

**前言：**C语言的类型转换是很多初学者的难点，但也是语言的重点。在此，介绍一下C语言类型转换的知识。注意本文是以gcc编译器为基准。

**一、            变量**

**1、 基本类型变量**

说到基本类型，各种类型就会浮现在脑海中：

char（%c、%d、%u）、short（%hd）、int（%d）、long（%ld）、long long（%lld）及其对应无符号类型unsigned char（%c、%d、%u）、unsigned short（%hu、%ho、%hx）、unsigned int（%u,、%o、%x）、unsigned long（%lu、%lo、%lx）、unsigned long long（%llu、%llo、%llx），无符号类型打印出的值就是内存中存数的实际二进制值，还有float（%f）、double（%lf）、long double(%Lf)，其中各个括号中是他们的输出格式符。

从上面基本类型的介绍，可以知道有符号的基本类型有char、short、int、long、long long、float、double这些类型。

此处还得谈一下类型提升，那什么是类型提升呢？

再说类型提升之前，先介绍一下符号扩展和零扩展:

**1）    符号扩展**：**对于要扩展量为有符号数，扩展存储位数的方法。在新的高位字节使用当前最高有效位即符号位的值进行填充。**

**例1**：

        char a=0xff;//有符号值为-1，二进制为11111111，其中最高位为符号位

        shortb=a;//b的有符号值为-1，在内存中存储的值为1111111111111111

**例2**：

        char a=1;//有符号值为1，二进制为00000001，其中最高位为符号位

     shortb=a;//b的有符号值为1，在内存中存储的值为0000000000000001

**2）    零扩展：对于要扩展量无符号数，扩展存储位数的方法。在新的高位直接填0.**

**例1**：

unsigned char a=0xff;//二进制为11111111，所有值都是有效值

unsigned short b=a;//b经过零扩展后，内存中存储的值为0000000011111111

几个例子看下来，但还有两种情况没有涉及到，一是有符号短变量扩展成无符号长变量；二是无符号短变量扩展成有符号长变量。这两种情况编译器该怎么处理呢？

**其实这里注重的是要扩展的量是有符号量还是无符号量。若要扩展量为有符号量，不管扩展成有符号还是无符号，都遵循符号扩展；若要扩展量为无符号量，不管扩展成有符号还是无符号，都遵循零扩展。**

**例1：**

        char a =0xff;//a为-1，其为有符号量，二进制为11111111

        unsigned shortb=a;//此处a要进行符号扩展,b的二进制为11111111 11111111

例2：

        unsigned chara=0xff;//a为无符号量，二进制为11111111

        short b=a;//此处a要进行零扩展，b的二进制为00000000 11111111

**2）类型转换和扩展**：

**i.                    有符号数的转换规则**

|  |  |  |
| --- | --- | --- |
| 从 | 到 | 方法 |
| char | short | 符号位扩展 |
| char | long | 符号位扩展 |
| char | unsigned char | 最高位失去符号位意义，变为数据位 |
| char | unsigned short | 符号位扩展到short;然后从short转到 unsigned short |
| char | unsigned long | 符号位扩展到long; 然后从long 转到unsigned long |
| char | float | 符号位扩展到long; 然后从long 转到float |
| char | double | 符号位扩展到long; 然后从long 转到double |
| char | long double | 符号位扩展到long; 然后从long 转到long double |
| short | char | 保留低位字节 |
| short | long | 符号位扩展 |
| short | unsigned char | 保留低位字节 |
| short | unsigned short | 最高位失去符号位意义，变为数据位 |
| short | unsigned long | 符号位扩展到long; 然后从long转到unsigned double |
| short | float | 符号位扩展到long; 然后从long 转到float |
| short | double | 符号位扩展到long; 然后从long 转到double |
| short | long double | 符号位扩展到long; 然后从long 转到double |
| long | char | 保留低位字节 |
| long | short | 保留低位字节 |
| long | unsigned char | 保留低位字节 |
| long | unsigned short | 保留低位字节 |
| long | unsigned long | 最高位失去符号位意义，变为数据位 |
| long | Float | 使用单精度浮点数表示。可能丢失精度。 |
| long | double | 使用双精度浮点数表示。可能丢失精度。 |
| long | long double | 使用双精度浮点数表示。可能丢失精度。 |

