Técnicas de Programação

TP0901

Prof. Giovane Barcelos
giovane_barcelos@uniritter.edu.br

Plano de Ensino
Conteúdo programático

1. Introdução à programação em C 2. Desenvolvimento estruturado de programas em C 3. Controle de programa 4. Funções 5. Arrays 6. Ponteiros	**N1**
7. Caracteres e strings 8. Entrada/Saída formatada 9. Estruturas, uniões, manipulações de bits e enumerações 10. Processamento de arquivos 11. Estruturas de dados 12. O pré-processador 13. Outros tópicos sobre C	**N2**

Objetivos

- A criar e usar estruturas, uniões e enumerações.
- A passar estruturas para funções por valor e por referência.
- A manipular dados com os operadores sobre bits.
- A criar campos de bit para armazenar dados de modo compacto.

Introdução

- **Estruturas — também chamadas de agregações — são coleções de variáveis relacionadas agrupadas sob um único nome.**
- **As estruturas podem conter variáveis de muitos tipos de dados diferentes — diferentemente dos arrays, que contêm apenas elementos do mesmo tipo de dado.**
- **As estruturas normalmente são usadas para declarar registros a serem armazenados em arquivos (ver Capítulo 11).**
- **Ponteiros e estruturas facilitam a formação de estruturas de dados mais complexas, por exemplo, listas interligadas, filas, pilhas e árvores (ver Capítulo 12).**

Declarações de estruturas

- Estruturas são tipos de dados derivados; elas são construídas a partir de objetos de outros tipos.
- Considere a declaração de estrutura a seguir:
  - ```
    struct card {
       char *face;
       char *suit;
    };
    ```
- A palavra-chave struct introduz a declaração de estrutura.
- O identificador card é a tag de estrutura, que dá nome à declaração de estrutura e é usado com a palavra-chave struct para declarar variáveis do tipo de estrutura.
    ```

Declarações de estruturas

- Nesse exemplo, o tipo de estrutura é struct card.
- As variáveis declaradas dentro das chaves de declaração da estrutura são os membros da estrutura.
- Os membros que têm o mesmo tipo de estrutura precisam ter nomes exclusivos, mas dois tipos de estrutura diferentes podem ter membros com o mesmo nome, sem conflito (logo veremos por quê).
- Todas as declarações de estrutura precisam terminar com pontos e vírgulas.

Declarações de estruturas

Erro comum de programação 10.1

Esquecer ponto e vírgula que termina uma declaração de estrutura provoca um erro de sintaxe.

Declarações de estruturas

- A declaração de struct card contém os membros face e suit do tipo char *.
- Os membros da estrutura podem ser variáveis dos tipos de dados primitivos (por exemplo, int, float etc.), ou agregações, como arrays e outras estruturas.
- Os membros da estrutura podem ser de muitos tipos.

Declarações de estruturas

- Por exemplo, a struct a seguir contém membros de array de caracteres para o nome e para o sobrenome de um funcionário, um membro int para sua idade, um membro char que conteria'M' ou 'F' para indicar a que sexo ele pertence e um membro double para seu salário:
  - ```
    struct funcionario {
       char nome[ 20 ];
       char sobrenome[ 20 ];
       int idade;
       char sexo;
       double salario;
    };
    ```
    ```

Declarações de estruturas

- Uma estrutura não pode conter uma instância de si mesma.
- Por exemplo, uma variável do tipo struct funcionario não pode ser declarada na definição para struct funcionario.
- Porém, um ponteiro para struct funcionario pode ser incluído.
- Por exemplo,

```
struct funcionario2 {
   char nome[ 20 ];
   char sobrenome[ 20 ];
   int idade;
   char sexo;
   double salario;
   struct funcionario2 pessoa; /* ERRO */
   struct funcionario2 *ePtr; /* ponteiro */
};
```

- struct funcionario2 contém uma instância de si mesma (pessoa), o que é um erro.

Declarações de estruturas

- Como ePtr é um ponteiro (para o tipo struct funcionario2), ele é permitido na declaração. Uma estrutura que contém um membro que é um ponteiro para o mesmo tipo de estrutura é chamada de **estrutura autorreferenciada**.
- As estruturas autorreferenciadas serão usadas no Capítulo 12 para criar estruturas de dados interligadas.

Declarações de estruturas

- As declarações da estrutura não reservam nenhum espaço na memória; em vez disso, cada declaração cria um novo tipo de dado, que é usado para declarar variáveis.
- As variáveis da estrutura são declaradas como variáveis de outros tipos.
- A declaração
  - `struct card aCard, deck[ 52 ], *cardPtr;`

  declara aCard como uma variável do tipo struct card, declara deck como um array com 52 elementos do tipo struct card e declara cardPtr como um ponteiro para struct card.

Declarações de estruturas

- As variáveis de determinado tipo de estrutura também podem ser declaradas colocando-se uma lista separada por vírgulas com os nomes de variável entre a chave que fecha a declaração de estrutura e o ponto e vírgula que encerra a declaração de estrutura.
- Por exemplo, a declaração anterior poderia ter sido incorporada na declaração de estrutura struct card da seguinte forma:

```
struct card {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Declarações de estruturas

- O nome para a tag de estrutura é opcional.
- Se uma declaração de estrutura não tiver um nome para a tag da estrutura, as variáveis do tipo da estrutura só poderão ser declaradas na declaração de estrutura, e não em uma declaração separada..

Declarações de estruturas

Boa prática de programação 10.1
Sempre forneça um nome para a tag de estrutura ao criar um tipo de estrutura. O nome para a tag de estrutura será
conveniente na declaração de novas variáveis do tipo de estrutura adiante no programa.

Boa prática de programação 10.2
Escolher um nome significativo para a tag de estrutura ajuda a tornar o programa documentado.

Declarações de estruturas

- **As únicas operações válidas que podem ser realizadas nas estruturas são as seguintes:**
  - **atribuição de variáveis da estrutura a variáveis da estrutura de mesmo tipo,**
  - **coleta de endereço (&) de uma variável de estrutura,**
  - **acesso aos membros de uma variável de estrutura (ver Seção 10.4) e**
  - **uso do operador sizeof para determinar o tamanho de uma variável de estrutura.**

Declarações de estruturas

Erro comum de programação 10.2
Atribuir uma estrutura de um tipo a uma estrutura de um tipo diferente provoca um erro de compilação.

Declarações de estruturas

- As estruturas não podem ser comparadas usando-se os operadores == e !=, pois os membros da estrutura não são necessariamente armazenados em bytes consecutivos de memória.
- Às vezes, existem 'buracos' em uma estrutura, pois os computadores podem armazenar tipos de dados específicos apenas em certos limites de memória, como os limites de meia palavra, palavra ou palavra dupla.
- Uma palavra é uma unidade de memória padrão usada para armazenar dados em um computador — normalmente, 2 bytes ou 4 bytes.

Declarações de estruturas

- **Considere uma declaração de estrutura em que sample1 e sample2 do tipo struct exemplo são declaradas:**
  - ```
    struct exemplo {
        char c;
        int i;
    } ex1, ex2;
    ```
- **Um computador com palavras de 2 bytes pode exigir que cada membro de struct exemplo seja alinhado em um limite de palavra (isso depende de cada máquina).**
    ```

Declarações de estruturas

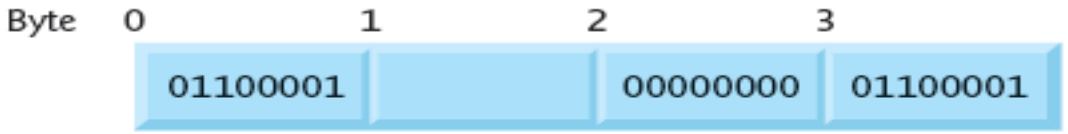


Figura 10.1 ■ Exemplo de alinhamento de armazenagem possível para uma variável do tipo `struct` que mostra uma área indefinida na memória.

