第3章 数据类型、运算符和表达式

CHAPTER 3

根据所执行任务的不同，C 语言程序经常要用到各种类型的数据，如整数、实数、字符等。不同类型的数据作为 C 语言程序的基本组成部分都是以特定的形式存储在计算机内存中的，其所占的存储空间和运算性质也不尽相同。

为方便理解数据类型的概念，这里需要先简单介绍一下计算机内存的相关知识。

计算机内存是依次逐字节排列的，一个字节(Byte，简写为 B)包含 8 位(Bit)。C 语言程序在经过编译器的编译、连接之后最终会以二进制序列的形式存储在计算机内存当中，如图 3.1 所示。

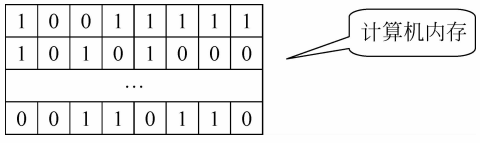

图 3.1 C 程序在内存中的存储示意图

可以看到，在图 3.1 中 C 语言程序是按照内存的结构逐字节存储的，因此 C 语言的基本存储单位是字节。图 3.1 中的每一个小方格都代表计算机内存中的一个最小存储单位——“位(Bit)”。每一位能且仅能存储一个二进制数，即“1”或“0”。在计算机中，一般用高电平代表二进制数“1”，用低电平代表二进制数“0”。

为方便用 C 语言程序对计算机内存进行数据的读写，计算机对内存进行了逐字节编址，即以字节(B)为单位，计算机内存中的第一个字节的地址是 0，之后计算机内存地址按字节递增 1，直至内存的最后一个字节。需要注意的是，计算机内存地址都是从 0 开始，但内存地址的最大值由计算机内存容量决定(不同的计算机，内存容量可能不同)。

计算机内存地址的作用有点类似于教室的房间号，学生可以通过教室的房间号找到上课教室。同样，C 语言程序可以通过内存地址找到存储在该内存单元的数据。

除了字节(B)和位(Bit)之外，常见的内存计量单位还有字(Word，简写

为 W)、千字节(KB,简写为 K)、兆字节(MB,简写为 M)、千兆字节或吉字节(GB,简写为 G)、太字节(1TB,简写为 T)等。各种单位的换算关系如下:

1B=8Bit;

1W=4B=32Bit;

1KB=2^{10}B=1024B;

1MB=2^{20}B=1024KB;

1GB=2^{30}B=1024MB;

1TB=2^{40}B=1024GB。

一个汉字一般占两个字节,假设硬盘有 500GB 的存储空间,那么可以存储的汉字个数 x 是多少呢? $x=500\times1024\times1024\times1024\div2=268\ 435\ 456\ 000$ 个。

3.1 数 据 类 型

按照数据结构中的定义,数据类型是指一个值的集合以及定义在这个值集上的一组操作。在 C 语言中,一个值的数据类型决定了这个值在计算机中的存储形式以及允许对这个值进行的操作种类。这里,以 VC++ 6.0 开发环境为例(不同的开发环境,相同数据类型的存储空间可能不同),一个整型数据(int 型)占 4 个字节的存储空间,允许对其进行加、减、乘、除、取余等运算;一个双精度浮点型数据(double 型)占 8 个字节的存储空间,允许对其进行加、减、乘、除运算,不允许进行取余运算。

为方便理解数据类型的必要性,抛开数据类型允许的操作不提,只考虑其存储情况,读者就可以明白数据类型的重要。

因为 C 语言程序运行时需要的数据都以字节为单位存储在内存中,而计算机的内存是有限的,所以为了有效地利用和管理内存,系统必须根据数据的类型给它分配内存单元,即不同类型数据所占的内存空间不尽相同。这就好比归纳物品时为节约空间,大的物品要放到大盒子里,小的物品要放到小盒子里一样。

C 语言中常见的数据类型如图 3.2 所示。

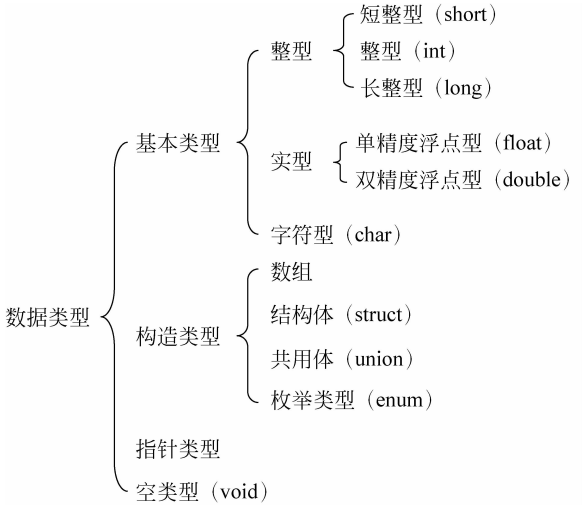

图 3.2 C 语言的数据类型

如图 3.2 所示，C 语言中常见的数据类型有以下 4 种。

(1) 基本类型：最基础的简单数据类型，其值无法再分解为其他类型。

(2) 构造类型：顾名思义是根据已定义的一个或多个数据类型用构造的方法来定义。构造数据类型是由多个其他数据类型组合而成的，所以一个构造类型的值可以分解成若干个“成员”或“元素”，其中每个成员要么是基本数据类型，要么是又一个构造类型。

(3) 指针类型：C 语言中比较重要、比较难理解的内容，在后面会详细讲解。

(4) 空类型：表示没有类型，常与函数、指针等相关内容结合使用。

3.2　常量与变量

按照取值是否可变，基本数据类型有常量和变量两种形式。

在程序执行过程中其值不变的量称为常量，其值可变的量称为变量。

常量和变量一般要与数据类型结合使用，如整型常量、整型变量、实型常量、实型变量、字符常量、字符变量等。

在程序中常量是可以不经说明而直接引用的。

变量必须先定义后使用。

例如：

```
int a;                          /* 定义了一个整型变量 a */
a=5;                            /* 将整型常量 5 赋给整型变量 a */
```

在上例中，整型常量 5 不用说明可以直接引用，整型变量 *a* 必须先定义再使用。如果对变量 *a* 不加说明直接使用，则程序编译时会报错，因为系统不知道 *a* 是什么。

3.2.1　标识符

标识符在 C 语言中的作用与人的姓名类似，是用来标识变量名、符号常量名、函数名、数组名、类型名、文件名的有效字符序列。

C 语言规定标识符只能由字母、数字和下画线三种字符组成，且第一个字符必须为字母或下画线。

C 语言对标识符有以下分类。

(1) 关键字：C 语言已经预先定义了一批标识符，它们在程序中都代表着固定的含义，不能另作他用，这些标识符称为关键字，如用来说明变量类型的关键字 int、double、char 及 if 语句中的 if、else 等，它们作为 C 语言的关键字不能再用作变量名或函数名，否则程序编译时会报错。

(2) 预定义标识符：指在 C 语言中预先定义并具有特定含义的标识符，如 C 语言提供的库函数名称(printf、scanf 等)和预编译处理命令(define、include 等)。C 语言允许把这类标识符重新定义另作他用，但这将使这些标识符失去预先定义的原意，因此为避免误解，建议用户不要把这些预定义标识符另作他用。

(3) 用户标识符：由用户根据需要自定义的标识符称为用户标识符，又称自定义标

识符。例如：

```
a、sum、_pointer、Array123、point_a1;        /* 合法自定义标识符 */
1b、string.a、#89、total-a、int.             /* 非法自定义标识符 */
```

在使用自定义标识符时有以下几点需要注意：

(1) C 语言区分大小写，同一字母的大写字母和小写字母对 C 编译器而言是两个不同的字符。为了与日常习惯保持一致，增加程序的可读性，变量名一般用小写字母表示。

(2) ANSIC 未规定标识符的长度，但为了提高程序的可移植性和阅读方便，变量名不宜过长(不同的编译器有自己的规定，一般不超过 32 个字符)。

(3) 如果用户标识符与关键字相同，则程序编译时系统报错。如果用户标识符与预定义标识符相同，系统不报错，只是预定义标识符将失去原定含义，代之以用户确认的含义，这可能会引发一些程序运行错误。

