基本概念

程序设计语言的功能

> 数据表达、流程控制

c语言的特点

> c语言是一种结构化语言
>
> c语言语句简洁紧凑，使用方便灵活
>
> c语言程序易于移植
>
> c语言有强大的处理能力
>
> 生成的目标代码质量高，运行效率高

实现问题求解的过程

> 问题设计与算法分析
>
> 编辑程序
>
> 编译
>
> 运行与测试

c语言语句组成

> 标识符：保留字（if、else等等）、用户自定义标识符（定义的变量名、函数名等等）
>
> 常量：整型123、实型12.34、字符常量'a'、字符串常量"hello world"
>
> 运算符：四则运算、求余、比较符、等号（赋值)等等；包括单目、双目（最常见）、三目
>
> 分隔符：分号、中括号、小括号、井号等等

合法自定义标识符的规则

1、只能由字母、数字和下划线组成

2、只能以字母或下划线开头，但是不建议下划线开头，因为编译器通常会定义一些下划线开头的标识符

3、不能使用保留字

c语言主要语法单位

表达式：由运算符与运算对象构成

变量定义

语句：单行、分支、循环、复合语句（一对花括号括起来的语句，例如循环体）等等

函数定义与调用

输入与输出

整型的储存

- **原码、补码、反码**

  整形数据一般在计算机内部以**补码**存储。

  正数：原码、补码、反码相同

  负数：原码符号位为1，其余位为绝对值的二进制表示；反码为符号位不变，其余位取反；补码为反码加1。

  有符号二字节整形存储范围为$[-2^7, 2^7-1]$

  例：对于`short a=32767`，`++a`=-32768 (补码加1后的结果)

  例：对于`unsigned short a=65535`，`++a`=0 (补码加1后的结果)

- **十进制、八进制、十六进制**

  十进制：首位不能是0

  八进制：首位必须是0

  十六进制：首位必须是0x或0X

- **整数的类型**

类型名	数据长度	取值范围	后缀	编码
int	32位（4字节）	-2^{31}~$2^{31}-1$		最高位为符号位
short	16位（2字节）	-2^{15}~$2^{15}-1$ (−32768~32767)		最高位为符号位
long	32位	-2^{31}~$2^{31}-1$	l或L	最高位为符号位
unsigned int	32位	0~$2^{32}-1$	u或U	
unsigned short	16位	0~$2^{16}-1$ (0~65535)		
unsigned long	32位	0~$2^{32}-1$	lu或LU	

注：

1、c语言并未规定各类整形数据的长度，只要求short不长于int，int不长于long。上表中的数据长度与Dev-C++编译系统的规定一致。在Turbo C编译系统中int和unsigned int都只有16位。

2、如果整数后面没有出现字母后缀，就就根据整型常量的值确定它的类型。例如[-32768,32767]之间的整数是short型，[32768,65535]之间的非负整数可以看成unsigned short类型，等等

3、const int为常量，不能修改

字符型的存储

在计算机内部以二进制ASCII码存储(8位二进制，**1字节**)

字符型变量的值可以是字符或整数（ch='A'或者ch=65），同时整型变量的值也可以是字符型数据，它可以被定义成字符型变量（前提是取值为有效的ASCII码）。因此**整型变量和字符型变量的值可以相互交换**。

`'\'`+字母为转义符

`'\'`+若干数字为**8进制字符**（每个数位都必须为0-7，否则从该位起不再是8进制字符而是普通数字）

常见转义符见书p128

实型的储存

	存储空间	有效数字	取值范围
`float`	4字节	7-8位	$\pm(10^{-38}\sim10^{38})$
`double`	8字节	15-16位	$\pm(10^{-308}\sim10^{308})$

语法与运行

数据类型转换

- **自动类型转换**

  c语言中不同类型的数据可以混合运算，但是在混合之前会自动转换类型，规则如图

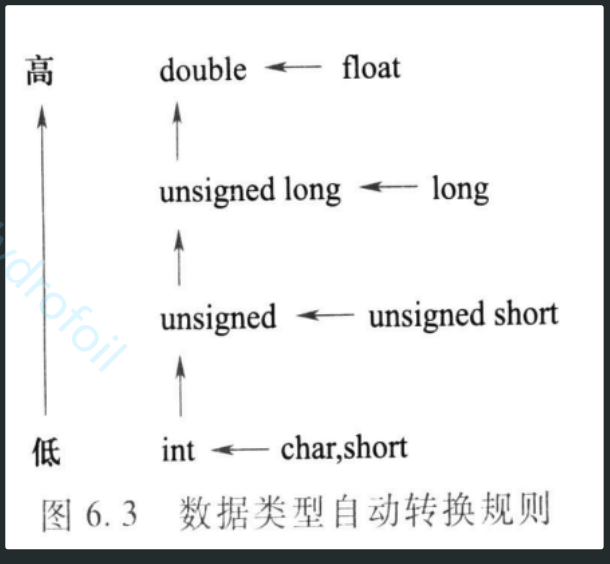


图 6.3　数据类型自动转换规则

  又如

```
short a;
char b;
long c;
double b;
c=a+b;
b=2.56;
```

运算时先把a和b都转换为int类型，再求和，最后将和转换为long类型

- **强制类型转换**

```
int a=7, b=3;
double c;
c=a/b;
printf("%lf ", c);
printf("%lf", (double)(a/b));
```

因为a/b是整数运算，结果会先被截断为整数，然后再赋给double型的c。

所以将a和b的定义改为double类型，然后取整输出c即可。

强制类型转换不改变数据定义。 c=(double)a/b之后a与b仍是int类型。

强制类型转换是运算符不是函数。 (int)x不能写成int(x)

(int)a+b等价于(int(a))+b

- **浮点数的表示**

小数表示法

```
float a=160.;
float b=.12;
float c=-.12;
float d=123; //没有小数点是不正确的浮点数表示，会被自动转换成int
```

指数表示法

```
float a=1e5; //e可大写
float b=1e-5;
float c=0e3; //等于0
float d=12e-0;
float e=e3; //不可以e开头
float d=3e; //不可以e结尾
```

表达式

表 6.8　部分运算符的优先级和结合性

运算符种类	运　算　符	结 合 方 向	优先级
逻辑运算符	!	从右向左(右结合)	高
算术运算符	++　--　+　-(单目)		
	*　/　%(双目)	从左向右(左结合)	
	+　-(双目)		
关系运算符	<　<=　>　>=		
	==　!=		
逻辑运算符	&&		
	||		
条件表达式	?:	从右向左(右结合)	
赋值运算符	=　+=　-=　*=　/=　%=		
逗号运算符	,	从左向右(左结合)	低

优先级	操作符	简释	用法示例	说明	同级结合律
1	( )	括号	(a + b)	-	-
	( )	函数调用	Fun(a , b)	取功能	左 --> 右
	[ ]	下标引用	arr[0]	取元素	
	.	访问结构体成员	structName.a		
	->	用指针访问结构体成员	pstructName->a		
2	++	后置++	num++	单目	左 --> 右
	--	后置--	num--		
	++	前置++	++num		右 --> 左
	--	前置--	--num		
	!	逻辑反(true<-->false)	!flag		
	~	按位取反(二进制位)	~a		
	+	表示正值	1		
	-	表示负值	-1		
	&	取地址	char*p=#		
	*	解引用	*p=10;		
	(type)	强制类型转换	q=(int*)p;	-	
	sizeof	求变量的长度,单位字节	len=sizeof(int);	亦可函数	
3	*	乘法	a*b	双目	左 --> 右
	/	除法	a/b		
	%	对整数取余数	a%b		
4	+	加法	a+b	双目	左 --> 右
	-	减法	a-b		
5	<<	左移位(二进制位)	a << 1	双目	左 --> 右
	>>	右移位(二进制位)	a >> 1		
6	>	大于	a > b	双目	左 --> 右
	>=	大于等于	a >= b		
	<	小于	a < b		
	<=	小于等于	a <= b		
7	==	等于	a == b	双目	左 --> 右
	!=	不等于	a != b		
8	&	按位与(二进制位)	a & b	双目	左 --> 右
9	^	按位异或(二进制位)	a ^ b	双目	左 --> 右
10	|	按位或 (二进制位)	a | b	双目	左 --> 右
11	&&	逻辑与	a && b	双目	左 --> 右
12	||	逻辑或	a || b	双目	左 --> 右
13	? :	条件操作符	a >0 ? a : b	三目	-
14	=	赋值	a = 2;	整式	右 --> 左
	+=	加等于	a += 2;		
	-=	减等于	a -= 2;		
	*=	乘等于	a *= 2;		
	/=	除等于	a /= 2;		
	%=	取余等	a %= 2;		
	<<=	左移等于	a <<= 1;		
	>>=	右移等于	a >>= 1;		
	&=	按位与等于	a &= b;		
	^=	按位异或等于	a ^= b;		
	|=	按位或等于	a |= b;		
15	,	逗号表达式	a = 0, a += 3	间隔符	左 --> 右

- **算术表达式**

  注意自增自减的运算对象只能是变量，不能是常量或表达式

  `3++` 和 `++(i+j)` 均为非法

- **赋值表达式**

运算过程：计算等号右侧表达式的值 -> 将右侧的值赋给左侧的变量 -> 将左侧变量的值作为表达式的值

可以复合运算：`x=(y=3)`等价于`x=y=3`

`x*=y-3`等价于`x=x*(y-3)`而非`x=x*y-3`

- **关系表达式**

例如`if(2<x<5)`语句逻辑上是错误的，因为c语言没有布尔型变量，只有1和0表示真和假。不管`x`为多少，`2<x`的值只能为1或0，然后比较`1<5`或`0<5`，发现都成立，最终返回1。

而如果是`if(-4<x<-2)`，那么最后只可能比较`0<-2`或`1<-2`，都返回0

- **逻辑表达式**

- **条件表达式**

`(n>0) ? 2.9 : 1`根据自动转换规则，若n为负数则返回1.0

- **逗号表达式**

逗号既可作分隔符，也可做运算符。

一般形式：`表达式1，表达式2，... ，表达式n`

运算过程：一次计算表达式1到n，然后将最后一个表达式n的值作为整个逗号表达式的值

```
a=3;
b=2;
 return (a, b);
// 等价于
 return (a=3, b=2);
```