Declarações de estruturas

- A Figura 10.1 mostra um exemplo de alinhamento de armazenagem para uma variável do tipo struct exemplo que recebeu o caractere 'a' e o inteiro 97 (as representaçãos de bit dos valores são mostradas).
- Se os membros forem armazenados começando nos limites de palavra, existe um intervalo de 1 byte (byte 1 na figura) no armazenamento para as variáveis do tipo struct exemplo.
- O valor no intervalo de 1 byte é indefinido.
- Mesmo que os valores dos membros de ex1 e ex2 sejam realmente iguais, as estruturas não serão necessariamente iguais, pois os intervalos indefinidos de 1 byte provavelmente não conterão valores idênticos.

Declarações de estruturas

Dica de portabilidade 10.1
Como o tamanho dos itens de dados de determinado tipo e as considerações de alinhamento de armazenagem dependem da máquina, o mesmo acontece com a representação de uma estrutura.

Inicialização de estruturas

- As estruturas podem ser inicializadas a partir de listas de inicializadores, assim como acontece com os arrays.
- Para inicializar uma estrutura, siga o nome da variável na declaração com um sinal de igual e uma lista, delimitada por chaves, de inicializadores separados por vírgulas.
- Por exemplo, a declaração
  - `struct card aCard = { "Três", "Copas" };`

  cria a variável aCard para ser do tipo struct card (conforme definido na Seção 10.2) e inicializa o membro face como "Três" e o membro suit como "Copas".

Inicialização de estruturas

- **Se o número de inicializadores na lista for menor que os membros na estrutura, os membros restantes serão automaticamente inicializados em 0 (ou NULL se o membro for um ponteiro).**
- **As variáveis de estrutura que forem declaradas fora de uma declaração de função (ou seja, que sejam externos a ela) serão inicializadas em 0 ou NULL, se não forem inicializadas explicitamente na declaração externa.**
- **As variáveis da estrutura também podem ser inicializadas nas instruções de atribuição, atribuindo uma variável de estrutura do mesmo tipo, ou valores aos membros individuais da estrutura.**

Acesso aos membros da estrutura

- Dois operadores são usados para acessar membros das estruturas: o **operador de membro de estrutura (.)** — também chamado de operador de ponto — e o **operador de ponteiro de estrutura (->)** — também chamado **operador de seta**.
- O operador de membro da estrutura acessa um membro da estrutura por meio do nome da variável da estrutura.
- Por exemplo, para imprimir o membro suit da variável da estrutura aCard declarada na Seção 10.3, use a instrução
  - `printf( "%s", aCard.suit ); /* mostra Copas */`

Acesso aos membros da estrutura

- O operador de ponteiro da estrutura — que consiste em um sinal de subtração (-) seguido de um sinal de maior (>) — acessa um membro da estrutura por meio de um ponteiro para a estrutura.
- Suponha que o ponteiro cardPtr tenha sido declarado para apontar para struct card, e que o endereço da estrutura aCard tenha sido atribuído a cardPtr.
- Para imprimir o membro suit da estrutura aCard com o ponteiro cardPtr, use a instrução
  - `printf( "%s", cardPtr->suit ); /* mostra Copas */`

Acesso aos membros da estrutura

- A expressão cardPtr->suit é equivalente a (*cardPtr).suit, que desreferencia o ponteiro e acessa o membro suit usando o operador de membro da estrutura.
- Aqui, os parâmetros são necessários porque o operador de membro da estrutura (.) tem uma precedência maior que o operador de desreferência de ponteiro (*).
- O operador de ponteiro da estrutura e o operador de membro da estrutura, com os parênteses (para chamar funções) e colchetes ([]) usados para subscrito de array, possuem a precedência de operador mais alta, e são associados da esquerda para a direita.

Acesso aos membros da estrutura

Boa prática de programação 10.3
Não coloque espaços em torno dos operadores ‘–>’ e ‘.’ — a omissão dos espaços ajuda a enfatizar o fato de que
as expressões em que os operadores estão contidos são basicamente nomes de variável únicos.

Erro comum de programação 10.3
A inserção de espaço entre os componentes – e > do operador de ponteiro da estrutura (ou entre os componentes de qualquer outro operador de múltiplos caracteres, com exceção de ?:) provoca um erro de sintaxe.

Acesso aos membros da estrutura

Erro comum de programação 10.4

Tentar referenciar a um membro de uma estrutura usando apenas o nome do membro provoca um erro de sintaxe.

Erro comum de programação 10.5

*Não usar parênteses ao referenciar um membro da estrutura que usa um ponteiro e o operador de membro da estrutura (por exemplo, *cardPtr.suit) provoca um erro de sintaxe.*

Acesso aos membros da estrutura

```
1  /* Figura 10.2: fig10_02.c
2     Usando os operadores de membro da estrutura
3     e de ponteiro da estrutura */
4  #include <stdio.h>
5
6  /* declaração da estrutura da carta */
7  struct card {
8     char *face; /* declara ponteiro face */
9     char *suit; /* declara ponteiro suit */
10 }; /* fim da estrutura card */
```

Figura 10.2 ■ Operador de membro da estrutura e operador de ponteiro da estrutura. (Parte 1 de 2.)

Acesso aos membros da estrutura

```
11
12  int main( void )
13  {
14     struct card aCard; /* declara uma variável struct card */
15     struct card *cardPtr; /* declara ponteiro para uma struct card */
16
17     /* coloca strings em aCard */
18     aCard.face = "Ás";
19     aCard.suit = "Espadas";
20
21     cardPtr = &aCard; /* atribui endereço de aCard a cardPtr */
22
23     printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " de ", aCard.suit,
24        cardPtr->face, " de ", cardPtr->suit,
25        ( *cardPtr ).face, " de ", ( *cardPtr ).suit );
26     return 0; /* indica conclusão bem-sucedida */
27  } /* fim do main */
```

```
Ás de Espadas
Ás de Espadas
Ás de Espadas
```

Figura 10.2 ■ Operador de membro da estrutura e operador de ponteiro da estrutura. (Parte 2 de 2.)

Acesso aos membros da estrutura

- O programa da Figura 10.2 demonstra o uso dos operadores de membro da estrutura e de ponteiro da estrutura.
- Usando o operador de membro da estrutura, os membros da estrutura aCard recebem os valores "Ace" e "Spades", respectivamente (linhas 18 e 19).
- O ponteiro cardPtr recebe o endereço da estrutura aCard (linha 21).
- A função printf imprime os membros da variável da estrutura aCard usando o operador de membro da estrutura com nome de variável aCard, o operador de ponteiro da estrutura com o ponteiro cardPtr e o operador de membro da estrutura com o ponteiro desreferenciado cardPtr (linhas 23 a 25).

Uso de estruturas com funções

- As estruturas podem ser passadas a funções ao passar membros da estrutura individuais, ao passar uma estrutura inteira ou um ponteiro para uma estrutura.
- Quando as estruturas ou membros individuais da estrutura são passados a uma função, eles são passados por valor.
- Portanto, os membros da estrutura passados por valor não podem ser modificados pela função utilizada.
- Para passar uma estrutura por referência, passe o endereço da variável da estrutura.

Uso de estruturas com funções

- **Os arrays das estruturas — assim como todos os outros arrays — são automaticamente passados por referência.**
- **Para passar um array por valor, crie uma estrutura que tenha o array como membro.**
- **As estruturas são passadas por valor, de modo que o array também seja passado por valor.**

Uso de estruturas com funções

Erro comum de programação 10.6
Supor que estruturas como arrays sejam passadas automaticamente por referência e tentar modificar os valores da
estrutura passados por valor na função utilizada consiste em um erro lógico.

