以汉字起首的词条

按位或表达式

种类

表达式

语法

〈按位或表达式〉::=〈按位异或表达式〉

|〈按位或表达式〉|〈按位异或表达式〉

描述

按位或表达式是一种以按位或运算符|为运算符、以两个整类型的子表达式作为其运算分量的二元整类型表达式，它用于对两个运算分量的位模式逐位进行"或"运算，即，仅当两个运算分量的位模式中对应位均为0时，结果的相应位才为0；否则，结果的相应位为1。

按位或表达式的求值结合规则为自左而右，其优先级比其它按位表达式都低。C中另外还有一种与按位或表达式类似的逻辑或表达式。

隶属词条

表达式、逻辑与表达式

相关词条

按位与表达式，移位表达式，逻辑或表达式，字位

下属词条

|，按位异或表达式

按位异或表达式

种类

表达式

语法

〈按位异或表达式〉::=〈按位与表达式〉

|〈按位异或表达式〉^〈按位与表达式〉

描述

按位异或表达式是一种以按位异或运算符^为运算符、以两个整类型的子表达式作为其运算分量的二元整类型表达式，它用于对两个运算分量的位模式逐位进行"异或"运算，即，仅当两个运算分量位模式中对应位不相同时，结果的相应位才为1；否则，结果的相应位为0。

按位异或表达式的求值结合规则为自左至右，其优先级比按位与表达式低，比按位或表达式高。注意，C中没有提供与按位异或表达式对应的逻辑异或表达式。

隶属词条

表达式，按位或表达式

相关词条

移位表达式，～，字位

下属词条

^，按位与表达式

按位与表达式

种类

表达式

语法

〈按位与表达式〉::=〈相等类表达式〉

|〈按位与表达式〉&〈相等类表达式〉

描述

按位与表达式是一种以按位与运算符 & 为运算符、以两个整类型的子表达式作为其运算分量的二元整类型表达式。它用于对两个运算分量的位模式逐位进行"与"运算，即，仅当两个运算分量位模式中对应位均为 1 时，结果的相应位才为 1；否则，结果的相应位为 0。

按位与表达式求值的结合规则是自左至右。

隶属词条

表达式，按位异或表达式

相关词条

移位表达式，按位或表达式，逻辑与表达式，字位

下属词条

&，相等类表达式

按位运算符

种类

表达式

描述

按位运算符除～之外都是二元运算符(～是一元运算符)，用于对它所作用的两个运算分量进行按位运算。这些运算符中除了移位运算符<<与>>位于同一优先级上外，其余均处于不同的优先级。C 语言共提供了六个按位运算符，现将它们及其功能列出如下：

～　按位求补运算符，用于对运算分量逐位求补。

<<　左移位运算符，用于使左运算分量值左移右运算分量值所指定的位数。

>>　右移位运算符，用于使左运算分量值右移右运算分量值所指定的位数。

&　按位与运算符，用于对两个运算分量值的机器表示按位进行与运算。

^　按位异或运算符，用于对两个运算分量值的机器表示按位进行异或运算。

|　按位或运算符，用于对两个运算分量值的机器表示按位进行或运算。

这六个按位运算符都要求各自的一个或两个运算分量都必须是整类型的(即只能是字符、枚举或整数类型)。必要时，在进行实际按位运算之前要对运算分量作整提升。对于<<与>>这两个移位运算符，运算结果的类型为左运算分量整提升后的类型；对于&、^ 与 | 这三个按位逻辑运算符，由于两个运算分量要整提升为同一类型的，故运算结果的类型即为运算分量整提升后的类型；对于按位求补运算符～，运算结果类型亦为运算分量整提升后的类型。

实例

下面的程序用于把一整数翻译成由 0 和 1 组成的二进制数，以显示其位模式。该程序中使用了多个按位运算符：

```
#include<limits.h>
#include<stdlib.h>
#include<stdio.h>

/*PRINTBIT 用于把一无符号整数翻译成二进制串*/
void
PRINTBIT(unsigned int number)
{ unsigned int i;
  unsigned int bits=sizeof(int)*CHAR_BIT;      /*bits 中存放执行环境中 int 类型对象所占字
                                                 位数*/
  unsigned int ckecker=1;      /*ckecker 用于指示由右至左的位数，初置设为1*/

  ckecker<<=bits-1;      /*将 ckecker 中非零位移到其机器表示的最左边，这样做是为了移植
                           性*/

  /*下一语句从左至右打印各位，每四位之间留一空隔*/
  for(i=1;i<=bits;i++){
    putchar((number & ckecker)?'1','0';   /*打印 ckecker 非零位所对应的 number 中对应位*/
    if (i%4==0)
      putchar(' ');      /*每四位之间打印一空隔*/
```

```
    ckeckker>>=1;      1 * 准备打印 number 中下一位 * /
  }
  putchar('\n');
}

main(int argc,char * argv[])
{
  int n1,n2;

  if (argc != 3){
    fprintf(stderr,"Usage: example int1 int2\n");
    exit(EXIT_FAILURE);
  }

  n1=atoi(argv[1]);
  n2=atoi(argv[2]);

  printf("bit patterns for %d and %d:\n",n1,n2);
  PRINTBIT(n1);
  PRINTBIT(n2);

  printf("~%d:\n",n1);
  PRINTBIT(~n1);

  printf("~%d:\n",n2);
  PRINTBIT(~n2);

  printf("%d & %d:\n",n1,n2);
  PRINTBIT(n1 & n2);

  printf("%d ^ %d:\n",n1,n2);
  PRINTBIT(n1 ^ n2);

  printf("%d|%d:\n",n1,n2);
  PRINTBIT(n1|n2);

  return(EXIT_SUCCESS);
}
```

隶属词条

表达式，运算符

相关词条

一元表达式，移位表达式，按位与表达式，按位异或表达式，按位或表达式，字位

下属词条

~，<<，>>，&，^，|

按引用传递参数

种类

函数

描述

按引用传递又叫按地址传递、按单元传递、按引用调用、按地址调用、传地址等，是C语言的一种参数与变元传递方式。具有这种传递方式的变元在传递给相应的参数时，不是把变元的值传给(赋给)相应参数，而是把变元的地址传给(赋给)参数。在函数体中，所有对参数的操作实际上就是对相应变元的操作。