逗号表达式常用于`for`循环中

后缀/前缀运算

后缀自增：先返回变量的值，然后将变量+1，**左结合**

前缀自增：先将变量+1，然后返回变量的值

```c
int a=2,b=2,d=0;
d=a+++b++;
printf("%d %d %d", d, a, b);
```

运行结果：`4 3 3`

```c
int a=2,b=2,d=0;
d=++a+++b;
printf("%d %d %d", d, a, b);
```

运行结果：报错，从左往右编译，先读到`++a`，之后又读到连续2个`+`，自动与左边的`++a`结合(等价于`(++a)++`)，而`a++`是不允许赋值的。

```c
int a=2,b=2,d=0;
d=(++a)+(++b);
printf("%d %d %d", d, a, b);
```

运行结果：`6 3 3`

在for循环中`i++`与`++i`对运行不造成影响

```c
int s=0, n=4;
while(n--)
{
    s += n;
}
printf("%d,%d\n", n, s);
```

运行结果：`-1,6`

位运算

运算符	名称
&	按位“与”
|	按位“或”
^	按位“异或”
~	取反
<<	左移
>>	右移

位运算的只能针对**整形、字符型及它们的变体**

- **位逻辑运算**

  除了~为单目运算符，其余均为双目

  特殊操作：交换变量a与b： `a^=b^=a^=b`

  从右往左计算，相当于`a^=b` -> `b^=a` -> `a^=b`

- **位移运算**

  位移运算不改变原操作数的值

  a>>b将a右移b位

  循环位移：移出的位在另一端补回

  逻辑位移：移出的位舍弃，移入的位取0

  算术位移：移出的位舍弃，左移入的位取0，右移入的位与符号位相同

  位移运算的具体方式取决于编译器。一般无符号位的操作数采用逻辑位移，有符号位的数采用算术位移

sizeof()运算

`sizeof()` 是 C 语言中的一个**运算符**，用于计算数据类型或变量的**字节大小**。它是编译时的运算符（也就是说，**`sizeof` 的结果在编译时就可以确定**）。

- **基本类型的大小**

```
printf("Size of int: %zu bytes\n", sizeof(int)); // 常见为 4 字节
printf("Size of char: %zu bytes\n", sizeof(char)); // 通常为 1 字节
printf("Size of double: %zu bytes\n", sizeof(double)); // 常见为 8 字节
```

- **变量的大小**

```
int x=10;
printf("Size of x: %zu bytes\n", sizeof(x)); // 输出与 sizeof(int) 相同
printf("Size of x: %zu bytes\n", sizeof(x++)); // 输出与 sizeof(int) 相同
```

- **数组的大小**

```
int arr[10];
printf("Size of arr: %zu bytes\n", sizeof(arr)); // 数组总大小：10 * sizeof(int)
printf("Size of one element: %zu bytes\n", sizeof(arr[0])); // 单个元素大小
printf("Number of elements: %zu\n", sizeof(arr) / sizeof(arr[0])); // 元素个数（这种写法很有用）
```

- **指针的大小**

```c
int *ptr;
printf("Size of pointer: %zu bytes\n", sizeof(ptr)); //指针大小与其数据类型无关，32位系统中为4字节，64位中为8字节
```

注意：在函数参数中，数组会退化为指针，所以：

```c
void foo(int arr[])
{
    printf("%zu\n", sizeof(arr)); // 输出指针大小，不是数组大小
}
```

其他运算

2. 特殊运算符

C 语言中，还有一些比较特殊的、具有专门用途的运算符。例如：

(1) ()括号：用来改变运算顺序。

(2) [ ]下标：用来表示数组元素，详见本书第 7 章。

(3) * 和 &：与指针运算有关，详见本书第 8 章。

(4) ->和.：用来表示结构分量，详见本书第 9 章。

数组

虽然c语言规定只有静态储存的数组才能初始化，但一般的c编译系统都允许对动态储存的数组赋初值。

```c
static int a[5]={1, 2, 3, 4, 5};
int b[5]={1, 2, 3, 4, 5};
```

静态储存数组没有赋初值则默认全0，动态储存数组初始值为随机数（垃圾值）

初始化可以只针对部分元素，如

```
static int a[5]={1, 2, 3}; //对前三个元素赋值
int b[5]={1, 2};
```

无论是静态还是动态，剩余元素初值为0

只有当对所有元素都初始化时才能省略数组长度

二维数组初始化：

- 分行赋初值：

```
int a[4][3]={{1, 2, 3}, {}, {4, 5}}; //只对第0行的全部元素和对第2行的前两个元素赋初值
```

- 顺序赋初值：

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

static int b[3][3]={1, 2, 3, 0, 0, 0, 4, 5};
static int a[4][3]={{1, 2, 3}, {}, {4, 5}};
//以上两行等价
```

若在分行赋初值时**出现了所有行**，或者顺序赋初值时**所有元素均赋值**，则可以省略行数（**只能省略行数**，不能省略列数，若n维数组则也只能省略第一个下标）

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

注意易错点：

```
int a[1][2]={{1}, {3}}; //报错，1行的数组，即为a[2]，但是赋值了两行
```

字符串

```
char x[]="abcdefg";
char x1[]={"abcdefg"};
// 上述两行等价

char y[]={'a', 'b', 'c', 'd', 'e', 'f', 'g'};

char z[]={'a', 'b', 'c', 'd', 'e', 'f', 'g', '\0'};
char z1[]={'a', 'b', 'c', 'd', 'e', 'f', 'g', 0};
// 上述两行等价
```

字符数组x属于字符串，长度为8（结束符\0）；字符数组y的长度为7，不属于字符串；字符数组z的长度为8，属于字符串。

z数组和z1数组等价。**因为'\0'就是ASCII码为0的字符**。

可以在**定义语句**中通过赋值运算符"="对字符数组整体赋值，但**不可以**在**赋值语句**中通过赋值运算符"="对字符数组整体赋值

- **求字符串长度strlen()**

  但是strlen(x)=strlen(z)=7,strlen(y)不确定。因为x还有结束符'\0'，因为strlen()函数是一直读内容直到遇到'\0'（**'\0'不计入长度**），但是y中没有'\0'，所以函数就会读取y后面的内存内容，直到遇到一个随机的'\0'

  注意特殊字符！

  ```
  char str1[]="zju\101\\red";
  char str2[]="zju\109\red";
  char str3[]="123\029\08";
  ```

  以'\'及其后紧跟的**至多3位数字为8进制字符**，'\101'是一个**8进制字符**。'\\'双反斜杠取消转义，记为一个反斜杠。因此str1共3+1+1+3=8个字符(8个字节)

  '9'不是8进制字符，因此\10'是一个字符。'\'后紧邻的一个字符会被识别为一个转义符(不管是否有定义)，因此'\r'是一个字符。因此str2共3+1+1+1+2=8个字符(8个字节)

  关于'\'开头的8进制字符：

以'\'及其后紧跟的**至多3位连续**的属于0~7的数字为**8进制字符**。若不足3位则默认在最高位补0

8进制字符即为转为10进制后ASCII码对应的字符。

例如：`101(8)`=`65(10)`，所以`'\101'`=`'A'`；

`'\10'`=`'\010'`=`'8'`=`'\b'`；

`\66`=`'54'`=`'6'`

若`sizeof()`则要+1（计入`'\0'`），如果是**问字符个数，也要+1**，因为它等同于要开辟多大的数组才能容纳这个字符串

`strlen(str3)`为5，因为函数读到第一个`'\0'`就认为字符串结束了

但是，若在定义字符串时已经定义了长度，则`sizeof()`就为该长度

```
1 char str[80]="abcdefg"; //sizeof(str)=80
```

- **字符串比较`strcpm()`**

  `strcmp(const char *str1, const char *str2)`函数：两个**字符串（均以'\0'结尾）**自左向右逐个字符相比（按ASCII值大小相比较），直到出现不同的字符或遇'\0'为止。**返回值视编译器而定。**

  PTA上：若`str1`=`str2`则返回0，否则返回ASCII之差（前者减后者）。**注意两个参数（字符串名称）实际上是指针。**`strcmp()`属于`string.h`库

  VSCode上：若`str1`=`str2`则返回0，若前者>后者则输出1，若前者<后者则输出-1

  不能直接使用关系运算符比较字符串的大小，因为这些运算符比较的是两个字符串**首地址的值**，而不是字符串的内容。

- 字符数组中的字符串可以**整体输入输出：**`gets()`与`puts()`都**属于`stdio.h`库**

gets(str)函数 会读取整行输入行，直至遇到'\n'，然后**丢弃换行符**，储存其余字符，并**在字符末尾添加一个'\0'**使其成为C字符串。但是若读入的那一行超过了str本身定义的长度，**多余的字符会占用str之后的内存空间，可能导致修改了程序中的其他数据**。

puts(str)函数 用于输出一个**字符串**（'\0'结尾）并**换行**，其中括号内的参数是输出字符串的起始地址

注意：字符串整体输出时**其中的转义符会影响输出结果**。

例如：

```
char str1[]="zju\red"; //'\r'是回车符，相当于把光标移到行首，但不会删除前面的字符
//光标移到行首之后继续输出，就会将原先位置的字符覆盖
```

输出：

```
edu
```

例如：

```
char str1[]="zju\10ed" //'\10'即为'\8'即为'\b'，退格符，会把'u'给删掉
```

输出：

```
zjed
```

- **字符串复制strcpy()**

```
strcpy(str1,str2);
strcpy(str1, "form");
```

将字符串str2复制到str1，**直到遇到'\0'时停止复制，但是'\0'会被复制到str1中**

str1必须为字符型数组的基地址，str2可以是字符数组名或字符串常量

str1要有足够空间容纳str2，否则编译不会报错，用puts()可以输出（指针一直后移直到遇到'\0'），但用for逐位输出就会卡死(下标越界)。

若str1长于str2，则str1中被str2覆盖后**剩余的字符会被保留**

```
char str1={'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k'};
char str2="happy";
strcpy(str1, str2);
```

运行后：