Dica de desempenho 10.1
Passar estruturas por referência é mais eficiente do que passar estruturas por valor (o que requer que a estrutura
inteira seja copiada).

typedef

- A palavra-chave typedef oferece um mecanismo de criação de sinônimos (ou aliases) para tipos de dados previamente definidos.
- Os nomes dos tipos de estrutura normalmente são definidos a partir de typedef que se criem nomes de tipo mais curtos.
- Por exemplo, a instrução
  - typedef struct card Card;

  define o novo nome de tipo Card como um sinônimo para o tipo struct card.
- Programadores de C normalmente usam typedef para definir um tipo da estrutura, de modo que a tag da estrutura não é necessária.

typedef

- Por exemplo, a declaração a seguir

```
typedef struct {
   char *face;
   char *suit;
} Card;
```

cria o tipo de estrutura Card sem a necessidade de uma instrução typedef separada.

typedef

Boa prática de programação 10.4
Coloque a primeira letra dos nomes de typedef em maiúscula para enfatizar o fato de que eles são sinônimos de outros nomes para tipos.

typedef

- Card agora pode ser usado para declarar variáveis do tipo struct card.
- A declaração
  - Card deck[52];

  declara um array de 52 estruturas Card (ou seja, variáveis do tipo struct card).
- Criar um novo nome com typedef não cria um novo tipo; typedef simplesmente cria um novo nome de tipo, que pode ser usado como um alias para um nome de tipo existente.

typedef

- Um nome significativo ajuda a tornar o programa autoexplicativos.
- Por exemplo, quando lemos a declaração anterior, sabemos que “deck é um array de 52 Cards.”
- Frequentemente, typedef é usado para criar sinônimos para os tipos de dados básicos.
- Por exemplo, um programa que exige inteiros de 4 bytes pode usar o tipo int em um sistema e o tipo long em outro.
- Os programas projetados para portabilidade normalmente usam typedef para criar um alias para inteiros de 4 bytes, como Integer.
- O alias Integer ser alterado uma vez no programa para fazê-lo funcionar nos dois sistemas..

typedef

Dica de portabilidade 10.2

Use typedef para ajudar a tornar um programa mais portável.

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

```
1  /* Figura 10.3: fig10_03.c
2     Programa para embaralhar e distribuir cartas usando estruturas */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  /* declaração da estrutura card */
8  struct card {
9     const char *face; /* declara ponteiro face */
10    const char *suit; /* declara ponteiro suit */
11 }; /* fim da estrutura card */
12
13 typedef struct card Card; /* novo nome de tipo para struct card */
14
```

Figura 10.3 ■ Simulação de alto desempenho de embaralhamento e distribuição de cartas. (Parte 1 de 2.)

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

```
15  /* protótipos */
16  void fillDeck( Card * const wDeck, const char * wFace[],
17     const char * wSuit[] );
18  void shuffle( Card * const wDeck );
19  void deal( const Card * const wDeck );
20
21  int main( void )
22  {
23     Card deck[ 52 ]; /* declara array de cartas */
24
25     /* inicializa array de ponteiros */
26     const char *face[] = { "Ás", "Dois", "Três", "Quatro", "Cinco",
27        "Seis", "Sete", "Oito", "Nove", "Dez",
28        "Valete", "Dama", "Rei"};
29
30     /* inicializa array de ponteiros */
31     const char *suit[] = { "Copas", "Ouros", "Paus", "Espadas"};
32
33     srand( time( NULL ) ); /* torna aleatório */
34
35     fillDeck( deck, face, suit ); /* carrega o baralho com Cards */
36     shuffle( deck ); /* coloca Cards em ordem aleatória */
37     deal( deck ); /* distribui as 52 Cards */
38     return 0; /* indica conclusão bem-sucedida */
39  } /* fim do main */
40
41  /* coloca strings nas estruturas Card */
42  void fillDeck( Card * const wDeck, const char * wFace[],
43     const char * wSuit[] )
44  {
45     int i; /* contador */
46
47     /* loop por wDeck */
48     for ( i = 0; i <= 51; i++ ) {
49        wDeck[ i ].face = wFace[ i % 13 ];
50        wDeck[ i ].suit = wSuit[ i / 13 ];
51     } /* fim do for */
52  } /* fim da função fillDeck */
```

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

```
53
54 /* mistura cartas */
55 void shuffle( Card * const wDeck )
56 {
57    int i; /* contador */
58    int j; /* variável para manter valor aleatório entre 0 - 51 */
59    Card temp; /* declara estrutura temporária para trocar Cards */
60
61    /* loop por wDeck trocando cartas aleatoriamente */
62    for ( i = 0; i <= 51; i++ ) {
63       j = rand() % 52;
64       temp = wDeck[ i ];
65       wDeck[ i ] = wDeck[ j ];
66       wDeck[ j ] = temp;
67    } /* fim do for */
68 } /* fim da função shuffle */
69
70 /* distribui cartas */
71 void deal( const Card * const wDeck )
72 {
73    int i; /* contador */
74
75    /* loop por wDeck */
76    for ( i = 0; i <= 51; i++ ) {
77       printf( "%5s of %-8s%s", wDeck[ i ].face, wDeck[ i ].suit,
78          ( i + 1 ) % 4 ? "  " : "\n" );
79    } /* fim do for */
80 } /* fim da função deal */
```

Figura 10.3 ■ Simulação de alto desempenho de embaralhamento e distribuição de cartas. (Parte 2 de 2.)

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

Três de Copas	Valete de Paus	Três de Espadas	Seis de Ouros
Cinco de Copas	Oito de Espadas	Três de Paus	Dois de Espadas
Valete de Espadas	Quatro de Copas	Dois de Copas	Seis de Paus
Dama de Paus	Três de Ouros	Oito de Ouros	Rei de Paus
Rei de Copas	Oito de Copas	Dama de Copas	Sete de Paus
Sete de Ouros	Nove de Espadas	Cinco de Paus	Oito de Paus
Seis de Copas	Dois de Ouros	Cinco de Espadas	Quatro de Paus
Dois de Paus	Nove de Copas	Sete de Copas	Quatro de Espadas
Dez de Espadas	Rei de Ouros	Dez de Copas	Valete de Ouros
Quatro de Ouros	Seis de Espadas	Cinco de Ouros	Ás de Ouros
Ás de Paus	Valete de Copas	Dez de Paus	Dama de Ouros
Ás de Copas	Dez de Ouros	Nove de Paus	Rei de Espadas
Ás de Espadas	Nove de Ouros	Sete de Espadas	Dama de Espadas

Figura 10.4 ■ Apresentação dos resultados da simulação de embaralhamento e distribuição de cartas.

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

- O programa da Figura 10.3 baseia-se na simulação de embaralhamento e distribuição de cartas discutida no anteriormente.
- O programa representa o baralho de cartas como um array de estruturas.
- O programa usa algoritmos de alto desempenho para embaralhar e distribuir as cartas.
- A saída do programa de embaralhamento e distribuição de cartas aparece na Figura 10.4.

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

- No programa, a função fillDeck (linhas 42-52) inicializa o array Card na ordem de Ás a Rei de cada naipe.
- O array Card é passado (na linha 36) à função shuffle (linhas 55-68), em que o algoritmo de embaralhamento de alto desempenho é implementado.
- A função shuffle usa um array de 52 estruturas Card como argumento.
- A função percorre as 52 cartas (subscritos de array 0 a 51) usando uma estrutura for nas linhas 62-67.

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

- Para cada carta, um número entre 0 e 51 é escolhido aleatoriamente.
- Em seguida, a estrutura Card atual e a estrutura Card selecionada aleatoriamente são trocados no array (linhas 64 a 66).
- Um total de 52 trocas é feito em uma única passada do array inteiro, e o array das estruturas Card é embaralhado!
- Esse algoritmo não pode sofrer adiamento indefinido, como o algoritmo de embaralhamento apresentado no Capítulo 7.
- Como as estruturas Card foram trocadas no espaço do array, o algoritmo de alto desempenho implementado na função deal (linhas 71-80) exige apenas uma passada do array para distribuir as cartas embaralhadas.