与Pascal等语言不同，在C语言中，决定参数传递方式的因素不是参数说明形式，而是参数的类型。以下几种类型的参数所对应的变元只能通过按引用传递方式传递给被调用函数：

- 数组类型
- 函数类型

由于对按引用传递参数在函数体中的操作就是对相应变元的操作，因此当希望在函数体中的操作不会改变变元的值时，应该把拟作为变元的对象的值拷贝给某一临时变量，再把这个临时变量作为实际变元传递给函数，尤其是在调用一函数时不知道被调用函数体中是否改变了有关参数的值而不希望改变变元的值时就可以这样做。

在系统为按引用传递参数分配存储单元时，只为它分配存储指针值所需要的单元数，因而占用存储空间很少；而在系统为按值传递参数分配存储单元时，它要为存储相应类型值分配所需要的存储空间，因而可能要为一个参数分配较多的存储单元，例如，当参数类型为较大的记录类型时。

隶属词条

参数，变元

相关词条

按值传递参数，数组类型参数与变元，函数类型参数与变元

按值传递参数

种类

函数

描述

按值传递又叫按值调用传值，是C语言的一种参数与变元传递方式，具有这种传递方式的变元在传递给相应参数时，先把变元的值求出来，再赋给(传给、拷贝给)相应的参数。在函数体中，所有对参数的操作均与相应变元无关，对参数值的改变不会影响到相应的变元。

与Pascal等语言不同，在C语言中，决定参数传递方式的因素不是参数说明形式，而是参数的类型。以下类型的参数所对应的变元只能通过按值传递方式传递给被调用函数：

- 整数类型
- 指针类型
- 字符类型
- 结构类型
- 浮点类型
- 联合类型
- 枚举类型

由于按值传递参数在被调用函数执行过程中不能直接访问相应变元，其结果也不能赋给相应的变元，因此，如果要在函数体中直接使用变元，那么就要把参数说明成指向这些类型对象的指针类型的，例如，如果要写一个用于交换两个结构变量值的函数SwapRec，那么这个函数的说明不能写成：

```
void SwapRec(Rec l,Rec r);
```

而必须改成：

```
void SwapRec(Rec *l,Rec *r);
```

其中Rec是通过typedef存储类区分符定义的一个结构类型的名字。

与按值传递方式相对的另一种参数传递方式是按引用传递。

隶属词条

参数，变元

相关词条

按引用传递参数，数值类型参数与变元，指针类型参数与变元，结构类型参数与变元，联合类型参数与变元

保无符号规则

种类

类型

描述

对于整提升转换，在许多UNIX系统支持下实现的C编译系统中实现的是与ANSI C中保值规则有区别的保无符号规则(有人亦称保符号规则)。

在运用这种规则时，如果某表达式一运算符两边的运算对象分别为unsiged short int(或unsiged char、枚举等)类型与signed类型，那么就将前一运算分量的类型提升成unsigned int类型，同时再把后一运算分量的类型转移成unsigned int类型，然后再进行实际的运算。

当某一表达式中一运算符的两个运算对象分别为unsigned int类型与long int类型时，可类似进行讨论。一般而言，当采用这种规则时，当一表达式中既有有符号对象又有无符号对象时，该表达式的计算结果就是无符号的。

在许多情况下，保无符号规则与保值规则对表达式的执行结果而言并无什么区别。但是，在有些情况下，施用保无符号规则可能产生很奇怪的结果。例如，设 usi 为一 unsigned short int 类型的变量，si 为一 signed int 类型的变量，再假定在计算表达式

usi－si

时二者的取值分别为2与3，那么这个表达式的计算结果不是－1（这是施用保值规则时求得的结果），而是一个很大的 unsigned int 类型的值。

备注

若两个运算分量均为有符号的类型或均为无符号的类型，则施用这两种规则所产生的结果完全相同。

隶属词条

类型转换，整提升

相关词条

保值规则

保值规则

种类

类型

描述

对于整提升，标准 C（ANSI C）采用了保值规则。这种转换规则保持被提升类型对象的值不变。在使用这种规则时，如果某表达式一运算符两边的运算对象分别为 unsigned short int（或 unsigned char、枚举）类型与 singed int 类型，那么，当 unsigned short int（或 unsigned char、枚举）类型的所有可能取值在 signed int 的取值范围内时，就将 unsigned int（或 unsigned char、枚举）类型转换成 signed int 类型；而当 signed int 类型的取值范围不包括被转换类型的取值范围时，就将之转换成 unsigned int 类型。当某表达式中一运算符的两运算对象分别为 unsigned int 类型与 long int 类型时，可类似进行讨论。

实例

设有如下程序段：

```
long int li;
unsigned int ui;
...
li=ui+li;
```

在其中的加法运算中，在实际加运算执行前，如果 long int 类型可以存放 unsigned int 类型的所有可能取值，那么就首先把 ui 提升转换成 long int 类型，然后进行长整数加法运算并把结果赋给变量 li。如果 long int 类型不足以表示 ui 所存放的值（注意到在某些 C 编译系统中 int 类型与 long int 类型占同样大小），那么就首先把 ui 与 li 均转换成 unsigned long int 类型，然后再进行实际加法运算。

隶属词条

类型转换，整提升

相关词条

保无符号规则

编译

种类

技术术语

描述

编译指把源程序翻译成目标程序，这一工作是由编译程序完成的。编译程序是这样一种翻译程序，它把高级语言源程序（即 C 源程序）作为输入，经过编译处理后产生机器语言程序，这种机器语言程序能完成源程序所要求的功能。经过编译后产生的目标语言程序不能直接执行，而必须在运行之前先用连接程序把分散在各个目标程序文件中的目标程序以及函数库中的有关函数连接起来后才能执行。编译一般要分成若干个阶段进行，如预处理、词法分析、语法分析、语义分析、代码优化与生成等。参见"翻译阶段"词条。

相关词条

翻译，解释，连接，翻译阶段

变元

种类

函数，预处理

语法

〈变元〉::=〈赋值表达式〉

描述

变元又叫实在变元或实在参数，用在函数调用或函数类宏调用中。一个函数调用或函数类宏调用中可以使用多个变元，这时各个变元之间要用逗号隔开，从而构成一个变元表达式表（单个变元是变元表达式表的特例）。在变元表达式表的两边要用一对圆括号括起来。例如，下面是几个包含变元的函数调用与函数类宏调用：

```
printf("%d\n",a>b? a:b)
swap(&x,&y)
fact(10)
display(abs(-5))
```

