● 如果一个运算符两边的运算数类型不同，先要将其转换为相同的类型，即较低类型转换为较高类型，然后再参加运算，转换规则如下图所示。

double ←—— float 高

↑

long

↑

unsigned

↑

int ←—— char,short 低

图中横向箭头表示必须的转换，如两个float型数参加运算，虽然它们类型相同，但仍要先转成double型再进行运算，结果亦为double型。 纵向箭头表示当运算符两边的运算数为不同类型时的转换，如一个long 型数据与一个int型数据一起运算，需要先将int型数据转换为long型， 然后两者再进行运算，结果为long型。所有这些转换都是由系统自动进行的， 使用时你只需从中了解结果的类型即可。这些转换可以说是自动的，但然，C语言也提供了以显式的形式强制转换类型的机制。

● 当较低类型的数据转换为较高类型时，一般只是形式上有所改变， 而不影响数据的实质内容， 而较高类型的数据转换为较低类型时则可能有些数据丢失。

赋值中的类型转换

当赋值运算符两边的运算对象类型不同时，将要发生类型转换， 转换的规则是：把赋值运算符右侧表达式的类型转换为左侧变量的类型。具体的转换如下：

(1) 浮点型与整型

● 将浮点数(单双精度)转换为整数时，将舍弃浮点数的小数部分， 只保留整数部分。

将整型值赋给浮点型变量，数值不变，只将形式改为浮点形式， 即小数点后带若干个0。注意：赋值时的类型转换实际上是强制的。

(2) 单、双精度浮点型

● 由于C语言中的浮点值总是用双精度表示的，所以float 型数据只是在尾部加0延长为double型数据参加运算，然后直接赋值。double型数据转换为float型时，通过截尾数来实现，截断前要进行四舍五入操作。

(3) char型与int型

● int型数值赋给char型变量时，只保留其最低8位，高位部分舍弃。

● char型数值赋给int型变量时， 一些编译程序不管其值大小都作正数处理，而另一些编译程序在转换时，若char型数据值大于127，就作为负数处理。对于使用者来讲，如果原来char型数据取正值，转换后仍为正值;如果原来char型值可正可负，则转换后也仍然保持原值， 只是数据的内部表示形式有所不同。

(4) int型与long型

● long型数据赋给int型变量时，将低16位值送给int型变量，而将高16 位截断舍弃。(这里假定int型占两个字节)。

将int型数据送给long型变量时，其外部值保持不变，而内部形式有所改变。

(5) 无符号整数

● 将一个unsigned型数据赋给一个占据同样长度存储单元的整型变量时(如：unsigned→int、unsigned long→long，unsigned short→short)，原值照赋，内部的存储方式不变，但外部值却可能改变。

● 将一个非unsigned整型数据赋给长度相同的unsigned型变量时， 内部存储形式不变，但外部表示时总是无符号的。

```
/*例:赋值运算符举例 */
main()
{ unsigned a,b;
   int i,j;
   a=65535;
   i=-1;
   j=a;
   b=i;
   printf("(unsigned)%u→(int)%d\n",a,j);
   printf("(int)%d→(unsigned)%u\n",i,b);
}
```

运行结果为：

```
(unsigned)65535→(int)-1
(int)-1→(unsigned)65535
```

● 计算机中数据用补码表示，int型量最高位是符号位，为1时表示负值，为0时表示正值。如果一个无符号数的值小于32768则最高位为0，赋给 int型变量后、得到正值。如果无符号数大于等于32768，则最高位为1， 赋给整型变量后就得到一个负整数值。反之，当一个负整数赋给unsigned型变量时，得到的无符号值是一个大于32768的值。

● C语言这种赋值时的类型转换形式可能会使人感到不精密和不严格，因为不管表达式的值怎样，系统都自动将其转为赋值运算符左部变量的类型。

● 而转变后数据可能有所不同，在不加注意时就可能带来错误。 这确实是个缺点，也遭到许多人们批评。但不应忘记的是：c面言最初是为了替代汇编语言而设计的，所以类型变换比较随意。当然， 用强制类型转换是一个好习惯，这样，至少从程序上可以看出想干什么。

C语言规定,不同类型的数据需要转换成同一类型后才可进行计算,在整型、实型和字符型数据之间通过类型转换便可以进行混合运算(但不是所有类型之间都可以进行转换)

.当混合不同类型的变量进行计算时，便可能会发生类型转换

相同类型的数据在转换时有规则可循：

字符必须先转换为整数(C语言规定字符类型数据和整型数据之间可以通用)

short型转换为int型(同属于整型)

float型数据在运算时一律转换为双精度(double)型，以提高运算精度(同属于实型)

