```
struct rect r = { 0, 0, 10, 20, 0xFF00FF };
```

将指向结构的指针从一个地方传递到另一个地方，而不是复制它们，这是很常见的。例如，下面的函数计算长方形的面积，这里，传递给函数的就是一个指向长方形 struct 的指针：

```
long area(struct rect *rp) {
    return (*rp).width * (*rp).height;
}
```

表达式 (*rp).width 间接引用了这个指针，并且选取所得结构的 width 字段。这里必须要用括号，因为编译器会将表达式 *rp.width 解释为 * (rp.width)，而这是非法的。间接引用和字段选取结合起来使用非常常见，以至于 C 语言提供了一种替代的表示法 -> 。即 rp-> width 等价于表达式 (*rp).width。例如，我们可以写一个函数，它将一个长方形顺时针旋转 90 度：

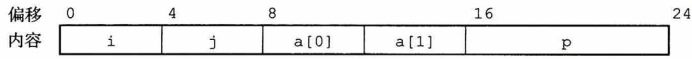
```
void rotate_left(struct rect *rp) {
    /* Exchange width and height */
    long t = rp->height;
    rp->height = rp->width;
    rp->width  = t;
    /* Shift to new lower-left corner */
    rp->llx   -= t;
}
```

C++ 和 Java 的对象比 C 语言中的结构要复杂精细得多，因为它们将一组可以被调用来执行计算的方法与一个对象联系起来。在 C 语言中，我们可以简单地把这些方法写成普通函数，就像上面所示的函数 area 和 rotate_left。

让我们来看看这样一个例子，考虑下面这样的结构声明：

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

这个结构包括 4 个字段：两个 4 字节 int、一个由两个类型为 int 的元素组成的数组和一个 8 字节整型指针，总共是 24 个字节：



可以观察到，数组 a 是嵌入到这个结构中的。上图中顶部的数字给出的是各个字段相对于结构开始处的字节偏移。

为了访问结构的字段，编译器产生的代码要将结构的地址加上适当的偏移。例如，假设 struct rec* 类型的变量 r 放在寄存器 %rdi 中。那么下面的代码将元素 r->i 复制到元素 r->j：

```
        Registers: r in %rdi
1       movl    (%rdi), %eax        Get r->i
2       movl    %eax, 4(%rdi)       Store in r->j
```