(4) 在自定义标识符时要尽量做到“见名知意”，建议优先使用有含义的英文单词或缩写作为标识符。

3.2.2 常量和符号常量

常量作为数据的一种，从字面形式上即可判别，因此也叫字面常量。常量也分数据类型，如整型常量、实型常量、字符常量和字符串常量等。

例如：

整型常量：－3、0、3；

实型常量：－0.33、4.56；

字符常量：'a' 'b' 'c' 'd'；

字符串常量："aBcdEfg""Hello"。

除以上形式外，在 C 语言中还允许用一个标识符来表示一个常量，称为符号常量。

符号常量在使用之前必须先定义，其定义格式如下：

```
#define 标识符   常量
```

其中，define 是一条预处理命令(以“#”开头)，称为宏定义命令(第 6 章将详细介绍)，其功能是用标识符表示后面的常量。

例如：

```
#define PI 3.14
```

在符号常量 PI 定义后，程序中出现的所有 PI 都表示常量 3.14。为方便与变量标识符区分，符号常量的标识符习惯使用大写字母，变量标识符使用小写字母。

在使用符号常量时一定要注意符号常量与字符常量、字符串常量的区别。

例如：

```
#define PI 3.14                  /* PI 为符号常量，代表 3.14 */
printf("PI");                    /* "PI"为字符串常量，表示一串字符 */
```

例 3.1　符号常量的使用，计算半径为 2 的圆的面积。

程序如下：

```
#define PI 3.14                                  /*定义符号常量 PI,代表 3.14*/
#include<stdio.h>
int main(void)
{
    int r=2;                                     /*定义 int 型变量 r,表示圆的半径*/
    float area;                                  /*定义 float 型变量 area,表示圆的面积*/
    area=PI*r*r;                                 /*计算圆面积*/
    printf("半径为 2 的圆面积 area=%d\n",area);      /*输出圆面积*/
    return 0;
}
```

程序运行结果如下：

```
半径为2的圆面积为:
12.560000
```

程序第 1 行用宏定义 #define 定义了符号常量 PI，用于代表常量 3.14（圆周率 π），之后本程序中出现的 PI 都表示常量 3.14，它可以像常量一样直接参与运算。

使用符号常量的优点及注意事项如下：

（1）含义清楚，见名知意。如例 3.1 的程序中一看 PI 就可以知道它代表圆周率 π。

（2）一改全改，在需要改变某个常量时只需修改其对应的符号常量，程序中所有用到它的地方也就做了相应修改。在例 3.1 中，如果需要改变圆周率 π（由 3.14 变为 3.141 592 6），使计算结果更精确，只需修改符号常量的定义即可。

```
#define PI 3.1415926                             /*修改符号常量的定义,将 π 改为 3.1415926*/
```

（3）符号常量与变量不同，符号常量一经定义就不能再对其进行赋值操作。

例如：

```
#define PI 3.1415
PI=5;                                            /*错误,符号常量不可赋值*/
```

3.2.3　变量

顾名思义，变量就是可以改变的量。在程序运行期间，变量的值是可以改变的。

变量存储于内存当中，因此变量其实就是内存中的一个具有特定属性的存储单元，用于存放数据，而这个数据就是变量的值。

在 C 语言中，所有的变量都必须“先定义，后使用”。

在定义变量时需要给变量指定一个名称（变量名），以便程序引用该变量。此外，变量的定义还要指定变量的数据类型，明确所定义的变量用于存储什么类型的数据。

例如：

```
int x;                                           /*基本整型变量的定义*/
```

上述语句定义了一个变量，变量名为 x，变量的数据类型是基本整型（int 型），这说明

变量 x 对应的内存单元可存储 int 型数据。

若假设 x 的值为 1，则变量 x 的存储情况如图 3.3 所示。

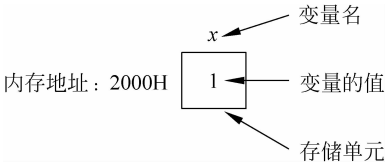

图 3.3 变量存储示意图

如图 3.3 所示，x 是变量名，1 是变量的值，2000H 是该变量对应的存储单元在内存中的地址（H 表示 2000 是十六进制数。内存地址常用十六进制或十进制表示）。

C 编译器在对 C 程序进行编译、连接时会给每一个变量都分配一个内存单元，用于存储变量的值，而这个内存单元的地址就是变量名。这样，从变量中取值的过程实际上就是通过变量名得到内存地址，并从该内存单元读取数据的过程。

简单来说，变量名其实就是以一个名字代表一个内存地址。因此，对程序而言，变量名就是该变量在内存中的地址。程序通过变量名可以在内存中找到并引用该变量的值。

变量的定义一般放在函数体的开头部分。通过变量定义（也叫变量声明）编译器就建立了变量符号表。在此之后，程序会用到哪些变量，每个变量在哪里，它们都是什么数据类型，编译器就都掌握了。如果某个变量未定义就使用，编译器就无法了解该变量的信息，在程序编译时会报错。

对于变量的使用有以下几点需要注意：

（1）变量在定义时必须指定数据类型，以便编译器为其分配适合的存储空间。以 VC++ 6.0为例，int 型数据占 4 个字节的内存，char 型数据占一个字节的内存，float 型数据占 4 个字节的内存，double 型数据占 8 个字节的内存。

（2）不同数据类型的变量，其运算性质是不同的。例如，C 语言允许对整数进行取余运算，如果 x 和 y 都是整型变量，则“$x\%y$”是正确的；如果 x 和 y 都是或其中之一是实型变量，则“$x\%y$”是错误的，程序在编译时会报错。

（3）对于未定义的变量，编译器无法识别。

（4）变量名和变量的值是不同的概念。

例如：

```
int x;
x=10;
```

在上述语句中，x 是变量名，代表变量的内存地址，10 是变量值，是保存在变量中的数据。

3.3 C 语言的常用数据类型

3.3.1 整型数据

1. 整型常量

整型常量就是整常数，包括正整数、负整数和 0。正整数可以在前面加“+”，也可以

不加。负整数要在前面加“－”。

在 C 语言中，整型常量可以用八进制、十六进制和十进制三种形式表示。为方便识别，C 语言对整型常量做了以下分类。

（1）十进制整型常量：无前缀，数码取值为 0～9。

例如：

```
237、－568、65535、1627;                    /* 正确的十进制整型常量表示 */
023、23D.                                   /* 错误的十进制整型常量表示 */
```

（2）八进制整型常量：必须以“0”开头，即以“0”作为八进制数的前缀，数码取值为 0～7。八进制数通常是无符号数。

例如：

```
015、0101、0177777;                         /* 正确的八进制整型常量表示 */
256、03A2、－0127.                          /* 错误的八进制整型常量表示 */
```

（3）十六进制整型常量：前缀为“0X”或“0x”，数码取值为 0～9、A～F 或 a～f。

例如：

```
0X2A、0XA0、0XFFFF;                         /* 正确的十六进制整型常量表示 */
5A、0X3H.                                   /* 错误的十六进制整型常量表示 */
```

C 语言程序是根据前缀来区分进制的，因此用户在书写常数时不要把前缀弄错，以免结果不正确。

2. 整型变量

根据在内存中所占存储空间的不同，整型变量可分为基本整型、短整型、长整型和无符号型。其中，无符号整型变量只能存放不带符号的正整数，不能存放负数。无符号整型变量又可分为无符号基本整型、无符号短整型和无符号长整型。

