

标准C编程

C/C++教学体系

“

个人简介
闵卫

minwei@tarena.com.cn”

“

编程基础

”

全程目标

- C语言的历史背景
- C程序的开发步骤
- gcc的常用选项
- #include指令
- C语言的注释风格
- C语言的编程规范
- 常量与变量
- 基本数据类型
- 三个常用函数



C语言的历史背景

- 1960年，**Algol 60**，最早的块结构语言。远离硬件，不适合编写系统程序
- 1963年，剑桥大学，将Algol 60发展成**CPL**语言。过于庞大，难学难用
- 1967年，Martin Richards，将CPL语言简化为**BCPL**语言。速度太慢，缺乏运行时支持
- 1970年，贝尔实验室，Ken Thompson，在BCPL语言的基础上发明了**B**语言，专门用于系统编程。字符处理、浮点运算和指针开销不够理想

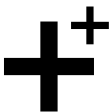


C语言的历史背景

- 1972年，贝尔实验室，**Dennis Ritchie**，为B语言增加了数据类型，并做了大量修改，使之在功能、性能和易用性方面，取得了长足进步。这就是**C语言**！

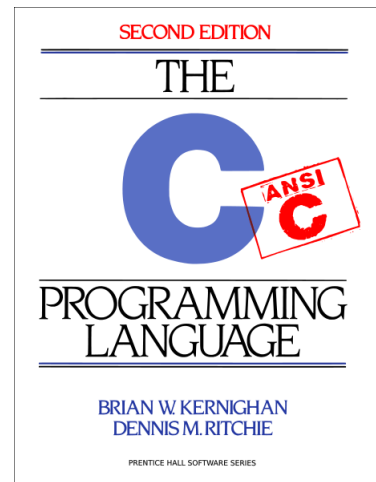


Dennis M. Ritchie，丹尼斯·里奇
1941-2011
C语言之父、Unix之父、黑客之父



C语言的历史背景

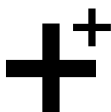
- 1973年，Dennis Ritchie用C重写了Unix内核
- 1978年，不朽名著《The C Programming Language》正式出版。C从贝尔实验室走向世界
- 1989年，美国国家标准化协会，**ANSI C89**
- 1999年，国际标准化组织，**ISO C99**
- 2011年，国际标准化组织，**ISO C11**



练习时间

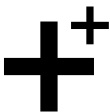
第一个C程序：hello.c

体验C语言程序的开发步骤



C程序的开发步骤

- 编辑源代码
- 编译预处理
- 将源预处理的结果编译为汇编代码
- 将汇编代码汇编为目标模块
- 将目标模块和库链接为可执行程序
- 将可执行程序加载到内存形成进程映像
- 处理器执行进程映像中指令



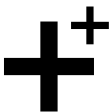
gcc的常用选项

- -o : 指定**输出**文件路径
- -E : 编译**预处理**
 - gcc -E hello.c -o hello.i
- -S : **编译**产生汇编文件
 - gcc -S hello.i
- -c : **汇编**产生目标模块
 - gcc -c hello.s
- -l : 指定**链接库**
 - gcc hello.o -lc



gcc的常用选项

- -x：指定源代码的**语言**
- -std=C89/C99：指定语言**标准**
- -Wall：产生尽可能多的**警告**
- -g：生成**调试**信息
- -I：指定**头文件**搜索目录
- -L：指定**库文件**搜索目录
- -O1/O2/O3：指定**优化**等级



#include指令

- 头文件扩展
- #include <xxx.h>
 - 先找-I目录，再找系统目录
- #include "xxx.h"
 - 先找-I目录，再找当前目录，最后找系统目录
- 头文件的系统目录
 - /usr/include
 - /usr/local/include
 - /usr/lib/gcc/i686-linux-gnu/4.6.3/include



C语言的注释风格

- 从/*开始，到*/结束，中间是注释，**无行数限制**
 - a /* b c
d */ e
f
- 从//开始，到本行结束，**单行注释**
 - a // b c
d // e
f



C语言的编程规范

- 单条语句可写在任意**多行**内
- 必要的**空格**令代码更加清晰
- 严格的**缩进**令代码层次分明
- 适度的**空行**划分出逻辑单元
- 统一用**驼峰**或者**下划线**命名

```
int compare( const void *arg1, const void *arg2 );

int main( int argc, char **argv )
{
    int i;
    /* Eliminate argv[0] from sort: */
    argv++;
    argc--;

    /* Sort remaining args using Quicksort algorithm: */
    qsort( (void *)argv, (size_t)argc, sizeof( char * ), compare );

    /* Output sorted list: */
    for( i = 0; i < argc; ++i )
        printf( " %s", argv[i] );
    printf( "\n" );
}

int compare( const void *arg1, const void *arg2 )
{
    /* Compare all of both strings: */
    return _stricmp( * ( char** ) arg1, * ( char** ) arg2 );
}
```



常量与变量

- 在程序执行过程中，其值**不发生改变**的量称为常量
 - 直接常量(字面值)
 - ✓ 整型量：10，10U，10L，10LL，012，0xA
 - ✓ 实型量：0.12，0.12F，1.2E-1
 - ✓ 字符量：'A'，'\101'，'\x41'，'\n'
 - ✓ 字符串："Hello, World !\n"
 - 标识符
标识变量名、数组名、函数名、类型名的有效字符序列
 - 符号常量
符号化的常量，如宏定义、枚举元素等



常量与变量

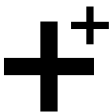
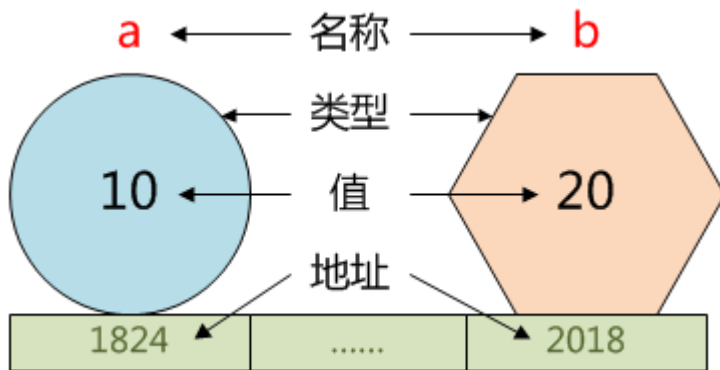
- 在程序执行过程中，其值可以改变的量称为变量
 - 变量在内存中占据一定的存储单元，其内容可变
 - 变量在使用前，必先定义其类型，且只能定义一次
 - 欲使变量拥有确定的值，必为其赋值
 - 定义变量的同时赋予初值叫做初始化
 - 变量名必须是合法的标识符
 - ✓ 字母或下划线开头
 - ✓ 包含字母、下划线和数字
 - ✓ 不能与关键字冲突
 - ✓ 大小写敏感



常量与变量

- 变量的四要素：

- 名称
叫什么？
- 类型
是什么？
- 值
存什么？
- 地址
在哪里？



基本数据类型

- 字符型([signed/unsigned] char)
 - 字符常量需要用一对单引号"括起来
 - 每个字符占8位，即1个字节
 - 取值范围
 - ✓ char : 有符号, -128 ~ 127
 - ✓ unsigned char : 无符号, 0 ~ 255
 - 底层存储的是整数，即字符的ASCII编码
 - printf/scanf格式化标记%c



ASCII编码

- American Standard Code for Information Interchange , 美国信息交换标准代码
- 1967年首次发表，1986年最后一次修订
- 包括128个字符，其中：
 - 33个控制字符多数已废弃不用
 - 95个可显示字符包括26个基本拉丁字母大小写、10个阿拉伯数字、32个标点及数学符号、1个空格
- 扩展版本EASCII增加了部分欧州语言字符
- 无法涵盖所有语言字符，逐渐被Unicode编码取代



<div> <div> <div>b_7</div> <div>b_6</div> <div>b_5</div> </div> <div> <div></div> <div></div> <div></div> </div> </div>						0	0	0	0	1	1	1	1
						0	0	1	0	1	0	1	1
<div> <div> <div>Column →</div> </div> <div> <div>Row ↓</div> </div> </div>						0	1	2	3	4	5	6	7
Bits	b_4	b_3	b_2	b_1									
	0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
	0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
	0	0	1	0	2	STX	DC2	"	2	B	R	b	r
	0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
	0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
	0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
	0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
	0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
	1	0	0	0	8	BS	CAN	(8	H	X	h	x
	1	0	0	1	9	HT	EM)	9	I	Y	i	y
	1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
	1	0	1	1	11	VT	ESC	+	;	K	[k	{
	1	1	0	0	12	FF	FC	,	<	L	\	l	
	1	1	0	1	13	CR	GS	-	=	M]	m	}
	1	1	1	0	14	SO	RS	.	>	N	^	n	~
	1	1	1	1	15	SI	US	/	?	O	_	o	DEL

常用控制字符

ASCII码	缩写	含义
0	NUL	空字符
7	BEL	响铃
8	BS	退格
9	HT	水平制表
10	LF	换行
11	VT	垂直制表
12	FF	换页
13	CR	回车



常用可显示字符

ASCII码	字符
32	空格
33-47	!"#\$%&'()*+,-./
48-57	0123456789
58-64	;<=>?@
65-90	ABCDEFGHIJKLMNOPQRSTUVWXYZ
91-96	[\ ^ _ `
97-122	abcdefghijklmnopqrstuvwxyz
123-126	{ } ~



转义字符

字面值	含义	字面值	含义
\a	响铃	\"	"
\b	退格	\\	\
\t	水平制表	\ooo	1-3位8进制编码
\n	换行	\xhh	1-2位16进制编码
\v	垂直制表		
\f	换页		
\r	回车		
\'	'		



练习时间

把数字字符转换为整数

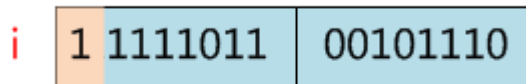
把小写字母转换为大写字母



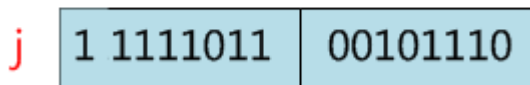
基本数据类型

- 整型([signed/unsigned] short/int/long)
 - 有符号数(signed, 缺省)最高位为符号位, 0正1负
 - 无符号数(unsigned)最高位为数字位, 大于等于0

short i = -1234



unsigned short j = 64302



- printf/scanf格式化标记%[#][h/l/lld/u/o/x/X



基本数据类型

- 整型([signed/unsigned] short/int/long)
 - 字面值前缀
 - ✓ 无：十进制
 - ✓ 0：八进制
 - ✓ 0x：十六进制
 - 字面值后缀
 - ✓ 无：int
 - ✓ U/u：unsigned
 - ✓ L/l：long
 - ✓ LL/ll：long long



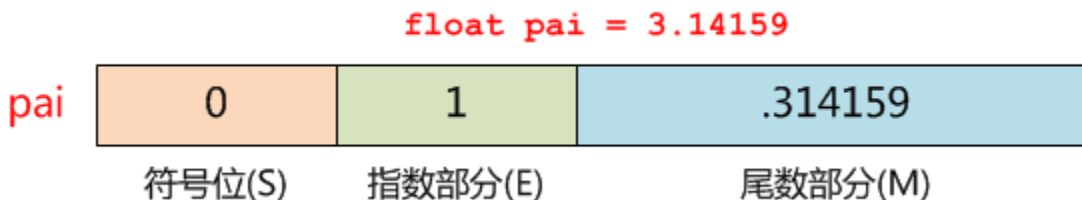
整型量的字长与值域

数据类型	字长	值域
short	2字节	$[-32768, 32767]$
unsigned short	16位	$[0, 65535]$
int	4字节	$[-2147483648, 2147483647]$
unsigned int	32位	$[0, 4294967295]$
long	32位系统同int, 64位系统同long long	
unsigned long	32位系统同unsigned int, 64位系统同unsigned long long	
long long	8字节	$[-2^{63}, 2^{63}-1]$
unsigned long long	64位	$[0, 2^{64}-1]$

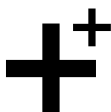


基本数据类型

- 实型(float/double/long double)
 - 实型数据在内存中以幂的形式存储，谓之阶码



- 指数部分位数越多，所能表示的数值范围越大
- 尾数部分位数越多，有效数字越多，精度越高
- 受尾数部分字长的限制，实型数据总是近似值



基本数据类型

- 实型(float/double/long double)
 - printf/scanf格式化标记%[l/L]f/e/g
 - 字面值后缀
 - ✓ 无 : double
 - ✓ F/f : float
 - ✓ L/l : long double



实型量的精度与值域

数据类型	字长	有效数字	值域
float	4字节/32位 1S+8E+23M	6/7	$\pm[1.2 \times 10^{-38}, 3.4 \times 10^{38}]$
double	8字节/64位 1S+11E+52M	15/16	$\pm[2.2 \times 10^{-308}, 1.8 \times 10^{308}]$
long double	10字节/80位 1S+15E+64M	18/19	$\pm[3.4 \times 10^{-4932}, 1.2 \times 10^{4932}]$