```
str1={'h', 'a', 'p', 'p', 'y', '\0', 'g', 'h', 'i', 'j', 'k'};
```

但是若执行puts(str1)则只会输出happy，因为puts()遇到'\0'就停止输出。

- **字符串连接strcat()**

```
strcat(str1,str2);
strcat(str1, "form");
```

将str2接到str1的后面，**str1中原有的'\0'会被放在拼接后的结束位置上**。
同理，str1必须为字符型数组的基地址，str2可以是字符数组名或字符串常量，str1要有足够空间容纳str2

- **字符查找strchr()**

```
char *p = strchr(str1, 'a');
printf("%d", p-str1);
```

返回'a'在str1中第一次出现的指针地址，没找到则返回NULL

- **子串查找strstr()**

```
char *p = strstr(str1, str2);
char *p = strstr(str1, "form");
```

返回str2在str1中第一次出现的指针地址，没找到则返回NULL

str2可以是字符数组名或字符串常量

if语句

- **悬空else**

悬空else的意思是在**没有括弧的多层if语句**后面突然跟一个else，他总会跟与他最近的，并且**没有跟别的else匹配过的** if相匹配，与缩进无关

```
#include<stdio.h>
int main()
{
    int a = 1, b = 2;
    if (a == 2)
        if (b == 2)
            printf("1");
    else
        printf("2");
    return 0;
}
```

运行结果是没有任何输出。因为第一个if (a==2)不成立，而else和最近的if (b==2)匹配，故从第一个if直接跳到return 0;

但是如果第一个if (a == 2)有括弧：

```
#include<stdio.h>
int main()
{
    int a = 1, b = 2;
    if (a == 2)
    {
        if (b == 2)
            printf("1");
    }
    else
        printf("2");
    return 0;
}
```

非悬空，else直接和if (a == 2)匹配

- **if()中赋值语句（单个等号、逗号表达式）**
- **if()中的连续比较**

```
if(a<=x<=b) //从左往右，先比较a<=x，值为0或1，再比较0<=b或1<=b
```

- **短路运算**

```
if(a && b) //从左往右，先判定a，若a不成立则b就不看了
```

```
int a=1, b=2, c=9;
if((a=0)&&(b=4)) //(a=0)的值为0，已经不满足条件，(b=4)就不执行了
{
    c++;
}
printf("%d %d %d\n", a, b, c);

int a=1, b=2, c=9;
if((a=0)||(b=4)) //(a=0)的值为0，还需继续判断(b=4)
{
    c++;
}
printf("%d %d %d\n", a, b, c);
```

输出：

```
0 2 9
0 4 10
```

- **if( );中的分号**

```
int x=-1;
if(x==0);
{
    printf("%d", x);
}
```

输出结果：0

千万留意括号后面有没有分号！！！ 加了分号就表示if语句结束了，之后那一行是否执行就跟if无关了。

switch语句

```
switch (expression)
{
    case constant1:
// 当 expression 的值等于 constant1 时执行的代码
        break;
    case constant2:
// 当 expression 的值等于 constant2 时执行的代码
        break;
    default:
// 当 expression 的值不等于任何一个常量时执行的代码
        break;
}
```

1、从上往下比对每一个`constant`与`expression`是否一致，当`expression`的值与某个`constant`相等时，执行对应的代码块。若该代码块没有`break`，则依次执行之后的`case`代码块（即使下一个分支的条件并不匹配），直到遇到`break`（即**"贯穿现象"**）。

2、`case`后的常量值必须是**整数或字符常量（不能是字符串）**。

3、`default`代码块**只会**在所有`case`都不满足的时候被执行，此时若没有`default`则什么都不会被执行。

4、因为贯穿现象，`default`代码块的位置有时会影响程序的运行结果。

```
int k=1;
switch(k)
{
    case 2: printf("2");
    default: printf("default");
    case 3: printf("3");
}
```

输出：

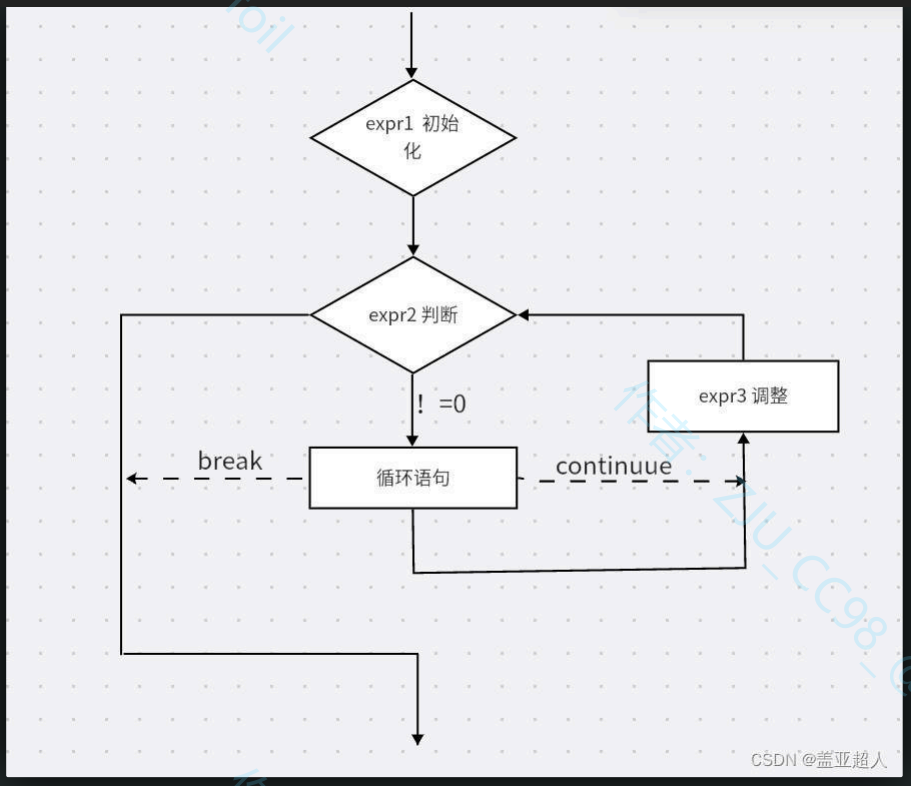
```
default3
```

4、**有意利用贯穿现象的情况**：在某些情况下，可以利用贯穿现象来简化代码。例如：

```c
switch (month)
{
    case 12:
    case 1:
    case 2:
        printf("这是冬季\n");
    break;
    case 3:
    case 4:
    case 5:
        printf("这是春季\n");
        break;
    case 6:
    case 7:
    case 8:
        printf("这是夏季\n");
        break;
    case 9:
    case 10:
    case 11:
        printf("这是秋季\n");
        break;
    default:
        printf("月份无效\n");
        break;
}
```

for循环

for(expr1;expr2;expr3) 注意下图，**三个expr不是一次性执行的**




```c
for (i=1;i<=10;++i)
{
    printf("%d", i);
    i=-1;
}
```

输出结果：100000000000……

分析：第一遍循环，执行expr1使得i=1，输出1，然后i赋为-1。第二遍循环，**先执行++i使得i=0，然后判断满足i<=10（即使i不在1到10的范围内）**，输出0，然后i又被赋为-1。之后同第二遍，无限循环。

特别注意i在循环体内有没有被再次修改！！！

for中的3个expr均可省略

注意for循环后**有没有分号**

```c
for(int i=1; i<=n; i++);
    printf("%d", i);
```

输出n+1

while循环

注意没有括弧时，循环体只能是一条语句

题目问while结束时i的值为多少，记得看看i每次加多少，甚至有没有i+=几这条语句

别落下不满足while条件的最后一次循环

while()括弧中的表达式值为0或'\0'时停止循环，因此while(*p)，每次p++，直到遇到'\0'停止

注意while循环后**有没有分号**

静态变量与动态变量

- **作用域**

  int：

  局部变量：作用域仅限于声明它的函数或代码块

  全局变量：作用域是整个文件

  static int：

  局部变量：作用域是**声明它的函数或代码块**

  全局变量：作用域是**声明它的文件**

- **存储周期**

  int：

  局部变量：从声明开始，到所在块结束。

  全局变量：整个程序运行期间。

  static int：

  无论是局部还是全局，生命周期都是程序运行期间。

- **初始值**

  int：

  局部变量：未初始化时值是未定义的（垃圾值）

  全局变量：未初始化时值默认是0。

  static int：

  未初始化时，默认值为0。

- **储存区域**

静态变量：内存的静态储存区

动态变量：内存的动态储存区（堆栈）

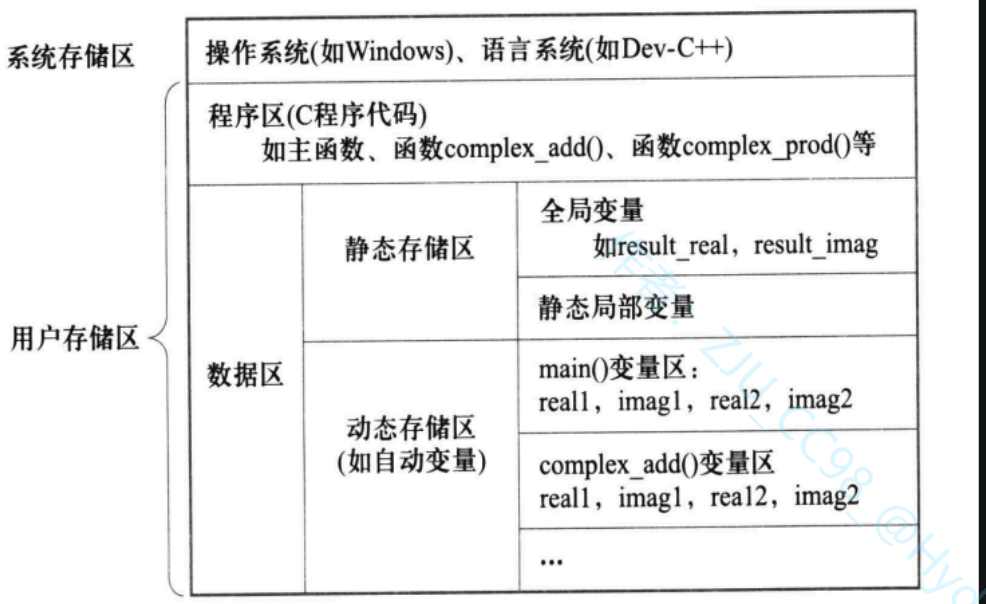


图 5.3　执行例 5-7 时的存储分布示意图

