第3章 简单的C程序

本章导读

任何一门高级语言都有其自身的一套语法规范，它是用高级语言编程的技术基础，也可以说是一套必备的编程工具。

在编程时，需要解决的问题包括如何对程序中的变量、函数命名？如何将程序要处理的数据放到计算机的内存中？设想一个学生的成绩是一个整数，一个班的平均分是浮点数，一个人的名字是一个字符串，这些不同类型的数据在计算机的内存中如何存储？我们对不同类型的数据又能进行哪些操作呢？另外，数据处理是计算机程序的一个非常重要且核心的任务，那么，程序怎样获得需要处理的数据？又该如何输出那些运算的结果呢？

凡此种种编程的基本问题，都需要我们打开C语言的工具箱，学习其基本的编程知识。

本章主要内容

- 标识符。
- 程序中的常量与变量。
- 简单数据类型：整型、浮点型、字符型。
- 常用运算符。
- 使用表达式。
- 进行数据的输入和输出。

3.1 标 识 符

标识符是程序中的变量、常量、函数、数据类型等任何用户自定义项目的名字。C程序标识符的定义规则为：

(1) 标识符是以字母或下画线开始的，由字母、数字和下画线所构成的字符串。其中，字母包括A～Z或a～z，数字包括0～9。

(2) 标识符对大小写敏感，即严格区分大小写。例如，book和BOOK是两个不同的

标识符。一般对变量名用小写字母,符号常量命名用大写字母。

(3) 不能把 C 语言的关键字[①]作为自定义的标识符。

以上是定义 C 语言标识符必须遵循的规则。如果标识符的命名不符合以上规则,会引起程序编译出错。

此外,在实际开发中还有一些约定俗成的标识符命名规范[②]。例如,标识符的命名应"见名知意",用 length 表示长度,用 sum 表示求和,用 age 表示年龄等。虽然这不受语法约束,但遵守这些规范会增加程序的可读性。

3.2 变　　量

1. 定义变量

程序运行时所使用的数据都必须放在计算机的内存中。运行程序时,从内存中读取存放的数据进行处理。进一步地说,程序必须能够把数据存储在内存中,才能对数据进行处理。那么,如何才能将数据放入内存呢?

变量(variable)是指程序运行中其值可以被改变的量。变量是程序操作计算机内存的一种手段。程序可以通过定义变量来申请内存空间,从而将待处理的数据放入内存。

程序 3.1 表明了变量的用法及含义。

【程序 3.1】 变量的定义及使用。

```
#include <stdio.h>
int main()
{
    int score;                          //定义一个 int 型变量,其名称为 score
    score=92;
    printf("My score is %d.",score);
    return 0;
}
```

本例程序在 main()函数中定义了一个变量 socre。

定义变量的方法为:

```
int score;                              //定义一个 int 型变量,其名称为 score
```

定义变量时,每个变量都有一个名字,这个名字是合法的 C 语言标识符,也是所申请到的内存空间的代名词。同时,还必须声明变量的数据类型,它表明了申请的内存空间的

① 在程序 2.2 中讨论过关键字,可以前往该程序对照查看。

② 2.3 节也讨论过标识符的命名规范问题。

大小，以及我们能对变量执行什么样的操作。根据数据类型不同，分配给变量的内存空间可能为 1、4、8 或更多字节。

例如，程序 3.1 中定义了一个名为 score 的变量，同时声明了 score 的数据类型为 int，它表明程序申请一个 int 型数据的内存空间，以便能将一个 int 型的数值放入内存。

一旦声明了变量，程序就可通过变量名来引用所申请的内存空间。例如：

```
score=92;
```

该语句是将整数 92 写入变量 score 所标识的内存空间。

再如：

```
printf("My score is %d.",score);
```

该语句是读取变量 score 所标识的内存空间中的数据，用 printf()函数将其输出到显示器上。

注意，变量的值之所以可变，是因为变量名所标识的内存空间既可读(读取内存中变量的值)、又可写(往变量的内存空间写入数据)。

2. 变量必须先定义，后使用

在程序中使用变量时，就是对变量进行读或写，本质都是访问变量的内存空间。因此，使用变量前，必须先定义变量，为其分配存储空间。可以说，定义变量是程序使用内存进行数据操作的前提。

3. 变量的初始化

可以在定义变量的同时赋给其初始值，这称为变量的初始化，也可以先定义，再赋予其初始值。相同数据类型的变量可以同时定义，中间用逗号分开，最后一个变量名之后必须以分号结尾。如下变量定义及初始化都是合法的。

```
int cols=3,rows=5;                    //定义变量的同时初始化
int C_score;
C_score=89;                           //先定义,再初始化
```