可见，不仅函数调用与函数类宏调用在变元表示方法上相兼容，而且两种调用的形式也一致。

但是，在调用的变元（参数）传递方式与效果上两者有很大的不同。两者相比，函数类宏调用的变元传递方法要简单一些，只要把宏定义中替换字符串（替换表）中出现的参数用相应调用中的对应变元替换。在替换过程中不需要对变元进行求值，而只要把变元当作字符串代进去即可。但在作函数调用时，则要先求出变元表达式的值，然后再把该值赋给（拷贝给）相应参数（实在参数）。

关于宏调用中变元的处理与使用方法已在有关词条中作了详细讨论，下面主要介绍函数调用中变元的使用与处理方法。

C 中变元与参数的传递方法是按值传递。如前所述，编译程序在处理函数调用中的变元时是把求得的变元的值赋给函数中对应的参数。因此，对应参数的值在函数体内可以再次改变，即参数在函数体内可以象普通变量一样使用。但是，对函数值的改变不会影响到相应的变元，即使该变元是一个变量。

按值传递变元方法既有优点也有缺点，一方面，任何赋值表达式都可以作为变元传递（而不只限定为变量），而且在变元只是单个变量时可以避免在函数体内对它所作的不必要的修改（赋值）。但另一方面，这种方法也阻止（不利于）把函数执行中需要返回的信息通过变元传回给函数的调用者。从而，按值传递实际上是一种单向信息传递，尽管变元传递方式基本上是按值传递，但对不同数据类型的变元在具体处理上也稍有不同。

对于字符类型、枚举类型、整数类型、浮点类型的参数所对应的变元表达式，要先把表达式的值计算出来，然后再将之拷贝给相应参数。在函数体中对参数值的修改不会影响到相应变元的原有值，即使相应变元表达式只是一个变量。

对于指针类型参数所对应的变元表达式，也是先把表达式的值计算出来，再赋给相应参数，这使得参数与变元指向同一对象。这样，即使在函数体中对指针参数值的改变不会影响到相应变元，但是，如果在函数体中不改变指针参数的值而只是改变参数所指向对象的值，那么这种改变也影响相应变元所指向的对象的值，即对参数所指向对象值的修改实际上就是对相应变元所指向对象值的修改。因此，尽管变元对参数的传递方式是按值传递，但可以利用指针实现按引用传递（或称按地址传递、按位置传递）。例如，当要按引用传递整数变量时，可以以指向整数类型的指针类型作为参数类型，而以前冠'&'字符的整数类型变量作为其相应变元。在调用库函数 scanf 时就需要这样使用变元：

```
scanf("%d",&I);
```

其中 I 为一整数类型的变量。

对于数组类型参数所对应的变元表达式，变元对参数的传递方式与前述各种类型不同。变元表达式必须能指名一个数组而不是指定一个数组元素，当把一数组作为变元传递给函数时，并不是把数组（元素）的值传递给函数，而是把该变元数组解释成该数组第一个元素的地址。当函数调用时，这个地址被拷贝（赋值）给对应的（形式）参数，从而，参数就成了指向数组第一个元素的指针，这即是说对数组变元实现的是按引用传递的方式。当在函数体中引用数组元素时，该元素的下标值被加到参数原有值上并以相加的和作为地址来访问这个元素。这样，在函数体内就可以访问数组的任一元素。而且，在函数体内对数组参数元素的改变也是对相应变元中数组元素值的改变。

调用函数时也可以把数组的一部分传递给相应参数，这时要把数组所要传送的那部分的第一个元素的地址作为函数调用的变元。这样，该数组从这个变元开始的以后各个元素就被传递给函数。参见"数组参数"词条。

对于结构类型与联合类型，在较早的 C 语言版本中不允许使用结构类型及联合类型参数，即不允许传递

整个结构与联合对象。在需要这样做时要通过指向结构(或联合)类型对象的指针进行。但在包括 ANSI C 在内的一些较新的 C 编译环境中则允许把整个结构(或联合)作为变元直接传递给函数。在这种情况下,传递是按值进行的,而不是按引用进行的,即不是把变元的地址传递给参数,而是把值本身直接拷贝给参数。从而,在函数体内对参数值的任何改变都不会影响到相应的变元所命名对象的值。

尽管在一些较早的 C 编译环境中不允许使用函数参数(即不允许把一函数作为另一函数的参数进行传递),而只能通过指向函数的指针来间接传递,但却可将一函数命名符作为变元传递给被调用函数,当然这样传送的并不是函数的整个代码,而是被传递函数的入口地址。

另外,由于指针与一维数组类型之间的特殊关系,一指针参数可以对应于数组变元。反过来,一指针参数也可以对应于指针变元,只要指针所指对象与数组元素两者具有相同的类型。

在函数调用时,如果变元的类型与参数的类型不相容,那么这种调用是非法的。如果两者虽不相同但却相容,那么就要进行必要的类型转换。如果在处理函数调用时所处理过的对应的函数说明或函数定义不是函数原型式的,那么当变元类型为整类型时,要进行整提升;当变元类型为浮点类型时,要提升为 double 类型的。这种提升叫做缺省变元提升。

对变元的另一个要求是,一个调用中变元的数目应与相应函数说明或定义中参数的数目相同。

实例

在"函数"、"函数调用"、"递归函数"等词条中有大量函数调用中变元用法的例子可供参考。

下面再举一个结构作为变元的例子。下面这个程序主要用于说明将结构作为变元与将指向结构的指针作为变元的区别,程序如下:

```
#include<stdio.h>

typedef struct{
        char *name;
        int acct_no;
        char acct_type;
        float balance;
      } person_record;

main()
{
  void adjust1(person_record customer);
  void adjust2(person_record *customer);

  static person_record customer={"Wang Yuan",1234,'c',12.34};

  printf("%s %d %c %2f\n",customer.name,customer.acct_no,customer.acct_type,customer.balance);

adjust1(customer);
printf("%s %d %c %2f\n",customer.name,customer.acct_no,customer.acct_type,customer.balance);

adjust2(&customer);
printf("%s %d %c %2f\n",customer.name,customer.acct_no,customer.acct_type,customer.balance);
}

void adjust1(person_record customer)
{
  customer.name="Xu Hong";
  customer.acct_no=9876;
  customer.acct_type='w';
  customer.balance=98.76;
  return;
}

void adjust2(person_record *pt)
{
  pc->name="Huang He";
  pc->acct_no=7667;
  pc->acct_type='r';
  pc->balance=76.67;
```

