第2章 数据类型与表达式

2.1 C++的数据类型
2.2 常量
2.3 变量
2.4 C++的运算符
2.5 算术运算符与算术表达式
2.6 赋值运算符与赋值表达式
2.7 逗号运算符与逗号表达式

2.1 C++的数据类型

计算机处理的对象是数据，而数据是以某种特定的形式存在的（例如整数、浮点数、字符等形式）。不同的数据之间往往还存在某些联系（例如由若干个整数组成一个整数数组）。数据结构指的是数据的组织形式。例如，数组就是一种数据结构。不同的计算机语言所允许使用的数据结构是不同的。处理同一类问题，如果数据结构不同，算法也会不同。例如，对10个整数排序和对包含10个元素的整型数组排序的算法是不同的。

C++可以使用的数据类型如下：

- 数据类型
  - 基本类型
    - 整型
      - 短整型(**short int**)
      - 整型(**int**)
      - 长整型(**long int**)
    - 字符型(**char**)
    - 浮点型
      - 单精度型(**float**)
      - 双精度型(**double**)
      - 长双精度型(**long double**)
  - 构造类型
    - 布尔型(**bool**)
    - 枚举类型(**enum**)
    - 数组类型
    - 结构体类型(**struct**)
    - 共用体类型(**union**)
    - 类类型(**class**)
  - 指针类型
  - 引用类型
  - 空类型(**void**)

布尔型就是逻辑型，空类型就是无值型。

C++的数据包括常量与变量，常量与变量都具有类型。由以上这些数据类型还可以构成更复杂的数据结构。例如利用指针和结构体类型可以构成表、树、栈等复杂的数据结构。

C++并没有统一规定各类数据的精度、数值范围和在内存中所占的字节数，各C++编译系统根据自己的情况作出安排。书中表2.1列出了Visual C++数值型和字符型数据的情况。

说明：

(1) 整型数据分为长整型**(long int)**、一般整型**(int)**和短整型**(short int)**。在**int**前面加**long**和**short**分别表示长整型和短整型。

(2) 整型数据的存储方式为按二进制数形式存储，例如十进制整数**85**的二进制形式为**1010101**，则在内存中的存储形式如图**2.1**所示。

0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1

图**2.1**

(3) 在整型符号**int**和字符型符号**char**的前面,可以加修饰符**signed**(表示“有符号”)或**unsigned**(表示“无符号”)。如果指定为**signed**，则数值以补码形式存放，存储单元中的最高位**(bit)**用来表示数值的符号。如果指定为**unsigned**，则数值没有符号，全部二进制位都用来表示数值本身。例如短整型数据占两个字节，见图**2.2**。

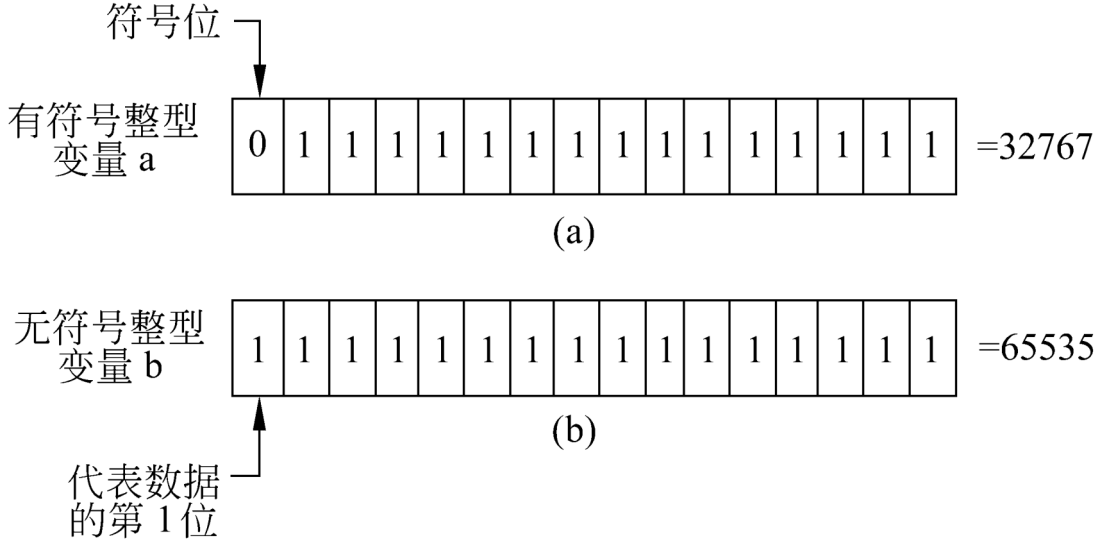


图**2.2**

有符号时，能存储的最大值为2^{15}-1，即32767，最小值为-32768。无符号时，能存储的最大值为2^{16}-1，即65535，最小值为0。有些数据是没有负值的，可以使用unsigned，它存储正数的范围比用signed时要大一倍。

(4) 浮点型(又称实型)数据分为单精度(float)、双精度(double)和长双精度(long double)3种，在Visual C++ 6.0中，对float提供6位有效数字，对double提供15位有效数字，并且float和double的数值范围不同。对float分配4个字节，对double和long double分配8个字节。

(5) 表中类型标识符一栏中，方括号［ ］包含的部分可以省写，如short和short int等效，unsigned int和unsigned等效。

2.2 常量

2.2.1 什么是常量

常量的值是不能改变的，一般从其字面形式即可判别是否为常量。常量包括两大类，即数值型常量(即常数)和字符型常量。如**12**，**0**，**-3**为整型常量，**4.6**，**-1.23**为实型常量，包含在两个单撇号之间的字符为字符常量，如**'a','x'**。这种从字面形式即可识别的常量称为“字面常量”或“直接常量”。