```c
#include <stdio.h>
int global_var = 0;// 全局变量
static int static_global_var = 0; // 静态全局变量
void func()
{
    int local_var = 0; // 局部变量
    static int static_local_var = 0; // 静态局部变量

    local_var++;
    static_local_var++;

    printf("local_var: %d, static_local_var: %d\n", local_var,
static_local_var);
}
int main()
{
    for(int i = 0; i < 3; i++)
    func();
}
```

输出

```
local_var: 1, static_local_var: 1
local_var: 1, static_local_var: 2
local_var: 1, static_local_var: 3
```

局部变量与全局变量

- **局部变量动态分配内存**：

  计算机从`main()`开始运行，因此`main()`中的变量一开始就被分配了内存。

  其他函数的局部变量**在调用前未被分配内存（静态变量除外）**，只有函数被调用时才会被分配到内存；一旦函数结束调用，局部变量的内存单元**被收回**，形参和局部变量不复存在。以上称为局部变量生存周期，采用**动态分配内存单元**的方式。

  因此也把**局部变量**称为**自动变量**，定义形式为

```
auto 类型名 变量表;
auto int a, i, j;
```

  一般定义局部变量时`auto`都可以省略

- 全局变量会被局部变量**覆盖**

```
int x = 5, y = 7;
void swap()
{
    int z ;
    z = x ;  x = y ;  y = z ;
}
int main()
{
    int x = 3, y = 8;
    swap();
    printf("%d,%d \n", x , y );
}
```

  输出：

```
3 8
```

- **`extern`关键字：**

  用来声明一个**全局变量**或**函数**（定义在其他文件中），而不是重新定义它。

```c
// file1.c
#include <stdio.h>
int sharedVar = 10;
void incrementSharedVar()
{
    sharedVar++;
}
```

```c
// file2.c
#include <stdio.h>
extern int sharedVar; // 声明全局变量
extern void incrementSharedVar(); // 声明外部函数
int main()
{
    printf("sharedVar = %d\n", sharedVar);
    incrementSharedVar();
    printf("sharedVar = %d\n", sharedVar);
}
```

  输出：

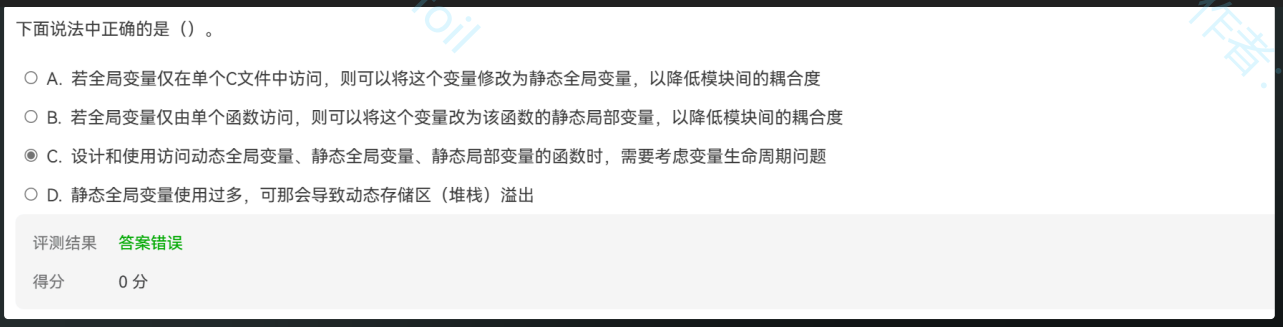
```
sharedVar = 10
sharedVar = 11
```

- **静态函数**

  为了避免各文件模块间相互干扰，c语言允许将函数定义为静态的，以便将函数的作用范围局限在文件模块中，即使别的文件用了`extern`来声明也不会调用

```
static 函数类型 函数名（参数表说明）；
```

  这样即使其他文件模块中有同名函数，也不会相互影响，增加了模块间的独立性。



答案A。B不会降低耦合度。C是它们的声明周期都是从程序运行开始到结束，所以不需要考虑。D是它们存储在**静态存储区**。

> **模块耦合度**是衡量软件系统中模块之间依赖程度的一个重要指标。模块耦合度**低**通常被认为是**良好**的设计特性，因为这意味着模块更加独立，修改某个模块时对其他模块的影响较小，从而提高了系统的可维护性和扩展性。
>
> 降低模块耦合度的方法通常有避免使用全局变量、使用接口和抽象等。

函数

- **形参与实参：**

  实参可以是任意类型的变量、常量与表达式

  形参只能是变量

函数不加最后一行没有return时，导致未定义行为（Undefined Behavior），返回值因编译器而异，也可能不返回

```
int f(int k, int j)
{
    k++;
// return k;
}
int main()
{
    int a=2, b=3;
    printf("%d", f(a,b));
}
```

输出：

```
2
```

函数分为`int`，`float`，`double`，`char`，`void`等类型，**若未显式声明函数类型，则默认为`int`类型**。

函数的声明与函数的定义：

- **函数的声明**只告诉编译器函数的名称、返回类型和参数信息，但不提供函数的具体实现，通常在头文件（`.h`或`.hpp`）中进行函数的声明

```
int f(int a, int b);
```

- **函数的定义**提供了函数的名称、返回类型、参数信息以及函数的具体实现，通常在源文件（`.c`或`.cpp`）中进行函数的定义

```
int f(int a, int b)
{
    return a+b;
}
```

举例：

```
void main()
{
    ...;
    swap(&a, &b);  // 错误：调用main()之前swap()未声明
}
void swap(int *x, int *y)
{
    ...;
}
```

`swap` 函数的定义在主函数`main`之后，导致编译器在看到 `swap(&a, &b)` 时不知道 `swap` 的具体类型，造成编译报错。

编译器默认 `swap` 是一个返回 `int` 的函数，这便是**隐式声明**。

所以需要在主函数之前加上`swap`函数的声明。

指针

指针的定义

```
int *p; //'*'为指针声明符
```

指针的赋值

```
p = &a; //'&'是取地址运算符，用于指针的赋值，&a的值就是a的内存地址。

p = 0;
p = NULL; //第二三行等价，表示p是一个空指针。NULL在stdio.h中被定义，其值为0。

p = (int * )1732; //用强制类型转换(int*)来避免编译错误。表示p指向内存地址为1732的变量。
```

指针的运算

```
p = &a; //取地址运算
*p = 3; //间接访问地址运算。'*'用于访问指针所指向的地址，并得到该地址储存的变量a。然后用赋值语句将其赋为3。
// 以上两行可以合并为 *(&a)=3

p2 = p; //使指针p2等于指针p
p2 == p; //比较指针p2是否等于指针p，返回1或0
p2 > p; //两个同类型的指针间可以比较大小，按照实际内存的高低位比较

x = *p++; //等价于x = *(p++)（指针后移）
x = (*p)++; //所指元素后缀自增
x = ++*p; //等价于x=++(*p);（所指元素前缀自增）

*p++; //等价于*(p++)

b = p2-p1; //两个相同类型指针变量相减，可以获得在之间相隔的同类型元素个数（p2<p1也可以是负数）
```

但是!

```
c = p1+p2; //是不可以的，因为两个指针相加什么都得不到，所以规定不允许相加，编译会报错。
printf("%d %d", p1, p2); //输出的两数（内存地址）之差为(间隔的元素个数)*(单个元素的内存字节数)
```

指针的初始化

```
int a;
int *p1 = &a;
int *p2 = p1;
int *p = 0; //相当于int *p = NULL
int *p = 1000; //这是非法的，不能将除0以外的数值作为指针变量的初值
```

指针作为函数的参数

```
void swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
// 通过指针访问来修改主函数中a和b的值（数组也一样），这样就可以不用定义全局变量，也避免了return只能返回一个值的麻烦。
int main()
{
    int a=1,b=2;
    int *pa=&a, *pb=&b;
    swap(pa, pb); //以上两行也可以简写作swap(&a, &b);
}
```

指针、数组和地址间的关系

数组的基地址是其第一个元素`a[0]`的地址，因此数组名本身就是一个地址即指针值。在内存地址方面指针和数组名唯一的区别在于数组名一经定义，指向的地址便不能修改；而指针指向的内存地址是可以修改的。

```
int a[100], *p;
p = a;
p = &a[0]; //以上两行是等价的。
p = a+1;
p++;
p = &a[1]; //以上三行也是等价的。都把p指向a[1]的地址
p += i; //将p指向a数组当前往后第i个元素的地址
a += i; //编译错误，因为数组名虽然是指针，但是不能修改
```

正因为**数组名本身就是指向第一个元素地址的指针值**，所以`*a`就是访问`a[0]`的运算。

```
printf("%c", a); //输出无结果
printf("%c", *a); //输出a[0]
```

- **关于`a[-n]`：**

在C语言中，指针支持**负偏移**，即可以指向数组范围之外的地址（但如果该地址未定义，则尝试解引用它时会导致未定义行为）。

```
char *p = a[-1];
```

则`p`指向`a[0]`前一个内存单位。因为没有实际访问该内存中的储存的值，所以不会触发未定义报错。

```
static int a[3][4]={{1,2,3,4},{5,6}};
```

则`a[1][-1]=a[0][3]=4`

数组名作为函数的参数

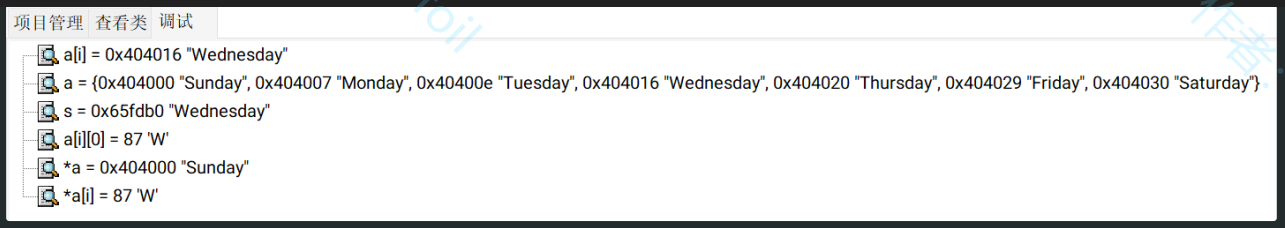
```c
int main()
{
    int a[10], b;
    b=sum(a);
}
int sum(int a[]) //int a[] 等价于int *a，因为数组名和指针都指向基地址。
```

指针数组

数组的每一个元素都是一个指针