- 32位GCC将long double对齐为12字节
- 64位GCC将long double对齐为16字节
- 微软编译器long double与double相同



三个常用函数

- `printf ("格式字符串", ...);`
 - 格式化输出
- `scanf ("格式字符串", 地址表);`
 - 格式化输入
- `sizeof (参数);`
 - 计算内存的大小，以**字节**为单位
 - 参数可以是**类型**、**变量**或**表达式**
 - **不计算参数的值**，只关注其**类型**



练习时间

读取用户输入的圆半径，计算并输出圆的面积和周长



“

进制转换 与运算符

”

全程目标

- 整数的二、八、十六进制与十进制相互转换
- 赋值运算与算术运算
- 关系运算与逻辑运算
- 条件运算与逗号运算
- 取地址与解引用运算
- 位运算
- 类型转换运算
- 运算符的优先级与结合序



二进制与十进制的转换

- 位与权

$$93 = 9 \times 10 + 3 \times 1$$

/ \

10 1

整数93在一个字节中的二进制存储形式：

+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+
	0		1		0		1		1		1		0		1					- 位
+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+	-	-	-	+
	128		64		32		16		8		4		2		1					- 权



二进制与十进制的转换

- 零和正数，符号位(最高位)为0
 - 二转十：一位加权，零位不见

$$01011101 = 64 + 16 + 8 + 4 + 1 = 93$$

$$01101101 = 64 + 32 + 8 + 4 + 1 = 109$$

- 十转二：有权添一，无权补零

$$93 - 64 = 29 - 16 = 13 - 8 = 5 - 4 = 1 = 01011101$$

$$109 - 64 = 45 - 32 = 13 - 8 = 5 - 4 = 1 = 01101101$$



二进制与十进制的转换

- 负数，符号位(最高位)为1
 - 二转十：取反加一，转十添负

10100011取反01011100加一01011101转十 93添负 -93
10010011取反01101100加一01101101转十109添负-109

- 十转二：去负转二，取反加一

-93去负 93转二01011101取反10100010加一10100011
-109去负109转二01101101取反10010010加一10010011



二进制与十六进制的转换

十	二	十六	十	二	十六
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



二进制与十六进制的转换

- 四位二进制对应一位十六进制

十进制	二进制	十六进制
93	0101 1101	0x5D
109	0110 1101	0x6D
-93	1010 0011	0xA3
-109	1001 0011	0x93
-191	1111 1111 0100 0001	0xFF41



二进制与八进制的转换

十进制	二进制	八进制
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7



二进制与八进制的转换

- 三位二进制对应一位八进制

十进制	二进制	八进制
93	01 011 101	0135
109	01 101 101	0155
-93	10 100 011	0243
-109	10 010 011	0223
-191	1 111 111 101 000 001	0177501



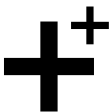
赋值运算

- 把右边的值赋给左边的变量，赋值从右向左算
`int i, j, k;`
`i = j = k = 0;` // `k=0` , `j=k` , `i=j` , 可以但不推荐
- 左边必须是一个左值(变量)，不能是常数或表达式
`0 = i;` // 错误
`i + j = 0;` // 错误
- 赋值表达式的左值即是赋值表达式的值
- 赋值可以和其它运算符结合使用，谓之复合赋值
`i += 3;` // `i = i + 3`
- `=`代表赋值，`==`代表相等



算术运算

- 加(+)、减(-)和乘(*)
- 除(/)和模(%)
 - 整数相除，**向0取整**(取更靠近0的数字)
 - 对0做/和%会**中断程序**，/0.0得到**无穷大**(inf)
 - %**只能用于整数**，实数不支持
 - %的正负号与%**前面数字**的正负号相同



练习时间

输入一个秒数，输出hh小时mm分ss秒



算术运算

- 自增减运算
 - ++/--, **变量**自增/自减1, 不能用于常数
5++; // 错误
 - 整型实型均可自增减, 但主要用于**整型**
 - 前缀表达式, **先自增减, 后运算**
 - 后缀表达式, **先运算, 后自增减**
 - i、i++和++i, 从内存上说是**同一块**内存区域
 - i++/++i最好**单独**作为一个语句出现



关系运算

- 大于($>$)和大于等于($>=$)
- 小于($<$)和小于等于($<=$)
- 等于($==$)和不同于($!=$)
- 关系表达式的值是**真和假**，分别用**1和0**表示



逻辑运算

- 与(&&)、或(||)和非(!)
 - 与：并且，全真则真，否则为假
 - 或：或者，全假则假，否则为真
 - 非：反之，非真即假，非假即真
- 逻辑表达式的值是**真和假**，分别用**1和0**表示
- **短路与**：若第一个表达式为假，则结果为假，后面的表达式不再计算
- **短路或**：若第一个表达式为真，则结果为真，后面的表达式不再计算



条件运算和逗号运算

- 条件表达式 ? 表达式1 : 表达式2;
 - 若条件表达式为真，则整个表达式的值取表达式1的值，否则取表达式2的值
 - 若表达式1和表达式2的类型不同，则自动转换为二者中较高等级的类型
- 表达式1, 表达式2, ..., 表达式n
 - 依次计算各表达式的值，以表达式n的值作为整个表达式的值
 - 并不是在所有出现逗号的地方都是逗号表达式
`int a = 2, b = 4, c = 6, x, y; // 不是逗号表达式`



取地址和解引用运算

- 内存中的地址是按字节编号的，多字节变量的地址是首(地址最低)字节的地址
- 取地址运算(&)，根据变量得到地址
- 解引用运算(*)，根据地址取得变量



位运算

- 位与(&)
 - 参与运算的两位都是1结果为1，否则为0
 - 位与可以置某一位为0，也可以得到某一位的值

```

    10110001  A
&) 11101111  B
-----  将A第四位置0
    10100001
    
```

```

    10110001  A
&) 00010000  B
-----  若A&B==B，则A第四位为1，否则为0
    00010000
    
```



位运算

- 位或(|)
 - 参与运算的两位都是0结果为0，否则为1
 - 位或可以置某一位为1，也可以得到某一位的值

```

    10100001  A
|)  00010000  B
-----  将A第四位置1
    10110001
    
```

```

    10100001  A
|)  11101111  B
-----  若A|B==B，则A第四位为0，否则为1
    11101111
    
```



位运算

- 异或(^)
 - 参与运算的两位相异为1，否则为0
 - 异或可以翻转某一位，想翻哪位哪位为1，其余为0

```

    10101010 A
^) 00100100 B
----- 将A第三位和第六位翻转
    10001110
    
```

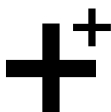


位运算

- 位反(~)
 - 1变0 , 0变1

```

~) 01100001
-----
    10011110
    
```



位运算

- 左移位(<<)
– 有符号数，右补0

10101010 << 1 = 01010100
-86 84

- 无符号数，右补0

10101010 << 1 = 01010100
170 84



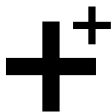
位运算

- 右移位(>>)
 - 有符号数，左补符号位

10101010 >> 1 = 11010101
-86 -43

- 无符号数，左补0

10101010 >> 1 = 01010101
170 85



位运算

- 在不发生高位溢出的前提下
 - 左移1位相当于乘以2

00001111 << 1 = 00011110
15 30

- 右移1位相当于除以2

00001111 >> 1 = 00000111
15 7



练习时间

输入一个整数，以二进制形式打印输出



类型转换运算

- 升级转换
 - 没有任何信息损失，包括总值和精度
 - 整型到整型，实型到实型
 - 少字节类型向多字节类型转换
 - 字长相等的有符号和无符号类型，有符号能表示无符号者向有符号类型转换，否则向无符号类型转换
- 标准转换
 - 有信息损失，总值或者精度



类型转换运算

- 隐式转换
 - 所有的升级转换
 - 整型到实型的标准转换
- 显示转换
 - (目标类型)源类型变量
- 类型转换实际是建立了一个新的目标类型的匿名变量(临时变量)，源类型变量在转换前后保持不变



运算符的优先级

- 单目高于双目
- 乘除高于加减
- 算术高于关系高于逻辑
- 条件高于赋值高于逗号



运算符的优先级

优先级	类别	运算符
12	初等运算	()、[]、..、->
11	单目运算	!、~、++、--、-、类型转换、&、*、sizeof
10	算数运算	*、/、%高于+、-
9	移位运算	>>、<<
8	关系运算	>、>=、<、<=高于==、!=
7	位与运算	&
6	异或运算	^
5	位或运算	



运算符的优先级

优先级	类别	运算符
4	逻辑运算	!高于&&高于
3	条件运算	?:
2	赋值运算	=、+=、-=、*=、/=、%=、&=、 =、^=、>>=、<<=
1	逗号运算	,



运算符的结合序

- 多数运算符具有左结合序

$a-b+c$ 等价于 $(a-b)+c$

- 单目、三目和赋值运算符具有右结合序

$-----a$ 等价于 $-(--(--a))$

$a>b?a:c>d?c:d$ 等价于 $a>b?a:(c>d?c:d)$

$a=b+=c$ 等价于 $a=(b+=c)$



“

流程控制

”

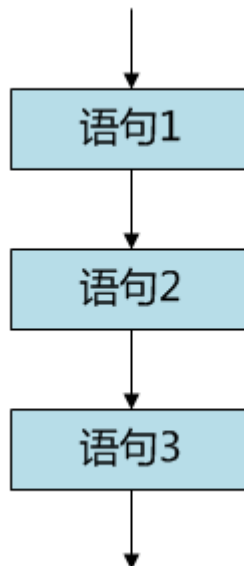
全程目标

- 顺序结构
- 条件分支结构
- 开关分支结构
- 循环结构
- 结构化程序设计



顺序结构

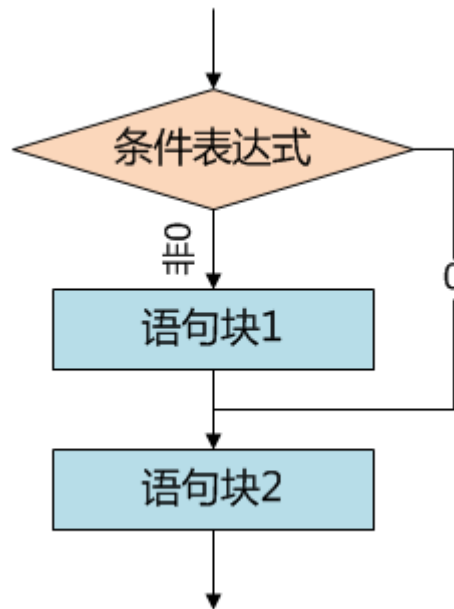
- 从上到下，顺序执行各条语句



条件分支结构

- if结构

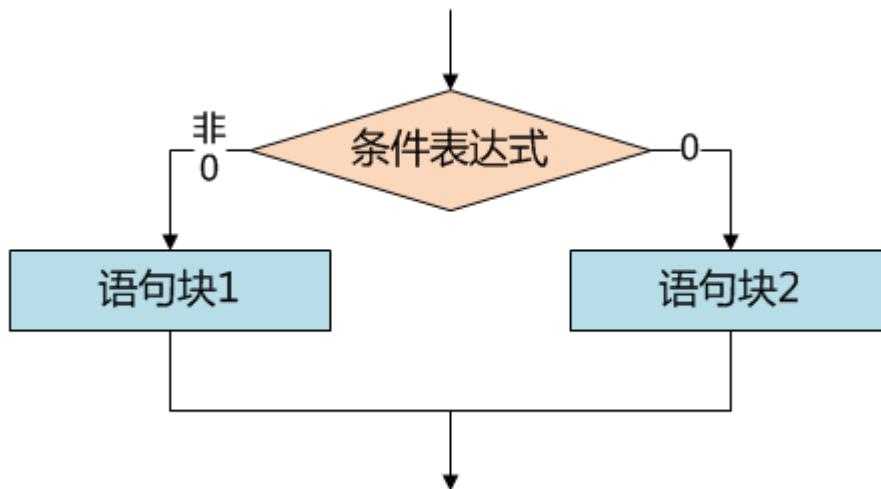
```
if (条件表达式) {  
    语句块1;  
}  
语句块2;
```