}

本程序中的 main 函数共三次打印记录 customer 的值，其中第一次打印的是它的初值：

Wang Yuan 1234 C 12.34

第二次是在执行了对函数 adjust1 的调用之后，这次打印的 customer 的值仍然是：

Wang Yuan 1234 C 12.34

这是因为 adjast1 函数执行时所改变的只是参数的值，对参数值的改变不会影响到相应的变元，即在调用前后，在 main 中所定义的静态变量 customer 的值保持不变。

第三次执行 printf 函数是在调用函数 adjst2 之后。由于在 adjust2 函数执行时通过指针修改了静态变量 customer 的值，这次所打印的内容变成：

Huang He 7667 r 76.67

如果在这之后再加入两个与调用 adjust2 函数的语句前面的两个语句完全相同的两个语句，那么还要将第三次所打印的内容再打印一次：

Huang He 7667 r 76.67

隶属词条

函数调用

相关词条

参数，按值传递参数，按引用传递参数

标号

种类

语句

描述

标号从语法上看就是标识符，它用于命名标记语句（而且只能用来命名语句）。

有两处可以使用标号，其一是用于命名语句以构成复合语句，其使用形式为：

〈标号〉:〈语句〉

其二是在 goto 语句中用于指定 goto 语句所要转向到的位置（标号标记处），其使用形式为：

goto〈标号〉;

由于标号定义（在带标号语句中）与标号引用（在 goto 语句中）与标识符的其它用法在语法上属于不同的上下文，从而作为标号的标识符可以与作为其它用途的标识符用在同样的行文区间中而不会引起歧义性或其它麻烦，即标号具有独立的名字空间。

在同一函数内两个带标号语句不得有相同的标号。

隶属词条

goto 语句，带标号语句

相关词条

标识符，名字空间，作用域

标记

种类

结构，联合，枚举

语法

〈标记〉::=〈结构标记〉
　　|〈联合标记〉
　　|〈枚举标记〉

〈结构标记说明〉::=struct〈标记标识符〉|〈结构说明表〉|;
　　|struct〈标记标识符〉;

〈联合标记说明〉::=union〈标记标识符〉|〈联合说明表〉|;
　　|union〈标记标识符〉;

〈枚举标记说明〉::=enum〈标记标识符〉{〈枚举符表〉}；
|enum〈标记标识符〉；

描述

标记分为结构标记、联合标记与枚举标记等三种，分别用于标记结构、联合与枚举。

标记是通过标记说明来说明的。标记说明由三部分组成：种类引入符 struct、union 或 enum，所要命名的标记的名字以及用于定义结构内容、联合内容或枚举内容的结构说明表、联合说明表或枚举符表。例如，下面的说明就说明了一个名为 struct_tag 的结构标记：

```
struct struct_tag
{
    char *f1;
    iwt f2;
    double f3;
};
```

如果去掉该说明最末的一个分号，那么其余部分实际上就是结构区分符。如果要用 struct_tag 说明一个结构类型 struct_type 与一个结构对象 struct_object，那么可以分别使用如下说明：

```
typedef struct struct_tag struct_type;
struct struct_tag struct_object;
```

有了对 struct_type 的定义，对 struct_object 也可以用如下说明：

```
struct_type struct_object;
```

这两个说明所说明的 struct_object 是等价的。

从上面对结构标记的使用可以看出，结构标记只是个标记，不是结构类型名字，也不是类型区分符，因此不能单独使用，而必须跟随关键字 struct 两者构成(结构)类型区分符后才能用在说明中定以其它实体。对联合标记与枚举标记亦如此。

在说明结构(或联合或枚举)标记时也可同时说明相应对象，例如：

```
union int_or_float
{
    int i;
    float f;
} if1,if2;
```

但是，如果前面已对某标记进行了说明，那么在说明该标记所标记的类型的对象时，说明中不得包含一对花括号括住的部分。例如，如下说明是非法的(假定此前已有如前所述的对 struct_tag 的说明)：

```
struct struct_tag
{
    char *f1;
    int f2;
    double f3;
}struct_object1,struct_object2;
```

然而，如果把其中的结构标记 struct_tag 或用花括号括住的结构说明表两者中去掉一个，则这个说明就是正确的。这是因为该说明在说明对象 struct_object1与 struct_object2的同时又对 struct_tag 再次作了定义，这是不允许的。

上述对标记的说明是完整说明，对它们还可以作不完整说明，即只说明标记的名字但不指定它所代表的内容。例如，

```
struct s_tag;
```

就是对结构标记的不完整说明。通过不完整说明所说明的标记只能用在不需要该标记所指定的类型的对象的大小的情况下。例如，此种标记可以用在通过 typedef 定义类型名字的说明中，也可以用在说明指向该标记所指定的类型的指针的说明中。即有了上述对 s_tag 的不完整说明，就可以作如下说明：

```
typedef struct s_tag s_type;
struct s_tag *ps;
```

但如下说明是不正确的：

```
struct s_tag s_object;
```

除非该说明不只位于对 s_tag 的不完整说明之后，而且也位于对 s_tag 的后继完整说明之后。

正如对其它不完整说明一样，对标记的不完整说明之后必须有对标记的完整说明。但有了对标记的完整说明之后，则不能再对之作不完整说明。

在对标记的不完整说明中也可以同时说明其它实体。例如，下一说明既不完整说明了标记 u_tag，也说明了指向它的指针 pu：

```
union u_tag * pu;
```

对枚举不需要作不完整说明，因为标记的不完整说明主要是为相互依赖的说明而存在的，而在枚举类型与其它类型的说明间不存在这种关系。

隶属词条

结构说明、联合说明、枚举类型

相关词条

名字空间、作用域

标识符

种类

词法元素

语法

〈标识符〉::=〈字母〉|〈下划线〉|〈标识符〉〈字母〉|〈标识符〉〈数字〉|〈标识符〉〈下划线〉

〈字母〉::=〈大写字母〉|〈小写字母〉

〈大写字母〉::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

〈小写字母〉::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

〈数字〉::=0|1|2|3|4|5|6|7|8|9

〈下划线〉::=_

描述

标识符是用来标识变量、常量、类型、函数、标号、宏、参数、文件名等实体的按一定规则组织起来的字符序列。C 的标识符与其它语言相比，有如下几个不同之点：

①一般语言中标识符必须以字母打头，而 C 语言中除了允许以字母打头外还允许以下划线打头，如

```
_PASCAL_, _DATE, _AX
```

②一般语言中虽然也允许下划线出现在标识符中，但不允许在一标识符中两下划线相连，而这在 C 中则允许，例如：

```
_ TIME_ , _ STDC_ _, My_ _Name
```