```c
int getindex(char *s)
{
    char *a[7]={"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};
    //*a[7]定义a[7]是一个指向char型的指针数组
    //每一个元素a[i]是一个指针（此处a[i]是一个char型数组，也属于指针，因为数组名与指针在内存地址上等价）。
    //a[i]指向每个数组的第一个元素，即储存这个数组的起始地址，所以*a[0]='S', *a[1]='M'
    int i;
    for(i=0;i<=6;i++)
    {
        if(strcmp(s,a[i])==0) return i;
        // 不能写if(s==a[i])因为这样比较的是数组的起始地址，而非数组的内容。
        // 不能写if(*s==*a[i])因为这样比较的是字符串的首字母
        // 注意strcmp的两个参数是指针（或数组名）
    }
    return -1;
}
```

在Dev的调试界面里可以直观看出区别：



其中`0x65fdb0`和`0x404000`是字符串基地址，87是`'W'`的ASCII码

```c
#include<stdio.h>
int main(void)
{
    int i, a[12] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 },
*p[4];
    for (i = 0; i < 4; i++)
    {
        [i] = &a[i*3];
    }
    printf("%d\n", p[3][2]);
}
```

`p[3]`指向`a[9]`，代表以`a[9]`开头的数组，即`{10,11,12}`，因此`p[3][2]`即为12

二级指针

```c
#include<stdio.h>
int main(void)
{
int x[5] = {2, 4, 6, 8, 10}, *p, **pp;
p = x;
pp = &p;
printf("%d ", *(p++));
printf("%d\n", **pp);
}
```

输出：

```
2 4
```

`pp`指向`p`，**而且`p`移动以后，`pp`中储存的`p`地址也会随之改变**，`pp`依旧指向新的`p`

二维数组的数组名就是一个二级指针，a[0]，a[1]，...，各自是一个一级指针

```c
#include<stdio.h>
int main()
{
    char p[3][4]={"ABC","DEF","XYZ"};
    char *q[3];
    q[0]=p[0];
    q[1]=p[1];
    q[2]=p[2];
    *(*(q+1)-1)=*p[2];
    puts(*p);
}
```

输出：

```
ABCXDEF
```

q指向p[0]，q+1表示指针后移，*(q+1)此时就是q[1]，指向'D'；

*(q+1)-1这里，优先级*比-高；所以q[1]-1表示'D'往前退一位，从'D'指向了'\0'(第一行的结尾，**注意不是'c'**)，**因为在内存中是连续储存的**；

A	B	C	\0	D	E	F	\0	X	Y	Z	\0

p[2]相当于是一级指针，*p[2]取出它实际的内容，则为'X'；然后将原来的'\0'赋为'X'；

最后输出*p，因为输出的结束标识符是'\0'，第1行没有了该字符，一路往下找所以就是第2行的，最终输出ABCXDEF

int *p[3] 和 int (*p)[3] 的区别：

- int *p[3]：

  p 是一个数组，这个数组有 3 个元素，每个元素是一个指向 int 类型的指针。

- int (*p)[3]：

  p 是一个指针，这个指针指向一个长度为 3 的数组，而这个数组的元素是 int 类型。

函数指针

函数指针定义的一般格式为`类型名(*函数指针名)(参数类型表)`

调用函数可以直接通过函数名，也可以通过函数指针

通过函数指针调用的一般格式为`(*函数指针名)(参数表)`

```c
#include<stdio.h>
double f1(double m, double n);
double f2(double m, double n);
double calc(double(*funp)(double, double), double a, double b);
int main()
{
    double(*funp)(double, double);
    double result1, result2, tmp, x, y;

    funp=f2; //对函数指针funp赋值
    tmp = (*funp)(x, y);

    result1 = calc(f1, x, y); //函数名f1直接作为函数calc的实参之一
    result2 = calc(funp, x, y); //函数指针funp作为calc的实参之一
}
```

动态内存分配

- `void *malloc(unsigned size)`

  在内存的动态储存区中分配长度为`size`的连续空间

  若成功则返回该指向该内存空间首地址的**指针**，否则返回`NULL`**(值为0)**

  `malloc()`的返回类型为`void *`，**代表通用指针**，使用时需要用**强制类型**转换将其**转换为具体类型的指针**（见链表）

- `'void *calloc(unsigned n, unsigned size)'`

  在内存的动态储存区中分配`n`**连续**的内存空间，每个长度为`size`，并且**内存中储存的内容初始化为0**

  若成功则返回该指向该内存空间首地址的**指针**，否则返回`NULL`**(值为0)**

同样的，使用时**也需要强制类型转换**

- `void *free(void *ptr)`

  释放由动态内存分配函数申请到的内存。`ptr`为指向该内存的首地址，若为空指针则函数什么都不做

  释放后不允许再用`ptr`访问已经该释放的块。所以释放后应**第一时间将`ptr`赋为`NULL`或指向别的已定义内存**。

- `void * realloc(void *ptr, unsigned size)`

  更改以前的内存分配。`size`为现在所需的内存大小。`ptr`**必须为以前通过内存分配所得到的指针**

  若成功，则返回一片大小为`size`的内存，并保证**新内存块中的内容与原块的一致**。如果`size`大于原块的大小，则原有数据存在新块的**前一部分**；如果`size`小于原块的大小，新块储存的是原块**前`size`范围内的数据**。

  若失败，则返回`NULL`，同时原来内存块的内容不变

  同样的，**释放后不允许再用`ptr`访问已经该释放的块**，应及时对`ptr`赋新的有效值

命令行参数

c语言源程序经过**编译**和**连接**处理，生成可执行程序（又称可执行文件或**命令**），它可以直接在操作系统环境中以命令的方式运行。

输入命令时可以在后面跟一些参数，称为命令行参数，格式为(以`test.exe`为例)：

```
命令名 参数1 ... 参数n
test hello world
```

若程序中**没有接收命令行参数的语句**，则命令后有无参数**不会影响**程序的运行结果。

c程序中，`main()`函数可以有两个参数，用于接收命令行参数，习惯上书写为：

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

其中argc接收命令行参数（**包括命令**）的个数，argv接收以**字符串常量**形式存放的命令行参数，argv[0]指向命令，从argv[1]开始指向命令行参数

main()中的形参可是是**任意合法标识符**，但一般习惯用argc与argv

系统会根据输入的命令行参数的数量与长度，**分配储存空间**存放这些参数（包括命令），并将参数（包括命令）的**数量与首地址**传递给形参argc和argv

2-11 假设下列程序保存在test.c中，编译后运行test hello world，则输出是

```
#include<stdio.h>
int main(int argc, char *argv[ ])
{
      printf("%d,%s", argc, argv[1]+1);

      return 0;
}
```

A. 2,est

B. 2,ello

C. 3,ello

D. 3,orld

参考答案

答案 C

结构体

结构的定义以分号结束，因为c语言将结构的定义看做一条语句。

嵌套定义结构类型时，必须先定义成员结构类型，再定义主结构类型。

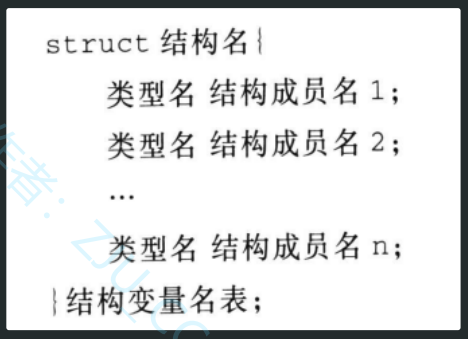
结构的三种定义类型：

- **单独定义**

```
struct student //struct和student必须连用，因为它们合起来表示一个数据类型名
{
    int index;
    char name[100];
}; //结构变量表在主程序中定义。
```

- **混合定义**

在定义结构类型的同时定义结构变量




```
struct student
{
    int index;
    char name[100];
    int age;
}s1, s2;
```

- **无类型名定义(匿名结构体)**

```
struct
{
    int index;
    char name[100];
    int age;
}s1, s2;
```

由于省略了结构名，后续语句无法再定义这个结构类型的其他变量。所以除非变量不再增加，不会使用此法。

结构变量所占的内存空间是各个成员所占的内存空间之和，可用`sizeof`运算来求得。`sizeof`运算对象可以是结构类型名也可以是结构变量名，计算结果以字节为单位。

结构类型变量的在内存中的储存布局按其成员的先后顺序排列。

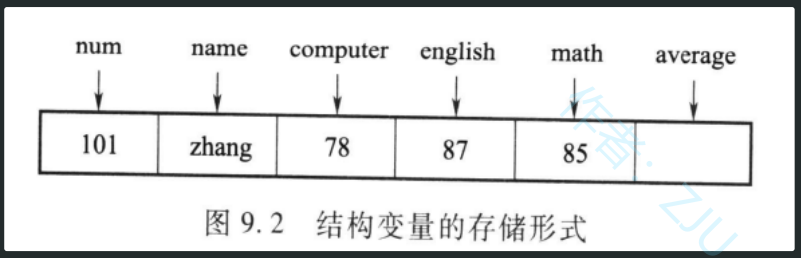


图 9.2　结构变量的存储形式

结构的整体赋值：若s1与s2类型相同，则可以整体赋值`s2 = s1`，这是结构中**唯一的整体操作方式**。

结构的初始化：