各种整型变量的类型符号、所占存储空间、取值范围以及可表示的数据个数如表 3.1 所示（以 VC++ 6.0 为例）。

表 3.1　整型变量分类情况表

数据类型	类型符号	所占字节数（位数）	取值范围	可表示的数据个数
基本整型	int	4(32)	－2 147 483 648～2 147 483 647，即$(-2^{31}) \sim (2^{31}-1)$	2^{32}＝4 294 967 296＝1GB
短整型	short	2(16)	－32 768～32 767，即$(-2^{15}) \sim (2^{15}-1)$	2^{16}＝65 536
长整型	long [int]	4(32)	－2 147 483 648～2 147 483 647，即$(-2^{31}) \sim (2^{31}-1)$	2^{32}＝4 294 967 296＝1GB
无符号基本整型	unsigned[int]	4(32)	0～4 294 967 295	2^{32}＝4 294 967 296＝1GB

续表

数据类型	类型符号	所占字节数（位数）	取值范围	可表示的数据个数
无符号短整型	unsigned short [int]	2(16)	0～65 535	$2^{16}=65\ 536$
无符号长整型	unsigned long [int]	4(32)	0～4 294 967 295	$2^{32}=4\ 294\ 967\ 296=1GB$

在表3.1中，用中括号“[]”括起来的内容表示可以省略，即long int可简写为long，unsigned int可简写为unsigned，unsigned short int可简写为unsigned short，unsigned lont int可简写为unsigned long。

整型变量的一般定义格式如下：

```
类型说明符  变量名;
```

其中，类型说明符用于说明整型变量的具体类型，后面跟变量名，如果同时定义多个变量，则多个变量名之间需要用“,”隔开。

例3.2 整型变量定义示例。

```
int x, y, z;                          /*定义基本整型变量x、y、z*/
short a;                              /*定义短整型变量a*/
long b;                               /*定义长整型变量b*/
unsigned c;                           /*定义无符号基本整型变量c*/
unsigned short d;                     /*定义无符号短整型变量d*/
unsigned long e;                      /*定义无符号长整型变量e*/
```

例3.3 整型变量的使用。

程序如下：

```
#include<stdio.h>
int main(void)
{
    int a,b,sum;                      /*定义基本整型变量a、b、sum*/
    unsigned c;                       /*定义无符号基本整型变量c*/
    a=3;                              /*分别给a、b、c赋值*/
    b=-4;
    c=5;
    sum=a+b+c;                        /*对a、b、c三值求和，并赋给变量sum*/
    printf("a、b、c之和为:\n%d\n",sum);   /*输出计算结果*/
    return 0;
}
```

程序运行结果如下：

```
a、b、c之和为:
4
```

在使用整型变量和整数变量时有以下几点需要注意：

(1) 整型常量后面可以加后缀，以表示常量的数据类型，如 235L 或 2351，表示常量 235 为长整型数据；345U 或 345u，表示 345 为无符号型数据。

(2) 给一个整型变量赋的值不应超过该整型变量规定的取值范围，否则会溢出。

例如：

```
short x;                    /*定义一个短整型变量 x*/
x=66656;                    /*非法，数据溢出，短整型取值范围为－32 768～
                              32 767*/
```

(3) 整型数据在内存中是以补码形式表示的，一个正整数的补码和它的原码(即该数的二进制表达形式)相同。一个负整数的补码是将该数的绝对值的二进制形式按位取反再加 1。在整型数据的存储单元中，最左边的一位(最高位)是符号位，“0”表示正、“1”表示负。

例如：

	符号位															
5 的补码：	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
–5 的补码：	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

(4) 无符号整型数据没有符号位，它的存储单元的最高位也用于存储数据。

3.3.2　实型数据

在 C 语言中只使用十进制的实型数据(实数)，也叫浮点型数据。

在 C 语言中浮点型数据有以下两种表示形式。

- 小数形式：如 0.0、27.0、5.89、4.13、－5.2、300.、－267.8230 等。在使用这种方式表示浮点型数据时可以没有整数部分或小数部分，但必须有小数点，如 1.25、1.0、.14、3. 都是正确形式。
- 指数形式：即利用科学记数法表示实数，如 3.65e3 表示 3.65×10^3(即 3650)，－1.35e－2表示 -1.35×10^{-2}(即－0.0135)。在使用这种方式表示浮点型数据时 e(或 E)之前必须有数字，且 e(或 E)后面的指数必须为整数，不能写成 e3 或 1e.5 等形式。

整数在计算机中的存储比较简单，而浮点型数据在计算机中的存储却相对麻烦。在计算机中，浮点型数据都是以指数形式存储在内存中的。

以实数 1.234 56 为例，其在内存中的存储情况如下。

＋	.123 456	1
符号位	小数部分	指数

＋　　.123 456　　$\times10^1$　　即 1.234 56

为方便读者理解，这里使用十进制数描述了浮点型数据在内存中的存储情况，而在实际的计算机中小数部分是用二进制数表示的，指数部分是用 2 的幂次表示的。

对浮点型数据的存储而言，C 语言的标准并未规定，小数部分应占多少位，指数部分应占多少位，不同编译环境的规定可能不同。但在大多数的编译环境中，小数部分（包括符号位）占 24 位，指数部分占 8 位。小数部分占的位数越多，实数的有效数字越多，精度越高。指数部分占的位数越多，实数的取值范围就越大。

1. 浮点型变量

C 语言中的浮点型变量可分为单精度（float 型）、双精度（double 型）和长双精度（long double 型）。

浮点型数据所占的字节数、有效数字和取值范围在不同的编译环境中可能不同。在多数编译环境中，单精度浮点型数据占 4 个字节，有效数字为 7 位，取值范围为 -3.4×10^{38}～3.4×10^{38}；双精度浮点型数据占 8 个字节，有效数字为 16 位，取值范围为 -1.7×10^{308}～1.7×10^{308}；长双精度浮点型数据所占的字节数可能是 8 字节、10 字节、12 字节或 16 字节，有效数字从 15 到 19 位不等，取值范围为 -1.2×10^{4932}～1.2×10^{4932}。由于 long double 型数据很少应用，故不做详述。

例如：

```
float x;                                    /*定义单精度浮点型变量 x*/
double y;                                   /*定义双精度浮点型变量 y*/
```

需要注意，因为浮点型变量是由有限的存储单元组成的，所以它能提供的有效数字是有限的。在计算时，有效位以外的数字将被舍去，这可能会产生一些计算上的误差，即舍入误差。

2. 浮点型常量

在 C 语言中，系统一般将浮点型常量按双精度浮点数处理。

例 3.4 浮点型数据的舍入误差示例。

程序如下：

```
#include<stdio.h>
int main(void)
{
    float x;                                /*定义单精度变量 x*/
    double y;                               /*定义双精度变量 y*/
    x=1.234*3.23456;                        /*将两个浮点型常量的乘积赋给 x*/
    y=1.234*3.23456;                        /*将同样的乘积赋给 y*/
    printf("x=%f\ny=%.10f\n",x,y);          /*分别输出 x 和 y*/
    return 0;
}
```

程序运行结果如下：

```
x=3.991447
y=3.9914470400
```

在上例中，系统先把两个浮点型常量 123.456 和 2.3456 当成双精度浮点数相乘，乘积(3.99144704)也是一个双精度浮点数，然后将乘积分别赋给变量 x 和 y。由于单精度变量只能接收 7 位有效数字，所以系统从乘积中取 7 位有效数字(3.991 447)赋给变量 x，乘积的最后两位“04”被舍去，导致舍入误差。而 y 是双精度变量，可接收的有效数字为 16 位，所以 y 可以接收整个乘积(3.991 447 04)。

需要注意，浮点型常量按双精度浮点数处理虽然精度高，但运算速度较慢。为了提高运算速度，可以在浮点型常量后面加后缀 f 或 F，此时系统会将其当成单精度浮点数处理。

例如：

```
x=1.234f*3.23456f;                    /*将两个单精度浮点型常量的乘积赋给变量 x*/
```

3.3.3　字符型数据

1. 字符常量

在 C 语言中，字符常量是指用单引号括起来的一个字符。凡是键盘可以正常输入的字符均可作为字符常量，如'a' 'b' '+' '=' '?' '0'等。如果没有单引号"，系统会将其当成变量或其他有名字的对象。