注意，在初始化时不允许连续赋值，如下操作是不合法的。

```
int a=b=c=5;                          //不合法的初始化方法
```

一般地，变量使用前必须赋初值，否则变量的值是不确定的。可以执行以下代码，观察和分析输出的结果。

```
int number;
printf("%d",number);                  //输出未初始化的变量 number,其值是不确定的
```

3.3 数据类型

在定义变量时，指定变量的数据类型(data type)。数据类型决定了数据在内存中的存储形式以及能对数据实施什么样的运算。

内置数据类型(built-in data type)是编程语言提供给程序员直接使用的数据类型，也称基本数据类型(primitive type)。C 语言的基本数据类型及其能够实施的常见运算如表 3.1 所示。

表 3.1 C 语言的基本数据类型及其能够实施的运算

数据类型	能够实施的运算
整型	算术运算：+、-、*、/、% 赋值运算：=、+=、-=、*=、/=、%= 关系运算：>、<、>=、<=、==、!= 其他运算：sizeof()、位运算、逻辑运算等
字符型	
浮点型	算术运算：+、-、*、/ 赋值运算：=、+=、-=、*=、/= 关系运算：>、<、>=、<=、==、!= 其他运算：sizeof()、逻辑运算等

以下讨论 C 语言的几种基本数据类型。每种类型都分别从常量、变量两方面进行讨论。

3.3.1 整型

1. 整型常量

常量是指在程序执行时，其值不会发生改变的量。C 程序中的整型常量有十进制、八进制、十六进制三种。

(1) 十进制整型常量。

其数码为 0～9，以下都是合法的十进制整型常量。

```
237   -568   65535   1627    0
```

(2) 八进制整型常量。

带前缀 0 的整型常量为八进制常量，其数码为 0～7。八进制常量通常是无符号数。以下都是合法的八进制整型常量。

```
015    0101    0177777
```

(3) 十六进制整型常量。

带前缀 0X 或 0x 的整型常量为十六进制常量，其数码为 0～9、A～F 或 a～f。十六进

制常量通常是无符号数。以下都是合法的十六进制整型常量。

```
0X2A    0XA0    0XFFFF
```

(4) 整型常数的后缀。

对整型常数的取值范围,取决于数据所占用的字节数,不同的编译器有自己的内部限定。例如,如果 C 语言编译器为整型分配 4 字节,其允许的整数值的范围如下:

```
十进制无符号数的范围:0~4294967295
十进制有符号数的范围:-2147483648~2147483647
```

如果使用的数超出了上述范围,可以用长整型数表示。长整型的常量用后缀 L 或 l 来表示。例如:

```
十进制长整型常量:162L    341000L
八进制长整型常量:012L    077L    0200000L
十六进制长整型常量:0X15L   0XA5L   0X10000L
```

长整数 162L 与基本整型常量 162 在数值上并无区别,但由于 162L 是长整型量,编译器将为它分配 8 字节的存储空间,而对基本整型 162 只分配 4 字节存储空间,所以在运算和输出格式上都应予以区分。

无符号数也可用后缀表示,整型常数的无符号数的后缀为 U 或 u。例如,下面的 358u 为十进制无符号整数,0x38Au 是前缀、后缀同时使用,它表示十六进制无符号整数,235Lu 表示十进制无符号长整数。

```
358u    0x38Au    235Lu
```

2. 整型变量

(1) 整型数据的类型。

C 语言中提供的整型数据类型如表 3.2 所示,其中给出了 Dev C++ 下各种整型及其取值范围的描述。各类整型数据间的本质差异是它们占用的存储空间大小各不相同,这直接影响了它们的取值范围。例如,在 Dev C++ 中,短整型在内存中占 2 字节,因此其取值范围为−32768～32767。

表 3.2　C 语言的整型数据

类型声明符	描　　述	字节数	数 值 范 围	
[signed]**int**	基本整型	4	−2147483648～2147483647	$-2^{31}\sim(2^{31}-1)$
unsigned [int]	无符号整型	4	0～4294967295	$0\sim(2^{32}-1)$
[signed]**short** [int]	短整型	2	−32768～32767	$-2^{15}\sim(2^{15}-1)$
unsigned short [int]	无符号短整型	2	0～65535	$0\sim(2^{16}-1)$

续表

类型声明符	描　　述	字节数	数 值 范 围	
[signed]**long** [int]	长整型	4	−2147483648～2147483647	$-2^{31}\sim(2^{31}-1)$
unsigned long [int]	无符号长整型	4	0～4294967295	$0\sim(2^{32}-1)$
[signed]**long long** [int]	64 位长整型	8	—	$-2^{63}\sim(2^{63}-1)$
unsigned long long [int]	无符号 64 位长整型	8	—	$0\sim(2^{64}-1)$