③一般语言中对标识符的大小写不加区分，即认为在标识符号大写字母与相应小写字母被认为是同一字母，但 C 中却要严格区分大小写字母，这样，下面是三个各不相同的标识符：

```
sum, SUM, Sum
```

在标识符定义与使用时应严格注意这一点。

④一般语言标准对标识符中有效字符个数基本未作规定，但 ANSI C 规定编译系统至少应区分一标识符的前31个字符(如果标识符用作外部连接的文件名字等，则编译系统至少应区分前6个字符)。当然，以前的许多编译系统并不符合这一规定，有的规定最少应区分前6个字符，有的则规定最少应区分前8个字符，C++未对此作统一规定，但 Turbo C++与 Borland C++只识别前32个字符，当然这个数字可随用户需要减小(用选择项)，但不能增大。

每一个标识符都有一定的作用范围，即作用域。所谓标识符的作用域指该标识符可以使用的行文区域。一个标识符的作用域可以是整个程序，某个文件，某个函数或某个局部分程序，这要视该标识符是在何处定义的，用什么方式定义的。在 C 语言中，有些标识符是系统预定义的，它们在程序中有固定的含义，如 printf、open、ctype、_cs、DBL_EPSILON 等。预定义标识符可以在程序中直接使用(当然，可能要用头文件等引入)，程序员在程序中也可以对它们重新定义。重新定义的标识符在定义的作用域内具有新定义的含义而不再具有预定义的含义。

有些字符串尽管在语法上也符合标识符语法，但不能作为标识符定义，这便是关键字(亦称保留字)，如 auto、int、typedef、while 等。

风格

在使用标识符时应注意如下几点：

①标识符应尽可能与它所代表的实体相一致，选择有明确含义的英文单词(或其缩写)作为标识符，做到"见名知义"，以提高程序的可读性，这样，诸如

```
a,b,c,x1,y1,y2
```

等均是不理想的标识符，而

```
Apple,Year,Month,City,Classes
```

则是比较理想的标识符。

②尽管 C 严格区分标识符中字母的大小写，但作为一种好的风格，最好不要使用仅在字母的大小写上有区别的不同标识符。例如，在同一作用域内最好不要同时使用如下三个不同的标识符：

```
SCHOOL,School,school
```

③尽管 C 允许一标识符可以以下划线打头并允许在同一标识符中有两个或多个下划线相连，但是，在 C 语言中，以下划线打头的标识符一般为特殊目的所用，如，用作预定义的宏名字，因此用户所取标识符最好不要以下划线打头，而仅以字母打头。同样，除非为了特殊目的或某种风格，在标识符中最好不要出现两下划线相连的情况。如无特殊需要，一般也不要把下划线用作标识符的最后一个字符。

④尽管 ANSI C 规定编译程序应能识别较长的标识符，且许多编译程序也这样做了。但为了保证程序的可移植性，在命名标识符时最好还是要考虑到个别编译程序的情况，不妨假定编译程序只能区别前6～8个字符，即假定标识符的前6～8个字符是有效字符。例如，应该把

```
TESTNUMBERONE 与 TESTNUMBERTWO
```

当作同一标识符(尽管不少编译程序能识别出这是两个标识符)，这两个标识符可用如下形式命名：

```
TEST_ONE,TEST_TWO
```

⑤当以英文单词来命名标识符时，可以让其中第一个字母大写，其余小写，如：

```
Programming,Name
```

当一个标识符由两个或两个以上单词构成时，每个单词的首字母均大写，其余小写，如：

```
ProgrammingLanguage,MyName
```

也可以在单词间插入下划线：

```
Programming_Language,My_Name
```

当一标识符是由某个或某几个单词缩写而成时，最好全用大写，如：

```
PRG,CNT
```

当然，也有不同的建议，如，有人认为，变量名、函数名与数据类型名标识符中的字母用小写，而符号常量名与宏名标识符中的标识符用大写。在取名时，变量名、类型名，常量名、有返回值的函数名最好用名词或名词短语作为标识符命名；而作为过程使用的不带返回值的函数最好用动词、动词短语或祈使子句作为标识符命名。此外，对有返回值的函数，如其结果类型为布尔型，则其名字应能表明其值具有判定性的真值的含义，如：

```
Completed,Is_Ready
```

隶属词条

词法元素

相关词条

关键字，作用域，名字空间

下属词条

预定义标识符

标准头文件

种类

术语，头文件

描述

标准头文件指在 ANSIC 中预定义的头文件，共有十五个，这十五个头文件的名字及相应功能如下：

assert. h	用于运行时断言检查	signal. h	用于异常信号处理
ctype. h	用于字符处理	stdarg. h	用于处理可变数目变元
errno. h	用于错误处理	stddef. h	用于定义公共的宏与类型
float. h	用于描述对浮点数的限制	stdio. h	用于定义标准输入输出函数
limits. h	用于描述一般实现限制	stdlib. h	用于定义通用实用函数
locale. h	用于建立与修改局部环境	string. h	用于字符串处理
math. h	一般数学函数库	time. h	用于时间与日期处理
setjmp. h	用于在程序中非局部跳转		

在具体实现中可能还会定义其它头文件，也可能会在上述标准文件中扩充定义新的宏、类型或函数。

隶属词条

头文件、库函数

相关词条

头文件名字

下属词条

assert. h，ctype. h，errno. h，float. h，limits. h，locale. h，math. h，setjmp. h，signal. h，stdarg. h，stddef. h，stdio. h，stdlib. h，string. h，time. h，库函数

