 课程介绍, C 语言简介, 数据输入输出, 基本数据类型与表达式
- II 选择结构程序设计
- III 循环结构程序设计
- IV 数组
- V 函数
- VI 指针
- VI 结构体

Part I

课程介绍, C 语言简介, 数据输入输出, 基本数据类型与表达式

Outlines

- 1 课程介绍
- 2 导论简介
- 3 C 语言程序设计简介
- 4 开发工具
- 5 数据类型
- 6 数据的输入输出
- 7 运算符和表达式
- 8 数学库函数
- 9 顺序程序设计举例

课程内容

- 计算机导论:了解计算机的基本知识;掌握计算机操作基本技能。
- 程序设计:掌握结构化程序设计方法,训练程序逻辑思维能力。会读、会编、会调试 C 语言程序。
- 学习方法:线上、线下相结合。认真书写课堂笔记,按时完成上机练习作业,鼓励大量编程练习。
- 教材
 - 大学计算机,龚尚福,贾澎湃,西安电子科技大学出版社
 - C 程序设计第五版,谭浩强,清华大学出版社
- 线上参考课程资源链接: [online resource.pdf](#)

考核

- 1 平时成绩: 10% 由上机练习, 课堂讨论等部分组成。
- 2 导论部分: 20% 结合线上资源, 自学字处理软件。总结知识点、课堂笔记, 撰写课程学习报告。
- 3 期中考试: 30% 根据机试系统给出的题目编写程序, 通过调试得到正确结果并通过机试系统提交。
- 4 期末考试: 40% 根据机试系统给出的题目编写程序, 通过调试得到正确结果并通过机试系统提交。

计算机导论主要内容

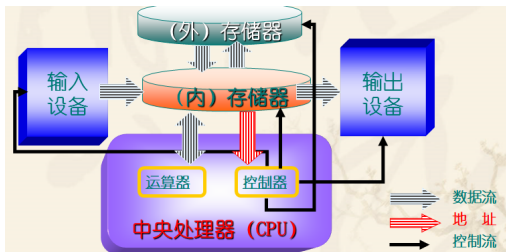
总体要求：了解计算机的基本知识；掌握计算机操作基本技能。

- 计算机系统组成
- 计算机工作原理
- 操作系统
- 字处理: Microsoft Word
- 电子表格: Microsoft Excel
- 演示文稿: Microsoft PowerPoint

计算机工作原理

工作原理：“存储程序” + “程序控制”

- 1 以二进制形式表示数据和指令
- 2 将程序存入存储器中, 由控制器自动读取并执行
- 3 外部存储器存储的程序和所需数据 \implies 计算机内存 \implies 在程序控制下由 CPU 周而复始地取出指令、分析指令、执行指令 \implies 操作完成。



可以被计算机理解并执行的基本操作命令。



一组计算机能识别和执行的指令。
一个特定的指令序列用来完成一定的功能。



与计算机系统操作有关的计算机程序、规程、规则，以及可能有的文件、文档及数据。

高级语言的发展

非结构化的语言

01

02

结构化语言

面向对象的语言

03

规定：

程序必须由具有良好特性的基本结构(顺序结构、选择结构、循环结构)构成，程序中的流程不允许随意跳转，程序总是由上而下顺序执行各个基本结构。

特点：

程序结构清晰，易于编写、阅读和维护。

C语言的特点

1 语言简洁、紧凑,使用方便、灵活

2 运算符丰富

3 数据类型丰富

4 C语言是完全模块化和结构化的语言

具有结构化的控制语句 (顺序、选择、循环结构)

用函数作为程序的模块单位,便于实现程序的模块化

5 兼具高级语言和低级语言的功能

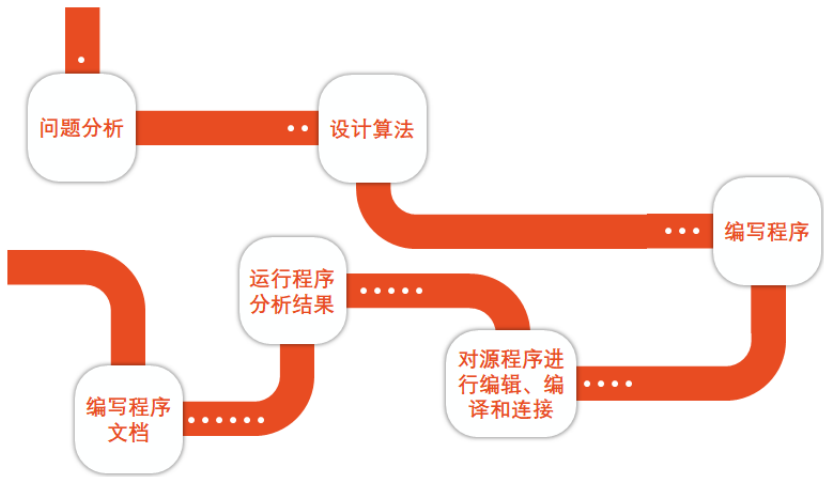
允许直接访问物理地址

能进行位 (bit) 操作

能实现汇编语言的大部分功能

可以直接对硬件进行操作

程序设计的任务



江濟

计算机导论与程序设计 [CS006001]

```
#include<stdio.h> // standard input/output编译预处理指令
int main() // 主函数
{ // 函数开始标志
    int a,b,sum; // 定义a,b,sum为整型变量
    a=123; // 对变量a赋值
    b=456; // 对变量b赋值
    sum=a+b; // 计算a+b, 并把结果存放在变量sum中
    printf("sum is %d\n",sum); // printf函数, 输出结果
    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
```


求 5! 的 C 语言程序

```

#include<stdio.h> // standard input/output编译预处理指令

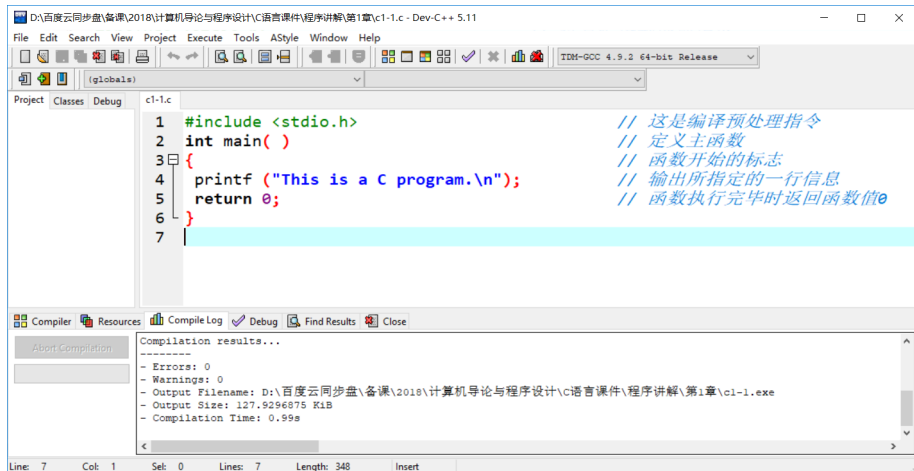
int main() // 主函数
{ // 函数开始标志

    int i,p; // p表示被乘数, i表示乘数
    p=1; // 对变量p赋值
    i=2; // 对循环变量i赋值
    while(i<=5) // 循环结构
    {
        p=p*i; // =右端p记录本语句执行前的值, 左端p记录执行后p的值。
        i++; // i = i + 1
    }
    printf("%d\n",p); // printf函数, 输出p的计算结果
    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
    
```


集成开发环境—编译系统

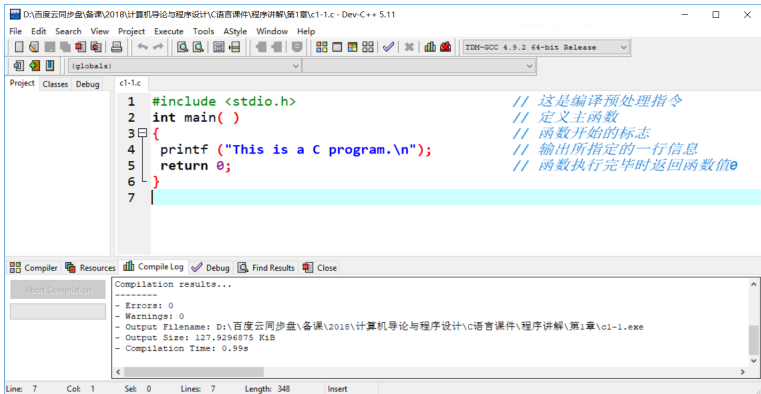
- Bloodshed Dev-C++
- Turbo C
- Visual C++6.0
- Visual Studio(VS2015, VS Community 2019 等)

Bloodshed Dev-C++ 集成开发环境



Bloodshed Dev-C++ 集成开发环境

- 选择“文件”菜单，选择“源文件”，编辑程序。
- 保存时，保存为.cpp 或.c 文件。
- 选择“编译和运行”菜单，生成.exe 文件，运行程序。



基本数据类型

- 整数`int`
- 单精度浮点数`float`
- 双精度浮点数`double`
- 字符`char`

机内用二进制表示, 不同数据类型占用存储空间大小不同。

变量在使用之前首先要定义它的数据类型

```

#include<stdio.h> // standard input/output编译预处理指令

int main() // 主函数
{ // 函数开始标志

    int a,b; // 定义变量a, b为整型数值, 同类型变量可以在一条语句中定义。
    float f; // 定义变量f为单精度浮点数
    double d; // 定义变量d为双精度浮点数
    char c; // 定义变量c为单个英文字母
    a=10; // 变量赋值
    b=20;
    f=10.2;
    d=20.3;
    c='A'; // 字符用单引号括起来
    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
    
```

标识符

标识符就是一个对象的名字。用于标识变量、符号常量、函数、数组、类型等。
以字母或下划线开始; 区分大小写; 不能使用关键字; 最好有含义。

```

#include<stdio.h>

int main()
{
    int r = 123; // 合法整型变量名
    int 3a; // 不合法的变量名
    int break; // 不合法的变量名, 因为break是关键字, 被系统使用。
    int Radius; // 变量名最好有含义
    int radius; // 与Radius是不同的变量, C语言是大小写敏感的语言
    return 0;
}
    
```


C 语言关键字

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

输出语句 printf(“原样输出,% 格式符”, 对应变量值);

```

#include<stdio.h> // standard input/output编译预处理指令

int main() // 主函数
{ // 函数开始标志

    int a=10,b; // 定义变量a, b为整型数值, 定义变量时, 可以指定变量的初值
    float f=10.2; // 定义变量f为单精度浮点数
    double d; // 定义变量d为双精度浮点数
    char c; // 定义变量c为单个英文字母

    f=10.2;
    d=20.356;
    c='A';

    printf("a=%d,b=%d,c=%c,f=%f,d=%.2lf\n",a,b,c,f,d); // %.2f, %.2lf保留
        两位小数, \n为换行符

    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
    
```

变量 b 没有被赋值, 将是一个随机值。

常用格式描述符与数据类型的对应关系

格式符	对应的数据类型	备注
%d	int	
%f	float	
%c	char	
%lf	double	
%.2f	float	保留两位小数, 四舍五入。不适用于 scanf()。
%.2lf	double	保留两位小数, 四舍五入。不适用于 scanf()。
%x(%X)	int,char	十六进制显示 (大写)
%ld	long int	

详见 p73, 表 3.6

十进制与二进制

十进制: 以 10 为底的幂展开式:

$$(123)_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0;$$

自低到高各位数 (除 10 取余至商为 0): $3 = 123 \% 10$, $2 = 123 / 10 \% 10 = 12 \% 10$,
 $1 = 123 / 10 / 10 \% 10 = 1 \% 10$

二进制: 以 2 为底的幂展开式:

$$(77)_{10} = (0100 \quad 1101)_2 = 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 \\ + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

自低到高各位数 (除 2 取余至商为 0): $1 = 77 \% 2$, $0 = 77 / 2 \% 2 = 38 \% 2$,
 $1 = 77 / 2 / 2 \% 2 = 19 \% 2$, $1 = 77 / 2 / 2 / 2 \% 2 = 9 \% 2$,
 $0 = 77 / 2 / 2 / 2 / 2 \% 2 = 4 \% 2$, $0 = 77 / 2 / 2 / 2 / 2 / 2 \% 2 = 2 \% 2, \dots$

10 进制、2 进制、16 进制的幂展开式

$$\begin{aligned}
 (D)_{10} &= D_{n-1} \times 10^{n-1} + D_{n-2} \times 10^{n-2} + \cdots + D_1 \times 10^1 + D_0 \times 10^0 \\
 &\quad + D_{-1} \times 10^{-1} + D_{-2} \times 10^{-2} + \cdots + D_{-m+1} \times 10^{-m+1} + D_{-m} \times 10^{-m}
 \end{aligned}$$

$$\begin{aligned}
 (B)_2 &= B_{n-1} \times 2^{n-1} + B_{n-2} \times 2^{n-2} + \cdots + B_1 \times 2^1 + B_0 \times 2^0 \\
 &\quad + B_{-1} \times 2^{-1} + B_{-2} \times 2^{-2} + \cdots + B_{-m+1} \times 2^{-m+1} + B_{-m} \times 2^{-m}
 \end{aligned}$$

$$\begin{aligned}
 (H)_{16} &= H_{n-1} \times 16^{n-1} + H_{n-2} \times 16^{n-2} + \cdots + H_1 \times 16^1 + H_0 \times 16^0 \\
 &\quad + H_{-1} \times 16^{-1} + H_{-2} \times 16^{-2} + \cdots + H_{-m+1} \times 16^{-m+1} + H_{-m} \times 16^{-m}
 \end{aligned}$$

进制对照表 $2^3 2^2 2^1 2^0 = 8 + 4 + 2 + 1$

十进制	二进制	十六进制	十进制	二进制	十六进制
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

十进制、二进制与十六进制数值分解举例

$$(123)_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0;$$

自低到高各位数: $3 = 123 \% 10$, $2 = 123 / 10 \% 10$, $1 = 123 / 10 / 10 \% 10$

$$\begin{aligned} (77)_{10} &= (0100 \quad 1101)_2 = 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 \\ &\quad + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \end{aligned}$$

$$(77)_{10} = (4D)_{16} = 4 \times 16^1 + 13 \times 16^0$$

数值在计算机中的表示 (以 8bit 编码为例)

- 原码: 正数的符号为 0, 负数的符号为 1, 其它位按一般的方法表示数的绝对值。

$$x = (+103)_{10} \quad [x]_{\text{原}} = (01100111)_2$$

$$x = (-103)_{10} \quad [x]_{\text{原}} = (11100111)_2$$

- 反码: 正数的反码与原码相同; 负数的反码是符号位不变, 其他位按位取反

- 补码: 正数的补码与其原码相同; 负数的补码为其反码最末位加 1. 即,

负数补码 = 反码 + 1 = 2^n - 该数的绝对值, n 是编码二进制位数.

$$(77)_{10} = (0100 \ 1101)_2, \quad (-77)_{10} = (1100 \ 1101)_2$$

$$(-77)_{\text{补}} = 2^8 - 77 = 1111 \ 1111 + 0000 \ 0001 - 0100 \ 1101$$

$$= \underbrace{1111 \ 1111 - 0100 \ 1101}_{(-77)_{\text{反}}} + 0000 \ 0001$$

$$= \underbrace{1011 \ 0010}_{(-77)_{\text{反}}} + 0000 \ 0001 = 1011 \ 0011$$



数值表示示例



机内以补码形式存储有符号数

- 1 对于正数, 原码 = 反码 = 补码
- 2 对于负数, 补码 = 反码 + 1
反码 = 符号位不变, 其他位按位取反
- 3 补码是可逆的, 即再对补码求补得到原码。
- 4 引入补码后, 使减法统一为加法。

$$(+77)_{\text{补}} + (-77)_{\text{补}} = 0100\ 1101 + 1011\ 0011 = 0000\ 0000$$

注意: 由于采用 8bit 编码, 从右到左的第 9 位的 1 被舍弃。

补码运算实例 (以 8bit 编码为例)

补码可逆:

$$[-25]_{\text{原}} = (1001\ 1001)_2 \quad [-25]_{\text{反}} = (1110\ 0110)_2$$

$$[-25]_{\text{补}} = [-25]_{\text{反}} + 1 = (1110\ 0110 + 1)_2 = (1110\ 0111)_2$$

$$[-25]_{\text{原}} = ([-25]_{\text{补}})_{\text{补}} = (1001\ 1000 + 1)_2 = (1001\ 1001)_2$$

减法统一为加法: $[a - b]_{\text{补}} = a_{\text{补}} + [-b]_{\text{补}}$

$$[102 - 25]_{\text{补}} = [77]_{\text{补}} = (0100\ 1101)_2 = 77$$

$$[102]_{\text{补}} + [-25]_{\text{补}} = (0110\ 0110)_2 + (1110\ 0111)_2 = (0100\ 1101)_2 = 77$$

$$\text{所以, } [102 - 25]_{\text{补}} = [102]_{\text{补}} + [-25]_{\text{补}}$$

$$\text{同样有, } [25 - 102]_{\text{补}} = [25]_{\text{补}} + [-102]_{\text{补}}$$

计算机数据存储单位

位 (bit) 是最小的存储单位, 每一位存储 1 位二进制码, 一个字节 (Byte) 由 8 位组成。

- $1\text{B} = 8\text{b}$
- $1\text{KB} = 1024\text{B}$
- $1\text{MB} = 1024\text{KB}$
- $1\text{GB} = 1024\text{MB}$
- $1\text{TB} = 1024\text{GB}$

整型数据输出 `printf("%d,%x", -25, -25);`

```
#include<stdio.h> // standard input/output编译预处理指令
int main()
{
    int a=-25,b=102; // 定义变量a, b为整型数值, 定义变量时, 可以指定变量的初值
    printf("a=%d,b=%d,a=%x,b=%x\n",a,b,a,b);
    printf("%d,%d,%x,%x\n",-25,102,-25,102);
    printf("%d,%d,%x,%x,%x\n", 77,-77,77,-77,-77);
    return 0;
}
```

// 编译运行, 解释输出结果。

`a=-25,b=102,a=ffffffe7,b=66`
`-25,102,ffffffe7,66`
`77,-77,4d,ffffffb3,FFFFFFB3`

$[-25]_{\text{补}} = (1110\ 0111)_2 = \text{E7H} (0\text{XE7})$

$[-77]_{\text{补}} = (1011\ 0011)_2 = \text{B3H} (0\text{XB3})$

$66\text{H} = 6 \times 16^1 + 6 \times 16^0 = 102$

$4\text{DH} = 4 \times 16^1 + 13 \times 16^0 = 77$

ASCII 编码表 $B_6B_5B_4B_3B_2B_1B_0$

$B_6B_5B_4$ $B_3B_2B_1B_0$	000	001	010	011	100	101	110	111
0000	NUL	DLE	空格	0	@	P	\	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	•	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

■ ASCII 码连续排列

‘0’~‘9’, ‘A’~‘Z’,
‘a’~‘z’

■ 数字 = 编码值 - ‘0’

9=‘9’-‘0’

■ 大小字符间隔:

‘a’ - ‘A’ = 32

‘a’=0110 0001=61H=0X61=97

‘A’=0100 0001=41H=0X41=65

字符类型

```

#include<stdio.h>

int main()
{
    char c1 = 'A', c2 = 'a', c3 = '\n'; // 字符型变量,
    // '\n': 表示特殊字符 --- 换行符
    printf("%c,%c,%d\n",c1,c2,c3); // A,a,10
    // 整型变量的整数值就是ASCII编码值
    printf("%c,0X%x,%d\n",c1,c1,c1); // A,0X41,65
    c1 = c1 + 1; // 在表达式中, char类型当作int处理
    printf("%c,0X%x,%d\n",c1,c1,c1); // B,0X42,66
    c1 = c1 + 32; // 转换为小写字母
    printf("%c,0X%x,%d\n",c1,c1,c1); // b,0X62,98
    printf("%d\n",'9'-'0'); // 数字 = 编码值- '0'
    printf("%c,%d,%c,%d\n",'A','A','a','a'); // 输出字符和相应的ASCII编码
    return 0;
}
    
```

常量

```

#include<stdio.h>

#define PI 3.14 // 符号常量，注意没有分号

int main()
{
    int a = 123; // 整型常量
    float f = 12.2, f1=123E-1; // 实型常量， $123 \times 10^{-1}$ 
    char c1 = 'A', c2='\n'; // 字符常量，特殊字符 --- 换行符
    char s[50] = "boy"; // 字符串常量
    printf("半径为%d的圆周长是%f\n", a, 2*PI*a);
    printf("回车换行\n");
    printf("单引号\ ', 双引号\"转义字符前缀\\, \\n");
    return 0;
}
    
```

转义字符(特殊字符), 见 p40, 表 3.1

常量与常变量

```

#include<stdio.h>

#define PI 3.14 // 符号常量，注意没有分号

int main()
{
    int r = 123; // 整型变量
    const int a = 425; // 常变量
    r = 100; // 合法，因为r是变量，可以随时更改它的值
    a = 100; // 不合法，因为a是常变量，不能更改
    printf("半径为%d的圆周长是%f\n", r, 2*PI*r);
    return 0;
}
    
```

长整型、无符号整型, 浮点型数据类型

```

#include<stdio.h>

int main()
{
    int a = 123; // 整型变量
    long int b = 1E+8; // 长整型变量
    unsigned int u = 0XFF; // 无符号整型, 最高位不作为符号位处理
    float f = 10.2; // 单精度浮点数
    double d = 1E-8; // 双精度浮点数
    printf("%x,%d\n",u,u); // ff, 255
    printf("%d,%ld,%x,%f,%lf\n",a,b,u,f,d);
    // sizeof函数返回类型分配的字节数, 整型数据存储空间和值的范围见p45, 表3.2
    printf("%d,%d,%d,%d\n",sizeof(int),sizeof(float),sizeof(double),
        sizeof(long int),sizeof(long long int));
    return 0;
}
    
```

Recall: 常用格式描述符与数据类型的对应关系

格式符	对应的数据类型	备注
%d	int	
%f	float	
%c	char	
%lf	double	
%.2f	float	保留两位小数, 四舍五入。不适用于 scanf()。
%.2lf	double	保留两位小数, 四舍五入。不适用于 scanf()。
%x(%X)	int,char	十六进制显示 (大写)
%ld	long int	

详见 p73, 表 3.6

Recall: 输出语句 printf(“原样输出,% 格式符”, 对应变量值);

```

#include<stdio.h> // standard input/output编译预处理指令
int main() // 主函数
{ // 函数开始标志
    int a=10,b; // 定义变量a, b为整型数值, 定义变量时, 可以指定变量的初值
    float f=10.2; // 定义变量f为单精度浮点数
    double d; // 定义变量d为双精度浮点数
    char c; // 定义变量c为单个英文字母

    f=10.2;
    d=20.356;
    c='A';
    printf("a=%d,b=%d,c=%c,f=%f,d=%.2lf\n",a,b,c,f,d); // %.2f, %.2lf保留
        两位小数
    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
    
```