条件分支结构

- if-else结构

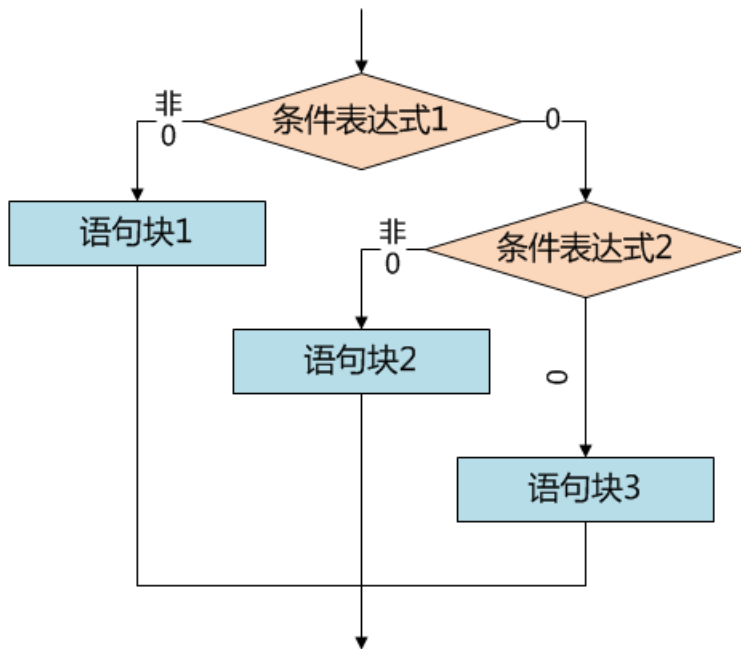
```
if (条件表达式) {
    语句块1;
}
else {
    语句块2;
}
```



条件分支结构

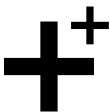
- if-else if-else结构

```
if (条件表达式1) {  
    语句块1;  
}  
else if (条件表达式2) {  
    语句块2;  
}  
else {  
    语句块3;  
}
```



条件分支结构

- if只能出现1次，else if可出现0-N次，else可出现0-1次
- if-else语句应用于需要根据不同条件执行不同代码的场合
- if-else语句最多只能执行1个分支，有else分支必选其一执行，无else分支可选其一执行
- 如果{}中仅一条语句，可以省略{}
- 建议仅对单条break/continue/return省略{}



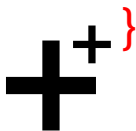
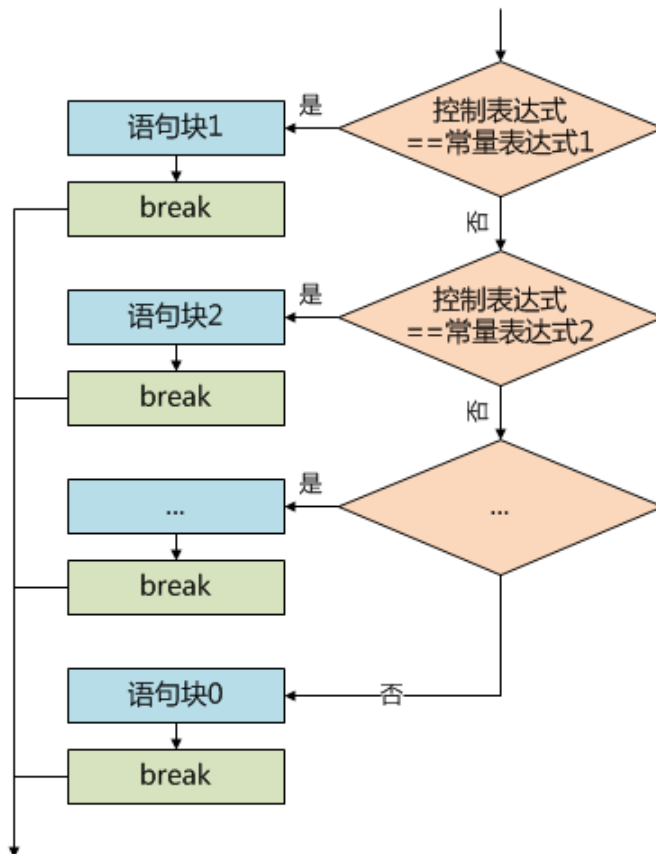
练习时间

1. 工资出行
2. 每月天数
3. 四数最大
4. 判断闰年



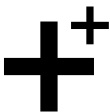
开关分支结构

```
switch (控制表达式) {
    case 常量表达式1:
        语句块1;
        break;
    case 常量表达式2:
        语句块2;
        break;
    ...
    default:
        语句块0;
        break;
}
```



开关分支结构

- 控制表达式被当做**整数**处理，可以是字符，但不能是浮点数和字符串
- 常量表达式必须是**常量**
如：3、'A'、2+5
- **不允许**出现值**相等**的常量表达式
- default分支放在最后，其中的break**可以**省略
- default分支不在最后，其中的break**不可**省略
- 开关分支仅用于根据**有限**数量的**整数**条件分支处理
- 条件分支**完全**可以取代开关分支，反之**不行**



练习时间

输入一个1到100的成绩，输出级别

90 - 100 : 优

80 - 89 : 良

70 - 79 : 中

60 - 69 : 差

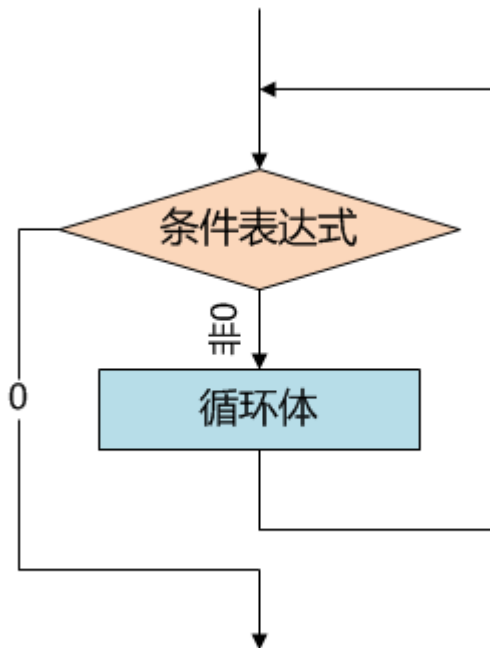
60分以下 : 劣



循环结构

- while循环

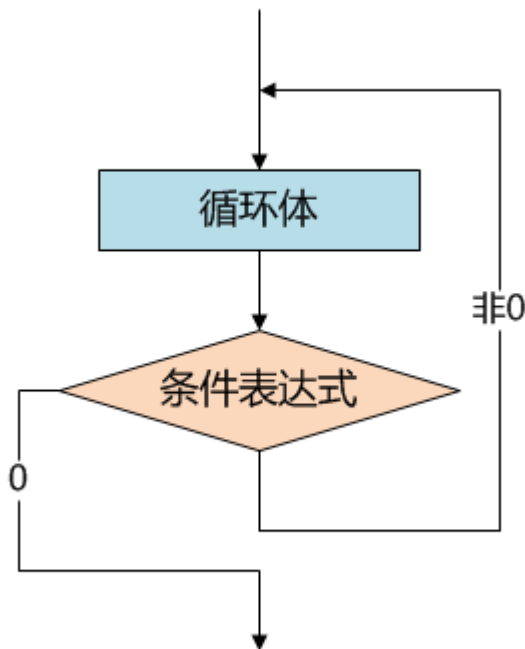
```
while (条件表达式) {  
    循环体;  
}
```



循环结构

- do-while循环

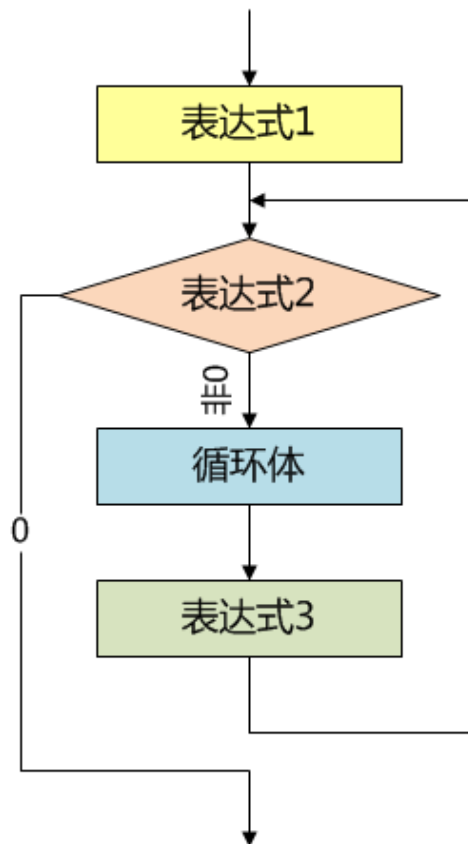
```
do {  
    循环体;  
} while (条件表达式);
```



循环结构

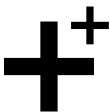
- for循环

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



循环结构

- 循环结构用于重复执行特定的代码段
- while循环与for循环先计算条件表达式，后执行循环体，因此循环体有可能一次也不执行
- do-while循环先执行循环体，后计算条件表达式，因此循环体至少执行一次
- 多重循环，多层嵌套的循环
- 不定循环，循环次数不确定
- 空循环，循环体为空语句
- 死循环，永远不结束的循环



循环结构

- 循环体内部三个常用关键字
 - continue
中断本次循环，继续下次循环
while/do-while循环，跳转到计算条件表达式
for循环，跳转到计算表达式3
 - break
结束循环，跳转到循环体之外
 - goto
跳转到标号处执行



练习时间

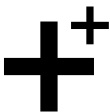
1. 打印1到100之间的全部奇数
2. 输入任意一个整数，打印其位数
3. 打印如下图形

```
      *
    * * * * *
  * * * * * * *
* * * * * * * * *
  * * * * * * *
    * * * * *
      *
```



结构化程序设计

- 顺序结构、条件分支结构、开关分支结构以及三种循环结构都是**单入口单出口**的流程控制结构
- 任意复杂的程序，都可以看做是由这些**基本**流程控制结构组合而成的，单入口单出口的系统
- 完全以这些基本流程控制结构为基础，构建复杂应用的程序设计方法，就叫做**结构化程序设计**



练习时间

利用goto语句实现三种循环结构
追忆非结构化程序设计的岁月



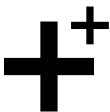
“

数组

”

全程目标

- 基本概念
- 数组的定义及初始化
- 数组的使用
- 动态数组
- 多维数组



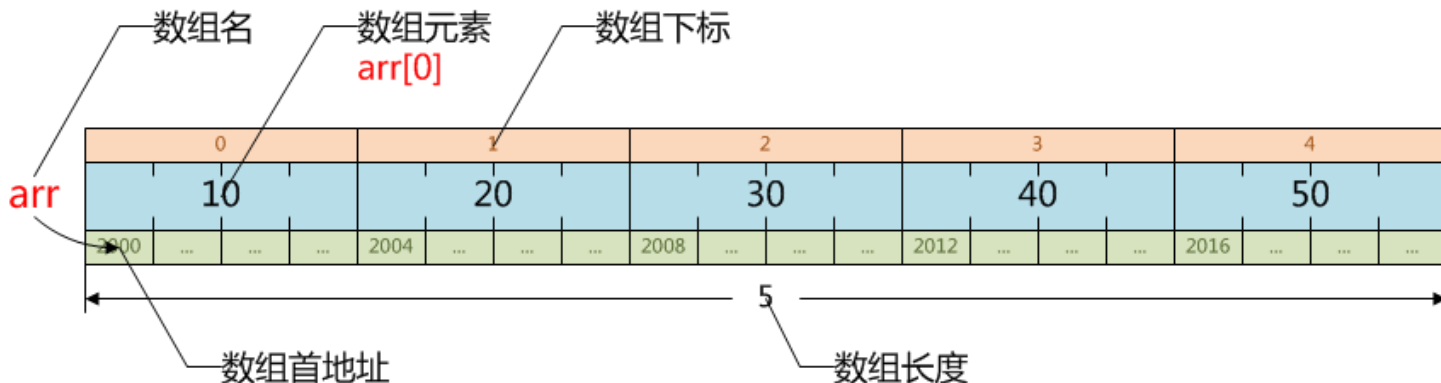
基本概念

- 数组是用来存储多个相同类型数据的数据结构
 - 多个、相同类型
 - 连续的内存区域
 - 数组名是数组首元素的符号地址，即数组的首地址
 - 数组是数据的容器，而非数据本身
 - 数组元素就是数组中存储的数据，一般有多个
 - 数组下标(索引)就是元素在数组中的位置，从0开始
 - 数组元素可以用“数组名[下标]”标识
 - 数组元素的个数称为数组长度



基本概念

- `int arr[5] = {10, 20, 30, 40, 50};`



数组的定义及初始化

- 数组的定义

元素类型 数组名[长度];

```
int arr[10];
```

arr是数组，arr[i]是数组元素，i是元素下标，从0到数组长度-1
数组下标>=数组长度，结果不确定，段错误/覆盖其它/没效果



数组的定义及初始化

- 数组的初始化

```
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
int arr[10] = {1};
```

```
int arr[10] = {};
```

未显示初始化的元素一律初始化为适当类型的0

如果不初始化，则必须指定长度

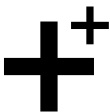
```
int arr[] = {1, 2, 3, 4, 5};
```

```
int arr[5];
```