```
struct student s1 = {101, "zhang", 15};
```

结构数组初始化：与二维数组相似。

```
struct st
{
    int n;
    struct st *next;
} a[3] = {5, &a[1], 7, &a[2], 9, NULL}, *p = &a;
  //等价于a[3] = {{5, &a[1]}, {7, &a[2]}, {9, NULL}}
```

结构变量可以作为函数的参数。

- **结构指针**

  ```
  struct student s1 = {101, "zhang", 15},  *p;
  ```

  结构指针实际上是结构变量的首地址，即第一个成员的地址。

用结构指针访问成员变量的两种方法：

```
(*p).age=15;
p->age=15 //'->'称为指向运算符
```

结构指针也可以作为函数参数或返回值，例如：

```
struct stud_node *InsertDoc(struct stud_node *head, int num);
```

由于在函数中定义的结构变量不能在主程序中被调用，所以如果要用函数创建结构，并在主程序中仍能使用，就要借助结构指针。

步骤：首先得到该结构类型的`size` -> 在函数中申请`size`大小的动态内存空间 -> 将其强制类型转化成该结构类型的指针 -> 最后用结构指针指向它。

之后都通过该指针对其进行操作。

```
#include <stdio.h>
#include <stdlib.h>
struct ListNode
{
    int data;
    struct ListNode *next;
};
  struct ListNode *createlist();
int main()
  {
    struct ListNode *p, *head = NULL;
    head = createlist();
    for (p=head; p!=NULL; p=p->next)
            printf("%d ", p->data);
    printf("\n");
    return 0;
}
struct ListNode *createlist()
  {
    int n=0, size=sizeof(struct ListNode);
    struct ListNode *p, *head, *tail;
        head=tail=NULL;
    scanf("%d", &n);
    while(n!=-1)
    {
        p=(struct ListNode *)malloc(size); //申请一个新的内存，转换成结构体指针类型，用p指向它
```

```
        //对p初始化
            p->data=n;
            p->next=NULL;
            if(head==NULL)
                head=p;
            else
                tail->next=p;
            tail=p;
            scanf("%d", &n);
        }
        return head;
}
```

链表的删除：

```
ptr1->next = ptr2->next;
  free(ptr2);
```

宏定义

`#define`用来定义一些符号常量

以#开头表示在**编译预处理中（即在对源程序其他语句正式编译之前进行）**起作用，**而不是真正的c语句，因此行尾无分号**。一般为了区别，使用大写字符串作宏名。

宏的作用范围从定义开始，到源文件结束或被取消定义（`#undef`）为止。(所以即使宏定义在函数内部，它仍然可以在函数外被使用)

宏替换不占用运行时间，只占用编译时间

宏名与宏定义字符串间用空格分隔，因此**宏名中不能有空格，但含参数的宏定义除外**，宏定义字符串可以是任意字符串（可以包含空格），因为宏只是实现**文本替代**的功能。所以**不用定义宏定义中参数的类型**

编译时，所有出现宏名的地方都会用宏定义字符串来替代

宏定义用途：

- **符号常量**

```
#define TRUE 1
#define FALSE 0; //若有';'，则FALSE就会被替代额为"0;"
#define PI 3.1415926
#define PI^2 PI*PI //宏定义可以嵌套定义
```

- **书写方便**

```
#define STRING "It represents a long string \
that is used as an example"
// '\'表示该行未结束，与下一行合并为完整一行。
//STRING代表带引号的字符串，因此在printf函数等中就不必再加引号
```

- **直接替代c语句**

```
#define F for(int i=1;i<=n;i++)
```

- **简单函数（含参数的宏定义）**

```
#define MAX(a, b) a>b ? a:b //宏只能限制在一行中，因此只能实现简单的功能
#define SQR(x) x*x
int main()
{
    int x, y;
    x = MAX(x, y);
}
```

特别注意宏定义的嵌套情况：考试时老老实实把展开式写出来

```
#define f(a,b,x) a*x+b
printf("%d\n", f(f(1,2,3),4,2));
// 替换结果为 1*3+2*2+4 = 11
```

```c
#define N 2
#define M N+1
#define NUM (M+1)*M/2
int i;
for(i = 1; i <= NUM; i++)
{
    printf("%d",i);
}
// NUM的替换结果为 (2+1+1)*2+1/2 = 8
```

注意：**当宏名中出现括号时，预处理器会将其识别为带参数的宏定义**，例如：

```c
#define value() 5 //编译错误：需要参数
#define val()ue  //编译错误：ue未定义（"ue"不是任何c保留字，因此被认为是变量，但未在括号中声明，故报错）
#define val(ue)ue //编译通过，等价于#define val(ue) ue（即空格可省略）
#define val(ue 5 //编译错误：括号不匹配
#define value(x) 42 //编译通过，但使用时括号中一定要有一个变量
```

含参宏定义编译预处理时（以`MAX(a, b)`为例），先用x和y替换a和b，于是`a>b ? a:b`变成`x>y ? x:y`，然后再把`x = MAX(x, y);`替换为`x = x>y ? x:y`，最后编译的是`x = x>y ? x:y`这条语句

所以，若执行`ans = SQR(x+y)`，最终编译的会是`ans = x+y*x+y`因为宏只是进行替换

所以应把宏定义改为：

```c
#define SQR(x) (x)*(x)
```

最终编译的才会是`ans = (x+y)*(x+y)`**对宏定义中的变量增加括号可提高运算优先级**，避免替换带来的副作用。

含参宏定义与函数的区别：

含参宏定义是**先替换后编译**；函数是先进行参数传递，然后主函数停止执行，转而执行函数，通过return()返回结果后再执行主函数。**宏定义不占用运行时间，只占用编译时间；函数两者都占用**

二义性问题：

括号最多的即为没有二义性的：

```
#define POWER(x) ((x)*(x))
```

下列选项中不会引起二义性的宏定义是(　)。

- A. `#define POWER(x) x*x`
- B. `#define POWER(x) (x)*(x)`
- C. `#define POWER(x) (x*x)`
- D. `#define POWER(x) ((x)*(x))`

条件编译

一般的程序经过编译后，所有的C语句都生成到目标程序中，如果只想把源程序中一部分语句生成目标代码，可以使用条件编译。它广泛运用于商业软件，可以为一个程序提供多个版本，不同的用户使用不同的版本，运行不同的程序功能。

```
#define FLAG 1
#if FLAG
    程序段1
#else
 程序段2
#endif
```

程序段1和程序段2只有一个会被生成到目标程序中，由于FLAG被定义成1，编译预处理选择程序段1编译；若FLAG改为0的话，编译预处理选择程序段2编译。条件编译起作用的时刻在**编译预处理**的时候。一旦经过处理后，只有一段程序生成到目标程序中，**另一段被舍弃**。

#的条件**只能是宏名**，不能是程序表达式，因为在编译预处理时是无法计算表达式的，必须在程序运行时才做计算。

文件

文件类型

- c语言中按数据储存的编码形式，可以分为**文本文件（ASCII码储存）**与**二进制文件（二进制储存）**。
- 从逻辑结构上讲，c语言将文件看做数据流，并将数据按一维方式组织储存。数据流分为**字符流**与**二进制流**。
- c语言源程序是文本文件。**目标文件**（通常以 .o 为扩展名，是在编译过程中由编译器生成的中间文件）和**可执行文件**二进制文件。

缓冲文件系统

- 进行文件操作时，系统自动为每一个文件分配一块文件内存缓冲区**（内存单元）**。C程序对文件的操作就通过对文件缓冲区的操作来完成。缓冲文件系统提高文件操作速度。**且c程序处理文件时不用考虑外部磁盘的物理特性**。

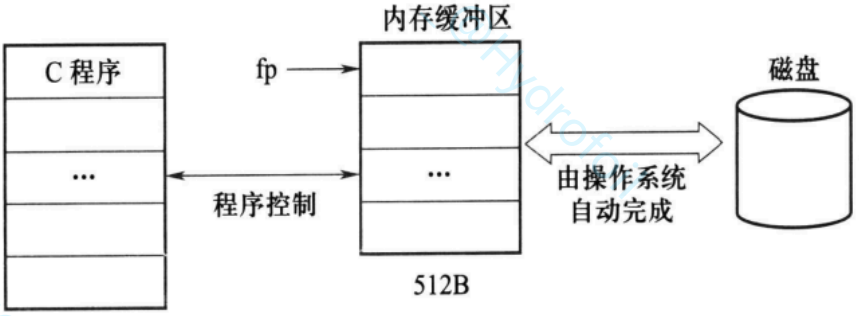


图 12.2　缓冲文件系统的工作原理图

- 磁盘数据的组织方式按**扇区**进行。每个扇区512B。一般**微型计算机上的文件缓冲区也定为512B**。
- 写数据时：把数据写入缓冲区直到写满512B -> 操作系统把数据写入磁盘的一个扇区 -> 清空缓冲区 -> 新的数据写入缓冲区

  读数据时：数据从磁盘导入缓冲区直到满512B -> c程序读入数据 -> 系统把下一个扇区的数据导入缓冲区

文件结构（结构体）与文件类型指针

- 自定义类型`typedef`：将c语言中已有的类型（包括已经自定义的类型）重命名，用新的名称代替已有的数据类型，常用于简化对复杂数据类型定义的描述。

```
typedef <已有类型名> <新类型名>; //新类型名一般用大写
typedef int* POINTER;
POINTER p1, p2;

typedef int NUM[10]; //定义了一个名为NUM的新类型，它是一个包含10个int元素的数组
NUM a; //这将创建一个名为a的数组，类型为NUM，即int[10]
```

e.g. 已知一个函数指针类型，它所指的函数返回值类型为空类型，接收两个参数：一个是字符指针类型，一个是整型。用`typedef`将该函数指针类型命名为`FunType`，具体形式为：

```
typedef void (*FunType)(char *, int);
```

- 文件结构体：用`typedef`将`struct`重命名为`FILE`的一个结构体，包含文件操作相关信息。其**成员指针**指向缓冲区，通过移动指针实现对文件的操作。**FILE的大小16B**

每个文件都有自己的FILE结构体和文件缓冲区。

下面是 FILE 文件类型的说明：