变量 b 没有被赋值, 将是一个随机值。

输入语句 scanf(“% 变量格式符”, & 变量名);

```

#include<stdio.h> // standard input/output编译预处理指令

int main() // 主函数
{ // 函数开始标志

    int a=10,b; // 定义变量a, b为整型数值, 定义变量时, 可以指定变量的初值
    float f=10.2; // 定义变量f为单精度浮点数
    double d; // 定义变量d为双精度浮点数
    char c='A'; // 定义变量c为单个英文字母, 字符输入以后讲
    printf("请输入整数和浮点数, 空格隔开:\n"); // 提示语句[可选]
    scanf("%d%f", &a, &f); // 尽量简单, 不要有其它字符和'\n'
    printf("请输入两个浮点数, 空格隔开:\n"); // 提示语句[可选]
    scanf("%f%lf", &f, &d);
    printf("a=%d,b=%d,c=%c,f=%f,d=%lf\n", a, b, c, f, d); // \n为换行符
    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
    
```

字符输出函数 putchar

```

#include<stdio.h>

int main()
{
    char a = 'B', b = 'O', c = 'Y'; //定义3个字符变量并初始化
    putchar(a); //向显示器输出字符B
    putchar(b); //向显示器输出字符O
    putchar(c); //向显示器输出字符Y
    putchar ('\n'); //向显示器输出一个换行符
    return 0;
}
    
```

字符输入函数 `getchar`, 遇到回车, 开始从缓冲区中接收字符。

```
#include<stdio.h>

int main()
{
    char a,b,c; //定义字符变量a,b,c
    a = getchar(); //从键盘输入一个字符, 送给字符变量a
    b = getchar(); //从键盘输入一个字符, 送给字符变量b
    c = getchar(); //从键盘输入一个字符, 送给字符变量c
    putchar(a); //将变量a的值输出
    putchar(b); //将变量b的值输出
    putchar(c); //将变量c的值输出
    printf("\na=%d,b=%d,c=%d,a=%c,b=%c,c=%c\n",a,b,c,a,b,c);
    return 0;
}
```

字符输入函数 `getchar`, 遇到回车, 开始从缓冲区中接收字符。

```
char a,b,c; //定义字符变量a,b,c

a = getchar(); //从键盘输入一个字符, 送给字符变量a
b = getchar(); //从键盘输入一个字符, 送给字符变量b
c = getchar(); //从键盘输入一个字符, 送给字符变量c

putchar(a); //将变量a的值输出
putchar(b); //将变量b的值输出
putchar(c); //将变量c的值输出

printf("\na=%d,b=%d,c=%d,a=%c,b=%c,c=%c\n",a,b,c,a,b,c);
```

从键盘输入 abc 回车, 观察结果, 应该是正确的结果。遇到回车, 开始从缓冲区中接收字符。

开发平台上演示讲解

在开发平台, 以具体的示例, 详细讲解以下内容:

- `int`, `float`, `double`, `char` 数据类型, `sizeof()` 函数
- `%d`, `%f`, `%c`, `%lf`, `%x` 格式符的使用 (见 ppt 中的表格)
- `if()`{ }, `while()`{ } 简单语句
- `char c; scanf("%c", &c);` 接收输入的字符
`char op; int x,y; scanf("%d%c%d",&x,&op,&y);`
- `char c; c=getchar();` 接收输入的字符, `putchar(c)` 输出一个字符
- 编程理解数字 ASCII 码与整数的对应关系以及大小写字符之间的关系。
- 重点理解字符缓冲区的概念, 以及消费无用字符的技巧。

算术运算符 $+$, $-$, $*$, $/$, $\%$, $++$, $--$

整数 = 整数/整数, 结果不会四舍五入。

```
#include<stdio.h>

int main()
{
    int a=5, b=2; float c=5,d=2,f;
    f = a/b; printf("%f\n",f); // 2.000000
    f = c/d; printf("%f\n",f); // 2.500000
    f = (float)a/b; printf("%f\n",f); // 2.500000
    printf("%f\n",5.0/2); // 2.500000
    printf("%d\n",2a); // 错误
    printf("%d\n",2*a); // 正确
    return 0;
}
```

52/230

算术运算符 ++, --

++i, --i: 先加 (减)1, 再使用。**i++, i--:** 先使用, 再加 (减)1

```
#include<stdio.h>

int main()
{
    int a,b=10;
    a = ++b;
    printf("a=%d,b=%d\n",a,b); // a=11,b=11
    a = b++;
    printf("a=%d,b=%d\n",a,b); // a=11,b=12
    a--;
    b--;
    printf("a=%d,b=%d\n",a,b); // a=10,b=11
    return 0;
}
```

数学库函数, 详见附录 E p365.

```
#include<math.h>
```

函数原型	功能	调用举例
<code>int abs(int x);</code>	求整数 x 的绝对值	<code>int x=-10,y; y=abs(x);</code>
<code>double fabs(double x);</code>	求浮点数 x 的绝对值	<code>double x=-0.5,y; y=fabs(x);</code>
<code>double sqrt(double x);</code>	计算 \sqrt{x}	<code>double x=0.5,y; y=sqrt(x);</code>
<code>double pow(double x, double y);</code>	计算 x^y	<code>double x=0.5,y=2,z; z=pow(x,y);</code>
<code>int rand(void);</code>	产生-90~32767 的随机整数	<code>int y; y=rand();</code>
<code>double log(double x);</code>	求 $\log_e x$, 即 $\ln x$	<code>double x=2.0; y=log(x);</code>
<code>double log10(double x);</code>	求 $\log_{10} x$	<code>double x=2; y=log10(x);</code>

因为函数内是用二进制计算浮点数的指数值, 注意, 调用 `pow` 函数使用整型参数有精度问题, 可能得不到正确的值。以后学习函数章节后, 最好编写自己的函数计算整数的指数值。

Example (例 3.5 p64)

求 $ax^2 + bx + c = 0$ 方程的根。 a, b, c 由键盘输入, 设 $b^2 - 4ac > 0$ 。

```

#include<stdio.h>
#include<math.h> // 数学库函数
int main()
{
    double a,b,c,x1,x2,delta;
    scanf("%lf%lf%lf",&a,&b,&c);
    if(b*b-4*a*c <= 0) { printf("输入错误!"); return 0; } // 程序结束
    delta = sqrt(b*b-4*a*c);
    x1 = -b + delta/(2*a);
    x2 = -b - delta/(2*a);
    printf("x1=%lf,x2=%lf\n",x1,x2);
    return 0;
}
    
```

Example (例 3.1 p37)

有人用温度计测量出用华氏法表示的温度 (如 64°F), 今要求把它转换为以摄氏法表示的温度 (如 17.8°C)。

$$c = \frac{5}{9}(f - 32)$$

其中, f 代表华氏温度, c 代表摄氏温度.

特别注意: 整数/整数 = 整数, 不会四舍五入。

Example (例 3.2 p38)

计算存款利息。有 1000 元, 想存一年。有 3 种方法可选:

- (1) 活期, 年利率为 r_1 ;
- (2) 一年期定期, 年利率为 r_2 ;
- (3) 存两次半年定期, 年利率为 r_3 。

请分别计算出一年后按 3 种方法所得到的本息和。

$$p_1 = p_0(1 + r_1), p_2 = p_0(1 + r_2), p_3 = p_0(1 + \frac{r_3}{2})(1 + \frac{r_3}{2})$$

Example

- 1 自定义各变量类型和值, 求 $y = |x^3 + \log_{10} x|$
- 2 自定义各变量类型和值, 求 $y = \frac{3ae}{cd}$
- 3 自定义各变量类型和值, 求 $y = \frac{ax + \frac{a+x}{4a}}{2}$
- 4 m 是一个已知 3 位整数, 从左到右用 a, b, c 表示各位数字。
 - 1 求数 bac 的值;
 - 2 计算 m 的最后一个字节;
 - 3 思考, 如果 m 是多位数, 如何计算获取每位数字?

开发平台上演示讲解

- 回顾基本数据类型 `int`, `float`, `double`,

输出输入语句 `printf()`; `scanf()`; `putchar()`, `getchar()`,

格式转换符 `%d`, `%f`, `%lf`, `%c`, `%x`, ASCII 编码与整数之间的对应关系。

- 无符号整型

```
unsigned int a=0x85; int b=-5; printf("%X,%X,%d",a,b,b); //补码存储
```

- 定义常数 `#define` `PI 3.14`

- 标识符, 以字母或下划线开始。区分大小写, 不能使用关键字。

- 算术运算符 `+`, `-`, `*`, `/`, `%`, `++`, `--`。特别注意: 整数 = 整数/整数, 不会四射五入。

- `++i`, `--i`: 先加(减)1, 再使用。 `i++`, `i--`: 先使用, 再加(减)1

- 数学库函数 `int abs(int x); double fabs(double x); double sqrt(double x); double pow(double x, double y); double log10(double x)`

- 作业: 练习编程: 上页 Example, 下节课检查。

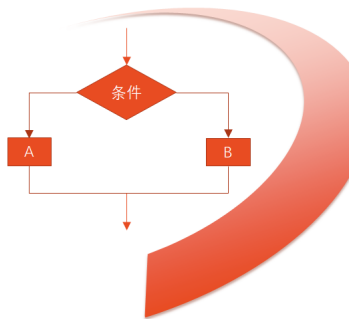
Part II

选择结构程序设计

Outlines

- 10 选择结构和条件判断
- 11 if 语句的一般形式
- 12 关系运算符及其优先次序
- 13 逻辑运算符
- 14 条件运算符和条件表达式
- 15 数学表达式与 C 语言表达式的不同
- 16 用 switch 语句实现多分支选择结构

选择结构和条件判断



C 语言有两种选择语句

- if 语句, 用来实现两个分支的选择结构
- switch 语句, 用来实现多分支的选择结构

if(条件表达式){ 表达式为真 (非 0) 时执行语句; }

```
#include<stdio.h> // standard input/output编译预处理指令

int main() // 主函数
{ // 函数开始标志

    int a=10; // 定义变量a为整型数值, 定义变量时, 可以指定变量的初值
    if(a>=10)
    {
        printf("a>=10\n"); // \n为换行符
    }
    else
    {
        printf("a<10\n"); // \n为换行符
    }

    return 0; // 函数执行完毕返回函数值0
} // 函数结束标志
```

[例 4.1 p84] 求 $ax^2 + bx + c = 0$ 方程的根。 a, b, c 由键盘输入。

```
#include<stdio.h>
#include<math.h> // 数学库函数
int main()
{
    double a,b,c,x1,x2,delta;
    scanf("%lf%lf%lf",&a,&b,&c);
    if(b*b-4*a*c < 0)
    { printf("This equation hasn't real roots!\n"); }
    else
    {
        delta = sqrt(b*b-4*a*c);
        x1 = (-b + delta)/(2*a); x2 = (-b - delta)/(2*a);
        printf("x1=%.2lf,x2=%.2lf\n",x1,x2);
    }
    return 0;
}
```

[例 4.2 p85] 输入两个实数, 按由小到大的顺序输出这两个数。

```
#include<stdio.h>

int main()
{
    float a,b,t;
    scanf("%f%f",&a,&b);
    //不好: scanf("%f,%f",&a,&b);
    // 假定a<=b, 否则交换a,b的值
    if(a>b)
    { // 将a和b的值互换
        t=a; a=b; b=t;
    }
    printf("%.2f,%.2f\n",a,b);
    return 0;
}
```

两变量值互换基本技巧

//把变量 b 的值赋给变量 a, a 的值等于 b 的值

a=b;

//再把变量 a 的值赋给变量 b, 变量 b 值没有改变

b=a;

因此, 为了实现互换, 必须借助于第三个变量。

[例 4.3 p86] 输入 3 个数 a, b, c, 要求按由小到大的顺序输出。

```
#include<stdio.h>

int main()
{
    float a,b,c,t;
    scanf("%f%f%f",&a,&b,&c); //不好: scanf("%f,%f,%f",&a,&b,&c);
    if(a>b) // 假定a<=b<=c, 否则交换。
    {
        t=a; a=b; b=t; //借助变量t, 实现变量a和变量b互换值
    } //互换后, a小于或等于b
    if(a>c)
    {
        t=a; a=c; c=t; //借助变量t, 实现变量a和变量c互换值
    } //互换后, a小于或等于c
    if(b>c) //还要
    {
        t=b; b=c; c=t; //借助变量t, 实现变量b和变量c互换值
    } //互换后, b小于或等于c
    printf("%.2f,%.2f,%.2f\n",a,b,c); //顺序输出a,b,c的值
    return 0;
}
```

if 语句的一般形式

条件表达式 \Rightarrow 关系表达式; 逻辑表达式; 数值表达式。

形式 1(无 else)

```
// 形式1(无else)
if(条件表达式)
{
    多条语句(复合语句);
}
```

形式 2

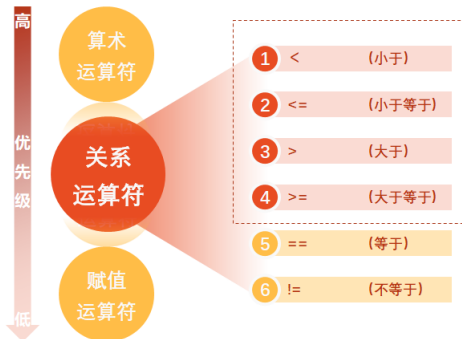
```
// 形式2
if(条件表达式)
{
    多条语句(复合语句);
}
else
{
    多条语句(复合语句);
}
```

形式 3(排除式)

```
// 形式3(排除式)
if(条件表达式1)
{
    多条语句(复合语句);
}
else if(条件表达式2) //可多个
{
    多条语句(复合语句);
}
else
{
    多条语句(复合语句);
}
```

关系运算符及其优先次序

```
int a=5,b=10,c=20; //以int为例
if(a<b+c) // 相当于a<(b+c)
{ a小于(b+c)时执行 }
if(a<=b+c)
{ a小于等于(b+c)时执行 }
if(a>b+c)
{ a大于(b+c)时执行 }
if(a>=b+c)
{ a大于等于(b+c)时执行 }
if(a==b+c) // 与a=(b+c)不同
{ a等于(b+c)时执行 }
if(a!=b+c)
{ a不等于(b+c)时执行 }
```



分析:

```
if(a>b==c) // 相当于(a>b)==c
if(a=b>c) // 相当于a=(b>c), 避免==写为=
```

关系表达式的值, 非0即真

关系表达式

- 用关系运算符将两个数值或数值表达式连接起来的式子, 称为关系表达式。
- 关系表达式的值是一个逻辑值, 即“真”或“假”。
- 在C的逻辑运算中, 以“1”表示“非0”, 代表“真”; 以“0”代表“假”。

```
int a=3, b=2, c=1, d1, d2;
d1 = a>b; // d1=(a>b), d1=1
d2 = a>b>c; // d2=(a>b>c), d2=0
if(d1) // d1=1, 非0即真
{ printf("执行此语句"); }
if(d2) // d2=0, 条件为假
{ printf("不执行此语句"); }
```

```
if(d1 = a>b) //d1=(a>b), 赋值后d1=1, 非0即真
{ printf("执行此语句"); }
if(d2 = a>b>c) //d2=(a>b>c), 赋值后d2=0, 即假
{ printf("不执行此语句"); }
if(a>b) //表达式=1, 非0即真
{ printf("执行此语句"); }
if(a>b>c), 表达式值=0, 即假
{ printf("不执行此语句"); }
```

逻辑运算符 (与 &&, 或 ||, 非!)

```
int a=5,b=10,c=0; //以int为例
```

```
if(!a) // 逻辑非(NOT), a是非0, 所以!a的值是0
```

```
{ ... }
```

```
if(a && b) // 逻辑与(AND), a,b均为非0, 所以(a && b)的值为1
```

```
{ ... }
```

```
if(a || c) // 逻辑或(OR), a,c之一是非0, 即为真
```

```
{ ... }
```

逻辑运算符真值表, 非 0 即真

a	b	!a	!b	a&&b	a b
真 (非 0)	真 (非 0)	假 (0)	假 (0)	真 (1)	真 (1)
真 (非 0)	假 (0)	假 (0)	真 (1)	假 (0)	真 (1)
假 (0)	真 (非 0)	真 (1)	假 (0)	假 (0)	真 (1)
假 (0)	假 (0)	真 (1)	真 (1)	假 (0)	假 (0)

- “&&” 和 “||” 是双目运算符, 要求有两个运算对象 (操作数);
“!” 是单目运算符, 只要有一个运算对象
- 由高到低优先次序: !(非)→&&(与)→||(或);
逻辑运算符中的 “&&” 和 “||” 低于关系运算符, “!” 高于算术运算符
- 逻辑运算结果不是 0 就是 1, 不可能是其他数值。
而运算对象可以是 0(假) 或任何非 0 的数值 (按 “真” 对待)

逻辑运算示例 (1)

判别用 year 表示的某一年是否闰年。闰年的条件是符合下面二者之一: (1) 能被 4 整除,但不能被 100 整除。(2)能被 100 整除,又能被 400 整除。

```
int year;
scanf("%d",&year);
// 闰年
if(year%4 == 0 && year%100 != 0)
{ printf("%d是闰年\n", year); }
else if(year%100 == 0 && year%400 == 0)
{ printf("%d是闰年\n", year); }
else
{ printf("%d不是闰年\n", year); }
```

逻辑运算示例 (2), 使用标志变量是基本技巧

判别用 year 表示的某一年是否闰年。闰年的条件是符合下面二者之一: (1) 能被 4 整除, 但不能被 100 整除。(2) 能被 100 整除, 又能被 400 整除。

```
int year, flag = 'N'; // flag称为标志变量, 等效设置 char flag = 'N';
scanf("%d", &year);
// 闰年
if(year%4 == 0 && year%100 != 0)
    { flag = 'Y'; }
else
{
    if(year%100 == 0 && year%400 == 0)
        { flag = 'Y'; }
}
if(flag == 'Y')
    { printf("%d是闰年\n", year); }
else
    { printf("%d不是闰年\n", year); }
```


逻辑运算示例 (3), 善用 &&, ||, ! 组成逻辑表达式, 简化程序逻辑表达

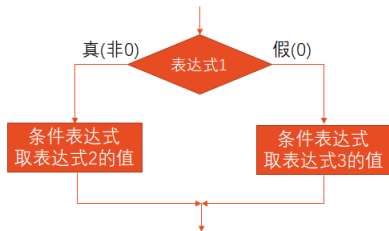
判别用 year 表示的某一年是否闰年, 可以用一个逻辑表达式来表示。闰年的条件是符合下面二者之一: (1) 能被 4 整除, 但不能被 100 整除。(2) 能被 100 整除, 又能被 400 整除。

```
int year;
scanf("%d", &year);
// 闰年
if((year%4 == 0 && year%100 != 0) || (year%100 == 0 && year%400 == 0))
{ printf("%d是闰年\n", year); }
else
{ printf("%d不是闰年\n", year); }
```

条件运算符和条件表达式

```
int a,b,max;
scanf("%d%d",&a,&b);
if(a>b)
{ max = a; }
else
{ max = b; }
// 等效于条件表达式
max = (a>b) ? a : b;
// 或
a>b ? (max=a) : (max=b);
// 甚至可用在语句中
printf("%d\n",a>b ? a : b);
```

表达式 1 ? 表达式 2 : 表达式 3



例：大写转小写字母

[例 4.4, p96] 输入一个字符, 判别它是否为大写字母, 如果是, 将它转换成小写字母; 如果不是, 不转换。然后输出最后得到的字符。

```
char ch;
scanf("%c", &ch);
ch = (ch>='A' && ch<='Z') ? (ch+32) : ch;
// 等效于
if(ch>='A' && ch<='Z')
{
    ch = ch+32; // 可简写为 ch += 32;
}
printf("ch=%c\n", ch);
```

特别注意: 数学表达式与 C 语言表达式的不同

```
int a = 100;

if(20 <= a && a <= 30) //表达式的值为假(0), 条件表达式与数学含义( $20 \leq a \leq 30$ )相同
// 或 if(a>=20 && a<=30)
{ ... }

if(20 <= a <= 30) // (20<=a)<=30, 表达式为真(1), 与数学含义( $20 \leq a \leq 30$ )不同
{ ... }

// 类似的
// if(a==20) 与 if(a=20) 意义不同
if(a==20) //表达式的值是假(0), a的值没有变化
{ ... }

if(a=20) //表达式的值是20, 非0, 表示为真, 并且a被赋值为20(赋值语句)
{
    printf("%d\n", a); // 20
}

printf("%d\n", a); // 20
```

用 switch(int 或 char 型表达式), 实现多分支选择结构

```
int a;
scanf("%d", &a)
switch(a) // a必须是int或char类型
{
    case 10: 多条语句1;
        //break表示该条件执行完毕
        break;
    case 20: 多条语句2;
        break;
    case 30: 多条语句3;
        break;
    default:
        // 以上不满足时执行
        多条语句4;
}
```

```
// 等效于
int a;
scanf("%d", &a)
if(a == 10)
    { 多条语句1; }
else if(a == 20)
    { 多条语句2; }
else if(a == 30)
    { 多条语句3; }
else
    { 多条语句4; }
```

无 break, 直接往下执行

```
char a;
scanf("%c", &a)
swach(a) // a必须是int或char类型
{
    case 'A':
    case 'a': 多条语句1;
        break;
    case 'B':
    case 'b': 多条语句2;
        break;
    case 'C':
    case 'c': 多条语句3;
        break;
    default: 多条语句4;
}
```

// 等效于

```
char a;
scanf("%d", &a)
if(a == 'A' || a == 'a')
    { 多条语句1; }
else if(a == 'B' || a == 'b')
    { 多条语句2; }
else if(a == 'C' || a == 'c')
    { 多条语句3; }
else
    { 多条语句4; }
```

[例 4.10,p99] 运输公司对用户计算运输费用。路程越远, 运费越低。标准如下:

$s < 250$	没有折扣
$250 \leq s < 500$	2% 折扣
$500 \leq s < 1000$	5% 折扣
$1000 \leq s < 2000$	8% 折扣
$2000 \leq s < 3000$	10% 折扣
$3000 \leq s$	15% 折扣

```
int c,s; //c是分类整数, s是距离
float p,w,d,f; //单价,重量,折扣,运费
// 运费 f = p*w*s*(1-d%)
scanf("%f%f%d", &p, &w, &s);
if(s>=3000) { c = 12; }
else {c = s/250; }
switch(c) {
    case 0: d=0; break;
    case 1: d=2; break;
    case 2: case 3: d=5; break;
    case 4: case 5: case 6: case 7:
        d=8; break;
    case 8: case 9: case 10: case 11:
        d=10 break;
    case 12: d=15; break;
}
f = p*w*s*(1-d/100);
printf("%.2f\n", f);
```

给出一个百分制的成绩,要求输出成绩等级'A','B','C','D','E'。90 分以上为'A', 80 ~ 89 分为'B',70 ~ 79 分为'C',60 ~ 69 分为'D',60 分以下为'E'。

```
int grade;
scanf("%d",&grade);
grade /= 10; // 等效于 grade=grade/10;
switch(grade)
{
    case 0: case 1: case 2: case 3: case 4:
    case 5: printf("E"); break;
    case 6: printf("D"); break;
    case 7: printf("C"); break;
    case 8: printf("B"); break;
    case 9:
    case 10: printf("A"); break;
}
```

思考: 如果输入成绩等级, 输出分数段, 如何修改程序?