数组的使用

- 数组元素的下标为从0开始的连续整数
- 利用针对下标的循环处理数组会很方便
- 声明数组的同时用{0}做清零初始化
- 通过sizeof(数组)/sizeof(数组元素)计算数组长度
- 对数组的访问常常可以归结为对下标的处理



练习时间

输入10个整数，逆序输出



数组的使用

- 输入流缓冲区问题

用scanf("%d",&i)读取十进制整数，如果所输入的不是合法十进制整数字符0-9，scanf函数会出错返回，并将非法字符留在输入流缓冲区中。这有可能对后续scanf函数的调用造成不利影响。

为此可通过如下方法将输入流缓冲区中的非法字符**清除**掉：

```
scanf ("%*[^\\n]"); // 忽略全部字符直到换行符
scanf ("%*c");      // 忽略换行符
```



数组的使用

- 输出流缓冲区问题

调用printf函数并不会立即输出，而只是将需要输出的信息放入输出流缓冲区中，直到以下情况之一发生时才**实际**输出缓冲区中的内容：

- ✓ 遇到换行符
- ✓ 程序结束
- ✓ 输出流缓冲区满
- ✓ 需要输入
- ✓ 调用fflush函数



练习时间

输入某班同学(人数不确定)的考试成绩，保存到数组中，输出总分、平均分和最高分



动态数组

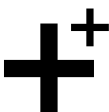
- C99支持动态数组，亦称变长数组，即在定义数组时通过变量指定数组的长度

```
int n;  
scanf ("%d", &n);  
int a[n]; // C99动态数组
```



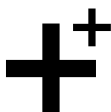
练习时间

分别用素数定义法和筛法求给定数以内的全部素数



多维数组

- 二维数组
 - 由多个一维数组组成的数组
 - 二维数组的每个元素都是一维数组
- 三维数组
 - 由多个二维数组组成的数组
 - 三维数组的每个元素都是二维数组
- N维数组
 - 由多个N-1维数组组成的数组
 - N维数组的每个元素都是N-1维数组



二维数组

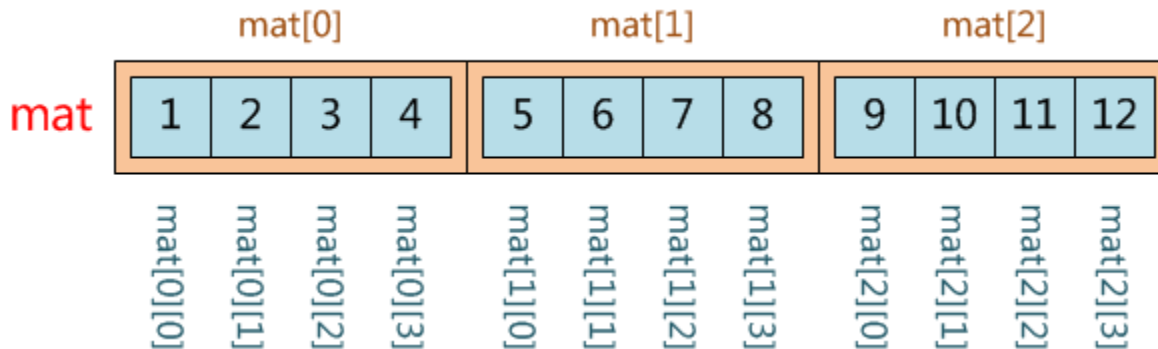
- `int mat[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};`

	列	0	1	2	3
0	行	1	2	3	4
1		5	6	7	8
2		9	10	11	12



二维数组

- `int mat[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};`



二维数组

- 二维数组的定义及初始化

元素类型 数组名[二维数组长度][一维数组长度] = {{...},{...},...};

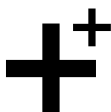
- 二维数组最右边的[]内必须有长度

```
int mat[][4] = {{1,2,3,4},{5,6,7},{9,10}};
```



练习时间

输入12个整数，构建一个3X4的矩阵A，再
输入12个整数，构建一个4X3的矩阵B，输
出矩阵 $C=AXB$



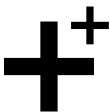
“

函数

”

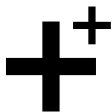
全程目标

- 基本概念
- 函数定义
- 函数声明
- 函数调用
- 递归与递推

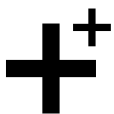
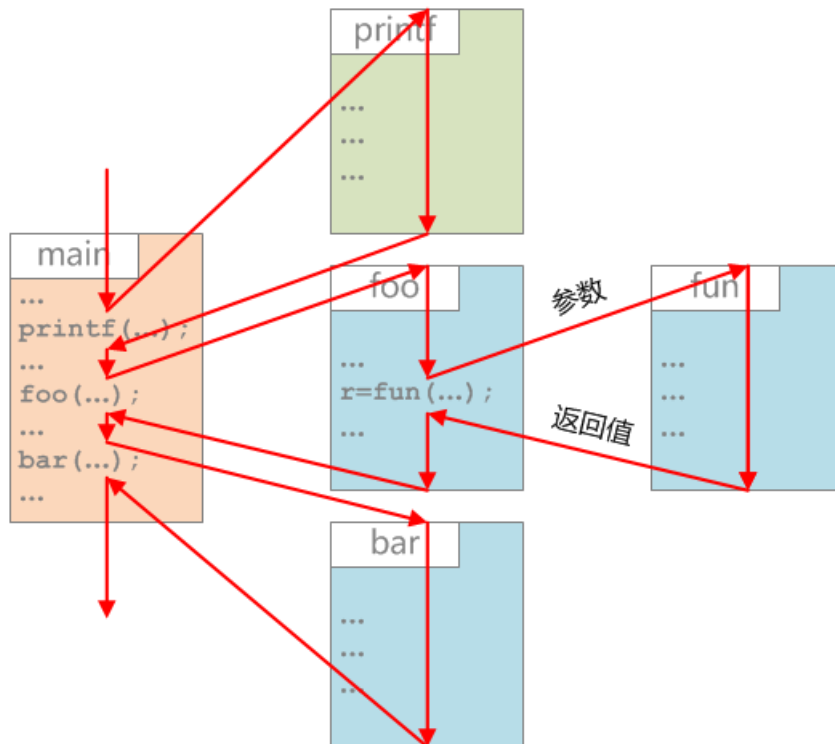


基本概念

- 函数就是一系列语句的组合，包括函数名、返回值、形参表和函数体
- main函数、库函数和自定义函数
- 程序的全部工作都由函数完成，函数式语言
- 函数不能嵌套定义，但可以相互调用
- main函数可以调用其它函数，但不能被其它函数调用
- 函数之间通过参数和返回值交换数据
- 函数也可以没有参数和(或)返回值



基本概念



函数定义

```
返回值类型 函数名(形参表) {  
    函数体;  
}
```

- ✓ 函数的返回值类型默认为int，若没有返回值，则需要用void标明
- ✓ 若函数的返回值类型为void，则不需要return任何值，或者省略return语句
- ✓ 若函数的返回值类型不是void，但没有通过return语句显式返回任何值，则会返回一个不确定的值
- ✓ 若函数的返回值类型与return值的类型不一致，则会发生类型转换



函数声明

- 函数在**使用前必须声明**，声明可以是隐式的，隐式声明的固定形式为
 - `int f ();` // 该函数可接受任意参数，并返回整型值
- 如果函数的返回类型不是int，最好对其做**显式声明**，即函数的原型
 - **返回值类型 函数名 (形参表);**
- 声明函数时，省略参数表，表示其**可接受任意**参数；参数表为void，表示其**不接受任何**参数
- 下面代码调用上面定义的函数，可以不再单独声明



函数调用

- 函数在定义时使用的参数叫形参，调用函数时传入的参数叫实参
- 函数调用时将实参传递给形参的过程，类似于把实参赋值给形参
- 形参只是实参的一份拷贝，而非实参本身，这种参数传递方式叫值传递

```
fun(actual);
```

actual

10

值传递

```
void fun(int formal){
    ...
}
```

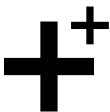
formal

10



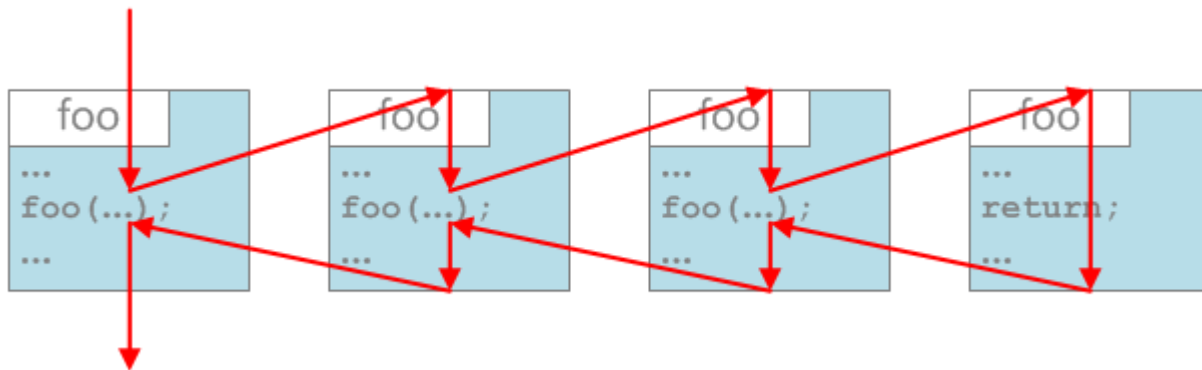
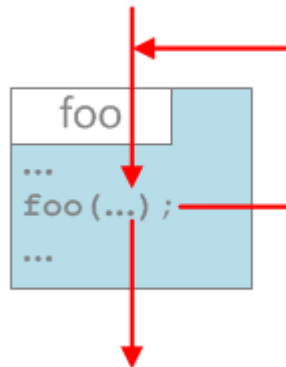
函数调用

- 对于隐式声明的函数，若实参的类型是 char/short/float，则会被自动提升为 int/int/double
- 传递数组参数，需要两个形参，第一个形参是不指定长度的数组，第二个形参是数组的长度
- return语句退出所在函数，exit函数退出整个程序，后者需要包含stdlib.h
- 通过man命令可以查看库函数的手册页，有时候需要-S2/3



递归与递推

- 函数在函数体内调用其自身称为递归调用该函数称为递归函数
- 调用递归函数的过程逐层调用，逐层返回



递归与递推

- 递推就是循环迭代
- 递归有可能形成无限递归，或者增加算法的时间复杂度，因此使用递归时，需要注意：
 - 必须有递归终止条件
 - 必须保证应用递归确实使算法得到简化
- 经典递归问题
 - 汉诺塔



练习时间

1. 根据如下迭代公式：

$$R_{n+1} = (2R_n + X/R_n^2)/3, R_1 = X$$
 实现计算立方根的函数
`double cbirt (double x);`
2. 实现计算排列数和组合数的函数：
`int arrange (int n, int m);`
`int combine (int n, int m);`



“

作用域与可见性

”

全程目标

- 局部变量与全局变量
- 静态变量与静态函数
- 进程空间的内存布局



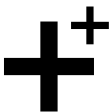
局部变量与全局变量

- 在函数内部定义的变量(包括形参)叫局部变量
- 在函数外部定义的变量叫全局变量
- 局部变量在函数执行期间有效，仅在函数内部可见
- 全局变量在进程运行期间有效，在所有函数中可见
- 在其它文件中访问全局变量需要使用外部声明
- 局部变量和全名变量同名，优先选择局部变量



静态变量与静态函数

- **静态局部变量**不会因函数返回而被回收，进程退出才被回收，但其可见性仍然局限于函数内部
- **静态全局变量**和**静态函数**的可见性，仅局限于定义该变量和函数的源文件，即使使用外部声明



作用域与可见性

	作用域	可见性
局部变量	函数/块	函数/块
静态局部变量	进程	函数/块
全局变量	进程	所有文件(外部变量)
静态全局变量	进程	定义文件
全局函数	进程	所有文件
静态全局函数	进程	定义文件

注意：C语言中既没有局部函数，也没有成员函数！



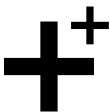
进程空间的内存布局

- **程序**是保存在磁盘上的可执行文件
- 将可执行文件加载到内存形成**进程**
- 一个程序可以同时**实例化**多个进程
- 进程在内存中的布局构成**进程映像**



进程空间的内存布局

- 进程在内存中的布局，从低地址到高地址依次为：
 - 代码区(text)
可执行指令
字面值常量、具有常属性的全局和静态变量
进程中唯一的只读区域
 - 数据区(data)
被初始化的全局和静态变量
 - BSS区
未初始化的全局和静态变量
进程一经加载此区即被清零