```
typedef struct{
        short           level;          /* 缓冲区使用量 */
        unsigned        flags;          /* 文件状态标志 */
        char            fd;             /* 文件描述符 */
        short           bsize;          /* 缓冲区大小 */
        unsigned char   *buffer;        /* 文件缓冲区的首地址 */
        unsigned char   *curp;          /* 指向文件缓冲区的工作指针 */
        unsigned char   hold;           /* 其他信息 */
        unsigned        istemp;
        short           token;
} FILE;
```

- 文件指针：`FILE *fp`，文件指针指向文件类型结构，FILE中的`curp`指向文件缓冲区中数据储存的位置。因此不能进行`fp++`或`*fp`操作。前者将**指向下一个FILE结构（如果存在）**

文件处理

- 文件处理步骤：定义文件指针 -> 打开文件 -> 文件处理（读写） -> 关闭文件
- **打开文件：**

```c
fp=fopen("文件名",  "文件打开方式");
//或者
char *p = "文件名";
fp=fopen(p, "文件打开方式");
```

`fopen()`运行原理：在磁盘中找到指定文件 -> **在内存中分配一个16B的单元保存FILE** -> **在内存中分配文件缓冲区单元（512B）** -> 返回`FILE`的地址给`fp`

`fopen()`执行成功返回对应文件的FILE结构地址；**失败返回`NULL`，此时需要`exit(0)`关闭打开的所有文件并终止程序运行**。

文件名若包含路径则要双写反斜杠取消转义，否则默认与c程序同一目录下

表 12.1　文件打开方式

文本文件(ASCII)		二进制文件	
使用方式	含　义	使用方式	含　义
"r"	打开文本文件进行只读	"rb"	打开二进制文件进行只读
"w"	建立新文本文件进行只写	"wb"	建立二进制文件进行只写
"a"	打开文本文件进行追加	"ab"	打开二进制文件进行写/追加
"r+"	打开文本文件进行读/写	"rb+"	打开二进制文件进行读/写
"w+"	建立新文本文件进行读/写	"wb+"	建立二进制新文件进行读/写
"a+"	打开文本文件进行读/写/追加	"ab+"	打开二进制文件进行读/写/追加

- **关闭文件：**

```c
fclose(fp);
```

成功关闭返回0。否则返回非0，此时也要加上`exit(0)`

文件关闭成功时，若缓冲区中有剩余数据（因为512B不会自动写入），则也将其强制写入磁盘扇区。若关闭失败，则**缓冲区中剩余数据丢失**。

- **文件读写：**

```
ch=fgetc(fp);//读到有效字符时后移指针，读到文件末尾时返回EOF

fputc(ch,fp);//成功则返回ch,否则返回EOF
//执行fput()与fget()时fp->curp会自动改变，即等价于
{
    *(fp->curp)=ch;
    fp->curp++;
}
```

注：EOF是一个值为-1的常量

```
fputs(s,fp); //s可以是字符数组名、字符串常量或字符型指针；末尾'\0'不写入文件；成功则返回s的最后一个字符，否则赶回EOF

fgets(s, n, fp); //s是待存入的字符数组名或指针，n是指定读入的字符数，最多读入n-1个。当达到n-1个，或读到换行符，或读到文件结束标志符EOF时，自动在字符串末尾追加'\0'（也就是第n个字符）；成功则返回字符串，否则返回NULL，此时s的内容不确定
```

```
fscanf(文件指针， 格式字符串，输入表);
fprintf(文件指针， 格式字符串，输出表);
fscanf(fp, "%d,%f", &n, &x);
fprintf(fp, "%d,%f", n, x);
//n与x在内存中以二进制储存，但文件以文本形式打开，这之间的格式转换由系统自动处理
```

```
fread(buffer, size, count, fp); //从二进制文件中读入数据块到变量
//buffer是待输入变量的指针，size是数据块的字节数，count表示要读写的数据块数
fwrite(buffer, size, count, fp); //向二进制文件写入数据块
fread(fa, 4, 5, fp); //表示从fp所指的文件中每次读4个字节（一个实数），连续读5次，存入fa中。
// 二进制文件的读写比文本文件更安全。因为无法用记事本打开
```

- 其他文件操作

```
rewind(fp); //用于将文件读写位置指针重新定位到文件首地址，一般用于重新读写

fseek(fp, offset, from); //用于移动指针。offset表示移动的偏移量。从前往后为正，否则为负。应为long型变量。若为常量则应加上后缀'L'。from表示偏移起始位置。取值为0，1，2或者SEEK_SET, SEEK_CUR, SEEK_END, 分别对应文件首部、当前位置和文件尾部。例如
fseek(fp, 20L, 0);

ftell(fp); //用于获取指针当前的位置（相对于文件开头，以字节数为单位）

feof(fp); //检查是否已到文件末尾。若到末尾则返回1，否则返回0

ferror(fp); //检查读写时是否出错。无错误则返回0，否则返回非0

clearerr(fp); //用来清除出错的标志和文件结束标志使他们为0值。
```

字符与字符串处理

```
k=(char)(cnt+65);
```

格式化输出

```c
printf("%.6d\n", 123); //至少输出6位，不足补0
printf("%6d\n", 123); //至少输出6位，不足左边空格补齐
printf("%-6d\n", 123); //至少输出6位，不足右边空格补齐

printf("%.6f\n", 3.1415926); //精确到小数点后6位，超出截断（末位四舍五入），不足补0
printf("%6f\n", 3.1415926); //同上
printf("%.f\n", 3.1415926); //保留0位，即四舍五入取整。与%.0f等价

printf("%.6g\n", 3.1415926); //对于g和G（易疏忽！），限制小数点前后总的输出位数（末位四舍五入）

printf("%.6s\n", "abcd efgh"); //最多输出6位，空格计算在内，不足不补齐，遇到\0停止输出
printf("%6s\n", "abcd\0efgh"); //至少输出6位，不足在左边空格补齐
printf("%-6s\n", "abcd\0efgh"); //至少输出6位，不足在右边空格补齐
```

注意，对实数进行末尾舍入输出时**会出现类似`-0.0`的情况**，有些题目会判错。

例如，$p \in (-0.05, 0)$，执行`printf("%.1lf\n", p);`，由于p仍是个负数，就会输出`-0.0`

解决方法：加一行`p=(-0.05<p && p<0) ? -p : p;`

```c
printf("%c\n", 'a'); //字符型
printf("%s\n", "abc"); //字符串

printf("%d\n", 10); //有符号整型十进制
printf("%u\n", 10); //无符号整型十进制

printf("%o\n", 10); //无符号整型八进制

printf("%x\n", 10); //无符号整型十六进制小写
printf("%X\n", 10); //无符号整型十六进制大写

printf("%f\n", 10.0); //浮点型float(小数形式)
printf("%e\n", 10.0); //浮点型float(指数形式)

printf("%lf\n", 10.0); //浮点型double(小数形式)
```

```
printf("%le\n", 10.0); //浮点型double(指数形式)

printf("%p\n", &a); //指针型，输出地址
```

```
getchar()
putchat()
```

输入的处理

要注意scanf()读到空格、回车或制表符时就会结束当前变量的读入。所以如果要读取一整行含空格字符串，就只能用gets()

同时，若**一行要读多种数据类型**，就不能用gets()。例如：

```
int a, b;
char st[100];
scanf("%d", &a);
gets(st);
scanf("%d", &b);
```

输入12 abc 34，就会将12读入a，将abd 34读入st，b成空的了。

正解：

```
scanf("%d%s%d", &a, st &b); //注意st本身就是指针，所以不用加&
```

先读入n，再读入n行，每行存为一个字符串：

输入样例:

```
3
Programming in C
21.5
Programming in VB
18.5
Programming in Delphi
25.0
```

```
char a[10][1000];
scanf("%d", &n);
for(int i=1;i<=n;i++)
{
    scanf("\n");
//加这一行因为gets函数的问题，上一个输入n是\n结束，如果不加这行，那么gets函数接收到的就是\n，它就结束了，什么都没读入.
//以后出现需要输入字符串但是前面又出现了scanf不妨加这一行，就可以不用for循环逐位输入，直接使用gets
    gets(a[i]);
}
```

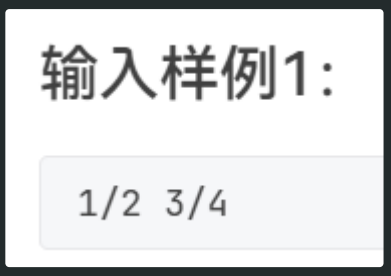
每行读入多种类型的数据，用空格分割：

输入样例:

```
5
00001 zhang 70
00002 wang 80
00003 qian 90
10001 li 100
21987 chen 60
```

```
char a[10][1000];
int n, m[10], n[10];
scanf("%d", &n);
for(int i=1;i<=n;i++)
{
    scanf("%d %s %d", &m[i], &a[i], &n[i]);
}
```

分数的输入：




```
int a,b,c,d;
scanf("%d/%d %d/%d", &a, &b, &c, &d);
```

注意：**如果a,b,c,d定义为float型则不能读入！**

求数组元素个数

```
printf("Number of elements: %zu\n", sizeof(arr) / sizeof(arr[0]));
// 元素个数（这种写法很有用）
```

部分函数集

调用math.h

```
int abs(int x);
double fabs(double x);
double pow(double x, double y); //x^y
double log(double x); //ln(x)
double log10(double x); //log(x)
double ceil(double x);
double floor(double x);
```

数值转换，调用stdlib.h

```
double atof(char *s); //字符串转双精度浮点数

int atoi(char *s);
//字符串转整数
//若s是非数的字符串，则返回0；若s中间含有空格或字母等，则将第一个异常字符之前的部分转换为int；
//s开头可以有'+'，'-'或空格

long atol(char *s);
//字符串转长整型
//若s是非数的字符串，则返回0；若s中间含有空格或字母等，则将第一个异常字符之前的部分转换为double；
//s开头可以有'+'，'-'或空格，可以是"3.14E-2"型指数，
```

字符判别，调用ctype.h