**ii.                  无符号数的转换规则**

|  |  |  |
| --- | --- | --- |
| 从 | 到 | 方法 |
| unsigned char | char | 最高位作为符号位 |
| unsigned char | short | 0扩展 |
| unsigned char | long | 0扩展 |
| unsigned char | unsigned short | 0扩展 |
| unsigned char | unsigned long | 0扩展 |
| unsigned char | float | 转换到long; 再从 long 转换到float |
| unsigned char | double | 转换到long; 再从 long 转换到double |
| unsigned char | long double | 转换到long; 再从 long 转换到double |
| unsigned short | char | 保留低位字节 |
| unsigned short | short | 最高位作为符号位 |
| unsigned short | long | 0扩展 |
| unsigned short | unsigned char | 保留低位字节 |
| unsigned short | unsigned long | 0扩展 |
| unsigned short | float | 转换到long; 再从 long 转换到float |
| unsigned short | double | 转换到long; 再从 long 转换到double |
| unsigned short | long double | 转换到long; 再从 long 转换到double |
| unsigned long | char | 保留低位字节 |
| unsigned long | short | 保留低位字节 |
| unsigned long | long | 最高位作为符号位 |
| unsigned long | unsigned char | 保留低位字节 |
| unsigned long | unsigned short | 保留低位字节 |
| unsigned long | float | 转换到long; 再从 long 转换到float |
| unsigned long | double | 直接转换成double |
| unsigned long | long double | 转换到long; 再从 long 转换到double |

用文字总结一下上面提到的知识：

**1）、短数据类型扩展为长数据类型**

**i.要扩展的短数据类型为有符号数的**

  进行符号扩展，即短数据类型的符号位填充到长数据类型的高字节位（即比短数据类型多出的那一部分），保证扩展后的数值大小不变

1：char x=10001001b;   short y=x;  则y的值应为11111111 10001001b；

  2：charx=00001001b;   short y=x;  则y的值应为00000000 00001001b；

**ii. 要扩展的短数据类型为无符号数的**

  进行零扩展，即用零来填充长数据类型的高字节位

 1：unsignedchar x=10001001b;   short y=x;  则y的值应为00000000 10001001b；

 2：unsignedchar x=00001001b;   short y=x;  则y的值应为00000000 00001001b；

**2)、长数据类型缩减为短数据类型**

如果长数据类型的高字节全为1或全为0，则会直接截取低字节赋给短数据类型；如果长数据类型的高字节不全为1或不全为0，则转会就会发生错误。

**3)、同一长度的数据类型中有符号数与无符号数的相互转化**

直接将内存中的数据赋给要转化的类型，数值大小则会发生变化。另短类型扩展为长类型时，但短类型与长类型分属有符号数与无符号数时，则先按规则一进行类型的扩展，再按本规则直接将内存中的数值原封不动的赋给对方

**2、 数组变量及其成员**

由于数组名是常量，不能用数组名赋给另一个数组名，如下的情况是不被允许的：

                   int a[10]={1,2,3,4,5,6,7,8,9,10},b[10]={11,12,13,14,15,16,17,18,19,20};

                   a=b;//这是不被允许的，因为a,b都是常量，常量不允许被改变

注意在数组定义声明时，编译器就会给它分配了空间。数组名为该空间的首地址。

虽然数组名不能直接赋值，但是数组成员可以直接赋值和计算，规则遵循前面提到的“基本类型变量转换”，例如：

                   int a[10]={1,2,3,4,5,6,7,8,9,10}；

                   a[3] = 16;