给出一个百分制的成绩,要求输出成绩等级'A','B','C','D','E'。90 分以上为'A', 80 ~ 89 分为'B',70 ~ 79 分为'C',60 ~ 69 分为'D',60 分以下为'E'。

```
int grade;
scanf("%d",&grade);
grade /= 10; // 等效于 grade=grade/10;
switch(grade)
{
    case 0: case 1: case 2: case 3: case 4:
    case 5: printf("E"); break;
    case 6: printf("D"); break;
    case 7: printf("C"); break;
    case 8: printf("B"); break;
    case 9:
    case 10: printf("A"); break;
}
```

思考: 如果输入成绩等级, 输出分数段, 如何修改程序?

Part III

循环结构程序设计

Outlines

- 17 `while(表达式){...}`
- 18 `do{...}while(表达式);`
- 19 `for(表达式1;表达式2;表达式3){...}`
- 20 循环的嵌套
- 21 `break,continue` 改变循环执行的状态
- 22 循环结构程序设计举例
- 23 循环结构程序设计举例 (续)

为什么需要循环控制

- 要向计算机输入全班 50 个学生的成绩;(重复 50 次相同的输入操作)
- 分别统计全班 50 个学生的平均成绩;(重复 50 次相同的计算操作)

```
float score1,score2,score3,score4,score5,aver; // 5门课成绩及平均成绩
// 输入第1个学生5门课的成绩
scanf("%f%f%f%f%f",&score1,&score2,&score3,&score4,&score5);
// 求第1个学生平均成绩
aver=(score1+score2+score3+score4+score5)/5;
printf("aver=%7.2f",aver); // 输出第1个学生平均成绩
// 输入第2个学生5门课的成绩
scanf("%f%f%f%f%f",&score1,&score2,&score3,&score4,&score5);
// 求第2个学生平均成绩
aver=(score1+score2+score3+score4+score5)/5;
printf("aver=%7.2f",aver); // 输出第2个学生平均成绩
...
```

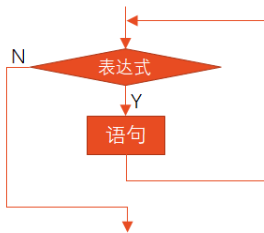
用循环控制处理重复操作

- 要向计算机输入全班 50 个学生的成绩;(重复 50 次相同的输入操作)
- 分别统计全班 50 个学生的平均成绩;(重复 50 次相同的计算操作)

```
float score1, score2, score3, score4, score5, aver; // 5门课成绩及平均成绩
int i=1; // 设整型变量i初值为1
while( i<=50 ) // 当i的值小于或等于50时执行花括号内的语句
{
    scanf("%f%f%f%f%f", &score1, &score2, &score3, &score4, &score5);
    aver=(score1+score2+score3+score4+score5)/5;
    printf("aver=%7.2f", aver);
    i++; // 每执行完一次循环使i的值加1
}
```

while(表达式){...}

```
while(表达式)
{
    // 循环体
    执行多条语句;
}
```



while 循环特点

每轮循环: 首先判断表达式的值, 若“真”(表达式值为非 0) 时, 就执行循环体语句; 为“假”(表达式值为 0) 时, 就不执行循环体语句。

常见错误

```
while(表达式);
{
    // 循环体
    执行多条语句;
}
```

变体

```
while(1)
{
    if(表达式) break; // 退出循环
    执行多条语句;
}
```

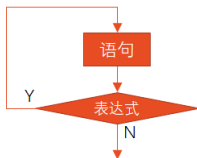
[例 5.1] 求 $1 + 2 + 3 + \cdots + 100$, 即 $\sum_{i=1}^{100} i$

```
int i=1, sum=0; //定义变量i的初值为1, sum的初值为0
while(i <= 100) //当i>100, 条件表达式的值为假(0), 不执行循环体
{ //循环体开始
    sum=sum+i; //第1次累加后, sum的值为1
    i++; //加完后, i的值加1, 为下轮累加做准备
} //循环体结束
printf("sum=%d\n", sum); //输出1+2+3...+100的累加和
```

- 1 循环体如果包含一个以上的语句, 必须用花括号括起来, 作为复合语句出现。
- 2 不要忘记给 i 和 sum 赋初值, 否则它们的值是不可预测的, 结果显然不正确。
- 3 在循环体中应有使循环趋向于结束的语句。如本例中的 $i++$; 语句。如果无此语句, 则 i 的值始终不改变, 循环永远不结束。

do{...}while(表达式);

```
do
{
    // 循环体
    执行多条语句;
} while(表达式);
```



循环特点

首先无条件地执行循环体,然后判断循环条件是否成立。

易犯错误

```
do
{
    // 循环体
    执行多条语句;
} while(表达式)
```

变体

```
do
{
    执行多条语句;
    if(!表达式) break; // 退出循环
} while(1);
```


[例 5.2] 求 $1 + 2 + 3 + \cdots + 100$, 即 $\sum_{i=1}^{100} i$

```
int i=1,sum=0; //定义变量i的初值为1,sum的初值为0
do
{ //循环体开始
    sum=sum+i; //第1次累加后, sum的值为1
    i++; //加完后, i的值加1, 为下次累加做准备
}while(i <= 100); //当i>100, 条件表达式i<=100的值为假, 不执行循环体
printf("sum=%d\n",sum); //输出1+2+3...+100的累加和
```

- 1 在一般情况下, 用 while(){...} 语句和用 do{...}while(); 语句处理同一问题时, 若二者的循环体部分是一样的, 那么结果也一样。
- 2 但是如果 while 后面的表达式一开始就为假 (0 值) 时, 两种循环的结果是不同的。

while(表达式){...} 与 do{...}while(表达式);

[例 5.3] 求 $\sum_{i=n}^{100} i$ 。考虑输入 $n > 100$ 时的情况, 以下程序的不同。

```
int i, sum=0;
scanf("%d", &i);
while(i <= 100)
{
    sum=sum+i;
    i++;
}
printf("sum=%d\n", sum);
```

```
int i, sum=0;
scanf("%d", &i);
do
{
    sum=sum+i;
    i++;
}while(i <= 100);
printf("sum=%d\n", sum);
```

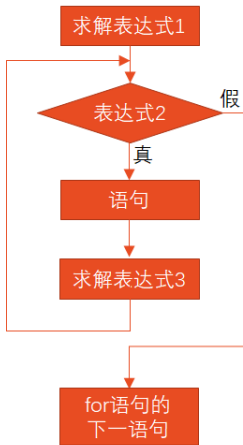
如果输入 101, 左边程序输出: sum=0; 右边输出: sum=101.

for(表达式 1; 表达式 2; 表达式 3){...}

```
for(表达式1;表达式2;表达式3)
{
    // 循环体
    执行多条语句;
}
```

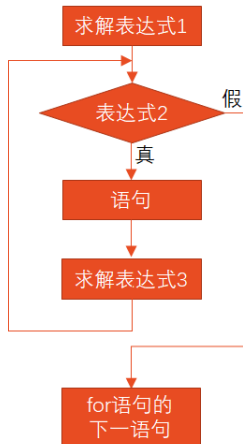
≡

```
表达式1;
while(表达式2)
{
    // 循环体
    执行多条语句;
    表达式3;
}
```



for(表达式 1; 表达式 2; 表达式 3){...}

```
for(表达式1;表达式2  
;表达式3)  
{  
    // 循环体  
    执行多条语句;  
}
```



- 表达式 1: 设置初始条件, 只执行一次。可以为零个、一个或多个变量 (逗号隔开) 设置初值。
- 表达式 2: 是循环条件表达式, 用来判定是否继续循环。在每次执行循环体前先执行此表达式 (包括第 1 次循环), 决定是否继续执行循环。
- 表达式 3: 作为循环的调整, 例如使循环变量增值, 它是在执行完循环体后才进行的。

for(表达式 1; 表达式 2; 表达式 3){...}, 省略表达式

```
int i;
for(i=0;i<=100;i++)
{
    printf("%d\n",i);
}
```

```
int i=0;
for(;i<=100;i++)
{
    printf("%d\n",i);
}
```

```
int i=0;
for(i=0;;i++)
{
    if(i>100) break;//退出循环
    printf("%d\n",i);
}
```

```
int i;
for(i=0;i<=100;)
{
    printf("%d\n",i);
    i++;
}
```

```
int i=0;
for(;i<=100;)
{
    printf("%d\n",i);
    i++;
}
```

```
int i=0;
for(;;)
{
    if(i>100) break;//退出循环
    printf("%d\n",i);
    i++;
}
```

循环的嵌套

01

```
while()
{
    :
    while()
    {...}
}
```

} 内层
循环

02

```
do
{
    :
    do
    {...}
    while();
}while();
```

} 内层
循环

03

```
for(;;)
{
    :
    for(;;)
    {...}
}
```

} 内层
循环

04

```
while()
{
    :
    do
    {...}
    while();
    :
}
```

} 内层
循环

05

```
for(;;)
{
    :
    while()
    {...}
    :
}
```

} 内层
循环

06

```
do
{
    :
    for(;;)
    {...}
}while();
```

} 内层
循环

几种循环的比较

- 1 3 种循环都可以用来处理同一问题,一般情况下它们可以互相代替。
- 2 在 while 循环和 do...while 循环中,只在 while 后面的括号内指定循环条件,因此为了使循环能正常结束,应在循环体中包含使循环趋于结束的语句(如 i++ 等)。
- 3 for 循环可以在表达式 3 中包含使循环趋于结束的操作,甚至可以将循环体中的操作全部放到表达式 3 中(逗号隔开)。因此 for 语句的功能更强,凡用 while 循环能完成的,用 for 循环都能实现。
- 4 用 while 和 do...while 循环时,循环变量初始化的操作应在 while 和 do...while 语句之前完成。而 for 语句可以在表达式 1 中实现循环变量的初始化。
- 5 while 循环、do...while 循环和 for 循环都可以用 break 语句跳出循环,用 continue 语句结束本次循环。

while 循环中, 用 break,continue 改变循环执行的状态



```
while (表达式)
{
    printf("语句1\n");
    //提前终止循环
    if (条件表达式) break;
    printf("语句2\n");
}
// break后到此
```

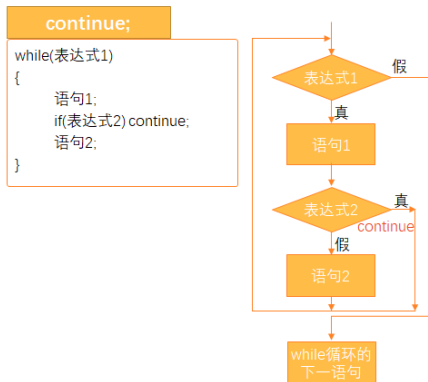
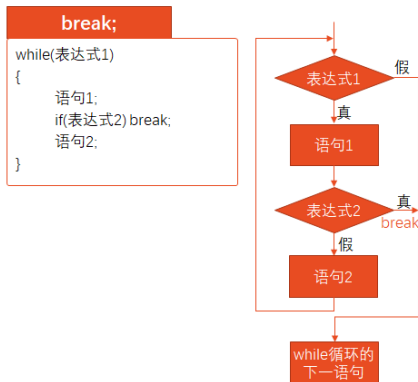
```
while (表达式)
{
    printf("语句1\n");
    //结束本轮循环, 进入下轮循环
    if (条件表达式) continue;
    // continue后到while( )
    printf("语句2\n");
}
```


while 循环中, 体会 break 与 continue 的不同

```
int i=0;
while (i<4)
{
    if (i==2)
    {
        i++;
        //终止本轮循环,开始下轮循环
        continue;
    }
    printf("%d,",i);//0,1,3,
    i++;
}
printf("\n%d\n",i);//4
```

```
int i=0;
while (i<4)
{
    if (i==2)
    {
        i++;
        // 终止整个循环
        break;
    }
    printf("%d,",i);//0,1,
    i++;
}
printf("\n%d\n",i);//3
```

while 循环中,break 语句和 continue 语句的区别

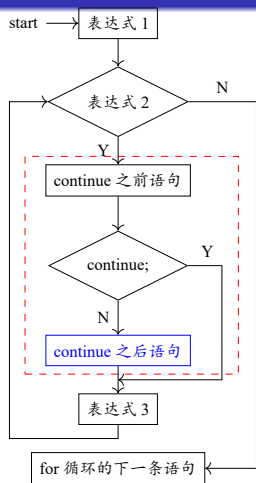


注意: continue 语句只结束本次循环,而非终止整个循环。break 语句结束整个循环,不再判断执行循环的条件是否成立。

注意: for 循环的表达式 3 被视为下轮循环的首语句

[例 5.4] 要求输出 100~200 之间的不能被 3 整除的数。

```
#include <stdio.h>
int main()
{
    int n;
    for (n=100;n<=200;n++) //(表达式1;表达式2;表达式3)
    {
        //终止本轮循环,但会执行for语句的表达式3(n++),开始下
        //轮循环。表达式3被视为下轮循环的首语句。
        if (n%3==0) continue;
        printf("%d ",n);
    }
    printf("\n");
    return 0;
}
```



for 循环中, break 语句和 continue 语句的区别

[例 5.5] 输出以下 4×5 的矩阵。

```
int i,j,n=0; //n: 累计输出数据个数
for(i=1;i<=4;i++) // 行
{
    for(j=1;j<=5;j++,n++) //列
    {
        if(n%5==0) printf("\n");//5数换行
        if (i==3 && j==1) break;
        printf("%d\t",i*j);
    }
}

printf("\n");
```

```
int i,j,n=0; //n: 累计输出数据个数
for(i=1;i<=4;i++) // 行
{
    for(j=1;j<=5;j++,n++) // 列
    {
        if(n%5==0) printf("\n");//5数换行
        if (i==3 && j==1) continue;
        printf("%d\t",i*j); // \t就是Tab
        键, 是特殊字符, 表示多个空格
    }
}

printf("\n");
```

for 循环中, 体会 break 与 continue 的不同

```
int i=0;
for(i=0;i<4;i++)
{
    if(i==2) continue; //终止本轮循环,但会
                       执行for语句的表达式3 (i++), 开始下
                       轮循环
    printf("i=%d,",i); //i=0,i=1,i=3,
}
printf("\nend i=%d\n",i);
//end i=4
```

```
int i=0;
for(i=0;i<4;i++)
{
    if(i==2) break; //终止整个循环,也不
                   会执行for语句的表达式3 (i++)
    printf("i=%d,",i); // i=0,i=1,
}
printf("\nend i=%d\n",i);
//end i=2, 此处根据循环变量i的值可以判
断上述循环是否正常结束, if(i==4)
正常结束。
```

for 循环中, 用 break 语句提前终止循环示例

[例 5.4] 在全系 1000 名学生中举行慈善募捐, 当总数达到 10 万元时就结束, 统计此时捐款的人数以及平均每人捐款的数目。

```
#define SUM 100000 //指定符号常量SUM代表10万
float amount, aver, total; // 单人捐款数, 平均捐款, 捐款总量
int i; // 捐款人数
for (i=1, total=0; i<=1000; i++) // 表达式1给多个变量赋初值, 用逗号隔开。
{
    printf("please enter amount:");
    scanf("%f", &amount);
    total = total + amount;
    if(total >= SUM) break; // 终止整个循环, 也不会执行for语句的表达式3 (i++)
}
i=(i<=1000)?i:(i-1); // 技巧: 判断循环是否提前终止
aver=total/i;
printf("num=%d\naver=%10.2f\n", i, aver);
```

注意: break 语句只能用于循环语句和 switch 语句之中, 而不能单独使用。

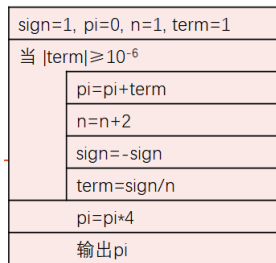
注意事项小结

- 1 while(){ }; do { } while(); for(;;){ } 执行顺序;
- 2 循环变量的开始和结束条件;
- 3 循环体是复合语句时,必须用 { } 扩起来;
- 4 必要时,用 break 结束整个循环,用 continue 结束本轮循环;
- 5 关键是找出循环规律,必要时设计流程图,指导代码实现。

[例 5.7] 用公式 $\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ 求 π 的近似值, 直到发现某一项的绝对值小于 10^{-6} 为止 (该项不累加)。

解题思路: 找规律

- 1 每项的分子都是 1。
- 2 后一项的分母是前一项的分母加 2。
- 3 第 1 项的符号为正, 从第 2 项起, 每一项的符号与前一项的符号相反。在每求出一项后, 检查它的绝对值是否大于或等于 10^{-6} 。




```
#include <stdio.h>
#include <math.h> //程序中用到数学函数fabs, 应包含头文件math.h
int main()
{
    int sign=1; //sign用来表示数值的符号
    double pi=0.0,n=1.0,term=1.0; //pi开始代表多项式的值, 最后代表 $\pi$ 的值, n代表分母, term代表当前项的值
    while(fabs(term)>=1e-6) //检查当前项term的绝对值是否大于或等于 $10^{-6}$ 
    {
        pi=pi+term; //把当前项term累加到pi中
        n=n+2; //n+2是下一项的分母
        sign=-sign; //sign代表符号, 下一项的符号与上一项符号相反
        term=sign/n; //求出下一项的值term
    }
    pi=pi*4; //多项式的和pi乘以4, 才是 $\pi$ 的近似值
    printf("pi=%10.8f\n",pi); //输出 $\pi$ 的近似值
    return 0;
}
```

[例 5.8] 求 Fibonacci(斐波那契) 数列的前 40 个数。这个数列有如下特点: 第 1, 2 两个数为 1, 1。从第 3 个数开始, 该数是其前面两个数之和。即该数列为 1, 1, 2, 3, 5, 8, 13, ..., 用数学方式表示为:

$$\begin{cases} F_1 = 1 & (n = 1) \\ F_2 = 1 & (n = 2) \\ F_n = F_{n-1} + F_{n-2} & (n \geq 3) \end{cases}$$

这是一个有趣的古典数学问题: 有一对兔子, 从出生后第 3 个月起每个月都生一对兔子。小兔子长到第 3 个月后每个月又生一对兔子。假设所有兔子都不死, 问每个月的兔子总数为多少?

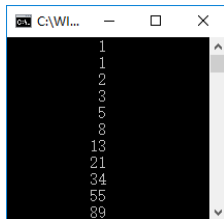
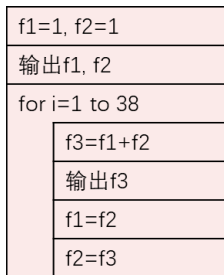
	月数	小兔子对数	中兔子对数	老兔子对数	兔子总对数
兔 子 繁 殖 的 规 律	1	1	0	0	1
	2	0	1	0	1
	3	1	0	1	2
	4	1	1	1	3
	5	2	1	2	5
	6	3	2	3	8
	7	5	3	5	13
	⋮	⋮	⋮	⋮	⋮

不满 1 个月的为小兔子, 满 1 个月不满 2 个月的为中兔子, 满 2 个月以上的为老兔子。

解法一: 利用递推(迭代) 公式: $F_1 = F_2 = 1; F_3 = F_1 + F_2; F_1 = F_2; F_2 = F_3;$

```
#include <stdio.h>

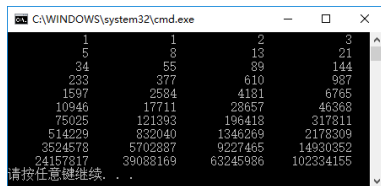
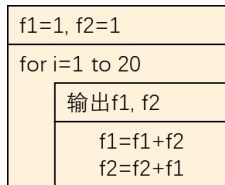
int main()
{
    int f1=1, f2=1, f3;
    int i;
    printf("%12d\n%12d\n", f1, f2);
    for(i=1; i<=38; i++)
    {
        f3=f1+f2;
        printf("%12d\n", f3);
        f1=f2;
        f2=f3;
    }
    return 0;
}
```



解法二: 利用递推(迭代) 公式: $F_1 = F_2 = 1; F_1 = F_1 + F_2; F_2 = F_1 + F_2;$

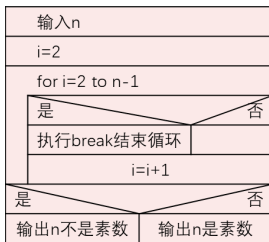
```
#include <stdio.h>

int main()
{
    int f1=1,f2=1;
    int i;
    for(i=1; i<=20; i++)
    {
        printf("%12d%12d",f1,f2);
        if(i%2==0) // 等效 if(!(i%2))
            printf("\n");
        f1=f1+f2;
        f2=f2+f1;
    }
    return 0;
}
```



[例 5.9] 输入一个大于 3 的整数 n,判定它是否为素数 (prime, 又称质数)。

```
#include <stdio.h>
int main()
{
    int n,i;
    printf("please enter a integer number,n=?");
    scanf("%d",&n);
    for (i=2;i<n;i++)
        if(n%i==0) break;
    if(i<n) // for提前结束
        printf("%d is not a prime number.\n",n);
    else // for正常结束
        printf("%d is a prime number.\n",n);
    return 0;
}
```



只要在循环结束后检查循环变量 i 的值,就能判定循环是提前结束还是正常结束的。从而判定 n 是否为素数。这种判断循环结束的方法以后会常用到。

优化: n 不必被 $2 \sim (n-1)$ 内的各整数去除,只须将 n 被 $2 \sim \sqrt{n}$ 之间的整数除即可。因为 n 的每一对因子,必然有一个小于 n , 另一个大于 n 。

```
#include <stdio.h>
#include <math.h>

int main()
{
    int n,i,k;
    printf("please enter a integer number,n=?");
    scanf("%d",&n);
    k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
    for (i=2;i<=k;i++)
        if(n%i==0) break;
    if(i<=k)
        printf("%d is not a prime number.\n",n);
    else
        printf("%d is a prime number.\n",n);
    return 0;
}
```

使用标志变量,判断循环结束条件。

```
int n,i,k,flag=1; // flag: 标志变量
k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
for (i=2;i<=k;i++)
    if(n%i==0) { flag=0; break; }
if(!flag) // for提前结束
    printf("%d is not a prime number.\n",n);
else // for正常结束
    printf("%d is a prime number.\n",n);
```

```
int n,i,k,flag=1; // flag: 标志变量
k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
for (i=2;i<=k && flag;i++) // 比较与上面for的不同
    if(n%i==0) { flag=0; }
if(!flag)
    printf("%d is not a prime number.\n",n);
else
    printf("%d is a prime number.\n",n);
```

使用标志变量,判断循环结束条件。

```
int n,i,k,flag=1; // flag: 标志变量
k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
for (i=2;i<=k;i++)
    if(n%i==0) { flag=0; break; }
if(!flag) // for提前结束
    printf("%d is not a prime number.\n",n);
else // for正常结束
    printf("%d is a prime number.\n",n);
```

```
int n,i,k,flag=1; // flag: 标志变量
k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
for (i=2;i<=k && flag;i++) // 比较与上面for的不同
    if(n%i==0) { flag=0; }
if(!flag)
    printf("%d is not a prime number.\n",n);
else
    printf("%d is a prime number.\n",n);
```