2.2.2 数值常量

数值常量就是通常所说的常数。在C++中，数值常量是区分类型的，从字面形式即可识别其类型。

1. 整型常量(整数）的类型

在上一节中已知道： 整型数据可分为**int,short int,long int**以及**unsigned int,unsigned short,unsigned long**等类别。整型常量也分为以上类别。为什么将数值常量区分为不同的类别呢？因为在进行赋值或函数的参数虚实结合时要求数据类型匹配。

那么，一个整型常量怎样从字面上区分为以上的类别呢？

(1) 一个整数，如果其值在**-32768~+32767**范围内，认为它是**short int**型，它可以赋值给**short int**型、**int**型和**long int**型变量。

(2) 一个整数，如果其值超过了上述范围，而在**-2147483648~+2147483647**范围内，则认为它是**long int**型，可以将它赋值给一个**int**或**long int**型变量。

(3) 如果某一计算机系统的**C++**版本（例如**Visual C++**）确定**int**与**long int**型数据在内存中占据的长度相同，则它们能够表示的数值的范围相同。因此，一个**int**型的常量也同时是一个**long int**型常量，可以赋给ｉｎｔ型或**long int**型变量。

(4) 常量无**unsigned**型。但一个非负值的整数可以赋值给**unsigned**整型变量，只要它的范围不超过变量的取值范围即可。

一个整型常量可以用3种不同的方式表示：

(1) 十进制整数。如1357,-432，0等。在一个整型常量后面加一个字母l或L，则认为是long int型常量。例如123L，421L,0L等，这往往用于函数调用中。如果函数的形参为long int，则要求实参也为long int型，此时用123作实参不行，而要用123L作实参。

(2) 八进制整数。在常数的开头加一个数字0，就表示这是以八进制数形式表示的常数。如020表示这是八进制数20，即$(20)_8$，它相当于十进制数16。

(3) 十六进制整数。在常数的开头加一个数字0和一个英文字母X(或x)，就表示这是以十六进制数形式表示的常数。如0X20表示这是十六进制数20，即$(20)_{16}$，它相当于十进制数32。

2. 浮点数的表示方法

一个浮点数可以用两种不同的方式表示：

(1) 十进制小数形式。如**21.456,-7.98**等。它一般由整数部分和小数部分组成，可以省略其中之一(如**78.**或**.06**，**.0)**，但不能二者皆省略。**C++**编译系统把用这种形式表示的浮点数一律按双精度常量处理，在内存中占**8**个字节。如果在实数的数字之后加字母**F**或**f**，表示此数为单精度浮点数，如**1234F,-43f**，占**4**个字节。如果加字母**L**或**l**，表示此数为长双精度数**(long double)**，在**GCC**中占**12**个字节，在**Visual C++ 6.0**中占**8**个字节。

(2) 指数形式(即浮点形式)

一个浮点数可以写成指数形式，如**3.14159**可以表示为**0.314159**×10^{1}，**3.14159**×10^{0},**31.4159**×10^{-1},**314.159**×10^{-2}等形式。在程序中应表示为：**0.314159e1**，**3.14159e0**,**31.4159e-1** ,**314.159e-2**,用字母**e**表示其后的数是以**10**为底的幂，如**e12**表示10^{12}。

其一般形式为

数符 数字部分 指数部分

上面各数据中的**0.314159**，**3.14159**, **31.4159**, **314.159** 等就是其中的数字部分。可以看到： 由于指数部分的存在，使得同一个浮点数可以用不同的指数形式来表示，数字部分中小数点的位置是浮动的。例如：

a=0.314159e1;

a=3.14159e0;

a=31.4159e-1;

a=314.159e-2;

以上**4**个赋值语句中，用了不同形式的浮点数，但其作用是相同的。

在程序中不论把浮点数写成小数形式还是指数形式，在内存中都是以指数形式(即浮点形式)存储的。例如不论在程序中写成**314.159**或**314.159e0**，**31.4159e1**，**3.14159e2, 0.314159e3**等形式，在内存中都是以规范化的指数形式存放，如图**2.3**所示。

+	.314159	3
数符	数字部分	指数部分

图**2.3**

数字部分必须小于**1**，同时，小数点后面第一个数字必须是一个非**0**数字，例如不能是**0.0314159**。因此**314.159**和**314.159e0**，**31.4159e1**，**3.14159e2**，**0.314159e3**在内存中表示成**0.314159**×10^3。存储单元分为两部分，一部分用来存放数字部分，一部分用来存放指数部分。为便于理解，在图**2.3**中是用十进制表示的，实际上在存储单元中是用二进制数来表示小数部分，用**2**的幂次来表示指数部分的。

对于以指数形式表示的数值常量，也都作为双精度常量处理。

2.2.3 字符常量

1. 普通的字符常量

用单撇号括起来的一个字符就是字符型常量。如**'a','#','%','D'**都是合法的字符常量，在内存中占一个字节。注意：①字符常量只能包括一个字符，如**'AB'** 是不合法的。②字符常量区分大小写字母，如**'A'**和**'a'**是两个不同的字符常量。③撇号**(')**是定界符，而不属于字符常量的一部分。如**cout<<'a';** 输出的是一个字母**“a”**，而不是**3**个字符**“'a' ”**。

2. 转义字符常量

除了以上形式的字符常量外，C++还允许用一种特殊形式的字符常量，就是以“\”开头的字符序列。例如，'\n'代表一个“换行”符。“cout<<'\n';”将输出一个换行，其作用与“cout<<endl;”相同。这种“控制字符”，在屏幕上是不能显示的。在程序中也无法用一个一般形式的字符表示，只能采用特殊形式来表示。

