Estruturas

Módulo 9 Aula 2

Linguagem C, o Curso Definitivo WR Kits

Autor: Dr. Eng. Wagner Rambo

structs

Estruturas são tipos de dados que podem ser definidos pelo usuário onde podemos relacionar variáveis de tipos diferentes referenciadas pelo mesmo nome. É uma boa prática declarar os membros da estrutura de modo que estejam relacionados entre si, como por exemplo, informações para o cadastro de clientes (nome, endereço, telefone, etc). A palavra reservada utilizada para criar uma estrutura em C é a *struct* e apresentamos um exemplo de estrutura no Box 1.

```
struct cliente
{
  char nome[30];
  char endereco[40];
  char cidade[30];
  char telefone[11];
  int cep;
};
```

Box 1 - Exemplo de declaração de estrutura em C.

Organizamos os dados entre chaves e ao final devemos inserir o ponto-e-vírgula, pois a definição de uma estrutura consiste em um comando em C. O nome da estrutura (no caso, cliente) é o seu identificador e você deve escolher um nome intuitivo, para que possamos deduzir facilmente os dados organizados na referida estrutura. Para declarar uma variável que poderá acessar os dados da estrutura, pode-se utilizar a sintaxe do Box 2.

```
struct cliente infos;
```

Box 2 - Declaração de uma variável struct cliente.

Portanto, *infos* será uma variável do tipo *struct cliente*. De fato, as estruturas consistem em tipos complexos de variáveis. Após a declaração, será alocado automaticamente um espaço de memória para armazenar todos os dados da estrutura criada. No caso da estrutura do Box 1, teremos: 30+40+30+11+4 = 115 bytes. No Box 3 apresentamos outro método de declarar uma ou mais variáveis do tipo *struct cliente*.

```
struct cliente
{
  char nome[30];
  char endereco[40];
  char cidade[30];
  char telefone[11];
  int cep;

} infos, dados, cadast;
```

Box 3 - Declarando mais variáveis para a mesma estrutura.

Outro detalhe é que, se apenas uma variável for necessária para a estrutura, a mesma dispensará um identificador. Confira a sintaxe no Box 4.

```
struct
{
  char nome[30];
  char endereco[40];
  char cidade[30];
  char telefone[11];
  int cep;

} infos;
```

Box 4 - Apenas uma variável para a estrutura.

No Box 5 apresentamos a sintaxe geral para a declaração de uma estrutura em C.

```
struct identificador
{
  tipo nome_variavel;
  tipo nome_variavel;
  tipo nome_variavel;
  .
  .
  .

} variaveis_estrutura;
```

Box 5 - Sintaxe para declaração de uma estrutura.

Para acessar um membro da estrutura, utilizamos o operador ponto (.) entre a variável da estrutura e o referido membro. Confira o aspecto geral no Box 6.

```
variavel_estrutura.membro;
```

Box 6 - Acesso a um membro de estrutura com o operador ponto.

Experimente o código do Box 7, que ilustra como carregar valores nos membros da estrutura cliente e imprimir na tela.

```
struct cliente
{
  char nome[30];
  char endereco[40];
  char cidade[30];
  char telefone[11];
  int cep;

} infos;

main()
{
  printf("Nome: ");
  gets(infos.nome);
  printf("Endereco: ");
  gets(infos.endereco);
  printf("Cidade: ");
  gets(infos.cidade);
  printf("Telefone: ");
  gets(infos.telefone);
  printf("CEP: ");
  scanf("%d",&infos.cep);

  printf("Dados do cliente: \n");
  printf(infos.nome);
  putchar('\n');
  printf(infos.endereco);
  putchar('\n');
  printf(infos.cidade);
  putchar('\n');
  printf(infos.telefone);
  putchar('\n');
  printf("%d\n",infos.cep);


} /* end main */
```

Box 7 - Código para teste de estrutura.

Você pode passar os elementos de estruturas para funções, considere o trecho de código do Box 8.

```
struct exemplo
{
  float alpha;
  int   bravo;
  char  whiskey[30];

}charlie;

/* para passer o elemento alpha para a função f_test: */

f_test(charlie.alpha);
```

Box 8 - Passando elemento de estrutura para função.