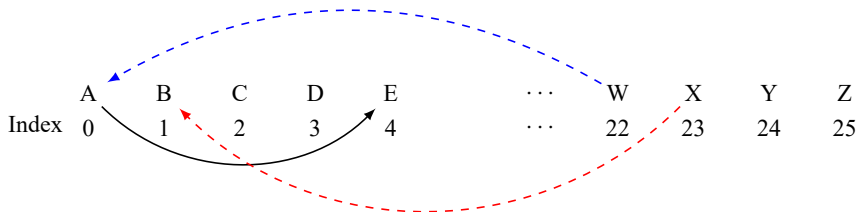

[例 5.10] 求 100 ~ 200 间的全部素数。

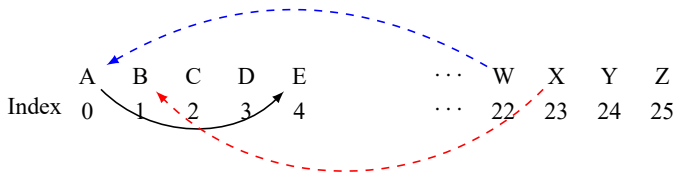
```
#include <stdio.h>
#include <math.h>
int main()
{
    int n,i,k;
    for (n=101;n<=200;n+=2) //n从101变化到200, 对每个奇数n进行判定
    {
        k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
        for(i=2;i<=k;i++)
            if(n%i==0) break;
        if(i>k)
            printf("%d ",n);
    }
    printf("\n");
    return 0;
}
```

[例 5.10] 求 100 ~ 200 间的全部素数。

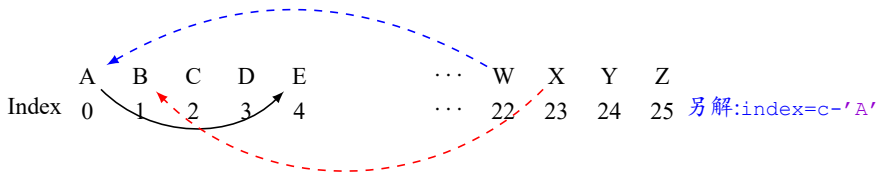
```
#include <stdio.h>
#include <math.h>
int main()
{
    int n,i,k;
    for (n=101;n<=200;n+=2) //n从101变化到200, 对每个奇数n进行判定
    {
        k=sqrt(n); // 自动转换为整数(不会四舍五入), 相当于k=(int)sqrt(n);
        for(i=2;i<=k;i++)
            if(n%i==0) break;
        if(i>k)
            printf("%d ",n);
    }
    printf("\n");
    return 0;
}
```

[例 5.11] 译密码。为使电文保密,往往按一定规律将其转换成密码,收报人再按约定的规律将其译回原文。例如,可以按以下规律将电文变成密码:将字母 A 变成字母 E, a 变成 e, 即变成其后的第 4 个字母, W 变成 A, X 变成 B, Y 变成 C, Z 变成 D。





```
char c;  
c=getchar(); //输入一个字符给字符变量c  
while(c!='\n') //检查c的值是否为换行符'\n'  
{  
    if((c>='a' && c<='z') || (c>='A' && c<='Z')) //c如果是字母  
    {  
        if((c>='W' && c<='Z') || (c>='w' && c<='z')) c = c-22; //如  
            果是26个字母中最后4个字母之一就使c-22  
        else c = c + 4; //如果是前面22个字母之一, 就使c + 4  
    }  
    printf("%c", c); //输出已改变的字符  
    c=getchar(); //再输入下一个字符给字符变量c  
}  
printf("\n");
```



```
char c;  
c=getchar(); //输入一个字符给字符变量c  
while(c!='\n') //检查c的值是否为换行符'\n'  
{  
    if((c>='A' && c<='Z')) //c如果是大写字母  
        c='A' + (c-'A' + 4) % 26;  
    else if((c>='a' && c<='z')) //c如果是小写字母  
        c='a' + (c-'a' + 4) % 26;  
    printf("%c", c); //输出已改变的字符  
    c=getchar(); //再输入下一个字符给字符变量c  
}  
printf("\n");
```

在循环条件中接收输入的字符是一种常见技巧

```
char c;
while((c=getchar())!='\n') //首先从键盘接收一个字符c, 再检查c的值是否为换行
    符'\n'
{
    if((c>='A' && c<='Z')) //c如果是大写字母
        c='A'+(c-'A'+4)%26;
    else if((c>='a' && c<='z')) //c如果是小写字母
        c='a'+(c-'a'+4)%26;
    printf("%c",c); //输出已改变的字符
}
printf("\n");
```

附加题 1: 求 $s = a + aa + aaa + \dots + a \dots a$, 其中 a 是一个 $1 \sim 9$ 的数字。例如 $a = 2, n = 4$ 时, $s = 2 + 22 + 222 + 2222$, a 和 n 由键盘输入。

```
int i,s,n,term = 0;
for(i=1,s=0; i<=n; i++) // 初始化循环变量用逗号隔开
{
    term = term*10 + a;
    s += term;
}
```

附加题 1: 求 $s = a + aa + aaa + \dots + a \dots a$, 其中 a 是一个 $1 \sim 9$ 的数字。例如 $a = 2, n = 4$ 时, $s = 2 + 22 + 222 + 2222$, a 和 n 由键盘输入。

```
int i,s,n,term = 0;
for(i=1,s=0; i<=n; i++) // 初始化循环变量用逗号隔开
{
    term = term*10 + a;
    s += term;
}
```


附加题 2: 韩信点兵。韩信有一队兵,他想知道有多少人,便让士兵排队报数:

按从 1 至 5 报数,最末一个士兵报的数为 1;

按从 1 至 6 报数,最末一个士兵报的数为 5;

按从 1 至 7 报数,最末一个士兵报的数为 4;

按从 1 至 11 报数,最末一个士兵报的数为 10;

计算韩信至少有多少兵。

```
int x=1;
for(;;x++) // 循环体仅含if()结构,看作一条语句,'{}'可省略
    if(x%5==1 && x%6==5 && x%7==4 && x%11==10)
    {
        printf("%d\n",x);
        break;
    }
```

附加题 2: 韩信点兵。韩信有一队兵,他想知道有多少人,便让士兵排队报数:

按从 1 至 5 报数,最末一个士兵报的数为 1;

按从 1 至 6 报数,最末一个士兵报的数为 5;

按从 1 至 7 报数,最末一个士兵报的数为 4;

按从 1 至 11 报数,最末一个士兵报的数为 10;

计算韩信至少有多少兵。

```
int x=1;
for(;;x++) // 循环体仅含if()结构,看作一条语句,'{}'可省略
    if(x%5==1 && x%6==5 && x%7==4 && x%11==10)
    {
        printf("%d\n",x);
        break;
    }
```

附加题 3-1: 求水仙花数。如果一个三位数的个位数、十位数和百位数的立方和等于该数自身,则称该数为水仙花数。

编程求出所有的水仙花数。

解法一: 采用三重循环

```
int i,j,k; // 百、十、个位
for(i=1;i<=9;i++) // 百位
    for(j=0;j<=9;j++) // 十位
        for(k=0;k<=9;k++) // 个位
            if(i*100+j*10+k == i*i*i+j*j*j+k*k*k)
                printf("%d\n",i*100+j*10+k);
```

附加题 3-1: 求水仙花数。如果一个三位数的个位数、十位数和百位数的立方和等于该数自身,则称该数为水仙花数。

编程求出所有的水仙花数。

解法一: 采用三重循环

```
int i,j,k; // 百、十、个位
for(i=1;i<=9;i++) // 百位
    for(j=0;j<=9;j++) // 十位
        for(k=0;k<=9;k++) // 个位
            if(i*100+j*10+k == i*i*i+j*j*j+k*k*k)
                printf("%d\n",i*100+j*10+k);
```

附加题 3-2: 求水仙花数。如果一个三位数的个位数、十位数和百位数的立方和等于该数自身,则称该数为水仙花数。

编程求出所有的水仙花数。

解法二: 采用一重循环

```
int m,i,j,k;  
for(m=100;m<=999;m++)  
{  
    i=m/100; j=m/10%10; k=m%10;  
    if(i*100+j*10+k == i*i*i+j*j*j+k*k*k)  
        printf("%d\n",i*100+j*10+k);  
}
```

思考: 输出共有多少个水仙数?

附加题 3-2: 求水仙花数。如果一个三位数的个位数、十位数和百位数的立方和等于该数自身,则称该数为水仙花数。

编程求出所有的水仙花数。

解法二: 采用一重循环

```
int m,i,j,k;
for (m=100;m<=999;m++)
{
    i=m/100; j=m/10%10; k=m%10;
    if(i*100+j*10+k == i*i*i+j*j*j+k*k*k)
        printf("%d\n",i*100+j*10+k);
}
```

思考: 输出共有多少个水仙数?

附加题 3-3: 求整数区间 $[a, b]$ 中水仙花数的个数。

```
int n=0; //计数
int a,b; // a,b 区间
int i,t; // 循环变量, 代表a,b区间的每个数
int sum; // i的各位立方和
scanf("%d%d",&a,&b);
for(i=a;i<=b;i++) // 考察i是否水仙数
{
    sum = 0; t=i; // 临时变量记住i; 易遗漏每次内层循环前sum要归0
    while(t!=0) // 累加各位立方
    {
        sum=sum+(t%10)*(t%10)*(t%10); // 不好: sum+=pow(t%10,3);
        t=t/10;
    }
    if(sum==i) n++; // i是水仙数
}
printf("%d\n",n);
```

附加题 3-3: 求整数区间 $[a, b]$ 中水仙花数的个数。

```
int n=0; //计数
int a,b; // a,b 区间
int i,t; // 循环变量, 代表a,b区间的每个数
int sum; // i的各位立方和

scanf("%d%d",&a,&b);
for(i=a;i<=b;i++) // 考察i是否水仙数
{
    sum = 0; t=i; // 临时变量记住i; 易遗漏每次内层循环前sum要归0
    while(t!=0) // 累加各位立方
    {
        sum=sum+(t%10)*(t%10)*(t%10); // 不好: sum+=pow(t%10,3);
        t=t/10;
    }
    if(sum==i) n++; // i是水仙数
}

printf("%d\n",n);
```


附加题 4: 百钱百鸡, 已知公鸡 5 个钱 1 只, 母鸡 3 个钱 1 只, 小鸡 1 个钱 3 只, 用 100 个钱买了 100 只鸡。问公鸡、母鸡、小鸡各几只?

```
int x,y,z; // 公鸡、母鸡、小鸡个数
for(x=0;x<=100;x++)
    for(y=0;y<=100;y++)
        for(z=0;z<=100;z++)
            if(5*x+3*y+z/3 == 100 && x+y+z == 100 && z%3 == 0) //全部条件
                printf("%d,%d,%d\n",x,y,z);
```

如何考虑无解的情况?

附加题 4: 百钱百鸡, 已知公鸡 5 个钱 1 只, 母鸡 3 个钱 1 只, 小鸡 1 个钱 3 只, 用 100 个钱买了 100 只鸡。问公鸡、母鸡、小鸡各几只?

```
int x, y, z; // 公鸡、母鸡、小鸡个数
for (x=0; x<=100; x++)
    for (y=0; y<=100; y++)
        for (z=0; z<=100; z++)
            if (5*x+3*y+z/3 == 100 && x+y+z == 100 && z%3 == 0) //全部条件
                printf("%d,%d,%d\n", x, y, z);
```

如何考虑无解的情况?

附加题 5-1: 求整数 a, b 的最大公约数, 当两个数中有一个为 0 时, 公约数是不为 0 的那个整数; 当两个整数互质时最大公约数为 1。输入两个整数 a 和 b , 求最大公约数。

```
int main() // 暴力循环求解, 效率低
{
    int a,b,t=-1,i; // t给初值是好习惯, 否则下面程序逻辑有可能使t得到随机值。
    scanf("%d%d",&a,&b); // 机试系统不要想当然给提示语句, 除非题目要求
    if(a<b) { t=a; a=b; b=t; } // 交换a,b,使a是较大者
    if(b==0) t=a; // 考虑分母为0的情况, 比如: 5,0的最大公约数为5
    else
    {
        for(i=b;i>0;i--)
        {
            if(a%i==0 && b%i==0)
            { t=i; break; } // 求得最大公约数, a,b互质, 必然t=1
        }
    }
    printf("%d\n",t);
    return 0;
}
```

附加题 5-1: 求整数 a, b 的最大公约数, 当两个数中有一个为 0 时, 公约数是不为 0 的那个整数; 当两个整数互质时最大公约数为 1。输入两个整数 a 和 b , 求最大公约数。

```
int main() // 暴力循环求解, 效率低
{
    int a,b,t=-1,i; // t给初值是好习惯, 否则下面程序逻辑有可能使t得到随机值。
    scanf("%d%d", &a, &b); // 机试系统不要想当然给提示语句, 除非题目要求
    if(a<b) { t=a; a=b; b=t; } // 交换a,b, 使a是较大者
    if(b==0) t=a; // 考虑分母为0的情况, 比如: 5,0的最大公约数为5
    else
    {
        for(i=b; i>0; i--)
        {
            if(a%i==0 && b%i==0)
            { t=i; break; } // 求得最大公约数, a,b互质, 必然t=1
        }
    }
    printf("%d\n", t);
    return 0;
}
```

求整数 a, b 的最大公约数, 欧几里得算法

古希腊数学家欧几里德在其著作《The Elements》中最早描述了这种算法。

定理: 两个整数的最大公约数等于其中较小的那个数和两数相除余数的最大公约数。



令 $a = bq + r, q = \lfloor \frac{a}{b} \rfloor, r = a \% b$, 设 a, b 的公约数 $u, (a = su, b = tu)$, 我们有 $r = a - bq = su - qtu = (s - qt)u$, 表明 a, b 的公约数 u 也整除 r 。

类似的, 设 b 和 r 的公约数 $v, (b = s'v, r = t'v)$, 则

$a = bq + r = s'vq + t'v = (s'q + t')v$, 表明 b 和 r 的公约数 v 也整除 a 。

(1) 如果 $r = a \% b = 0$, 最大公约数为 b . (2) 如果 $r \neq 0$, 令 $a = b, b = r$, 返回到 (1).
 重复迭代 (1), (2) 步, 直至余数为 0, 即求得最大公约数。

注: $\lfloor \frac{a}{b} \rfloor$ 表示 $\frac{a}{b}$ 的结果向下取整, 在 C 语言中, $a/b = \lfloor \frac{a}{b} \rfloor$

欧几里得算法 — 求整数 a, b 最大公约数: 迭代过程

$$q_1 = \left\lfloor \frac{a}{b} \right\rfloor$$

$$a = bq_1 + r_1$$

$$r_1 = a - bq_1$$

$$q_2 = \left\lfloor \frac{b}{r_1} \right\rfloor$$

$$b = q_2r_1 + r_2$$

$$r_2 = b - q_2r_1$$

$$q_3 = \left\lfloor \frac{r_1}{r_2} \right\rfloor$$

$$r_1 = q_3r_2 + r_3$$

$$r_3 = r_1 - q_3r_2$$

$$q_4 = \left\lfloor \frac{r_2}{r_3} \right\rfloor$$

$$r_2 = q_4r_3 + r_4$$

$$r_4 = r_2 - q_4r_3$$

...

$$q_n = \left\lfloor \frac{r_{n-2}}{r_{n-1}} \right\rfloor$$

$$r_{n-2} = q_nr_{n-1} + r_n$$

$$r_n = r_{n-2} - q_nr_{n-1}$$

$$q_{n+1} = \left\lfloor \frac{r_{n-1}}{r_n} \right\rfloor$$

$$r_{n-1} = q_{n+1}r_n + 0$$

$$r_n = r_{n-1}/q_{n+1}$$

迭代至 $q_{n+1}, r_{n-1} \% r_n = 0$, 得到 a, b 的最大公约数: $r_n = r_{n-1}/q_{n+1}$. 就是余数为 0 时的分母值.

欧几里得算法 — 求整数 a, b 最大公约数: 迭代过程 (例)

$$a = 42, b = 30$$

$$q_1 = \lfloor \frac{a}{b} \rfloor = \lfloor \frac{42}{30} \rfloor = 1 \quad r_1 = 12$$

$$q_2 = \lfloor \frac{b}{r_1} \rfloor = \lfloor \frac{30}{12} \rfloor = 2 \quad r_2 = 6$$

$$q_3 = \lfloor \frac{r_1}{r_2} \rfloor = \lfloor \frac{12}{6} \rfloor = 2 \quad r_3 = 0$$

Greatest Common Divisor of (42, 30) is

$$r_2 = 6$$

$$a = 144, b = 55$$

$$r_1 = a \% b = 34 \quad r_2 = b \% r_1 = 21$$

$$r_3 = r_1 \% r_2 = 13 \quad r_4 = r_2 \% r_3 = 8$$

$$r_5 = r_3 \% r_4 = 5 \quad r_6 = r_4 \% r_5 = 3$$

$$r_7 = r_5 \% r_6 = 2 \quad r_8 = r_6 \% r_7 = 1$$

$$r_9 = r_7 \% r_8 = 0$$

Greatest Common Divisor of (144, 55) is

$$r_8 = 1$$

欧几里得算法 — 求整数 a, b 的最大公约数: 伪代码

定理: 两个整数的最大公约数等于其中较小的那个数和两数相除余数的最大公约数。

a (大), b (小) 的最大公约数: 因为: $a = qb + r$, $q = a/b$; $r = a \% b$, $\Rightarrow a, b$ 的公约数能整除 b 和 r .
 $r = a \% b$, r 为 0, 则 b 就是最大公约数。否则迭代循环, $a = b$, $b = r$, 直到余数为零, 则分母就是最大公约数。

```
while(1)
{
    if(b==0) { gcd=a; break; } // 分母(上轮计算的余数)为0时, a就是最大公约数
    r = a%b; // 注意b为0时, 不能计算余数, a就是最大公约数
    if(r==0) { gcd=a; break; } // 本轮循环的a(上轮循环的b)就是最大公约数
    a=b; b=r; // 准备下一轮迭代
}
```


求整数 a, b 的最大公约数, 欧几里得算法, 参考代码 (1)

```
int a,b,r,t;

scanf("%d%d",&a,&b); // 机试系统不要想当然给提示语句, 除非题目要求

if(a<b) { t=a; a=b; b=t; } // 交换a,b,使a是较大者

while(1)
{
    if(b==0) { t=a; break; } // 分母(上轮计算的余数)为0时, a就是最大公约数
    r = a%b;
    if(r==0) {t=b; break;} // b就是最大公约数
    a=b; b=r; // 准备下一轮迭代
}

printf("%d\n",t); // 输出最大公约数
```

求整数 a, b 的最大公约数, 欧几里得算法, 参考代码 (2)

```
int main()
{
    int a,b,r,t;
    scanf("%d%d",&a,&b); // 机试系统不要想当然给提示语句, 除非题目要求
    if(a<b) { t=a; a=b; b=t; } // 交换a,b,使a是较大者
    if (b==0) // 考虑分母为0的情况, 比如: 5,0的最大公约数为5
    {
        printf("%d\n",a);
        return 0; // 主函数结束
    }
    while((r=a%b) !=0) // 去除了分母为0的情况
    {
        a=b; b=r; // 准备下一轮迭代
    }
    printf("%d\n",b);
    return 0; // 主函数结束
}
```

求整数 a, b 的最大公约数, 欧几里得算法, 参考代码 (3)

```
int main()
{
    int a,b,r,t;
    scanf("%d%d",&a,&b); // 机试系统不要想当然给提示语句, 除非题目要求
    if(a<b) { t=a; a=b; b=t; } // 交换a,b,使a是较大者
    if (b==0) // 考虑分母为0的情况, 比如: 5,0的最大公约数为5
    {
        printf("%d\n",a);
    }
    else
    {
        // 排除了分母为0时不能求余数的情况
        while ((r=a%b)!=0) // a/b的余数赋值给r,r不等于0时执行循环体
        { a=b; b=r; }
        printf("%d\n",b);
    }
    return 0; // 主函数结束
}
```

注意事项小结

- 1 while(){ }; do { } while(); for(;;){ } 执行顺序;
- 2 循环变量的开始和结束条件;
- 3 循环体是复合语句时,必须用 { } 扩起来;
- 4 必要时,用 break 结束整个循环,用 continue 结束本次循环;
- 5 关键是找出循环规律,必要时设计流程图,指导代码实现。

Part IV

数组

Outlines

24 定义数组: int a[10];

25 引用数组

26 初始化数组: int a[5]={1,2,3,4,5};

27 冒泡排序

28 定义和引用二维数组: int a[10][20];

29 引用数组: int i=0,j=0; a[i][j]

30 二维数组的初始化: int a[3][3]={ {1,2,3},{4,5,6},{7,8,9} }

31 用字符数组表示字符串 char s[81]

32 字符串处理函数 #include<string.h>

33 程序举例

为什么需要数组

- 记录全班 50 个学生一门课的成绩, 用于后续其它计算, 例如排序;

Recall: 用循环迭代计算平均成绩时, 并不需要记录 50 个成绩.

- 用 50 个 float 型简单变量表示学生的成绩
 - 烦琐, 如果有 1000 名学生怎么办呢?
 - 没有反映出这些数据间的内在联系, 实际上这些数据是同一个班级、同一门课程的成绩, 它们具有相同的属性。

```
float s0,s1,s2,...,s49; //50名学生一门课的成绩
float s[50]; // float型数组, 记录50名学生一门课的成绩
int i; // 表示数组下标
for(i=0;i<50;i++) scanf("%f",&s[i]);
```

数组

- 数组是一组有序数据的集合。下标代表数据在数组中的序号。
- 用数组名和下标即可唯一地确定数组中的元素。
- 数组中的每一个元素都属于同一个数据类型。

为什么需要数组

- 记录全班 50 个学生一门课的成绩, 用于后续其它计算, 例如排序;

Recall: 用循环迭代计算平均成绩时, 并不需要记录 50 个成绩.