注：类型声明符[]中的关键字为可省略项。

对各种数据类型所占用存储空间的大小，不同的编译器有不同的内部限定[①]。一般来说，short 型不会比 int 型占的字节数多，long 型不会比 int 型占的字节数少。

在编程时，程序员应首先了解所使用编译环境，再根据实际问题的数据范围，选用合适的数据类型。程序 3.2 是用 sizeof()查看 Dev C++ 环境下各种整型所占用的字节数。

【程序 3.2】 用 sizeof()查看 Dev C++ 整型的字节数。

```
#include <stdio.h>
int main()
{
    printf("sizeof([signed]int)=%d\n",sizeof(signed int));
    printf("sizeof([signed]short)=%d\n",sizeof(signed short));
    printf("sizeof([signed]long)=%d\n",sizeof(signed long));
    printf("sizeof([signed]long long)=%d\n",sizeof(signed long long));
    printf("sizeof(unsigned int)=%d\n",sizeof(unsigned int));
    printf("sizeof(unsigned short)=%d\n",sizeof(unsigned short));
    printf("sizeof(unsigned long)=%d\n",sizeof(unsigned long));
    printf("sizeof(unsigned long long)=%d\n",sizeof(unsigned long long));
    return 0;
}
```

程序 3.2 的运行结果如下：

```
sizeof([signed]int)=4
sizeof([signed]short)=2
sizeof([signed]long)=4
sizeof([signed]long long)=8
sizeof(unsigned int)=4
sizeof(unsigned short)=2
sizeof(unsigned long)=4
sizeof(unsigned long long)=8
```

① 不同的编译器支持的 C 标准可能不一样，因此其支持的数据类型也有差异。例如，Visual Studio C++ 6.0 不支持 long long int 数据类型。

(2) 什么是数据溢出?

数据溢出是程序运行时可能发生的错误,一般是由于数的取值超出了其数据类型的表示范围。

程序 3.3 示例了发生数据溢出的情况。

【程序 3.3】 数据溢出问题。

```
#include <stdio.h>
int main()
{
    int n=14,fac=1,i;
    for(i=1;i<=n;i++)
    {
        fac*=i;
        printf("%2d!=%-11d",i,fac);
        if(i%4==0) printf("\n");
    }
    return 0;
}
```

该程序的运行结果如下:

```
1!=1              2!=2              3!=6              4!=24
5!=120            6!=720            7!=5040           8!=40320
9!=362880         10!=3628800       11!=39916800      12!=479001600
13!=1932053504    14!=1278945280
```

程序 3.3 的运行环境为 Dev C++,其 int 类型所占的字节数为 4 字节。可见,在计算 13、14 的阶乘时,运行结果出错(这里将其加粗显示),这是因为它们的阶乘值超出了 int 所能存储的最大的正整数,因此发生数据溢出。为此,将程序 3.3 改写为程序 3.4。

【程序 3.4】 计算整数 1～14 的阶乘。

```
#include <stdio.h>
int main()
{
    double n=14,fac=1;
    int i;
    for(i=1;i<=n;i++)
    {
        fac*=i;
        printf("%2d!=%-13.0lf",i,fac);
        if(i%4==0) printf("\n");
    }
```

```
    return 0;
}
```

程序 3.4 的运行结果为：

```
1!=1              2!=2              3!=6              4!=24
5!=120            6!=720            7!=5040           8!=40320
9!=362880         10!=3628800       11!=39916800      12!=479001600
13!=6227020800    14!=87178291200
```

在程序 3.4 中换用 double 类型来计算整数的阶乘值。由于在 Dev C++ 中，double 占用 8 字节，因此解决了程序 3.3 中的溢出问题。注意，编译器对运行结果溢出的情况无法报错和报警，只有在程序运行过程中才可能出现溢出，并导致无法预知的运行结果，因此，必须在问题分析阶段注意选取合适的数据类型，防止数据溢出的情况发生。

3.3.2 浮点型

1. 浮点型常量

浮点型也称为实型，浮点型常量(floating-point number)也称为实数(real number)。在 C 语言中，浮点型常量有两种描述形式：十进制小数形式与指数形式。

(1) 十进制小数形式。

由数码 0～ 9 和小数点组成。以下均为合法的实数：

```
0.0    251.0    53.76   0.37   -7.9   4.   -267.294   10.
```

其中，数字 4.、0.0 和 10. 也是合法的实数，其小数点前(或后)省略的数字为 0。

一般地，浮点型常量默认为 double 类型的数据，也可以在浮点型常量的末尾添加一个 f 或 F，表明其是一个 float 型常量，以区别于 double 类型。

例如，以下声明并初始化了两个浮点型变量 area 和 maxsize：

```
float area=4.6f;                          //4.6f 是 float 类型的常量
double maxsize=12E15;                     //12E15 是 double 类型的常量
```

