操控位的第2种方法是位字段（bit field）。位字段是一个signed int或unsigned int类型变量中的一组相邻的位（C99和C11新增了_Bool类型的位字段）。位字段通过一个结构声明来建立，该结构声明为每个字段提供标签，并确定该字段的宽度。例如，下面的声明建立了一个4个1位的字段：

```
struct {
unsigned int autfd : 1;
unsigned int bldfc : 1;
unsigned int undln : 1;
unsigned int itals : 1;
} prnt;
```

根据该声明，prnt包含4个1位的字段。现在，可以通过普通的结构成员运算符（.）单独给这些字段赋值：

```
prnt.itals = 0;
```

由于每个字段恰好为1位，所以只能为其赋值1或0。变量prnt被储存在int大小的内存单元中，但是在本例中只使用了其中的4位。

带有位字段的结构提供一种记录设置的方便途径。许多设置（如，字体的粗体或斜体）就是简单的二选一。例如，开或关、真或假。如果只需要使用 1 位，就不需要使用整个变量。内含位字段的结构允许在一个存储单元中储存多个设置。

有时，某些设置也有多个选择，因此需要多位来表示。这没问题，字段

不限制 1 位大小。可以使用如下的代码：

```
struct {
unsigned int code1 : 2;
unsigned int code2 : 2;
unsigned int code3 : 8;
} prcode;
```

以上代码创建了两个2位的字段和一个8位的字段。可以这样赋值：

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

但是，要确保所赋的值不超出字段可容纳的范围。

如果声明的总位数超过了一个unsigned int类型的大小会怎样？会用到下一个unsigned int类型的存储位置。一个字段不允许跨越两个unsigned int之间的边界。编译器会自动移动跨界的字段，保持unsigned int的边界对齐。一旦发生这种情况，第1个unsigned int中会留下一个未命名的“洞”。

可以用未命名的字段宽度“填充”未命名的“洞”。使用一个宽度为0的未命名字段迫使下一个字段与下一个整数对齐：

```
struct {
unsigned int field1     : 1 ;
unsigned int            : 2 ;
```

```
unsigned int field2       : 1 ;

unsigned int              : 0 ;

unsigned int field3       : 1 ;

} stuff;
```

这里，在stuff.field1和stuff.field2之间，有一个2位的空隙；stuff.field3将储存在下一个unsigned int中。

字段储存在一个int中的顺序取决于机器。在有些机器上，存储的顺序是从左往右，而在另一些机器上，是从右往左。另外，不同的机器中两个字段边界的位置也有区别。由于这些原因，位字段通常都不容易移植。尽管如此，有些情况却要用到这种不可移植的特性。例如，以特定硬件设备所用的形式储存数据。

通常，把位字段作为一种更紧凑储存数据的方式。例如，假设要在屏幕上表示一个方框的属性。为简化问题，我们假设方框具有如下属性：

方框是透明的或不透明的；

方框的填充色选自以下调色板：黑色、红色、绿色、黄色、蓝色、紫色、青色或白色；

边框可见或隐藏；

边框颜色与填充色使用相同的调色板；

边框可以使用实线、点线或虚线样式。

可以使用单独的变量或全长（full-sized）结构成员来表示每个属性，但是这样做有些浪费位。例如，只需1位即可表示方框是透明还是不透明；只需1位即可表示边框是显示还是隐藏。8种颜色可以用3位单元的8个可能的值来表示，而3种边框样式也只需2位单元即可表示。总共10位就足够表示方框的5个属性设置。

一种方案是：一个字节储存方框内部（透明和填充色）的属性，一个字节储存方框边框的属性，每个字节中的空隙用未命名字段填充。struct box_props声明如下：

```
struct box_props {
bool opaque                    : 1 ;
unsigned int fill_color        : 3 ;
unsigned int                   : 4 ;
bool show_border               : 1 ;
unsigned int border_color      : 3 ;
unsigned int border_style      : 2 ;
unsigned int                   : 2 ;
};
```

加上未命名的字段，该结构共占用 16 位。如果不使用填充，该结构占用 10 位。但是要记住，C 以unsigned int作为位字段结构的基本布局单元。因此，即使一个结构唯一的成员是1位字段，该结构的大小也是一个unsigned int类型的大小，unsigned int在我们的系统中是32位。另外，以上代码假设C99新增的_Bool类型可用，在stdbool.h中，bool是_Bool的别名。