- 用 50 个 float 型简单变量表示学生的成绩
 - 烦琐, 如果有 1000 名学生怎么办呢?
 - 没有反映出这些数据间的内在联系, 实际上这些数据是同一个班级、同一门课程的成绩, 它们具有相同的属性。

```
float s0,s1,s2,...,s49; //50名学生一门课的成绩
float s[50]; // float型数组, 记录50名学生一门课的成绩
int i; // 表示数组下标
for(i=0;i<50;i++) scanf("%f",&s[i]);
```

数组

- 数组是一组有序数据的集合。下标代表数据在数组中的序号。
- 用数组名和下标即可唯一地确定数组中的元素。
- 数组中的每一个元素都属于同一个数据类型。

定义数组: int a[10];

定义一维数组:

元素类型 数组名 [常量表达式 (表示元素个数—数组的长度)]

#define NUM 100 // 一般在main函数之前定义

float s[50]; // 50名学生一门课的成绩, 边界: s[0], s[49]

int a[10]; // 10个元素的整型数组, 边界: a[0], a[9]

char b[50]; // 50个元素的char型数组, 边界: b[0], b[49]

// 好的定义, 易控制数组的边界.

double c[NUM]; // 常量NUM个元素的double数组, 边界: c[0], c[NUM-1]

Notes

数组元素的下标从 0 开始, int a[10]; 10 个整型元素, 则最大下标值为 9, 不存在数组元素 a[10]. 定义数组常见错误: int n,a[n];

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
------	------	------	------	------	------	------	------	------	------

引用数组: 读写数组 a 的第 i 个元素的值: a[i]

```
int a[10]; // 10个元素的整型数组, 边界: a[0],a[9]
int i;
a[0]=10; // 给a数组的第1个元素赋值(写)
a[9]=10; // 给a数组的最后1个元素赋值(写)
printf("%d,%d",a[0],a[9]);
for(i=0;i<10;i++) a[i] = i+1; // 给数组的第i个元素赋值, 注意边界a[0],a[9]
for(i=0;i<10;i++) printf("%d\t",a[i]); // 输出数组a的第i个元素(读):a[i]
for(i=0;i<10;i++) scanf("%d",&a[i]); // 输入10个整数, 存入数组a中。注意'&'
```

Notes

数组元素的下标从 0 开始, int a[10]; 10 个整型元素, 则最大下标值为 9, 不存在数组元素 a[10]. 常见错误: scanf("%d",a[i]);

数组定义与引用的一般范式

```
#define N 100 // 估计数组的最大长度

int main()
{
    int a[N], n, i; // 按照最大长度定义数组，用n表示数组的实际长度。
    scanf("%d",&n); // 一般从键盘接收数组的实际长度
    // 举例输入数组各个元素，自动按空格隔开输入数据。
    for(i=0;i<n;i++) scanf("%d",&a[i]); // 错误形式: scanf("%d ",&a[i]);
    // 举例输出数组各个元素
    for(i=0;i<n;i++) printf("%d\t",a[i]);
    printf("\n");
}
```

正确理解以空格隔开读取各个元素

```
#define N 100 // 估计数组的最大长度
```

```
int main()
```

```
{
```

```
    int a[N], n, i; // 按照最大长度定义数组, 用n表示数组的实际长度.
```

```
    scanf("%d",&n); // 一般从键盘接收数组的实际长度
```

```
    // 最简单的形式, 自动按空格隔开输入数据.
```

```
    for(i=0;i<n;i++) scanf("%d",&a[i]); // 错误形式: scanf("%d ",&a[i]);
```

```
    // 或, 显式以空格隔开
```

```
    for(i=0;i<n-1;i++) scanf("%d ",&a[i]); // Read: a[0], ..., a[n-2]
```

```
    scanf("%d",&a[n-1]); // 单独输入最后一个元素值
```

```
}
```

$a[0]$ 空格 $a[1]$ 空格 \dots $a[n-2]$ 空格 $a[n-1]$
 1^{th} 2^{th} $(n-1)^{th}$ n^{th}
 $n-1$

[例 6.1] 对 10 个字符型数组元素依次赋值为'a','b',…。要求按逆序输出。

```
#include <stdio.h>

#define N 10

int main()
{
    char c[N]; // 边界: c[0], c[N-1]
    int i;
    for(i=0;i<N;i++) c[i]='a'+i;
    for(i=N-1;i>=0;i--) printf("%c ", c[i]);
    printf("\n");
    return 0;
}
```

[例 6.1] 对 10 个字符型数组元素依次赋值为'a','b',…。要求按逆序输出。

```
#include <stdio.h>

#define N 10

int main()
{
    char c[N]; // 边界: c[0], c[N-1]
    int i;
    for(i=0;i<N;i++) c[i]='a'+i;
    for(i=N-1;i>=0;i--) printf("%c ", c[i]);
    printf("\n");
    return 0;
}
```

初始化数组: int a[5]={1,2,3,4,5};

为了使程序简洁,常在定义数组的同时给各数组元素赋值,这称为数组的初始化。

```
#define N 10 // 一般在main函数之前定义
int a[N]={1,2,3,4,5,6,7,8,9,10}; // 在定义数组时对全部数组元素赋予初值。
char c[10]={'a','b'}; // 仅给数组第1,2个元素赋值。其他元素的值不确定
double d[]={10.0,10.2,10.3}; // 等效于 double d[3]={10.0,10.2,10.3}

d[2] = 20.2; // 修改第3个元素

// 利用循环结构, 给各元素赋值。
int i;
for(i=0;i<N;i++) a[i]=0;
// 注意:
int a[N]={0}; // 仅表示a[0]=0; 其它元素值是不确定的。
```

// [例6.2] 用数组来处理求Fibonacci数列问题。

```
#include <stdio.h>
```

```
#define N 20
```

```
int main()
```

```
{
```

```
    int i, f[N]={1,1}; //对最前面两个元素f[0]和f[1]赋初值1
```

```
    for(i=2;i<N;i++) // 边界f[0], f[N-1]
```

```
        f[i]=f[i-2]+f[i-1]; //先后求出f[2]~f[N-1]的值
```

```
    for(i=0;i<N;i++)
```

```
    {
```

```
        if(i%5==0) printf("\n"); //控制每输出5个数后换行
```

```
        printf("%12d", f[i]); //输出一个数
```

```
    }
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```


冒泡排序 (第 1 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

9
8
5
0

第 1 趟原始数据

8
9
5
0

第 1 趟第 1 次
相邻两数比较

8
5
9
0

第 1 趟第 2 次
相邻两数比较

8
5
0
9

第 1 趟第 3 次相
邻两数比较 (第
1 个大数沉底)

第 j=1 趟比较, 共进行 N-j=3 次相邻两数比较

冒泡排序 (第 1 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

9
8
5
0

第 1 趟原始数据

8
9
5
0

第 1 趟第 1 次
相邻两数比较

8
5
9
0

第 1 趟第 2 次
相邻两数比较

8
5
0
9

第 1 趟第 3 次相
邻两数比较 (第
1 个大数沉底)

第 j=1 趟比较, 共进行 N-j=3 次相邻两数比较

冒泡排序 (第 1 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

9
8
5
0

第 1 趟原始数据

8
9
5
0

第 1 趟第 1 次
相邻两数比较

8
5
9
0

第 1 趟第 2 次
相邻两数比较

8
5
0
9

第 1 趟第 3 次相
邻两数比较 (第
1 个大数沉底)

第 j=1 趟比较, 共进行 N-j=3 次相邻两数比较

冒泡排序 (第 1 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

9
8
5
0

第 1 趟原始数据

8
9
5
0

第 1 趟第 1 次
相邻两数比较

8
5
9
0

第 1 趟第 2 次
相邻两数比较

8
5
0
9

第 1 趟第 3 次相
邻两数比较 (第
1 个大数沉底)

第 j=1 趟比较, 共进行 N-j=3 次相邻两数比较

冒泡排序 (第 2 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

8
5
0
9

第 2 趟原始数据

5
8
0
9

第 2 趟第 1 次
相邻两数比较

5
0
8
9

第 2 趟第 2 次
相邻两数比较
(第 2 大沉底)

第 j=2 趟比较, 共进行 N-j=2 次相邻两数比较

冒泡排序 (第 2 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

8
5
0
9

第 2 趟原始数据

5
8
0
9

第 2 趟第 1 次
相邻两数比较

5
0
8
9

第 2 趟第 2 次
相邻两数比较
(第 2 大沉底)

第 j=2 趟比较, 共进行 N-j=2 次相邻两数比较

冒泡排序 (第 2 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

8
5
0
9

第 2 趟原始数据

5
8
0
9

第 2 趟第 1 次
相邻两数比较

5
0
8
9

第 2 趟第 2 次
相邻两数比较
(第 2 大沉底)

第 j=2 趟比较, 共进行 N-j=2 次相邻两数比较

冒泡排序 (第 3 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

5
0
8
9

第 3 趟原始数据

0
5
8
9

第 3 趟第 1 次

相邻两数比较

(第 3 大沉底)

第 $j=3$ 趟比较, 共进行 $N-j=1$ 次相邻两数比较

总结: 外层循环 (j) 执行 $(N-1)$ 次, 内层循环 (i) 执行 $(N-j)$ 次相邻两数比较

冒泡排序 (第 3 趟比较)

int a[4]={9,8,5,0}; // N=4 个元素, 要求从小到大顺序排列

5
0
8
9

第 3 趟原始数据

0
5
8
9

第 3 趟第 1 次

相邻两数比较

(第 3 大沉底)

第 $j=3$ 趟比较, 共进行 $N-j=1$ 次相邻两数比较

总结: 外层循环 (j) 执行 $(N-1)$ 次, 内层循环 (i) 执行 $(N-j)$ 次相邻两数比较

冒泡排序 (总结)

#define N 4 // 数组长度

int a[N]; 原始

a[0] 9

a[1] 8

a[2] 5

a[3] 0

j=1			j=2		j=3
i=0	i=1	i=2	i=0	i=1	i=0
8	8	8	5	5	0
9	5	5	8	0	5
5	9	0	0	8	8
0	0	9	9	9	9

for (j=1; j<=N-1; j++) // N-1次循环

{

// N-j次比较相邻元素比较

for (i=0; i<=N-1-j; i++)

{ 比较 a[i], a[i+1]}

}

- $(N-1)$ 趟外层循环, $(j = 1, 2, \dots, N-1)$, 表示第 j 趟比较。
- $(N-j)$ 次内层循环, $(i = 0, 1, \dots, N-1-j)$ 相邻元素比较 $(a[i], a[i+1])$, 必要时交换。
- 检查数组边界 (两端 $a[i], a[i+1]$), 不要越界, 令 $N=1, 2, 3, 4$. 检查 $a[i+1]$.

冒泡排序 (核心程序)

```
#define N 10 // 估计数组的最大长度

int a[N]={9,8,7,6,5,4,3,2,1,0},i,j,t, n; // n是数组的实际长度
scanf("%d",&n); // 一般从键盘输入数组的实际长度
for(j=1;j<=n-1;j++) //进行n-1次循环, 实现n-1趟比较
{
    for(i=0;i<=n-1-j;i++) //在每一趟中进行n-j次比较相邻元素两两比较
    {
        if(a[i]>a[i+1]) //相邻两个数比较, 注意检查数组不要越界
        { t=a[i]; a[i]=a[i+1]; a[i+1]=t; } // 交换
    }
}

printf("\n the sorted numbers :\n");
for(i=0;i<n;i++) printf("%d ",a[i]);
```

冒泡排序 (优化)

优化: 第 j 趟排序中, 没有进行相邻元素的交换, 表示数据已经排序好, 没有必要进行此后的 $(n-j)$ 趟排序。

```
#define N 10 // 估计数组的最大长度
int a[N]={9,8,7,6,5,4,3,2,1,0}, i, j, t, n, flag; // n是数组的实际长度, flag
        是标志变量
scanf("%d", &n); // 一般从键盘输入数组的实际长度
for(j=1; j<=n-1; j++) // 进行n-1次循环, 实现n-1趟比较
{
    flag = 0; // 每趟排序, 初始化标志变量flag, 表示未进行交换
    for(i=0; i<=n-1-j; i++) // 在每一趟中进行n-j次相邻元素两两比较
    {
        if(a[i]>a[i+1]) // 相邻两个数比较, 注意检查数组不要越界
        { t=a[i]; a[i]=a[i+1]; a[i+1]=t; flag=1; } // 交换, 设置标志变量
    }
    if(!flag) break; // 表示第j趟未交换, 排序好了!
}
printf("\n the sorted numbers :\n");
for(i=0; i<n; i++) printf("%d ", a[i]);
```

冒泡排序 (输出每趟排序的结果, 检查排序中间过程.)

```
for(j=1;j<=n-1;j++) //进行n-1次循环, 实现n-1趟比较
{
    flag = 0; // 每趟排序, 初始化flag, 表示未进行交换
    for(i=0;i<=n-1-j;i++) //在每一趟中进行n-j次相邻元素两两比较
    {
        if(a[i]>a[i+1]) //相邻两个数比较, 注意检查数组不要越界
        { t=a[i]; a[i]=a[i+1]; a[i+1]=t; flag=1; } // 交换, 设置标志变量
    }
    printf("\n 第%d趟排序: \n", j);
    for(t=0;t<N;t++) printf("%d ",a[t]); // 临时变量t的复用
    if(!flag) break; // 表示第j趟未交换, 排序好了!
}

printf("\n the sorted numbers :\n");
for(i=0;i<n;i++) printf("%d ",a[i]);
```

二维数组应用场景

Example

有 3 个小分队, 每队有 6 名队员, 要把这些队员的工资用数组保存起来以备查。

	队员 1	队员 2	队员 3	队员 4	队员 5	队员 6
第 1 分队	1000	2000	1500	2400	3000	5000
第 2 分队	3000	4000	2500	2300	2500	4000
第 3 分队	4000	5000	1200	2300	3200	5500

`float pay[3][6];` // 定义 3 行 6 列的二维数组, 称为 3x6 矩阵 (matrix)。

1 分队	pay[0][0]	pay[0][1]	pay[0][2]	pay[0][3]	pay[0][4]	pay[0][5]
2 分队	pay[1][0]	pay[1][1]	pay[1][2]	pay[1][3]	pay[1][4]	pay[1][5]
3 分队	pay[2][0]	pay[2][1]	pay[2][2]	pay[2][3]	pay[2][4]	pay[2][5]

定义二维数组: int a[10][20];

定义二维数组:

元素类型 数组名 [行数][列数]

```
int a[5][10]; // 定义5行10列的整型二维数组, 边界:a[0][0], a[4][9]
```

```
float pay[3][6]; // 定义3行6列的单精度浮点型二维数组
```

```
double pay1[3][6]; // 定义3行6列的双精度浮点型二维数组
```

```
char c[2][20]; // 定义2行20列的字符型二维数组
```

// 好的定义, 行数列数定义为常量, 易控制数组的边界。

```
#define M 5 // 估计最大行数
```

```
#define N 10 // 估计最大列数
```

```
int a[M][N]; // 定义M行N列的整型二维数组, 边界:a[0][0], a[M-1][N-1]
```

二维数组的存储

```
float a[3][4]; // 定义3行4列的单精度浮点型二维数组
```

Notes

用矩阵形式(如3行4列形式)表示二维数组,是逻辑上的概念,能形象地表示出行列关系。而在内存中,各元素是连续存放的,不是二维的,是线性的。

数组元素的下标从0开始, int a[3][4]; 12个整型元素,先行后列存储数组元素。右图左列是数组元素在内存中的“虚拟”地址。

a[0][0]	a[0][1]	a[0][2]	a[0][3]	← 第0行
a[1][0]	a[1][1]	a[1][2]	a[1][3]	← 第1行
a[2][0]	a[2][1]	a[2][2]	a[2][3]	← 第2行

1000	a[0][0]
1004	a[0][1]
1008	a[0][2]
1012	a[0][3]
1016	a[1][0]
1020	a[1][1]
1024	a[1][2]
1028	a[1][3]
1032	a[2][0]
1036	a[2][1]
1040	a[2][2]
1044	a[2][3]

引用数组: int i=0,j=0; a[i][j]

```
#define M 5 // 行数
#define N 10 // 列数
float a[M][N]; // 定义M行N列的单精度浮点型二维数组
int i,j;
a[0][0]=10; // 给第0行第0列元素赋值
a[M-1][N-1]=20; // 给第M-1行第N-1列元素赋值
```

```
// 数组元素赋值
for(i=0;i<M;i++) //遍历第i行
    for(j=0;j<N;j++) //遍历第j列
        a[i][j]=i*j; //给第i行第j列元素赋值

// 键盘输入数组元素
for(i=0;i<M;i++) // 行
    for(j=0;j<N;j++) // 列
        scanf("%f", &a[i][j]);
```

```
// 输出数组元素
for(i=0;i<M;i++) // 行
    for(j=0;j<N;j++) // 列
    {
        printf("%d\t",a[i][j]);
        if(j==N-1) printf("\n");
    }
```

输出每个小分队的平均工资

```
#define M 100 // 估计最大的行数
#define N 100 // 估计最大的列数

float pay[M][N]; //按照最大值定义数组的大小
int m,n,i,j; //m,n表示实际pay数组行数和列数
float sum; //每个小分队所有人员的工资总和

scanf("%d%d",&m,&n); // 输入小分队数和队员数, 就是pay数组的实际行数和列数
for(i=0;i<m;i++) // 依次输入每个队员的工资
    for(j=0;j<n;j++) scanf("%f",&pay[i][j]); // 注意取地址符'&'

for(i=0;i<m;i++) // 行
{
    sum=0; // 内层循环执行前, 必须置零。易忘记!!!
    for(j=0;j<n;j++) // 列
        sum += pay[i][j]; // 累加第i小分队所有人员的工资
    printf("第%d小分队平均工资=%f\n",i+1,sum/n);
}
```

1 分队	pay[0][0]	pay[0][1]	pay[0][2]	pay[0][3]	pay[0][4]	pay[0][5]
2 分队	pay[1][0]	pay[1][1]	pay[1][2]	pay[1][3]	pay[1][4]	pay[1][5]
3 分队	pay[2][0]	pay[2][1]	pay[2][2]	pay[2][3]	pay[2][4]	pay[2][5]

二维数组的初始化: int a[3][3]={ {1,2,3},{4,5,6},{7,8,9} }

```
#define M 3 // 行数
#define N 3 // 列数

int a[M][N]={ {1,2,3},{4,5,6},{7,8,9} }; // 定义二维数组并初始化
int b[M][N]={ {1,2},{4,5},{7,8} }; // 定义二维数组并初始化, 第N-1列未赋值
int c[M][N]={ {1,2},{},{7,8} }; // 定义二维数组并初始化, 第N-1列以及第1行未赋值
// 给未初始化的元素赋值

b[0][N-1]=3; b[1][N-1]=6; b[2][N-1]=9;
c[0][N-1]=3;
c[1][0]=4; c[1][1]=5; c[1][N-1]=6;
c[2][N-1]=9;

// 利用循环结构, 给各元素赋值

int i,j;
for(i=0;i<M;i++)
    for(j=0;j<N;j++)
        a[i][j]=0;
```

例 6.4 将二维数组行列互换, 存到另一个二维数组中。

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```
#define M 2
#define N 3
int a[M][N]={ {1,2,3}, {4,5,6} };
int b[N][M],i,j;
```

```
for(i=0;i<M;i++) // 行
{
    for(j=0;j<N;j++) // 列
    {
        printf("%5d",a[i][j]);
        b[j][i]=a[i][j];
    }
    printf("\n"); // 换行
}
```

```
for(i=0;i<N;i++) // 行
{
    for(j=0;j<M;j++) // 列
    {
        printf("%5d",b[i][j]);
    }
    printf("\n"); // 换行
}
```

例 6.4 将二维数组行列互换, 存到另一个二维数组中。

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```
#define M 2
#define N 3
int a[M][N]={1,2,3},{4,5,6}};
int b[N][M],i,j;
```

```
for(i=0;i<M;i++) // 行
{
    for(j=0;j<N;j++) // 列
    {
        printf("%5d",a[i][j]);
        b[j][i]=a[i][j];
    }
    printf("\n"); // 换行
}
```

```
for(i=0;i<N;i++) // 行
{
    for(j=0;j<M;j++) // 列
    {
        printf("%5d",b[i][j]);
    }
    printf("\n"); // 换行
}
```

例 6.4 将二维数组行列互换, 存到另一个二维数组中。

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```
#define M 2
#define N 3
int a[M][N]={1,2,3},{4,5,6}};
int b[N][M],i,j;
```

```
for(i=0;i<M;i++) // 行
{
    for(j=0;j<N;j++) // 列
    {
        printf("%5d",a[i][j]);
        b[j][i]=a[i][j];
    }
    printf("\n"); // 换行
}
```

```
for(i=0;i<N;i++) // 行
{
    for(j=0;j<M;j++) // 列
    {
        printf("%5d",b[i][j]);
    }
    printf("\n"); // 换行
}
```

例 6.4 将二维数组行列互换, 存到另一个二维数组中。

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \Rightarrow b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```
#define M 2
#define N 3
int a[M][N]={1,2,3},{4,5,6}};
int b[N][M],i,j;
```

```
for(i=0;i<M;i++) // 行
{
    for(j=0;j<N;j++) // 列
    {
        printf("%5d",a[i][j]);
        b[j][i]=a[i][j];
    }
    printf("\n"); // 换行
}
```

```
for(i=0;i<N;i++) // 行
{
    for(j=0;j<M;j++) // 列
    {
        printf("%5d",b[i][j]);
    }
    printf("\n"); // 换行
}
```

例 6.5 求二维数组中的最大元素及其行号和列号。

```
#define M 2
#define N 3

int a[M][N]={ {1,2,3}, {4,5,6} }, i, j, max, row=0, col=0;
max=a[0][0]; // 首先认为a[0][0]最大, 这是max最好的初值
for(i=0; i<M; i++) // 行
{
    for(j=0; j<N; j++) // 列
    {
        if(a[i][j]>max)
        {
            max=a[i][j]; row=i; col=j;
        }
    }
}

printf("max=%d, row=%d, col=%d\n", max, row, col);
```


例: 使方阵(行数=列数) 右对角线元素置 1, 其它元素为 0.

$i \setminus j$ 0 1 2 3 左对角线 $i+j=3$

0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3

右对角线 $i=j$

```
#define M 10
int a[M][M], i, j;
for(i=0; i<M; i++) // 行
{
    for(j=0; j<M; j++) // 列
    {
        if(i==j) a[i][j]=1;
        else a[i][j]=0;
    }
}
```

```
// 输出
for(i=0; i<M; i++) // 行
{
    for(j=0; j<M; j++) // 列
        printf("%5d", a[i][j]);
    printf("\n"); // 换行
}
```

方阵 (行数 = 列数) 的上下三角形

$i \backslash j$	0	1	2	3
0	0,0	0,1	0,2	0,3
1	1,0	1,1	1,2	1,3
2	2,0	2,1	2,2	2,3
3	3,0	3,1	3,2	3,3

下三角形 ($j < i$)

上三角形 ($j > i$)

对角线 ($j = i$)

```
#define M 10
int a[M][M], i, j;
for (i=0; i<M; i++) // 行
{
    for (j=0; j<M; j++) // 列
    {
        // 下三角形 (含对角线) 元素置1
        if (j<=i) a[i][j]=1;
        else a[i][j]=0; // 上三角形元素置0
    }
}
```

例: 使方阵(行数=列数) 下三角阵(含对角线) 元素置 1, 其它元素为 0.

