第1章 C语言程序设计基础

1.1 程序设计及程序设计语言

程序是能被机器识别并执行的一系列的**指令代码**，这些指令代码是用程序设计语言描述的。

程序设计语言是人与计算机对话的工具。

程序设计语言一般分为**低级语言**和**高级语言**。低级语言一般包括**机器语言**和**汇编语言**。

1.2 C语言的特点

- 语言简洁、紧凑，使用方便、灵活：32个关键字，9中控制语句。
- 运算符丰富：34个运算符。
- 数据结构丰富，具有现代化语言的各种数据结构；
- 具有结构化的控制语句；
- 语法限制不太严格，程序设计自由度大；
- 能进行位操作，实现汇编语言的大部分功能，直接对硬件进行操作；
- 生成目标代码质量高，程序执行效率高；
- 程序可移植性好。

1.3 C程序结构

（1）C程序的基本单位是函数

C程序由函数构成，C语言是函数式的语言。

一个C源程序**至少**包含**一个**main函数，也可以包含**一个**main函数和**若干个**其他函数。

（2）main函数是程序的起始点

main表示**主函数**。它是每一个C语言程序的执行**起始点**（入口），每个C语言程序都必须有一个main函数。

不论main函数在程序中的什么位置，程序总是从main函数开始执行。

（3）函数由函数首部和函数体两部分组成

- 函数首部：函数的第一行：`返回值类型 函数名([参数列表])`
- 函数体：函数首部后用一对`{}`括起来的部分。函数体一般包括声明和执行两部分。

注意：函数可以没有参数，但是后面的`{}`不能省略。

（4）C程序书写格式自由

一行可以写几个语句，一个语句也可以写在多行上。

- 1、预处理一行写不下：把一个预处理指示写成多行**要用 \ 续行**，因为根据定义，一条预处理指示只能由一个逻辑代码行组成。

- 2、字符串常量跨行：在行尾使用 \ ，然后回车换行，就可以把字符串常量跨行书写，注意下一行顶格写。
- 3、正常程序一行写不下：把C代码写成多行则不必使用续行符，因为换行在C代码中只不过是一种空白符，在做语法解析（语法分析）时所有空白符都被丢弃了。

```c
#include<stdio.h>

#define PI 3.14\
526

int main(){
    char* s = "fhjak\
    dsd";
    int a
    =
    10
    ;
    printf("%d\n",a);  // 10
    printf("%f\n", PI);  // 3.145260
    printf("%s\n", s);  // fhjak   dsd
    printf("fdvdfv\
    dfdfd");            // fdvdfv  dfdfd
    return 0;
}
```

注意：每条语句必须以`;`结尾，表示语句的结束。

（5）注释

注释可以提高程序的可读性。注释可以放在任何位置，编译会跳过注释。

- 单行注释：`//`后面的内容是注释。
- 多行注释：需要注释的内容放在`/*`和`*/`之间。

（6）C语言本身不提供输入/输出语句

输入/输出操作是通过库函数（`scanf()/printf()`）完成的。

1.4 源程序的编辑、编译、链接与运行

用高级语言编写的程序叫做源程序，简称程序。

编辑：对象是源程序，是以ASCII代码的形式输入和存储的，不能被计算机执行。

包括以下内容

- 将源程序逐个字符输入到计算机内存
- 修改源程序
- 将修改好的源程序保存在磁盘文件中

编译：将已编辑好的源程序（已存储在磁盘文件中）翻译成二进制的目标代码。

这种转换的过程称为**汇编**，完成汇编的系统软件称为**汇编程序。**

编译后得到的文件是`.obj`后缀或者`.o`；

链接：将各模块的二进制目标代码与系统标准模块链接处理后，得到具有绝对地址的可执行文件，它是计算机能直接执行的文件。

链接后得到的文件是`.exe`后缀。

执行：一个经过编译和链接的可执行的目标文件只有在操作系统的支持和管理下才能执行。

总结：C语言源程序文件的后缀是.c，经过编译后生成的文件后缀是.obj，经过链接后生成的文件后缀是.exe。

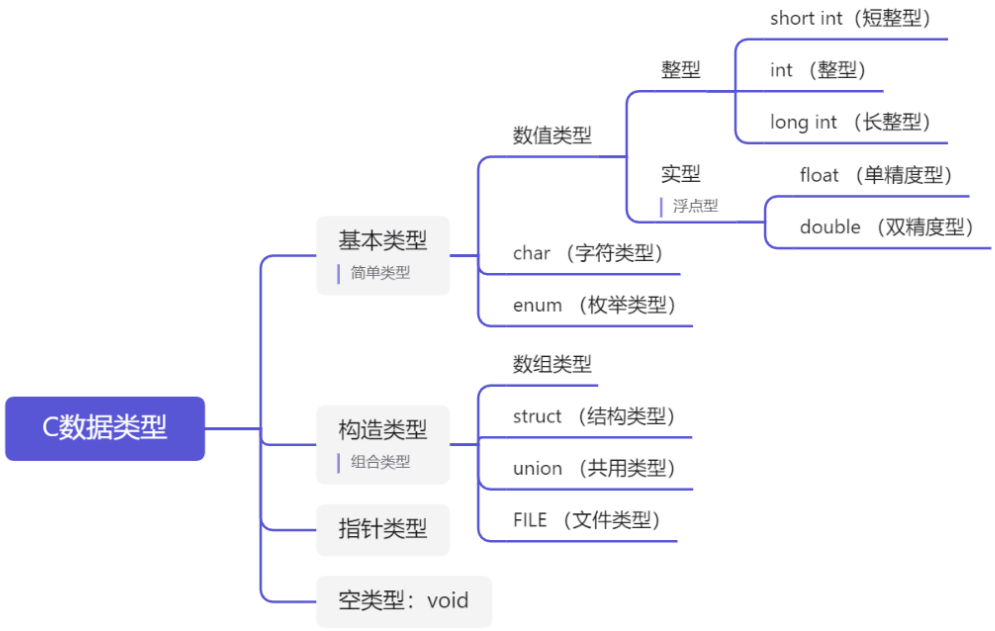
第2章 C语言的数据类型与基本操作

数据是指能够输入计算机并被计算机处理的数字、字母和符号的集合。

数据与操作是程序的两个要素，数据处理是计算机的用途之一。

C语言的数据结构是以数据类型形式出现的：

- 从广义上根据量的值是否可变，可分为**常量**和**变量**两种数据类型。
- 根据数据复杂程度，划分为**基本数据类型**和**构造数据类型**这两大类。
- 构造数据类型指可以通过其他的数据类型进行构造，可由程序员自定义，故又称为 "**自定义数据类型**"。



无论什么类型数据都有常量和变量两种表现形式。任何数据类型都必须且只能以指定的类型和形式在程序中出现。

2.1 常量与变量

2.1.1 常量

在程序运行过程中，其值不改变的量称为**常量**。

（1）整型常量

整型常量通常是指数学上的整数

- 十进制：15， 19...
- 二进制：**以数字0和字母b(B)开头**，0b0010，0B0100...
- 八进制：**以数字0开头**，05，012...

- 十六进制：**以数字0和字母x(X)开头**，0x10，0Xff...

注意：八进制和十六进制只是数的书写方式，对数的存储方式没有影响。因为整数都是二进制形式存储的，跟表示方式无关。

（2）实型常量

也称数值常量，有正值和负值之分，只能用十进制形式表示。可以使用小数形式或者指数形式表示。

注意：当使用指数形式表示时：e/E前后必须都要有数字，且e/E之后的数字必须是整数。

- e/E前面的数叫做**尾数**，e/E后面的数叫做**阶码/指数**。
- 尾数部分的小数点不能省略；当尾数尾零点几时，可以省去0，但不能省去小数点。
  比如$0.5*10^3$，表示为：`.5e3`。
- 在字母e或E的前后以及数字之间**不得**插入空格。
- 指数必须为整数，正负均可；e / E之前必须有数字。如：$12345=1.2345*10^4$表示为1.2345e4。

实型常量不分单精度型和双精度型，但是可以赋给一个float或者double型的变量。

实型常量必须要有小数点，如果小数点前后为0的话，可以省略0。

以下实型常量的形式都是正确的：

```
float a = -.18;
float b = 160.;
float c = .234;
printf("%f %f %f\n", a, b, c);  // -0.180000 160.000000 0.234000
```

（3）字符常量

是一对用**单引号**括起来的一个字符。

注意：单引号**不是**字符常量的一部分。

（4）字符串常量

简称字符串。是用**双引号**括起来的零个或多个字符系列。

注意：双引号**不属于**字符串常量的一部分。

字符串中**字符数**称为该字符串的长度。

字符串常量在机器内存储时，系统会自动在末尾加一个“字符串结束标志”：`\0`。

比如字符串"hello"，在内存中存储为：

```
h   e   l   l   o   \0
```

也就是说字符串在存储时要多占用一个字节来存储"\0"。

而实际上每个字符都是用其ASCII码来存储的，"\0"的ASCII码值为0，所以上面的字符串实际上是这样存储的：

```
104 101 108 111 0
```

这里就要注意字符串常量和字符常量的区别，比如 'A' 和 "A"的存储：

```
'A' ：字符常量，存储为：
65
"A"：字符串常量，存储为：
65 0
```

另外，`""`表示一个空字符串，在内存中占用一个字节，仅存储一个'\0'。

（5）符号常量

C语言中也可以用标识符代替常量。

使用`#define`命令行可以定义一个常量，这样用一个标识符代表一个常量的，称为符号常量。符号常量的值在其作用域中不能被改变，也不能再次赋值。

（6）布尔类型

在C89标准中，没有定义布尔类型；在C99标准中，提供了`_Bool`类型。

`_Bool`是无符号的整型变量，C99中还提供了一个新的头文件`<stdbool.h>`，其中提供了`true`和`false`，分别代表1和0。

2.1.2 变量

变量是指在程序运行中其值可以发生变化的量。

变量在内存中占据一定的存储单元，该存储单元中存放变量的值。

（1）变量的声明

C语言规定，在程序中用到的每个变量都要声明它们属于哪一种类型。

声明格式：`数据类型 变量名;`或`数据类型 变量名1, 变量名2...;`

比如：

```c
int x;
int y;
```

等价于：

```c
int x, y;
```

注意：

- 在一个定义语句中，可以同时定义多个变量，变量之间使用`,`隔开。
- 对变量的定义可以在函数体之外，也可以在函数体或复合语句中。

（2）变量的初始化

C语言允许在声明变量的同时对其初始化。

比如：

```c
int a = 10;
int b = 20;
```

也可以对声明的变量一部分初始化：

```
int a, b = 20, c;
```

注意：

- 不同类型的数据在内存中占据不同长度的存储区，而且采用不用的表达方式（数据在机器内部的表达方式）。
- 一种数据类型对应着一个值的范围。
- 一种数据类型对应着一组允许的操作。

总之，在引用变量前必须先用声明语句指定变量的类型，这样在编译时就会根据指定的类型分配存储空间，并决定数据的存储方式和允许操作的方式。

2.1.3 转义字符

转义字符有两种

- 字符转义
- 数字转义

C语言用反斜杠 `\` 来表示转义字符的起始符，有三种表示方法

- 反斜杠后面跟一个字母代表一个控制字符；
- 反斜杠后面跟一个符号，比如`\\`代表`\`，`\'`代表`'`，`\"`代表`"`；
- 还能用来输出不能直接从键盘上输入或不能用字符常量书写出的ASCII字符。

字符转义系列

字符形式	功能
\n	换行
\t	横向跳格（即跳到下一个输出区）
\v	竖向跳格
\b	退格
\r	回车
\f	走纸换页
\\	反斜杠字符`\`
\'	单引号字符`'`

数字转义系列

字符形式	功能
\ddd	1-3位八进制所代表的字符
\xhh	1-2位十六进制所代表的字符

作为字符常量使用时，转义字符必须用一对单引号括起来。如：'\n'。

当转义字符用来输出不能直接从键盘上输入或不能用字符常量书写出的ASCII字符时，要在 `\` 后面跟一个代码值，这个代码值最多用**3位不加前缀的八进制数**或者**2位以x为前缀的十六进制数**表示。

使用八进制时：

其实`\ddd`最多只支持三位数字，并且三位数字也不是任意的(每个数字不能大于8，一旦大于8它就不是八进制数了)，一旦大于等于八进制数`'/400'`（十进制256=8进制400）就超过了ASCII码的范围，编译器就会报错。另外大于三位，或者遇到非八进制数字时则转换结束，直接取末尾数字。

```
int main(){
    char b = '\65';  // 5
    char a = '\658'; // 8
    char c = '\6587';  // 7
    char d = '\65876';  // 6
    // char e = '\658765';  // err:字符常量中字符太多
    // 这里的差异取决于编译器。
    return 0;
}
```

比如以上代码，当转义符`\`后的数字不符合八进制时，直接取最后一个数字，并忽略掉前面的所有字符。

无效转义

当转义字符`\`后跟非特殊字符，则反斜杠会被忽略。

2.2 标识符和关键字

基本的语法单位是指具有一定语法意义的最小语法成分。C语言的语法单位分为六类：标识符、关键字、常量、字符串、运算符及分隔符。

标识符

是给程序中的实体（变量、常量、函数、数组及结构体）以及文件所起的名字。可以由程序员指定，也可以由系统指定。

C语言的标识符可分为：关键字、用户标识符、自定义标识符三类。

标识符命名规则：

- 以字母（A-Z, a-z）和下划线 "_" 开头，由字母、数字（0-9）和下划线组成。
- 建议用户尽量不用下划线开头定义标识符。系统内部使用了一些以下划线开头的标识符，防止冲突。
- 长度无规定。
- 区分大小写。

