Bitsel (bitwise) Operatörler

C, bir struct veya union yapısının işaretsiz veya işaretli bir üyesinin depolandığı bit sayısını belirlemenizi sağlar. Buna bit alanı denir. Bit alanları, verileri gereken minimum bit sayısında depolayarak daha iyi bellek kullanımına olanak tanır. Bit alanı üyeleri int veya unsigned int olarak bildirilmelidir.

Aşağıdaki struct yapısı tanımını göz önünde bulunduralım:

```
struct bitCard {
        unsigned int face : 4;
        unsigned int suit : 2;
        unsigned int color : 1;
};
```

Burada, 52 kartlık bir desteden bir kartı temsil etmek için kullanılan üç işaretsiz int bit alanı (face, suit ve color) içerir. Bir bit alanı, alanın genişliğini temsil eden bir tamsayı (yani, üyenin depolandığı bit sayısı-kaç tane bit ile ifade edilebildiği) ile bildirilir.

Önceki struct yapısı tanımı, face'in 4 bit ile, suit'in renginin 2 bit ile ve color'ın 1 bit ile saklandığını gösterir. Bit sayısı, her bir yapı elemanı için istenen değer aralığına bağlıdır. face:, 0 (As) ile 12 (King) arasındaki değerleri saklar (4 bit, 0–15 aralığındaki değerleri saklayabilir). suit: 0 ile 3 arasındaki değerleri saklar (0 = Kupa, 1 = Karo, 2 = Sinek, 3 = Maça) (2 bit, 0–3 aralığında değerleri saklayabilir). Son olarak, color 0 (Kırmızı) veya 1 (Siyah) (1 bit, 0 veya 1 saklayabilir).

Şekil 10.16 (çıktısı Şekil 10.17'de gösterilmiştir), 20. satırda 52 struct bit-Card yapısı içeren dizi destesi oluşturur. Function fillDeck (31-39. satırlar) card dizisine 52 kartı ekler ve function deal (43-55. satırlar) 52 kart'ı yazdırır.

```c
// Fig. 10.16: fig10_16.c
// Representing cards with bit fields in a struct
#include <stdio.h>
#define CARDS 52

// bitCard structure definition with bit fields
struct bitCard {
   unsigned int face : 4; // 4 bits; 0-15
   unsigned int suit : 2; // 2 bits; 0-3
   unsigned int color : 1; // 1 bit; 0-1
};

typedef struct bitCard Card; // new type name for struct bitCard

void fillDeck(Card * const wDeck); // prototype
void deal(const Card * const wDeck); // prototype

int main(void)
{
   Card deck[CARDS]; // create array of Cards

   fillDeck(deck);

   puts("Card values 0-12 correspond to Ace through King");
   puts("Suit values 0-3 correspond Hearts, Diamonds, Clubs and Spades");
   puts("Color values 0-1 correspond to red and black\n");
   deal(deck);
}

// initialize Cards
void fillDeck(Card * const wDeck)
{
   // loop through wDeck
   for (size_t i = 0; i < CARDS; ++i) {
      wDeck[i].face = i % (CARDS / 4);
      wDeck[i].suit = i / (CARDS / 4);
      wDeck[i].color = i / (CARDS / 2);
   }
}

// output cards in two-column format; cards 0-25 indexed with
// k1 (column 1); cards 26-51 indexed with k2 (column 2)
void deal(const Card * const wDeck)
{
   printf("%-6s%-6s%-15s%-6s%-6s%s\n", "Card", "Suit", "Color",
      "Card", "Suit", "Color");

   // loop through wDeck
   for (size_t k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; ++k1, ++k2) {
      printf("Card:%3d  Suit:%2d  Color:%2d   ",
         wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
      printf("Card:%3d  Suit:%2d  Color:%2d\n",
         wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
   }
}
```

```
Card values 0-12 correspond to Ace through King
Suit values 0-3 correspond Hearts, Diamonds, Clubs and Spades
Color values 0-1 correspond to red and black

Card  Suit  Color          Card  Suit  Color
0     0     0              0     2     1
1     0     0              1     2     1
2     0     0              2     2     1
3     0     0              3     2     1
4     0     0              4     2     1
5     0     0              5     2     1
6     0     0              6     2     1
7     0     0              7     2     1
8     0     0              8     2     1
9     0     0              9     2     1
10    0     0              10    2     1
11    0     0              11    2     1
12    0     0              12    2     1
0     1     0              0     3     1
1     1     0              1     3     1
2     1     0              2     3     1
3     1     0              3     3     1
4     1     0              4     3     1
5     1     0              5     3     1
6     1     0              6     3     1
7     1     0              7     3     1
8     1     0              8     3     1
9     1     0              9     3     1
10    1     0              10    3     1
11    1     0              11    3     1
12    1     0              12    3     1
```

Şekil 10.17 | Şekil 10.16'daki programın çıktısı.

