

这个结构总共需要 24 个字节：type 是 4 个字节，info.internal.left 和 info.internal.right 各要 8 个字节，或者是 info.data 要 16 个字节。我们后面很快会谈到，在字段 type 和联合的元素之间需要 4 个字节的填充，所以整个结构大小为 $4+4+16=24$ 。在这种情况下，相对于给代码造成的麻烦，使用联合带来的节省是很小的。对于有较多字段的数据结构，这样的节省会更加吸引人。

联合还可以用来访问不同数据类型的位模式。例如，假设我们使用简单的强制类型转换将一个 double 类型的值 d 转换为 unsigned long 类型的值 u：

```
unsigned long u = (unsigned long) d;
```

值 u 会是 d 的整数表示。除了 d 的值为 0.0 的情况以外，u 的位表示会与 d 的很不一样。再看下面这段代码，从一个 double 产生一个 unsigned long 类型的值：

```
unsigned long double2bits(double d) {
    union {
        double d;
        unsigned long u;
    } temp;
    temp.d = d;
    return temp.u;
};
```


在这段代码中，我们以一种数据类型来存储联合中的参数，又以另一种数据类型来访问它。结果会是 u 具有和 d 一样的位表示，包括符号位字段、指数和尾数，如 3.11 节中描述的那样。u 的数值与 d 的数值没有任何关系，除了 d 等于 0.0 的情况。

当用联合来将各种不同大小的数据类型结合到一起时，字节顺序问题就变得很重要了。例如，假设我们写了一个过程，它以两个 4 字节的 unsigned 的位模式，创建一个 8 字节的 double：

```
double uu2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

在 x86-64 这样的小端法机器上，参数 word0 是 d 的低位 4 个字节，而 word1 是高位 4 个字节。在大端法机器上，这两个参数的角色刚好相反。

 **练习题 3.43** 假设给你个任务，检查一下 C 编译器为结构和联合的访问产生正确的代码。你写了下面的结构声明：

```
typedef union {
    struct {
        long    u;
        short   v;
        char     w;
```