(2) 指数形式。

浮点数也可以写成指数形式，类似于科学计数法。表 3.3 是浮点型常量的十进制小数形式、指数形式及其对应描述。

表 3.3 C 语言的浮点数表示法

十进制小数形式	指 数 形 式	指数形式的描述
572.91	5.7291e2	5.729×10^2
12345.	1.2345e4	1.2345×10^4

续表

十进制小数形式	指 数 形 式	指数形式的描述
－0.003219	－3.219e-3	-3.219×10^{-3}
.004562	4.562e-3	4.562×10^{-3}

在指数形式中，字母 e 代表指数(exponent)。字母 e 的前后都必须有数字，e 后面的数字称为阶码，阶码只能是整数，可以带符号。字母 e 也可以是大写 E。

注意，以下指数形式是不合法的：

```
E5:字母 E 之前无数字
123.-E4:负号的位置不对
2.7E:字母 E 之后无数字
```

2. 浮点型变量

浮点型变量只能存储浮点数。C 语言的浮点型变量有单精度(float)、双精度(double)和长双精度(long double)三种类型。与整数一样，不同的浮点型变量在内存中所占的字节数取决于编译器。在某些编译器上，float 型变量占 4 字节，有效位数为 7 位，double 型变量占 8 字节，有效位数为 15 位。如果问题中有更精确、更大范围的数，则可以使用 long double 型。

表 3.4 是三种浮点类型及其取值范围的描述。

表 3.4　C 语言的浮点类型及相关描述

类型声明符	描述	字节数	数 值 范 围	有效位数
float	单精度	4	$-3.4\times10^{38}\sim 3.4\times10^{38}$	6～7
double	双精度	8	$-1.7\times10^{308}\sim1.7\times10^{308}$	15～16
long double	长双精度	16	$-1.2\times10^{4932}\sim1.2\times10^{4932}$	18～19

浮点型变量的定义形式与整型变量相同。以下是定义浮点型变量的示例：

```
float x,y;                                  //x,y 为单精度浮点型变量
double a,b,c;                               //a,b,c 为双精度浮点型变量
```

3.3.3　字符型

1. 字符常量

字符常量是用单引号括起来的单个字符，例如，以下均是合法的字符常量。

```
'a'   'b'   '='   '+'   '?'
```

C 语言的字符常量具有以下特点：

(1) 字符常量只能用单引号括起来,不能用双引号或其他括号。

(2) 字符常量只能是单个字符。

一般地,字符常量以 ASCII 码的形式存储在计算机中。ASCII 码(American Standard Code for Information Intercharge)是一个包含 256 个代码的字符集。其中,每个字符对应一个 ASCII 码,每个 ASCII 码占 1 字节(8 比特)。

例如,字符'a'的 ASCII 码为 01100001,对应十进制的整数 97,其存储形式如下:

'a'	0	0	1	1	0	0	0	1

ASCII 码表见附录 A。

2. 转义字符

转义字符也称为转义序列(escape sequence),是一种特殊的字符常量。转义字符以反斜线字符(\)开头,后跟一个或几个字符。转义字符具有特定的含义,不同于反斜线后字符的原有意义,故称转义字符。例如,以下 printf()函数的格式串中的\n 就是一个转义字符,其含义是"回车并换行"。

```
printf("%.0lf!=%.0lf\n",i,fac);
```

C 语言的转义字符如表 2.1 所示。

3. 字符变量

字符变量用来存储字符常量,即存储单个字符。

字符变量的类型声明符是 char。以下定义了两个字符变量 a 和 b:

```
char a,b;
```

每个字符变量被分配一字节的内存空间,用于存放单个字符值。字符值以 ASCII 码的形式被存放到字符变量的内存单元中。

例如,以下语句将字符常量'k'和'q'分别赋给字符变量 a 和 b:

```
a='k';
b='q';
```

由于字符'k'的 ASCII 码是 01101011(十进制 107),字符'q'的 ASCII 码是 01110001(十进制 113),因此,以上语句实际上是将十进制整数 107 和 113 分别存放到变量 a、b 的内存单元中,存储形式为:

变量 a	0	1	1	0	1	0	1	1
变量 b	0	1	1	1	0	0	0	1

因此,也可以把 char 型变量 a、b 分别看成是一个整型量。允许将整型值赋给字符型变量,也允许将字符值赋给整型变量。程序 3.5 示例了字符变量的这一特性。

【程序 3.5】　对字符变量的基本操作。

```
#include <stdio.h>
int main()
{
    char a,b;
    a=97;
    b=98;
    printf("a=%c,a=%c\n",a,b);          //语句①:以字符形式输出两个字符变量
    printf("a=%d,a=%d",a,b);            //语句②:以整数形式输出两个字符变量
    return 0;
}
```