```
#define M 10

int a[M][M],i,j;

for(i=0;i<M;i++) // 行
    for(j=0;j<M;j++) // 列
    {
        //下三角形(含对角线)
        if(j<=i) a[i][j]=1;
        else a[i][j]=0;
    }

printf("下三角阵\n");

for(i=0;i<M;i++) // 行
{
    for(j=0;j<=i;j++) // 列
        printf("%5d", a[i][j]);
    printf("\n"); // 换行
}
```

```
printf("上三角阵\n");
for(i=0;i<M;i++) // 行
{
    for(j=0;j<M;j++) // 列
        if(j>=i) printf("%5d", a[i][j]);
        else printf("%5c", ' ');
    printf("\n"); // 换行
}
```

```
下三角阵
1
1 1
1 1 1
1 1 1 1
1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1

上三角阵
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
  1 1 0 0 0 0 0 0 0 0
    1 1 0 0 0 0 0 0 0 0
      1 1 0 0 0 0 0 0 0 0
        1 1 0 0 0 0 0 0 0 0
          1 1 0 0 0 0 0 0 0 0
            1 1 0 0 0 0 0 0 0 0
              1 1 0 0 0 0 0 0 0 0
                1 1 0 0 0 0 0 0 0 0
```

用字符数组表示字符串 char s[81]

字符数组是用来存放字符数据的数组是字符数组。字符数组中的一个元素存放一个字符。

两种方式初始化字符数组

```
// s[0]='a', s[1]='b', s[2]='c', s[3]='d', s[4]以后的字符未赋值
char s[81]={'a','b','c','d'}; // 4个有效字符
```

```
// 以'\0'结尾的字符串是C语言表示字符串的标准形式。
// str[0]='a', str[1]='b', str[2]='c', str[3]='d', str[4]='\0'
// 自动追加'\0', 表示字符串结束。特别注意, 字符串结尾字符'\0'
char str[]="abcd"; // 4个有效字符, 还有末尾str[4]='\0'
```

使用 printf 函数输出字符串

```
// s[0]='a', s[1]='b', s[2]='c', s[3]='d', s[4]以后的字符未赋值
char s[81]={'a','b','c','d'}; // 4个有效字符
for(i=0;i<4;i++)
    printf("%c\t",s[i]); // a b c d

// str[0]='a', str[1]='b', str[2]='c', str[3]='d', str[4]='\0'
// 自动追加'\0', 表示字符串结束
char str[]="abcd"; // 4个有效字符, 还有末尾str[4]='\0'
for(i=0;a[i]!='\0';i++)
    printf("%c\t",str[i]); // a b c d

// 或使用格式描述符%s, 输出以'\0'结尾的字符串
printf("%s\n",str); // abcd, 输出'\0'以前的字符

printf("%s\n",s); // 错误, 由于s不是以'\0'结尾的字符串
s[4]='\0'; // 使字符串以'\0'结尾, 就可以使用上一句正常输出了
```

使用 scanf 函数输入字符串

```
char a[81],b[81],c[81];
```

```
// 遇空格或回车结束, 自动追加字符串结束字符'\0'
```

```
scanf("%s",a); // 注意字符数组前没有取地址符号'&'
```

```
// 例如输入: abc回车, 则a[0]='a',a[1]='b',a[2]='c',a[3]='\0'
```

```
printf("%s\n",a); // abc
```

```
// 以空格隔开, 输入3个字符串。自动追加字符串结束字符'\0'
```

```
scanf("%s%s%s",a,b,c);
```

```
// 例如输入: How are You回车
```

```
printf("%s,%s,%s\n",a,b,c); // How,are,You
```

推荐使用 gets 函数和 puts 函数输入输出字符串

```
char a[81]; // 定义字符数组的长度要足够大，以容纳最后一个字符'\0'
```

```
// 可以接收带空格的字符串，遇回车结束，自动追加字符串结束字符'\0'
```

```
gets(a);
```

```
// 例如输入: ab cd回车
```

```
// 遇字符串结束字符'\0'，换行退出
```

```
puts(a); // ab cd
```

字符串处理函数 #include<string.h>

- 使用字符串处理库函数, 必须包含头文件 #include<string.h>
- 要求字符串必须以 '\0' 结尾

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[81]="ab"; // 自动追加s1[2]='\0'
    char s2[81]="cdef"; // 自动追加s2[4]='\0'

    // 连接两个字符串, 结果被放入s1中, s1数组要足够大, 能够容纳两个字符串连接后的长度
    // +1, 多一个字符长度是留给'\0'使用
    strcat(s1,s2);

    puts(s1); // abcdef, 最后一个字符是'\0'
    puts(s2); // cdef, s2保持不变, 最后一个字符是'\0'
    return 0;
}
```


连接和复制字符串 strcat(s1,s2); strcpy(s1,s2); strncpy(s1,s2,n)

```
char s1[81]="ab"; // 自动追加s1[2]='\0'
char s2[81]="cdef"; // 自动追加s2[4]='\0'
// 连接两个字符串，结果被放入s1中，s1数组要足够大，能够容纳两个字符串连接后的长度
// +1，多一个字符长度是留给'\0'使用
strcat(s1,s2);
puts(s1); // abcdef, 最后一个字符是'\0'
puts(s2); // cdef, s2保持不变，最后一个字符是'\0'
// 把s2复制给s1，因此s1数组也要足够大
strcpy(s1,s2); // s1=s2; 是错误的，因为数组名表示地址常量
puts(s1); // cdef, 最后一个字符是'\0'
puts(s2); // cdef, s2保持不变，最后一个字符是'\0'
strncpy(s1,"1234",2); // 复制s2的前n个字符给s1,覆盖s1相应位置的字符，n要小于
// s1的数组长度
puts(s1); // 12ef
```

字符串比较 strcmp(s1,s2); strncmp(s1,s2,n)

```
char s1[81]="ab"; // 自动追加s1[2]='\0'
char s2[81]="cdef"; // 自动追加s2[4]='\0'
// 从左到右逐字符比较串s1和s2, 大于返回1, 小于返回-1, 同时到达'\0' 返回0表示相等
printf("%d",strcmp(s1,s2)); // -1
if(strcmp(s1,s2)==-1) printf("s1 < s2\n");
printf("%d\n",strcmp("abad","abld")); // 1
printf("%d\n",strcmp("1234","1234")); // 0
printf("%d\n",strcmp("12","1234")); // -1
printf("%d\n",strcmp("1234","12")); // 1
// 仅比较前n个字符
printf("%d\n",strncmp("1234","12",2)); // 0
```

获取字符串的长度 strlen(s)

```
char s1[81]="ab"; // 自动追加s1[2]='\0'
char s2[81]="cdef"; // 自动追加s2[4]='\0'
int i;
// 获取字符串的长度, 不包括字符串结尾字符'\0'
printf("%d,%d\n",strlen(s1), strlen(s2)); // 2,4
for(i=0;i<strlen(s1);i++) printf("%c ",s[i]); // a b
// 相当于下面程序段计算出的字符串长度len
int len=0;
for(i=0; s1[i]!='\0'; i++) len++;
printf("\n字符串s1的长度=%d\n", len);
```

大小写转换 strlwr(s);strupr(s)

```
char s1[81]="abCD12"; // 自动追加s1[2]='\0'
char s2[81]="cD12ef"; // 自动追加s2[4]='\0'
int i;
strlwr(s1); // 大写转小写
puts(s1); // abcd12
```

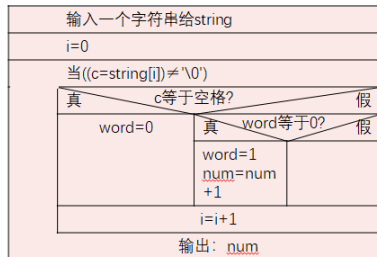
```
strupr(s2); // 小写转大写
puts(s2); // CD12EF
```

// 小写转大写，等效于下面程序段

```
for(i=0;s2[i]!='\0';i++)
    if(s2[i]>='a' && s2[i]<='z') s2[i]-=32; // s2[i]=s2[i]-32
```

例 6.8 输入一行字符, 统计其中有多少个单词, 单词之间用空格 (可能多个) 分隔开。

```
// 定义变量
char string[81]; // 用于存放字符串。
int i; // 计数器, 用于遍历字符串中的每个字符。
int word=0; // 用于判断是否开始了一个新单词的标志。若word=0表示未出现新单词, 如
            出现了新单词, 就把word置成1。
int num=0; // 用于统计单词数。
```



word = 0; num = 0;

$w=0$	$w=0$	$w=0$
□ □	□	□
$w=0$		
H	A	Y
$w=1$	$w=1$	$w=1$
$n++$	$n++$	$n++$

ow re ou

解: 例 6.8 输入一行字符, 统计其中有多少个单词, 单词之间用空格(可能多个) 分隔开。

```
char string[81];
int i,num=0,word=0;
char c;
gets(string); //输入一个字符串给字符数组string
```

```
for(i=0;(c=string[i])!='\0';i++) //只要字符不是'\0'就循环,条件表达式:第i个字符
    赋值给c, 并且c!='\0'
```

```
{
    if(c==' ') word=0; //若是空格字符, 使word置0
    else if(word==0) //如果不是空格字符且word原值为0
    {
        word=1; //使word置1
        num++; //num累加1, 表示增加一个单词
    }
}
```

```
printf("%d words in this line.\n",num); //输出单词数
```

word = 0; num = 0;

\square \square *H ow* \square *A re* \square *Y ou*
 $w=0$ \downarrow $w=0$ \downarrow $w=0$ \downarrow
 $w=1$ $w=1$ $w=1$
 $n++$ $n++$ $n++$

例 6.9 有 3 个字符串, 要求找出其中“最大”者。

```
char str[3][20]; //定义二维字符数组, 存放3个字符串。【重点学习】
char string[20]; //定义一维字符数组, 作为交换字符串时的临时字符数组
int i;
for(i=0;i<3;i++)
    gets(str[i]); //读入3个字符串, 分别给str[0],str[1],str[2]

if(strcmp(str[0],str[1])>0) //若str[0]大于str[1]
    strcpy(string,str[0]); //把str[0]的字符串赋给字符数组string
else //若str[0]小于等于str[1]
    strcpy(string,str[1]); //把str[1]的字符串赋给字符数组string
if(strcmp(str[2],string)>0) //若str[2]大于string
    strcpy(string,str[2]); //把str[2]的字符串赋给字符数组string
printf("\nthe largest string is:%s\n",string); //输出string
```

例 6.9 有 3 个字符串, 要求找出其中“最大”者。

```
char str[3][20]; //定义二维字符数组, 存放3个字符串。【重点学习】
char string[20]; //定义一维字符数组, 作为交换字符串时的临时字符数组
int i;
for(i=0;i<3;i++)
    gets(str[i]); //读入3个字符串, 分别给str[0],str[1],str[2]

if(strcmp(str[0],str[1])>0) //若str[0]大于str[1]
    strcpy(string,str[0]); //把str[0]的字符串赋给字符数组string
else //若str[0]小于等于str[1]
    strcpy(string,str[1]); //把str[1]的字符串赋给字符数组string
if(strcmp(str[2],string)>0) //若str[2]大于string
    strcpy(string,str[2]); //把str[2]的字符串赋给字符数组string
printf("\nthe largest string is:\n%s\n",string); //输出string
```


另解: 例 6.9 有 3 个字符串, 要求找出其中“最大”者。

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[20],max[20]; int i;
    for(i=0;i<3;i++)
    {
        if(i==0) gets(max); // 常用技巧: 初始, 第一个字符串就是最大者。
        else // 随后的字符串与max比较
        {
            gets(s);
            if(strcmp(s,max)>0) strcpy(max,s); // 如果该字符串比max大, 替换之。
        }
    }
    printf("\nthe largest string is:\n%s\n",max);
    return 0;
}
```

例: 矩阵乘法

$$c[M][N], a[M][P], b[P][N];$$

$$c[i][j] = \sum_{k=0}^{P-1} a[i][k]b[k][j] = a[i][0]b[0][j] + a[i][1]b[1][j] + \cdots + a[i][P-1]b[P-1][j]$$

$$a = \begin{bmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \end{bmatrix} \quad b = \begin{bmatrix} b[0][0] & b[0][1] \\ b[1][0] & b[1][1] \\ b[2][0] & b[2][1] \end{bmatrix}$$

$$c = \begin{bmatrix} a[0][0]b[0][0] + a[0][1]b[1][0] + a[0][2]b[2][0] & a[0][0]b[0][1] + a[0][1]b[1][1] + a[0][2]b[2][1] \\ a[1][0]b[0][0] + a[1][1]b[1][0] + a[1][2]b[2][0] & a[1][0]b[0][1] + a[1][1]b[1][1] + a[1][2]b[2][1] \end{bmatrix}$$

$c[i][j]$ = **a** 的第 **i** 行各列 (**k**) 和 **b** 的第 **j** 列各行 (**k**) 的乘积累加. $k = 0 \rightarrow (P-1)$

例: 矩阵乘法

$$c[M][N], a[M][P], b[P][N];$$

$$c[i][j] = \sum_{k=0}^{P-1} a[i][k]b[k][j] = a[i][0]b[0][j] + a[i][1]b[1][j] + \cdots + a[i][P-1]b[P-1][j]$$

$$a = \begin{bmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \end{bmatrix} \quad b = \begin{bmatrix} b[0][0] & b[0][1] \\ b[1][0] & b[1][1] \\ b[2][0] & b[2][1] \end{bmatrix}$$

$$c = \begin{bmatrix} a[0][0]b[0][0] + a[0][1]b[1][0] + a[0][2]b[2][0] & a[0][0]b[0][1] + a[0][1]b[1][1] + a[0][2]b[2][1] \\ a[1][0]b[0][0] + a[1][1]b[1][0] + a[1][2]b[2][0] & a[1][0]b[0][1] + a[1][1]b[1][1] + a[1][2]b[2][1] \end{bmatrix}$$

$c[i][j]=a$ 的第 i 行各列 (k) 和 b 的第 j 列各行 (k) 的乘积累加. $k = 0 \rightarrow (P - 1)$

解: 矩阵乘法

```
#define M 100 // 估计的最大值
#define P 50
#define N 100

int c[M][N], a[M][P]={{1,2,4}}, b[P][N]={{2,1,3},{7,9,10},{5,7,1}}; // c
    ={{36,47,27}}

int m=4,n=2,p=3; // M,N,P实际大小
int i,j,k; // i,j,k分别是m,n,p的循环变量
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        c[i][j]=0;
        for(k=0;k<p;k++)
            c[i][j]+=a[i][k]*b[k][j]; // a的第i行各列(k)和b的第j列各行(k)的乘积累加
        printf("c[%d][%d]=%d\n", i, j, c[i][j]); // 输出测试
    }
}
```

曾经的测试用例

```
//int c[M][N],a[M][P]={ {1,2,4} },b[P][N]={ {2,1,3},{7,9,10},{5,7,1} }; //
    c={36,47,27}

//int m=1,n=3,p=3; // M,N,P实际大小

//int c[M][N],a[M][P]={ {2,1},{4,3} },b[P][N]={ {1,2},{1,0} }; // c
    ={{3,4},{7,8}}

//int m=2,n=2,p=2; // M,N,P实际大小

//int c[M][N],a[M][P]={ {1,0,2},{-1,3,1} },b[P][N]={ {3,1},{2,1},{1,0} };
    // c={{5,1},{4,2}}

//int m=2,n=2,p=3; // M,N,P实际大小

int c[M][N],a[M][P]={ {5,2,4},{3,8,2},{6,0,4},{0,1,6} },b[P][N
    ]={ {2,4},{1,3},{3,2} }; // c={{24,34},{20,40},{24,32},{19,15}}

int m=4,n=2,p=3; // M,N,P实际大小

int i,j,k; //i,j,k分别是m,n,p的循环变量

// 添加输入语句

scanf("%d%d%d\n", &m,&n,&p);

for(i=0;i<m;i++)
    for(k=0;k<p;k++) scanf("%d",&a[i][k]);

for(k=0;k<p;k++)
    for(j=0;j<n;j++) scanf("%d",&b[k][j]);

...
```

函数

为什么使用函数(1)

函数调用

```
#include<stdio.h>
#include<math.h>
int main() // 主函数
{
    int a;
    // 库函数调用
    scanf("%d",&a);
    a=(int) fabs((double) a);
    printf("%d\n",a);
    return 0;
}
```

函数定义

```
int scanf(char format[], args, ...)
{ ...;
    return 整型值;
}

int printf(char format[], args, ...)
{ ...;
    return 整型值;
}

double fabs(double a)
{ // 模拟代码, 没有考虑精度
    if(a<0) return -a;
    else return a;
}
```


为什么使用函数 (2)

函数调用

```
#include<stdio.h>
#include<string.h>
int main() // 主函数
{
    int a;
    char s1[81],char s2[81]="1234";
    // 库函数调用
    strcpy(s1,s2);
    a=strcmp(s1,s2);
    printf("%d ",a);
    printf("%d\n",strlen(s1));
    return 0;
}
```

函数定义

```
char[] strcpy(char s1[],char s2[])
{ ...; return s1;
}

int strcmp(char s1[],char s2[])
{ ...; return 整型值; // 1,-1,0
}

int strlen(char s[])
{
    int len=0;
    while(s[len]!='\0') len++;
    return len;
}
```

利用函数进行功能分解 — 简化程序设计的有效手段

功能分解, 函数调用

```
#include<stdio.h>
#include<math.h>
int add(int a,int b); // 函数原型声明
void output(double a);
int main() // 主函数
{
    int a=10,b=20,c;
    //传递参数a,b的值, 计算结果赋值给变量c
    c=add(a,b); // 函数调用
    output(10.5*c); // 函数调用
    return 0;
}
```

分解功能实现, 函数定义

```
// 通过参数a,b的值, 进行相关计算, 返回整
// 型数据给调用者
int add(int a,int b)
{ return a+b; } // 返回整型值

// 通过参数a的值, 进行相关计算, 不需要返
// 回数据
void output(double a)
{
    printf("%lf\n", sqrt(a)); // 函数
    // 中可调用别的函数
}
```

使用函数进行程序设计的优点

- 1 使用函数可使程序清晰、精炼、简单、灵活。
- 2 函数就是功能。每一个函数用来实现一个特定的功能。函数名应反映其代表的功能。
- 3 在设计较大程序时,往往把它分为若干个程序模块,每一个模块包括一个或多个函数,每个函数实现一个特定的功能。
- 4 一个 C 程序可由一个主函数和若干个其他函数构成。由主函数调用其他函数,其他函数也可以互相调用。

函数三要素: 函数原型声明, 函数定义, 函数调用

函数原型声明, 函数调用

```
#include<stdio.h> // 库函数原型声明
// 函数原型声明, 使编译器认识这个函数, 如
// 果函数定义在调用之前, 可省略声明
int add(int a,int b);
void output(double a);
int main() // 主函数
{
    int a=10,b=20,c;
    // 函数调用, 执行函数功能
    c=add(a,b); // c接收函数返回值
    output(10.5*c); // 无返回值
    return 0;
}
```

函数定义, 定义函数功能

```
// 通过参数a,b的值, 进行相关计算, 返回整
// 型数据给调用者
int add(int a,int b)
{ return a+b; } // 返回整型值

// 通过参数a的值, 进行相关计算, 不需要返
// 回数据
void output(double a)
{
    printf("%lf\n", sqrt(a)); // 函数
    // 中可调用别的函数
}
```

函数定义: `int add(int a,int b){ }`

函数定义: 返回类型 函数名 (参数类型 参数,...) {...;}

```
int add(int a, int b)
{ ...;
    return 整型值; // 必须含return语句, 函数执行结束, 并将返回值返回给调用者
}

double fun(void) // 无参函数, 等效 double fun()
{ ...;
    return 双精度值; // 必须含return语句, 函数执行结束, 并将返回值返回给调用者
}

void output(double a) // 无返回值函数
{ ...;
    return; // 可选return语句 (注意没有表达式), 仅表示函数执行结束
}
```

实参和形参间的数据传递 (值传递)

```
#include<stdio.h> // 库函数原型声明
// 函数原型声明, 使编译器认识这个函数
int max(int x,int y);
int main() // 主函数
{
    int a=10,b=20,c;
    scanf("%d%d",&a,&b);
    // 把此处的a,b值拷贝给函数形式参数x,y
    c=max(a,b); // 实际参数
    printf("较大者=%d\n",c); //与下一句等效
    printf("较大者=%d\n",max(a,b));
    return 0;
}
```

```
// 定义函数求形参x,y中的较大者并返回
// 给调用者
int max(int x,int y) // 形式参数
{
    int z;
    z=x>y ? x : y;
    return z;
}
```

在调用函数过程中发生的实参与形参间的数据传递称为“虚实结合”。

实参和形参间的数据传递 (地址传递)

例: 将数组 a 中 n 个整数按相反顺序存放。

```
#include<stdio.h>

void inv(int x[],int n); // 要求x是地址传递

#define N 100

int main()
{
    int i, n, a[N]={1,2,3,4,5,6,7,8,9,10};
    scanf("%d",&n);
    for(i=0; i<n; i++) scanf("%d",&a[i]);
    // 把实参a数组的地址拷贝给形参x, n的值拷贝给形
    // 式参数n
    inv(a,n); // 实参a数组的内容被改变
    for(i=0; i<n; i++) printf("%d ",a[i]);
    return 0;
}
```

```
// 倒置数组x的内容, n是x的长度
// 要求x是地址传递

void inv(int x[],int n)
{
    int i,temp,m=(n-1)/2;
    //以中间元素为界, 前后元素交换
    for(i=0; i<=m; i++)
    {
        temp=x[i];
        x[i]=x[n-1-i];
        x[n-1-i]=temp;
    }
    return; //可选, 函数执行完毕
}
```

值传递与地址传递的不同点

- 值传递, 对形参值的改变不会引起实参值的改变。
- 地址传递虽然不能改变实参的地址, 但是对地址指向内容的改变会引起实参指向内容的改变。
- 数组名表示数组元素在内存中的首地址, 数组元素是连续存放的。
- 用数组名作为参数传递, 就是地址传递。在函数内部对数组元素的改变, 就是改变实参数组元素的值。

```
// a是地址传递, n是值传递
void fun(double a[], int n)
{
    // 改变地址a指向的内容, 就是改变实
    // 参数组的元素值
    a[1]=20.5;
    // n的改变不会影响实参的值
    n=30;
}
```

```
int main()
{
    double a[2]={0.8,0.3};
    int n=2;
    // 实参a的地址拷贝给形参a, 实参n的值拷贝给形参n
    fun(a,n);
    printf("%d,%lf,%lf\n",n,a[0],a[1]); //
        2,0.8,20.5
}
```


数组名作为函数参数的注意事项

```
#define M 100 // 估计的数组的第一维长度
#define N 100 // 估计的数组的第二维长度
// 函数定义时省略第一维大小
// a[]表示a接受地址传递，以区别于值传递
void fun(int a[],int n)
{
    int i;
    for(i=0;i<n;i++)
        printf("%d "a[i]);
    printf("\n");
}
// 函数调用
int x[M]={1, 2, 3, 4, 5};
fun(x,5); //调用时仅用数组名传递x的地址
```

```
// 函数定义时省略第一维大小，第二维不能省略
void fun(int a[][N],int m,int n)
{
    int i;
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
            printf("%d "a[i][j]);
        printf("\n");
    }
}
// 函数调用
int x[M][N]={ {1,2},{3,4},{5,6},
               {7,8},{9,10} };
fun(x,5,2); //调用时仅用数组名传递x的地址
```

函数调用时数据类型的隐式转换和显式转换