在 C 语言中字符常量有以下特点：

(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，例如'a'是字符常量，而"a"是一个字符串(后面会介绍)。

(2) 字符常量只能是单个字符，如'abc'是非法的。

(3) 字符可以是字符集中的任意字符。

(4) 数字被定义为字符型之后不宜参与算术运算，否则其运算结果和预期结果会截然不同，如'6'是字符常量，它如果参与算术运算('6'+6)，其结果不是 12 而是 60，因为'6'在进行加法运算时使用的值是它的 ASCII 码值 54，所以('6'+6)相当于 54+6，详见例 3.5。

(5) 大写字母和小写字母代表不同的字符常量，如'A'和'a'是不同的字符常量。

(6) 单引号中的空格也是一个字符常量，但不能写成''(中间没有空格)的形式。

(7) 字符常量在内存中占一个字节，存放的是字符的 ASCII 码值。在 C 语言中，所有的字符型常量(变量)和整型常量(变量)是通用的，其 ASCII 码值就是对应的整数值。

例 3.5　字符型常量和整型常量示例。

程序如下：

```
#include<stdio.h>
int main(void)
{
    char x,y;                         /*定义两个字符型变量 x 和 y*/
    int z;                            /*定义 int 型变量 z*/
    x='M';                            /*分别给 x 和 y 赋值*/
    y='m';
    z='6'+6;
    printf("x=%c,y=%c\n",x,y);        /*以字符形式输出 x 和 y*/
```

```
    printf("x=%d,y=%d\n",x,y);          /*以整数形式输出 x 和 y*/
    printf("z=%c\n",z);                 /*以字符形式输出 z*/
    printf("x=%d\n",z);                 /*以整数形式输出 z*/
    return 0;
}
```

程序运行结果如下：

```
x=M,y=m
x=77,y=109
z=<
x=60
```

在上例中，程序第 6 行和第 7 行分别给字符型变量 x 和 y 赋值为'M'和'm'，因为字符型数据和整型数据是通用的，所以字符型数据也可以进行算术运算，参与运算的操作数为字符型数据的 ASCII 码值。

程序第 8 行将'6'+6 的和赋给了变量 z，在这个加法运算中，第 1 个操作数是字符'6'的 ASCII 码值 54，第 2 个操作数是整型常量 6，两者之和为 60。

程序第 9 行以字符形式输出了变量 x 和 y 的值。

程序第 10 行以整数形式输出了变量 x 和 y 的值，其值为对应的字符型数据'M'和'm'的 ASCII 码值。同一个字母(仅限 26 个英文字母)的小写形式的 ASCII 码值比大写形式的 ASCII 码值大 32。

程序第 11 行以字符形式输出的变量 z 是字符'<'，其 ASCII 码值为 60。

程序第 12 行以整数形式输出的是变量 z 的 ASCII 码值 60。

2. 转义字符

除了上述普通的字符常量之外，C 语言还允许使用转义字符。转义字符是一种特殊形式的字符常量，是以反斜线“\”开头的字符序列，如\n、\t、\f 等。

转义字符具有特定的含义，主要用来表示用一般字符不便于表示的控制代码，如表 3.2 所示。

表 3.2 C 语言常用的转义字符及其含义

转义字符	含　义	ASCII 码值
\n	按回车键换行	10
\t	水平制表(横向跳到下一制表位置)	9
\v	竖向制表(竖向跳到下一制表位置)	9
\b	退格	8
\r	按回车键	13
\f	走纸换页	12
\\	代表一个反斜杠字符	92
\'	代表一个单引号字符	39
\"	代表一个双引号字符	34
\0	代表空字符(NULL)	0
\ddd	1～3 位八进制数代表的字符	
\xdd	1～2 位十六进制数代表的字符	

在使用转义字符时需要注意以下几点：

(1) 转义字符是常量，只能用小写字母表示，且每个转义字符只代表一个字符，如'\n' '\101'只代表一个字符。

(2) 反斜线后的八进制数可以不用 0 开头，如'\101'。

(3) 反斜线后的十六进制数只可由小写字母 x 开头，不允许用大写字母 X，也不能用 0x 开头，如'\x41'。

(4) C 语言中的任何一个字符均可用转义字符来表示，表 3.2 中的\ddd 和\xhh 就是为此提出的。比如'\101'表示的是 ASCII 码值为八进制数 101 的字符，因为八进制数 101 对应十进制数 65，因此'\101'相当于 ASCII 码值为 65 的字符'A'，即'\101'与'A'等价。再如'\x41'表示的是 ASCII 码值为十六进制数 41 的字符，因为十六进制数 41 对应的十进制数也是 65，所以'\x41'也相当于 ASCII 码值为 65 的字符'A'，即'\x41'也与'A'等价。换而言之，'\101' '\x41' 'A'三者等价。

(5) 部分转义字符操作的屏幕显示结果和打印机显示结果不同，如\v 和\f 对屏幕无影响，而打印机则会执行相应操作。

3. 字符串常量

C 语言允许使用字符串常量。字符串常量是由双引号括起来的一串字符，如"Hello!" "My name is Henry!" "a" "3.15"等。

在字符串常量的双引号中允许插入任何转义字符，如"I am \097 boy.\n" "\\abc"等。

例 3.6　输出字符串常量。

程序如下：

```
#include<stdio.h>
int main(void)
{
    printf("I am a boy!\n");                    /*字符串中包含转义字符\n*/
    printf("I am \141 boy!\n");                 /*字符串中包含转义字符\141 和\n*/
    printf("I am \x61 boy!\n");                 /*字符串中包含转义字符\x61 和\n*/
    return 0;
}
```

程序运行结果如下：

```
I am a boy!
I am a boy!
I am a boy!
```

上述程序包含三个字符串输出语句。每个语句的内容都不同，但输出结果相同，请读者自行分析原因。

C 语言规定以字符'\0'作为字符串常量的结束标志，即系统会在每个字符串的最后自动加入一个字符'\0'作为字符串的结束标志。

这里以字符串"Welcome"为例，它在内存中的存储情况如下。

W	e	l	c	o	m	e	\0

因此，字符常量和字符串常量是不同的，两者主要存在以下不同：

（1）表达形式不同，字符常量用单引号括起来，而字符串常量用双引号括起来，如'a'是字符常量，而"a"是字符串常量。

（2）存储长度不同，字符常量固定占用一个字节的存储空间，而字符串常量占用的字节数为字符串中的字符个数加 1。如上例中，字符串常量"Welcome"有 7 个字符，但是它占用 8 个字节的存储空间，因为最后一个字节要用于存储字符串结束标志"\0"。

（3）单纯的一对双引号" "也是一个字符串常量，称为空串，它也要用一个字节的存储空间存放字符串结束标志'\0'。

（4）可以把一个字符常量赋给一个字符变量，但不能把一个字符串常量赋给一个字符变量。在 C 语言中没有字符串变量的概念，这和 BASIC 语言是不同的。C 语言是用字符数组来存放字符串常量的，这部分内容将在第 5 章中详细介绍。

4. 字符变量

字符变量用于存储字符常量，能且仅能存放一个字符。

字符变量的类型说明符是“char”，字符变量的定义格式和书写规则与整型、浮点型变量相同。

例如：

```
char x, y;                  /*定义字符变量 x 和 y*/
x='a';                      /*给字符变量 x 赋值*/
y='A';                      /*给字符变量 y 赋值*/
```

上述语句定义了字符变量 x 和 y 并进行了赋值。变量 x 和 y 能且仅能存放一个字符。变量 x 被赋予'a'，其值为字符'a'的 ASCII 码值；变量 y 被赋予'A'，其值为字符'A'的 ASCII 码值。