对于opaque成员，1表示方框不透明，0表示透明。show_border成员也用类似的方法。对于颜色，可以用简单的RGB（即red-green-blue的缩写）表示。这些颜色都是三原色的混合。显示器通过混合红、绿、蓝像素来产生不

同的颜色。在早期的计算机色彩中，每个像素都可以打开或关闭，所以可以使用用 1 位来表示三原色中每个二进制颜色的亮度。常用的顺序是，左侧位表示蓝色亮度、中间位表示绿色亮度、右侧位表示红色亮度。表15.3列出了这8种可能的组合。fill_color成员和border_color成员可以使用这些组合。最后，border_style成员可以使用0、1、2来表示实线、点线和虚线样式。

表**15.3** 简单的颜色表示

位组合	十进制	颜色
000	0	黑色
001	1	红色
010	2	绿色
011	3	黄色
100	4	蓝色
101	5	紫色
110	6	青色
111	7	白色

程序清单15.3中的程序使用box_props结构，该程序用#define创建供结构成员使用的符号常量。注意，只打开一位即可表示三原色之一。其他颜色用三原色的组合来表示。例如，紫色由打开的蓝色位和红色位组成，所以，紫色可表示为BLUE|RED。

程序清单**15.3 fields.c**程序

```
/* fields.c -- 定义并使用字段 */

#include <stdio.h>

#include <stdbool.h>    // C99定义了bool、true、false

/* 线的样式 */

#define SOLID    0

#define DOTTED   1

#define DASHED   2
```

```
/* 三原色 */
#define BLUE    4
#define GREEN    2
#define RED      1
/* 混合色 */
#define BLACK    0
#define YELLOW   (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE    (RED | GREEN | BLUE)
const char * colors[8] = { "black", "red", "green", "yellow",
"blue", "magenta", "cyan", "white" };
struct box_props {
bool opaque : 1;          // 或者 unsigned int （C99以前）
unsigned int fill_color : 3;
unsigned int : 4;
bool show_border : 1;   // 或者 unsigned int （C99以前）
unsigned int border_color : 3;
unsigned int border_style : 2;
```

```
unsigned int : 2;
};
void show_settings(const struct box_props * pb);
int main(void)
{
/* 创建并初始化 box_props 结构 */
struct box_props box = { true, YELLOW, true, GREEN, DASHED };
printf("Original box settings:\n");
show_settings(&box);
box.opaque = false;
box.fill_color = WHITE;
box.border_color = MAGENTA;
 box.border_style = SOLID;
 printf("\nModified box settings:\n");
 show_settings(&box);
 return 0;
 }
 void show_settings(const struct box_props * pb)
 {
```

```
 printf("Box is %s.\n",
 pb->opaque == true ? "opaque" : "transparent");
 printf("The fill color is %s.\n", colors[pb->fill_color]);
 printf("Border %s.\n",
 pb->show_border == true ? "shown" : "not shown");
 printf("The border color is %s.\n", colors[pb->border_color]);
 printf("The border style is ");
 switch (pb->border_style)

{
case SOLID:    printf("solid.\n"); break;
case DOTTED: printf("dotted.\n"); break;
case DASHED: printf("dashed.\n"); break;
default:       printf("unknown type.\n");
}
}
```

下面是该程序的输出：

```
Original box settings:
Box is opaque.
The fill color is yellow.
```

Border shown.

The border color is green.

The border style is dashed.

Modified box settings:

Box is transparent.

The fill color is white.

Border shown.

The border color is magenta.

The border style is solid.

该程序要注意几个要点。首先，初始化位字段结构与初始化普通结构的语法相同：

struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};

类似地，也可以给位字段成员赋值：

box.fill_color = WHITE;

另外，switch语句中也可以使用位字段成员，甚至还可以把位字段成员用作数组的下标：

printf("The fill color is %s.\n", colors[pb->fill_color]);

注意，根据 colors 数组的定义，每个索引对应一个表示颜色的字符串，而每种颜色都把索引值作为该颜色的数值。例如，索引1对应字符串"red"，枚举常量red的值是1。