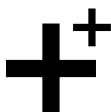
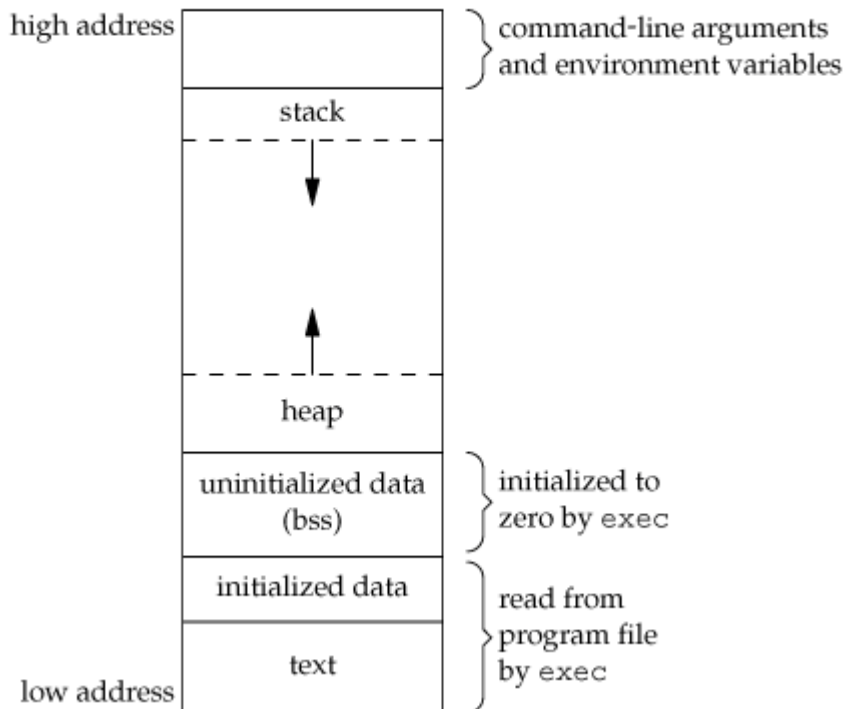


进程空间的内存布局

- 进程在内存中的布局，从低地址到高地址依次为：
 - 堆区(heap)
动态内存分配
从低地址向高地址扩展
 - 栈区(stack)
局部变量(包括函数的参数和返回值)
从高地址向低地址扩展
 - 参数与环境区
命令行参数和环境变量
- 通常将数据区和BSS区合称为静态区



Memory layout on an Intel-based Linux system



练习时间

用全局数组实现一个内存栈，后进先出
要求提供如下函数：

压入：void push (int data);

弹出：int pop (void);

栈顶：int top (void);

判空：int empty (void);

判满：int full (void);



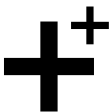
“

指针

”

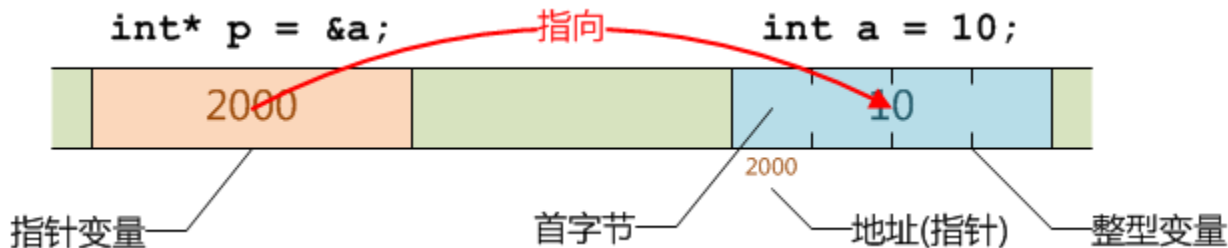
全程目标

- 地址、指针和指针变量
- 取地址与解引用
- 野指针与空指针
- 指针计算
- 指针与函数
- 指针与数组
- 常量指针与指针常量



地址、指针和指针变量

- 内存以**字节**为单位，不同类型数据的字节数不同
- 为了访问内存中的数据，必须为每个字节**编上号**
- 字节的编号就是**地址**，每个字节都有唯一的地址

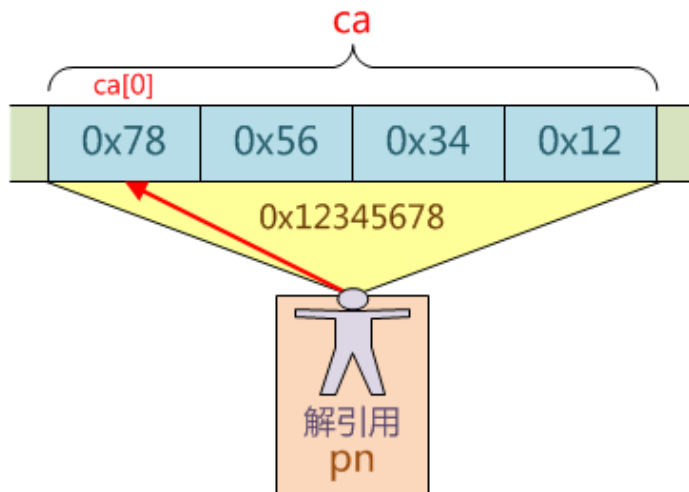


- 多字节数据，将其**首字节**地址作为该数据的地址
- 根据地址可以找到相应的数据，故地址亦称**指针**
- 将指针存放在一个变量中，该变量称为**指针变量**

地址、指针和指针变量

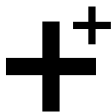
- 指针变量的类型取决于该变量所指向**目标的类型**
- 指针变量的类型与其目标的类型**不一定**严格一致
- 指针变量的类型决定了指针的**视野**、**行为**和**规则**

```
char ca[4] = {  
    \x78, \x56, \x34, \x12};  
int* pn = &ca[0];  
printf ("%#x\n", *pn);  
// 0x12345678
```



取地址与解引用

- **取地址**，即通过取地址操作符“&”，获取其操作数变量的地址
如：`int* p = &a;`
- **解引用**，亦称取目标，即通过解引用操作符“*”，获取以其操作数的值为地址的变量
如：`*p = 20;`
- 注意星号“*”在不同上下文中的不同语义
 - `c = a * b;` // 将a和b**相乘**
 - `int* p;` // 说明p是指向int类型变量的**指针**
 - `*p = 20;` // 表示指针p所指向的**目标**



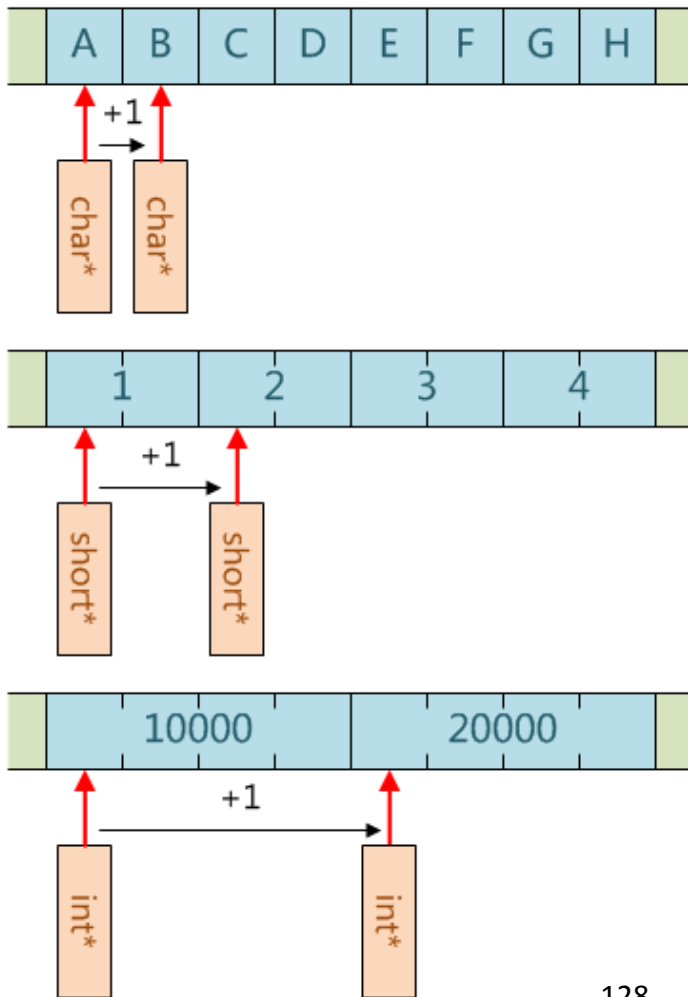
野指针与空指针

- 野指针，即指向**不可用**内存区域的指针
- 操作野指针将导致**未定义**的结果，构成潜在的风险
- 产生野指针的原因
 - 指针变量没有被初始化
 - 指针变量所指向的内存已被释放
- 空指针，即值为**0**的指针，可用宏**NULL**表示
- 任何情况下，操作空指针的结果都是**确定的**——崩溃——操作系统保证这一点
- 空指针比野指针更适合作为指针**有效性**的判断依据



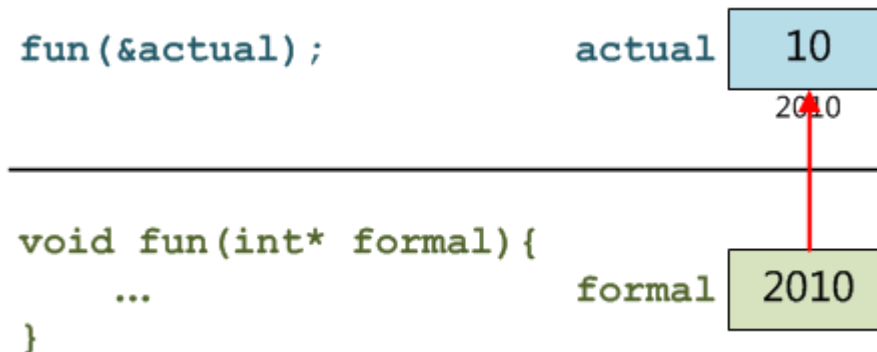
指针计算

- 指针支持加减整数、关系比较和相减运算
- 指针计算的单位由指针的**类型**决定
- 指针计算结果的安全性取决于程序设计者



指针与函数

- 可以将函数的形参定义为指针，并向其传递**实参的地址**，以达到在不同函数中访问同一个变量的目的



- 可以从函数中返回指针，但不要返回**指向局部变量的指针**，因为该变量的内存在函数返回后即被释放



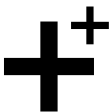
指针与数组

- 数组名本身就是一个**指针**，代表数组的**首元素地址**
- 对数组元素进行下标访问的本质，就是对数组名和下标做**指针计算**的结果**解引用**
 - `arr[i]`等价于`*(arr+i)`
- 与通常意义上的指针变量不同
 - 数组名是个**指针常量**，不能通过再次赋值令其指向其它数据
 - 数组名在某些上下文具有**容器语义**，即代表数组整体而不仅仅是首元素的地址，如`sizeof`



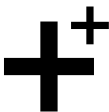
常量指针与指针常量

- const型变量
 - 被const关键字修饰的变量具有只读属性
 - 必须在定义的同时初始化
 - 只能做右值，不能做左值



常量指针与指针常量

- 当const作用于指针
 - **常量指针**，指向常量的指针，指针目标只读
 - ✓ `const int*`或`int const*`
 - ✓ 常量指针常做为函数的输入参数，在避免值复制传递参数开销的同时，有效防止在函数中意外地修改实参
 - **指针常量**，指针类型的常量，指针本身只读
 - ✓ `int* const`
 - ✓ 数组名就是指针常量
 - **常量指针常量**，指针的目标和指针本身都只读
 - ✓ `const int* const`或`int const* const`



练习时间

如下代码的输出是什么？

```
int main (void) {  
    int a[] = {1,2,3,4,5};  
    int* p = a, *q = &a[8];  
    printf ("%d\n", a[(q-p)/2]);  
    return 0;  
}
```



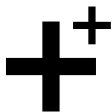
“

字符串

”

全程目标

- 字符串的存储
- 字符串的表示
- 字符串的输入和输出
- 字符串函数
- 字符串数组
- 命令行参数



字符串的存储

- C语言存储字符串的方式
 - 将字符串中每个字符的ASCII码按先后顺序存储在一段连续的内存中，每个字符占一个字节，最后用**空字符**，即ASCII码为0的字符**结尾**

72	101	108	108	111	44	32	87	111	114	108	100	32	33	0
H	e	l	l	o	,		W	o	r	l	d		!	