前面说过，在 C 语言中字符型数据和整型数据是通用的，字符变量可以进行任何整型变量允许的运算。在编程时可以给整型变量赋字符值，也可以给字符变量赋整型值（ASCII 码值）。在变量输出时可以把字符变量当成整型数据输出，输出字符的 ASCII 码值，也可以把整型值（ASCII 码值）当成字符数据输出，输出该 ASCII 码值对应的字符，见下例。

例 3.7 字符变量示例。

程序如下：

```
#include<stdio.h>
int main(void)
{
    char x,y,z;             /*定义三个字符型变量 x、y、z*/
    x='A';                  /*将字符常量 A 赋给变量 x*/
    y=x+32;                 /*计算 x+32 的结果并将其赋给字符变量 y*/
    z=65;                   /*将十进制数 65 赋给字符变量 z*/
    printf("x=%c,y=%c,z=%c\n",x,y,z);      /*以字符形式输出 x、y 和 z*/
    return 0;
}
```

程序运行结果如下：

```
x=A,y=a,z=A
```

请读者结合前面的内容认真分析程序，思考程序的运行结果。

3.3.4　为变量赋初值

在 C 语言中有时需要给一些变量赋初值，C 语言允许在定义变量的同时为其赋初值。例如：

```
int x=5, y=6;              /*定义基本整型变量 x 和 y,并分别给它们赋初值 5 和 6*/
float m=3.0;               /*定义单精度变量 m,初值为 3.0*/
char ch1='A';              /*定义字符变量 ch1,初值为'A'*/
```

C 语言允许为定义的一部分变量赋初值。

例如：

```
int x, y, z=3;             /*定义基本整型变量 x、y 和 z,但仅对 z 赋初值 3*/
```

如果想把一个数同时赋给多个变量，应写成以下形式：

```
int x=1, y=1, z=1;         /*将变量 x、y、z 的初值均赋为 1*/
```

注意，上述语句绝不能写成以下形式：

```
int x=y=z=1;               /*非法*/
```

需要说明的是，给变量赋初值的操作不是在程序编译阶段完成的，而是在程序运行阶段完成的，即在程序编译时编译器知道整型变量 x 的存在，但此时变量 x 没有变量值。变量 x 的初值是在程序运行时赋予的。

例如：

```
int x=1;
```

其实相当于：

```
int x;
x=1;
```

3.4　不同数据类型间的转换

3.4.1　混合运算中的数据类型转换

C 语言允许整型、单精度型、双精度型和字符型数据进行混合运算。

当不同类型的数据进行混合运算时，需要先按照一定的顺序（如图 3.4 所示）将不同类型的数据转换成同一种类型的数据，然后再进行运算。

具体的数据转换工作由编译器自动完成，转化规则如下：

(1) 混合运算时首先进行水平方向的转换，这种水平方向的转换是必须进行的，如

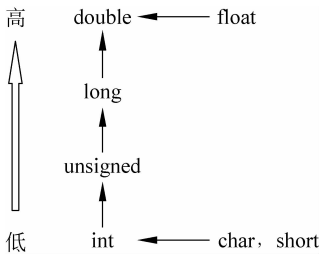

图 3.4 不同数据类型混合运算时的转换顺序

图 3.4所示，char 和 int 处于同一水平。如果两个 char(short)变量想进行运算，必须先转换成 int 型的变量，然后再运算。同理，float 型数据的运算也要先转换成 double 型数据再运算。

(2) 在进行水平方向的转换后，若仍存在不同类型数据，则按照垂直方向由低到高继续转换，直至所有数据类型一致。例如，int 型数据和 double 型数据进行运算，需要先把 int 型数据按从低到高的方向转换成 double 型数据，然后再运算。int 型数据和 float 型数据进行运算，均需转换成 double 型数据才能运算。

例 3.8 C 语言混合计算示例。

程序如下：

```
#include<stdio.h>
int main(void)
{
    int a=1;
    float b=2.0;
    double c=3.0,y;
    long d=6;
    y=1+'m'+a*b-d/c;
    printf("%f\n",y);
    return 0;
}
```

程序运行结果如下：

```
110.000000
```

上例在求 y 值时，编译器对表达式“1+'m'+$a*b-d/c$”的执行是从左到右的，具体运算过程如下：

(1) 先计算 1+'m'，'m'转换成整数 109，求得两者之和为 110。

(2) 因为“*”比“+”的优先级高，所以先计算 $a*b$，因为 a 为 int 型、b 为 float 型，所以 a 和 b 都要转换成 double 型，两者之积也为 double 型。

(3) 将 110 与 $a*b$ 的积相加，计算“110+$a*b$”，其结果为 double 型。

(4) 因为 c 为 double 型，所以要将变量 d 转换成 double 型之后再计算 d/c，其结果也是 double 型。

(5) 将“110+$a*b$”与“d/c”相减，计算结果为 double 型。

需要注意的是，上述运算过程中的转换都是系统自动完成的。

3.4.2 赋值运算中的数据类型转换

C 语言规定，在赋值运算中若赋值运算符“=”两边的数据类型不一致，且都是数值型或字符型，将进行数据类型转换，此时会将赋值运算符右边的数据类型转换成左边的数据

类型。具体规则如下：

(1) 浮点型数据赋予整型变量，浮点型数据的小数部分将被舍去，而不是四舍五入。例如，x 为整型变量，执行 $x=3.98$ 的赋值运算，赋值结果为 $x=3$，浮点数 3.98 的小数部分被舍去(注意不是四舍五入)。

(2) 整型数据赋予浮点型数据，数值不变，但将以浮点数的形式存放，即增加小数部分，且小数部分的值为 0。例如，x 为 float 型变量(7 位有效数字)，将整数 10 赋给 x，则先转换成 10.00000，然后再赋给变量 x，$x=10.00000$。

(3) float 型数据赋给 double 型变量，数值不变，但有效位由 7 位扩至 16 位，占据的存储空间也由 4 字节扩至 8 字节。

(4) double 型数据赋给 float 型变量时会产生舍入误差，系统取 double 型数据的前 7 位有效数字存入 float 型变量(前提是取值范围不能溢出)，其余数字舍去。

例如：

```
float x=1e20;              /*正确*/
floaty=1e100;              /*非法,数据溢出*/
```

(5) 在将 char 型数据赋予 int 型变量时，将 char 型数据的 ASCII 码值以二进制的形式放入 int 型变量的低 8 位中，左侧高位均为 0。

(6) 在将 int、short、long 型数据赋给 char 型变量时，只将数据的低 8 位作为 ASCII 码值存入 char 型变量，剩余高位的内容都舍去。

(7) 在将 short 型数据赋给 int、long 型变量时(以 VC++ 6.0 为例)需进行符号扩展。先将 short 的 16 位二进制数送入 int、long 型变量的低 16 位，然后进行符号扩展，若 short 型数据为正，则 int、long 型变量的高位补 0，否则补 1。

(8) 在将 unsigned short 型数据赋给 int、long 型变量时不存在符号扩展问题，只需将高位补 0 即可。

(9) 在将非 unsigned 型数据赋给长度相同的 unsigned 型变量时也是原样赋值，即符号位也作为数值一起传送。

以上转换规则看起来有些烦琐，其实总结起来就是一点，即赋值时的类型转换其实就是按存储单元中的存储形式直接传送。

3.4.3　强制转换

前面两种数据转换是由编译器自动完成的。除此之外，C 语言还允许对数据进行强制转换。

强制转换的作用是利用强制转换运算符“( )”把一个表达式强制转换成需要的数据类型，一般格式如下：

(类型说明符)(表达式)

例如：

```
(float) a                  /*把变量 a 强制转换为 float 型*/
(int) (x+y)                /*把 x+y 的结果强制转换为整型*/
```