标识符命名规范：

- 见名知意。
- 变量名和函数名用小写，符号常量用大写。
- 在容易出现混淆的地方避免使用容易认错的字符。如`l`和1，`o`和0。

关键字

是由编译程序**预定义的、具有固定含义的、在组成结构上均由小写字母构成**的标识符。因此，用户不能用它们作为自己定义的变量、常量、类型或函数的名字。

ANSI C标准中定义了32个关键字，C99新增了5个关键字。

32个关键字如下：

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

2.3 整型数在计算机中的存储方式

计算机中，内存储器的最小存储单位称为“位”(bit)，每一个位中或存放0，或存放1，因此称为二进制位。

大多数计算机把8个二进制位组成一个“字节”(byte)，并给4个字节分配一个地址。若干字节组成一个“字”(word)，用一个字来存放一条机器指令或一个数据。一个字含多少个字节随机器而不同，如果一台计算机系统字长为4个字节（1个字节为8个二进制位），就称这台计算机的字长为32位。

一个 int 数据通常用4个字节（32位二进制）存放，其中最高位（最左边的一位）用以存放整数的符号：

- 0，表示正数。
- 1，表示负数。

原码反码补码

正数的原码、反码、补码相同，三码合一。

负数的原码、反码、补码：

- 原码：把整数的绝对值用二进制表示，最高位用于表示符号，0正1负。
- 反码：原码除符号位外各位取反（0变1，1变0）。
- 补码：反码加1（最后一位+1）。

两数的**补码之和**等于**两数和的补码**。

在采用补码运算时，符号位也当作一位二进制数一起参与计算，结果为补码。

引入补码后，计算机中所有的加减运算都可以用补码的加法实现，反码只起原码转换补码的中介作用。

char型数据的存储形式

在C语言中，**char型数据在内存中是以“ASCII码”形式存储的。**C语言中，将一个字符常量放到一个字符变量中，实际并不是把该字符本身放到内存单元中，而是将与该字符相对应的ASCII码放到存储单元中。

2.4 有符号的数据类型和无符号的数据类型

在内存中存储整数时，一般用最高位（即最左边一位）表示符号，以0表示正数，以1表示负数，数值是以补码形式存放的。

之前提到的`int short long`型数据，都是指带符号的，也就是含有`signed(带符号)`，因此，`int`等价于`signed`，`short`等价于`signed short`，`long`等价于`signed long`。

C语言还允许使用无符号（`unsigned`）的整型数据，它将二进制形式的最左位不作为符号位，而与右边各位一起用来表示数值。也就是说，`unsigned int`只能用来放正数。

字符型的数据也有`singed`和`unsigned`，字符型的数据占一个字节（8位）。ASCII字符的允许范围是0-127，用7位就可以表示，最左边一位补0。

实型数据没有`unsigned`和`signed`之分，都带符号。

基本数据类型的数据所占空间与取值范围

类型	类型标识符	长度（Byte）	取值范围及精度
字符型	char（字符型）	1	-128~127
	unsigned char（无符号字符型）	1	0~255
	【signed】char（有符号字符型）	1	-128~127
整型	int（整型）	4	-2147483648~2147483647
	unsigned int（无符号整型）	4	-0~4294967295
	【signed】int（有符号整型）	4	-2147483648~2147483647
	short【int】（短整型）	2	-32768~32767
	unsigned short【int】（无符号短整型）	2	0~65535
	【signed】short【int】（有符号短整型）	2	-32768~32767
	long【int】（长整型）	4	-2147483648~2147483647
	unsigned long【int】（无符号长整型）	4	0~4294967295
	【signed】long【int】（有符号长整型）	4	-2147483648~2147483647
实型	float（单精度浮点数	4	-3.4*10^-38~3.4*10^38，6位有效数字
	double（双精度浮点数	8	-1.7*10^-308~1.7*10^308，16位有效数字

C99中提供了额外的两个类型：`long long int`和`unsigned long long int`。

2.5 运算符和表达式

运算（即操作）是对数据的加工，C语言对数据的基本操作和处理几乎完全是由运算符来完成的，这些符号称为运算符或操作符。

运算符按操作数的数目分类：

- 单目（一元）运算符
- 双目（二元）运算符
- 三目（三元）运算符

按运算符的功能分类：

- 算术运算符

- 关系运算符
- 逻辑运算符
- 自增、自减运算符
- 赋值运算符
- 条件运算符

具体归纳如下

类别	运算符
算术运算符	+、-、*、/、++、--
关系运算符	>、<、==、>=、<=、!=
逻辑运算符	!、&&、||
位运算符	<<、>>、|、^、&
赋值运算符	=，及其扩展赋值运算符
条件运算符	？：
逗号运算符	,
指针运算符	*、&
求字节运算符	sizeof
强制类型转换运算符	(类型)
分量运算符	.、->
下标运算符	[]
其他	如函数调用运算符

表达式的运算规则是由运算符的功能和运算符的优先级与结合性所决定的。为使表达式按一定的顺序求值，编译程序时所有运算符分成若干组，按运算符执行的先后顺序为每组规定一个等级，称为运算符的优先级，优先级高的运算符先执行运算。

处于同一优先级的运算符顺序称为运算符的结合性。运算符的结合性有从左到右和从右到左两种顺序，简称左结合和右结合。

关于运算符优先级

2.5.1 赋值运算符和赋值表达式

`=`就是赋值运算符，由赋值运算符组成的表达式就是赋值表达式。它的一般形式为：`<变量> = <表达式>`。将赋值运算符右侧的表达式的值赋给左侧的变量。

赋值的含义是将赋值运算符右边的表达式的值存放到左边变量名称标识的存储单元中。

注意：

- 赋值运算符**左边必须是变量**，右边可以是常量、变量、函数调用或者由常量、变量、函数调用组成的表达式。
- 赋值符号`=`不同于数学中的等号，没有相等的含义。
- 赋值运算时，当赋值运算符两边的数据类型不同时，将由系统自动进行类型转换。
  转换原则是：先将赋值号右边表达式的类型转换为左边变量的类型，然后赋值。

- 赋值运算符的优先级仅仅高于逗号运算符。

复合赋值运算符

在赋值符`=`前面加上某些运算符，可以构成复合赋值运算符。它的一般形式为：`<变量> 算术复合赋值运算符 <表达式>`。

需要注意的是，复合赋值运算符的右边是一个**整体**。

比如：`a *= b + c;`等价于`a = a * (b + c);`而不是`a = a * b + c;`。

2.5.2 算术运算符和算术表达式

算术运算符包括：`+ - * / % ++ --`。

双目运算符`+ - * /`的操作数可以为任何**整数**或**浮点数**。`+ -`还可以用于指针加或减一个整数。

双目运算符两边的两个数类型不同时，会自动进行“小转大”的类型转换。

> 双目运算符的两个操作数类型可以不同，运算前遵循 类型的一般算术转换规则 自动转换成相同的类型，运算结果的类型与转换后操作数的类型相同。类型的一般算术转换规则的基本原则是：值域较窄的类型向较宽的类型转换，也就是小类型向大类型转换。

对于除运算符`/`，如果两操作数都是整数，则执行整数除，运算结果也是整数，值为商的整数部分，小数部分被截去；否则执行实数除，运算结果是浮点数。

对于求余运算符`%`，规定两操作数必须都为**整数**，运算结果也为整数，值为左操作数除以右操作数所得的余数。

对于`/`和`%`用于负数时：

- C89标准规定：如果两个操作数中有一个是负数，除法运算的结果既可以向上取整也可以向下取整。
- C99标准中：除法的结果总是向零截取的，`i%j`的值的符号与`i`相同。

比如：

```c
#include<stdio.h>

int main(){
    printf("%d %d %d %d\n",-9/7, 9/-7, -9%7, 9%-7);  // -1 -1 -2 2
    return 0;
}
```

单目运算符`++`和`--`既可以作变量的前缀，也可以作变量的后缀：

- 前缀式，先自增/自减，后运算/引用。
- 后缀式，先运算/引用，后自增/自减。

注意：`++、--`运算符只能用于整型变量，**不能用于常量或表达式。**

2.5.3 长度测试运算符sizeof

`sizeof`可用来测试某个类型的变量所**占用**计算机**内存**空间的**字节长度**。格式为：`sizeof(类型名)`。

基本数据类型所占字节长度

类型	字节长度
short	2
int	4
long	4
long long	8
double	8
float	4
char	1

2.5.4 关系运算符和关系表达式

关系运算符

实际上就是比较运算，即比较两个对象的大小。

关系运算符

符号	优先级
<（小于） >（大于） <=（小于或等于） >=（大于或等于）	高
==（等于） !=（不等于）	低

说明：

- 关系运算符为双目运算符，结合方向为自左至右。
- 关系运算符的结果为真（1）或假（0）。**C语言中没有逻辑值。**
- 算术运算符高于关系运算符。
  比如：`a = 3，a >= 2 + 4;`等价于`a = 3，a >= (2 + 4);`。

关系表达式

由关系运算符、运算对象以及小括号组成的表达式称为关系表达式。当表达式成立时，结果为1，否则，结果为0。

注意：

- 一个关系式中含有多个关系表达式时，要注意与数学式的区别。比如：`0<x<6`。
- 应避免对实数相等或不等的判断，因为实数在内存中存放时有一定误差。

2.5.5 逻辑运算符和逻辑表达式

逻辑运算符

逻辑运算符具有自左至右的结合性。

逻辑运算符

运算符	含义	级别
!	非	高
&&	与	中
||	或	低

`&&`和`||`是双目运算符；`!`是单目运算符，应该出现在运算对象的左边。

优先级：`!`（逻辑非）> 算术运算 > 关系运算 > `&&`（逻辑与）> `||`（逻辑或）> 赋值运算。

比如：`! (5>3) && (2<4)`，等价于`(!(5>3)) && (2<4)`。

逻辑表达式

用逻辑运算符将关系表达式连接起来，构成逻辑表达式。逻辑表达式的结果或者为1（真）或者为0（假）。

在进行逻辑运算时，C语言规定：非0得真，0得假。意思是只要是非0的数，都是“真”。

短路现象

> C语言逻辑表达式的特性：在计算逻辑表达式时，只有在 必须执行下一个表达式才能求解时，才求该表达式，即并不是所有的表达式都被求解。

短路与：逻辑与`&&`运算表达式中，当`&&`左边的表达式为假(0)时，整个逻辑表达式为假(0)，`&&`右边的表达式不再执行（后边的表达式被短路）。

短路或：逻辑或`||`运算表达式中，当`||`左边的表达式为真(1)时，整个逻辑表达式为真(1)，`||`右边的表达式不再执行（后边的表达式被短路）。

2.5.6 条件运算符与条件运算表达式

条件运算符`?:`是C语言唯一的三目运算符。一般格式为：`表达式1 ? 表达式2 : 表达式3`。

它的执行顺序为：`?`前面的为真，结果则是`?`后面的表达式，`?`前面的为假，结果则是`:`后面的表达式。

- 先执行表达式1，判断结果是否为真，
- 为真，则执行表达式2，并以表达式2的结果作为整个表达式的结果；
- 为假，则执行表达式3，并以表达式3的结果作为整个表达式的结果。

注意：

- 条件运算符的优先级仅高于赋值运算符以及逗号运算符（倒数第三优先级）。
- 结合方向为自右至左。比如`a>b?a:b>c?b:c;`等价于`a>b?a:(b>c?b:c);`，这里括号指表示结合性，不表示先运算。

当同时出现多个`? :`时，从右边往左打小括号：`a>b?a:(b>c?b:(c>d?c:d));`

注意以下程序：

```
int main(){
    int a = 2, b;
    a > 0 ? b = a : b = -a;
}
```

编译的时候在程序的第三行会出现报错：

```
[Error] lvalue required as left operand of assignment
```

这是因为条件运算符的优先级比赋值运算符高，当它们混合运算的时候先结合条件运算符。

以上程序我们想要效果是这样的：

```
a > 0 ? (b = a) : (b = -a);
```

而以上程序实际上的效果是这样的：

```
(a > 0 ? (b = a) : b) = -a;
```

所以在多个运算符混合运算时，最好是按照我们的意愿添上小括号。

2.5.7 逗号运算符与逗号表达式

逗号运算符又称"顺序求值运算符"。一般格式为：`表达式1，表达式`或者`表达式1，表达式2，表达式3，...，表达式n`。

运算过程是：自左至右依次计算每一个表达式，最后以最后一个表达式的值作为整个逗号表达式的结果。

注意：

- 逗号表达式可以和另一个表达式组成一个新的逗号表达式。
  `(a=6，a*3)，a+10;`该逗号表达式的结果为：16。
- 并不是所有出现逗号的地方都是逗号表达式。
  比如：`int a，b，c;`或者`void fun(int a，int b){}`，这里的逗号只是间隔符。
- 逗号表达式只是把各个表达式**串联起来**。
  比如：`for(i = 0，j = 0; ;){}`这里只是希望`i j`都能被赋值。

2.5.8 位运算

位运算的作用是对运算对象按照二进制位进行操作的运算，它能够对字节或字中的 实际位 进行检测、设置或位移，它运算的对象只能是**字符型**或**整型**变量以及它们的变体，对其他类型的数据不适用。

位运算符

位运算符	作用	举例
~	按位取反	~a，对a中**全部位**取反
<<	左移	a<<2，将a中各位全部左移2位
>>	右移	a>>2，将a中各位全部右移2位
&	按位与	a&b，将a和b中各位进行与运算
|	按位或	a|b，将a和b中各位进行或运算
^	按位异或	a^b，将a和b中各位进行异或运算

位运算符还可以和赋值运算符相结合，成为位运算赋值操作。

位运算赋值操作

位运算符	作用	举例	等价表达式
<<=	左移赋值	a <<= n	a = a << n
>>=	右移赋值	b >>= n	b = b >> n
&=	位与赋值	a &= b	a = a & b
|=	位或赋值	a |= b	a = a | b
^=	位异或赋值	a ^= b	a = a ^ b