常用的以“\”开头的特殊字符见书中表2.2。

3. 字符数据在内存中的存储形式及其使用方法

将一个字符常量存放到内存单元时，实际上并不是把该字符本身放到内存单元中去，而是将该字符相应的**ASCII**代码放到存储单元中。如果字符变量**c1**的值为**'a',c2**的值为**'b'**，则在变量中存放的是**'a'**的**ASCII**码**97**，**'b'** 的**ASCII**码**98**，如图**2.4(a)**所示，实际上在内存中是以二进制形式存放的，如图**2.4(b)**所示。

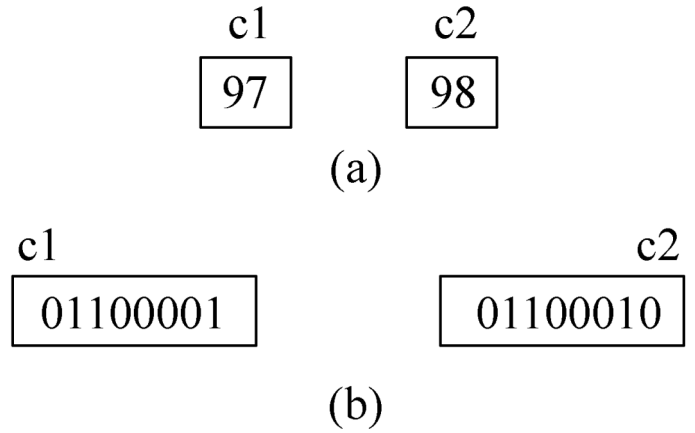


图**2.4**

既然字符数据是以**ASCII**码存储的，它的存储形式就与整数的存储形式类似。这样，在**C++**中字符型数据和整型数据之间就可以通用。一个字符数据可以赋给一个整型变量，反之，一个整型数据也可以赋给一个字符变量。也可以对字符数据进行算术运算，此时相当于对它们的**ASCII**码进行算术运算。

例**2.1** 将字符赋给整型变量。

```
#include <iostream>
using namespace std;
int main( )
   {int  i,j;                  //i和j是整型变量
i='A';                  //将一个字符常量赋给整型变量i
j='B';                  //将一个字符常量赋给整型变量j
cout<<i<<' '<<j<<'\n';    //输出整型变量i和j的值，'\n'是换行符
return 0;
}
```

执行时输出

65 66

i和**j**被指定为整型变量。但在第**5**和第**6**行中，将字符**'A'**和**'B'**分别赋给**i**和**j**，它的作用相当于以下两个赋值语句：

i＝65；j＝66;

因为**'A'**和**'B'**的**ASCII**码为**65**和**66**。在程序的第**5**和第**6**行是把**65**和**66**直接存放到**i**和**j**的内存单元中。因此输出**65**和**66**。

可以看到：在一定条件下，字符型数据和整型数据是可以通用的。但是应注意字符数据只占一个字节，它只能存放**0~255**范围内的整数。

例**2.2** 字符数据与整数进行算术运算。下面程序的作用是将小写字母转换为大写字母。

```
#include <iostream>
using namespace std;
int main( )
{char c1,c2;
 c1='a';
 c2='b';
 c1=c1-32;
 c2=c2-32;
 cout<<c1<<' '<<c2<<endl;
 return 0;
}
```

运行结果为

A B

'a'的ASCII码为97，而'A'的ASCII码为65，'b'为98，'B'为66。从ASCII代码表中可以看到每一个小写字母比它相应的大写字母的ASCII代码大32。C++符数据与数值直接进行算术运算，'a'-32得到整数65，'b'-32得到整数66。将65和66存放在c1,c2中，由于c1,c2是字符变量，因此用cout输出c1,c2时，得到字符A和B(A的ASCII码为65,B的ASCII码为66)。

4. 字符串常量

用双撇号括起来的部分就是字符串常量，如"abc"，"Hello!"，"a+b"，"Li　ping"都是字符串常量。字符串常量"abc"在内存中占4个字节(而不是3个字节)，见图2.5。

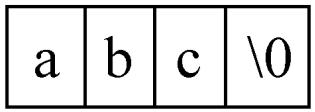


图2.5

编译系统会在字符串最后自动加一个'\0'作为字符串结束标志。但'\0'并不是字符串的一部分，它只作为字符串的结束标志。如

```
cout<<"abc"<<endl;
```

输出3个字符abc，而不包括'\0'。

注意："a"和'a'代表不同的含义，"a"是字符串常量，'a' 是字符常量。前者占两个字节，后者占1个字节。请分析下面的程序片段：

```
char c;              //定义一个字符变量
c='a';               //正确
c="a";               //错误，c只能容纳一个字符
```

字符串常量要用字符数组来存放，见第5章。

请思考： 字符串常量"abc＼n"包含几个字符?不是5个而是4个字符，其中“＼n”是一个转义字符。但它在内存中占5个字节(包括一个“＼0”字符) 。编译系统遇到“＼”时就会把它认作转义字符的标志，把它和其后的字符一起作为一个转义字符。

如果“\”后面的字符不能与“\”组成一个合法的转义字符(如"\c")，则在编译时显示出错信息。如果希望将“\”字符也作为字符串中的一个字符，则应写为"abc\\n"，此时字符包括5个字符，即a,b,c,\,n。如果有以下输出语句：

```
cout<<"abc\\\n"<<endl;
```

则会输出： abc\，然后换行。同理执行

```
cout<<"I say \"Thank you！\"\n";
```

的输出是： I say "Thank you！"

如果在一个字符串中最后一个字符为“\”，则表示它是续行符，下一行的字符是该字符串的一部分，且在两行字符串间无空格。如

cout<<"We must study C＼ //本行最后的“＼”后面的空格和换行均不起作用

++ hard!"; //本行的字符紧连在上一行最后的“＼”前面字符之后

则输出：

We must study C++ hard!