在使用强制转换时应注意以下两点：

(1) 类型说明符和表达式都必须加括号，单个变量可以不加括号。若把(int)($x+y$)写成(int)$x+y$，则相当于把 x 转换成 int 型后再与 y 相加。

(2) 强制转换只是为了运算需要对数据进行的临时性转换。在强制转换后，程序会得到一个所需数据类型的临时变量，而原有变量的数据类型并未改变。

例 3.9 强制转换示例。

程序如下：

```
int main(void)
{
    float x=3.14;
    printf("(int)x=%d\n",(int)x); /* float 型变量 x 被强制转换后得到一个 int 型临时变量 */
    printf("x=%f\n",x);           /* 被强制转换的 float 型变量 x 本身并未改变 */
    return 0;
}
```

程序运行结果如下：

```
(int)x=3
x=3.140000
```

由上例可知，float 型变量 x 在程序的第 4 行中被强制转换，得到一个临时的 int 型数据并输出，而 x 原有的数据类型并未改变，仍为 float 型，如程序运行结果的第 2 行所示。

3.5 运算符和表达式

除控制语句和输入输出以外，C 语言几乎把所有的基本操作都作为运算符处理。因此，C 语言具有极为强大的运算功能，运算符种类很多，如算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、条件运算符、逗号运算符、指针运算符、求字节数运算符、强制类型转换运算符、分量运算符、下标运算符等。

各种运算符的数量及作用分别如下。

(1) 7 种算术运算符：用于各类数值运算，包括加“+”、减“-”、乘“*”、除“/”、求余“%”、自增“++”、自减“--”。

(2) 6 种关系运算符：用于比较运算，包括大于“>”、小于“<”、等于“==”、大于等于“>=”、小于等于“<=”和不等于“!=”。

(3) 3 种逻辑运算符：用于逻辑运算，包括与“&&”、或“||”、非“!”。

(4) 6 种位运算符：对参与运算的数据按二进制位进行运算，包括按位与“&”、按位或“|”、按位取反“~”、按位异或“^”、左移“<<”、右移“>>”。

(5) 11 种赋值运算符：用于赋值运算，包括简单赋值运算符“=”，复合算术赋值运算符“+=”“-=”“*=”“/=”“%=”，复合位运算赋值符“&=”“|=”“^=”“>>=”“<<=”。

(6) 条件运算符“?:”：C 语言中唯一的三目运算符，用于条件求值。

(7) 逗号运算符“,”：用于把两个表达式连在一起构成一个表达式。

(8) 指针运算符：包括取内容运算符“*”和取地址运算符“&”。

(9) 求字节数运算符“sizeof”：用于计算数据类型所占的字节数。

(10) 强制类型转换运算符“()”：用于将表达式的值强制转换成需要的数据类型。

(11) 分量运算符“.->”：用于结构指针指向成员名的操作，也叫指向结构体成员运算符。

(12) 下标运算符“[]”：用于引用数组元素。

(13) 其他：如函数调用运算符“()”等。

在 C 语言中使用运算符时需要考虑运算符的优先级和结合性问题。在表达式中，各运算量参与运算的先后顺序不仅要遵守运算符优先级别的规定，还要受运算符结合性的制约。这种结合性是其他高级语言的运算符所没有的，这也增加了 C 语言的复杂性。

在 C 语言中，表达式的涵盖范围非常广，它可以是任何计算结果为数值的东西，包括简单表达式和复杂表达式。

简单变量、常量、符号常量都是简单表达式，也叫最小表达式，是表达式求值的最小单位。例如，简单变量 x、整型常量 5、字符常量'a'、符号常量 PI 等都是简单表达式。

表达式的值就是该表达式的运算结果。

所有表达式都有一个值及其类型，如算术表达式的值为整型或浮点型常量；关系或逻辑表达式的值为逻辑值“1”(表示“真”)或“0”(表示“假”)；赋值表达式的值是表达式最左侧变量的值；函数调用也是一种表达式，叫函数表达式，它的值是函数的返回值。

简单表达式的值是其本身的值或程序赋给它的当前值。

复杂表达式由多个简单表达式组成，各个简单表达式之间用运算符相连，因此复杂表达式的值是该表达式的运算结果。例如，$x+y-z$、$4+5/6-7$、$x=5$、$a\&\&b\|c$、$m=n=5+7$等都是复杂表达式，它们各自运算的结果就是各表达式的值。

有的读者可能会奇怪“$m=n=5+7$”怎么会是表达式呢？这里一定要把 C 语言中表达式的概念与数学中表达式的概念加以区分。

在 C 语言中，“$m=n=5+7$”是一个复杂表达式，该表达式的值就是它最终的运算结果，因此表达式“$m=n=5+7$”的值的求解过程如下：

(1) 对表达式“5+7”进行加法运算，计算结果就是该表达式的值，即表达式“5+7”的值为 12。

(2) 在表达式“$n=5+7$”中，将表达式“5+7”的值 12 赋给变量 n，且表达式“$n=5+7$”的值也是 12。

(3) 在表达式“$m=n=5+7$”中，将表达式“$n=5+7$”的值 12 赋给变量 m，且表达式“$m=n=5+7$”的值也是 12。

综上所述，复杂表达式的值的求解过程相对复杂，有时还需要考虑运算符的优先级和结合性问题，后面会结合具体内容详细讲解。

3.5.1　算术运算符和算术表达式

1. 算术运算符

本章重点要介绍的是算术运算符和赋值运算符。在 C 语言中，基本算术运算符有以

下 5 种。

(1) 加法运算符“+”：双目运算符，即应有两个数参与加法运算，如 a+b、4+8 等。

(2) 减法运算符“-”：双目运算符，但“-”也可作为负值运算符使用。此时，负值运算符“-”为单目运算，如$-x$、-5。

(3) 乘法运算符“*”：双目运算符。

(4) 除法运算符“/”：双目运算符。需要注意的是，当参与除法运算的两个数都是正整数时，其结果也是整数。例如，7/2 的结果不是 3.5 而是 3，小数部分 0.5 被舍去。若参与除法的两个数中有一个是负数，则不同的编译系统计算结果不同。大多数编译系统(如 VC++ 6.0、TC 等)遵循“向零凑整”的原则。例如，-7/2 的计算结果为-3，-8/3 的计算结果为-2，因为除法运算的运算结果应取整后向 0 靠拢。

(5) 取余运算符“%”：双目运算符，%两侧都应是整数，如 5%3 的运算结果为 2，-5%3 的运算结果为-2，6%1 的运算结果为 0。

对于+、-、*、/这 4 种基本算术运算，若两个参与运算的数中有一个是 float 型或 double 型，则计算结果为 double 型，因为对于不同数据类型混合运算而言，必须按照 C 语言的转换规则将参与运算的数转换成相同的类型，相关类型转换规则在 3.4 节中已详细讲解，这里不再赘述。

2. 算术表达式

算术表达式就是用算术运算符和括号将运算对象(操作数)连接起来的符合 C 语言语法规则的式子，其中运算对象包括常量、变量、函数等。

算术表达式的值就是该表达式最终的运算结果，是一个算术值。

例如，$a+b/c$、1+5、$x\%2$、$x*y-3.1+$'C'等都是算术表达式。

对于包含多种运算符的算术表达式的值的求解必须遵循一定的顺序进行，这种顺序称为运算符优先级。

C 语言对运算符的优先级有严格的规定，每种运算符都有一个优先级。在计算表达式时首先执行优先级高的运算符，然后执行优先级低的运算符。若表达式中出现了多个优先级相同的运算符，则按照运算符的结合方向进行运算。

下面以算术表达式为例说明复杂表达式的值的求解过程：

(1) 在求算术表达式的值时先按算术运算符的优先级别从高到低执行。例如，对算术表达式“$x-y*z$”而言，因为“*”和“/”的优先级别高于“+”和“-”，所以“$x-y*z$”相当于“$x-(y*z)$”。