表达式

种类

表达式

语法

〈表达式〉::=〈赋值表达式〉
|〈表达式〉，〈赋值表达式〉
〈赋值表达式〉::=〈条件表达式〉
|〈一元表达式〉〈赋值运算符〉〈赋值表达式〉
〈赋值运算符〉::=＝|＊＝|/＝|％＝|＋＝|－＝|<<＝|>>＝|&＝|＾＝|｜＝
〈条件表达式〉::=〈逻辑或表达式〉
|〈逻辑或表达式〉？〈表达式〉：〈条件表达式〉
〈逻辑或表达式〉::=〈逻辑与表达式〉
|〈逻辑或表达式〉｜｜〈逻辑与表达式〉
〈逻辑与表达式〉::=〈按位或表达式〉
|〈逻辑与表达式〉&&〈按位或表达式〉
〈按位或表达式〉::=〈按位异或表达式〉
|〈按位或表达式〉｜〈按位异或表达式〉
〈按位异或表达式〉::=〈按位与表达式〉
|〈按位异或表达式〉＾〈按位与表达式〉
〈按位与表达式〉::=〈相等类表达式〉
|〈按位与表达式〉&〈相等类表达式〉
〈相等类表达式〉::=〈关系表达式〉
|〈相等类表达式〉＝＝〈关系表达式〉
|〈相等类表达式〉！＝〈关系表达式〉
〈关系表达式〉::=〈移位表达式〉
|〈关系表达式〉<〈移位表达式〉
|〈关系表达式〉>〈移位表达式〉
|〈关系表达式〉<＝〈移位表达式〉
|〈关系表达式〉>＝〈移位表达式〉
〈移位表达式〉::=〈加法类表达式〉
|〈移位表达式〉<<〈加法类表达式〉

```
        |〈移位表达式〉>>〈加法类表达式〉
〈加法类表达式〉::=〈乘法类表达式〉
        |〈加法类表达式〉+〈乘法类表达式〉
        |〈加法类表达式〉 -〈乘法类表达式〉
〈乘法类运算符〉::=〈强制转换表达式〉
        |〈乘法类表达式〉*〈强制转换表达式〉
        |〈乘法类表达式〉/〈强制转换表达式〉
        |〈乘法类表达式〉%〈强制转换表达式〉
〈强制转换表达式〉::=〈一元表达式〉
        |(〈类型名字〉)〈强制转换表达式〉
〈一元表达式〉::=〈后缀表达式〉
        |++〈一元表达式〉
        |--〈一元表达式〉
        |〈一元运算符〉〈强制转换表达式〉
        |sizeof(--元表达式)
        sizeof(〈类型名字〉)
〈一元运算符〉::=&|*|+|-|~|!
〈后缀表达式〉::=〈初等表达式〉
        |〈后缀表达式〉[〈表达式〉]
        |〈后缀表达式〉() | 〈后缀表达式〉(〈变元表达式表〉)
        |〈后缀表达式〉.〈标识符〉
        |〈后缀表达式〉->〈标识符〉
        |〈后缀表达式〉++
        |〈后缀表达式〉--
〈变元表达式表〉::=〈赋值表达式〉
        |〈变元表达式表〉,〈赋值表达式〉
〈初等表达式〉::=〈标识符〉
        |〈常量〉
        |〈字符串字面值〉
        |(〈表达式〉)
```

描述

表达式是由运算分量(又称操作数、运算对象)、运算符与括号按照一定的规则组成的语言成分。表达式可用于求值、调用函数,产生副作用,或用于同时完成这些功能中的两种或全部。

表达式用于表示(求得)作为值的数据项,如,可用于表示一个数或一个字符。最简单的表达式只包含一个运算分量,如常量、变量、数组元素,函数引用等都是表达式的特例,更复杂的表达式中则要包含各种运算符,必要时还要用括号来改变求值次序。

在C语言中,表达式的组成规则与语义同其它常见语言基本类似,表达式除了可用于表示数值值、字符值、枚举值、聚集值(由简单值复合而成的值)外,也可用于表示逻辑值(亦称布尔值)。但是,C没有把逻辑值作为一种特殊值来处理(象Pascal、Ada等那样),而是把逻辑值真与假分别用整数值1与0来表示,因此,逻辑表达式实际上就是一种特殊的数值表达式,从而,对逻辑表达式的值(或逻辑值)可以象对普通数值一样施行数值(算术)运算与关系运算。

不仅如此,对字符类型(char类型)也可以施行各种算术运算及其它运算,因为在C中是把字符类型当作整数类型处理的。

C语言的表达式功能很强,也很复杂,它提供了许多种运算符,这些运算符中既有可用于进行独立于机器的诸如加减乘除等抽象运算符,也有用于进行低级机器运算的按位运算符,这些运算符用于组识各种各样的子表达式。

最基本的表达式是初等表达式,它可以是变量、常量、(包括字符串字面量),也可以是用一对圆括号括起来的一般表达式。

由初等表达式加上后缀运算符(及其所必须具有的其它表达式成分)就构成了后缀表达式,再在后缀表达式的前面加上诸如一元运算符等运算符号就可以构成一元表达式,C的一元运算符要比诸如Pascal、FORTRAN等语言丰富些。对一元表达式施行乘法类运算可以构成乘法类表达式,而对乘法类表达式施行加法类运算则可以构成加法类表达式。如果再对加法类表达式施行左右移位运算,那么就可以得到移位表达式。

对上述任一种子表达式施行关系运算可以得到关系表达式。这里的关系表达式仅指施行大小次序比较运算的表达式，而不把用于进行相等或不等测试的表达式（在 C 中叫相等类表达式）包含在内。对这些表达式施行按位逻辑运算可以得到按位表达式。如果再对它们进行逻辑运算，那么可以得到逻辑表达式，在逻辑表达式之上还可以再构成条件表达式。

在 C 中赋值号也被当成一种运算符，所得到的表达式叫做赋值表达式。赋值表达式就是最高"层次"的表达式。

与许多语言不同的是，任何表达式都可以作为语句出现在程序中应该（可以）出现语句的位置，即表达式可以作为语句使用，即所谓的表达式语句。

对表达式中各种运算的求值次序视运算符的不同而有所区别。首先，C 中的运算符被分成了各种不同的优先级。有些运算符具有相同的优先级，例如，加法与减法运算符具有同一优先级，相等与不等运算符具有相同的优先级。但绝大多数运算符都不在同一优先级上，例如，加法类运算符就与乘法类运算符不在同一优先级上，前一类运算符的优先级比后一类运算符的优先级低。当不同优先级的运算符出现在同一表达式中时，优先级高的运算先进行。