2.2.4 符号常量

为了编程和阅读的方便，在C++程序设计中，常用一个符号名代表一个常量，称为符号常量，即以标识符形式出现的常量。

例2.3 符号常量的使用。

```
#define PRICE 30          //注意这不是语句，末尾不要加分号
int main ( )
{ int num，total;
num=10;
total=num * PRICE;
cout<<"total="<<total<<endl;
return 0;
}
```

程序中用预处理命令**#define**指定**PRICE**在本程序单位中代表常量**30**，此后凡在本程序单位中出现的**PRICE**都代表**30**，可以和常量一样进行运算，程序运行结果为

total=300

请注意符号常量虽然有名字，但它不是变量。它的值在其作用域(在本例中为主函数)内是不能改变的，也不能被赋值。如用赋值语句**“PRICE=40;”**给**PRICE**赋值是错误的。 使用符号常量的好处是：

（1） 含义清楚。

（2） 在需要改变一个常量时能做到“一改全改”。如

#define PRICE 35

2.3 变量

2.3.1 什么是变量

其实在前面的例子中已经多次用到了变量。在程序运行期间其值可以改变的量称为变量。一个变量应该有一个名字，并在内存中占据一定的存储单元，在该存储单元中存放变量的值。请注意区分变量名和变量值这两个不同的概念，见图**2.6**。

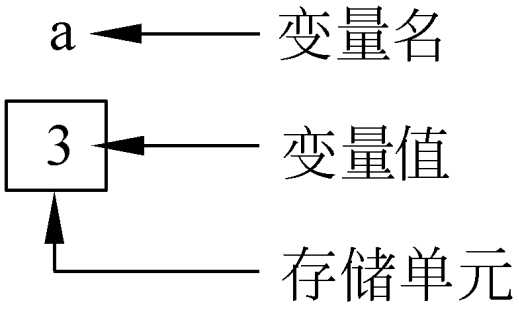


图**2.6**

2.3.2 变量名规则

先介绍标识符的概念。和其他高级语言一样，用来标识变量、符号常量、函数、数组、类型等实体名字的有效字符序列称为标识符（**identifier**）。简单地说，标识符就是一个名字。变量名是标识符的一种，变量的名字必须遵循标识符的命名规则。

C++规定标识符只能由字母、数字和下划线**3**种字符组成，且第一个字符必须为字母或下划线。下面列出的是合法的标识符，也是合法的变量名：

sum, average, total, day, month, Student_name, tan, BASIC, li_ling

下面是不合法的标识符和变量名：

M.D.John, $123, #33, 3G64, Ling li， C++, Zhang-ling, U.S.A.

注意，在**C++**中，大写字母和小写字母被认为是两个不同的字符。因此，**sum**和**SUM**是两个不同的变量名。一般地，变量名用小写字母表示，与人们日常习惯一致，以增加可读性。应注意变量名不能与**C++**的关键字、系统函数名和类名相同。在国外软件开发工作中，常习惯在变量前面加一个字母以表示该变量的类型，如**iCount**表示这是一个整型变量，**cSex**表示这是一个字符型变量。

C++没有规定标识符的长度（字符个数），但各个具体的**C**编译系统都有自己的规定。有的系统取**32**个字符，超过的字符不被识别。

2.3.3 定义变量

在C++语言中，要求对所有用到的变量作强制定义，也就是必须“先定义，后使用”，如例2.2和例2.3那样。定义变量的一般形式是

变量类型 变量名表列;

变量名表列指的是一个或多个变量名的序列。如

```
float a,b,c,d,e;
```

定义a,b,c,d,e为单精度型变量，注意各变量间以逗号分隔，最后是分号。

可以在定义变量时指定它的初值。如

```
float a=83.5,b,c=64.5,d=81.2,e;   //对变量a,c,d指定了初值，b和d未指定初值
```

C语言要求变量的定义应该放在所有的执行语句之前，而C++则放松了限制，只要求在第一次使用该变量之前进行定义即可。也就是说，它可以出现在语句的中间，如

```
int a;          //定义变量a(在使用a之前定义)
a=3;            //执行语句，对a赋值
float b;        //定义变量b(在使用b之前定义)
b=4.67;         //执行语句，对b赋值
char c;         //定义变量c(在使用c之前定义)
c='A';          //执行语句，对c赋值
```

C++要求对变量作强制定义的目的是：

(1) 凡未被事先定义的，不作为变量名，这就能保证程序中变量名使用得正确。例如，如果在声明部分写了

int student;

而在执行语句中错写成**statent**。如

statent=30;

在编译时检查出**statent**未经定义，作为错误处理。输出“变量**statent**未经声明”的信息，便于用户发现错误，避免变量名使用时出错。

(2) 每一个变量被指定为一确定类型，在编译时就能为其分配相应的存储单元。如指定 a 和 b 为**int**型，一般的编译系统对其各分配**4**个字节，并按整数方式存储数据。

(3) 指定每一变量属于一个特定的类型，这就便于在编译时，据此检查该变量所进行的运算是否合法。例如，整型变量**a**和**b**，可以进行求余运算：

a%b

％是“求余”（见**2．4** 节），得到**a/b**的余数。如果将**a**和**b**指定为实型变量，则不允许进行“求余”运算，在编译时会给出有关的出错信息。

2.3.4 为变量赋初值

允许在定义变量时对它赋予一个初值，这称为变量初始化。初值可以是常量，也可以是一个有确定值的表达式。如

float a,b=5.78*3.5,c=2*sin(2.0);

表示定义了**a,b,c**为单精度浮点型变量，对**b**初始化为**5.78*3,** 对**c**初始化为**2*sin(2.0)**，在编译连接后，从标准函数库得到正弦函数**sin(2.0)**的值，因此变量**c**有确定的初值。变量**a**未初始化。

如果对变量未赋初值，则该变量的初值是一个不可预测的值，即该存储单元中当时的内容是不知道的。例如，若未对**a**和**b**赋值，执行输出语句

cout<<a<<" "<<b<<" "<<c<<endl;