- 注意：一个汉字字符包含多个字节，因编码而异



字符串的表示

- C语言表示字符串的方式

- 字面值方式

- ✓ 用一对双引号括起来，自动追加结尾空字符
 - ✓ 直接引用字符串字面值在代码区中的首地址
 - ✓ 不能直接作为变量，但可以赋值给指针变量
 - ✓ 字面值过长，可以用"XXX" "YYY"形式连接

- 字符数组方式

- ✓ 数组名即字符串首地址
 - ✓ 可以用字面值初始化，也可以用{}初始化
 - ✓ 用{}初始化需要手动显示注明结尾空字符



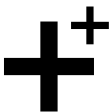
字符串的表示

- C语言表示字符串的方式
 - 字符指针方式
 - ✓ 用字符指针存放字符串的首地址
 - ✓ 可以指向字面值字符串，也可以指向字符数组字符串
- 三种表示方式的区别
 - 字面值在代码区，只读，内容相同只存一份
 - 字符数组在堆栈区，可写，但数组名是常量
 - 字符指针既可以指向代码区中的字面值，只读，也可以指向堆栈区中的字符数组，可写，指针本身只要没有常属性，也可以被修改



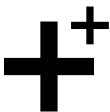
字符串的输入和输出

- 字符串的输入
 - `scanf ("%s", ...);`
 - `char* gets (char* s);`
从标准输入读取一行字符串，读走换行符即返回，
并将所读到的换行符替换为结尾空字符
- 字符串的输出
 - `printf ("%s", ...);`
 - `int puts (const char* s);`
向标准输出写入一行字符串，并追加一个换行符



字符串函数

- 标准库提供一套专门针对字符串的函数
 - 字符串头文件
`#include <string.h>`
 - 字符串长度
`size_t strlen (const char* s);`
 - 字符串拷贝
`char* strcpy (char* dest, const char* src);`
`char* strncpy (char* dest, const char* src, size_t n);`
 - 字符串连接
`char* strcat (char* dest, const char* src);`
`char* strncat (char* dest, const char* src, size_t n);`



字符串函数

- 标准库提供一套专门针对字符串的函数

- 字符串比较

```
int strcmp (const char* s1, const char* s2);  
int strncmp (const char* s1, const char* s2, size_t n);  
int strcasecmp (const char* s1, const char* s2);  
int strncasecmp (const char* s1, const char* s2, size_t n);
```

- 使用字符串函数时需要注意

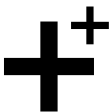
- ✓ 所有字符串都必须以空字符结尾
- ✓ 字符串长度以char为单位且不包含结尾空字符
- ✓ 拷贝和连接的目标缓冲区必须可写且足够大
- ✓ 不能使用关系运算符比较字符串



练习时间

实现字符串合并函数，将两个有序字符序列合并为一个，结果依然有序

```
void merge (const char* p1,  
            const char* p2, char* p3);
```



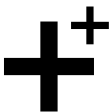
字符串数组

- 二维数组形式的字符串数组

```
char sa[][10] = {"beijing", "tianjin",  
                 "shanghai", "chongqing"};
```

sa

b	e	i	j	i	n	g	\0	\0	\0
t	i	a	n	j	i	n	\0	\0	\0
s	h	a	n	g	h	a	i	\0	\0
c	h	o	n	g	q	i	n	g	\0



字符串数组

- 指针数组形式的字符串数组

```
const char* sa[] = {"beijing", "tianjin",  
                    "shanghai", "chongqing"};
```

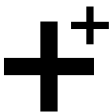
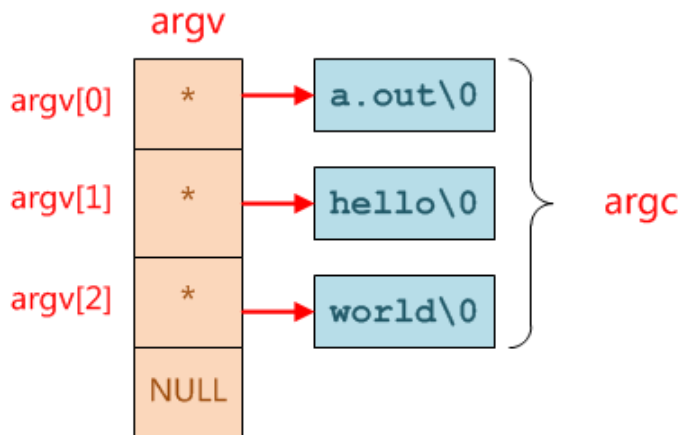


命令行参数

- 通过main函数的参数接受命令行信息

```
int main (int argc, char* argv[]) { ... }
```

```
$ a.out hello world
```



练习时间

迷你备忘录：

输入月内日期和事件，按日期的先后顺序打印全部事件列表



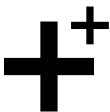
“

预处理与 大型程序

”

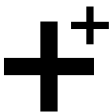
全程目标

- 文件包含指令
- 宏定义指令
- 预定义宏
- 条件编译指令
- 头文件与头文件卫士
- make与makefile



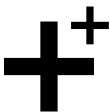
文件包含指令

- #include
 - 将所包含文件的内容粘贴到该指令处
 - 尖括号包含：#include <xxx.h>
先找-I目录，再找系统目录
适用于系统头文件
 - 双引号包含：#include "xxx.h"
先找-I目录，再找当前目录，最后找系统目录
适用于自己编写的头文件



宏定义指令

- #define
 - 用一个标识符来表示一个字符串，谓之宏
 - 被定义为宏的标识符称为宏名
 - 预处理器对程序中出现的所有宏名，都用宏定义中的字符串去替换，这个过程叫做宏替换或者宏扩展
 - 宏替换只是针对源代码的文本替换，不涉及类型检查，更不会计算表达式或者调用函数
 - 宏定义分为无参宏定义和有参宏定义两种
 - ✓ #define 宏名 字符串
 - ✓ #define 宏名(形参表) 字符串



宏定义指令

- 无参宏定义
 - 宏替换只是简单的文本替换，预处理器对所替换的内容**不做任何检查**，如有错误，只能在编译时发现
 - 宏定义不是说明或语句，**行末不必加分号**，若加上分号则连分号一起做宏替换
 - 宏定义必须写在**所有函数之外**，其作用域从宏定义指令开始一直到源程序结束
 - 宏名在源程序中若用**引号**括起来，则预处理器不对其做宏替换
 - 在宏定义的字符串中可以**使用已定义过的宏名**



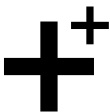
宏定义指令

- 无参宏定义
 - 宏名习惯用大写字母表示，以区别于变量和函数，但也允许用小写字母
 - 定义常量，便于修改
 - 定义类型，简化书写
 - ✓ 用typedef定义类型别名效果更好
 - 定义模式，语言扩展
 - ✓ 不要滥用



宏定义指令

- 有参宏定义
 - 宏名和形参表必须在同一行且中间不能有空格
 - 宏形参不分配内存空间，因此无需说明其类型
 - 调用带参宏只是符号替换，不存在参数传递问题
 - 宏调用中的实参可以是表达式，但对实参表达式并不计算，直接用它替换宏定义中的形参
 - 宏定义字符串内的形参，通常要用括号括起来，以避免出错
 - 宏定义必须书写在一行中，如有必要可加续行符“\”



宏定义指令

- 有参宏定义
 - 调用有参宏的实参表达式中不要使用++/--运算符
 - 宏定义字符串中的“#形参”表示将形参扩展为用双引号括起来的实参表达式
 - 宏定义字符串中的“##形参”表示将形参扩展为实参表达式并与前面的字符粘连在一起
 - 用有参宏取代函数可以提高程序的执行效率，但会占用更多的磁盘和内存空间，因此更适于实现那些频繁使用的简单功能



宏定义指令

- `#undef`
 - 取消一个已定义的宏，令其宏名处于未定义状态



预定义宏

预定义宏	说明
<code>__BASE_FILE__</code>	正在编译的源文件名
<code>__FILE__</code>	所在文件名
<code>__LINE__</code>	所在行号
<code>__FUNCTION__</code> / <code>__func__</code>	所在函数名
<code>__DATE__</code>	编译日期
<code>__TIME__</code>	编译时间
<code>__INCLUDE_LEVEL__</code>	包含层数，从0开始
<code>__STDC__</code>	是否支持标准C，1/0



条件编译指令

条件编译指令	说明
#if 常量表达式	如果常量表达式的值非零
#ifdef 宏名	如果宏名已定义
#ifndef 宏名	如果宏名未定义
#elif 常量表达式	否则如果常量表达式的值非零
#else	否则
#endif	结束判断
常量表达式	字面值/defined(宏名)/&&/ /!

根据不同条件，编译不同部分，产生不同目标代码



头文件与头文件卫士

- 头文件中放什么？
 - 包含公共头文件
`#include <sys/types.h>`
 - 宏定义
`#define TRUE 1`
 - 类型定义
`typedef int BOOL`
 - 函数声明
`BOOL ConnectServer (const char* URL);`
 - 外部变量声明
`extern const char* g_dns;`



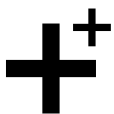
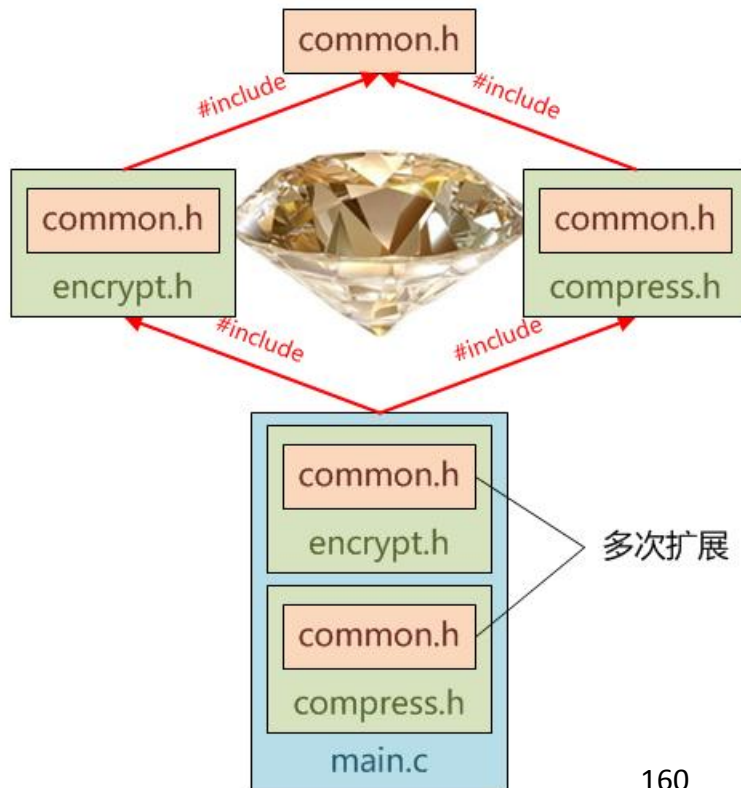
头文件与头文件卫士

- 头文件中不放什么？
 - 全局变量的定义
 - 函数的定义
- 若该头文件被多个源文件包含，定义在头文件中的全局变量和函数，将在每个包含该头文件的源文件中各被定义一次，这将在链接阶段引发重定义错误



头文件与头文件卫士

- 钻石包含可能导致编译错误
- 公共头文件在源文件中会被**多次扩展**
- 公共头文件中的代码会在源文件中**重复出现**
- 为避免重定义错误需要使用**头文件卫士**



头文件与头文件卫士

- 头文件卫士

```
#ifndef _COMMON_H
```

```
#define _COMMON_H
```

```
...
```

```
#endif // _COMMON_H
```



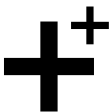
make与makefile

- make是一种文件转换工具，通过某种操作，将依赖文件转换为目标文件
- make命令可以根据makefile脚本中定义的规则，完成从依赖文件到目标文件的转换
- 只有当依赖文件比目标文件新时，make才会重新生成目标文件
- 具体到C语言程序，可以认为可执行程序依赖于目标模块，而目标模块又依赖于源程序和头文件
- 从源程序和头文件到目标模块，以及从目标模块到可执行程序所需要的操作就是各种GCC命令



make与makefile

- makefile需要描述的就是目标、依赖以及从依赖产生目标所需要执行的各种命令
- makefile的基本语法要素
 - 目标：依赖
 - <制表符>命令1
 - <制表符>命令2
- 一个makefile可以包含多个目标，通过make命令的参数指定期望实现的目标，缺省实现第一个
- 命令可以是GCC命令，也可以是普通SHELL命令
- 目标的依赖和/或命令可以为空



make与makefile

- makefile样例

```
editor: main.o text.o
    gcc -o editor main.o text.o
main.o: main.c def.h
    gcc -c main.c
text.o: text.c com.h
    gcc -c text.c
install: editor
    mv editor /usr/local
clean:
    rm editor *.o
```



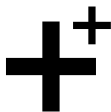
“

复合类型与 高级指针

”