Além disso, pode-se passar o endereço dos membros para a estrutura, como você pode notar, é igual ao método que utilizamos anteriormente com variáveis convencionais, porém nas estruturas, utilizamos o operador ponto que dá acesso aos membros, veja algumas linhas exemplo no Box 9.

```
func(&charlie.bravo);    /* passa o endereço de bravo */
func2(&charlie.alpha);   /* passa o endereço de alpha */
func3(charlie.whiskey);  /* passa o endereço do primeiro element
                            da string whiskey, que é o próprio
                            nome da string. */
```

Box 9 - Passando endereços e strings de membros de estrutura para funções.

Utilizamos portanto o operador & antes da variável complexa de estrutura. Caso você precisar passar um vetor, matriz ou *string*, não utilize o operador &, pois lembramos que o nome do vetor/matriz/*string* já é uma abstração para o endereço do seu primeiro elemento.

Podemos passar variáveis estruturas para funções também, conforme a sintaxe do Box 10.

```
struct clock
{
  int hours;
  int minutes;

}watch;


void f1(struct clock var);


main()
{
  watch.hours   = 11;
  watch.minutes = 34;

  f1(watch);

}

void f1(struct clock var)
{
  printf("Horas:   %d\n", var.hours);
  printf("Minutos: %d\n", var.minutes);

}
```

Box 10 - Passando variáveis estruturas para funções.

Da mesma forma que utilizamos ponteiros com variáveis convencionais, podemos utilizar ponteiros para estruturas. Vamos entender algumas funcionalidades particulares dos ponteiros nestas condições. O Box 11 ilustra dois métodos de declarar ponteiros para estruturas, assim como você também pode declarar as variáveis para estruturas. Lembrando que os ponteiros devem ser declarados com o operador de indireção *.

```
/* método 1 */

struct protocolos
{
  int spi,
      i2c,
      can;

} *serial;  /* declara ponteiro para a estrutura protocolos */

/* ================== */
/* método 2 */

struct drinks
{
  int margarita,
      dry_martini,
      caipirinha;

};

struct drinks *cheers; /* declara ponteiro para estrutura drinks */
```

Box 11 - Declarando ponteiros para estruturas.

Passar ponteiros de estruturas para funções torna os códigos mais rápidos de processar, uma vez que apenas o endereço para o qual o ponteiro aponta é alocado na pilha, ao invés da estrutura completa. Por exemplo, para utilizar o ponteiro *cheers*, podemos declarar uma variável para estrutura *drinks*, onde o ponteiro apontará para o seu endereço, veja no Box 12.

```
struct drinks
{
  int margarita,
      dry_martini,
      caipirinha;

} open_bar;

struct drinks *cheers;

main()
{
  cheers = &open_bar; /* ponteiro cheers aponta para o endereço
                         da estrutura open_bar */
}
```

Box 12 - Utilizando ponteiro para estrutura.

No exemplo do Box 12, *cheers* aponta para o endereço de *open_bar*. Observe que utilizamos esta declaração dentro da função *main*, mas também poderemos passar ponteiros de estruturas para funções, como veremos mais adiante.

Vamos aproveitar o exemplo do Box 12 para demonstrar como poderemos acessar os elementos da estrutura através do ponteiro. Para isso, podemos utilizar a sintaxe presente no Box 13.

```
main()
{
  cheers = &open_bar;

  (*cheers).margarita = 741;

  printf("%d\n",(*cheers).margarita); /* imprime 741 */

}
```

Box 13 - Acessando elemento de estrutura através de ponteiro para estrutura (método não recomendado).

Apesar do método apresentado no Box 13 funcionar perfeitamente, ele é desencorajado pela maioria dos programadores profissionais, pois pode causar algum tipo de confusão no momento de interpretar o código, ou até mesmo erros de semântica.