输出结果可能为

1.48544e-38 15 1.81858　　(各次运行情况可能不同)

初始化不是在编译阶段完成的（只有在第**4**章中介绍的静态存储变量和外部变量的初始化是在编译阶段完成的），而是在程序运行时执行本函数时赋予初值的，相当于执行一个赋值语句。例如，

int a=3;

相当于以下两个语句：

int a;　　// 指定 a 为整型变量

a=3;　　// 赋值语句，将 3 赋给 a

对多个变量赋予同一初值，必须分别指定，不能写成

float a=b=c=4.5;

而应写成

float a=4.5,b=4.5,c=4.5;

或

float a,b,c=4.5;

a=b=c;

2.3.5 常变量

在定义变量时，如果加上关键字**const**，则变量的值在程序运行期间不能改变，这种变量称为常变量**(constant variable)**。例如，

```
const int a=3;          //用const来声明这种变量的值不能改变，指定其值始终为3
```

在定义常变量时必须同时对它初始化(即指定其值)，此后它的值不能再改变。常变量不能出现在赋值号的左边。例如上面一行不能写成

```
const int a;
a=3;                    //常变量不能被赋值
```

可以用表达式对常变量初始化，如

const int b=3+6,c=3*cos(1.5);　//b的值被指定为9，c的值被指定为3*cos(1.5)

但应注意，由于使用了系统标准数学函数**cos**，必须将包含该函数有关的信息的头文件**“cmath”(**或**math.h)**包含到本程序单位中来，可以在本程序单位的开头加上以下**#include**命令：

#include <cmath> 或 **#include <math.h>**

变量的值应该是可以变化的，怎么值是固定的量也称变量呢？其实，从计算机实现的角度看，变量的特征是存在一个以变量名命名的存储单元，在一般情况下，存储单元中的内容是可以变化的。对常变量来说，无非在此变量的基础上加上一个限定：存储单元中的值不允许变化。因此常变量又称为只读变量**(read-only-variable)**。

请区别用**#define**命令定义的符号常量和用**const**定义的常变量。符号常量只是用一个符号代替一个字符串，在预编译时把所有符号常量替换为所指定的字符串，它没有类型，在内存中并不存在以符号常量命名的存储单元。而常变量具有变量的特征，它具有类型，在内存中存在着以它命名的存储单元，可以用**sizeof**运算符测出其长度。与一般变量惟一的不同是指定变量的值不能改变。用**#define**命令定义符号常量是**C**语言所采用的方法，**C++**把它保留下来是为了和**C**兼容。**C++**的程序员一般喜欢用**const**定义常变量。虽然二者实现的方法不同，但从使用的角度看，都可以认为用了一个标识符代表了一个常量。有些书上把用**const**定义的常变量也称为定义常量，但读者应该了解它和符号常量的区别。

2.4 C++的运算符

C++的运算符十分丰富，使得C++的运算十分灵活方便。例如把赋值号(=)也作为运算符处理，这样，a=b=c=4就是合法的表达式，这是与其他语言不同的。C++提供了以下运算符：

(1) 算术运算符

+(加)-(减) *(乘) /(除) %(整除求余)++(自加) --(自减)

(2) 关系运算符

＞(大于)＜(小于) ＝＝(等于)＞＝(大于或等于)＜＝(小于或等于)！＝(不等于)

(3) 逻辑运算符

&&(逻辑与) ||(逻辑或) ！(逻辑非)

(4) 位运算符

<<(按位左移) >>(按位右移) &(按位与) |(按位或) ∧(按位异或) ~(按位取反)

(5) 赋值运算符 (＝及其扩展赋值运算符)

(6) 条件运算符 (?:)

(7) 逗号运算符 (，)

(8) 指针运算符 (*)

(9) 引用运算符和地址运算符 (&)

(10) 求字节数运算符（ s i z e o f ）

(11) 强制类型转换运算符（ (类型) 或类型( )）

(12) 成员运算符 （**.**）

(13) 指向成员的运算符 （**->**）

(14) 下标运算符 （［ ］）

(15) 其他 （如函数调用运算符（））

在本章中主要介绍算术运算符与算术表达式，赋值运算符与赋值表达式，逗号运算符与逗号表达式，其他运算符将在以后各章中陆续介绍。

2.5 算术运算符与算术表达式

2.5.1 基本的算术运算符

＋（加法运算符，或正值运算符。如**3+5**，**+3**）

－（减法运算符，或负值运算符。如**5-2**，**-3**）

*** (**乘法运算符。如**3*5)**

／（除法运算符。如**5/3**）

％（模运算符，或称求余运算符，％两侧均应为整型数据，如７％４的值为３）。

需要说明，两个整数相除的结果为整数，如**5/3**的结果值为**1**，舍去小数部分。但是，如果除数或被除数中有一个为负值，则舍入的方向是不固定的。例如，**-5/3**在有的**C++**系统上得到结果**-1**，有的**C++**系统则给出结果**-2**。多数编译系统采取“向零取整”的方法，即**5/3**的值等于**1**，**-5/3**的值等于**-1**，取整后向零靠拢。

如果参加**+, -, *, /** 运算的两个数中有一个数为**float**型数据，则运算的结果是**double**型，因为**C++**在运算时对所有**float**型数据都按**double**型数据处理。

2.5.2 算术表达式和运算符的优先级与结合性

用算术运算符和括号将运算对象（也称操作数）连接起来的、符合C++语法规则的式子，称C++算术表达式。运算对象包括常量、变量、函数等。例如，下面是一个合法的C++算术表达式：

a*b/c-1.5+'a'

C++语言规定了运算符的优先级和结合性。在求解表达式时，先按运算符的优先级别高低次序执行，例如先乘除后加减。如有表达式**a-b*c**，**b**的左侧为减号，右侧为乘号，而乘号优先于减号，因此，相当于**a-(b*c)**。如果在一个运算对象两侧的运算符的优先级别相同，如**a-b+c**，则按规定的“结合方向”处理。