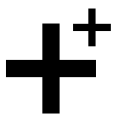
全程目标

- 结构、联合与枚举
- 指针数组与数组指针
- 多维数组与指针
- 二级指针与多级指针
- 泛型指针
- 动态内存分配与释放
- 函数指针



结构

- 结构是一种由若干成员组成的构造类型
- 结构的成员既可以是基本类型的数据，也可以又是结构类型的数据
- 结构是一种构造而成的数据类型，在使用结构之前必须先行定义，因此结构也是一种自定义数据类型
- 结构类型的数据在存储上类似于数组，所有成员按其被声明的顺序，由低地址到高地址连续(或接近连续)存放
- 与数组不同，结构的成员可以是不同类型的数据，因此结构也是一种复合数据类型



结构

- 直接定义结构型变量

```
struct {
    数据类型1 成员1;
    数据类型2 成员2;
    ...
} 变量;
```



结构

- 先定义结构类型，再用该类型定义变量

```
struct 结构类型 {  
    数据类型1 成员1;  
    数据类型2 成员2;  
    ...  
};  
  
struct 结构类型 变量;
```



结构

- 通过typedef定义类型别名，再用该别名定义变量

```
typedef struct [结构类型] {
    数据类型1 成员1;
    数据类型2 成员2;
    ...
} 类型别名;
```

类型别名 变量;



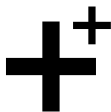
结构

- 结构变量的初始化

```
typedef struct Date {  
    int year;  
    int mon;  
    int day;  
}    DATE;
```

```
typedef struct Student {  
    char name[128];  
    int age;  
    DATE birthday;  
}    STUDENT;
```

```
STUDENT student = {"Liming", 25, {1988, 6, 20}};
```



结构

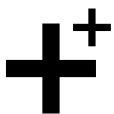
- 结构数组的初始化

```
STUDENT students[] = {  
    {"Liming", 25, {1988, 6, 10}},  
    {"Liukai", 22, {1991, 8, 22}},  
    {"Wangli", 27, {1986, 2, 12}},  
    {"Zhaole", 30, {1983, 1, 17}},  
    {"Xiemin", 19, {1994, 9, 29}}  
};
```



结构

- 指向结构变量和结构数组的指针
 - 结构指针的计算以整个结构的大小为单位
- 访问结构的成员
 - 通过结构变量访问其成员，用成员访问运算符 “.”
 - 通过结构指针访问其目标的成员，用间接成员访问运算符 “->”
- 如果函数的参数是结构类型，那么形参只是实参的拷贝，在函数内部无法修改实参的成员
- 如果函数的参数是结构指针，并接受实参的地址，那么就可以在函数内部修改实参的成员



练习时间

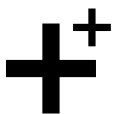
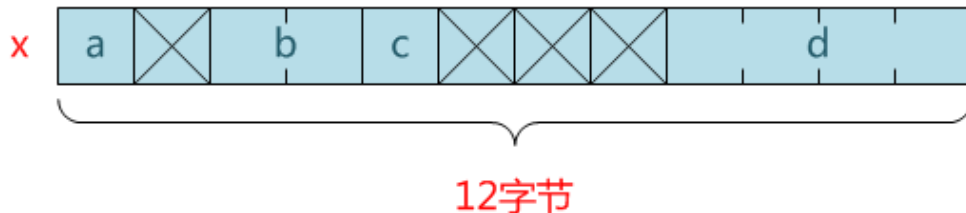
定义一个点结构和一个矩形结构
定义针对这两种结构的读取和打印函数
定义判断矩形合法性的函数
定义判断点是否在矩形内的函数



结构

- 内存对齐
 - 结构中每个成员的起始地址，必须是其自身大小的整数倍，超过4字节的按4字节算(32位系统)，为此可能需要在中间填充若干字节

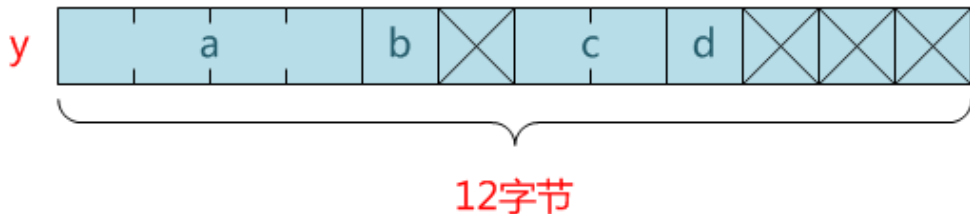
```
struct X {  
    char a;  
    short b;  
    char c;  
    int d;  
} x;
```



结构

- 内存补齐
 - 结构的总字节数，必须是其最大成员大小的整数倍，超过4字节的按4字节算(32位系统)，为此可能需要在末尾填充若干字节

```
struct Y {  
    int    a;  
    char   b;  
    short  c;  
    char   d;  
} y;
```



结构一位域

- 位域可以指定每个成员的**二进制位数**，以更加紧凑的方式存储数据，节省宝贵的内存空间
- 位域的用法和结构一样，但位域**不能**对成员**取地址**
- 位域成员的数据类型必须是**整型类**
[unsigned] char/short/int/long/long long

```
#pragma pack (1)
```

```
struct {
```

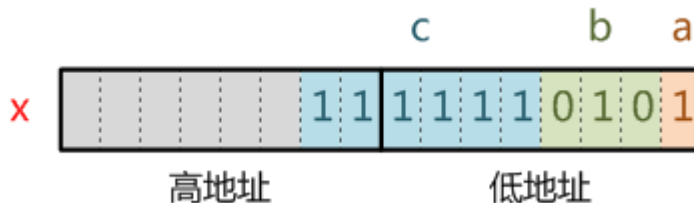
```
    int a : 1;
```

```
    int b : 3;
```

```
    int c : 6;
```

```
};
```

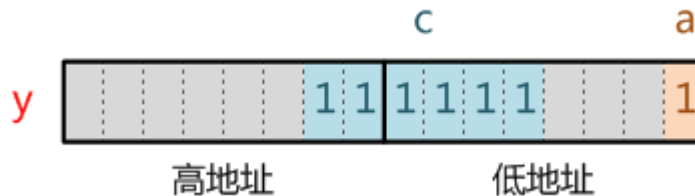
```
x.a = 1; x.b = 2; x.c = 63;
```



结构一位域

- 位域中可以包含空域，即不使用的二进制位

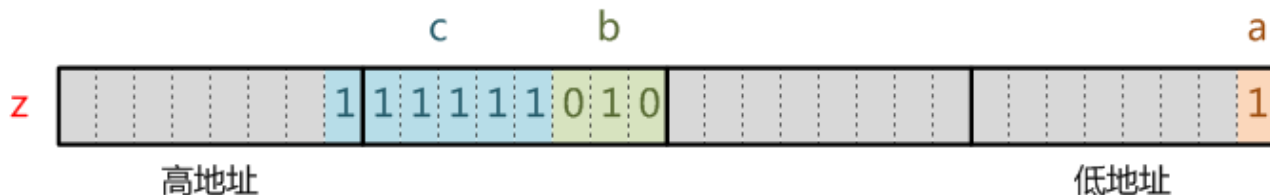
```
#pragma pack (1)
struct Y {
    int a : 1;
    int   : 3;
    int c : 6;
} y;
y.a = 1; y.c = 63;
```



结构一位域

- 位域中可以指定域的起始边界

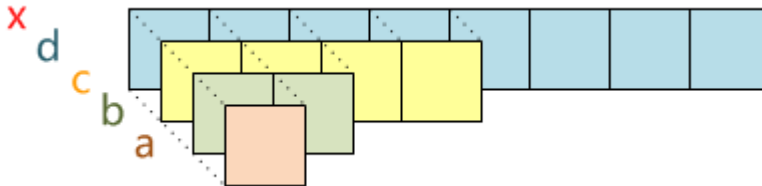
```
#pragma pack (1)
struct Z {
    int a : 1;
    short : 0;
    int b : 3;
    int c : 6;
} z;
z.a = 1; z.b = 2; z.c = 63;
```



联合

- 联合的语法和用法与结构非常相似
- 联合与结构的主要区别在于，联合的所有成员从同一个地址开始，即**共享**同一块内存区域
- 联合变量的字节数就是其**最大成员**的字节数
- 利用联合可以将同一份数据解释为**不同的类型**

```
union X {  
    char    a;  
    short   b;  
    int     c;  
    double  d;  
    x;  
}
```



枚举

- 枚举是一个有限整型常量的列表
- 每个枚举值都是一个符号常量，默认从0开始
- 枚举值也可以人为设置，没有显示设置的枚举值=上一个枚举值+1
- 枚举可以提升程序的可读性

```
typedef enum Color {  
    CYAN,           // 0  
    MAGENTA,        // 1  
    YELLOW = 10,    // 10  
    BLACK           // 11  
} COLOR;
```



练习时间

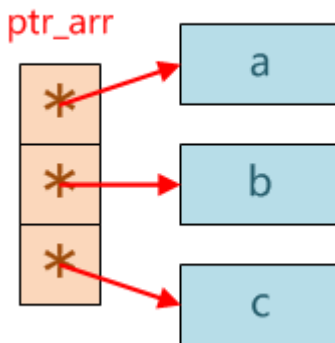
录入学生(10个)成绩单，每条成绩记录包括：
学生姓名，性别，年龄，学号，分数，
分别列出90分以上、80-89分、70-79分、
60-69分，以及60分以下的成绩记录



指针数组与数组指针

- 指针数组是每个元素都是**指针**的数组

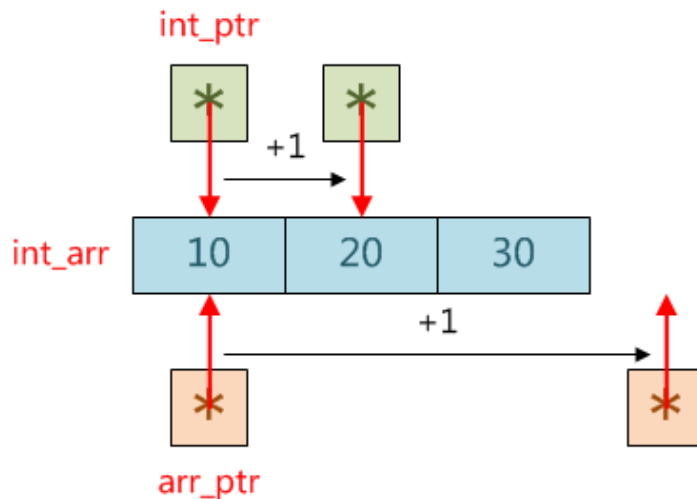
```
int a, b, c;  
int* ptr_arr[] = {&a, &b, &c};
```



指针数组与数组指针

- 数组指针是指向**整个**数组的指针

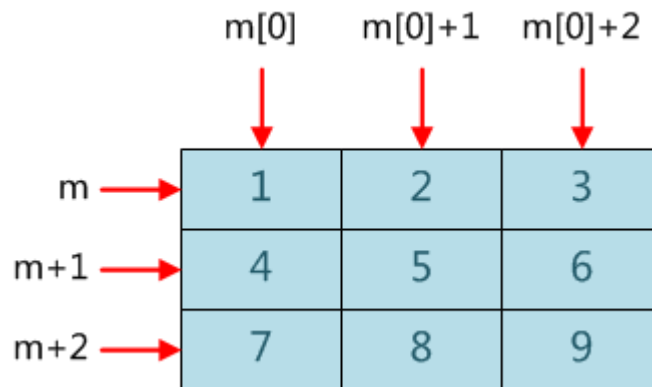
```
int int_arr[] =
    {10, 20, 30};
int* int_ptr =
    int_arr;
int (*arr_ptr)[3] =
    &int_arr;
```



多维数组与指针

- N维数组的数组名是指向N-1维数组的数组指针

```
int m[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```



- 对于任意二维数组`m`

$$m[i][j] = *(m[i] + j) = *((m + i) + j)$$



二级指针与多级指针

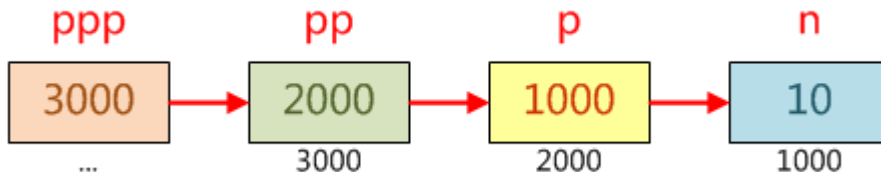
- 指针变量的地址即**指针的指针**
- 将一级指针变量的地址保存在另一个指针变量中即构成**二级指针**
- 二级指针是指向**一级指针的指针**
- 常用二级指针类型的函数形参，接受一级指针地址形式的实参，以**修改调用者指针的目标**，或为其**分配资源**
- 对一维数组的数组名取地址，得到的不是二级指针而是**数组指针**



二级指针与多级指针

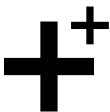
- N级指针是指向N-1级指针的指针