赋值时，一律是右部值转换为左部类型

[注]

当整型数据和双精度数据进行运算时，C先将整型数据转换成双精度型数据，再进行运算，结果为双精度类型数据

当字符型数据和实型数据进行运算时，C先将字符型数据转换成实型数据，然后进行计算，结果为实型数据

2.4　数据类型转换在C语言的表达式中，准许对不同类型的数值型数据进行某一操作或混合运算。当不同类型的数据进行操作时，应当首先将其转换成相同的数据类型，然后进行操作。数据类型转换有两种形式，即隐式类型转换和显示类型转换。

2.4.1　隐式类型转换所谓隐式类型转换就是在编译时由编译程序按照一定规则自动完成，而不需人为干预。因此，在表达式中如果有不同类型的数据参与同一运算时，编译器就在编译时自动按照规定的规则将其转换为相同的数据类型。

C语言规定的转换规则是由低级向高级转换。例如，如果一个操作符带有两个类型不同的操作数时，那么在操作之前行先将较低的类型转换为较高的类型，然后进行运算，运算结果是较高的类型。更确切地说，对于每一个算术运算符，则遵循图2-2所示的规则。

图2-2　数据类型转换规则之一

图2-2　数据类型转换规则之一

注意：在表达式中，所有的float类型都转换为double型以提高运算精度。

在赋值语句中，如果赋值号左右两端的类型不同，则将赋值号右边的值转换为赋值号左边的类型，其结果类型还是左边类型。

因为函数参数是表达式，因此，当参数传递给函数时，也发生类型转换。具体地说，char和short均转换为int；float转换为double。这就是为什么我们把函数参数说明为int和double，尽管调用函数时用char和float .

也可以将图2-2所示的规则用图2-3表示。图2-3中的水平箭头表示必定转换，纵向箭头表示两个操作对象类型不同时的转换方向。

图2-3　数据类型转换规则之二

下面举行说明类型转换的规则。例如执行：

x=100+'a'+1.5 * u+f/'b'-s * 3.1415926

其中，u为unsigned型，f为float型，s为short型，x为float型。式中右面表达式按如下步骤处理：

（1）首先将'a'、'b'和s换成int，将1.5和f转换为double型。

（2）计算100+'a'，因'a'已转换为int型，于是此运算结果为197。

（3）计算1.5*u，由于1.5已转换为double，u是unsigned型，于是首先u转换为double，然后进行运算，运算结果为double。

（4）计算197+1.5 * u，先将197转换为double（如197.00…00），其结果为double。

（5）计算f/ 'b'，f已转换为double，'b'已转换为int，于是先将'b'再转换为double，其结果为double。

（6）计算（197+1.5 * u）+f / 'b'，者均为double，于是结果也为double。

（7）计算s * 3.1415926，先将s由int转换为double，然后进行运算，其结果为double。

（8）最后与前面得的结果相减，结果为double。

（9）最后将表达式的结果转换为float并赋给x。

2.4.2　显式类型转换显示类型转换又叫强制类型转换，它不是按照前面所述的转换规则进行转换，而是直接将某数据转换成指定的类型。这可在很多情况下简化转换。例如，

```
int i;
…
i=i+9.801
```

按照隐式处理方式，在处理i=i+9.801时，首先i转换为double型，然后进行相加，结果为double型，再将double型转换为整型赋给i。

```
int i;
…
i=i+（int）9.801
```

这时直接将9.801转换成整型，然后与i相加，再把结果赋给i。这样可把二次转换简化为一次转换。

显示类型转换的方法是在被转换对象（或表达式）前加类型标识符，其格式是：

（类型标识符）表达式

例如，有如下程序段：

```
main（）
{
int a, b;
float c;
b=a+int（c）；
printf（"b=d% \ n", b）；
}
```

在上述程序的运行过程中，在执行语句b=a+int（c）时，将c的值临时强制性转化为int型，但变量c在系统中仍为实型变量，这一点很重要，不少初学者在这个问题上忽略了这个问题。