其次，在同一优先级内，两个相连的运算之间也有一定的次序要求，否则有可能会得到不同结果，例如，5－4－3这一表达式中，如果从左到右计算，那么计算结果为－2；如果从右至左计算，那么计算结果为4。C 中一般要求对同一优先级的运算要从左至右进行，但对条件运算与赋值运算则从右至左进行，对前缀运算符亦要从右至左执行相应的运算。

事实上，运算符的优先级可以从语法部分给出的语法构成规则中看出来。而同一优先级的运算符所作用表达式的求值次序也可从中看出来。例如，从加法类表达式的语法

〈加法类表达式〉＋〈乘法类表达式〉

中就可以看出，应把 x＋y＊z 分析成等价于 x＋(y＊z)，即乘法运算符比加法运算符的优先级高，这是因为乘法类表达式是加法类表达式的运算分量，而不是反之。另外，从这个加法类表达式的语法可以看出，应把 x＋y－z 分析成等价于(x＋y)－z，即加法表达式应由左而右计算。如果把加法类表达式的语法改成

〈乘法类表达式〉＋〈加法类表达式〉

那么就应把 x＋y＋z 分析成等价于 x＋(y＋z)，这样就要由右至左计算。如果再把加法类表达式的语法改成

〈加法类表达式〉＋〈加法类表达式〉

注意到乘法类表达式是一种特殊的加法类表达式（从而可推得变量与常量也是特殊的加法类表达式），既可以把 x－y＋z 分析成等价于(x＋y)＋z，也可以把它分析成等价于 x＋(y＋z)，即这时的求值次序可以由左至右，也可以由右至左，从而就有了歧义性。另外，从赋值表达式的语法

〈一元表达式〉〈赋值运算符〉〈赋值表达式〉

中，可以看出，应把 a＝b＝c 分析成等价于 a＝(b＝c)，即对赋值运算符应由右而左计算。

如所知，在数学中，加减乘除四则运算有交换律、结合律、分配律等运算定律。在 C 语言程序中这些定律在运算分量为浮点数（浮点变量或浮点常量）时却不一定成立，因而 C 语言中没有接受这些定律。例如，从数学上讲，(x＋y)＋z 等于 x＋(y＋z)，但在 C 程序中，它们却不一定相等，设 x 是一个比较大的正浮点数，y 等于－x，z 是一个其绝对值比 x 小得多的负数（即假定|z/x|即|z/y|<DBL_EPSILON），则(x＋y)＋z 等于0＋z 即 z，而 x＋(y＋z)等于 x＋y 即0。可见两者并不相等。

在表达式中还有三个问题需要注意，一个是副作用，另一个是顺序点，还有一个是别名。

副作用并不是 C 语言所独有的，在几乎所有语言中几乎都有副作用的问题，但是在这些语言中副作用主要表现在函数的副作用上，即通过在函数体中给非局部变量的赋值而引起。在 C 语言中，不只在函数中可能会产生副作用，而且在表达式中也会产生副作用。表达式中的副作用除了象其它语言那样由于在表达式中调用了带副作用的函数而引起外，还有因使用了表达式中运算分量的左值而引起的副作用。例如，在表达式中使用加一运算符、减一运算符、各种赋值运算符都会产生副作用。在表达式语句

v ＝(s＝l＊l)＊l；

中，v 是求得的一正方体的体积，该语句除了计算 v 外，还同时求得了该正方体的一面的面积，这就是表达式的副作用。表达式的副作用有其正效应，如在上述表达式语句中那样，但也有其副效应。如果用得不好，可能会使程序难以阅读、理解、易于出错，因此，在利用表达式副作用时应尽量小心，避免产生不必要的负面效应。

顺便指出，由于语言没有规定诸如加减乘除运算符等二元运算符两个运算分量的求值次序（而只是规定了各个运算符的求值次序），编译程序在实现时可以先求左运算分量的值，也可以先求右运算分量的值（虽然

在一般实现时是先求左运算分量的值后求右运算分量的值)。从数学角度看,按哪种次序求值都是等价的。但在C程序中却不一定如此,请看下述表达式:

```
a=(b--)*b*1000
```

这个表达式中包含两个乘法运算,按照求值规则,自然应先做左边的乘法,但由于语言没有规定两个运算分量谁先计算,设在这个表达式计算前b的值为100,如果左运算分量先计算,那么这个表达式等价于求a=100*(100-1)*1000,求得的a值为9900000;反之,如果右运算分量先计算,那么这个表达式等价于求a=100*100*1000,求得的a值为10000000,后者比前者大100000!

为了解决这个问题,C语言中引入了顺序点的概念,所谓顺序点,指程序中这样一些位置,一个表达式中位于顺序点之前的成分必须在位于顺序点之后的成分之前求值,而不能颠倒过来。如在表达式

```
left_opd&&right_opd
```

中,left_opd必须在right_opd之前求值。但C中只为逻辑运算符与条件运算符等少数运算符引入了顺序点,对加减乘除等运算符则未引入顺序点,在使用时应注意(参见"顺序点"词条)。

别名也是表达式中必须慎重、小心的一个问题,别名由于使用了指针而引入,它会使函数体中的两个不同的对象因指针的使用而变成同一对象。考虑下例:

```
int x
void f1(int *y)
{
    x=1;
    *y=100;
    f2(x)
}
```

从表面上看,函数f1体中要用x的值来调用f2,即其中的f2(x)等价于f2(1)。但是,如果该程序段所在的程序中,调用f1的函数调用中的实在参数为指向x的指针,即如果在这一程序段的行文后面有如下两行语句:

```
Z=&X;
F1(Z)
```