C++规定了各种运算符的结合方向（结合性），算术运算符的结合方向为“自左至右”，即先左后右，因此b先与减号结合，执行a-b的运算，再执行加c的运算。“自左至右的结合方向”又称“左结合性”，即运算对象先与左面的运算符结合。以后可以看到有些运算符的结合方向为“自右至左”，即右结合性（例如赋值运算符）。关于“结合性”的概念在其他一些高级语言中是没有的，是C和C++的特点之一，希望能弄清楚。附录B列出了所有运算符以及它们的优先级别和结合性。

2.5.3 表达式中各类数值型数据间的混合运算

在表达式中常遇到不同类型数据之间进行运算，如

10+'a'+1.5-8765.1234*'b'

在进行运算时，不同类型的数据要先转换成同一类型，然后进行运算。转换的规则按图**2.7**所示。

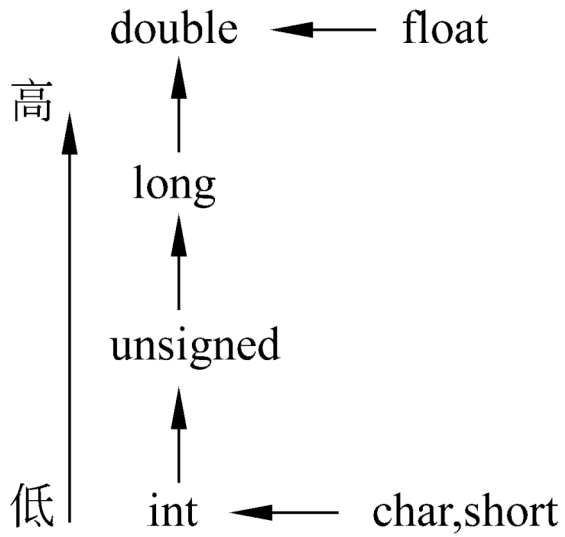


图**2.7**

假设已指定 i 为整型变量， f 为**float**变量， d 为**double**型变量， e 为**long**型，有下面表达式：

10+'a'+i*f-d/e

运算次序为： ①进行**10+'a'**的运算，先将' a '转换成整数**97**，运算结果为**107**。②进行**i*f**的运算。先将**i**与**f**都转换成**double**型，运算结果为**double**型。③整数**107**与**i*f**的积相加。先将整数**107**转换成双精度数（小数点后加若干个**0**，即**107.000…00**），结果为**double**型。④将变量**e**转换成**double**型，**d/e**结果为**double**型。⑤将**10+'a'+i*f**的结果与**d/e**的商相减，结果为**double**型。

上述的类型转换是由系统自动进行的。

2.5.4 自增和自减运算符

在C和C++中，常在表达式中使用自增(++)和自减(--)运算符，他们的作用是使变量的值增1或减1，如

++i（在使用i之前，先使i的值加1，如果i的原值为3，则执行j=++i后，j的值为4）

--i (在使用i之前，先使i的值减1，如果i的原值为3，则执行j=--i后，j的值为2)

i++ (在使用i之后，使i的值加1，如果i的原值为3，则执行j=i++后，j的值为3，然后i变为4)

i--(在使用i之后，使i的值减1，如果i的原值为3，则执行j=i--后，j的值为3，然后i变为2)

++i是先执行**i＝i+1**后，再使用**i**的值；而**i++**是先使用**i**的值后，再执行**i＝i+1**。

正确地使用**++**和**--**，可以使程序简洁、清晰、高效。

请注意：

(1) 自增运算符**(++)**和自减运算符**(--)**只能用于变量，而不能用于常量或表达式。

(2) ++和**--**的结合方向是“自右至左”，见附录**B**。

(3) 自增运算符（**++**）和自减运算符（**--**）使用十分灵活，但在很多情况下可能出现歧义性，产生“意想不到”的副作用。

(4) 自增（减）运算符在**C++**程序中是经常见到的，常用于循环语句中，使循环变量自动加**1**。也用于指针变量，使指针指向下一个地址。

2.5.5 强制类型转换运算符

在表达式中不同类型的数据会自动地转换类型，以进行运算。有时程序编制者还可以利用强制类型转换运算符将一个表达式转换成所需类型。例如：

(double) a　　（将**a**转换成**double**类型）

（**int**）（**x+y**）　　（将**x+y**的值转换成整型）

（**float**）**(5%3)**　　（将**5%3**的值转换成**float**型）

强制类型转换的一般形式为

（类型名）（表达式）

注意： 如果要进行强制类型转换的对象是一个变量，该变量可以不用括号括起来。如果要进行强制类型转换的对象是一个包含多项的表达式，则表达式应该用括号括起来。如果写成

（ｉｎｔ）ｘ＋ｙ

则只将ｘ转换成整型，然后与ｙ相加。

以上强制类型转换的形式是原来**C**语言使用的形式，**C++**把它保留了下来，以利于兼容。**C++**还增加了以下形式：

类型名（表达式）

如**int(x)** 或 **int(x+y)**

类型名不加括号，而变量或表达式用括号括起来。这种形式类似于函数调用。但许多人仍习惯于用第一种形式，把类型名包在括号内，这样比较清楚。

需要说明的是在强制类型转换时，得到一个所需类型的中间变量，但原来变量的类型未发生变化。例如：

（int）x

如果 x 原指定为**float**型，值为**3.6**，进行强制类型运算后得到一个**int**型的中间变量，它的值等于**3**，而 x 原来的类型和值都不变。

例**2.4** 强制类型转换。

```
#include <iostream>
using namespace std;
int main( )
{ float x;
  int i;
  x=3.6;
  i=(int)x;
  cout<<"x="<<x<<",i="<< i<<endl;
  return 0;
}
```