按位取反（~）

单目运算符，用来对一个二进制位取反。

规则：1变0，0变1。

按位与（&）

双目运算符，对两个二进制位进行“与”运算。

规则：有0为0，全1为1。即：0&1=0，1&0=0，0&0=0，1&1=1。

按位与的功能：

- **清零。**可以将一个数和与之相反的数（可由~操作获得）进行与运算，达到清零的效果。
  比如：a&~a。
- **取一个数中的某些指定位。**
  假设要取X的指定位，找一个数，用来对应X要取的位，该数对应X的位为1，其余位为零，此数与X进行“与运算”可以得到X中的指定位。
  比如：设X=10101110，取X的低4位，用 X & 0000 1111 = 0000 1110即可得到；

按位或（|）

双目运算符，对两个二进制位进行“或”运算。

规则：有1为1，全0为0。即：0|1=1，1|0=1，0|0=0，1|1=1。

按位或的功能：

- **将一个数的某些特征位 置为1。**
  假设X，找到一个数，用来对应X要置1的位，该数的对应位为1，其余位为零，此数与X相或可使X中的某些位 置1。
  将X=10100000的低4位置1 ，用 X| 0000 1111 = 1010 1111即可得到。

按位异或（^）

双目运算符，对两个二进制位进行“异或”运算。也称 **XOR** 运算符。

规则：

- 相同为0，相异为1。即：0^1=1，1^0=1，0^0=0，1^1=0。
- 两个相同的数异或为0。
- 任意一个数与0异或都是这个数本身。

按位异或的功能：

- 翻转特定位。
  比如：X=10101110，使X低4位翻转，用X ^ 0000 1111 = 1010 0001即可得到。
- 与0相异或保留原值。
- **不用中间变量即可交换两个数的值。**

```
int main(){
    int a = 2, b = 3;
    // a : 010 ; b : 011
    a ^= b;  // a : 001
    b ^= a;  // b : 010
    a ^= b;  // a : 011
    return 0;
}
```

按位异或相当于**没有进位的加法。**

在二进制中，1+1=10，第一位变成0，向第二位进1，而异或使第一位变0，但是没有向第二位进1。所以说，异或可以当作无进位加法。

左移（<<）

双目运算符，对一个数的二进制位向左移动指定位数。

规则：

- 左边丢弃，右边补0。
- 左移 1 位相当于乘以一个2。

右移（>>）

双目运算符，对一个数的二进制位向右移动指定位数。

规则：

- 算术右移（考虑符号位）：右边丢弃，左边补原符号位。
- 逻辑右移（不考虑符号位）：右边丢弃，左边补0。
- >>一位相当于除以一个2（结果取整）。

C语言中，对于移位操作执行的是逻辑左移和算术右移，不过对于无符号类型，所有的移位操作都是逻辑的（不考虑符号位）。

2.6 不同类型数据间的转换

C语言允许一种类型的值转换为另一种类型。以下情况会引起类型转换：

- 当双目运算符的两个操作数类型不相同时，引起一般算术转换，或称运算符转换。
- 当一个值赋于一个不同类型的变量时，引起赋值转换。
- 当一个值被强制为另一个类型时，引起强制类型转换。
- 当某个值作为参数传给一个函数时，引起函数调用转换。

其中，赋值转换、一般算术转换和函数调用转换是由系统自动隐含进行的，强制类型转换是由程序员使用强制运算符指定进行的显式类型转换。

一般算术转换

简称算术转换。

基本规则为：小类型向大类型转换。

> 双目运算符的两个操作数中，值域较窄的那个类型向值域较宽的那个类型转换。值域是类型所能表示的值的最大范围。被转换的两个操作数可为任意类型。

转换规则：

- `char, short` 转换为 `int`
- `unsigned char, unsigned short` 转换为 `unsigned int`
- `float` 转换为 `double`
- `long, unsigned long` 不进行类型转换，或者转换为 `long long` 和 `unsigned long long`。

强制类型转换

一般格式：`(类型名)操作数`。将操作数转换为由“s类型名”指定的类型，表达式的结果和类型与转换后的操作数的值和类型相同。

强制类型转换在效果上和赋值转换完全相同，它的转换方向都不受算术转换规则的约束，强制转换与赋值转换的区别是：

- 强制转换是显式方式，赋值转换是隐式方式。前者是人为的，后者是自动的。
- 强制转换的结果类型由强制运算符指定的类型名决定，赋值转换的结果类型由赋值运算符的左操作数的类型决定。

强制转换应用于 除赋值转换以外的 任何与算术转换方向不同的 类型转换。

注意：**类型强制符属于最高优先级。**

实现四舍五入运算

有以下定义：

```
float n = 3.458;
int m = 0;
```

想要将n的小数位通过四舍五入保留两位小数可通过以下代码：

```
m = n * 100 + 0.5;
    n * 100 = 345.8，还是浮点型
    + 0.5 = 346.3，还是浮点型
    这时候赋值给int类型的m，就会发生类型转换，
    m = 346
n = m / 100.0;
    m / 100.0 = 3.460000，这里会发生类型转换，结果为浮点型
```

以下代码不能实现四舍五入：

```
n = (n * 100 + 0.5) / 100.0;
因为整个式子一直都是浮点型，结果也是浮点型，所以最后结果是3.463000
```

所以四舍五入是用float转换为int类型来实现的。

第3章 顺序结构程序设计

著名的计算机科学家尼古拉斯·沃斯（Niklaus Wirth）提出：程序=算法+数据结构。

3.1 算法

3.1.1 算法的组成要素

为解决一个问题而采取的方法和步骤称为“算法”。

算法的特性

- 有穷性：一个算法应当包含有限的步骤，而不是能无限的步骤；同时一个算法应该在执行一定数量的步骤后结束，不能陷入死循环。
- 确定性：算法中的每一个步骤都应当是确定，的而不是含糊的、模棱两可的。
- 有0个或多个输入：输入是指算法执行时从外界获取必要信息。
- 有1个或多个输出：算法的目的是为了获得结果，没有输出的算法是没有意义的。
- 有效性：算法的每个步骤都应当能有效执行，并能得到确定的结果。

算法的组成元素

算法含有两大要素

- 操作：每个操作的确定不仅取决于问题的需求，还取决于他们来自哪个操作集，同时还与使用的工具系统有关。
- 结构控制：就是如何控制组成算法的各个操作的执行顺序。

结构化的程序设计方法要求一个程序只能由三种基本控制结构组成：

- 顺序结构
- 选择结构（分支结构）
- 循环结构（重复结构）

由这三种基本结构可以组成任何结构的算法。由基本结构组成的算法叫“结构化”算法。

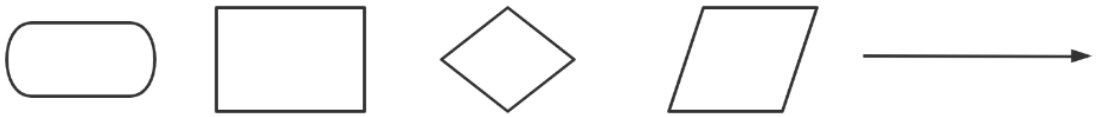
3.1.2 算法的表示方法

自然语言描述

自然语言就是人们日常使用的语言。用自然语言表示算法通俗易懂，但文字冗长，容易出现歧义。

流程图表示

用一些图框表示各种操作，用箭头表示算法流程。用图形表示算法直观形象，易于理解。



（a）起始框　（b）处理框　（c）判断框　（d）输入输出框　（e）流程线

N-S流程图表示

一种新的流程图，去掉了带箭头的流程线，每种结构用一个矩形框表示。

优点：

- 更加直观、形象，易于理解。
- 比传统的流程图紧凑易画。
- 废除流程线，整个算法结构是由各个基本结构按顺序组成的。

伪代码表示

伪代码是介于自然语言和计算机语言之间的文字和符号。

计算机语言表示

就是程序。

3.1.3 自顶向下、逐步细化的过程

包括：

- 将一个复杂问题的解法分解和细化成由若干个模块组成的层次结构。
- 将一个模块的功能逐步分解细化为一系列的处理步骤，直到某种程序设计语言的语句和某种机器指令。

优点：

- 符合人们解决复杂问题的普遍规律，可以显著提高设计的效率。
- 设计出的程序具有清晰的层次结构，容易阅读和理解。

3.2 C语句概述

语句是构成程序的基本成分。C语句可以分为三大类：**控制语句、表达式语句和复合语句。**

语句和表达式的区别：

- 表达式：是由运算符和运算对象组成的，每个表达式都存在一个值。比如以下都是表达式：

```
2
-3
3 + 2
a = 4 * 4
```

- 语句：语句一般由分号`;`结尾，大多数语句就是表达式后加上分号，但C语言存在空语句。

3.2.1 控制语句

C语言有9中控制语句：

- `if-else(条件语句)`
- `for(循环语句)`
- `whiel(循环语句)`
- `do-while(循环语句)`
- `continue(结束本次循环语句)`
- `break(终止switch或循环语句)`
- `switch(多分枝选择结构)`
- `goto(转向语句)`
- `return(从函数返回语句)`

3.2.2 表达式语句

表达式语句是在表达式最后加上一个`;`所组成的语句。C程序中大多数语句是表达式语句（包括函数调用语句）。

表达式语句的一般形式：`表达式;`。

表达式语句常见的形式有：赋值语句、函数调用语句和空语句。

赋值语句：由赋值表达式加上一个分号构成。

先计算赋值运算符右边的子表达式的值，然后将此值赋给赋值运算符左边的变量。

函数调用语句：由函数调用表达式加一个分号构成函数调用语句。

空语句：只有一个分号的语句。

3.2.3 复合语句

用“{”“}”把一些语句（语句序列，表示一些列工作）括起来成为复合语句，又称语句块，分程序。

一般情况凡是允许出现语句的地方都允许使用复合语句。

3.3 数据的输入和输出

C语言本身不提供输入输出语句。输入输出操作是由标准输入输出函数来完成的。

3.3.1 数据的输出函数

格式输出函数`printf`：使用格式控制符指定输出数据的类型。

一般形式为：`printf("格式控制参数", 输出项1, 输出项2, ...);`，格式控制参数以字符串的形式描述，也称“格式控制字符串”。

原样输出格式

`printf("要输出的字符串");`。

输出变量的值

按照格式控制符输出：`printf("格式控制字符串", 输出列表);`。

- 当输出列表中的参数大于1个时，用逗号分隔。
- 格式控制符必须和输出列表相对应，格式控制符的个数确定后续输出参数的个数。

格式控制字符串中有两类字符：

- 非格式字符：普通字符，一律原样输出
- 格式说明符：由`%`和转换字符组成，将输出的数据转为指定的格式输出。

格式说明符总是由`%`字符开始，以转换字符结束。完整格式为：

```
[开始符][对齐字符][填充形式][宽度指示符][精度指示符][长度修正符][指出输出类型]
%        +-        0         m          n            h/l       格式字符
```

格式说明符+数字

有以下程序：

```c
int main(){
    int a = 10;
    printf("%d"+1, a);
    return 0;
}
```

最开始以为是编译错误，结果发现程序正常运行，只不过输出结果不是a的值，而是`d`。

这是因为`"%d"`本质上是一个字符数组，对他进行`+1`呢就是将数组名往后移一位，所以实际上传给`printf()`的字符串是`"d"`，所以就只会输出一个d。

如果格式说明符字符串后面加的整数大于了格式说明符的长度，那么会是一个随机字符。

printf详情见附录

%d 格式字符

- `%d`：整数的实际位数输出一个整数
- `%md`：在m列的位置上，以数据右对齐的方式输出一个整数。
  - m大于整数的位数时，多余位数以空格留在数据前面。
  - m小于整数的位数时，m不起作用，系统正确输出该整数。
- `%-md`：在m列的位置上，以数据左对齐的方式输出一个整数。
  - m大于整数的位数时，多余位数以空格留在数据后面。
  - 规则与上述一样
- `%0md`：在m列的位置上，以数据左对齐的方式输出一个整数。

    - m大于整数的位数时，多余位数在数据前面补0。
    - m小于整数的位数时，m不起作用，系统正确输出该整数。
- `%o %x %u`与`%d`一样。

注意：没有`%-0md`的形式，当出现这种形式时，`0`会被`-`忽略，相当于`%-md`。比如：

```
int main(){
    long y = -43456;
    printf("y=%-8lda\n", y);
    printf("y=%-08lda\n", y);  // [Warning] '0' flag ignored with '-' flag in gnu_printf format [-Wformat=]
    printf("y=%08ld\n", y);
    return 0;
}
```

以上程序的输出结果为：

```
y=-43456  a
y=-43456  a
y=-0043456
```

%s 格式字符

- `%s`：原样输出。
- `%ms`：按给定的宽度m右对齐输出
    - m大于字符串的长度，多余位数在字符串左边补空格。
    - m小于字符串的长度，输出全部字符串。
- `%-ms`：按给定的宽度m左对齐输出
    - m大于字符串的长度，多余位数在字符串右边补空格。
    - m小于字符串的长度，输出全部字符串。
- `%m.ns`：在m列的位置上输出字符串的前n个字符。
    - `m>n`：右对齐输出，左侧用空格补齐。
    - `m<n`：输出n个字符。
    - n大于字符串的长度，输出整个字符串。
- `%-m.ns`：在m列的位置上输出字符串的前n个字符。
    - `m>n`：左对齐输出，右侧用空格补齐。
    - `m<n`：输出n个字符。
    - n大于字符串的长度，输出整个字符串。