程序 3.5 的运行结果为：

```
a=a,a=b
a=97,a=98
```

程序 3.5 中定义了 a 和 b 两个字符型变量，分别将它们赋给整型值 97、98。语句①用格式符%c 输出变量 a 和 b 的字符形式，语句②用格式符%d 输出变量 a 和 b 的整数形式，此时输出的是它们的 ASCII 码。

由于字符型变量中存储的是字符的 ASCII 码值，因此，字符变量也可用于数值运算[①]，此时是用它的 ASCII 码值参与运算。程序 3.6 示例了这一操作方法。

【程序 3.6】　字符变量参与数值计算。

```
#include <stdio.h>
int main()
{
    char a,b;
    a='a';                                  //'a'的 ASCII 码为 97
    b='b';                                  //'b'的 ASCII 码为 98
    printf("before: a=%d,b=%d\n",a,b);
    a=a-32;                                 //语句①:将变量 a 的值修改为 65
    b=b-32;                                 //语句②:将变量 b 的值修改为 66
    printf("after: a=%c,b=%c",a,b);
    return 0;
}
```

① char 型变量存储的整数可以表示为带符号数或无符号数，这与所使用的编译器相关。如果表示为无符号数，则 char 型变量存储的整数范围为 0～255；若为带符号数，则其存储的整数范围为－128～127。在将 char 型变量看成整数进行运算时，必须保证所操作的值在其可存储的值域内。

程序 3.6 的运行结果为：

```
before: a=97,b=98
after: a=A,b=B
```

本例中，变量 a 和 b 被声明为字符变量，并分别赋给字符值'a'和'b'，此时，变量 a 中存放着字符'a'的 ASCII 码 97，变量 b 中存放着字符'b'的 ASCII 码 98。在语句①、语句②中，分别用变量 a 和 b 的值减去 32，这种操作是合法的，实际是将它们存储的 ASCII 码分别修改为 65、66，由此得到字符'A'、字符'B'的 ASCII 码。

4. 字符串常量

字符串是一种十分常用的数据类型，许多应用场景中的数据都是字符串形式，例如人名、单位地址、身份证号码、书籍的出版社等。在开发处理这类数据的系统时，就需要使用字符串。

字符串常量是由一对双引号括起的字符序列，序列中可以包含 0 个或若干个字符。例如，以下都是合法的字符串常量：

```
"CHINA"    "C Language"    "+8618002110203"    "A"
```

字符串常量在内存中存储时，除了其本身包含的每个字符要占用一字节外，系统还在其最后一个字符的末尾增加一个字符\0（ASCII 码为 0），作为字符串的结束标志，因此，一个字符串所占用的字节数等于该字符串的字符数加 1。

例如，字符串“C Language”在内存中的存储形式如下，在最后一个字符 e 之后补了字符\0，因此该字符串包含 10 个字符，但实际占用 11 字节。

C		L	a	n	g	u	a	g	e	\0

再来比较一下字符常量'a'和字符串常量"a"的区别，两者在内存中的存储形式如下：

'a' | a |　　"a" | a | \0 |

可见，虽然两者都只含一个字符，但其存储却完全不同，'a'在内存中占 1 字节，"a"则占 2 字节。

3.4 运 算 符

C 语言提供了丰富的运算符，这些运算符的操作对象称为操作数。将不同的运算符和操作数组合起来，可构造出十分灵活、多样的表达式，这也是 C 语言的主要特点之一。

表达式(expression)是由运算符、常量、变量、函数调用所构成的式子，例如，以下都是合法的 C 语言表达式：

```
a+b-c
x-20/15*(p-q)
(x+y)*(m-n)/(sqrt(a*a+b*b)+sin(5))
```

注意，分式表达式的分子和分母都写在同一行上，因此，必须仔细考虑运算优先级的问题。一般来说，表达式通过小括号来明确优先级。无论有多少层括号，应该全部都是小括号。

每个表达式都有其值和类型。表达式求值按运算符的优先级、结合性两方面所规定的顺序进行。

C 语言的运算符都有其优先级和结合性。在表达式求值时，各运算符按其优先级从高到低的顺序依次计算，如果两个运算符的优先级相同，则按它们的结合性要求进行计算。

1. 运算符的优先级

C 语言运算符的优先级共分为 15 级。1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算。例如，如下表达式：

```
1+2*3
```

