

嵌入式C语言之- 数据运算的类型转换

讲师：叶大鹏

助力你成为优秀的电子工程师！



数据运算的底层机制

- 我们从前面的课程知道，单片机中存储和运算的数据都是1010这样的二进制数据，不管是正数、负数、小数，编译器都会将其转换为二进制（十六进制），然后在单片机中进行二进制运算：

```
volatile uint8_t a = -3;//将-3赋值给无符号的整形变量，编译器完成的工作是将-3的十六进制赋值给a  
volatile uint8_t b = 20;  
volatile uint8_t c = a + b;
```

- 对应底层的汇编代码：

```
main  
0x000004d0:    b50e    ..    PUSH    {r1-r3,lr}  
0x000004d2:    20fd    .    MOVS    r0,#0xfd  
0x000004d4:    9002    ..    STR     r0,[sp,#8]  
0x000004d6:    2014    .    MOVS    r0,#0x14  
0x000004d8:    9001    ..    STR     r0,[sp,#4]  
0x000004da:    f89d0008    ....    LDRB    r0,[sp,#8]  
0x000004de:    f89d1004    ....    LDRB    r1,[sp,#4]  
0x000004e2:    4408    .D    ADD     r0,r0,r1
```

应用案例

```
/*  
*第一步，获取日期。  
*日期是最后一个byte，也就是最后8位  
*/  
uint32_t date = 0x1413061D;  //00010100 00010011 00000110 00011101;  
uint8_t day = date;  //(计算结果是00011101，十进制表示是29，也就是日期是29)。  
  
/*  
*第二步，获取月份。  
*月份是倒数第2个byte，此时需要先将最后一个byte砍掉(也就是右移8位)  
*/  
date = date >> 8;  //(计算结果是00010100 00010011 00000110)  
uint8_t month = date;  //(计算结果是00000110，十进制表示是6，也就是月份是6月)。
```

类型转换

- 在编写程序时，可以把运算表达式的结果硬性转换为另一种数据类型值，它也是一个一元运算符，格式为：

(类型) 表达式

优先级

运算符（优先级从上往下）	运算符说明及应用场景	结合性
() [] -> .	括号（函数等），数组，结构体指针变量的成员访问，普通结构体变量的成员访问	由左向右
! ~ ++ -- + -	逻辑非，按位取反，自增1，自减1，正号，负号	由右向左
* & (类型) sizeof	间接，取地址，强制类型转换，求占用空间大小	
* / %	乘，除，取模	由左向右
+ -	加，减	由左向右
<< >>	左移，右移	由左向右

整体介绍

运算符（优先级从上往下）	运算符说明及应用场景	结合性
< <= >= >	是否小于, 是否小于等于, 是否大于等于, 是否大于	由左向右
== !=	是否等于, 是否不等于	由左向右
&	按位与	由左向右
^	按位异或	由左向右
	按位或	由左向右
&&	逻辑与	由左向右
	逻辑或	由左向右
?:	条件	由右向左
= += -= *= /= %= &= ^= = <<= >>=	各种赋值运算符	由右向左
,	逗号（顺序）	由左向右

应用案例

- 我们看下面的代码：

```
volatile uint8_t m = 5;  
printf("(float)m / 2 = %f\n", (float)m / 2);  
printf("(float)(m / 2) = %f\n", (float)(m / 2));  
printf("m / 2.0f = %f\n", m / 2.0f);
```

- 输出结果：

```
(float)m / 2 = 2.500000  
(float)(m / 2) = 2.000000  
m / 2.0f = 2.500000
```

应用案例

- 上面的代码引申出几个知识点:

```
volatile uint8_t m = 5;
```

```
printf("(float)m / 2 = %f\n", (float)m / 2);
```

1. 类型转换带来两种后果:

1) 数据扩充, 像这个例子中, **m**是整形类型扩充为浮点数类型;

2) 数据截断。

```
printf("m / 2.0f = %f\n", m / 2.0f);
```

2. `(float)m / 2`也被称为显示类型转换, 使用`float`明确的告诉编译器转换规则; 而像`m / 2.0f`这种表达式被称作 **自动类型转换: 隐式类型转换**, 对于这种情况, 有时是故意为之, 有时则是编写失误, 所以需要格外引起注意。

数据截断

- 通常出现在赋值的场景，表示将数据类型较大的变量赋给数据类型较小的变量：
- 规则：对于整形数据之间的转换，无论有无符号，丢弃高位数据


```
volatile int32_t a = -5;
```


```
volatile int32_t b = 5;
```


```
volatile int8_t c = a;
```


```
volatile int8_t d = b;
```

- 输出结果：

..  a

..  b

..  c

..  d

0xFFFFFFFFB

0x00000005

0xFB '?'