Exemplo: uma simulação de alto desempenho de embaralhamento e distribuição de cartas

Erro comum de programação 10.7

Esquecer de incluir o subscrito do array ao se referir a estruturas individuais em um array de estruturas consiste em um erro de sintaxe.

Uniões

- Uma **união** é um tipo de dado derivado — como uma estrutura — com membros que compartilham o mesmo espaço de armazenamento.
- Para diferentes situações em um programa, algumas variáveis podem não ser relevantes, mas outras variáveis o são, de modo que uma união compartilha o espaço em vez de desperdiçar armazenamento em variáveis que não são mais usadas.
- Os membros de uma união podem ser de qualquer tipo de dado

Uniões

- O número de bytes usados para armazenar uma união precisa ser, pelo menos, o suficiente para manter o maior membro.
- Na maior parte dos casos, as uniões contêm dois ou mais tipos de dados.
- Apenas um membro, e, portanto, um tipo de dado, pode ser referenciado a cada vez.
- É de responsabilidade do programador garantir que os dados em uma união sejam referenciados com o tipo apropriado.

Uniões

Erro comum de programação 10.8

Referir-se aos dados em uma união com uma variável do tipo errado consiste em um erro lógico.

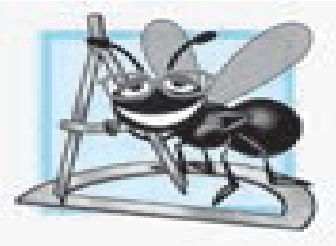
Dica de portabilidade 10.3

Se os dados são armazenados em uma união como um tipo e referenciados como outro tipo, os resultados dependerão da implementação.

Uniões

- Uma união é declarada com a palavra-chave union no mesmo formato de uma estrutura.
- A declaração de union
  - ```
    union número {
       int x;
       double y;
    };
    ```
- indica que número é um tipo union com os membros int x e double y.
- A declaração da união normalmente é colocada no cabeçalho e incluída em todos os arquivos-fonte que usam o tipo união.
    ```

Uniões



Observação sobre engenharia de software 10.1

Assim como a declaração de struct, uma de union simplesmente cria um novo tipo. Colocar uma declaração de union ou struct fora de uma função não cria uma variável global.

Uniões

- As operações que podem ser realizadas em uma união são as seguintes:
  - atribuição de uma união a outra união do mesmo tipo,
  - coleta do endereço (&) de uma variável de união
  - e acesso dos membros da união usando o operador de membro da estrutura e o operador de ponteiro da estrutura.
- As uniões não podem ser comparadas com os operadores == e != pelos mesmos motivos pelos quais as estruturas não podem ser comparadas.

Uniões

- Em uma declaração, uma união pode ser inicializada com um valor do mesmo tipo que o primeiro membro da união.
- Por exemplo, com a união anterior, a declaração
  - `union número valor = { 10 };`

  é uma inicialização válida de uma variável de união valor, pois a união é inicializada com um int, mas a declaração seguinte truncaria a parte de ponto flutuante do valor inicializador, e normalmente produziria uma advertência do compilador:

Uniões

Erro comum de programação 10.9
Comparar uniões consiste em um erro de sintaxe.

Dica de portabilidade 10.4
A quantidade de armazenamento exigida no caso de uma união depende da implementação, mas sempre será,
pelo menos, tão grande quanto o maior membro da união.

Uniões

Dica de portabilidade 10.5
Algumas uniões podem não ser facilmente portáveis para outros sistemas de computação. Se uma união é portável
ou não, isso depende dos requisitos de alinhamento de armazenagem para os tipos de dados de membro da união
em determinado sistema.

Dica de desempenho 10.2
As uniões economizam espaço de armazenamento.

Uniões

```
 1  /* Figura 10.5: fig10_05.c
 2     Exemplo de uma união */
 3  #include <stdio.h>
 4
 5  /* declaração da union number */
 6  union number {
 7     int x;
 8     double y;
 9  }; /* fim da união number */
10
11  int main( void )
12  {
13     union number value; /* declara variável de union */
14
15     value.x = 100; /* coloca um inteiro na union */
16     printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n\n\n",
17        "Coloca um valor no membro inteiro",
18        "e mostra os dois membros.",
19        "int:", value.x,
20        "double:", value.y );
21
```

Uniões

```
22      value.y = 100.0; /* coloca um double na mesma union */
23      printf( "%s\n%s\n%s\n  %d\n\n%s\n  %f\n",
24         "Coloca um valor no membro de ponto flutuante",
25         "e mostra os dois membros.",
26         "int:", value.x,
27         "double:", value.y );
28      return 0; /* indica conclusão bem-sucedida */
29   } /* fim do main */
```

```
Coloca um valor no membro inteiro
e exibe os dois membros.
int:
  100

double:
  -92559592117433136000000000000000000000000000000000000000.000000

Coloca um valor no membro de ponto flutuante
e exibe os dois membros.
int:
  0
double:
  100.000000
```

Figura 10.5 ■ Exibição do valor de uma união nos dois tipos de dados membros.

Uniões

- O programa na Figura 10.5 usa a variável value(linha 13) do tipo union number para exibir o valor armazenado na união como um int e um double.
- A saída do programa depende da implementação.
- A saída do programa mostra que a representação interna de um valor double pode ser muito diferente da representação de int.

Operadores sobre bits

- Computadores representam internamente todos os dados como sequências de bits.
- Os bits podem assumir o valor 0 ou o valor 1.
- Na maioria dos sistemas, uma sequência de 8 bits forma um byte — unidade de armazenamento-padrão para uma variável do tipo char.
- Outros tipos de dados são armazenados em números maiores de bytes.
- Os operadores sobre bits são usados para manipular os bits de operandos inteiros (char, short, int e long; tanto signed quanto unsigned).
- Os inteiros sem sinal normalmente são usados com os operadores sobre bits.

Operadores sobre bits

Dica de portabilidade 10.6

Manipulações de dados sobre bits são dependentes da máquina.

Operadores sobre bits

- As discussões de operador sobre bits nessa seção mostram as representações binárias dos operandos inteiros. Para uma explicação detalhada do sistema numérico binário (também chamado de base 2), consulte o Apêndice C.
- Além disso, os programas nas seções 10.9 e 10.10 foram testados usando o Microsoft Visual C++.
- Devido à natureza dependente da máquina quanto às manipulações de bits, esses programas podem não funcionar no seu sistema.
- Os operadores sobre bits são **AND sobre bits (&)**, **OR inclusivo sobre bits (|)**, **OR exclusivo sobre bits (^)**, **deslocamento à esquerda (<<)**, **deslocamento à direita (>>)** e **complemento (~)**.

Operadores sobre bits

- Os operadores AND sobre bits, OR inclusivo sobre
- bits e OR exclusivo sobre bits comparam seus dois operandos bit a bit.
- O operador AND sobre bits define como 1 cada bit do resultado, se o bit correspondente nos dois operandos for 1.
- O operador OR inclusivo sobre bits define como 1 cada bit do resultado se o bit corresponde em um ou em ambos os operandos for 1.
- O operador OR exclusivo define como 1 cada bit do resultado, se o bit correspondente em exatamente um operando for 1.

Operadores sobre bits

- O operador de deslocamento à esquerda desloca os bits de seu operando esquerdo para a esquerda pelo número de bits especificado em seu operando da direita.
- O operador de deslocamento à direita desloca os bits em seu operando da esquerda para a direita pelo número de bits especificado em seu operando da direita.
- O operador de complemento sobre bits define todos os bits 0 em seu operando como 1 no resultado e define todos os bits 1 como 0 no resultado.
- Discussões detalhadas sobre cada operador sobre bits aparecem nos exemplos a seguir.
- Os operadores sobre bits estão resumidos na Figura 10.6.

Operadores sobre bits

Operador		Descrição
&	AND sobre bits	Os bits são definidos como 1 no resultado, se os bits correspondentes em ambos os operandos forem 1.
|	OR inclusivo sobre bits	Os bits são definidos como 1 no resultado, se pelo menos um dos bits correspondentes em ambos os operandos for 1.
^	OR exclusivo sobre bits	Os bits são definidos como 1 no resultado, se exatamente um dos bits correspondentes em ambos os operandos for 1.
<<	deslocamento à esquerda	Desloca os bits do primeiro operando à esquerda pelo número de bits especificado pelo segundo operando; preenche a partir da direita com 0 bits.
>>	deslocamento à direita	Desloca os bits do primeiro operando à direita pelo número de bits especificado pelo segundo operando; o método de preenchimento a partir da esquerda depende da máquina.
~	complemento de um	Todos os bits 0 são definidos como 1, e todos os bits 1 são definidos como 0.

Figura 10.6 ■ Operadores sobre bits.

Operadores sobre bits