其中，运算符乘(*)的优先级为 3，加(+)的优先级为 4，乘的优先级更高，因此先计算 2*3，再计算 1+6。

但表达式中还可能出现一个操作数两侧的运算符优先级相同的情况，例如：

```
1+2-3
```

其中，运算符加(+)和减(-)的优先级同为 4，此时应先计算加还是减呢？这就需要考虑运算符的结合性了。

2. 运算符的结合性

C 语言的表达式中，当一个操作数两侧的运算符优先级相同时，则按运算符的结合性来确定表达式的计算顺序。C 语言运算符的结合性分两种，即左结合性(自左至右)和右结合性(自右至左)。

例如，对以下表达式：

```
1+2-3
```

操作数 2 左侧的+、右侧的-的优先级同为 4，因此考虑它们的结合性。算术运算符是左结合的，因此，2 先与其左侧的+结合，即先计算 1+2，再与-结合，执行-3 运算。这种自左至右的结合过程称为左结合性。

相对地，自右至左的结合方向称为右结合性。典型的右结合运算符是赋值运算符。

例如，对表达式：

```
int x=10,y;
y=x=20;
```

以上定义了变量 x,其初始值为 10,对表达式“y＝x＝20”,由于赋值运算符(＝)是右结合的,因此先执行“x＝20”,再执行“y＝x”,这样,执行该表达式后,变量 x 和 y 的值均为 20。

C 语言中大多数运算符都是左结合的,只有单目运算符、赋值运算符、条件运算符是右结合的。

运算符的优先级和结合性见附录 C。

3.4.1 算术运算符

C 语言的算术运算符有＋、－、＊、/、％,它们的运算规则如表 3.5 所示。

表 3.5 C 语言的算术运算符及运算规则

运算符	符号描述	操 作 描 述	优先级	结合性
＋	加	双目运算符,即应有两个操作数参与加法运算	4	左结合
－	减	双目运算符,但也可作负值运算符,此时为单目运算	4	左结合
＊	乘	双目运算符,即应有两个操作数参与乘法运算	3	左结合
/	除	双目运算符,如果参与运算的操作数均为整型,结果也为整型,舍去小数;如果操作数中有一个是实型,则结果为双精度实型	3	左结合
％	取余	双目运算符,要求参与运算的操作数均为整型,运算结果为两操作数相除后的余数	3	左结合

程序 3.7 是一个用算术运算符进行计算的例子。

【程序 3.7】 算术运算符的基本操作。

```
int main()
{
    int x=9,y=2,z1,z2,z3;
    double z4;
    z1=x+y-4;
    z2=x*y+y/5;                                    //语句①
    z3=x%y;                                        //语句②
    z4=(x+y)/2.0;                                  //语句③
    printf("%d %d %d %.2lf",z1,z2,z3,z4);
    return 0;
}
```

程序 3.7 的运行结果为:

```
z1=7 z2=18 z3=1 z4=5.50
```

本例中：

对语句①的表达式 x * y＋y/5，先计算其优先级高的 * 和/，再计算＋。其中，y/5 的值为 0，因此表达式 x * y＋y/5 的值为 18。

语句②的表达式 x％y 是 x 与 y 相除，取其余数，因此该表达式值为 1。注意，％运算符要求其两边的操作数均为整数，否则会发生编译错误。如下表达式是错误的：

```
float x=3.0,y=6.0,z;
z=x%y;                                          //本语句编译会报错
```

另外，对取余运算符，无论其两个操作数是否同号，其结果总是与左操作数的符号相同。例如：

```
45%-7 的值为 3
-45%7 的值为-3
-45%-7 的值为-3
```

语句③的表达式 (x＋y)/2.0 是将 x 与 y 相加，将其结果除以 2.0，即 11/2.0，由于/运算符的右操作数是浮点数，因此结果为双精度浮点数 5.50。

3.4.2　赋值运算符

1. 赋值运算符(=)

赋值运算符(＝)是最常用的运算符之一，由＝连接的式子称为赋值表达式，其一般形式为：

```
变量=表达式
```

以下都是合法的赋值表达式：

```
x=a+b
w=sin(a)+sin(b)
```

赋值表达式的功能是计算＝右边的表达式的值，再将其赋给左边的变量。赋值表达式的值就是赋值运算符左边的变量的值，例如：

```
z=3+5
```

以上赋值表达式先计算 3＋5 的值，然后赋给 z，此时，z 的值为 8，整个表达式的值也为 8。

需要特别注意的是，赋值运算符本质是进行了一个“写内存”的动作，因此，使用＝，可能引起其左边变量的值发生改变。

由于赋值运算的本质是写内存，因此要求其左边的操作数必须标识了一个内存空间。例如，赋值运算符的左边是一个变量名（因为变量名标识了该变量对应的内存空间），而不能是一般的表达式或常量。因此，以下赋值表达式都是不合法的。

```
x+5=10                                    //不合法的表达式
x-y=2                                     //不合法的表达式
x+y=5                                     //不合法的表达式
```