```
int a=2,b=3,c;
```

```
// pow函数原型: double pow(double x,double y);
```

```
// 编译器自动把实际参数a,b的int类型“隐式”转换为形式参数要求的double类型
```

```
// 计算 $a^b$ 的结果是double类型, 赋值语句隐式转换为int, 但是会引起警告信息
```

```
c=pow(a,b);
```

```
// 将函数的返回值强制转换为int, 不会有警告信息
```

```
c=(int)pow(a,b);
```

```
// Visual Studio等编译器要求显式强制转换
```

```
c=(int)pow((double)a,(double)b);
```

整数 pow 函数问题

`double pow(double x, double y)`; 数学库函数真对双精度浮点数设计。用它计算整数 x^y 会有复杂的精度问题, 因此整数运算尽量不要用此函数。

```
#include<stdio.h>
#include<math.h> // 数学库函数
int main()
{
    int x=2,y=3;

    // 有些编译器会输出99,124, 因为转换前是99.999, 124.999
    printf("%d,%d\n", (int)pow(10,x), (int)pow(5,y));

    return 0;
}
```

自定义整数 pow 函数

```
#include<stdio.h>

// 自定义整数pow函数, 返回x^y

int intpow(int x, int y)
{
    int i,p=1;
    // 实现0^y = 0
    if (x==0) return 0;

    for(i=0;i<y;i++)
        p=p*x;

    return p;//包含了0^0 = 1
}
```

```
int main()
{
    int x=2,y=3;

    printf("%d,%d\n",intpow(10,x),intpow(5,y));
    // 100,125
    printf("%d,%d\n",intpow(10,2),intpow(5,3));
    // 100,125
    printf("%d,%d\n",intpow(0,2),intpow(0,0));
    // 0,1

    return 0;
}
```

例: PM2.5

给出一组 PM2.5 数据,按以下分级标准统计各级天气的天数,并计算出 PM2.5 平均值。

PM2.5 分级标准为:一级优 ($0 \leq PM2.5 \leq 50$); 二级良 ($51 \leq PM2.5 \leq 100$); 三级轻度污染 ($101 \leq PM2.5 \leq 150$); 四级中度污染 ($151 \leq PM2.5 \leq 200$); 五级重度污染 ($201 \leq PM2.5 \leq 300$); 六级严重污染 ($PM2.5 > 300$)。

输入说明,输入分为两行,

第一行是一个整数 n 表示天数 ($1 < n \leq 100$)

第二行为 n 个非负整数 P_i ($0 \leq P_i \leq 1000$) 表示每天的 PM2.5 值,整数之间用空格分隔。

输出说明,输出两行数据,

第一行为 PM2.5 平均值,结果保留 2 位小数;

第二行依次输出一级优,二级良,三级轻度污染,四级中度污染,五级重度污染,六级严重污染的天数。

例: PM2.5(非模块化设计)

```
#include <stdio.h>

int main()
{
    int i = 0, n, pm25, day[6] = {0, 0, 0, 0, 0, 0},
        sum = 0;
    scanf("%d", &n);
    while(i < n) {
        scanf("%d", &pm25);
        sum += pm25;
        if(pm25 >= 0 && pm25 <= 50 ) day[0]++;
        else if(pm25 >= 51 && pm25 <= 100 ) day[1]++;
        else if(pm25 >= 101 && pm25 <= 150 ) day[2]++;
        else if(pm25 >= 151 && pm25 <= 200 ) day[3]++;
        else if(pm25 >= 201 && pm25 <= 300 ) day[4]++;
        else day[5]++;
        i++;
    }
```

```
printf("%.2f\n", (float) sum/n);
for(i = 0; i < 6; i++)
    if(i == 5) printf("%d\n", day[i]);
    else printf("%d ", day[i]);
return 0;
}
```

例: PM2.5(模块化设计, 主程序)

```
#include <stdio.h>

// 函数声明

float haze(int pm25[], int day[], int
           n, int m);

#define N 1000 // 估计的最大值

int main()
{
    int i = 0, n, pm25[N], day[6];
    scanf("%d", &n);
    while(i < n)
    {
        scanf("%d", &pm25[i]);
        i++;
    }
}
```

```
// 函数调用, 并输出函数中计算的平均值
printf("%.2f\n", haze(pm25, day, n, 6));
// 输出函数中计算的数组元素值
for(i = 0; i < 6; i++)
    if(i == 5) printf("%d\n", day[i]);
    else printf("%d ", day[i]);
return 0;
```

例: PM2.5(模块化设计, 雾霾统计信息)

```
float haze(int pm25[], int day[], int n, int m)
{
    int i=0, sum=0;
    // 初始化day数组
    for(i=0; i<m; i++) day[i]=0;
    while(i < n)
    {
        sum += pm25[i];
        if(pm25[i] >= 0 && pm25[i] <= 50 ) day[0]++;
        else if(pm25[i] >= 51 && pm25[i] <= 100 ) day[1]++;
        else if(pm25[i] >= 101 && pm25[i] <= 150 ) day[2]++;
        else if(pm25[i] >= 151 && pm25[i] <= 200 ) day[3]++;
        else if(pm25[i] >= 201 && pm25[i] <= 300 ) day[4]++;
        else day[5]++;
        i++;
    }
    return (float)sum/n;
}
```

函数参数说明: pm25[]: PM2.5 值,
day[]: 不同标准对应的统计天数; n 是
pm25 数组的长度, m 是数组 day 的长
度, 返回 PM2.5 平均值.

要点:

用数组做参数, 可获得函数计算结果的
多值传递;

而 return 仅返回一个值。

例: 数字排序 (主程序)

给定 n 个整数, 请计算每个整数各位数字和, 按各位数字和从大到小的顺序输出。

```
#include <stdio.h>

#define N 1000 // 估计的数组大小

int bitsSum(int a); // 计算整数a的各位之和
void sort(int a[], int n); // 从大到小排序
void output(int a[], int n); // 输出

int main()
{
    int i, n; // n是实际数组长度
    int num[N], sum[N]; // num表示N个整数, sum存放对应整数的各位数字的和
    scanf("%d", &n);
    for(i=0; i<n; i++) { scanf("%d", &num[i]); sum[i]=bitsSum(num[i]); }
    sort(sum, n); // 从大到小排序
    output(sum, n); // 输出
    return 0;
}
```

例: 数字排序 (子函数: 计算整数的各位之和, 输出)

// 计算整数a的各位之和

```
int bitsSum(int a)
```

```
{
```

```
    int sum=0;
```

```
    while(a)
```

```
    {
```

```
        sum += a%10;
```

```
        a /= 10;
```

```
    }
```

```
    return sum;
```

```
}
```

// 输出, n是数组长度

```
void output(int a[], int n)
```

```
{
```

```
    int i;
```

```
    for(i=0;i<n;i++)
```

```
        printf("%d ",a[i]);
```

```
    printf("\n");
```

```
}
```

例: 数字排序 (排序子函数—方法 1: 冒泡排序)

```
void sort(int a[], int n) // 从大到小排序, 冒泡, n是数组长度
{
    int i,j,flag,temp;
    for(j = 1; j <= n-1; j++) // 第j趟比较
    {
        flag=0;
        for(i = 0; i < n - j; i++) // 相邻两数比较
        {
            if (a[i] < a[i+1]) // 交换
            {
                temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
                flag=1;
            }
        }
        if(!flag) break;
    }
}
```

选择法排序

从大到小**选择法排序**: 先将 n 个数中最大的数与 $a[0]$ 对换, $a[0]$ 就是最大的数; 再将 $a[1] \sim a[n-1]$ 中最大的数与 $a[1]$ 对换, \dots , 每比较一轮, 找出一个未经排序的数中最大的一个。共比较 $n-1$ 轮。

```
int a[]={5,8,2,9,3}; // 待排序的数
```

```
int j,k; // j: 外层循环次数, 共 $n-1$ 次, k: 每次循环选择的最大数下标
```

$a[0]$	5	9	9	9	9	4
$a[1]$	8	8	8	8	8	3
$a[2]$	2	2	2	2	5	2
$a[3]$	9	5	5	5	2	1
$a[4]$	3	3	3	3	3	0
	$j=0, k=j$		$j=1, k=j$		$j=2, k=j$	
	找出 $k=3$		找出 $k=1$		找出 $k=3$	
	交换 $a[j], a[k]$		$k=j$ 无需交换		交换 $a[j], a[k]$	
	交换 $a[j], a[k]$		交换 $a[j], a[k]$		交换 $a[j], a[k]$	

例: 数字排序 (排序子函数—方法 2: 选择法排序)

```
void sort(int a[], int n) // 从大到小排序, 选择
{
    int i, j, k, temp;
    for(j = 0; j <= n-2; j++) // 第j趟比较, 共n-1次循环
    {
        k=j; // 首先假设j就是未经排序的数中, 最大元素的下标
        for(i = j+1; i < n; i++) // 未经排序的数从j+1开始
        {
            if (a[i] > a[k]) k=i; // 如果第i个元素比第k个元素大, 置换k为i
        }
        if(k!=j) // 未经排序数中的最大元素与a[j]交换, a[j]及其以前元素是已经排好的数据
        {
            temp = a[j]; a[j] = a[k]; a[k] = temp;
        }
    }
}
```

例: 数字排序 (联动输出: 修改主程序)

给定 n 个整数, 请计算每个整数各位数字和, 按各位数字和从大到小的顺序输出。

要求联动输出 (整数跟随其数字和的排序次序输出): 整数 该整数的各位之和

```
#include <stdio.h>

#define N 1000 // 估计的数组大小

int bitsSum(int a); // 计算整数a的各位之和

void sort(int a[], int b[], int n); // a从大到小排序, b联动改变

void output(int a[], int b[], int n); // 输出a,b

int main()
{
    int i,n; // n是实际数组长度
    int num[N],sum[N]; // num表示N个整数, sum存放对应整数的各位数字的和
    scanf("%d",&n);
    for(i=0;i<n;i++) { scanf("%d",&num[i]); sum[i]=bitsSum(num[i]); }
    sort(sum,num,n); // sum从大到小排序,num联动改变
    output(num,sum,n); // 输出num,sum
    return 0;
}
```

例: 数字排序 (联动输出: 修改排序子函数)

```
void sort(int a[], int b[],int n) // a从大到小排序, b联动改变
{
    int i,j,flag,temp;
    for(j = 1; j <= n-1; j++) // 第j趟比较
    {
        flag=0;
        for(i = 0; i < n - j; i++) // 相邻两数比较
        {
            if (a[i] < a[i+1]) // 同时交换a和b
            {
                temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
                temp = b[i]; b[i] = b[i+1]; b[i+1] = temp;
                flag=1;
            }
        }
        if(!flag) break;
    }
}
```

例: 数字排序 (联动输出: 修改输出子函数)

```
// 输出a,b
void output(int a[], int b[], int n)
{
    int i;
    for(i=0;i<n;i++) printf("%d %d",a[i],b[i]);
    printf("\n");
}
```


例: 数字排序 (联动输出: 修改主程序, 二维数组)

给定 n 个整数, 请计算每个整数各位数字和, 按各位数字和从大到小的顺序输出。

要求联动输出 (整数跟随其数字和的排序次序输出): 整数 该整数的各位之和

```
#include <stdio.h>

#define N 1000 // 估计的数组大小

int bitsSum(int a); // 计算整数a的各位之和

void sort(int a[][2], int n); // a的第0列是整数, 第1列是其各位数字之和(排序列)
void output(int a[][2], int n); // 输出a, 第0列是整数, 第1列是其各位数字之和

int main()
{
    int i, n; // n是实际数组长度
    int num[N][2]; // 第0列表示整数, 第1列是其各位数字的和
    scanf("%d", &n);
    for(i=0; i<n; i++)
    { scanf("%d", &num[i][0]); num[i][1]=bitsSum(num[i][0]); }
    sort(num, n); // sum从大到小排序, num联动改变
    output(num, n); // 输出num
    return 0;
}
```

例: 数字排序 (联动输出: 修改排序子函数, 二维数组)

```
void sort(int a[][2],int n) // a的第0列是整数, 第1列是其各位数字之和 (排序列)
```

```
{
    int i,j,flag,temp;
    for(j = 1; j <= n-1; j++) // 第j趟比较
    {
        flag=0;
        for(i = 0; i < n - j; i++) // 相邻两数比较
        {
            if (a[i][1] < a[i+1][1]) // 同时交换a的第0列和第1列
            {
                temp = a[i][0]; a[i][0] = a[i+1][0]; a[i+1][0] = temp;
                temp = a[i][1]; a[i][1] = a[i+1][1]; a[i+1][1] = temp;
                flag=1;
            }
        }
        if(!flag) break;
    }
}
```

例: 数字排序 (联动输出: 修改输出子函数, 二维数组)

// 输出, a的第0列是整数, 第1列是其各位数字之和 (排序列)

```
void output(int a[][2], int n)
{
    int i;
    for(i=0;i<n;i++) printf("%d %d",a[i][0],a[i][1]);
    printf("\n");
}
```

模块化程序设计的优点

综上, 程序功能的变化, 仅修改相应子函数即可, 逻辑清晰。

因此, 利用函数进行功能分解是进行复杂程序设计的有效手段。

函数的嵌套调用

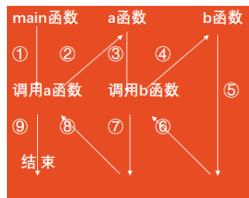
```
#include<stdio.h>

int a(); int b(); // 函数声明

int main()
{
    int c; // 与a()中的c无关
    c=a(); // 函数调用
    return 0;
}
```

```
int a()
{
    int c; // 与main()中的c无关
    ...;
    c=b(); // 函数调用
    ...;
    return c;
}

int b()
{
    ...;
    return 10;
}
```



例: 求 4 个整数中的最大者。

```
#include<stdio.h>

int max2(int x,int y);

int max4(int a,int b,int c,int d);

int main() // 主函数
{
    int a=b,c,d;
    scanf("%d%d%d%d",&a,&b,&c,&d);
    printf("较大者=%d\n", max4(a,b,c,d));
    printf("较大者=%d\n", max2(max2(a,b),
        max2(c,d))); // 等效
    return 0;
}
```

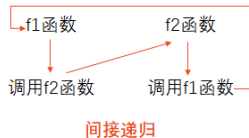
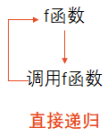
```
// 定义函数
int max2(int x,int y) // 形式参数
{
    int z;
    z=x>y ? x : y;
    return z;
}

int max4(int a,int b,int c,int d)
{
    return max2(max2(a,b),max2(c,d))
        ;
}
```

函数的递归调用

在调用一个函数的过程中又出现直接或间接地调用该函数本身,称为函数的递归调用。

```
// 递推公式:  $f(0)=0, f(1)=1,$   
//  $n>1: f(n)=n+f(n-1)$   
int f(int n)  
{  
    int sum;  
    if(n==0 || n==1) sum=n;  
    else sum=n+f(n-1);  
    return sum;  
}
```



切记不应出现无终止的递归调用。应是有限次数的,有终止的递归调用,这可以用 if 语句来控制,只有在某一条件成立时才继续执行递归调用;否则就不再继续。

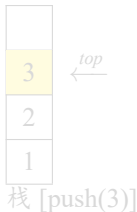
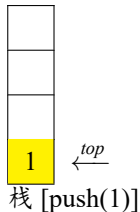
栈 — 数据结构, push 操作, 压入数据至栈顶

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



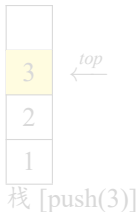
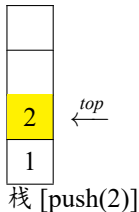
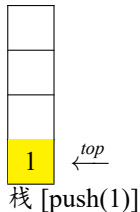
栈 — 数据结构, push 操作, 压入数据至栈顶

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



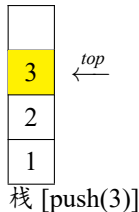
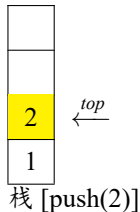
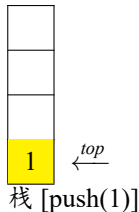
栈 — 数据结构, push 操作, 压入数据至栈顶

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



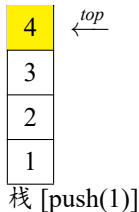
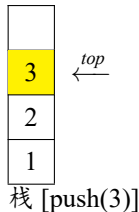
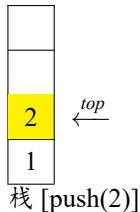
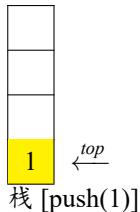
栈 — 数据结构, push 操作, 压入数据至栈顶

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



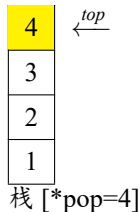
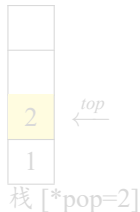
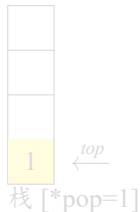
栈 — 数据结构, pop 操作, 弹出栈顶数据

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



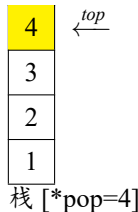
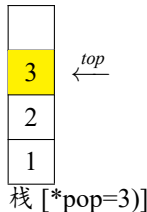
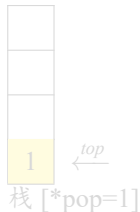
栈 — 数据结构, pop 操作, 弹出栈顶数据

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



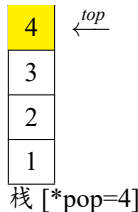
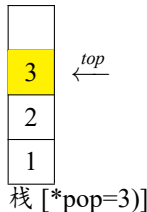
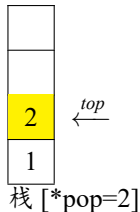
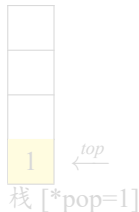
栈 — 数据结构, pop 操作, 弹出栈顶数据

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

第一个 push 的数据, 最后一个被 pop。



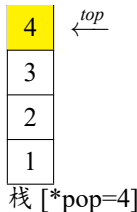
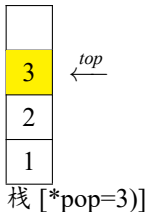
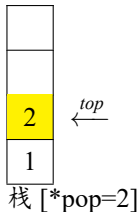
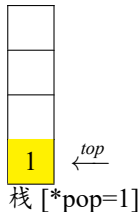
栈 — 数据结构, pop 操作, 弹出栈顶数据

系统内部自动维护一个称作“栈”的存储数据的空间。

栈是一种“先进后出 (FILO)= 后进先出 (LIFO)”的数据结构。

向栈中存储数据操作称作 push, 取出栈顶数据操作称作 pop。

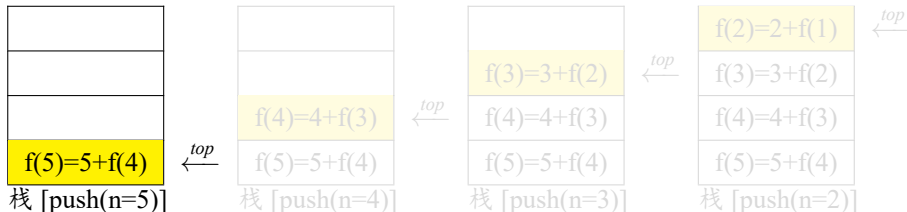
第一个 push 的数据, 最后一个被 pop。



递归调用过程分析 (push)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

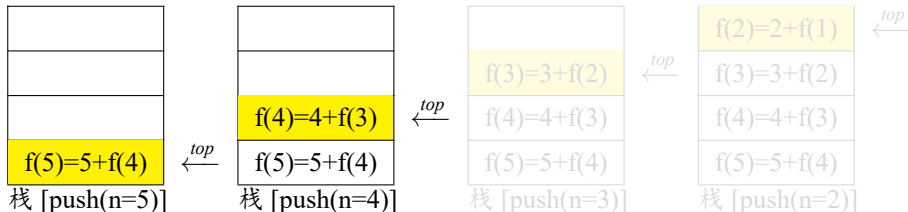
```
int f(int n)
{
    int sum;
    if (n==0 || n==1) sum=n;
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
}
```



递归调用过程分析 (push)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
{
    int sum;
    if (n==0 || n==1) sum=n;
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
}
```



递归调用过程分析 (push)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
```

```
{
```

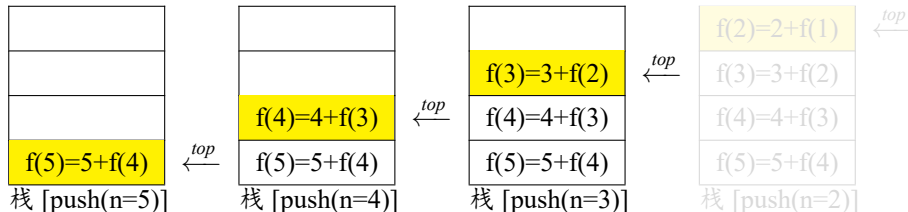
```
    int sum;
```

```
    if (n==0 || n==1) sum=n;
```

```
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
```

```
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

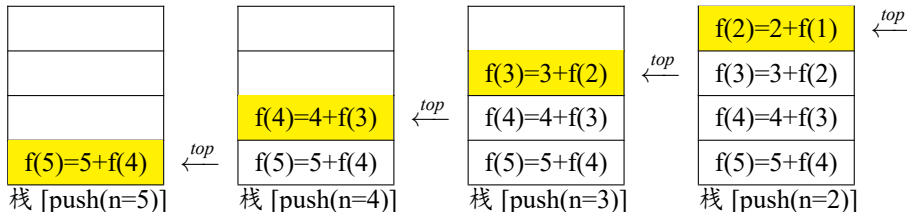
```
}
```



递归调用过程分析 (push)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
{
    int sum;
    if(n==0 || n==1) sum=n;
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来(push)
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
}
```



递归调用过程分析 (pop)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
```

```
{
```

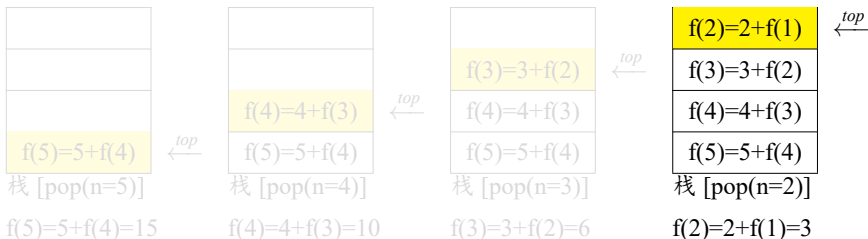
```
    int sum;
```

```
    if (n==0 || n==1) sum=n;
```

```
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
```

```
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

```
}
```



递归调用过程分析 (pop)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
```

```
{
```