那么,在执行F1时,其中的函数调用F2(x)就不再等于F2(1),而等于F2(100)了。

别名的存在使程序的优化变得麻烦了。

实例

下面给出各种表达式的例子。

初等表达式:

```
PI
3.1415926535
(x+y*z)>>5)
```

后缀表达式:

```
func(4,5,sum)
matrix_a[3,6]
a++
s--++
```

一元表达式:

```
--b
++--b++
&d
sizeof(double)
```

强制转换表达式:

```
(int)(3.1415926535*r*r)
(double)'c'
```

乘法类表达式:

```
l*w*h
(-b+sqrt(b*b-4*a*c))/(2*a)
(x+y-z)%(x-y+z)
```

加法类表达式:

```
b*b-4*a*c)
-b+sqrt(b*b-4*a*c)
34*5+g*(4+f-1)-x-y+a++
```

移位表达式:

```
x+y>>(int)sqrt(x*x-y*z)
a*c-d--<<'9'-'0'
```

关系表达式:

```
x<y<=z
b*b-4*a*c<=delta
5*(a+k)>(4-3*y)
```

相等类表达式:

```
a<=b==b<=a
a<=b!=b<=a
```

按位与表达式:

```
(x|y)&(x|y)
```

按位异或表达式:

```
a*b^x/y
```

按位或表达式:

```
--g+f(6)|a[4]
```

逻辑与表达式:

```
(a<=b)&&(b<=a)
```

逻辑或表达式:

```
a<b||b<a
```

条件表达式:

```
d=b*b-4*a*c?
root1=(-b+sqrt(d))/(2*a),
root2=(-sqrt(d))/2/a;
```

root1=root2=-b/2/a

赋值表达式：

m=n=f(x)/g(y)

v=s=3.14159*r*r*r/3

表达式：

s=3.14159*r*r,v=s*r/3

相关词条

语句，函数，副作用，常量，变量，别名

下属词条

运算符，运算分量，运算符优先级，表达式求值次序，顺序点，表达式语句，左值，右值，常量表达式，初等表达式，后缀表达式，一元表达式，强制转换表达式，乘法类表达式，加法类表达式，移位表达式，关系表达式，相等类表达式，按位与表达式，按位异或表达式，按位或表达式，逻辑与表达式，逻辑或表达式，条件表达式，赋值表达式。

表达式求值次序

种类

表达式

描述

一表达式中可以包含多个一元、二元与(或)三元运算符，可以包含多个运算，各个运算不能以任意顺序进行，而必须按照一定的次序逐步进行，表达式的求值次序可以从表达式的语法规则(参见"表达式"词条)推出，它具体可以通过三个规则来进行规范：

规则一：对于不同优先级的运算符，优先级高的运算符所相应的运算在优先级低的运算符所相应的运算之前进行(参见"运算符优先级"词条)。

这样，乘除运算要在加减运算之前进行，一元运算要在二元、三元运算之前进行，数值运算要在移位运算之前进行，移位运算要在关系运算之前进行，关系运算要在相等类运算之前进行，相等类运算要在按位运算之前进行，按位运算要在逻辑运算之前进行，逻辑运算要在条件运算之前进行，条件运算要在赋值类运算之前进行，赋值类运算要在逗号运算之前进行，如此等等。

规则二：对于两个相连的运算即使优先级相同，也不能任意进行。一般要求优先级相同的运算由左至右进行，但也有例外：

①两个相连的前缀运算要由右至左进行；

②两个相连的条件运算要由右至左进行；

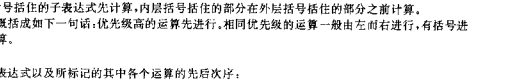
③两个相连的赋值类运算要由右至左进行。

规则三：用括号括住的子表达式先计算，内层括号括住的部分在外层括号括住的部分之前计算。

总而言之可概括成如下一句话：优先级高的运算先进行，相同优先级的运算一般由左而右进行，有括号进作括号里面的运算。

实例：

下面是一个表达式以及所标记的其中各个运算的先后次序：



当我们不知道各个运算符的优先级与各优先级运算符的求值顺序时，可以用括号来表示所需要的求值顺序，例如，表达式

-A+B%C==(-D&&E|F>>G?(H>>I+J):(H<<K))==(L>K*M<<N)

完全等价于如下任一表达式：

(-A+B%C)==((-D&&E|F>>G)?(H>>I+J):(H<<K))==(L>K*M<<N)

(-A+B%C)==((-L&&(E|(F>>G)))?(H>>I+J):(H<<K))==(L>K*M<<N)

(((-A)+(B%C))==(((-D)&&E|(F>>G))?(H>>(I+J)):(H<<K)))==(L>((K*M)<<N))

表达式语句

种类

语句

语法

〈表达式语句〉::=〈表达式〉;

描述

表达式语句的主体是表达式，由表达式后缀一个分号组成。

表达式语句的执行过程实际上就是其中表达式的计算过程。使用表达式的目的就是利用表达式的副作用来给某些对象(变量)赋值，但表达式的值在该语句执行完时被丢弃掉，即表达式语句中的表达式等价于一个 void 表达式。于是，语句

〈表达式〉;

等价于

(void)〈表达式〉;

如果一表达式语句中的表达式没有副作用，那么执行这个语句等于劳而无功，浪费机器时间。因此，在表达式语句中应包含加一运算符或减一运算符或赋值运算符或函数调用。

按照标准 C 的语法，表达式语句中的表达式可缺省，即表达式语句可以只由一个分号组成，但一般情况下，这种表达式语句叫做空语句。

实例

下面是几个由加一运算符与减一运算符构成的表达式语句：

```
i++;
--j;
(&a)++;
(*p)--;
```

下面几个表达式语句中都包含赋值运算符，这几个语句因此在其它语言中叫做赋值语句：

```
v=length * width * height;
(*p)+=1;
```

这两个表达式语句分别等价于下面两个表达式语句：

```
(void)(v=length * width * height);
(void)(*p+=1);
```

单独的函数调用也可作成一表达式语句，即使其结果类型不是 void 类型的，例如：

```
power(x,n);
(void)power(x,n);
```

这种语句在其它语言中叫做过程调用语句或函数调用语句或子程序调用语句。

隶属词条

语句

相关词条

表达式，副作用，void 表达式，空语句

表意常量

种类

预处理

描述

表意常量又叫表征常量、定义常量、其对应英文为 manifest constant，是一种具有定义名字的值，这是程序员及一般用户常用的术语，但在 ANSI C 等中一般称之为宏。

表意常量用 #define 指令定义。例如，由

```
#define MaxFiles 9
```

所定义的表意常量 MaxFiles 的值为9。在翻译程序的预处理阶段，翻译程序将把被翻译 C 程序中的所有 MaxFile 用字符"9"代替。

在 C 程序中，使用表意常量一般有两个优点：首先，当在程序中要修改由表意常量所表示的值时，只要修改表意常量的定义(及相关的 #define 指令)即可，其次，在程序中使用表意常量要比直接使用它所表示的值使程序更可读。例如，当人们阅读语句

```
if (File_Number>Max_Files)
……
```