Struct yapılarında dolgu olarak kullanılacak isimsiz bir bit alanı belirtmek mümkündür. Örneğin, yapı tanımı şöyle olabilir;

```
struct example {
        unsigned int a : 13;
        unsigned int : 19;
        unsigned int b : 4;
};
```

Bu yapı dolgu olarak adsız 19 bitlik bir alan kullanır—bu 19 bitte hiçbir şey depolanamaz. Üye b (4 baytlık bilgisayarımızda) başka bir depolama biriminde saklanmaktadır.

Bir sonraki bit alanını yeni bir depolama birimi sınırında hizalamak için sıfır genişliğe sahip adsız bir bit alanı kullanılır. Örneğin yapı tanımı;

```
struct example {
        unsigned int a : 13;
        unsigned int : 0;
        unsigned int b: 4;
};
```

a'nın depolandığı depolama biriminin kalan bitlerini (ne kadar çok varsa) atlamak ve b'yi bir sonraki depolama birimi sınırında hizalamak için adsız bir 0-bitlik alan kullanır.

Enum anahtar kelimesi tarafından tanıtılan bir numaralandırma, tanımlayıcılar tarafından temsil edilen bir tamsayı numaralandırma sabitleri kümesidir. Bir enum'daki değerler, aksi belirtilmedikçe 0 ile başlar ve 1 ile artırılır. Örneğin, aşağıdaki numaralandırma tanımlayıcıların sırasıyla 0 ila 11 tamsayılarına ayarlandığı yeni bir tür, ay sıralaması oluşturur;

```
enum months {
JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
        };
```

1'den 12'ye kadar olan ayları numaralandırmak için şunu kullanın:

```
enum months {
JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
        };
```

Önceki numaralandırmadaki ilk değer açıkça 1'e ayarlandığından, kalan değerler 1'den 12'ye sıralanır. Aynı kapsamda erişilebilen herhangi bir numaralandırmadaki tanımlayıcılar benzersiz olmalıdır. Bir numaralandırmanın her bir numaralandırma sabitinin değeri, tanımlayıcıya bir değer atayarak tanımda açıkça ayarlanabilir. Bir numaralandırmanın birden çok üyesi aynı sabit değere sahip olabilir.

Şekil 10.18'deki programda, MonthName dizisinden yılın aylarını yazdırmak için bir for ifadesinde ay numaralandırma değişkeni kullanılır. MonthName [0] öğesini boş "" dizesi yaptık. Bir mantık hatasının oluştuğunu belirtmek için MonthName[0] öğesini ***ERROR*** gibi bir değere ayarlayabilirsiniz.

```c
// Fig. 10.18: fig10_18.c
// Using an enumeration
#include <stdio.h>

// enumeration constants represent months of the year
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
};

int main(void)
{
   // initialize array of pointers
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

   // loop through months
   for (enum months month = JAN; month <= DEC; ++month) {
      printf("%2d%11s\n", month, monthName[month]);
   }
}
```

Eğer bilgisayarınızda program çalışmazsa 18 nci satırdaki "enum months" yerine "int" yazın.

Bu bölümün başlarında struct ve union yapılarını tanıttık. C Adlandırılmış struct ve union yapıları içine yerleştirilebilen anonim struct ve union yapılarını da desteklemektedir. Bir anonim struct veya union yapısı içindeki üyelerin, çevreleyen struct veya union üyeleri olduğu kabul edilir ve bunlara, çevreleyen türdeki bir nesne aracılığıyla doğrudan erişilebilir. Örneğin, aşağıdaki yapı bildirimini göz önünde bulunduralım:

```
struct MyStruct {
        int member1;
        int member2;
        struct {
                int nestedMember1;
                int nestedMember2;
        }; // end nested struct
}; // end outer struct
```

Buradaki myStruct değişkeni için struct MyStruct türünde üyelere şu şekilde erişebilirsiniz:

```
myStruct.member1;
myStruct.member2;
myStruct.nestedMember1;
myStruct.nestedMember2;
```