%f 格式字符

- `%f`：按系统默认的宽度输出，整数部分全部输出，并输出六位小数。
    - 单精度数有7位的有效数字，双精度数有16位的有效数字。
    - 这里的有效数字表示准确的数字，7位或16位后的数字不一定是准确的。
- `%m.nf`：在m列的位置上输出一个实数，保留n位小数，系统自动对数据进行四舍五入处理。
    - m大于实数位数时，多余的位数以空格留在数据左侧。
    - m小于实数位数时，m不起作用，系统正确输出该整数。
- `%-m.nf`：在m列的位置上输出一个实数，保留n位小数，系统自动对数据进行四舍五入处理。
    - m大于实数位数时，多余的位数以空格留在数据右侧。
    - m小于实数位数时，m不起作用，系统正确输出该整数。

%e 格式字符

- `%e`：按规范的指数形式输出，即小数点前1位非零数字，小数点后6位小数，指数部分占5位，加上小数点一共13位。
  - 如：`1 . 23456 e+007`
- `%m.ne`：按指定的宽度m输出，包括n位小数和5位指数。
  - m大于数据位数，在数据左侧补空格。
  - m小于数据位数，按实际输出。
- `%-m.ne`：按指定的宽度m输出，包括n位小数和5位指数。
  - m大于数据位数，在数据右侧补空格。
  - m小于数据位数，按实际输出。

%g 格式字符

以实数形式输出，能根据数值的大小自动选用`%f`格式或`%e`格式，以使输出宽度较小，且不输出无意义的零。

总结:

- 使用`m`时，如果输出的字段长度小于m，结果会用前导空格填充；否则，照常输出。
- 使用`-`时，左对齐输出，右边填充空格。
- 使用`%f`并且使用`m`时，输出会自动四舍五入。

3.3.2 刷新输出

`printf()`语句将输出传递给一个被成为缓冲区的中介存储区域。缓冲区中的内容再不断地被传递给屏幕。

标准C规定以下几种情况会将缓冲区内容传递给屏幕：

- 缓冲区满的时候
- 遇到换行符`\n`的时候
- 需要输出的时候

将缓冲区的内容传递给屏幕或者文件被称为 **刷新缓冲区**。

3.3.3 scanf函数

`scanf()`函数用于输入数据，具体地说它是按格式参数的要求，从终端上把数据传送到地址参数所指定的内存空间中。

一般形式为：`scanf("格式控制参数", 地址1, 地址2...);`。

scanf详情见附录

地址参数

C语言允许程序员间接地使用内存地址，这个地址是通过对变量名“求地址”运算得到的。求地址的运算符为`&`，得到的地址是一种符号地址。

比如：

```
int a;
float b;
```

则`&a、&b`为两个符号地址。`&a`给出的是变量a四字节空间的首地址，`&b`给出的是变量b四字节空间的首地址。这种符号地址在编译时才会被算成实际地址。

格式控制参数

`scanf`函数的格式参数有两种成分：

- 格式说明项
- 输入分隔符

格式说明项

基本组成如下：

```
[开始符][赋值抑制符][宽度指示符][长度修正符][指出输出类型]
%       *          m           h/l          格式字符（用法与printf一样）
```

注意：不能在输入的时候指定浮点数的精度。

```
scanf("%.2f", &a);
```

也就是说以上代码是不合法的（在VS2022中执行出错，编译没问题）。

以下是指定列宽：

```
scanf("%2f", &a);
```

指定列宽在输入、输出中都是合法的。

数据输入流分隔

`scanf`函数是从输入数据流中接收到非空的字符，再转换成格式项描述的格式，传送到与格式项对应的地址中去。

系统如何知道哪几个字符算作一个数据项？有以下情况：

- 根据格式字符的含义从输入流中取得数据，当输入流中数据类型与格式字符要求不符时，就认为这一数据项结束。
- 根据格式项中指定的域宽分隔出数据项
- 用分隔符。空格、跳格符（'\t'）、换行符（'\n'）都是数据分隔符。
- C语言允许在输入数据时使用用户自己指定的字符（必须是非格式字符）来分隔数据。

注意：scanf函数可用`%c`来输入空格、换行、制表符等。比如：

```
int main(){
   int a, b;
   char c, d;
   scanf("%d%c%d%c", &a, &c, &b, &d);
   printf("%d %c %d %c\n", a, c, b, d);
   return 0;
}
```

以上程序想要`a=10, b=A, c=20, d=B`，那么正确的输入方式应该是：`10A 20B<enter>`或者`10A20B<enter>`。

分析：

```
10A 20B<enter> ：按%d读走10，按%c读走A，此时理应读一个整数，但碰到了空格，当作分隔符跳过，读走20，再按%c读走B
10A20B<enter>  ：按%d读走10，按%c读走A，按%d读走20，按%c读走B
10 A 20 B<enter>  ：按%d读走10，此时理应读一个字符，碰到空格，把空格当成一个字符读走，然后按%d理应读一个整数，碰到了字符A，发现格式项与输入域不匹配，scanf函数结束，此时c和d的值不确定。
```

所以，当`分隔符：空格、换行、制表符`碰到`%c`格式符，分隔作用不在，只被当作普通字符，除非`scanf`中给定的分隔符就是这些字符。

抑制字符 *

作用是在按格式说明读入数据后不送给任何变量，即“虚读”。

比如：

```
scanf("%3d%*4d%f",&i,&f);
输入：12345678765.43<enter>
则：
    123给了i
    4567丢掉
    8765.43给了f
```

在利用已有的一批数据时，若有一两个数据不需要，可以使用此法“跳过”这些无用数据。

scanf的停止与返回

遇到以下情况结束：

- 格式参数中的格式项用完——正常结束。
- 发生格式项与输入域不匹配时——非正常结束

`scanf`是一个函数，也有一个返回值，这个返回值就是成功匹配的项数。

比如：

```c
#include<stdio.h>

int main(){
    int a, b , c;
    printf("%d\n", scanf("%3d-%2d-%4d", &a, &b, &c));
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}
/*
输入：123-45-6789<enter>
输出：
    3
    a = 123, b = 45, c = 6789

输入：12-345-6789
输出：
    2
    a = 12, b = 34, c = 0
*/
```

以上代码的第二种情况中，第一个参数`%3d`读取的数据是`12-`，第3个数据不是数字，所以截止读取，将已正常读取的数据`12`赋给了a；然后第二个参数`%2d`接着向下读取，读取到了`34`，但是34读完后应该有个`-`分隔符，但是没有，而是一个5，所以格式项域输入域不匹配，结束`scanf`，只成功匹配到了两个项，返回2。

scanf函数与输入缓冲区

在输入数据时，实际上并不是输入完一个数据项就被读取送给一个变量，而是在键入一行字符并按回车键之后才被输入，这一行字符先放在一个缓冲区中，然后按scanf函数格式说明的要求从缓冲区中读数据。如果输入的数据多于一个scanf函数所要求的个数，余下的数据可以为下一个scanf函数接着使用。

3.4 getchar函数与putchar函数

`getchar()`和`putchar()`都包含在`stdio.h`中。

3.4.1 字符输出函数putchar()

一般调用格式：`putchar(ch);`

该函数的功能是：将变量`ch`的值输出到终端（显示器）设备上。其中，`ch`为**字符型**变量或者**整型**变量。

3.4.2 字符输入函数getchar()

一般调用格式：`getchar();`

该函数为无参函数，功能是：从终端（或指定输入设备）获取一个输入字符。

在执行`getchar`函数时，虽然是读入一个字符，但并不是从键盘读入一个字符，该字符就被读入送给一个字符变量，而是等到输入完一行按回车键之后，才将该行的字符输入缓冲区，然后`getchar`函数从缓冲区取一个字符给一个字符变量。

可以有这样的形式：`putchar(getchar());`.即读入一个字符，然后将它输出到终端。

第4章 选择结构程序设计

4.1 if语句

用来判定是否满足指定的条件，并根据判定结果执行相应的操作。

4.1.1 if语句的形式

单if语句

一般形式为：`if(表达式)　　语句组`。

执行过程：系统首先对表达式求解，当结果为“真”（非0）时，则执行指定的语句组；否则跳过指定语句组，接着执行if语句的下一句执行。

if-else语句

一般形式为：

```
if(表达式)
    语句组1
else
    语句组2
```

执行过程：系统首先对表达式求解，当结果为“真”（非0）时，则执行语句组1；当结果为“假”（0）时，执行语句组2。

if-else if-else语句

一般形式为：

```
if(表达式1)
    语句组1
else if(表达式2)
    语句组2
else if(表达式3)
    语句组2
    ...
else if(表达式n-1)
    语句组n-1
else
    语句组n
```

执行过程：系统首先对表达式1求解，当结果为“真”（非0）时，则执行语句组1，然后跳出该选择结构；否则求解表达式2的值，依次类推。如果最后所有的表达式都为“假”（0），则执行最后一个else部分的语句组n，从而结束整个if语句。

注意：要注意条件的表示形式和先后顺序。

if语句需要注意的问题

1、if语句中表达式的值只能为“真”（非0）或“假”（0）。

if语句中的表达式通常是 逻辑表达式或关系表达式，也可以是其他类型的表达式，如 赋值表达式，也可以是一个变量。

2、在if语句中，表达式必须用括号括起来。

3、在if语句的3种形式中，语句组可以是单个语句，也可以是多个语句。

如果是单个语句，可以省略if或else后面的`{}`；如果是多个语句，则必须用`{}`括起来。

4.1.2 if语句的嵌套

在if语句中又包含一个或多个if语句称为if语句的嵌套。

if语句可以内嵌在if子句中，也可以内嵌在else子句中。

注意：else总是与它上面最近的if配对。

如果if和else的数目不一样，可以加`{}`来确定配对关系。

4.1.3 良好结构的程序

一个程序应该层次分明，具有必要的注释，才便于理解和查找错误。

4.2 switch语句

C语言还提供了另一种用于多分支选择的switch语句。

一般形式为：

```
switch(表达式){
    case 常量表达式1: 语句组1;
    case 常量表达式1: 语句组2;
    ...
    case 常量表达式n: 语句组n;
    [default: 语句组n+1;]
}
```

注意：表达式只能为整数类型、字符类型或者枚举类型。整数型即除了`float和double`：`short\int\long\long long`。

各个常量表达式代表switch后面的表达式的各个不同的取值。

执行过程：系统首先求解表达式的值，然后依次与各个case后面给出的常量表达式的值相比较。当表达式的值与某个case后的常量表达式的值相等时，就从此处开始执行该case后面的语句，而不再进行判断；如果所有的case后面的常量表达式的值都和表达式的值不匹配，系统就执行default后面的语句，如果程序省略了default语句那么将不做任何处理，接着执行该选择结构下面的语句。

在switch语句中，“case 常量表达式：”只相当于一个语句标号。当表达式的值和某标号相等，则转向该标号执行，但不能在执行完该标号的语句后自动跳出整个switch语句，所以会继续执行所有后面case语句。

C语言还有一种break语句，用于跳出switch语句。break语句只有关键字break。

注意：

- <表达式> 只能是整型、字符型、枚举类型表达式。
- case后面必须是常量表达式，且各常量表达式不能相同。
- case后面可以有多个语句，可以不用`{}`括起来。
- case子句和default子句的顺序可以任意。
- 多个case可以公用一组语句。
- switch可以内嵌在某个case语句中。

另外，switch语句中的case子句是从上而下执行的，不会从下往上跳着执行。比如：

```
int main(){
    int a = 3;
    switch(a){
        default:printf("0");
        case 1:printf("1");break;
        case 2:printf("2");break;
        case 3:printf("3");
    }
    return 0;
}
```

以上程序`a=3`时，会只输出3，不会再执行default语句；如果`a=5`时，会输出01。

if-else if-else 语句用于多条件并列测试，从中取一的情况；switch语句用于单条件测试，从其多种结果中取一的情形。

4.3 程序举例

【例1】求一元二次方程$ax^2+bx+c=0$的根。

分析：一元二次方程的根有以下情况：

（1）、当$a=0,b=0$时：方程无解。

（2）、当$a=0,b\neq 0$时：只有一个实根：$-c/b$。

（3）、当$a\neq 0$时：根为：$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$。

- 当$b^2-4ac\geqslant 0$时，有两个实根
  - 大于0时，两个不同的实数根
  - 等于0时，两个相同的实数根
- 当$b^2-4ac<0$时，有两个虚根。
  - $\frac{-b\pm\sqrt{-(b^2-4ac)i}}{2a}$。i是虚数单位。

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

int main(){
    float a, b, c, x1, c2, x3, x4;
    scanf("%d%d%d", &a, &b, &c);
    if(fabs(a) <= 1e-6 && fabs(b) <= 1e-6)
        printf("方程无解");
    else if(fabs(a) <= 1e-6 && fabs(b) > 1e-6)
        printf("方程有一个跟：%d", -c/b);
    else{
        d = b * b - 4 * a * c;
        if(fabs(b) >= 1e-6){
            x1 = (-b + sqrt(d)) / (2 * a);
            x2 = (-b - sqrt(d)) / (2 * a);
            printf("方程有两实根：%d, %d", x1, x2);
        }else{
            realpart = -b / (2 * a);
            imagpart = sqrt(-d) / (2 * a);
            printf("方程有两虚根：%f + %fi, %f - %fi", realpart, imagpart);
        }
    }
    return 0;
}
```

第5章 循环结构程序设计

循环结构又称重复结构，是按照一定的条件重复执行某段语句的程序控制结构。

C语言提供了如下基本的循环结构：

- goto语句和if语句构成的循环结构
- `while`语句构成的循环结构
- `do-while`语句构成的循环结构
- `for`语句构成的循环结构

5.1 while语句

`while`语句用来构造“当型”循环，多用于解决循环次数事先不确定的问题。

一般形式：