```
int isalpha(char c);
int isupper(char c);
int islower(char c);
int isdigit(char c);
char tolower(char c);
char toupper(char c);
//是：返回非0；否：返回0
```

零碎的注意点

`(-3)%5=-(3%5)`

双写`%`为表示输出一个`%`；双写`\`表示单个`\`

实验考小技巧：在考试前有时间试一下编译器，可以利用这段时间先把可能用到的函数给写了（质数判断、字符与子串的增删改查、数位分解、辗转相除求最大公约数）前提是得提前背熟

PTA刷题

函数题5-7（麦克劳林求和）

本题要求实现一个函数，用下列公式求$\cos(x)$的近似值，精确到最后一项的绝对值小于e：

$$\cos(x) = x^0/0! - x^2/2! + x^4/4! - x^6/6! + \cdots$$

函数接口定义：

```
double funcos( double e, double x );
```

其中用户传入的参数为误差上限`e`和自变量`x`；函数`funcos`应返回用给定公式计算出来、并且满足误差要求的$\cos(x)$的近似值。输入输出均在双精度范围内。

裁判测试程序样例：

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

double funcos( double e, double x );

int main()
{
    double e, x;

    scanf("%lf %lf", &e, &x);
    printf("cos(%.2f) = %.6f\n", x, funcos(e, x));

    return 0;
}

/* 你的代码将被嵌在这里 */
```

输入样例：

```
0.01 -3.14
```

输出样例：

```
cos(-3.14) = -0.999899
```

若对于每一项分别计算x^n和$n!$，$n!$会爆掉。

其实在上一项的基础上再乘以x^2再除以$n(n-1)$就得到了下一项，所以不用每次重新计算。而且下一项肯定是比上一项要小，所以不用担心爆掉。

另外题目要求最后一项的精确值小于e为止，可以在for循环中用`for(i=2;tmp>e;i+=2)`的条件，更简便。

函数题6-5（质数）

判断质数不要忘了特判1不是质数

函数题10-7（递归进制转换）

递归实现十进制转二进制

第一次提交的错误答案：

```
void dectobin(int n)
{
    if (n==0) printf("%d",0);
    else
    {
        dectobin(n/2);
        printf("%d",n%2 );
    }
}
```

这样写虽然保证了输入0，输出也是0；但是当输入大于0，由于递归每次将n除以2，最后n为0的时候还会进入`if (n==0)`这条语句，导致在最高位多输出一个不必要的0。

第二次提交正确：

```
void dectobin(int n)
{
    if (n>1)
    {
        dectobin(n/2);
    }
    printf("%d",n%2);
}
```

如果输入0，输出也是0；而当输入大于0，递归每次将n除以2，n在变为0的前一时刻肯定是1，此时直接进入`if (n==1)`这条语句，递归结束，不会输出多余的0。

另外，如果交换else里面两条语句的顺序，就做到了反序输出。

第三周作业T4（精度限制的无限小求和）

```c
#include<stdio.h>
int main()
{
    double eps, tmp=1, sum=0,flag=1,kk;
    scanf("%lf", &eps);
    if(eps>=1)
    {
        printf("sum = %.6f",tmp);
        return 0;
    }
    for (int i=1;tmp>eps;i++)
    {
        kk=3*i-2;
        tmp=1/kk;
        sum+=flag*tmp;
        flag=-flag;
    }
    printf("sum = %.6f",sum);
}
```

```c
#include<stdio.h>
int main()
{
    double eps, tmp=1, sum=0,flag=1,kk;
    scanf("%lf", &eps);
    if(eps>=1)
    {
        printf("sum = %.6f",tmp);
        return 0;
    }
    for (int i=1;tmp>eps;i++)
    {
        tmp=1/(3*i-2);
        sum+=flag*tmp;
        flag=-flag;
    }
    printf("sum = %.6f",sum);
```

```
}
```

函数题习题11-5（格式问题）

```
char *p,*q,a='\0';
p=&a;
q=N
printf("%c",p);
printf("%s",p);
printf("%c",a);
printf("%")
printf("%s",'\0');
```

递归正反序函数题

输出从i到1：

```
void print(int i)
{
 if(i>0)
 {
     printf("%d", i);
 }
 print(i-1);
}
```

输出从1到i：

```
void print(int i)
{
    if(i>1)
    {
        print(i-1);
    }
    printf("%d ", i);
}
```

字符串复制

本程序的功能是将字符串a的所有字符传送到字符串b中，要求每传送三个字符后再存放一个空格。

```
#include <stdio.h>
int main()
{
   int i,k=0;
   char a[80], b[80], *p;
   p=a;
   gets(p);
   while(*p)
   {  for(i=1;  [1分]  ; p++, k++, i++)  b[k]=*p;
      if(  [1分]  )  { b[k]=' ' ;  k++; }
   }
   b[k]='\0';
   puts(b);
   return 0;
}
```

```
1  //第一空
2  i<=3 && *p //确保次搬运3个字符，但是末尾指针不越界
3  //第二空
4  i==3 //确保末尾未满3个字符就跳出循环时不加空格
```

2024-2025秋冬肖少拥班c类第二次小测T1（真的哭死）

本题的要求很简单，就是求`N`个数字的和。麻烦的是，这些数字是以有理数`分子/分母`的形式给出的，你输出的和也必须是有理数的形式。

输入格式:

输入第一行给出一个正整数`N`（≤100）。随后一行按格式`a1/b1 a2/b2 ...`给出`N`个有理数。题目保证所有分子和分母都在长整型范围内。另外，负数的符号一定出现在分子前面。

输出格式:

输出上述数字和的最简形式 —— 即将结果写成`整数部分 分数部分`，其中分数部分写成`分子/分母`，要求分子小于分母，且它们没有公因子。如果结果的整数部分为0，则只输出分数部分。

输入样例1:

```
5
2/5 4/15 1/30 -2/60 8/3
```

输出样例1:

```
3 1/3
```

输入样例2:

```
2
4/3 2/3
```

输出样例2:

```
2
```

输入样例3:

```
3
1/3 -1/6 1/8
```

输出样例3:

```
7/24
```

```c
#include<stdio.h>
long gcd(long int c, long int d)
{
    long int a, b, r;
    a=(c>d)?c:d;
    b=(c>d)?d:c;
    while(b!=0)
    {
        r=a%b;
        a=b;
        b=r;
    }
    return a;
}
int main()
{
    int n;
```

```
    long int a, b, aa, bb;
    scanf("%d", &n);
    scanf("%ld/%ld", &aa, &bb);

    long k=gcd(aa, bb);
    bb=bb/k;
    aa=aa/k;
//n=1时记得约分
    for(int i=1;i<n;i++)
    {
        scanf("%ld/%ld", &a, &b);
        aa=aa*b+a*bb;
        bb=bb*b;
        long k=gcd(aa, bb);
        bb=bb/k;
        aa=aa/k;
    }
    if(aa%bb==0)
        printf("%ld",aa/bb);
    else if(aa/bb==0)
        printf("%ld/%ld",aa%bb,bb);
    else
        printf("%ld %ld/%ld",aa/bb,aa%bb,bb);
}
```

历次作业与历年卷错题

输入一个非负整数，从高位开始逐位分割并输出它的各位数字。例如，输入9837，输出9 8 3 7

```
int digit, number, pow, t_number;
scanf ("%d", &number);
t_number = number;
pow = 1;
while ( t_number>0 1 分 ) {
    pow = pow * 10;
    t_number = t_number / 10;
}
while ( pow >= 1 ) {
    digit = number/pow 1 分 ;
    number = number%pow 1 分 ;
    pow = pow / 10;
    printf ("%d ", digit);
}
printf ("\n");
```

答案：第一空改为`t_number>=10`

若有以下说明，且0<=i<10，则对数组元素的错误引用是（）。

```
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, *p = a, i;
```

- (●) A. *(a+i)
- ( ) B. a[p-a+i]
- ( ) C. p+i
- ( ) D. *(&a[i])

评测结果　答案错误

得分　0 分

设有以下程序段，若 k 为 int 型变量且 $0 \leq k < 4$，则对字符串的不正确引用是（ ）。

```
char str[4][10] = {"first", "secone", "third", "fourh"}, *strp[4];
int i;

for(i = 0; i < 4; i++){
    strp[i] = str[i];
}
```

- A. strp
- B. str[k]
- C. strp[k]
- D. *strp

评测结果 答案错误

得分 0 分

答案：A

C语言中，在成功打开一个文件后，可以使用 fopen() 1 分 来获取文件缓冲区的FILE结构信息。

评测结果 答案错误

得分 0 分

答案：文件指针

4. 若变量已正确定义，表达式 (j=3, j++) 的值是_____.

A．0 B．3 C．4 D．5

6. 下列运算符中优先级最高的是_____.

A．^= B．++ C．[ ] D． &&

8. 设有定义 int a[3][3]={{1,2,3},{4,5,6},{7,8,9}}; 现要使 p=a;则 p 的定义必须为_____.

A．int p[3][3]; B．int *p[3]; C．int (*p)[3]; D．int **p;

1. 表达式 1<<4-1|3/2 的十进制值为_____.

4. 假设 int x = -4; 则循环语句 while(-6<x<-2) ++x; 运行以后的 x 值是

5. 以下程序段的输出是_____.

```
int i;  char s[80]={"apple\0grape\0pear\0coco\0"};
for(i=0; i<2; i++) s[strlen(s)] = '\n';
printf("%d#%d\n",strlen(s),sizeof(s) );
```

4. 变量 a=1, b='1', c=1.0, d="1"，下列运算不能进行的是_____.

A. b/a--　　B. ~a|!b

C. c^++a　　D. d+a-b

6. 下面四个选项中，均非浮点数正确表示的选项是_____.

A. 160.　0.12　e3　　B. -.18　123e4　0.0

C. -e3　.234　1e3　　D. 123　2e4.2　.e5

1. 若整型变量 x=2，则表达式 1<x<<x<4 的值为_____.

6. 下列程序段的输出是_____.

```
int c=0,k;
for (k=1;k<3;k++)
    switch (k)
    {
        default: c+=k;
        case 2: c++;
        case 4: c+=2;
    }
```

12. 以下程序段运行的结果是_____.

```
int i,j,sum;
for(i=11;i>=1;i-=3){
    for(j=1;j<=i;j+=2);
    sum+=i*j, sum=i+j;
}
printf("%d\n",sum);
```

注意每次计算`sum`时`j`的值