```
int n = 10;  
int* p = &n;           // 一级指针  
int** pp = &p;         // 二级指针  
int*** ppp = &pp;      // 三级指针
```



泛型指针—void*

- 仅存储内存地址，**不指定目标类型**
- 从任意类型指针**隐式转换**，类型泛化
- **显式转换**为任意类型指针，类型特化
- 目标类型不确定，**不能直接解引用**
- ANSI**禁止**对泛型指针做指针**计算**
- GNU**允许**对泛型指针做指针**计算**，以**1字节**为单位
- 某些标准库函数，以泛型指针作为**参数**和**返回值**
- C语言实现**泛型算法**的基础



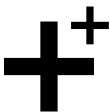
动态内存分配与释放

- 分配内存
 - #include <stdlib.h>
`void* malloc (size_t size);`
 - 从堆中分配size字节内存
 - 成功返回该内存块的起始地址，失败返回NULL
 - 对所分配的内存不做初始化
 - 若size取0，则返回NULL或者一个唯一的地址，保证后续对free函数的调用能够成功



动态内存分配与释放

- 分配数组并初始化
 - #include <stdlib.h>
`void* calloc (size_t nmemb, size_t size);`
 - 从堆中分配包含nmemb个元素的数组，其中每个元素占size字节
 - 成功返回该数组的起始地址，失败返回NULL
 - 对所分配数组的每个元素，用相应类型的0初始化
 - 若nmemb或size取0，则返回NULL或者一个唯一的地址，保证后续对free函数的调用能够成功



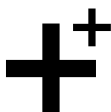
动态内存分配与释放

- 调整动态内存大小
 - #include <stdlib.h>
`void* realloc (void* ptr, size_t size);`
 - 将ptr所指向的动态内存大小调整为size字节，原内容保持不变，对新增部分不做初始化
 - 成功返回调整后内存块的起始地址，失败返回NULL
 - ptr必须是之前malloc/calloc/realloc函数的返回值
 - 若必须分配新内存，则原内存将被释放，原内容被复制到新内存中，返回新内存的起始地址
 - 若新内存分配失败，则原内存不被释放，原内容保持不变，但返回值为NULL



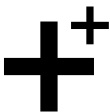
动态内存分配与释放

- 调整动态内存大小
 - 若ptr取NULL，则等价于**malloc**函数
 - 若size取0，则等价于**free**函数



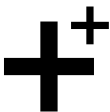
动态内存分配与释放

- 释放内存
 - #include <stdlib.h>
void free (void* ptr);
 - 释放ptr所指向的动态内存
 - ptr必须是之前**malloc/calloc/realloc**函数的返回值
 - 释放一块已被释放过的内存，将导致**未定义**的后果
 - 若ptr取**NULL**，则不执行任何操作
 - 所有通过动态内存分配得到的内存都必须释放，否则就会形成**内存泄漏**



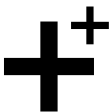
练习时间

录入多行文本，每行不超过255个字符，行数不限，输入“!”连成一行文本整体输出



函数指针

- 所有函数都存放在代码区，都有地址，即函数指针
- 函数名就是函数的符号地址，即函数指针常量
- 定义存放函数指针的指针变量，并用函数名初始化
 - 返回类型 (*函数指针变量) (形参表) = 函数名;
- 可以象通过函数名一样通过函数指针变量调用函数
 - 函数指针变量 (实参表);
- 函数指针可以作为函数的参数
 - 多态
 - 回调

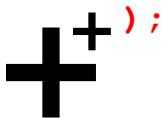


函数指针

- 快速排序

```
#include <stdlib.h>
```

```
void qsort (  
    void* base,           // 数组起始地址  
    size_t nmemb,         // 数组元素个数  
    size_t size,         // 数组元素大小  
    int (*compar) (       // 比较函数指针  
        const void*,     // 第一个待比较元素地址  
        const void*      // 第二个待比较元素地址  
    ) // 第一个待比较元素<=第二个待比较元素  
    // 返回负数/0/正数
```



练习时间

编写程序：录入10个学生的信息，每个学生包括姓名和年龄两个属性。要求按照年龄由小到大的顺序输出，年龄相同的学生，按照姓名由大到小的顺序输出



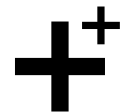
“

I/O流与标准库

”

全程目标

- I/O流的打开与关闭
- 格式化I/O
- 非格式化I/O
- 二进制I/O
- 文件位置与随机访问
- 可变参数表
- 标准库



I/O流的打开和关闭

- I/O流的打开

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

```
FILE* fopen (  
    const char* path, // 文件路径  
    const char* mode  // 打开模式  
);
```

成功返回I/O流指针，作为后续I/O流函数的参数，失败返回NULL



I/O流的打开和关闭

- 打开模式

- r** - 只读，文件必须存在，从头开始读
- w** - 只写，文件不存在就创建，存在就清空，从头开始写
- a** - 追加，文件不存在就创建，存在不清空，从尾开始写
- r+** - 读写，文件必须存在，从头开始读写
- w+** - 写读，文件不存在就创建，存在就清空，从头开始写读
- a+** - 追读，文件不存在就创建，存在不清空，从头开始读，从尾开始写
- t** - 纯文本(UNIX忽略)
- b** - 二进制(UNIX忽略)



I/O流的打开和关闭

- I/O流的关闭

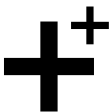
```
int fclose (
    FILE* fp // I/O流指针
);
```

成功返回0，失败返回EOF



I/O流的打开和关闭

- 系统每个进程缺省打开三个标准I/O流
 - 标准输入：`stdin`
 - 标准输出：`stdout`
 - 标准错误：`stderr`



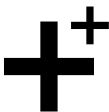
格式化I/O

- 格式化输出

```
int fprintf (  
    FILE*          stream, // I/O流指针  
    const char* format, // 格式字符串  
    ...            // 输出数据  
);
```

成功返回输出字符数，失败返回负数

```
int printf (const char* format, ...);  
int sprintf (char* str, const char* format, ...);  
int snprintf (char* str, size_t size,  
    const char* format, ...);
```



格式化I/O

- 格式字符串

%[标志][宽度][.精度][h|l|ll|L]类型标记

[标志]

- : 左对齐

+ : 输出正负号

: 输出进制前缀0/0x

0 : 用0补齐



格式化I/O

- 格式字符串

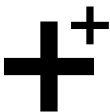
%[标志][宽度][.精度][h|l|ll|L]类型标记

[宽度]

整数部分，小数部分和小数点

[.精度]

小数部分，超出部分四舍五入



格式化I/O

- 格式字符串

%[标志][宽度][.精度][h|l|ll|L]类型标记

h : short int

l : long/double

ll : long long

L : long double

类型标记

c/d/u/o/x(X)/f/e/g/s/p



格式化I/O

- 格式化输入

```
int fscanf (  
    FILE*          stream, // I/O流指针  
    const char* format, // 格式字符串  
    ...            // 输入数据  
);
```

成功返回实际输入的数据项数，失败或遇到文件尾返回EOF

```
int scanf (const char* format, ...);  
int sscanf (const char* str, const char* format, ...);
```



格式化I/O

- 格式化输入
 - 以空白字符(空格、制表、换行)作为数据项的分隔符
`scanf ("%d%d%d%d", &a, &b, &c, &d);`
 - 匹配输入流中的下一个非空白字符
`scanf ("%d,%d+%d\n%d", &a, &b, &c, &d);`
 - 忽略输入流中数据项
`scanf ("%*d%d", &a);`
 - 根据指定字符集读取字符串
`scanf ("%[a-z]", s);`
`scanf ("%[^a-z]", s);`



非格式化I/O

- 输出字符

```
int fputc (  
    int    c,        // 字符  
    FILE*  stream    // I/O流指针  
);
```

成功返回实际输出的字符，失败返回EOF

```
int putc (int c, FILE* stream);  
int putchar (int c);
```



非格式化I/O

- 输入字符

```
int fgetc (  
    FILE* stream // I/O流指针  
);
```

成功返回实际输入的字符，失败或遇到文件尾返回EOF

```
int getc (FILE* stream);  
int ungetc (int c, FILE* stream);  
int getchar (void);
```



非格式化I/O

- 输出字符串

```
int fputs (  
    const char* s,      // 字符串首地址  
    FILE* stream // I/O流指针  
);
```

成功返回非负数，失败返回EOF，**不追加**换行符

```
int puts (const char* s); // 追加换行符
```



非格式化I/O

- 输入字符串

```
char* fgets (  
    char* s,          // 字符串缓冲区首地址  
    int    size,      // 字符串缓冲区大小  
    FILE* stream // I/O流指针  
);
```

最多读取size-1个字符，追加结尾空字符读到换行符返回，不把换行符换成空字符成功返回s，失败或遇到文件尾返回NULL

```
char* gets (char* s); // 把换行符换成空字符
```

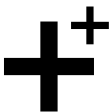


二进制I/O

- 二进制输出

```
size_t fwrite (  
    const void* ptr,      // 缓冲区地址  
    size_t      size,     // 元素字节数  
    size_t      nmemb,    // 期望输出元素数  
    FILE*       stream    // I/O流指针  
);
```

返回实际输出元素数，遇到错误，返回值比nmemb小或为0



二进制I/O

- 二进制输入

```
size_t fread (  
    void* ptr,      // 缓冲区地址  
    size_t size,    // 元素字节数  
    size_t nmemb,   // 期望输入元素数  
    FILE* stream    // I/O流指针  
);
```

返回实际输入元素数，遇到错误或文件尾，返回值比nmemb小或为0




文件位置与随机访问

- 设置文件位置

```
int fseek (  
    FILE* stream, // I/O流指针  
    long  offset, // 偏移字节数  
    int   whence  // 偏移起点  
                // SEEK_SET : 从文件头  
                // SEEK_CUR : 从当前位置  
                // SEEK_END : 从文件尾  
);
```

成功返回0，失败返回-1

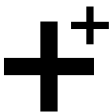
 `void rewind (FILE* stream); // fseek(stream, 0L, SEEK_SET)`

文件位置与随机访问

- 获取文件位置

```
long ftell (  
    FILE* stream // I/O流指针  
);
```

成功返回当前文件位置，失败返回-1



I/O流状态

- 是否到文件尾
 - `int feof (FILE* stream);`
- 是否出现错误
 - `int ferror (FILE* stream);`
- 清除I/O流错误状态
 - `void clearerr (FILE* stream);`
- 部分I/O流函数，仅根据返回值无法区分失败或遇到文件尾，可以通过`feof/ferror`函数做进一步判断
- 一旦I/O流出现错误即无法继续工作，可以通过`clearerr`函数清除错误状态，继续工作



可变参数表

- 可变参数表头文件
 - `#include <stdarg.h>`
- 可变参数表结构
 - `va_list`
- 可变参数表结构初始化宏
 - `void va_start (va_list ap, last);`
- 可变参数表结构枚举宏
 - `type va_arg (va_list ap, type);`
- 可变参数表结构终结化宏
 - `void va_end (va_list ap);`

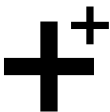


可变参数表

- 可变参数表编程范式

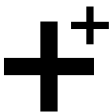
```
#include <stdarg.h>
```

```
void foo (<type> last, ...) {  
    va_list ap;  
    va_start (ap, last);  
    while (...) {  
        <type> x = va_arg (ap, <type>);  
    }  
    va_end (ap);  
}
```



标准库

- 截止目前，ISO C语言标准库共包括29个头文件
 - 1989年，C89标准库包括15个头文件
 - 1995年，C95标准库增加了3个头文件
 - 1999年，C99标准库增加了6个头文件
 - 2011年，C11标准库增加了5个头文件



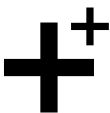
C89标准库(15)

标准库头文件	说明
assert.h	断言宏，在程序的调试版本中检测逻辑错误
ctype.h	字符分类
errno.h	错误码
float.h	依赖于实现的实型宏
limits.h	依赖于实现的整型宏
locale.h	本地化
math.h	数学
setjmp.h	非局部跳转setjmp/longjmp



C89标准库(15)

标准库头文件	说明
signal.h	信号
stdarg.h	可变参数表
stddef.h	常用类型和宏
stdio.h	I/O流
stdlib.h	数串转换、伪随机数、内存分配、进程控制等
string.h	字符串
time.h	日期和时间



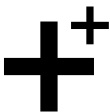
C95标准库(+3)

标准库头文件	说明
iso646.h	运算符别名宏
wchar.h	宽字符支持
wctype.h	宽字符分类



C99标准库(+6)

标准库头文件	说明
complex.h	复数计算
fenv.h	浮点环境
inttypes.h	各种位宽整型的格式化标志宏
stdbool.h	布尔类型宏
stdint.h	特定位宽数型
tgmath.h	泛型数学宏



C11标准库(+5)

标准库头文件	说明
stdalign.h	内存对齐
stdatomic.h	针对线程共享数据的原子操作
stdnoreturn.h	永不返回的函数
threads.h	多线程管理与同步，互斥锁、条件变量等
uchar.h	Unicode字符集支持



“

再见

”