```
while(表达式)
{
    循环体
}
```

功能：先判断表达式值得真假，若为真（非0）时，就执行循环体；否则，不进入循环结构或者退出循环结构。

注意：

- `while`语句的表达式一般是关系表达式或逻辑表达式，也可以是实数值表达式或字符表达式，只要其值为真（非0）即可继续执行循环体。
- 循环体语句可以为任意类型。一个以上语句需要有`{}`括起来，没有`{}`的话，`while`的范围直到下面第一个语句。
- 循环体中应有使循环趋于结束的语句，以避免死循环。
- 循环可以嵌套。

5.2 do-while语句

`do-while`语句用来构造“直到型”循环结构，也多用于循环次数事先不确定的问题。

一般形式：

```
do{
    循环体
}while(表达式);
```

功能：先执行一次循环体，再判断表达式的真假。表达式为真（非0）则继续执行循环体，否则退出循环结构。

注意：

- `while(表达式)`后面的`;`不能少。
- `do-while`语句中的循环体至少要被执行一次。

`do-while`和`while`：当`while`后面的表达式第一次的值为“真”时，两种循环的结果相同，否则不同。

5.3 for语句

`for`语句时C语言提供的一种功能比前面两种语句更强的循环语句。它不仅可用于循环次数确定的情况，还可以用于循环次数不确定而只给出了循环结束的条件。

一般形式（3个表达式之间必须用;隔开）：

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

执行流程：

1. 先执行表达式1
2. 判断表达式2的值，为真（非0）则执行循环体，否则结束循环
3. 执行完循环体后，执行表达式3
4. 进行第2步
5. 循环结束

也可以这样理解for的形式

```
for(循环变量初始化; 循环条件; 改变循环变量)
{
    循环体
}
```

注意：`for`的三个表达式都可以省略，但是;不能省略。表达式省略后，应该在循环前给出循环变量的初始化，循环内给出结束循环条件和改变循环变量。

循环体可以是空语句，产生延时效果。如：`for(i = 0, i < 10000; i++);`

注意：在C99标准中，`for`语句的第一个表达式可以替换为一个声明。`for(int i = 0; i < 10; i++)`或`for(int i = 0, j = 0; i < 10; i++)`。

5.4 循环的嵌套

多重循环使用要点：

- 注意给循环变量赋初值：只需执行一次的赋初值操作应该放在循环开始执行之前。
- 内、外循环变量不应同名。
- 应正确书写内、外循环的循环体：内循环中执行的所有语句必须使用`{}`括起来组成符合语句作为内循环体；外循环的语句应放在内循环体之外、外循环体之中。
- 不应在循环中的执行的操作应放在最外层循环之前或最外层循环结束之后。

5.5 break语句和continue语句

5.5.1 break语句

`break`语句可用于`switch`语句也可用于循环。在循环结构中执行到`break`语句时，循环将无条件终止，程序跳出循环结构。

注意：`break`只能跳出一层循环。

5.5.2 continue语句

`continue`语句的作用是终止本次循环，`continue`语句后面的语句不执行而进入下一次循环。

- 在`while`和`do-while`中，`continue`语句使程序直接转向条件测试处。
- 在`for`语句中，`continue`语句使程序转向循环变量的增值表达式，然后再判断条件表达式。
-

5.6 综合实例

5.6.1 列举算法

所谓列举算法，是根据提出的问题，列举所有可能的情况，并根据条件检验哪些是需要的，哪些是不需要的。

设计列举算法的关键是根据问题的性质确定判断的条件，从而列举所有条件进行判断。

【例题】某单位要在A、B、C、D、E、F六个人中选派若干个人去执行一项任务，选人的条件如下：

- 若C不去，则B也不去
- C和D两人中去一个
- D 和E要么都去，要么都不去
- A、B、F三个人中要去两个
- C和F不能一起去
- E和F两个人中至少去一个

```c
#include<stdio.h>
int main(){
    int a, b, c, d, e, f;
    for(a=0; a<=1; a++){
        for(b=0; b<=1; b++){
            for(c=0; c<=1; c++){
                for(d=0; d<=1; d++){
                    for(e=0; e<=1; e++){
                        for(f=0; f<=1; f++){
                            if((b+c==0)||c==1)
                                && (c+d==1)
                                && (d+e==0||d+e==2)
                                && (a+b+f==2)
                                && (c+f!=2)
                                && (e+f>1)) {
                                printf("A : %s\n", a ? "去" : "不去");
                                printf("B : %s\n", b ? "去" : "不去");
                                printf("C : %s\n", c ? "去" : "不去");
                                printf("D : %s\n", d ? "去" : "不去");
                                printf("E : %s\n", e ? "去" : "不去");
                                printf("F : %s\n", f ? "去" : "不去");
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}
```

以上程序的输出结果是：

```
A : 去
B : 不去
C : 不去
D : 去
E : 去
F : 去
```

5.6.2 试探算法

列举算法一般是直到列举量，其列举的情况总是有限的。而在有的问题中，可能其列举量事先并不知道，只能从初始情况开始，往后逐步试探，直到满足给定的条件为止，这就是逐步试探法，简称试探法。

【例题】某幼儿园按如下方法依次给A、B、C、D、E五个小孩发苹果。

- 将全部苹果的一半加上二分之一个苹果发给第一个小孩
- 将剩下苹果的三分之一再加三分之一个苹果放给第二个小孩
- 将剩下苹果的四分之一再加四分之一个苹果发给第四个小孩
- 将最后剩下的11个苹果发给第五个小孩

每个小孩得到的苹果数均为整数。

【分析】假设总苹果数为x，设五个小孩的苹果数分别为a, b, c, d, e，可按以下公式依次计算：

- $a = (x + 1)/2$
- $b = (x - a + 1)/3$
- $c = (x - a - b + 1)/4$
- $d = (x - a - b - c + 1)/5$
- $e = 11$

设当前试探点苹果数为n，通过前面的分析可知，n应该满足以下条件：

- 第k个小孩得到全部苹果的(k+1)分之一再加(k+1)分之一个苹果，即$(n + 1)/(k + 1)$个苹果，且该数为整数。
- 发完第k个小孩的苹果后，剩下的苹果数为$n - (n + 1)/(k + 1)$，这就是下一个n的值.

```c
#include <stdio.h>

int main(){
    int a, b, c, d, e, t;
    int n = 11;
    while(1){
        t = n;
        int ok = 0;
        for(int k = 1; k<= 4; k++){
            if((n+1) % (k+1) == 0){
                n = n - (n+1) / (k+1);
                continue;
            }
            ok = 1;
        }
        if(ok == 0)
            break;
```

```
        n = t + 1;
    }
    a = (t+1) / 2;
    b = (t-a+1) / 3;
    c = (t-a-b+1) / 4;
    d = (t-a-b-c+1) / 5;
    e = 11;
    printf("%d\n", t);
    printf("%d %d %d %d %d\n", a, b, c, d, e);
    return 0;
}
```

以上程序的输出结果是：

```
59
30 10 5 3 11
```

【例题】试求2000~2050年间的闰年。

判断闰年：该年份能被4整除但是不能被100整除，或者该年份能被400整除。

```
int i;
for(i = 2000; i <= 2050; i++){
    if(i % 4 == 0 && i % 100 != 0 || i % 400 == 0)
        printf("%5d\n", i);
}
```

第6章 函数

函数简单来说就是一连串的语句，这些语句被组合在一起，并被取了一个名字，它可以事先某些基本的功能。函数是C语言的基本组成单位

6.1 函数概述

6.1.1 模块化程序设计方法

如果软件划分为可独立命名和编程的部件，则每个部件称为一个模块。模块化就是把系统划分为若干个模块，每个模块完成一个子功能，把这些模块集中起来组成一个整体，从而完成指定的功能，满足问题的要求。

函数是C语言程序的基本模块。

6.1.2 函数的概念

C语言中，函数分为两种：

- 标准库函数。这种函数用户不必定义，可直接使用。
- 用户自定义函数。用以解决用户的专门问题，由用户自己编写。

注意：

- 一个完整的C程序可以由若干个函数组成，其中必须有且仅有一个主函数`main()`。

- 一个完成的C程序中的所有函数可以放在一个文件中，也可以放在多个文件中。如果一个C程序中的多个函数分别放在多个不同的文件中，在调用函数中用`#include`语句将各个被调用函数所在的文件包含进来。
- C语言中的函数没有从属关系，各函数之间相互独立，可以相互调用，但不能嵌套定义。

6.2 函数的定义与声明

函数定义：定义函数的功能。未经定义的函数是不可用的。库函数由C编译系统提供，不需要定义即可使用；用户自定义函数必须先定义再使用。

函数调用：执行一个函数。调用函数时，有参数先传参数，程序由主调函数跳转到被调函数的第一条语句开始执行，执行完被调函数或遇到`return`语句就回到主调函数继续向下执行。

函数声明：通知编译系统该函数已经定义过了。库函数不需要声明，只需要使用`#include`语句包含具有该库函数的头文件即可；用户自定义函数，如果函数定义的位置在函数调用之后，则前面必须要有声明。

6.2.1 函数定义

在C程序中，函数的定义可以放在任何位置，即可以放在主函数之前，也可以放在主函数之后。

一般形式：

```
[函数类型] 函数名 ([形参列表])      // 函数头
{                                  // 函数体
    [声明部分]
    [执行语句]
}
```

说明：一个函数（定义）由函数头（函数首部）和函数体两部分构成

- 函数头（首部）：说明函数类型、函数名称、函数参数。
  - 函数类型：函数的返回值类型，可以是基本类型，也可以是构造类型；如果省略则默认为int类型；如果不返回值，则定义为void型。
  - 函数名：给函数取的名字。命名规则与标识符相同。
  - 形参列表：形式参数列表，可以没有参数，但是`()`不能省略；形参列表说明形参的类型和名称，多个参数之间用`,`分隔。
- 函数体：函数头下面用`{}`括起来的部分，一般包括声明部分和执行部分。
  - 声明部分：定义本函数所使用的变量和进行有关声明。
  - 执行部分：程序段。

注意：函数不能单独运行，函数可以被其他函数调用也可以调用其他函数，但是不能调用主函数。

6.2.2 函数的参数和返回值

函数的参数分为形式参数和实际参数。

形式参数（形参）：函数定义使设定的参数，本质就是变量。

实际参数（实参）：调用函数时所使用的实际的参数。

形参和实参的功能是进行数据传递。只有在发生函数调用时，主调函数把实参的值传给被调函数的形参（实参给形参赋值），从而事先数据传递。在传递过程中，**应该保证实参与形参的类型一致、个数一致、顺序一致。**

C语言可以从函数返回值给调用函数。在函数内是通过`return`语句来实现的，使用`return`语句可以返回一个值或者不返回值（此时函数类型是void）。

`return`语句的格式为：

```
return [表达式];   // return 0; return;
或
return(表达式);    // return(0);
```

默认情况下，main的返回值类型为`int`型， main函数的返回值是个状态码，0表示main函数正常结束，不是0表示main函数异常终止。用`exit(0)`或`return 0`作用是一样的，都是向操作系统返回一个0。

函数的类型就是返回值的类型，`return`语句中表达式的类型应该与函数类型一致。如果不一致，以函数类型为准（赋值转化）。

如果函数没有返回值，函数类型应该说明为`void`（空类型）。

6.2.3 函数的声明

函数定义的位置可以在调用之前、调用之后或者其他源程序模块中。

- 函数定义位置在前，调用在后，不必声明。
- 函数定义位置在后，调用在前，或函数在其他源程序模块中，要在调用函数前进行声明。

声明一般形式：

```
函数类型 函数名 ([形式参数列表]);
```

声明的时候，形式参数列表可以不用写变量名，也可以写，如下：

```
void fun(int a, int b);
```

或者如下：

```
void fun(int, int);
```

以上两种格式都是正确的。

C语言的库函数就是位于其他模块的函数，C编译系统提供了相应的`.h`文件，里面许多都是函数声明，所以再使用库函数时，应当包含相应的头文件。

6.3 函数的调用

一个函数调用另一个函数称为函数调用，其调用者称为主调函数，被调用者称为被调函数。

6.3.1 函数的调用的一般形式

1、函数语句形式

C语言中的函数可以只进行某些操作而不返回值，这时的函数调用可作为一条独立的语句。

2、函数表达式形式

函数作为表达的一项，出现在表达式中，以函数返回值参与表达式的运算，这种方式要求函数必须有返回值。

3、函数实参形式

函数作为另一个函数调用的实际参数出现。这种情况是把函数的返回值作为实参进行传送，因此要求函数必须有返回值。

6.3.2 函数参数的传递方式

实参与形参的传递方式有两种：值传递和地址传递。

值传递：参数传递的是数据本身。C语言规定，数值只能由实参传递给形参，即传值是单向的，也就是说，形参的任何变化不会影响到实参。

6.3.3 函数的嵌套调用

C语言中函数定义都是互相平行、独立的，也就是说在定义函数时，一个函数不能包括另一个函数（不能嵌套定义函数），但是可以在调用一个函数的过程中又调用另一个函数（可以嵌套调用函数）。

6.3.4 函数的递归调用

一个函数在它的函数体内调用自身的过程，称为递归调用。表现为直接调用自己或间接调用自己，前者称为直接递归调用，后者称为间接递归调用。

递归调用包括两个阶段：

- 递推阶段：将原问题不断地分解为子问题，逐渐从未知的向已知的方向推测，最终达到已知的条件，即递归结束条件，这时递推结束。
- 回归阶段：从已知条件出发，按照“递推”的逆过程，逐一求值回归，最终达到“递推”的开始处，结束回归阶段，完成递归调用。

【例题】某幼儿园按如下方法依次给A、B、C、D、E五个小孩发苹果。

- 将全部苹果的一半加上二分之一个苹果发给第一个小孩
- 将剩下苹果的三分之一再加三分之一个苹果放给第二个小孩
- 将剩下苹果的四分之一再加四分之一个苹果发给第四个小孩
- 将最后剩下的11个苹果发给第五个小孩

每个小孩得到的苹果数均为整数。求原来共有多少个苹果。