```
 1  /* Figura 10.7: fig10_07.c
 2     Imprimindo um inteiro sem sinal em bits */
 3  #include <stdio.h>
 4
 5  void displayBits( unsigned value ); /* protótipo */
 6
 7  int main( void )
 8  {
 9     unsigned x; /* variável para manter entrada do usuário */
10
11     printf( "Digite um inteiro sem sinal: " );
12     scanf( "%u", &x );
13
14     displayBits( x );
15     return 0; /* indica conclusão bem-sucedida */
16  } /* fim do main */
17
18  /* mostra bits de um valor inteiro sem sinal */
19  void displayBits( unsigned value )
20  {
21     unsigned c; /* contador */
22
23     /* declara displayMask e desloca 31 bits à esquerda */
24     unsigned displayMask = 1 << 31;
25
26     printf( "%10u = ", value );
27
```

Operadores sobre bits

```
28      /* percorre os bits */
29      for ( c = 1; c <= 32; c++ ) {
30         putchar( value & displayMask ? '1' : '0' );
31         value <<= 1; /* desloca valor à esquerda em 1 */
32
33         if ( c % 8 == 0 ) { /* gera espaço após 8 bits */
34            putchar( ' ' );
35         } /* fim do if */
36      } /* fim do for */
37
38      putchar( '\n' );
39   } /* fim da função displayBits */
```

```
Digite um inteiro sem sinal: 65000
      65000 = 00000000 00000000 11111101 11101000
```

Figura 10.7 ■ Exibição de um inteiro sem sinal em bits.

Operadores sobre bits

- Ao usar os operadores sobre bits, é útil imprimir valores em sua representação binária para ilustrar os efeitos exatos desses operadores.
- O programa da Figura 10.7 imprime um inteiro unsigned em sua representação binária em grupos de 8 bits cada.
- Para os exemplos nessa seção, consideraremos que os inteiros unsigned sejam armazenados em 4 bytes (32 bits) de memória.

Operadores sobre bits

- A função displayBits (linhas 19-39) usa o operador AND sobre bits para combinar a variável value com a variável displayMask (linha 32).
- Frequentemente, o operador AND sobre bits é usado com um operando chamado máscara — um valor inteiro com bits específicos definidos como 1.
- As máscaras são usadas para ocultar alguns bits em um valor enquanto seleciona outros bits.
- Na função displayBits, a variável máscara displayMask recebe o valor
  - 1 << 31 (10000000 00000000 00000000 00000000)

Operadores sobre bits

- O operador de deslocamento à esquerda desloca o valor 1 a partir do bit de baixa ordem (mais à direita) para o bit de alta ordem (mais à esquerda) em displayMask, e preenche bits 0 a partir da direita.
- A linha 32.
  - putchar(value & displayMask ? '1' : '0');

  determina se 1 ou 0 deve ser impresso para o bit mais à esquerda da variável value.
- Quando value e displayMask são combinados usando &, todos os bits, exceto o bit de alta ordem na variável value são 'mascarados' (ocultados), pois qualquer bit que passe por AND com 0 gera 0.

Operadores sobre bits

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Figura 10.8 ■ Resultados da combinação de dois bits com o operador AND sobre bits (&).

Operadores sobre bits

- Se o bit mais à esquerda for 1, value & displayMask será avaliado como um valor diferente de zero (verdadeiro), e 1 será impresso — caso contrário, 0 será impresso.
- A variável value é, então, deslocada um bit à esquerda pela expressão value <<= 1 (isso equivale a value = value << 1).
- Essas etapas são repetidas para cada bit na variável unsigned value.
- A Figura 10.8 resume os resultados da combinação dos dois bits com o operador AND sobre bits.

Operadores sobre bits

Erro comum de programação 10.10

Usar o operador lógico AND (&&) para o operador AND sobre bits (&) e vice-versa consiste em um erro.

Operadores sobre bits

- Na linha 24 da Figura 10.7, codificamos o inteiro `31` para indicar que o valor `1` deveria ser deslocado para o bit mais à esquerda na variável `displayMask`.
- De modo semelhante, na linha 29, codificamos o inteiro `32` para indicar que o loop deveria ser repetido 32 vezes — uma vez para cada bit da variável `value`.
- Consideramos que os inteiros `unsigned` são armazenados em 32 bits (4 bytes) de memória.
- Muitos dos computadores populares de hoje usam arquiteturas de hardware com *word* (palavra) de 32 bits.

Operadores sobre bits

- Os programadores de C costumam trabalhar em muitas arquiteturas de hardware, e às vezes os inteiros unsigned são armazenados em números menores ou maiores de bits.
- O programa na Figura 10.7 pode se tornar mais escalável e mais portável se substituirmos o inteiro 31 na linha 24 pela expressão
  - `CHAR_BIT * sizeof( unsigned ) - 1`

  e se substituirmos o inteiro 32 na linha 29 pela expressão
  - `CHAR_BIT * sizeof( unsigned )`

Operadores sobre bits

- A constante simbólica CHAR_BIT (definida em <limits.h>) representa o número de bits em um byte (normalmente, esse número é 8).
- Em um computador que use palavras de 32 bits, a expressão sizeof(unsigned) será avaliada como 4, de modo que as duas expressões precedentes serão avaliadas como 31 e 32, respectivamente.
- Em um computador que use palavras de 16 bits, a expressão sizeof será avaliada como 2, e as duas expressões precedentes serão avaliadas como 15 e 16, respectivamente.

Operadores sobre bits

```
1  /* Figura 10.9: fig10_09.c
2     Usando operadores AND sobre bits, OR inclusivo sobre bits,
3     OR exclusivo sobre bits e de complemento sobre bits */
4  #include <stdio.h>
5
6     void displayBits( unsigned value ); /* protótipo */
7
8  int main( void )
9  {
10    unsigned number1;
11    unsigned number2;
12    unsigned mask;
13    unsigned setBits;
```

Figura 10.9 ■ Operadores AND, OR inclusivo, OR exclusivo e complemento sobre bits. (Parte 1 de 2.)

Operadores sobre bits

```
14
15      /* demonstra AND sobre bits (&)*/
16      number1 = 65535;
17      mask = 1;
18      printf( "O resultado da combinação dos seguintes\n" );
19      displayBits( number1 );
20      displayBits( mask );
21      printf( "usando o operador AND sobre bits & é\n" );
22      displayBits( number1 & mask );
23
24      /* demonstra OR inclusivo sobre bits (|) */
25      number1 = 15;
26      setBits = 241;
27      printf( "\nO resultado da combinação dos seguintes\n" );
28      displayBits( number1 );
29      displayBits( setBits );
30      printf( "usando o operador OR inclusivo sobre bits \n" );
31      displayBits( number1  |  setBits );
32
33      /* demonstra OR exclusivo sobre bits (^) */
34      number1 = 139;
35      number2 = 199;
36      printf( "\nO resultado da combinação dos seguintes\n" );
37      displayBits( number1 );
38      displayBits( number2 );
39      printf( "usando o operador OR exclusivo sobre bits ^ é\n" );
40      displayBits( number1 ^ number2 );
41
```

Operadores sobre bits