上述表达式都不合法，因为＝运算符的左边并非合法的内存空间，无法执行写内存操作。

赋值运算符具有右结合性。对以下语句：

```
int x=10,y;
y=x=20;
```

可理解为：

```
y=(x=20)
```

因此执行上述语句后，x 的值为 20，y 的值也是 20。

赋值表达式可以与其他表达式相组合，构成更复杂的表达式，例如：

```
x=(a=5)+(b=8)
```

该表达式是把 5 赋给 a，8 赋给 b，再将表达式(a＝5)和(b＝8)相加，也就是 5＋8，结果赋给 x，故 x 的值为 13。

C 语言规定，任何表达式在其末尾加上分号就构成为语句。因此，在赋值表达式的末尾加上分号，即构成一条赋值语句，例如，以下都是赋值语句。

```
x=8;
a=b=c=5;
```

2. 复合赋值运算符

在赋值运算符＝之前加上其他双目运算符，可构成复合赋值运算符。

C 语言赋值运算符有：

```
+=   -=   *=   /=   %=   <<=   >>=   &=   ^=   |=
```

复合赋值运算符的一般形式为：

```
变量 双目运算符 = 表达式
```

它等价于：

```
变量 = 变量　双目运算符　表达式
```

例如：

```
a-=8　等价于　a=a-8
x*=y+7 等价于　x=x*(y+7)
r%=p　等价于　r=r%p
```

复合赋值运算符可以看成是赋值运算的一种缩写，它的书写形式简单，可以使代码更加简洁。

3.4.3　强制类型转换

一般来说，赋值运算符左右两边操作数的数据类型应该完全相同。如果其两边的数据类型不相同，系统将进行自动类型转换，把赋值运算符右边的类型转换成左边的类型。由于这种转换是由编译器按一定的规则自动完成的，因此称为隐式类型转换[①](implicit type conversion)。

程序员也可根据需要进行显示转换，即强制类型转换，其形式为：

```
(类型声明符)(表达式)
```

强制类型转换的功能是把表达式的结果显示转换成类型声明符所表示的类型。通常，在赋值操作中，或在多种不同类型的表达式混合运算时，可根据需要进行强制类型转换。

以下是几个强制类型转换的例子：

```
int x=(int)2.5;                    //把 2.5 转换为 int 型
double y=(double)(4+9)/2;          //把(4+9)转换为 double 型
int q=(int)7.5+32;                 //把 7.5 转换为 int 型,与整数 32 进加法运算
```

3.4.4　关系运算符

在程序中经常需要比较两个操作数的大小关系，进而决定程序下一步的工作。C 语言提供了 6 种关系运算符，用于比较两个操作数的值，分别是：

```
<　<=　>　>=　==　!=
```

关系运算符的运算规则及其相关描述如表 3.6 所示。

---

① 我们不在这里详细讨论隐式类型转换的规则，因为它与编译系统紧密相关。另外，类型转换一般都伴随着数据精度的变化，因此也不建议读者不加任何干预地直接使用隐式类型转换。

表 3.6 C 语言的关系运算符

运算符	操 作 描 述	优先级	结合性
<	双目运算符,左操作数是否小于右操作数	6	左结合
<=	双目运算符,左操作数是否小于或等于右操作数	6	左结合
>	双目运算符,左操作数是否大于右操作数	6	左结合
>=	双目运算符,左操作数是否大于或等于右操作数	6	左结合
==	双目运算符,左操作数是否等于右操作数	7	左结合
!=	双目运算符,左操作数是否不等于右操作数	7	左结合

可以用关系运算符构造关系表达式。例如,如下都是关系表达式:

```
(a+b)>(c-d)        x>3/2        a!=c        j==(k+1)
```

关系表达式的值是“逻辑真”或“逻辑假”。在 C 语言中,通常用整数 1 表示“逻辑真”,用整数 0 表示“逻辑假”。程序 3.8 是一个用关系运算符进行计算的例子。

【程序 3.8】 关系运算符的基本操作。

```
int main()
{
    int x=9,y=2,z1,z2,z3,z4;
    char ch1='t',ch2='e';
    z1=(x+5)>(y+20);                                    //语句①
    z2=(x==y);                                          //语句②
    z3=(x!=y);                                          //语句③
    z4=ch1>ch2;                                         //语句④
    printf("z1=%d z2=%d z3=%d z4=%d",z1,z2,z3,z4);
    return 0;
}
```

程序 3.8 的运行结果为:

```
z1=0 z2=0 z3=1 z4=1
```

其中:

语句②首先判断 x 和 y 是否相等,因其结果为真,故将整数 1 赋给变量 z2,令 z2 的值为 1。

这里应该特别注意,一定要严格区分关系运算符(==)与赋值运算符(=)。前者用于判断两个操作数是否相等,而后者则是执行写内存的操作,两个运算符的形式上较相似,但其运算规则和结果却是本质的不同。