2.5　运算符和表达式

2.5.1　运算符和表达式概述

1．表达式一个表达式包含一个或多个操作，操作的对象称作运算元（或叫作操作数），而操作本身通过运算符体现的。例如a、a-b、c=9.801等都是一个表达式。

一个表达式完成一个或多个操作，最终得到一个结果，而结果的数据类型由参加运算的操作决定。

最简单的表达式是只含一个常量或变量的表达式，即只含一个操作数而不含运算符。

C语言中表达式的种类十分丰富，主要有如下一些：

n　　算术表达式：进行一般的计算。

n　　赋值表达式：进行赋值操作。

n　　关系表达式：进行比较判断。

n　　逻辑表达式：进行逻辑比较判断。

n　　条件表达式：进行条件满足与否的判断。

n　　逗号表达式：实际上是一种复杂运算，可以包含多个算术表达式。

2．C语言的操作符C语言的特点之一是具有丰富和使用灵活的运算符，概括起来它有如下的几类运算符：

n　　算术运算符。

n　　赋值运算符（包括符合赋值运算符）。

n　　关系运算符。

n　　逻辑运算符。

n　　条件运算符。

n　　逗号运算符。

n　　位运算符。

n　　指针运算符。

n　　求字节运算符（可以归并到函数的应用中去，它是通过函数sizeof（）来进行运算的）。

n　　强制类型转换运算符。

这些运算符如表2-4所示。

表2-4 C语言中的运算符

名称操作符自增，自减

+ +，－ －

逻辑与、或、非

＆＆，｜｜，！

续表2—4

名称操作符指针操作及引用

*，＆

加、减、乘、除、求模运算

+，—，*，/，%

关系操作符

<,<=,>,>=,==,!=

按位与、或、异或、求反

&, |, ^, ~

逗号表达式

,

类型转换

()

移位运算

< <, > >

条件运算

? :

求占用的字节数

sizeof

赋值

=, + =, — =, *=, / =, % =

先看程序:

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char**argv){
	unsigned int right = 1;
	char left = -1;
	if(left < right)printf("%d < %d\n", left, right);
	else if(left == right)printf("%d = %d\n", left, right);
	else printf("%d > %d\n", left, right);
	system("PAUSE");
	return 0;
}
```

运行结果是:-1 > 1

解释:按步骤进行.

1. 如果其中一个操作数为long double类型,则另一个操作数被转换为long double.

2. 否则,如果其中一个操作数为double, 则另一个操作数被转换为double.

3. 否则,如果其中一个操作数为float, 则另一个操作数也转换为float.

4. 否则,两个操作数进行 "整型升级":

	a. 如果其中一个操作数为unsigned long int, 则另一个操作数也被视为unsigned long int.

	b. 否则,如果其中一个操作数为long int,而另一个操作数类型是unsigned int, 并且long int能够表示unsigned int的所有值,则另一个操作数也被视为long int;如果long int不能表示unsigned int的所有值,则两个数都被视为unsigned long int.

	c. 否则, 如果其中一个操作数是long int,则另一个操作数也被视为long int.

d. 否则，如果其中一个操作数是unsigned int，则另一个操作数也被视为unsigned int.

e. 否则，两个操作数都被视为int.

例题1：

```
char a=0xb6;
short b=0xb600;
int c=0xb6000000;

if(a==0xb6)
puts("a");     //左边为有符号数，右边无符号数
if(b==0xb600)
puts("b");
if(c==0xb6000000)
puts("c");

//结果：gcc只打印出c(char默认为signed char)
```

==为一种运算符，则两边的值会进行类型转换为int型。见第一个图

gcc将三条语句解释为

```
if(0xffffffb6==0x000000b6)
puts("a");
if(0xfffffb600==0x0000b600)
puts("b");
if(0xb6000000==0xb6000000)
puts("c"
```

```
puts( c
);
```

例题2：

同样是有关“整形提升”

无符号数扩位补0

有符号书扩位补符号位，符号位为0则补0为1则补1

```
#include <stdio.h>
int main()
{
    char ca;
    unsigned char ucb;
    unsigned short usc;

    ca = 128;      //0x80
    ucb=128;

  //0x80

    usc = ca + ucb;    //0x0000=0xffffff80 + 0x00000080 然后截断取低四位（后同）
    printf("%d\n", usc);

    usc = ca + (short)ucb;  //0x0000=0xffffff80 +
```

	0x0000(0080) unsigned char -> short ->int
	    printf("%d\n", usc);
	
	    usc = (unsigned char)ca + ucb; //0x0100 = 0x000000(80) + 0x00000080
	    printf("%d\n", usc);
	
	    usc = ca + (char)ucb; //0xff00 = 0xffffff80 + 0xffffff(80)
	</stdio.h>

1	printf("%d\n", usc);
	
	getchar();
	
	return EXIT_SUCCESS;
	}

结果：0 0 256 65280

同时可以看到char<->unsigned char，short<->unsigned short 转换中，与类型等长的部分其实是相同的，不同的是截断去掉的那部分

char -128 127

unsigned char 0 256

0-127之间二者没有差别

例如：

```
unsigned char uca=128;
printf("%x\n",(char)uca);  //打印出来：ffffff80,unsigned char -> char若越界的话会整形提升。
```