(2) 若一个运算对象两侧的运算符的优先级别相同，则需按照 C 语言规定的“结合方向”进行运算。仍以基本算术运算为例，C 语言规定除负值运算符“-”之外，其他基本算术运算符的结合方向是“从左到右”，或者说“先左后右”。例如，在求解算术表达式“$x+y-z$”时，“+”和“-”的优先级相同，所以按照“从左到右”的结合方向先执行“$x+y$”的运算，再执行减 z 的运算。

(3) 如果一个运算符两侧的数据类型不同，则先自动进行类型转换，之后再对相同的数据类型进行运算。

(4) C 语言也有一些右结合性的运算符，如赋值运算符“=”、自增运算符“++”、自减运算符“--”、按位取反运算符“~”等，在后面会陆续讲到。

(5) 在 C 语言中，为了使运算顺序的表达更清楚，可以用圆括号来改变运算顺序。例如，在求解表达式“(1+5) * 3”时，需要先计算圆括号中的算术表达式“1+5”的值，在得到算术表达式的值 6 之后再进行乘法运算“6 * 3”。

(6) 在复杂表达式中圆括号可以嵌套，当出现圆括号嵌套的情况时从内向外进行运算。例如表达式“24/(1 * (3 * (2+2)))”，在运算时先算“2+2”，再算“3 * 4”，然后算“1 * 12”，最后算“24/12”，得到表达式的值为 2。对初学者而言，为了使表达式更清晰，可以适当使用圆括号。

3. 自增和自减运算符

自增运算符“++”和自减运算符“--”是 C 语言中使用非常多的算术运算符。

自增运算符“++”的作用是使操作数加 1；自减运算符“--”的作用是使操作数减 1。

自增和自减运算符可以放在操作数前面(前置)构成表达式，如++x、--y 等；也可放在操作数后面(后置)构成表达式，如 x++、y-- 等。

在不同的程序中，自增和自减运算符前置与后置所起的作用可能不同。

例如，假设变量 x 和 y 均为 int 型变量，x 的初值为 3。

从变量 x 的角度来看，++x 和 x++的作用是相同的，都相当于将变量 x 的值加 1，即对变量 x 来说，++x 和 x++都等价于“$x=x+1$”，自增运算符前置和后置的作用相同。

但在表达式 1“y=++x”和表达式 2“y=x++”中，对于变量 y 而言，++x 和 x++的作用是不同的。

在表达式 1 中，系统先将 x 的值加 1，x 的值由 3 变为 4，之后将 x 的值 4 作为“++x”的值赋给变量 y，最终 y 的值为 4、x 的值为 4。

在表达式 2 中，系统先将 x 的值 3 作为“x++”的值赋给变量 y，即 y 的值为 3，然后再将 x 的值加 1，最终 y 的值为 3、x 的值为 4。

综上所述，对变量 x 而言，++x 和 x++的作用相同，但对变量 y 而言，++x 和 x++的作用不同。

例 3.10　自增运算符前置与后置在表达式中的应用。

程序如下：

```
#include<stdio.h>
int main(void)
{
    int x=3,y=3,i,j;
    i=x++;                                          /*自增运算符后置*/
    j=++y;                                          /*自增运算符前置*/
    printf("x=%d,i=%d,y=%d,j=%d\n",x,i,y,j);        /*分别输出 x、i、y 和 j*/
    return 0;
}
```

程序运行结果如下：

```
x=4,i=3,y=4,j=4
```

在上例中求解程序第5行的表达式“$i=x++$”的值时，因为自增运算符“++”置于变量 x 之后，所以根据自增运算符后置的说明可知，程序先将变量 x 的值3作为“$x++$”的值赋给变量 i（即 $i=3$），之后再执行“$x=x+1$”的自增运算，使 x 的值变为4。

在求解程序第6行的表达式“$j=++y$”的值时，因为自增运算符“++”置于变量 y 之前，所以根据自增运算符前置的说明可知，程序先执行“$y=y+1$”的自增运算，使 y 的值变为4，并将其作为“$++y$”的值赋给变量 j，即 $j=4$。

通过上例可以看到，对于变量 x 和变量 y 而言，自增运算符前置与后置的作用一样，都是使变量的值加1，即 $x++$ 相当于 $x=x+1$、$++y$ 相当于 $y=y+1$。但是对于表达式“$i=x++$”和表达式“$j=++y$”而言，自增运算符前置与后置的效果是不一样的。

自减运算符的使用与自增运算符类似，也可分为前置和后置。

仍以整型变量 x 为例，假设 x 的初值为5。

对变量 x 而言，$--x$ 和 $x--$ 的作用相同，都相当于执行“$x=x-1$”的操作。

但在表达式1“$y=--x$”和表达式2“$y=x--$”中，对变量 y 而言，$--x$ 和 $x--$ 的作用不同。

在表达式1中，系统先将 x 的值减1（此时 x 的值为4），然后将其作为“$--x$”的值赋给变量 y，最终 y 的值为4、x 的值为4。

在表达式2中，系统先将 x 的值5作为“$x--$”的值赋给变量 y，即 y 的值为5，然后再将 x 的值减1，最终 y 的值为5、x 的值为4。

对于自增和自减运算符有以下几点需要注意：

（1）自增和自减运算符只能用于变量，不能用于常量或表达式。

例如：

表达式3++、--3、$(x+y)$++、--$(x+5)$都是不合法的。因为常量和表达式的值是不能改变的，或者说常量与表达式无法为自增或自减运算后的值提供存储空间。例如，对于表达式$(x+y)$++而言，假设 $x=1$、$y=3$，$(x+y)$的值等于4，那么$(x+y)$自增之后的值5放在哪里呢？无处可放。

（2）自增运算符“++”、自减运算符“--”的优先级与负号“-”相同，且运算符的结合方向都是自右至左。

例如：

假设整型变量 x 的值为3，则表达式“$-x++$”相当于“$-(x++)$”，先利用自增运算符后置得到“$x++$”的值为3，因此表达式“$-(x++)$”的值为-3，然后再将 x 的值加1，即 x 的值变为4。

而表达式“$-++x$”相当于“$-(++x)$”，先利用自增运算符前置，将变量 x 的值加1变为4，得到“$x++$”的值为4，然后再利用负号进行取反运算得到表达式“$-(++x)$”的值为-4。

（3）由于自增和自减运算符在表达式中前置或后置的作用不同，初学者在使用时容

易理解错误。

例如：

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

上述语句容易使初学者产生疑问，建议改成：

```
y=x++;
printf("%d,%d\n",x,y);
```

或直接改为：

```
printf("%d,%d\n",x,x+1);
```

3.5.2　赋值运算符和赋值表达式

在 C 语言中，赋值运算符包括简单赋值运算符和复合赋值运算符。

1. 赋值运算符

在 C 语言中，将赋值符号“=”称为简单赋值运算符（一般简称为赋值运算符），其作用是将一个值赋给一个变量。

例如：

```
int x;
x=5;
```

在上例中，在定义了整型变量 x 之后利用赋值运算符“=”将整数 5 赋给了变量 x。

赋值运算符也可将表达式的值赋给变量。

例如：

```
double x;
x=6.0/3.0;
```

在上例中，将表达式 6.0/3.0 的值 0.5 赋给了变量 x。

对于赋值运算符有以下几点需要注意：

（1）赋值运算符的结合方向为自右至左。

（2）C 语言中的赋值运算符“=”和数学中的等号“=”是不一样的。数学中的等号“=”是一种逻辑判断，常用于判断等号两边是否相等，类似于 C 语言后面要学的关系运算符中的“==”。C 语言中的赋值运算符“=”的作用是将一个值赋给一个变量，也就是说它是一种运算或者一种操作。

例如：

在数学中可以写“3+3=4+2”，表示等号两边相等，而 C 语言中赋值运算符的左边必须是一个变量，不允许出现常量。

（3）在数学中，“$x=y$”表示 x 和 y 相等。在 C 语言中，“$x=y$”则是将 y 的值赋给 x，即将 y 的值放入 x 的内存单元取代其原有的值。在 C 语言中，如果想表示变量 x 和变量 y 两者相等，应写为“$x==y$”，在后面介绍关系运算符时会详细讲解。