```c
float fun(float n){
    if(n == 5)
        return 11;
    float a = fun(n+1) + 1 / (n+1);
    a = a * (n+1) / n;
    return a;
}
int main(){
    printf("%.2f\n", fun(1));
    return 0;
}
```

6.4 局部变量和全局变量

6.4.1 局部变量

局部变量也称为内部变量，是在函数内做定义声明的，其作用域仅限于函数内。

关于局部变量的作用域：

- 主函数定义的变量只能在主函数中使用，不能在其他函数中使用；主函数也不能使用其他函数内定义的变量。因为主函数也是函数，和其他函数是平行的关系。
- 形参变量是属于被调函数的局部变量，实参变量是属于主调函数的局部变量。
- 在不同的函数中允许使用相同的变量名。
- 在复合语句中也可以定义变量，其作用域只在复合语句中。

【例题】写出程序的执行结果

```c
int main(){
    int i = 2, j = 3, k;
    k = i + j;
    {
        int k = 8;
        printf("%d\n", k);
    }
    printf("%d\n", k);
    return 0;
}
```

输出结果为：

```
8
5
```

6.4.2 全局变量

全局变量也称外部变量，它是在函数外部定义的变量。它属于一个源程序文件，作用域是从定义变量的为止开始到本源文件结束。

如果全局变量在文件的开头的定义，则作用域是整个文件范围；如果不在开头定义，又想在定义点之前使用，需要用 `extern` 进行声明。

变量的声明和定义是不同的：

- 变量的定义即为变量分配存储单元；
- 变量的声明即说明变量的性质，并不分配存储空间。

全局变量的定义形式：

```
类型说明符 变量名，变量名.....
```

全局变量声明形式：

```
extern 说明符 变量名，变量名.....
```

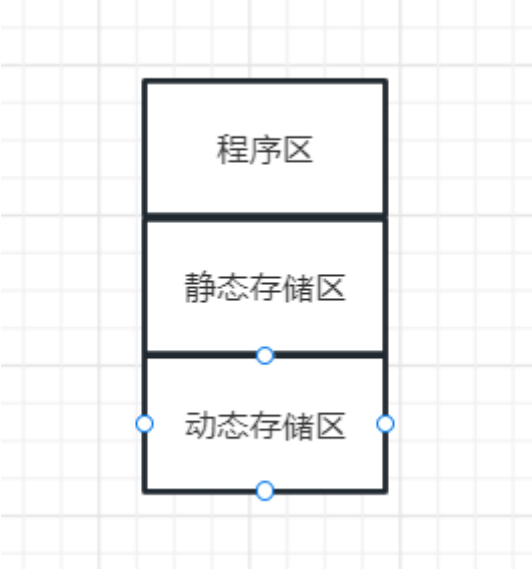
全局变量在定义时就已分配了内存单元，全局变量在定义时可以初始赋值，但不能声明时赋值，声明只表示在函数内要使用该全局变量。

在同一个源文件中，允许全局变量和局部变量同名。在局部变量的作用范围中，全局变量被“屏蔽”，即不起作用。

全局变量无论是否使用，程序执行时都占用固定的空间。

6.5 变量的存储属性

用户程序的存储分配：



程序区：用于存放程序；

静态存储区：在程序开始执行时就分配的固定存储单元，如全局变量；

动态存储区：在函数调用过程中进行动态分配的存储单元，如函数形参、自动变量。

变量的存储类型：

在C语言中，每个变量和函数都有两个属性：**操作属性**和**存储属性。**数据类型是变量的操作属性；变量的存储属性包括变量的存储器类型、变量的生存周期和变量的作用域。

从变量值存在的时间（即生存期）来分，变量分为**永久存储**和**动态存储。**

采用永久存储的变量在编译时分配内存单元，程序执行开始后这种变量即被创建，程序执行结束后才被撤销。这种变量的生存周期为程序执行的整个过程，在该过程中占有固定的存储空间。

而采用动态存储的变量只在程序执行的某一段时间内存在。

计算机的存储器分为内存（主存）、外存（辅存），除此之外，CPU中还有一块临时存储器：寄存器。

在C语言中，用"存储属性"来表示以上三个方面的属性，并且把它们分为四类：register、auto、static、extern。

存储属性是变量的重要属性之一。在定义变量时，除了指定数据类型，还可以指定存储属性。

例如：

```
register int a;
```

表示a是一个整型变量，存储在寄存器中。`int a`表示存储在主存中。

除了`register`类型，其他三种类型都存储在主存中。

6.5.1 自动变量（auto）

自动变量为局部变量，用说明符`auto`进行说明，是C语言程序中使用最广泛的一种变量。一般形式为：

```
[auto] 数据类型 变量名 [= 初值表达式],...;
```

说明：

- 自动变量是局部变量，自动变量的作用与仅限于定义该变量的个体内。
- 自动变量属于动态存储，只有在使用它，即定义该变量的函数被调用时，才给他分配存储单元，开始他的生存期。
- 由于自动变量的作用域和生存期都局限于定义它的个体内（函数、复合语句），因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量也可与该函数内部的复合语句中定义的自动变量同名。
- 在对自动变量赋值之前，它的值是不确定的。
- 函数的形参也是一种自动变量，但是在说明时不加`auto`。

6.5.2 寄存器变量（register）

寄存器变量具有与自动变量完全相同的性质。为了提高效率，C语言允许将局部变量的值放在CPU的寄存器中，这种变量叫寄存器变量，用关键字`register`声明。一般形式为：

```
register 数据类型 变量名 [= 初值表达式],...;
```

说明：

- 只有局部自动变量和形式参数可以作为寄存器变量。
- 一个计算机系统中的寄存器数目有限，不能定义多个寄存器变量。

6.5.3 静态变量（static）

有时希望函数中的局部变量值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为“静态局部变量”，用关键字`static`进行声明。一般形式为：

```
static 数据类型 变量名 [= 初始化常量表达式],...;
```

说明：

- 静态变量的存储空间在程序的整个运行期间是固定的。一个变量被指定为静态的，在编译时就为其分配空间，程序一开始执行便被建立，直到该程序执行结束都是存在的。
- 静态变量的初始化是在编译时进行的。在定义时只能使用常量或常量表达式进行显示初始化，未显示初始化时编译时将它们初始化为0（对int型）或0.0（对float型）。
- 在函数多次被调用的过程中，静态局部变量的值具有可继承性。

严格的讲，初始化指的是在程序运行前由编译器给变量的初始值。由于自动变量是在程序运行过程中被创建的，因而没有初始化的问题，只有静态变量和外部变量是在编译时被创建的，才有初始化问题。所以对自动变量称为“赋初值”，而对静态变量和外部变量称为“初始化”，以示区别。然而在不太严格的情况下，人们也通通把声明语句给定初值称为“初始化”。

6.5.4 外部变量

外部变量（即全局变量）是在函数的外部定义的，其作用域为从变量定义处开始，到本程序文件末尾。如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件的末尾。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字`extern`对该变量作“外部变量声明”，表示该变量是一个已经定义的外部变量。有了此声明就可以从“声明”处起合法地使用该外部变量。

声明一般形式为：

```
extern 说明符 变量名，变量名......
```

说明：

- 可以将外部变量的作用域扩充到其他文件。这时，在需要用到这些外部变量的文件中，对变量用`extern`做声明即可。
- 可以限定本文件的外部变量只在本文件中使用。可以在此外部变量前加一个`static`使其有限局部化，称为静态外部变量。

使用静态外部变量的好处是：当多人分别编写一个程序的不同文件时，可以按照需要命名变量，而不必考虑是否与其他文件中的变量同名，以保证文件的独立性。

四类存储属性的性质

存储属性	register	auto	static	extern
存储位置	寄存器	主存		
生存周期	动态存储		永久存储	
作用域	局部		局部或全局	全局

6.6 编译预处理

编译预处理是在编译前对源程序进行的一些预处理。预处理由编译系统中的预处理程序按源程序中的预处理命令进行。

C语言的预处理命令均以“#”打头，末尾不加分号，以区别于C语句。

它们可以出现在程序中的任何位置，其作用域是自出现点到所在源程序的末尾。

6.6.1 宏定义

在C语言源程序中，允许用一个标识符来表示一个字符串，称为“宏”。被定义为宏的标识符称为“宏名”。在编译预处理时，对程序中所有出现的“宏名”都用宏定义中的字符串去代换，这称为“宏代换”或“宏展开”。

宏定义是由源程序中的宏定义命令完成的，宏代换是由预处理程序自动完成的。

在C语言中宏分为有参数和无参数两种。

无参宏定义

无参宏的宏名后不带参数。定义的一般形式为：

```
#define 宏名 字符串
```

作为宏名的标识符习惯上用有意义且易理解的大写字母来表示；字符串可以是常数表达式或格式串等。

宏定义一般写在文件开头函数体的外面，有效范围是从定义宏命令之后到遇到终止宏定义命令`#undef`为止，否则其作用域将一直到源文件结束。

说明：

- 宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，这只是一种简单的代换，字符串中可以含任何字符，可以是常数，也可以是表达式，预处理程序对它不做任何检查，如有错误，只能在编译已被宏展开后的源程序时发现。
- 如果在一行中写不下整个宏定义，需要用两行或更多行来书写时，只需在每行的最后一个字符的后面加上反斜杠“\”，并在下一行的最开始接着写即可。
- 宏名在源程序中若用引号括起来，则预处理程序不对其作宏代换。
- 宏定义允许嵌套，在宏定义的字符串中，可以使用已经定义的宏名，在宏展开时由预处理程序层层代换。

带参宏定义

在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数，对带参数的宏在调用中不仅要进行宏展开，而且要用实参去代换形参。

定义一般形式为：

```
#define 宏名(形参列表) 字符串
```

调用一般形式为：

```
宏名(实参列表);
```

说明：

- 带参宏定义中，宏名和形参列表之间不能有空格出现。
- 带参宏定义中，形式参数不分配内存单元，因此不必做类型定义。

- 在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。
- 宏代换中对实参表达式不做计算，直接原样代换。
- 在宏定义中字符串内的形参通常要用括号括起来，以避免出错。

注意：已定义为常量的元素不能作形参使用。

以下代码是错误的：

```
#define a 1
int main(){
    void fun(int a);  // 这里不能用a
    int x = 10;
    fun(x);
    return 0;
}
void fun(int a){  // 这里也不能用a
    printf("%d\n", a);
}
```

改成：

```
#define a 1
int main(){
    void fun(int b);
    int x = 10;
    fun(x);
    return 0;
}
void fun(int b){
    printf("%d\n", a);
}
```

6.6.2 文件包含

文件包含是C语言预处理程序的另一个重要的功能。

一般形式为：

```
#include <文件名>
#include "文件名"
```

文件包含命令的功能是：把指定的文件插入该命令行位置，取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。

文件包含命令中的文件名可以用双引号括起来也可以用尖括号括起来。

区别是：使用`< >`表示在包含文件目录中查找（包含目录是由用户在设置环境时设置的），而不在源文件目录中查找；使用`" "`则表示首先在当前的源文件目录中查找，若未找到才到包含目录中查找。

说明：

- 一个`include`命令只能指定一个被包含文件。
- 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

6.6.3 条件编译

预处理程序提供了条件编译的功能。可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件。

第一种形式：

```
#ifdef 标识符
    程序段1
#else
    程序段2
#endif

或者可以没有#else
#ifdef 标识符
1   程序段
#endif
```

其中，标识符是指以用宏命令#define定义的宏名，程序段可以是编译预处理命令行，也可以是C语言组。

功能：如果标识符已被#define命令定义过则编译程序段1；否则编译程序段2。

示例：以下程序输出1

```
#define A 1
int main(){
    #ifdef A
        printf("1");
    #else
        printf("2");
    #endif
    return 0;
}
```

第二种形式：

```
#ifndef 标识符
    程序段1
#else
`   程序段2
#endif
```

和第一种形式的区别在于`#ifndef`。

功能：和第一种形式相反，如果标识符未被#define命令定义过则编译程序段1，否则编译程序段2。

第三种形式：

```
#if 表达式
    程序段1
#else
    程序段2
#endif
```

功能：如果表达式的值为真（非0）则编译程序1，否则编译程序2。

第8章 数组

数组是同类型数据的有序集合。

数组由若干个元素组成，其中所有元素都属于同一个基本数据类型，而且它们的先后次序是确定的。

8.1 一维数组

8.1.1 一维数组的定义

一般形式：

```
类型说明符    数组名[常量表达式]
```

数组的声明中包含了数组元素的数目和每个元素的类型。

说明：

- 数组名命名规则遵从标识符命名规则。
- 数组名后为`[]（取下标运算符）`，不能用圆括号。
- 常量表达式表示元素的个数，即数组长度。
- 常量表达式可以包含正整型常量和符号常量（仅限正整型），但不能包含变量。

```
int main(){
    int n = 10;
    int arr[n];  // 这种可以
    // int arr[n] = {0};  // 这种错误，error: variable-sized object may not be initialized
    return 0;
}
```

- 数组中的每个元素数据类型相同，在内存中是连续存放的

数组的下标（索引）始终从0开始，所以长度为n的数组的下标是从0到n-1。

8.1.2 一维数组的初始化

数组的初始化是在定义数组时就给数组中的元素赋予一个初始值。

一维数组初始化有四种方式：

（1）给所有元素赋初值

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

（2）只给一部分元素赋初值

```
int a[5] = {1, 2};
```

这种情况下只给数组里a[0]、a[1]分别赋值了1、2，其他元素均被系统自动赋初值为0。

（3）不指定长度赋初值

当使用空的方括号对数组进行初始化时，编译器会根据列表中的数值数目来确定数组的大小。

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

这种情况下数组a的大小就是5。

（4）指定初始化（C99标准）