0x05

数据截断

- 通常出现在赋值的场景，表示将数据类型较大的变量赋给数据类型较小的变量：
- 规则：对于浮点数转换为整形数据，只保留整数位

```
int32_t a = (int32_t)0.1f; //即a=0
```

```
int32_t a = (int32_t)1.1f; //即a=1
```

```
int32_t a = 1.5f; //即a=1
```

```
int32_t a = (int32_t)-1.5f; //即a=-1
```

数据扩充

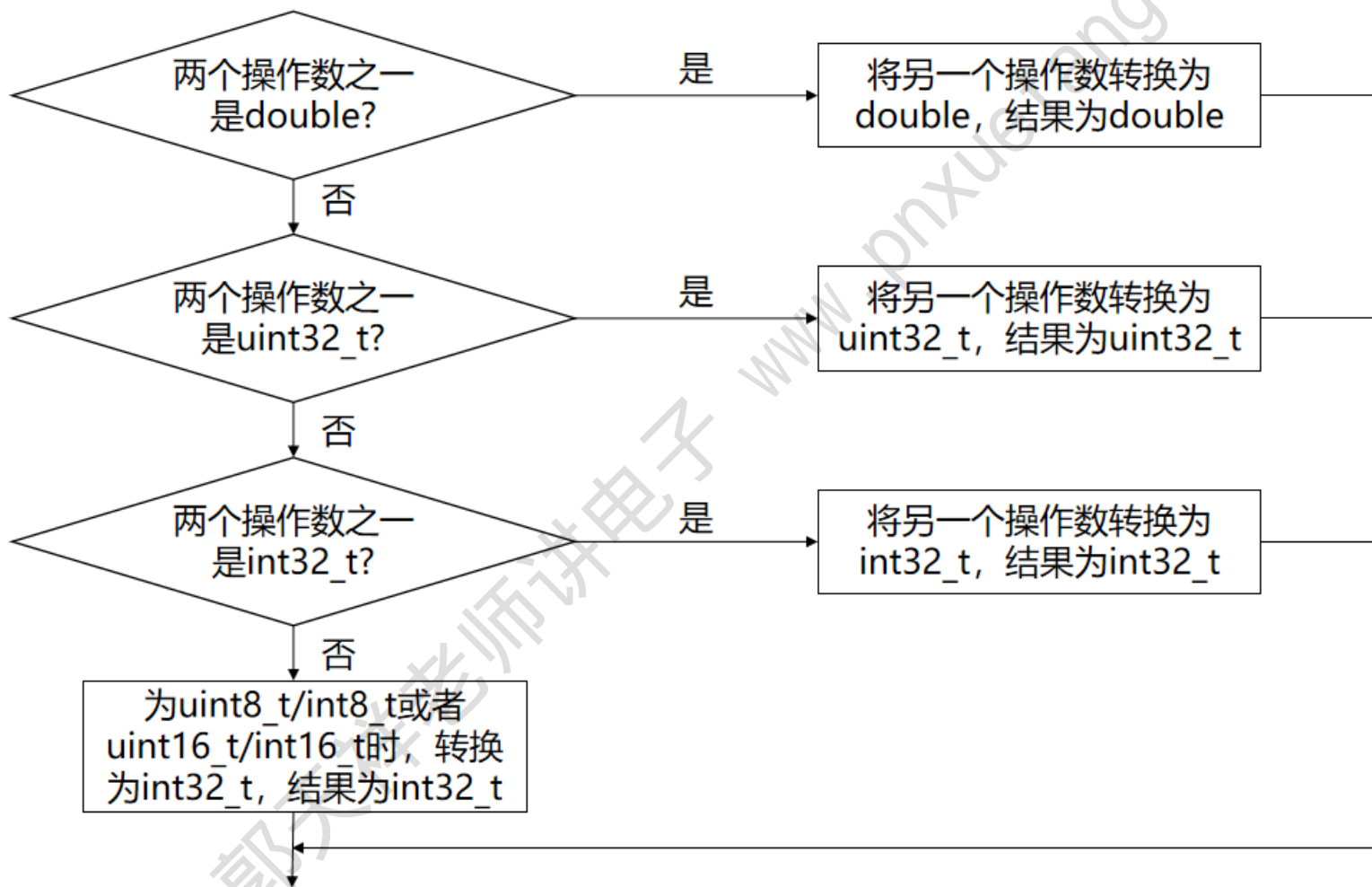
- 出现在赋值和运算表达式的场景，在表达式的运算符两侧当混有不同类型的常量及变量时，那么它们全都要先转换成同一类型，再进行运算；
- 对于隐式类型转换，有如下的扩充规则，即数值范围小的 向 数值范围大的进行转换

```
volatile uint8_t m = 5;
```

```
printf("m / 2.0f = %f\n", m / 2.0f); // 除法运算符左侧的m扩充为float类型，再进行运算
```

```
printf("(float)(m / 2) = %f\n", (float)(m / 2)); //并没有达到将结果扩充为float类型的目的，因为  
运算时先对m/2进行整数运算，结果为2，然后将2扩充为float类型
```