(4) 在使用赋值运算符时要根据变量定义时的数据类型明确其取值范围,若所赋的值超过变量的取值范围,编译器会报错。

(5) 如果赋值运算符两侧的类型不一致,但又都是数值型或字符型数据,那么赋值时要进行类型转换,具体规则详见 3.4.2 节,这里不再赘述。

2. 复合赋值运算符

为方便编程,除了赋值运算符"="之外,C 语言还提供了复合赋值运算符。

在赋值运算符的前面加上其他运算符就构成了复合赋值运算符。

C 语言规定凡是双目运算符(10 种)都可以和赋值运算符一起构成复合赋值运算符,包括+=、-=、*=、/=、%=、<<=、>>=、&=、^=和|=。

上述 10 种复合赋值运算符的结合方向都是自右至左,前 5 种用于算术运算,后 5 种用于位运算(详见第 9 章)。

例如:

$x+=5$ 等价于 $x=x+5$;

$x*=5$ 等价于 $x=x*5$;

$x\%=5$ 等价于 $x=x\%5$;

$x/=y+3$ 等价于 $x=x/(y+3)$。

上述语句都是对复合赋值运算符的运用。

下面以 $x+=5$ 为例说明复合赋值运算符的运算过程:

(1) 计算表达式"$x+5$"的值。

(2) 将表达式"$x+5$"的值重新赋给变量 x。

因此,表达式"$x+=5$"与表达式"$x=x+5$"是等价的。

表达式 $x*=5$、$x\%=5$ 和 $x/=5$ 的运算过程与"$x+=5$"类似,这里不再赘述。

需要注意的是,如果复合赋值运算符右侧的表达式中还有其他运算符,那就需要结合运算符的优先级和结合方向进行运算。

例如,对于表达式"$x/=y+3$"而言,因为加法运算符"+"的优先级高于复合赋值运算符"/=",所以要先计算表达式"$y+3$"的值,然后再进行"/="的复合赋值运算,即表达式"$x/=y+3$"等价于"$x=x/(y+3)$"。

在 C 语言中,使用复合赋值运算符的目的一是用于简化程序;二是用于提高程序的编译效率,使编译器生成的目标代码质量更高。

3. 赋值表达式

用赋值运算符把一个变量和一个表达式连起来构成的式子叫赋值表达式,其一般形式如下:

变量 赋值运算符 表达式

赋值表达式的值同样是赋值运算的结果,其值的求解过程如下:

(1) 计算赋值运算符右侧表达式的值。

(2) 将表达式的值赋给赋值运算符左侧的变量。

(3) 左侧变量的值即整个赋值表达式的值。

例如,在求解赋值表达式“$x=3$”的值时先计算赋值运算符“=”右边表达式的值,右边的表达式是一个常量,其值为 3; 之后将表达式的值 3 赋给变量 x; 最后,变量 x 的值就是表达式“$x=3$”的值。

简单来说,“赋值表达式的值”就是“赋值表达式最左侧的变量的值”。

赋值运算符“=”左侧的标识符也叫左值,只能是变量; 赋值运算符“=”右侧的表达式也叫右值,可以是简单表达式,也可以是复杂表达式。

请结合下例分析赋值表达式的值的求解过程。

例 3.11　分析赋值表达式的值。

程序如下:

```
#include<stdio.h>
int main(void)
{
    int x,y,z;
    x=y=z=5+7;                          /*利用赋值表达式给变量 x、y 和 z 赋值*/
    printf("%d,%d,%d\n",x,y,z);
    return 0;
}
```

程序运行结果如下:

```
12,12,12
```

对于上例,由前面的介绍可知,赋值运算符的结合方向是自右至左的,所以程序第 5 行的表达式相当于“$x=(y=(z=(5+7)))$”,其值的求解过程如下:

(1) 先求表达式“5+7”的值,其值为 12。

(2) 将表达式“5+7”的值 12 赋给变量 z,且变量 z 的值 12 就是赋值表达式“$z=(5+7)$”的值。

(3) 将赋值表达式“$z=(5+7)$”的值 12 赋给变量 y,且变量 y 的值 12 就是赋值表达式“$y=(z=(5+7))$”的值。

(4) 将赋值表达式“$y=(z=(5+7))$”的值 12 赋给变量 x,且变量 x 的值 12 就是赋值表达式“$x=(y=(z=(5+7)))$”的值。

综上所述,虽然变量 x、y、z 的值都是 12,但它们的值是由不同的赋值表达式的值赋予的。这一点与数学中等号的使用有很大的不同。

请读者结合以下表达式分析并练习赋值表达式的值的求解。

```
x=3+(y=2);                  /*y 的值为 2、x 的值为 5,整个表达式的值为 5*/
x=(y=6)/(z=3);              /*z 的值为 3、y 的值为 6、x 的值为 2,整个表达式的值为 2*/
```

同样,在赋值表达式中也可以使用复合赋值运算符,假设变量 y 的初值为 20。

```
x=2+(y-=3*5);               /*y 的值为 5、x 的值为 7,整个表达式的值为 7*/
```

3.5.3 逗号运算符和逗号表达式

为方便程序设计，C 语言还提供了一种特殊的运算符——逗号运算符“,”，其作用是将两个表达式连接起来构成一个表达式。

例如：

```
x=3,3*2                    /*逗号运算符“,”将表达式“x=3”和表达式“3*2”相连*/
```

利用逗号运算符可以将多个表达式连接在一起构成逗号表达式，其一般形式如下：

```
表达式 1,表达式 2,表达式 3,…,表达式 n
```

逗号表达式的结合方向是自左至右的，所以逗号表达式的求解是先求解表达式 1，再求解表达式 2，再求解表达式 3，……，最后求解表达式 n，且整个逗号表达式的值就是最右侧的表达式 n 的值。

例如：

```
y=(x=3*4,6/3);
```

在上述语句中“$x=3*4,6/3$”是一个逗号表达式，且整个逗号表达式的值为表达式“6/3”的值 2。因此，该语句的作用是先将 12 赋给变量 x，然后将 2 赋给变量 y。

又例如：

```
x=(y++,z++);
```

上述语句的运算过程如下：

首先使变量 y 自增 1；然后将变量 z 的值作为“$z++$”的值赋给逗号表达式“$y++,z++$”；之后将逗号表达式的值赋给变量 x；最后使变量 z 自增 1。

若变量 y 和 z 的初值均为 1，则上述语句运行后 x 值为 1、y 值为 2、z 值为 2。

请注意，上面两个例子中都使用了圆括号“()”，这是因为逗号运算符的优先级在所有运算符中是最低的。如果想实现上面所说的编程目的，必须用圆括号指定运算顺序。

在 C 语言中，逗号表达式其实就是用逗号运算符把若干个表达式连起来，其目的一般是希望分别得到各个表达式的值，整个逗号表达式的值有时反而不会用到。逗号表达式在 C 语言中应用最多的情况是在“for 循环语句”中(详见第 4 章)。

最后需要强调的是，在 C 语言程序中并非所有使用的逗号都是逗号运算符。在后面介绍函数调用时(详见第 6 章)多个函数参数之间要用逗号隔开，例如“sum(3,5)”；格式输入函数 scanf 和格式输出函数 printf 在使用时也会用到逗号，例如“printf("%d,%d,%d\n",x,y,z);”和“scanf("%d,%d",&x,&y);”。

例 3.12 逗号表达式的应用。

程序如下：

```
int main(void)
{
    int a=2,b=4,c=6,x,y;
    y=(x=a+b),(b+c);        /*利用逗号表达式给变量 x 和 y 赋值*/
```

```
    printf("y=%d,x=%d",y,x);
    return 0;
}
```

程序运行结果如下：

```
m=10,n=6
```