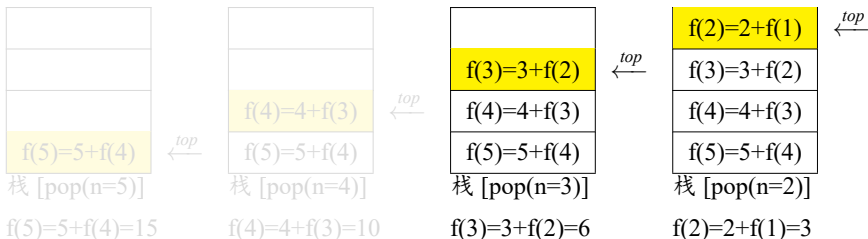
```
    int sum;
```

```
    if (n==0 || n==1) sum=n;
```

```
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
```

```
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

```
}
```



递归调用过程分析 (pop)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
```

```
{
```

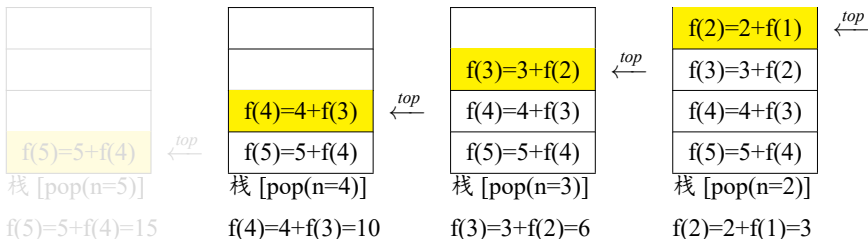
```
    int sum;
```

```
    if (n==0 || n==1) sum=n;
```

```
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
```

```
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

```
}
```



递归调用过程分析 (pop)

// 递推公式: $f(0)=0, f(1)=1, n>1: f(n)=n+f(n-1)$

```
int f(int n)
```

```
{
```

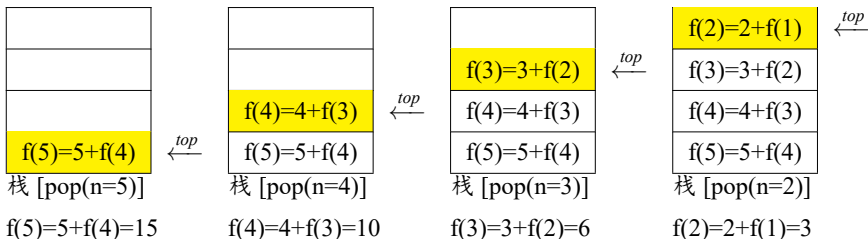
```
    int sum;
```

```
    if (n==0 || n==1) sum=n;
```

```
    else sum=n+f(n-1); // 未完成的计算用“栈”存储起来 (push)
```

```
    return sum; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

```
}
```



例: age(n)

有 5 个学生坐在一起,问第 5 个学生多少岁,他说比第 4 个学生大 2 岁。问第 4 个学生岁数,他说比第 3 个学生大 2 岁。问第 3 个学生,又说比第 2 个学生大 2 岁。问第 2 个学生,说比第 1 个学生大 2 岁。最后问第 1 个学生, he 说是 10 岁。请问第 5 个学生多大。

$$\text{第 } n \text{ 个学生年龄} \begin{cases} \text{age}(n) = 10 & (n = 1) \\ \text{age}(n) = \text{age}(n - 1) + 2 & (n > 1) \end{cases}$$

例: age(n)

有 5 个学生坐在一起, 问第 5 个学生多少岁, 他说比第 4 个学生大 2 岁。问第 4 个学生岁数, 他说比第 3 个学生大 2 岁。问第 3 个学生, 又说比第 2 个学生大 2 岁。问第 2 个学生, 说比第 1 个学生大 2 岁。最后问第 1 个学生, 他说是 10 岁。请问第 5 个学生多大。

$$\text{第 } n \text{ 个学生年龄} \begin{cases} \text{age}(n) = 10 & (n = 1) \\ \text{age}(n) = \text{age}(n - 1) + 2 & (n > 1) \end{cases}$$

例: age(n), 递归调用过程分析 (push)

// 递推公式: $\text{age}(1)=10$; $n>1$: $\text{age}(n)=\text{age}(n-1)$

```
int age(int n)
```

```
{
```

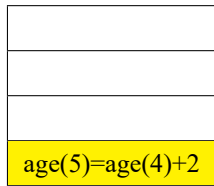
```
    int y;
```

```
    if(n==1) y=10;
```

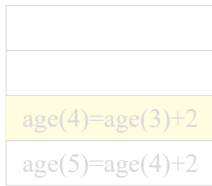
```
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
```

```
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

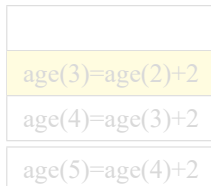
```
}
```



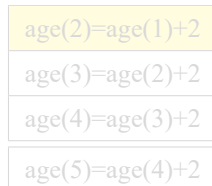
栈 [push(n=5)]



栈 [push(n=4)]



栈 [push(n=3)]



栈 [push(n=2)]

例: age(n), 递归调用过程分析 (push)

// 递推公式: $\text{age}(1)=10$; $n>1$: $\text{age}(n)=\text{age}(n-1)$

```
int age(int n)
```

```
{
```

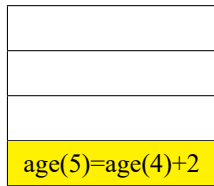
```
    int y;
```

```
    if(n==1) y=10;
```

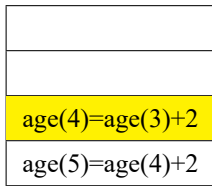
```
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
```

```
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

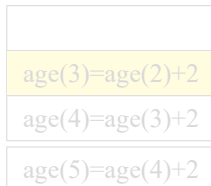
```
}
```



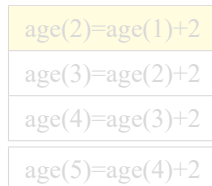
栈 [push(n=5)]



栈 [push(n=4)]



栈 [push(n=3)]



栈 [push(n=2)]

例: age(n), 递归调用过程分析 (push)

// 递推公式: $\text{age}(1)=10$; $n>1$: $\text{age}(n)=\text{age}(n-1)$

```
int age(int n)
```

```
{
```

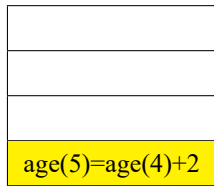
```
    int y;
```

```
    if(n==1) y=10;
```

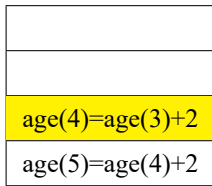
```
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
```

```
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

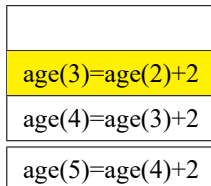
```
}
```



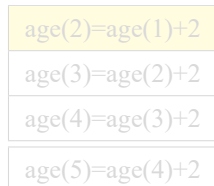
栈 [push(n=5)]



栈 [push(n=4)]



栈 [push(n=3)]



栈 [push(n=2)]

例: age(n), 递归调用过程分析 (push)

// 递推公式: $\text{age}(1)=10$; $n>1$: $\text{age}(n)=\text{age}(n-1)$

```
int age(int n)
```

```
{
```

```
    int y;
```

```
    if(n==1) y=10;
```

```
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
```

```
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

```
}
```

age(5)=age(4)+2

栈 [push(n=5)]

age(4)=age(3)+2
age(5)=age(4)+2

栈 [push(n=4)]

age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

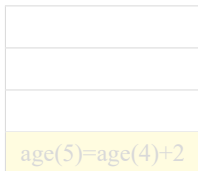
栈 [push(n=3)]

age(2)=age(1)+2
age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [push(n=2)]

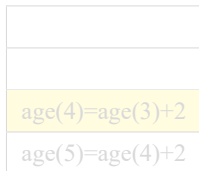
例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
    int y;
    if(n==1) y=10;
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
}
```



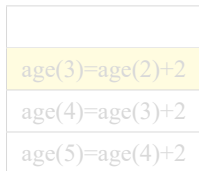
栈 [pop(n=5)]

age(5)=age(4)+2=18



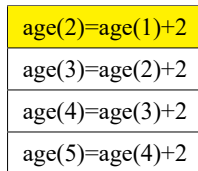
栈 [pop(n=4)]

age(4)=age(3)+2=16



栈 [pop(n=3)]

age(3)=age(2)+2=14

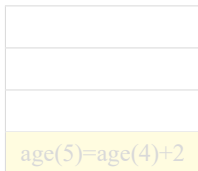


栈 [pop(n=2)]

age(2)=age(1)+2=12

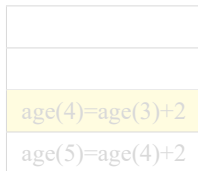
例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
    int y;
    if(n==1) y=10;
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
}
```



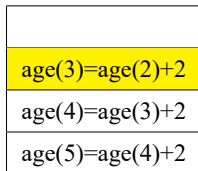
栈 [pop(n=5)]

age(5)=age(4)+2=18



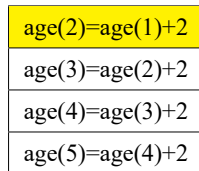
栈 [pop(n=4)]

age(4)=age(3)+2=16



栈 [pop(n=3)]

age(3)=age(2)+2=14



栈 [pop(n=2)]

age(2)=age(1)+2=12

例: age(n), 递归调用过程分析 (pop)

// 递推公式: $\text{age}(1)=10$; $n>1$: $\text{age}(n)=\text{age}(n-1)$

```
int age(int n)
```

```
{
```

```
    int y;
```

```
    if(n==1) y=10;
```

```
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
```

```
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
```

```
}
```

age(5)=age(4)+2

栈 [pop(n=5)]

age(5)=age(4)+2=18

age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=4)]

age(4)=age(3)+2=16

age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=3)]

age(3)=age(2)+2=14

age(2)=age(1)+2
age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=2)]

age(2)=age(1)+2=12

例: age(n), 递归调用过程分析 (pop)

```
// 递推公式: age(1)=10; n>1: age(n)=age(n-1)
int age(int n)
{
    int y;
    if(n==1) y=10;
    else y=age(n-1)+2; // 未完成的计算用“栈”存储起来(push)
    return y; // 函数return前, 从“栈”顶取数据, 计算, 直到“栈”空
}
```

age(5)=age(4)+2

栈 [pop(n=5)]

age(5)=age(4)+2=18

age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=4)]

age(4)=age(3)+2=16

age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=3)]

age(3)=age(2)+2=14

age(2)=age(1)+2
age(3)=age(2)+2
age(4)=age(3)+2
age(5)=age(4)+2

栈 [pop(n=2)]

age(2)=age(1)+2=12

例: $n!$

`double fac(int n) // $n!$ 较大, 因此函数返回类型设置为double`

```
{
    if(n==0 || n==1) return 1;
    else return n*fac(n-1);
}
```

`int main()`

```
{
    printf("%.0lf\n", fac(2)); // 2
    printf("%.0lf\n", fac(3)); // 6
    printf("%.0lf\n", fac(4)); // 24
    printf("%.0lf\n", fac(20)); // 2432902008176640000
    return 0;
}
```

$$n! = \begin{cases} 1 & (n = 0, 1) \\ n(n-1)! & (n > 1) \end{cases}$$

例: 斐波那契数列 (double)

```
double F(int n) // Fn较大, 因此函数返回类型设置为double
```

```
{
    if (n==0 || n==1) return 1;
    else return F(n-1)+F(n-2);
}

int main()
{
    int i;
    for (i=0; i<20; i++)
    {
        printf("%d\t", (int)F(i));
        if ((i+1)%4==0) printf("\n");
    }
    return 0;
}
```

$$\begin{cases} F_n = 1 & (n = 0, 1) \\ F_n = F_{n-1} + F_{n-2} & (n > 1) \end{cases}$$

例: 数字处理

编写一个程序,从键盘输入一个非零整数 n ($0 < n \leq 1000000000$),对整数 n 进行如下处理:

将整数的各位数字取出来相加,如果结果是一位数则输出该数,否则重复上述过程,直到得到的结果为一位数,并输出该结果。

例如: $n=456$,变换过程如下

$$4+5+6=15$$

$$1+5=6$$

输出结果为 6

例: 数字处理—非递归实现

```
#include<stdio.h>

int bitsSum(int a);

int main()
{
    int n,sum;
    scanf("%d",&n);
    while(1)
    {
        sum=bitsSum(n);
        if(sum<=9) break; //1位数字
        else n=sum; // 继续下一轮迭代
    }
    printf("%d\n",sum);
    return 0;
}
```

// 整数a的各位数字之和

```
int bitsSum(int a)
{
    int sum=0;
    while(a)
    {
        sum += a%10;
        a /= 10;
    }
    return sum;
}
```

例: 数字处理—递归实现

```
#include<stdio.h>

int bitsSum(int a);
int bits1(int n);

int main()
{
    int n,sum=0;
    scanf("%d",&n);
    printf("%d\n",bits1(n));
    return 0;
}
```

```
// 整数a的各位数字之和
int bitsSum(int a)
{
    int sum;
    if(a==0) sum=0;
    else sum=bitsSum(a/10)+a%10;
    return sum;
}

// 确保最后是1位数字
int bits1(int n)
{
    int result;
    result=bitsSum(n);
    if(result>9) result=bits1(result); // 递归
    return result;
}
```

例：二进制输出 (正序)

```
void to_binary(unsigned long n)
{
    int r;
    r=n%2; //01001011
    // push(先计算的二进制位r)
    if(n>=2) to_binary(n/2);
    // pop(后进先出FILO=先进后出FILO)
    putchar('0'+r); //11010010
}

int main()
{
    // 无符号长整型, 可赋值10进制数
    unsigned long x=0XD2; //11010010
    scanf("%x",&x); //十六进制输入
    //等效10进制无符号长整型
    //scanf("%uld",&x);
    to_binary(x);
    putchar('\n');
    return 0;
}
```

以 $x=0XD2$ 为例, 分析递归过程。

$r=n\%2$;

计算顺序 $2^0, 2^1, \dots, 2^7$

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
0	1	0	0	1	0	1	1

$to_binary(n/2)$;

push(先计算的 r), 栈顶 (top, 最左端), 栈底

(bottom, 最右端)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

$putchar('0'+r)$;

pop(后进先出), 栈顶元素先出, 栈底元素最后出

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

例：二进制输出 (倒序)

```
void to_binary(unsigned long n)
{
    int r;
    r=n%2; //01001011
    putchar('0'+r); //01001011
    // push(先计算的二进制位r)
    if(n>=2) to_binary(n/2);
}

int main() // 与正序一致
{
    // 无符号长整型
    unsigned long x=0XD2; //11010010
    scanf("%x",&x); //十六进制输入
    //等效10进制无符号长整型
    //scanf("%uld",&x);
    to_binary(x);
    putchar('\n');
    return 0;
}
```

以x=XD2为例,分析递归过程。

r=n%2;

计算顺序 $2^0, 2^1, \dots, 2^7$

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
0	1	0	0	1	0	1	1

to_binary(n/2);

push(先计算的r), 栈顶 (top, 最左端), 栈底

(bottom, 最右端)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

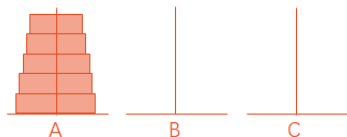
pop(后进先出), 栈顶元素先出, 栈底元素最后出

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	0	1	0

例: Hanoi(汉诺) 塔问题

古代有一个梵塔, 塔内有 3 个座 A,B,C。开始时 A 座上有 64 个盘子, 盘子大小不等, 大的在下, 小的在上。有一个老和尚想把这 64 个盘子从 A 座移到 C 座, 但规定每次只允许移动一个盘, 且在移动过程中在 3 个座上都始终保持大盘在下, 小盘在上。在移动过程中可以利用 B 座。

要求编程输出移动盘子的步骤。



例: Hanoi(汉诺) 塔问题—解题思路

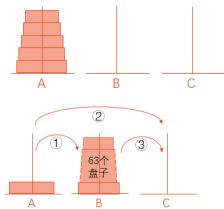
- 老和尚想: 假如有第 2 个和尚将上面 63 个盘子从 A 座移到 B 座。我就能将 64 个盘子从 A 座移到 C 座。

- 1 命令第 2 个和尚将 63 个盘子从 A 座移到 B 座;
- 2 自己将 1 个盘子 (最底下的、最大的盘子) 从 A 座移到 C 座;
- 3 再命令第 2 个和尚将 63 个盘子从 B 座移到 C 座。

- 第 2 个和尚想: 假如有第 3 个和尚能将上面 62 个盘子从 A 座移到 C 座。我就能将 63 个盘子从 A 座移到 B 座。

- 1 命令第 3 个和尚将 62 个盘子从 A 座移到 C 座;
- 2 自己将 1 个盘子从 A 座移到 B 座;
- 3 再命令第 3 个和尚将 62 个盘子从 C 座移到 B 座。

■ ...



例: Hanoi(汉诺) 塔问题—解题思路 (3 层)

■ 将 A 座的 3 个盘子移到 C 座, 分 3 步:

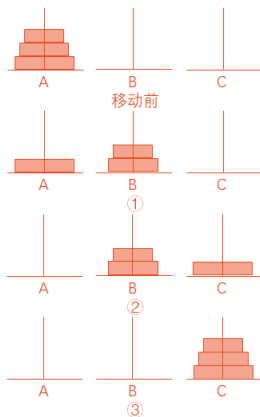
- 1 将 A 上 2 个盘子移到 B(借助 C 座)。
- 2 将 A 上 1 个盘子移到 C(直接实现)。
- 3 将 B 上 2 个盘子移到 C(借助 A 座)。

■ 分解第 1 步: 将 A 上的 2 个盘子移到 B:

- 1 将 A 上 1 个盘子移到 C(借助 C 座)。
- 2 将 A 上 1 个盘子移到 B。
- 3 将 C 上 1 个盘子移到 B。

■ 分解第 3 步: 将 B 座的 2 个盘子移到 C 座:

- 1 将 B 上 1 个盘子移到 A(借助 A 座)。
- 2 将 B 上 1 个盘子移到 C。
- 3 将 A 上 1 个盘子移到 C。



例: Hanoi(汉诺) 塔问题—解

```

int main()
{
    int n=3;
    scanf("%d",&n);
    //n个盘由'A'移到'C',借助'B'
    hanoi(n,'A','B','C');
    return 0;
}

void move(char one,char two)
{
    printf("%c->%c\n",one,two);
}

//将n个盘从one移到three,借助two
void hanoi(int n,char one,char two,char three)
{
    if(n==1) move(one,three);
    else
    {
        //n-1盘从one移到two,借three
        hanoi(n-1,one,three,two);
        //将1个盘从one移到three
        move(one,three);
        //n-1盘从two移到three,借one
        hanoi(n-1,two,one,three);
    }
}

```

■ 将 A 座的 3 个盘子移到 C 座,

分 3 步:

- 1 将 A 上 2 个盘子移到 B(借 C)。
- 2 将 A 上 1 个盘子移到 C(直接)。
- 3 将 B 上 2 个盘子移到 C(借 A)。

■ 分解第 1 步: 将 A 上的 2 个盘子移到 B:

- 1 将 A 上 1 个盘子移到 C(借 C)。
- 2 将 A 上 1 个盘子移到 B。
- 3 将 C 上 1 个盘子移到 B。

■ 分解第 3 步: 将 B 座的 2 个盘子移到 C 座:

- 1 将 B 上 1 个盘子移到 A(借 A)。
- 2 将 B 上 1 个盘子移到 C。
- 3 将 A 上 1 个盘子移到 C。

//将n个盘从one移到three,借助two

```
void hanoi(int n,char one,char two,char three)
{
    // 递归终止条件, 出栈pop(2,...,n-1,n), 回归, 回溯
    if(n==1) move(one,three);
    else
    { //压栈push(n,n-1,n-2,...2)
        //n-1盘从one移到two,借three
        hanoi(n-1,one,three,two);
        //将1个盘从one移到three
        move(one,three);
        //n-1盘从two移到three,借one
        hanoi(n-1,two,one,three);
    }
}
```

$A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, A \rightarrow C$

递归小结

■ 递归函数必须满足两个条件:

- 1 在每一次调用自己时,必须是 (在某种意义上) 更接近于解;
- 2 必须有一个终止处理的语句。

■ 递归能够解决的问题:

- 1 数据的定义是按递归定义的。如 Fibonacci 函数。
- 2 问题解法按递归算法实现。如 Hanoi 问题。
- 3 数据的结构形式是按递归定义的。如二叉树、广义表等。

- 使用递归的关键在于将问题分解为小部分,递归不能永远进行下去,因为它总是以最小可能性问题结束,
- 递归函数的优点是定义简单,逻辑清晰,但是代码难于理解。
- 理论上,所有的递归函数都可以写成循环的方式,但循环逻辑不如递归清晰。
- 层次越深,调用栈 (内存) 越大,效率越低,并且可能会溢出。因此,能用循环解决的问题,尽量不要用递归。

定义在函数内部或复合语句中的变量: 局部变量

局部变量的作用域是从定义语句开始至函数或复合语句结束, 不会影响作用域

以外的同类型的同名变量值。

// 各函数中的局部变量, 不会相互影响。

```
float f1(float a)
{
    float c; // 本函数局部变量
    ....;
}

float f2(float a)
{
    float c; // 本函数局部变量
    ....;
}

int main()
{
    float c=10, d; // 本函数局部变量
    d=f1(c); d=f2(d);
}
```

// 各复合语句中的局部变量, 不会相互影响。

```
float f1(float a)
{
    float c; // 本函数局部变量
    for(int i=0; i<10; i++) // i: for复合语句中的局部变量
    {
        int t; // 复合语句中的局部变量
        ....
    }
    if(c>10)
    {
        int t; // 复合语句中的局部变量
        ....
    }
    ...
}
```

形式参数被当作本函数的局部变量

形式参数被当作本函数的局部变量,因此它不会影响实际参数的值。

```
float f1(float a)
{
    float c; //函数内部的局部变量不得与形式参数同名
    a=30; // 改变形式参数的值不会影响实际参数的值
}

int main()
{
    float c=10,d; //主函数局部变量
    d=f1(c); //f1函数对形参的改变不会影响实参c的值
    float x[2]={0.1,0.2};
    f2(x); // 函数内部可以改变数组元素的值
    printf("%f,%f,%f\n",c,x[0],x[1]); //
        10,10.1,10.2
}
```

```
//函数内部可以改变数组元素的值,但是对地址的改
    变不会影响实参的地址
//形式参数的数组名是地址变量, 不是常量.
//回忆: int m[5]; // 数组名是地址常量.

float f2(float a[])
{
    //对数组元素的改变,就是对实参数组元素的改变
    a[0]=10.1;
    a[1]=10.2;
    //对数组名(地址)的改变, 不会影响实参的地址.
    a=a+1;
}
```

定义在函数外部的变量: 全局变量

与模块化函数封装思想冲突, 不推荐使用。全局变量的作用域是从定义语句开始至该文件结束, 会影响作用域内的同类型的同名变量值。

```
int g=0; // 全局变量
float f1(float a)
{
    g=10; // 改变全局变量的值
}
float f2(float a)
{
    g=20; // 改变全局变量的值
}
int main()
{
    g=30; // 改变全局变量的值
}
```