也可以这样：

                   \*(a+3)=16;//这也是后面要说的

**3、 结构、联合和枚举变量及其成员变量**

同类型的结构可以相互赋值，例如

typedef struct test{

                   char\*name;

                   unsignedshort age;

} Struct\_test;

Struct\_test a,b={“zhangsan”,20};

a = b;//a和b中的数据是一样的，但在不同的存储空间

a.name=”lisi”;//结构中的成员赋值

                            a.age = 25;

                            b.age = a.age;

                            在此，特别介绍一下，通过成员变量来获取到结构的首地址的方法：

                            先定义一个宏来确定该成员在结构中的偏移量

                            #define offsetof(type,name) (size\_t)((char \*)&((type \*)0)->name – (char \*)(type \*)0)

                            size\_t offset =offsetof(Struct\_test, age);

                            Struct\_test test ;

                            If((char \*)&test==(char\*)&test.age-offset){

                                     printf(“structureaddr!\n”);

}

下面在介绍一下，通过共用体来确定机器是大端序还是小端序的例子：

#include <stdio.h>

union test{

         chara[4];

         intb;

};

int main(int argc, char \*\*argv){

         uniontest tst;

         tst.a[0]=0x04;

         tst.a[1]= 0x03;

         tst.a[2]= 0x02;

tst.a[3] = 0x01;

printf(“0x%x\n”, tst.b);

return 0;

}

若打印出0x4030201则为大端序，若为0x1020304则为小端序

要将一个整型赋给枚举，要强制类型转换，否则会报错，如：

      enum week{

              SUN,

              MON,

              TUE,

              WED,

              THU,

              FRI,

              SAT

      };

       unsigned short k;

    printf("inputa value from 1 to 7:\n");

    scanf("%hu",&k);

    DAY = (enum week)k;

**变量与指针**

此时引用两个操作符：“->”和“&”，其中“->”是成员选择符，使用方法：对象指针->成员名；“&”取址运算符

**1、 基本类型、结构和联合变量与指针**

基本类型变量可以转化为指针变量，指针变量也可以转化为基本类型变量，见下例：

int tst = 10;

int \*ptst = malloc(sizeof(int));

typedef structtest{

                   char \*name;

                   unsigned short age;

} Struct\_test;

Struct\_test Stst ={“zhangsan”, 20};

   Struct\_test \*pStst = malloc(sizeof(structtest));

   pStst = &Stst;

基本类型变量转化为指针变量：

**i.** tst是int型，采用&tst转化为int \*型；

**ii.**Stst是Struct\_test结构类型，采用&Stst转化为Struct\_test\*型，此时就可以用Stst.name、Stst.age、(&Stst)->name、(&Stst)->age来访问成员变量；

指针变量转化为基本类型变量：

**i.**ptst是int \*，采用\*ptst转化为int型；

**ii.**pStst是Struct\_test\*型，采用\*pStst转化为Struct\_test型，此时就可以用(\*pStst).name、(\*pStst).age、pStst->name和pStst->age来访问成员变量。

**2、 数组变量及其成员与指针**

数组名即为常量指针，指向分配给数组的空间的首地址，数组名可以像指针一样使用，例如：

int tst[5] = {1,2,3,4,5};

可以用以下两种方法来访问成员：

通过数组下标：tst[3]的值为4；

通过指针：\*(tst+3)的值为4，等于tst[3]；

**二、            指针**

指针是初学者的难点，也是C语言的精华所在，还有个人感觉指针类型的强制类型转换更加实用和常见，像转换成char \*类型，void \*类型等

**1、 对象指针**

**a)        基本类型指针的转换**

可参考“**基本类型、结构和联合变量与指针**”这一节。

**i.                    指针的大小和所指对象的大小**

指针的大小是指指针变量在内存中所占存储空间的大小，除了函数指针的大小外，其他指针变量的大小在同一台机器的前提下都一样大，例如：

sizeof(char \*)=sizeof(short\*)=sizeof(int \*)=sizeof(long \*)=sizeof(long long \*)=sizeof(float \*)=sizeof(double\*)=sizeof(long double \*)=sizeof(unsigned char \*)=8或4，其中“=”不是赋值运算符