数据扩充规则（前提是运算符两侧操作数类型不同）



应用案例

```
float temp = 20.5f;  
//double temp = 20.5;  
temp = temp * 1.2;
```

Program Size: Code=1164 RO-data=992 RW-data=4 ZI-data=1028

```
38:      float temp = 20.5f;  
39:      //double temp = 20.5;  
0x000004D4 4C08      LDR      r4, [pc, #32] ; @0x000004F8  
40:      temp = temp * 1.2;  
41:  
0x000004D6 4620      MOV      r0, r4  
0x000004D8 F000F942 BL.W      __aeabi_f2d (0x00000760)  
0x000004DC 4605      MOV      r5, r0  
0x000004DE F04F3233 MOV      r2, #0x33333333  
0x000004E2 4B06      LDR      r3, [pc, #24] ; @0x000004FC  
0x000004E4 F000F83E BL.W      __aeabi_dmul (0x00000564)  
0x000004E8 4607      MOV      r7, r0  
0x000004EA F000F809 BL.W      __aeabi_d2f (0x00000500)  
0x000004EE 4604      MOV      r4, r0
```

应用案例

```
int8_t a = -5;
uint32_t b = 10;
if (a > b)
{
    printf("a > b\n");
}
else
{
    printf("a <= b\n");
}
```

- 分析:

a和b进行比较时, int8_t 类型的a会自动转换成uint32_t类型;

a = -5对应1111 1011, 将其扩充到4字节, 对应的就是

1111 1111 1111 1111 1111 1111 1111 1011

再将扩充后的1111 1111 1111 1111 1111 1111 1111 1011解析成无符号类型 (正数), 对应十进制远远大于10

应用案例

```
int8_t a = -5;
uint16_t b = 10;
if (a > b)
{
    printf("a > b\n");
}
else
{
    printf("a <= b\n");
}
```

- 分析:

当uint8_t类型和uint16_t类型进行转化时，都会转成有符号int32_t类型，所以结果是a < b

负整数在数据扩充时的表现形式

- 规则：负整数在向数值范围大的类型（不管是unsigned还是signed）扩充时，扩充位会填充1：

```
int8_t a = 0xFB; //十进制-5
```

```
int32_t b = a;
```

```
uint32_t c = a;
```

- 输出结果：

....	◆ a	0xFB '?'
....	◆ b	0xFFFFFFFF
....	◆ c	0xFFFFFFFF

...	◆ a	-5 ?
...	◆ b	-5
...	◆ c	4294967291

```
uint8_t a = 0xFB;
```

```
int32_t b = a;
```

```
uint32_t c = a;
```

- 输出结果：

....	◆ a	0xFB '?'
....	◆ b	0x000000FB
....	◆ c	0x000000FB

....	◆ a	251 ?
....	◆ b	251
....	◆ c	251

应用案例

```
for (int32_t i = -3; i < sizeof(int); i++)  
{  
    printf("%d ", i);  
}
```

- 分析:

sizeof(int) == 4的值是一个uint32_t类型

比较时, int32_t 会转成uint32_t类型

-3: 1111 1111 1111 1111 1111 1111 1111 1101

4: 0000 0000 0000 0000 0000 0000 0000 0100

以无符号来解读1111 1111 1111 1111 1111 1111 1111 1101
是很大的数 > 4, 所以不进入循环。

总结

- 按照经验：

1. 操作数全为有符号数，即使类型大小不一样，没有问题；
2. 操作数全为无符号数，即使类型大小不一样，没有问题；
3. 操作数混合了有符号数，无符号数，并且有正数有负数，很有可能出问题；

所以编写代码时，最好要**同符号类型（同为有符号或者无符号）进行运算**，可以避免正负数带来的错误问题，减少这些隐形难以发现的错误。

THANK YOU!