```
int a[10] = {[2] = 2, [5] = 5, [8] = 7};
```

下标运算符`[]`中的数字为指示符，指定初始化中的指示符没有顺序要求。

这种情况下的数组中的其他元素均被系统自动初始化为0。

如以下代码的输出结果为：`0 0 2 0 0 5 0 0 7 0`。

```
int main(){
    int a[10] = {[2] = 2, [5] = 5, [8] = 7};
    for(int i = 0; i < 10; i++){
        printf("%d ", a[i]);
    }
    return 0;
}
```

以下的数组初始化都是合法的：

```
int a[10] = {};  // 全部为0
int b[] = {0};  // b数组只有一个元素为0
int c[10] = {10*1};  // c[0]为10，其余为0
```

8.1.3 一维数组元素的引用

数组必须先定义后使用。

C语言规定只能**逐个引用数组元素**而不能一次引用整个数组，也就是说，每次只能通过下标来引用数组中的单个元素，下标可以是整型常量或整型表达式。

8.1.4 一维数组的使用

`scanf()`和`printf()`不能一次处理整个数组的值，只能逐个处理数组元素，所以常常用到循环语句；应用循环语句时，需注意正确控制下标变量的范围。

注意：用输入语句给数组元素赋值的时候，数组元素前需要加上取地址符`&`。

数组的数组名是地址。

C语言中数组的名字不是变量，是数组开头元素的地址（数组首地址），它是编译时确定的一个地址常量。数组可以作为函数的参数来传递，在传递的时候是实参数组的首地址，而不是拷贝，所以在被调函数中对数组修改是会影响到实参数组本身的。

```
void fun(int a[]){
    a[0] = 100;
}
int main(){
    int a[10] = {[2] = 2, [5] = 5, [8] = 7};
    fun(a);
    for(int i = 0; i < 10; i++){
        printf("%d ", a[i]);
    }
    return 0;
}

```

比如以上程序会对a[0]元素进行修改，输出结果是`100 0 2 0 0 5 0 0 7 0`

数组名不能进行赋值操作，比如有以下定义：

```
int a[] = {1,2,3,4};
int b[10];
```

以下操作都是错误的：

```
a += 2;
a = b;
b = {1,2,3};
```

在结构体中也是不能直接对数组名进行赋值：

```
struct student
{
    int no;
    char name[10];
    float score;
}stu[5];

int main(){
    stu[0].no = 100;
    stu[0].score = 100;
    stu[0].name[0] = 'Z';
    stu[0].name = "Zhang";  // error: assignment to expression with array type
    return 0;
}
```

但是数组名可以进行加减操作：

```
printf("%d %d\n", *(a+1), *(a+2));  // 2 3
```

8.2 二维数组和多维数组

8.2.1 二维数组和多维数组的概念及定义

如果有一个一维数组，它的每一个元素都是类型相同的一维数组（数组类型相同包括其大小相同并各元素的类型相同）时，就形成了一个二维数组。

通常形象地把第一个下标称为行下标，第二个下标称为列下标。

定义一般形式为：

```
类型标识符 数组名[常量表达式][常量表达式];
```

可以在一行中定义同一类型的变量、一维数组和二维数组（或多维数组）：

```
int a, arr1[3], arr2[3][4];
```

从二维数组各元素在内存中的排列顺序可以计算出一个数组元素在数组中的顺序号。

假如一个$m \times n$的二维数组a，其中第i行第j列元素$a[i][j]$在数组中的位置计算公式为：$i \times n + j + 1$。

8.2.2 二维数组的初始化

二维数组初始化有五种方式：

（1）使用{ }手动分行赋初值

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

（2）不使用{ }自动分行赋初值

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

这种情况是按数组的排列顺序赋初值。数组每一行有三个元素，就会先把前三个值赋值给第一行的元素，后三个值赋给第二行元素。

（3）对部分元素赋初值

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

这种情况只对`a[0][0]、a[1][0]、a[1][1]`给定了初值，其余元素均被自动化初始为0。

也可以对某几行赋值：

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

这种情况下只对`a[0]、a[2]`两行进行了赋值。

（3）不指定一维长度赋初值

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

这种情况下，系统会根据数值总个数分配存储空间，因为指定了列数为3，数值总个数为9，所以可以得出行数为3。

在定义时，也可以只对部分元素赋初值且省略行数，但应当分行赋值。

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

注意：定义二维数组时行数可以省，**但是列数一定不能省。**

（5）指定初始化式对多维数组初始化（C99标准）

```
int a[3][4] = {[1][2] = 12, [2][2] = 22, [0][1] = 1};
```

比如以下代码的结果为：

```
int main(){
    int a[3][4] = {[1][2] = 12, [2][2] = 22, [0][1] = 1};
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 4; j++){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
0 1 0 0
0 0 12 0
0 0 22 0
```

8.2.3 二维数组和多维数组的引用

引用二维数组一般形式为：

```
数组名[下标][下标]
```

每个下标都应该用方括号括起来，下标是整型表达式，同样注意下标不能超过定义的范围。

8.3 字符数组

字符数组即基本类型为字符的数组，用来存放字符数据。

8.3.1 字符数组的定义

一般形式为：

```
char 数组名[常量表达式];
```

8.3.2 字符数组的输入输出

对于字符数组，可以将其每个元素当作单个字符型变量使用，也可以利用整个数组对一个字符串进行输入输出。

比如：

```
char s[] = {"hello"};
```

程序中定义了一个字符数组，并赋值了一个字符串`hello`。此时的s数组中含有六个元素：`h e l l o \0`，其中最后的`\0`是字符串的结束标志（输入时系统自动输入，输出是不显示，`\0`占一个内存空间，但不计入字符串的长度）。

字符串需要用`" "`括起来，输入输出整个字符串时，都要用`%s`格式符，输入输出项用字符数组名。比如：

```
char s[10];
scanf("%s", s);
printf("%s", s);
```

8.3.3 常用字符串函数

1、字符数组输出函数puts

将一个字符串（以`\0`结束的字符序列）输出到终端。比如：

```
int main(){
    char a[] = "hello";
    char b[] = {'a', 'b', 'c', 'd', 'e', '\0'};
    char c[] = {'A', 'B', 'C', 'D', 'E', 0};
    char d[] = {'1', '2', '3', '4', '5', '0'};
    puts(a);  // hello
    puts(b);  // abcde
    puts(c);  // ABCDE
    puts(d);  // 123450ABCDE
}
```

需要注意的是，如果使用`puts()`输出字符数组的时候，一定要在字符数组的末尾加上字符串结束标志，`\0`和`0`都是可以的，但是`'0'`不行。

2、字符串输入函数gets

从终端输入一个字符串到字符数组。以换行为输入结束。

3、字符串连接函数strcat

调用格式：`strcat(字符数组1, 字符数组2);`

连接两个字符数组中的字符串，将字符数组2接到字符数组1的后面。

4、字符串拷贝函数strcpy

调用格式：`strcpy(字符数组1, 字符串2);`

将字符串2拷贝到字符数组1中。

说明：

- 字符数组1的长度应该大于等于字符串2的长度
- 字符数组1不能是常量字符串
- 复制时，将字符串2的`\0`也复制过去了
- 不能用赋值语句将一个字符串常量或字符数组直接赋值给一个字符数组。以下示例都是不正确的：

```
int main(){
    char s[20], t[20] = "world";
    s = "hello";  // error: assignment to expression with array type
    s[20] = "hello";  // warning: assignment to 'char' from 'char *'
makes integer from pointer without a cast
    s = t;  // error: assignment to expression with array type
    s[20] = t;  // warning: assignment to 'char' from 'char *' makes
integer from pointer without a cast
    printf("%s\n", s);
    return 0;
}
```

5、字符串比较函数strcmp

两个字符串比较大小，不能用`==`，只能使用`strcmp()`。调用格式：`strcmp(字符串1，字符串2);`。

- 如果字符串1 > 字符串2：返回正整数。
- 如果字符串1 < 字符串2：返回负整数。
- 如果字符串1 = 字符串2：返回0。

比较规则：两个字符串自左向右逐个字符进行比较（按ASCII码值），直到出现不同的字符或者`\0`时比较结束。

6、测串长度函数strlen

调用格式：`strlen(字符数组);`

求字符串长度，返回字符串中原有字符的个数，不包括`\0`。

7、字符串转小写函数strlwr

将字符串转换成小写。调用格式：`strlwr(字符串);`

8、字符串转大写函数strupr

将字符串转换成大写。调用格式：`strupr(字符串);`

[其余常见字符串函数及模拟实现可见此处]

8.3.4 常量数组

所有数组都可以在定义时加上关键字`const`而成为常量数组。

`const`数组的特点：程序不可能改变数组元素的值

8.3.5 C99标准中的变长数组

用变量来声明长度的数组称为变长数组，变长数组的长度是程序在执行时计算的，而不是编译时计算的。

变长数组可以这样定义：

```
int n = 10;
int arr[n];
```

变长数组不能这样赋初值：

```
int n = 10;
int arr[n] = {1,2,3};
```

附录

ASCCII 码值表

ASCII表

(American Standard Code for Information Interchange 美国标准信息交换代码)

高四位 / 低四位		ASCII控制字符 0000 0						ASCII控制字符 0001 1						ASCII打印字符 0010 2		ASCII打印字符 0011 3		ASCII打印字符 0100 4		ASCII打印字符 0101 5		ASCII打印字符 0110 6		ASCII打印字符 0111 7		
		十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl
0000	0	0		^@	NUL	\0	空字符	16	▶	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH		标题开始	17	◀	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂	^Backspace 代码：DEL

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。

运算符优先级

运算符优先级

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	-----
	()	圆括号	(表达式)/函数名(形参列表)	左到右	-----
	.	成员选择（对象）	对象.成员名	左到右	-----
	->	成员选择（指针）	对象指针->成员名	左到右	-----
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式	右到左	-----
	++	前置自增运算符	++变量名	右到左	-----
	++	后置自增运算符）	变量名++	右到左	单目运算符
	--	前置自减运算符	--变量名	右到左	单目运算符
	--	后置自减运算符	变量名--	右到左	单目运算符
	*	取值运算符	*指针变量	右到左	单目运算符
	&	取地址运算符	&变量名	右到左	单目运算符
	!	逻辑非运算符	!表达式	右到左	单目运算符
	~	按位取反运算符	~表达式	右到左	单目运算符
	sizeof	长度运算符	sizeof(表达式)	右到左	-----
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式	左到右	双目运算符
	%	余数（取模）	整型表达式/整型表达式	左到右	双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式	左到右	双目运算符

5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式	左到右	双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式	左到右	双目运算符
	<	小于	表达式<表达式	左到右	双目运算符
	<=	小于等于	表达式<=表达式	左到右	双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!= 表达式	左到右	双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式1? 表达式2: 表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	-----
	/=	除后赋值	变量/=表达式	右到左	-----
	*=	乘后赋值	变量*=表达式	右到左	-----
	%=	取模后赋值	变量%=表达式	右到左	-----
	+=	加后赋值	变量+=表达式	右到左	-----
	-=	减后赋值	变量-=表达式	右到左	-----
	<<=	左移后赋值	变量<<=表达式	右到左	-----
	>>=	右移后赋值	变量>>=表达式	右到左	-----
	&=	按位与后赋值	变量&=表达式	右到左	-----

	^=	按位异或后赋值	变量^=表达式	右到左	-----
	|=	按位或后赋值	变量|=表达式	右到左	-----
15	,	逗号运算符	表达式,表达式,...	左到右	-----

printf的格式控制

函数语法如下：

```
// C99 前
int printf( const char *format, ... );
// C99 起
int printf( const char *restrict format, ... );
```

`printf`的格式控制字符串 `format`中的转换说明组成如下，其中 [] 中的部分是可选的：

`%[flags][width][.precision][length]specifier`，**即：** `%[标志][最小宽度][.精度][类型长度]说明符`

说明符

说明符（specifier）用于规定输出数据的类型，含义如下：

说明符 (specifier)	对应数据类型	描述
d / i	int	输出类型为有符号的十进制整数，i 是老式写法
o	unsigned int	输出类型为无符号八进制整数（没有前导 0）
u	unsigned int	输出类型为无符号十进制整数
x / X	unsigned int	输出类型为无符号十六进制整数。x 对应的是 abcdef，X 对应的是 ABCDEF（没有前导 0x 或者 0X）。
f / lf	double	输出类型为十进制表示的浮点数。默认精度为6（lf 在 C99 开始加入标准，意思和 f 相同）
e / E	double	输出类型为科学计数法表示的数。此处 "e" 的大小写代表在输出时用的 “e” 的大小写，默认浮点数精度为6
g	double	根据数值不同自动选择 %f 或 %e，%e 格式在指数小于-4或指数大于等于精度时用使用
G	double	根据数值不同自动选择 %f 或 %E，%E 格式在指数小于-4或指数大于等于精度时用使用
c	char	输出类型为字符型。可以把输入的数字按照ASCII码]相应转换为对应的字符
s	char *	输出类型为字符串。输出字符串中的字符直至遇到字符串中的空字符（字符串以 '\0‘ 结尾，这个 '\0' 即空字符）或者已打印了由精度指定的字符数
p	void *	以16进制形式输出指针
%	不转换参数	不进行转换，输出字符‘%’（百分号）本身
n	int *	到此字符之前为止，一共输出的字符个数，不输出文本

标志

标志（flags）用于规定输出样式，含义如下：

flags（标志）	字符名称	描述
-	减号	在给定的字段宽度内左对齐，右边填充空格（默认右对齐）
+	加号	强制在结果之前显示加号或减号（`+` 或 `-`），即正数前面会显示 `+` 号；默认情况下，只有负数前面会显示一个 `-` 号
（空格）	空格	输出值为正时加上空格，为负时加上负号
#	井号	`specifier` 是 `o、x、X` 时，增加前缀 `0、0x、0X`；`specifier` 是 `e、E、f、g、G` 时，一定使用小数点；`specifier` 是 `g、G` 时，尾部的 `0` 保留
0	数字零	对于所有的数字格式，使用前导零填充字段宽度（如果出现了减号标志或者指定了精度，则忽略该标志）

最小宽度

最小宽度（width）用于控制显示字段的宽度，取值和含义如下：

width（最小宽度）	字符名称	描述
digit(n)	数字	字段宽度的最小值，如果输出的字段长度小于该数，结果会用前导空格填充； 如果输出的字段长度大于该数，结果使用更宽的字段，不会截断输出
*	星号	宽度在 `format` 字符串中规定位置未指定，使用星号标识附加参数，指示下一个参数是 `width`

`*` 的用法：