运行结果如下：

x ＝**3.6**，**i=3**

x 的型仍为**float**型，值仍等于**3.6**。

由上可知，有两种类型转换，一种是在运算时不必用户指定，系统自动进行的类型转换，如**3+6.5**。第二种是强制类型转换。当自动类型转换不能实现目的时，可以用强制类型转换。此外，在函数调用时，有时为了使实参与形参类型一致，可以用强制类型转换运算符得到一个所需类型的参数。

2.6 赋值运算符与赋值表达式

2.6.1 赋值运算符

赋值符号“＝”就是赋值运算符，它的作用是将一个数据赋给一个变量。如**“a=3”**的作用是执行一次赋值操作（或称赋值运算）。把常量**3**赋给变量**a**。也可以将一个表达式的值赋给一个变量。

2.6.2 赋值过程中的类型转换

如果赋值运算符两侧的类型不一致，但都是数值型或字符型时，在赋值时会自动进行类型转换。

(1) 将浮点型数据（包括单、双精度）赋给整型变量时，舍弃其小数部分。

(2) 将整型数据赋给浮点型变量时，数值不变，但以指数形式存储到变量中。

(3) 将一个**double**型数据赋给**float**变量时，要注意数值范围不能溢出。

(4) 字符型数据赋给整型变量，将字符的**ASCII**码赋给整型变量。

(5) 将一个**int**、**short**或**long**型数据赋给一个**char**型变量，只将其低**8**位原封不动地送到**char**型变量（发生截断）。例如

```
short int i=289;
char c;
c=i;                    //将一个int型数据赋给一个char型变量
```

赋值情况见图**2.8**。为方便起见，以一个**int**型数据占两个字节**(16**位**)**的情况来说明。

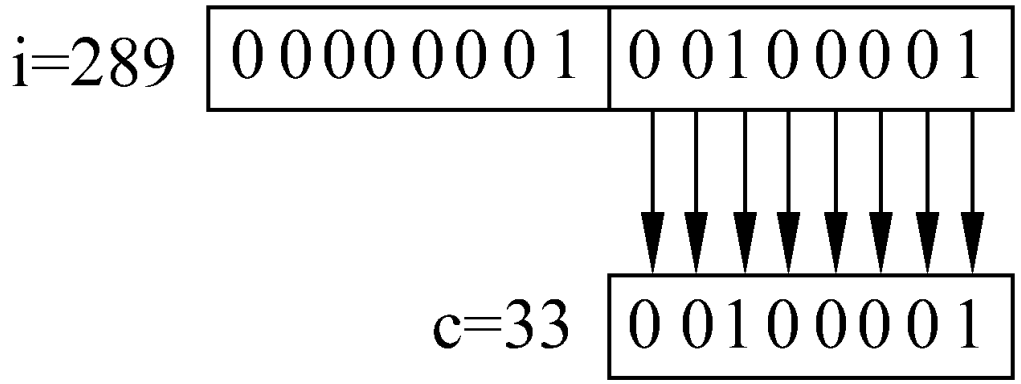


图**2.8**

(6) 将**signed(**有符号**)**型数据赋给长度相同的**unsigned(**无符号**)**型变量，将存储单元内容原样照搬（连原有的符号位也作为数值一起传送）。

例**2.5** 将有符号数据传送给无符号变量。

```
#include <iostream>
using namespace std;
int main( )
{ unsigned short a;
  short int b=-1;
  a=b;
  cout<<"a="<<a<<endl;
  return 0;
}
```

运行结果为

65535

赋给**b**的值是**-1**，怎么会得到**65535**呢？请看图**2.9**所示的赋值情况。

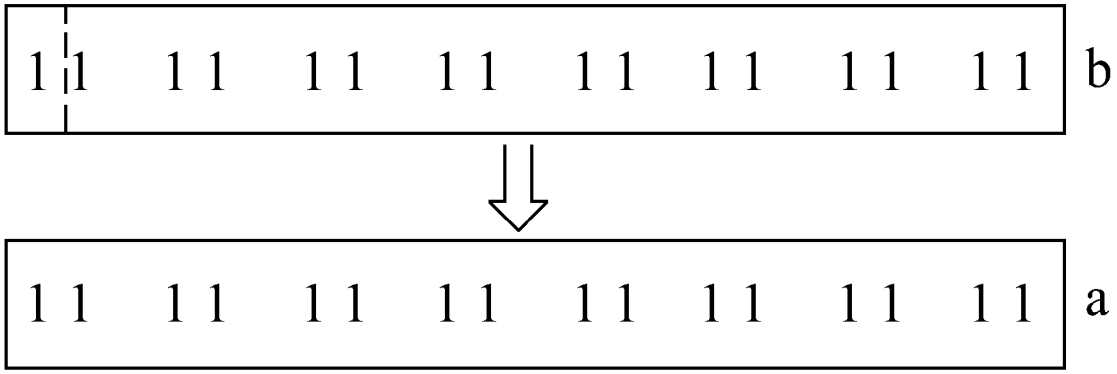


图**2.9**

-1的补码形式为**1111111111111111(**即全部**16**个二进制位均为**1)**，将它传送给**a**，而**a**是无符号型变量，**16**个位全**1**是十进制的**65535**。如果**b**为正值，且在**0~32767**之间，则赋值后数值不变。

不同类型的整型数据间的赋值归根结底就是一条：按存储单元中的存储形式直接传送。

C和**C++**使用灵活，在不同类型数据之间赋值时，常常会出现意想不到的结果，而编译系统并不提示出错，全靠程序员的经验来找出问题。这就要求编程人员对出现问题的原因有所了解，以便迅速排除故障。

