在一台 x86-64 Linux 机器上编译时，字段的偏移量、数据类型 S3 和 U3 的完整大小如下：

类型	c	i	v	大小
S3	0	4	16	24
U3	0	0	0	8

(稍后会解释 S3 中 i 的偏移量为什么是 4 而不是 1，以及为什么 v 的偏移量是 16 而不是 9 或 12。)对于类型 union U3 * 的指针 p，p-> c、p-> i[0]和 p-> v 引用的都是数据结构的起始位置。还可以观察到，一个联合的总的大小等于它最大字段的大小。

在一些下上文中，联合十分有用。但是，它也能引起一些讨厌的错误，因为它们绕过了 C 语言类型系统提供的安全措施。一种应用情况是，我们事先知道对一个数据结构中的两个不同字段的使用是互斥的，那么将这两个字段声明为联合的一部分，而不是结构的一部分，会减小分配空间的总量。

例如，假设我们想实现一个二叉树的数据结构，每个叶子节点都有两个 double 类型的数据值，而每个内部节点都有指向两个孩子节点的指针，但是没有数据。如果声明如下：

```
struct node_s {
    struct node_s *left;
    struct node_s *right;
    double data[2];
};
```

那么每个节点需要 32 个字节，每种类型的节点都要浪费一半的字节。相反，如果我们如下声明一个节点：

```
union node_u {
    struct {
        union node_u *left;
        union node_u *right;
    } internal;
    double data[2];
};
```

那么，每个节点就只需要 16 个字节。如果 n 是一个指针，指向 union node_u * 类型的节点，我们用 n-> data[0]和 n-> data[1]来引用叶子节点的数据，而用 n-> internal.left 和 n-> internal.right 来引用内部节点的孩子。

不过，如果这样编码，就没有办法来确定一个给定的节点到底是叶子节点，还是内部节点。通常的方法是引入一个枚举类型，定义这个联合中可能的不同选择，然后再创建一个结构，包含一个标签字段和这个联合：

```
typedef enum { N_LEAF, N_INTERNAL } nodetype_t;

struct node_t {
    nodetype_t type;
    union {
        struct {
            struct node_t *left;
            struct node_t *right;
        } internal;
        double data[2];
    } info;
};
```