指针所指对象的大小，是指指针指向对象的内存空间的大小，如int \*型所指对象为int型，大小为4byte，long \*型所指对象为long型，大小为8byte(我的操作系统是64位)，若指针指向的是结构，则指向对象的大小就是结构的大小。

**ii.                  指针变量的运算**

指针运算与指针指向对象的大小密切相关，指针的加减是根据其指向类型的大小进行运算的。

例如：

     int a[5] = {1,2,3,4,5};//数组中的元素都为整型，每个元素占4个字节

     假设a的地址指向地址0，其中存放的值为1；a+1指向地址4，其中存放的值为2；a+2指向地址8，其中存放的值为3；这样一次类推

**b)       void \*的转换**

C语言没有通用的指针类型，指向其他任意类型的指针可以被转换为void \*类型而不会丢失信息。如果将转换的结果在转换为初始指针类型，那么初始指针被恢复。也即，当void \*和其他类型相互赋值的时候，如果需要，它可以自动转换成其他类型。

**c)        void \*\*的转换**

问：能否像下边这样用void \*\*指针作为参数，使函数模拟按引用传递参数吗？

        void f(void \*\*);

        double \*dp;

        f((void \*\*)&dp);

答：不可移植，这种代码可能有效，而且有时鼓励这样用，但前提是所有指针的内部表示都是一样的，也即内存中数据存储格式一样。

        C语言中没有通用指针类型，void \*之所以可以用作通用指针，是因为当它和其他类型相互赋值的时候，如果需要，它可以自动转换成其他类型。但是，void \*\*就不会自动转换了，原因是，当你使用void \*\*指针的时候，例如用\*操作符访问void \*\*所指向的void \*值得时候，编译器无法知道void \*值是否从其他类型的指针转换而来，从而，编译器只能认为它仅仅是个void \*指针，所以程序就无法正确访问到想要的结果。

        换言之，你使用的任何void \*\*值必须是某个位置的void\*值得地址，（void \*\*）&dp这样的类型转化虽然能编译通过，但执行结果可能不是我们想要的。如果void \*\*指针指向的不是void \*类型，并且这个类型的大小和内存表示和void \*也不同，则程序就无法正确访问到此类型。

        如何让上面代码正确工作呢，我们要使用void\*类型做一下中转：

         void f(void \*\*);

        double \*dp;

        void \*vp = dp;//可以这样直接复制，编译器会自动帮我们类型转换

        f(&vp);

        dp = vp;//同上

据我所知，上面的情形不常见，但是会有，下面再看一个常见的，容易犯错误的例子：

        void incme(double \*p){

                  \*p +=1;

                  printf(“value =%lf\n”, \*p);

}

intmain(int argc, char \*\*argv){

int i=1;

incme((double\*)&i);

return 0;

        }

此程序的结构一定不是我们想要的结果。这是因为变量i所在内存空间中存放的是有符号整形的格式，这是板上钉钉的事。此时，incme函数中要用的类型是double的，我们知道double在内存中的格式和int的不同，所以意想不到的结果就产生了。如何修改呢？和前提到过一样用一个中转：

        void incme(double \*p){

                  \*p +=1;

                  printf(“value =%lf\n”, \*p);

}

intmain(int argc, char \*\*argv){

int i=1;

doubled=i;

incme(&d);

i=d;

return 0;

        }

**2、 函数指针的转换**

函数指针很大，比任何数据对象指针都大，所以不能将函数指针转换为void \*或者其他对象类型的指针。但是所有的函数指针类型都可以相互转换，只要在调用之前转回了正确类型既可。