例如,如果将语句②改为:

```
z2=(x=y);
```

该语句先将 y 的值赋给 x，然后将赋值表达式(x＝y)的值赋给 z2。语句执行后，x、y 和 z2 的值都是 2。

程序 3.8 的语句④比较了字符变量 ch1 和 ch2 的大小，将比较的结果(这里是 1)赋给变量 z4。对字符型数据执行关系运算是合法的，比较大小时，是用 ch1 和 ch2 存储的 ASCII 码参与运算。

读者可自行分析程序 3.8 中语句①、语句③的执行情况。

C 语言中，关系运算符的运算结果是逻辑真或假，是用整数 1 或 0 来表示的，因此，一个关系表达式可以嵌入另一个表达式中，此时是用其值(1 或 0)来参与计算。例如：

```
k=(5>4)+12;
```

执行这条语句后，k 的值为 13，这是因为其中表达式(5＞4)的值为 1。

这种用法令 C 语言的表达式形式更加灵活，但也容易导致一些不合逻辑的错误。例如，用以下表达式表达分数 score 在区间[80，90]中，但该表达式是不合逻辑的：

```
90>=score>=80          //一个不合逻辑的表达式
```

对以上表达式：

(1) 如果 score 为 75，则先计算 90＞＝score 为真，其值为 1，然后计算 1＞＝80，其结果为假。

(2) 如果 score 为 150，则先计算 90＞＝score 为假，其值为 0，然后计算 0＞＝80，其结果为假。

显然，无论 score 的值为多少，该表达式的值始终为假，这是因为其左边的表达式(90＞＝score)的结果是一个逻辑值(1 或 0)，判断该逻辑值是否＞＝80，其结论必然为假。可见，该表达式并不符合逻辑，不能表达“score 既大于或等于 80，又小于或等于 90”的条件。

需要注意的是，虽然该表达式不符合逻辑，但编译器并不会报错，不过程序的运行结果很可能不符合预期。

为了描述这种多个条件并列的情况，可以借助 3.4.5 节的逻辑运算符来实现。

3.4.5　逻辑运算符

在某些问题中，仅根据一个条件不足以做出判断，需要根据多个条件来进行决策。

例如，如果学生的成绩 80≤score≤90，则评定其成绩为良好，为此，需要以下两个条件同时成立，即

```
(score>=80) 并且 (score<=90)
```

也有其他组合方式，例如，如果购票者是老人或儿童，这两个条件中任一个成立即可

享受票价优惠政策，即

```
(passenger是老人) 或者 (passenger是小孩)
```

C语言中提供了三种逻辑运算符，分别是逻辑与(&&)、逻辑或(||)、逻辑非(!)，可以用它们来构造更为复杂的逻辑表达式。逻辑运算符的运算规则及相关描述如表3.7所示。

表3.7 C语言的逻辑运算符

<table>
<tr><th>运算符</th><th>描述</th><th colspan="6">运 算 规 则</th><th>优先级</th><th>结合性</th></tr>
<tr><td rowspan="4">&&</td><td rowspan="4">逻辑与
双目运算符</td><td>a</td><td>b</td><td>a&&b</td><td>a</td><td>b</td><td>a&&b</td><td rowspan="4">11</td><td rowspan="4">左结合</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="4">||</td><td rowspan="4">逻辑或
双目运算符</td><td>a</td><td>b</td><td>a||b</td><td>a</td><td>b</td><td>a||b</td><td rowspan="4">12</td><td rowspan="4">左结合</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="3">!</td><td rowspan="3">逻辑非
单目运算符</td><td colspan="2">a</td><td colspan="2">!a</td><td>a</td><td>!a</td><td rowspan="3">2</td><td rowspan="3">右结合</td></tr>
<tr><td colspan="2">1</td><td colspan="2">0</td><td>0</td><td>1</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

1. 逻辑与(&&)

当参与运算的两个操作数同为真，其结果才为真，否则为假。例如：

```
5>0 && 4>2
```

由于5>0为真，4>2也为真，因此该表达式的结果为真。

2. 逻辑或(||)

当参与运算的两个操作数任中一个为真，其结果就为真。如果两个操作数都为假，则其结果为假。例如：

```
5>0||5>8
```

由于5>0为真，因此该表达式的结果为真。

3. 逻辑非(!)

如果操作数为真，则非运算的结果为假；如果操作数为假，则结果为真。例如：

```
!(5>0)
```

由于5>0为真，因此!(5>0)的结果为假。

注意，C语言在判断一个表达式的值是真还是假时，将“0”看成假，将“非0”看成真，