```
42     /* demonstra complemento sobre bits (~)*/
43     number1 = 21845;
44     printf( "\nO complemento de um \n" );
45     displayBits( number1 );
46     printf( "é\n" );
47     displayBits( ~number1 );
48     return 0; /* indica conclusão bem-sucedida */
49  } /* fim do main */
50
51  /* mostra bits de um valor inteiro sem sinal */
52  void displayBits( unsigned value )
53  {
54     unsigned c; /* contador */
55
56     /* declara displayMask e desloca 31 bits à esquerda */
57     unsigned displayMask = 1 << 31;
58
59     printf( "%10u = ", value );
60
61     /* loop pelos bits */
62     for ( c = 1; c <= 32; c++ ) {
63        putchar( value & displayMask ? '1' : '0' );
64        value <<= 1; /* desloca valor 1 bit à esquerda */
65
66        if ( c % 8 == 0 ) { /* apresenta espaço após 8 bits */
67           putchar( ' ' );
68        } /* fim do if */
69     } /* fim do for */
70
71     putchar( '\n' );
72  } /* fim da função displayBits */
```

Figura 10.9 ■ Operadores AND, OR inclusivo, OR exclusivo e complemento sobre bits. (Parte 2 de 2.)

Operadores sobre bits

```
O resultado da combinação seguinte
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
usando o operador AND sobre bits & é
         1 = 00000000 00000000 00000000 00000001

O resultado da combinação seguinte
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
usando o operador OR inclusivo sobre bits | é
       255 = 00000000 00000000 00000000 11111111

O resultado da combinação seguinte
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
usando o operador OR exclusivo sobre bits ^ é
        76 = 00000000 00000000 00000000 01001100

O complemento de um de
     21845 = 00000000 00000000 01010101 01010101
é
4294945450 = 11111111 11111111 10101010 10101010
```

Figura 10.10 ■ Saída do programa da Figura 10.9.

Operadores sobre bits

- A Figura 10.9 demonstra o uso dos operadores AND sobre bits, OR inclusivo sobre bits, OR exclusivo sobre bits e de complemento sobre bits.
- O programa usa a função displayBits (linhas 53-74) para imprimir os valores inteiros unsigned.
- A saída aparece na Figura 10.10.

Operadores sobre bits

- Na Figura 10.9, a variável inteira number1 recebe o valor 65535 (00000000 00000000 11111111 11111111) na linha 16, e a variável mask recebe o valor 1 (00000000 00000000 00000000 00000001) na linha 17.
- Quando number1 e mask são combinados usando o operador AND sobre bits (&) na expressão number1 & mask (linha 22), o resultado é 00000000 00000000 00000000 00000001.
- Todos os bits, exceto o de baixa ordem na variável number1, são 'mascarados' (ocultados) pelo AND com a variável mask.
- O operador OR inclusivo sobre bits é usado para definir bits específicos como 1 em um operando.

Operadores sobre bits

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Figura 10.11 ■ Resultados da combinação de dois bits com o operador OR inclusivo |.

Operadores sobre bits

- **Na Figura 10.9, a variável number1 recebe 15 (00000000 00000000 00000000 00001111) na linha 25, e a variável setBits recebe 241 (00000000 00000000 00000000 11110001) na linha 26.**
- **Quando number1 e setBits são combinados usando o operador OR sobre bits na expressão number1 | setBits (linha 31), o resultado é 255 (00000000 00000000 00000000 11111111).**
- **A Figura 10.11 resume os resultados da combinação dos dois bits com o operador OR inclusivo sobre bits.**

Operadores sobre bits

Erro comum de programação 10.11
Usar o operador lógico OR (||) como operador OR sobre bits (|) e vice-versa consiste em um erro.

Operadores sobre bits

- O operador OR exclusivo sobre bits (^) define cada bit no resultado como 1, se *exatamente* um dos bits correspondentes em seus dois operandos for 1.
- Na Figura 10.9, as variáveis number1 e number2 recebem os valores 139 (00000000 00000000 00000000 10001011) e 199 (00000000 00000000 00000000 11000111) nas linhas 34-35.
- Quando essas variáveis são combinadas com o operador OR exclusivo na expressão number1 ^ number2 (linha 40), o resultado é 00000000 00000000 00000000 01001100.
- A Figura 10.12 resume os resultados da combinação dos dois bits com o operador OR exclusivo sobre bits.

Operadores sobre bits

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Figura 10.12 ■ Resultados da combinação dos dois bits com o operador OR exclusivo sobre bits ^.

Operadores sobre bits

- O operador de complemento sobre bits (~) define todos os bits 1 em seu operando como 0 no resultado e define todos os bits 0 como 1 no resultado — também referenciado como 'obter o complemento de um do valor'.
- Na Figura 10.9, a variável number1 recebe o valor 21845 (00000000 00000000 01010101 01010101) na linha 43.
- Quando a expressão ~number1 (linha 47) é avaliada, o resultado é 00000000 00000000 10101010 10101010.

Operadores sobre bits

```
1  /* Figura 10.13: fig10_13.c
2     Usando os operadores de deslocamento sobre bits */
3  #include <stdio.h>
4
5  void displayBits( unsigned value ); /* protótipo */
6
7  int main( void )
8  {
9     unsigned number1 = 960; /* inicializa number1 */
10
11    /* demonstra deslocamento à esquerda sobre bits */
12    printf( "\nO resultado do deslocamento à esquerda de\n" );
13    displayBits( number1 );
14    printf( "por 8 posições de bit usando o " );
15    printf( "operador de deslocamento à esquerda << é\n" );
16    displayBits( number1 << 8 );
17
18    /* demonstra deslocamento à direita sobre bits */
19    printf( "\nO resultado do deslocamento à direita de\n" );
20    displayBits( number1 );
21    printf( "por 8 posições de bit usando o " );
22    printf( "operador de deslocamento à direita >> é\n" );
23    displayBits( number1 >> 8 );
24    return 0; /* indica conclusão bem-sucedida */
25 } /* fim do main */
26
27 /* mostra bits de um valor inteiro sem sinal */
28 void displayBits( unsigned value )
29 {
30    unsigned c; /* contador */
31
```

Operadores sobre bits

```
32       /* declara displayMask e desloca 31 bits à esquerda */
33       unsigned displayMask = 1 << 31;
34
35       printf( "%7u = ", value );
36
37       /* loop pelos bits */
38       for ( c = 1; c <= 32; c++ ) {
39          putchar( value & displayMask ? '1' : '0' );
40          value <<= 1; /* desloca valor 1 bit à esquerda */
41
42          if ( c % 8 == 0 ) { /* mostra um espaço após 8 bits */
```

Figura 10.13 ■ Operadores de deslocamento sobre bits. (Parte 1 de 2.)

Operadores sobre bits

```
43            putchar( ' ' );
44         } /* fim do if */
45      } /* fim do for */
46
47      putchar( '\n' );
48   } /* fim da função displayBits */
```

```
O resultado do deslocamento à esquerda
    960 = 00000000 00000000 00000011 11000000
por 8 posições de bit usando o operador de deslocamento à esquerda << é
 245760 = 00000000 00000011 11000000 00000000