2.6.3 复合的赋值运算符

在赋值符“＝”之前加上其他运算符，可以构成复合的运算符。如果在“＝”前加一个“＋”运算符就成了复合运算符“＋＝”。例如，可以有

a+=3　　等价于 **a=a+3**

x*=y+8　　等价于 **x=x*(y+8)**

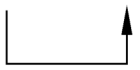
x%=3　　等价于 **x=x%3**

以“**a+=3**”为例来说明，它相当于使**a**进行一次自加**3**的操作。即先使**a**加**3**，再赋给**a**。同样，“**x*=y+8**”的作用是使**x**乘以（**y+8**），再赋给**x**。

为便于记忆，可以这样理解：

① **a+= b**　　**(**其中**a**为变量，**b**为表达式**)**

② **a+= b**　　**(**将有下划线的**“a+”**移到**“=”**右侧**)**

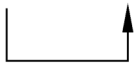


③ **a = a + b**　　（在**“=”**左侧补上变量名**a**）

注意，如果**b**是包含若干项的表达式，则相当于它有括号。如

① **x %= y+3**

② **x %= (y+3)**



③ **x = x%(y+3)**（不要错认为**x=x%y+3**）

凡是二元（二目）运算符，都可以与赋值符一起组合成复合赋值符。**C++**可以使用以下几种复合赋值运算符：

＋＝，－＝，*＝，／＝，％＝，<<＝，>>＝，&＝，∧＝，|=

其中后**5**种是有关位运算的。

C++之所以采用这种复合运算符，一是为了简化程序，使程序精炼，二是为了提高编译效率（这样写法与“逆波兰”式一致，有利于编译，能产生质量较高的目标代码）。专业的程序员在程序中常用复合运算符，初学者可能不习惯，也可以不用或少用。

2.6.4 赋值表达式

由赋值运算符将一个变量和一个表达式连接起来的式子称为“赋值表达式”。

它的一般形式为

<变量> <赋值运算符> <表达式>

如“a =5”是一个赋值表达式。对赋值表达式求解的过程是：先求赋值运算符右侧的“表达式”的值，然后赋给赋值运算符左侧的变量。一个表达式应该有一个值。赋值运算符左侧的标识符称为“左值”(left value，简写为lvalue)。并不是任何对象都可以作为左值的，变量可以作为左值，而表达式a+b就不能作为左值，常变量也不能作为左值，因为常变量不能被赋值。

出现在赋值运算符右侧的表达式称为“右值”(**right value**，简写为**rvalue**)。显然左值也可以出现在赋值运算符右侧，因而左值都可以作为右值。如

```
int a=3,b,c;
b=a;                    // b是左值
c=b;                    // b也是右值
```

赋值表达式中的“表达式”，又可以是一个赋值表达式。如

```
a=(b=5)
```

下面是赋值表达式的例子：

```
a=b=c=5          （赋值表达式值为5，a，b，c值均为5）
a=5+(c=6)        （表达式值为11，a值为11，c值为6）
a=(b=4)+(c=6)    （表达式值为10，a值为10，b等于4，c等于6）
a=(b=10)/(c=2)   （表达式值为5，a等于5，b等于10，c等于2）
```

请分析下面的赋值表达式：

(a=3*5)=4*3

赋值表达式作为左值时应加括号，如果写成下面这样就会出现语法错误：

a=3*5=4*3

因为**3*5**不是左值，不能出现在赋值运算符的左侧。

赋值表达式也可以包含复合的赋值运算符。如

a+=a-=a*a

也是一个赋值表达式。如果**a**的初值为**12**，此赋值表达式的求解步骤如下：

① 先进行**“a-=a*a”**的运算，它相当于**a=a-a*a=12-144＝-132**。

② 再进行“**a+=-132**”的运算，它相当于**a=a+(-132)＝－132-132＝-264**。

C++将赋值表达式作为表达式的一种，使赋值操作不仅可以出现在赋值语句中，而且可以以表达式形式出现在其他语句（如输出语句、循环语句等）中。这是**C++**语言灵活性的一种表现。

请注意，用**cout**语句输出一个赋值表达式的值时，要将该赋值表达式用括号括起来，如果写成“**cout<<a=b;**”将会出现编译错误。

2.7 逗号运算符与逗号表达式

C++提供一种特殊的运算符——逗号运算符。用它将两个表达式连接起来。如

3+5,6+8

称为逗号表达式，又称为“顺序求值运算符”。逗号表达式的一般形式为

表达式 1 ，表达式 2

逗号表达式的求解过程是：先求解表达式1，再求解表达式2。整个逗号表达式的值是表达式2的值。如，逗号表达式

a=3*5,a*4

从附录**B**可知：赋值运算符的优先级别高于逗号运算符，因此应先求解**a=3*5**（也就是把**“a=3*5”**作为一个表达式）。经计算和赋值后得到**a**的值为**15**，然后求解**a*4**，得**60**。整个逗号表达式的值为**60**。

一个逗号表达式又可以与另一个表达式组成一个新的逗号表达式，如

(a=3*5,a*4),a+5

逗号表达式的一般形式可以扩展为

表达式1，表达式2，表达式3，**…**，表达式n

它的值为表达式n的值。

从附录**B**可知，逗号运算符是所有运算符中级别最低的。因此，下面两个表达式的作用是不同的：

① **x=(a=3，6*3)**

② **x=a=3,6*a**

其实，逗号表达式无非是把若干个表达式“串联”起来。在许多情况下，使用逗号表达式的目的只是想分别得到各个表达式的值，而并非一定需要得到和使用整个逗号表达式的值，逗号表达式最常用于循环语句（**for**语句）中，详见第**3**章。

在用**cout**输出一个逗号表达式的值时，要将该逗号表达式用括号括起来，如

```
cout<<(3*5,43-6*5,67/3)<<endl;
```

C和**C++**语言表达能力强，其中一个重要方面就在于它的表达式类型丰富，运算符功能强，因而使用灵活，适应性强。