```c
int main(){
    int i = 1;
    printf("##%*d\n", i, i);
    i++;
    printf("##%*d\n", i, i);
    i++;
    printf("##%*d\n", i, i);
    return 0;
}
```

以上程序的输出结果是：

```
##1
## 2
##  3
```

精度

精度（.precision）用于指定输出精度，取值和含义如下：

.precision (精度)	字符名称	描述
.digit(n)	点+数字	对于整数说明符（d、i、o、u、x、X）：precision 指定了要打印的数字的最小位数。 　如果写入的值短于该数，结果会用前导零来填充。 　如果写入的值长于该数，结果不会被截断。 精度为 0 意味着不写入任何字符； 对于 e、E 和 f 说明符：要在小数点后输出的小数位数； 对于 g 和 G 说明符：要输出的最大有效位数； 对于 s 说明符：要输出的最大字符数。 默认情况下，所有字符都会被输出，直到遇到末尾的空字符； 对于 c 说明符：没有任何影响；当未指定任何精度时，默认为 1。 如果指定时只使用点而不带有一个显式值，则标识其后跟随一个 0。
.*	点+星号	精度在 format 字符串中规定位置未指定，使用点+星号标识附加参数，指示下一个参数是精度

类型长度

类型长度（length）用于控制待输出数据的数据类型长度，取值和含义如下：

ngth (类型长度)	描述
h	参数被解释为短整型或无符号短整型（仅适用于整数说明符：i、d、o、u、x 和 X）
l	参数被解释为长整型或无符号长整型，适用于整数说明符（i、d、o、u、x 和 X）及说明符 c（表示一个宽字符）和 s（表示宽字符字符串）

转义序列

转义序列在字符串中会被自动转换为相应的特殊字符。常见转义字符如下：

转义序列	描述
`\'`	单引号
`\"`	双引号
`\?`	问号
`\\`	反斜杠
`\a`	铃声（提醒）
`\b`	退格
`\f`	换页
`\n`	换行
`\r`	回车
`\t`	水平制表符
`\v`	垂直制表符

scanf的格式控制

下面是 scanf() 函数的声明。

```
int scanf(const char *format, ...)
```

参数：

- **format** -- 这是 C 字符串，包含了以下各项中的一个或多个：空格字符、非空格字符 和 *format* 说明符。

format 说明符形式为:

```
[=%[*][width][modifiers]type=]
```

具体讲解如下：

参数	描述
*	这是一个可选的星号，表示数据是从流 stream 中读取的，但是可以被忽视，即它不存储在对应的参数中。
width	这指定了在当前读取操作中读取的最大字符数。
modifiers	为对应的附加参数所指向的数据指定一个不同于整型（针对 d、i 和 n）、无符号整型（针对 o、u 和 x）或浮点型（针对 e、f 和 g）的大小： h：短整型（针对 d、i 和 n），或无符号短整型（针对 o、u 和 x） l：长整型（针对 d、i 和 n），或无符号长整型（针对 o、u 和 x），或双精度型（针对 e、f 和 g） L：长双精度型（针对 e、f 和 g）
type	一个字符，指定了要被读取的数据类型以及数据读取方式。具体参见下一个表格。

scanf 类型说明符：

类型	合格的输入	参数的类型
%a、%A	读入一个浮点值(仅 C99 有效)。	float *
%c	单个字符：读取下一个字符。如果指定了一个不为 1 的宽度 width，函数会读取 width 个字符，并通过参数传递，把它们存储在数组中连续位置。在末尾不会追加空字符。	char *
%d	十进制整数：数字前面的 + 或 - 号是可选的。	int *
%e、%E、%f、%F、%g、%G	浮点数：包含了一个小数点、一个可选的前置符号 + 或 -、一个可选的后置字符 e 或 E，以及一个十进制数字。两个有效的实例 -732.103 和 7.12e4	float *
%i	读入十进制，八进制，十六进制整数 。	int *
%o	八进制整数。	int *
%s	字符串。这将读取连续字符，直到遇到一个空格字符（空格字符可以是空白、换行和制表符）。	char *
%u	无符号的十进制整数。	unsigned int *
%x、%X	十六进制整数。	int *
%p	读入一个指针 。	
%[]	扫描字符集合 。	
%%	读 % 符号。	

- **附加参数** -- 根据不同的 format 字符串，函数可能需要一系列的附加参数，每个参数包含了一个要被插入的值，替换了 format 参数中指定的每个 % 标签。参数的个数应与 % 标签的个数相同。

返回值

如果成功，该函数返回成功匹配和赋值的个数。如果到达文件末尾或发生读错误，则返回 EOF。

习题练习

第1章

> 教材课后习题：

1、下面关于程序的说法中，正确的是（A）

A. 程序就是人与计算机交流的语言
B. 程序是指由二进制0、1构成的代码
C. 将需要计算机完成的工作写成一种形式化的指令，而这些单个的指令就是程序
D. 程序的设计形式是一致的

关键点：程序是能被机器识别并执行的一系列的指令代码，这些指令代码是用程序设计语言描述的。

- 用来描述的语言很多，不限于二进制，所以B错误。
- 能被机器识别并执行的一系列的指令代码才是程序，所以C错误。
- 程序的设计形式可以是面向对象、面向过程等，所以D错误。

2、C语言中，编译是指 将高级语言源程序翻译成目标程序 的过程。

3、程序调试的目的是：发现错误并改正错误。

> C程序设计试题汇编第三版：

4、C语言的源程序必须通过 编译 和 链接 后，才能被计算机执行。

5、在C语言源程序中，一个变量代表 内存中一个存储单元 。

第2章

> 教材课后习题：

1、在C语言中，正确的int类型的常数是（D）。

A. -2U B. 059 C. 3a D. 0xaf

- A： U是无符号数，而前面又有负号
- B：八进制中只能出现0-7数字
- C：十进制中只能出现0-9数字

2、下列变量定义正确的是（B）。

A. int x_1;y; B. int x=y=5; C. int for=4; D. printf=2,x_y=2;

- A：y没有定义
- C：for是关键字
- D：printf是预定义标识符，可以作为变量名，但是不能再使用printf函数输出功能。

3、设有以下语句，则C的二进制值是（A）。

```
char a=3, b=6, c;
c = a ^ b << 2;
```

A. 00011011 B. 00010100 C. 00011100 D. 00011000

- << 运算符的优先级要高于 ^ 运算符

4、执行下列语句后，z的值是（8）。

```
int x = 4, y = 25, z = 2;
z = (--y / ++x) * z--;
```

- 等于`z = (24 / 5) * z--`，即`z = 4 * z--`，而这里的`z = z--`可以这样看

```
z = 4 * z--;
int t = z;  // 后置--，先保留当前原值
z = z - 1;  // 而后进行-1
z = 4 * t;  // 再把值赋给等号左边
```

- 运算结果取决于编译器

5、已知：`int i,j,k; i=j=k=3;`，求`~i | i & j`的结果；（-1）。

- `~`运算符的优先级高于`&`，`&`优先级高于`|`，所以等价于`(~i) | (i & j)`。
- 需要注意的是，`~`不会改变操作数的值，所以`~i`的二进制：`11111111 111111111 11111111 11111100(补码)`。
- `i&j`的二进制结果：`00000000 00000000 00000000 00000011`。
- 所以两者执行`|`运算，结果：`11111111 11111111 11111111 11111111(补码)`。转为原码就是-1。
- 注意：执行运算的都是补码，所以`~i`的结果可以不转为原码，因为它还要继续运算。

6、写出下列程序的输出结果：

```
main(){
    char c = 'x';
    short i = -4;
    print("c：dec=%d，oct=%o，hex=%x，ASCII=%c\n", c,c,c,c);
    print("i：dec=%d，oct=%o，hex=%x，unsigned=%u\n", i,i,i,i);
}
```

输出结果：

```
c：dec=120，oct=170，hex=78，ASCII=x
i：dec=-4，oct=37777777774，hex=fffffffc，unsigned=4294967292
```

- `%d`是十进制格式输出，`%o`是八进制格式输出，`%x`是十六进制格式输出，`%u`是无符号格式输出。
- 需要输出八进制和十六进制前缀的话，加上一个`#`：`%#o`，`%#x`。
- 其中`short i = -4;`理论上讲是2个字节长度，但是实际是由编译器决定分配多少个字节的，只能说：short类型可能比int类型空间小

> **2.**使用多种整数类型的原因
>
> 为什么说short类型“可能”比int类型占用的空间少，long类型“可能”比int类型占用的空间多？因为C语言只规定了short占用的存储空间不能多于int，long占用的存储空间不能少于int。这样规定是为了适应不同的机器。例如，过去的一台运行Windows 3的机器上，int类型和short类型都占16位，long类型占32位。后来，Windows和苹果系统都使用16位储存short类型，32位储存int类型和long类型（使用32位可以表示的整数数值超过20亿）。现在，计算机普遍使用64位处理器，为了储存64位的整数，才引入了long long类型。

- 所以理论上输出结果是：`-4 17773 fffc`。

第3章

> 教材课后习题

4、下列程序执行后的输出结果是（A）。

```
main(){
    double d;
    float f;
    long l;
    int i;
    i=f=l=d=20/3;
    printf("%d%ld%f%f\n",i,l,f,d);
    return 0;
}
```

A. 6 6 6.000000 6.000000　　B. 6 6 6.700000 6.700000　　C. 6 6 6.000000 6.700000　　D. 6 6 6.700000 6.000000

`20/3`的结果是int型的，所以d被附的值就是6.000000，而不是6.666666，如果是`20.0/3`的话，那么d的值就是6.666667。

分析以下程序的结果：

```
int main(){
    int i = 1;
    printf("%d\n", ~i++);
    printf("%d\n", i);
    i = 1;
    printf("%d\n",~++i);
    printf("%d\n", i);
    return 0;
}
```

结果为：

```
-2
2
-3
2
```

`printf("%d\n", ~i++);`先执行`~i`，然后`i=i+1`；

`printf("%d\n",~++i);`先执行`i=i+1`，然后`~i`。

第4章

> 教材课后习题

2、阅读下面的程序，说法正确的是（D）

```c
int main(){
    int x = 3, y = 0, z = 0;
    if(x = y + z){
        printf("****\n");
    }
    else{
        printf("####\n");
    }
    return 0;
}
```

A. 有语法错误，不能通过编译　B. 输出****　C.可以通过编译，但是不能通过链接，因而不能运行　D. 输出####

1、switch语句的case表达式可以是 整型常量或字符型常量。

8、以下程序运行的结果是 b=0

```c
int main(){
    int a = 0, b = 0;
    if(a > 3)
        if(a < 7)
        b = 1;
    else
        b = 2;
    printf("b=%d\n", b);
    return 0;
}
```

关键点：else与前面最近的一个if配对，if else 是一个完整的语句。

3、由键盘输入三个数，计算以这三个数为边长的三角形的面积。 （
$s=\sqrt{p(p-a)(p-b)(p-c)}, p=(a+b+c)/2)$

第5章

> 教材课后习题

7、下面程序执行结果是（A）

A. 1　B. 30　C. 1-2　D. 死循环

```c
int main(){
    int x = 3;
    do{
        printf("%d\n", x-=2);
    }while(!(x--));
    return 0;
}
```

先把x的值拷贝一份，然后原值减一，拷贝值用来进行非操作。

第6章

> 教材课后习题

4、以下说法正确的是（D）

A. #define 和 printf 都是C语句

B. #define 是C语句，而 printf 不是

C. printf 是C语句，但 #define 不是

D. #define 和 printf 都不是C语句

7、有关宏定义的正确说明是（D）

A. 可出现在一行中的任何位置

B. 只能放在程序的开头，且每一个宏定义单独占一行

C. 可出现在程序的任何位置

D. 以#开头的行，可出现在程序的任何位置，通常每一个宏定义只能单独占一行，使用字符"\"可实现一个宏定义占多行

8、若程序中有宏定义行：`#define N 100`，则以下叙述中正确的是（B）

A. 宏定义行中定义了标识符N的值为整数100

B. 在编译程序时对C源程序进行预处理时用100替换标识符N

C. 对C源程序进行编译时用100替换标识符N

D. 在运行时用100替换标识符

16、以下函数调用语句的实参个数为（B）

```
fun((exp1, exp2), (exp3, exp4, exp5));
```

A. 1 B. 2 C. 4 D. 5

18、有以下程序，下列说法中不正确的是（C）

```
#include<stdio.h>
void f(int n);
int main(){
    void f(int n);
    f(5);
    return 0;
}
void f(int n){
    pritnf("%d\n", n);
}
```

A. 若只在主函数中对函数f进行说明，则只能在主函数中正确调用函数f。

B. 若在主函数前对函数f进行说明，则在主函数和其后的其它函数中都可以正确调用函数f。

C. 对于以上程序编译时，系统会提示出错信息，提示对函数f重复说明。

D. 函数f无返回值，所以可用Void将其类型定义为无值型。

1、编译预处理的三种形式除宏定义外，还有文件包含和条件编译。