O resultado do deslocamento à direita
    960 = 00000000 00000000 00000011 11000000
por 8 posições de bit usando o operador de deslocamento à direita >> é
      3 = 00000000 00000000 00000000 00000011
```

Figura 10.13 ■ Operadores de deslocamento sobre bits. (Parte 2 de 2.)

Operadores sobre bits

- O programa da Figura 10.13 mostra os operadores de deslocamento à esquerda (<<) e à direita (>>).
- A função displayBits é usada para imprimir os valores inteiros unsigned.

Operadores sobre bits

- O operador de deslocamento à esquerda (<<) desloca os bits de seu operando à esquerda para a esquerda pelo número de bits especificado em seu operando da direita.
- Os bits vagos à direita são substituídos por 0s; 1s deslocados para a esquerda são perdidos.
- Nas Figura 10.13, a variável number1 recebe o valor 960 (00000000 00000000 00000011 11000000) na linha 9.
- O resultado do deslocamento para a esquerda da variável number1 por 8 bits na expressão number1 << 8 (linha 16) é 49152 (00000000 00000000 11000000 00000000).

Operadores sobre bits

- O operador de deslocamento à direita (>>) desloca os bits de seu operando à esquerda para a direita pelo número de bits especificado em seu operando da direita.
- A realização do deslocamento à direita sobre um inteiro unsigned faz com que os bits vagos à esquerda sejam substituídos por 0s; 1s deslocados para a direita são perdidos.
- Na Figura 10.13, o resultado do deslocamento de number1 >> 8 (linha 23) é 3 (00000000 00000000 00000000 00000011).

Operadores sobre bits

Erro comum de programação 10.12

O resultado do deslocamento de um valor é indefinido se o operando da direita for negativo ou se o operando da direita for maior que o número de bits em que o operando da esquerda estiver armazenado.

Dica de portabilidade 10.7

O deslocamento para a direita é uma operação dependente da máquina. Deslocar um inteiro com sinal para a
direita preenche os bits vagos com 0s em algumas máquinas e com 1s em outras.

Operadores sobre bits

Operadores de atribuição sobre bits	
&=	Operador de atribuição sobre bits AND.
|=	Operador de atribuição sobre bits OR inclusivo.
^=	Operador de atribuição sobre bits OR exclusivo.
<<=	Operador de alinhamento à esquerda com atribuição.
>>=	Operador de alinhamento à direita com atribuição.

Figura 10.14 ■ Operadores de atribuição sobre bits.

Operadores sobre bits

- **Cada operador binário sobre bits tem um operador de atribuição correspondente.**
- **Esses operadores de atribuição sobre bits aparecem na Figura 10.14, e são usados de maneira semelhante à dos operadores de atribuição aritméticos apresentados anteriormente**

Operadores sobre bits

Operador	Associatividade	Tipo
`()  []  .  ->`	esquerda para direita	mais alto
`+ - ++ -- ! & * ~ sizeof (tipo)`	direita para esquerda	unário
`* / %`	esquerda para direita	multiplicativo
`+ -`	esquerda para direita	aditivo
`<< >>`	esquerda para direita	deslocamento
`< <= > >=`	esquerda para direita	relacional
`== !=`	esquerda para direita	igualdade
`&`	esquerda para direita	AND sobre bits
`^`	esquerda para direita	OR sobre bits
`|`	esquerda para direita	OR sobre bits
`&&`	esquerda para direita	AND lógico
`||`	esquerda para direita	OR lógico
`?:`	direita para esquerda	condicional
`= += -= *= /= &= |= ^= <<= >>= %=`	direita para esquerda	atribuição
`,`	esquerda para direita	vírgula

Figura 10.15 ■ Precedência e associatividade de operadores.

Operadores sobre bits

- A Figura 10.15 mostra a precedência e a associatividade dos diversos operadores apresentados até agora.
- Eles aparecem de cima para baixo em ordem decrescente de precedência.

Campos de bit

- C permite que você especifique o número de bits em que um membro unsigned ou int de uma estrutura ou união é armazenado.
- Isso é conhecido como campo de bits.
- Os campos de bits permitem uma utilização da memória mais competente, armazenando dados no número mínimo de bits exigido.
- Os membros de campo de bit *devem* ser declarados comoint ou unsigned.

Campos de bit

Dica de desempenho 10.3

Os campos de bit ajudam a economizar armazenamento.

Campos de bit

- Considere a seguinte definição de estrutura:

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

Ela contém três campos de bits unsigned — face, suit e color — usados para representar uma carta de um baralho de 52 cartas.

Campos de bit

- Um campo de bit é declarado ao se colocar um nome de membro unsigned ou int seguido por um sinal de dois-pontos (:) e uma constante inteira representando a largura do campo (ou seja, o número de bits em que o membro é armazenado).
- A constante que representa a largura precisa ser um inteiro entre 0 e o número total de bits usados para armazenar um int inclusive em seu sistema.
- Nossos exemplos foram testados em um computador com inteiros de 4 bytes (32 bits).
- A declaração da estrutura apresentada indica que o membro face é armazenado em 4 bits, o membro suit é armazenado em 2 bits e o membro color é armazenado em 1 bit.

Campos de bit

- O número de bits é baseado no intervalo de valores desejado para cada membro da estrutura.
- O membro face armazena valores de 0 (Ás) até 12 (Rei) — 4 bits podem armazenar valores no intervalo de 0 a 15.
- O membro suit armazena valores de 0 a 3 (0 = Ouros, 1 = Copas, 2 = Paus, 3 = Espadas) — 2 bits podem armazenar valores no intervalo 0–3.
- Finalmente, o membro color armazena 0 (vermelho) ou 1 (preto) — o bit 1 pode armazenar 0 ou 1.
- A Figura 10.16 (saída do programa mostrada na Figura 10.17) cria o array deck contendo 52 estruturas struct bitCard na linha 20.

Campos de bit

- A função fillDeck (linhas 28-38) insere as 52 cartas no array deck, e a função deal (linhas 42-54) imprime as 52 cartas.
- Observe que os membros de campo de bit das estruturas são acessados exatamente como qualquer outro membro da estrutura.
- O membro color é incluído como um meio de indicar a cor da carta em um sistema que permite exibições de cor. É possível especificar um campo de bit não nomeado para ser usado como preenchimento na estrutura.

Campos de bit

- Por exemplo, a definição da estrutura
  - ```
    struct exemplo {
       unsigned a : 13;
       unsigned   : 19;
       unsigned b : 4;
    };
    ```

  usa um campo de 19 bits não nomeado para preencher espaço — nada pode ser armazenado nesses 19 bits.
- O membro b (em nosso computador com palavra (*word*) de 4 bits) é armazenado em outra unidade de armazenamento.
    ```

Campos de bit

```
1  /* Figura 10.16: fig10_16.c
2     Representando cartas com campos em uma struct */
3
4  #include <stdio.h>
5
6  /* declaração da estrutura bitCard com campos de bit */
7  struct bitCard {
8     unsigned face : 4; /* 4 bits; 0-15 */
9     unsigned suit : 2; /* 2 bits; 0-3 */
10    unsigned color : 1; /* 1 bit; 0-1 */
11 }; /* fim da struct bitCard */
12
13 typedef struct bitCard Card; /* novo nome de tipo para a struct bitCard */
14
15 void fillDeck( Card * const wDeck ); /* protótipo */
16 void deal( const Card * const wDeck ); /* protótipo */
17
18 int main( void )
19 {
20    Card deck[ 52 ]; /* cria array de Cards */
21
22    fillDeck( deck );
23    deal( deck );
24    return 0; /* indica conclusão bem-sucedida */
25 } /* fim do main */
26
27 /* inicializa Cards */
28 void fillDeck( Card * const wDeck )
29 {
30    int i; /* contador */
31
```

Campos de bit

```
32       /* loop por wDeck */
33       for ( i = 0; i <= 51; i++ ) {
34          wDeck[ i ].face = i % 13;
35          wDeck[ i ].suit = i / 13;
36          wDeck[ i ].color = i / 26;
37       } /* fim do for */
38    } /* fim da função fillDeck */
39
40    /* apresenta cartas em formato de duas colunas; cartas 0-25 subscritadas
41       com k1 (coluna 1); cartas 26-51 subscritadas com k2 (coluna 2) */
42    void deal( const Card * const wDeck )
43    {
44       int k1; /* subscritos 0-25 */
45       int k2; /* subscritos 26-51 */
46
47       /* loop por wDeck */
48       for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
49          printf( "Carta:%3d  Naipe:%2d  Cor:%2d   ",
50             wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
51          printf( "Carta:%3d  Naipe:%2d  Cor:%2d\n",
52             wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
53       } /* fim do for */
54    } /* fim da função deal */
```

Figura 10.16 ■ Campos de bit para a armazenagem de um baralho.

Campos de bit

```
Carta:     0  Naipe:    0   Cor:      0   Carta:    0   Naipe:    2   Cor:      1
Carta:     1  Naipe:    0   Cor:      0   Carta:    1   Naipe:    2   Cor:      1
Carta:     2  Naipe:    0   Cor:      0   Carta:    2   Naipe:    2   Cor:      1
Carta:     3  Naipe:    0   Cor:      0   Carta:    3   Naipe:    2   Cor:      1
Carta:     4  Naipe:    0   Cor:      0   Carta:    4   Naipe:    2   Cor:      1
Carta:     5  Naipe:    0   Cor:      0   Carta:    5   Naipe:    2   Cor:      1
Carta:     6  Naipe:    0   Cor:      0   Carta:    6   Naipe:    2   Cor:      1
Carta:     7  Naipe:    0   Cor:      0   Carta:    7   Naipe:    2   Cor:      1
Carta:     8  Naipe:    0   Cor:      0   Carta:    8   Naipe:    2   Cor:      1
Carta:     9  Naipe:    0   Cor:      0   Carta:    9   Naipe:    2   Cor:      1
Carta:    10  Naipe:    0   Cor:      0   Carta:   10   Naipe:    2   Cor:      1
Carta:    11  Naipe:    0   Cor:      0   Carta:   11   Naipe:    2   Cor:      1
Carta:    12  Naipe:    0   Cor:      0   Carta:   12   Naipe:    2   Cor:      1
Carta:     0  Naipe:    1   Cor:      0   Carta:    0   Naipe:    3   Cor:      1
Carta:     1  Naipe:    1   Cor:      0   Carta:    1   Naipe:    3   Cor:      1
Carta:     2  Naipe:    1   Cor:      0   Carta:    2   Naipe:    3   Cor:      1
Carta:     3  Naipe:    1   Cor:      0   Carta:    3   Naipe:    3   Cor:      1
Carta:     4  Naipe:    1   Cor:      0   Carta:    4   Naipe:    3   Cor:      1
Carta:     5  Naipe:    1   Cor:      0   Carta:    5   Naipe:    3   Cor:      1
Carta:     6  Naipe:    1   Cor:      0   Carta:    6   Naipe:    3   Cor:      1
Carta:     7  Naipe:    1   Cor:      0   Carta:    7   Naipe:    3   Cor:      1
Carta:     8  Naipe:    1   Cor:      0   Carta:    8   Naipe:    3   Cor:      1
Carta:     9  Naipe:    1   Cor:      0   Carta:    9   Naipe:    3   Cor:      1
Carta:    10  Naipe:    1   Cor:      0   Carta:   10   Naipe:    3   Cor:      1
Carta:    11  Naipe:    1   Cor:      0   Carta:   11   Naipe:    3   Cor:      1
Carta:    12  Naipe:    1   Cor:      0   Carta:   12   Naipe:    3   Cor:      1
```

Figura 10.17 ■ Saída do programa na Figura 10.16.

Campos de bit

- Um campo de bit não nomeado com largura zero é usado para alinhar o campo de bit seguinte em um novo limite de unidade de armazenamento.
- Por exemplo, a definição da estrutura

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

usa um campo de 0 para saltar os bits restantes (tantos quantos houver) da unidade de armazenamento em que a está armazenado, e alinhar b no limite da unidade de armazenamento seguinte.

Campos de bit

Dica de portabilidade 10.8

As manipulações de campo de bit são dependentes da máquina. Por exemplo, alguns computadores permitem que
os campos de bit atravessem os limites de memória da palavra, enquanto outros não permitem.

Erro comum de programação 10.13

Tentar acessar bits individuais de um campo de bit como se eles fossem elementos de um array consiste em um erro de sintaxe. Os campos de bit não são ‘arrays de bits’.

Campos de bit

Erro comum de programação 10.14
Tentar usar o endereço de um campo de bit (o operador & não deve ser usado com campos de bits, pois não possui endereços).

Dica de desempenho 10.4
Embora os campos de bit economizem espaço, seu uso pode fazer com que o compilador gere um código em linguagem
de máquina de execução mais lenta. Isso ocorre porque são necessárias mais operações em linguagem de máquina para acessar apenas partes de uma unidade de armazenamento endereçável. Esse é um dos muitos exemplos dos tipos de dilemas de espaço-tempo que ocorrem na ciência da computação.

Constantes de enumeração

- Por fim, C oferece mais um tipo definido pelo usuário, chamado de enumeração.
- Uma enumeração, introduzida pela palavra-chave enum, é um conjunto de constantes de enumeração inteiras, representadas por identificadores.
- Os valores em um enum começam com 0, a menos que haja outras especificações, e são incrementados por 1.
- Por exemplo, a enumeração
  - ```
    enum months {
        JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET,
        OUT, NOV, DEZ };
    ```

  cria um novo tipo, enum months, em que os identificadores são definidos como os inteiros de 0 a 11, respectivamente.
    ```

Constantes de enumeração

- Para numerar os meses de 1 a 12, use a seguinte enumeração:
  - ```
    enum months {
        JAN = 1, FEV, MAR, ABR, MAI, JUN, JUL, AGO,
        SET, OUT, NOV, DEZ };
    ```
- Como o primeiro valor nessa enumeração é explicitamente definido como 1, os valores restantes serão incrementados a partir de 1, resultando nos valores de 1 a 12.
- Os identificadores em uma enumeração precisam ser exclusivos.
- O valor de cada constante em uma enumeração pode ser definido explicitamente pela declaração que atribui um valor ao identificador.
    ```

Constantes de enumeração

```
1  /* Figura 10.18: fig10_18.c
2     Usando um tipo de enumeração */
3  #include <stdio.h>
4
5  /* constantes de enumeração representam meses do ano */
6  enum months {
7     JAN = 1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ };
```

Figura 10.18 ■ Uso da enumeração. (Parte 1 de 2.)

Constantes de enumeração

```
 8
 9  int main( void )
10  {
11     enum months month; /* pode conter qualquer um dos 12 meses */
12
13     /* inicializa array de ponteiros */
14     const char *monthName[] = { "", "Janeiro", "Fevereiro", "Março",
15        "Abril", "Maio", "Junho", "Julho", "Agosto", "Setembro", "Outubro",
16        "Novembro", "Dezembro" };
17
18     /* loop pelos meses */
19     for ( month = JAN; month <= DEZ; month++ ) {
20        printf( "%2d%11s\n", month, monthName[ month ] );
21     } /* fim do for */
22
23     return 0; /* indica conclusão bem-sucedida */
24  } /* fim do main */
```

```
 1     Janeiro
 2     Fevereiro
 3     Março
 4     Abril
 5     Maio
 6     Junho
 7     Julho
 8     Agosto
 9     Setembro
10     Outubro
11     Novembro
12     Dezembro
```

Figura 10.18 ■ Uso da enumeração. (Parte 2 de 2.)

Constantes de enumeração

- Vários membros de uma enumeração podem ter o mesmo valor constante.
- No programa da Figura 10.18, a variável de enumeração month é usada em uma estrutura for para imprimir os meses do ano a partir do array monthName.
- Tornamos monthName[0] a string vazia "".
- Alguns programadores poderiam preferir definir monthName[0] como um valor do tipo ***ERROR*** para indicar a ocorrência de um erro lógico.

Constantes de enumeração

Erro comum de programação 10.15
Atribuir um valor a uma constante de enumeração depois que ela tiver sido declarada consiste em um erro de
sintaxe.

Boa prática de programação 10.5
Use apenas letras maiúsculas em nomes de constante de enumeração. Isso faz com que essas constantes se destaquem em um programa e o ajuda a lembrar-se de que as constantes de enumeração não são variáveis.

Lembre-se

" Conhecimento é poder. " Francis Bacon