Para solucionar este problema, a Linguagem C oferece o operador seta (->), utilizando um sinal de subtração seguido de um sinal de maior, que deve ser usado quando acessamos um elemento da estrutura através de um ponteiro para estrutura. O código se torna mais inteligível e elegante. Confira a sintaxe no Box 14 e procure sempre utilizá-la ao invés da apresentada no Box 13.

```
main()
{
  cheers = &open_bar;

  cheers->margarita = 741;

  printf("%d\n",cheers->margarita); /* imprime 741 */

}
```

Box 14 - Acessando elemento de estrutura através de ponteiro para estrutura (método recomendado).

Ainda tomando como exemplo a estrutura *drinks*, podemos passar ponteiros para estruturas como parâmetros para funções. A função do Box 15 está apta a receber o endereço da variável estrutura e acessar os elementos desta através de ponteiro para estrutura.

```
void barman(struct drinks *p);
```

Box 15 - Protótipo de função que recebe endereço para acesso de ponteiro para estrutura.

O programa do Box 16 mostra como atualizar o conteúdo dos membros da estrutura através de ponteiros para estruturas, passando como parâmetro para função.

```
/* Estrutura */
struct drinks
{
  int margarita,
      dry_martini,
      caipirinha;

}open_bar;

/* Protótipo da Função */
void barman(struct drinks *p);

/* Função Principal */
main()
{

  barman(&open_bar);

  printf("%4d\n",open_bar.margarita);   /* imprime  555 */
  printf("%4d\n",open_bar.dry_martini); /* imprime  741 */
  printf("%4d\n",open_bar.caipirinha);  /* imprime 4093 */


} /* end main */

/* Desenvolvimento barman */
void barman(struct drinks *p)
{
  p->margarita   =  555;
  p->dry_martini =  741;
  p->caipirinha  = 4093;

} /* end barman */
```

Box 16 - Acesso a elementos da estrutura através de ponteiros dentro de funções.

unions

Utilizando a mesma sintaxe das estruturas, você pode declarar uniões na Linguagem C, que basicamente são grupos de variáveis de tipos diferentes que vão compartilhar o mesmo endereço. Confira no Box 17 um exemplo de *union*.

```
union teste
{
  short val1;
  char  val2;

} valor;
```

Box 17 - Declaração de uma union.

No exemplo, os membros val1 e val2 ocuparão o mesmo endereço. No entanto, o compilador é responsável por reservar um espaço de memória igual ao maior tipo de dado presente na união. Para a *union* do Box 17, a memória reservada será de 2 bytes, pois é o tamanho necessário para o tipo *short*. Como as variáveis ocupam o mesmo endereço, a *union* poderá apresentar resultados pouco intuitivos, como ilustra o Box 18.

```
main()
{
  /* exemplo 1*/
  valor.val1 = 100;
  valor.val2 = 'A';

  printf("%hd\n",valor.val1);  /* imprimirá 65  */
  printf("%c\n", valor.val2);  /* imprimirá 'A' */

  /* exemplo 2*/
  valor.val2 = 'A';
  valor.val1 = 100;

  printf("%hd\n",valor.val1);  /* imprimirá 100  */
  printf("%c\n", valor.val2);  /* imprimirá 'd' */

} /* end main */
```

Box 18 - Acessando elementos da union.

No exemplo 1 do Box 18, teremos impresso na tela o 65 (que é o decimal referente ao caractere 'A' pela tabela ASCII) para o formato %hd e após 'A' para o formato %c. O 100 atribuído a val1 é desconsiderado, pois o valor mais atual é o 'A'. Já no exemplo 2, teremos 100 para o formato %hd e 'd' (caractere referente ao 100 decimal na ASCII) para o formato %c, pois o último valor atualizado foi val1; assim val2 será desconsiderada. Isso comprova que as variáveis da união de fato ocupam o mesmo endereço.

As *unions* são especialmente úteis quando você deseja fazer conversão de tipo, separação de bytes de um tipo específico e portabilidade entre máquinas. Um exemplo prático para aplicação de *unions* é no armazenamento de inteiros em arquivos binários separando-os por bytes, sem a necessidade de utilização da função *fwrite*. O programa do Box 19 ilustra esta aplicação, onde temos um código profissional e muito eficiente para o armazenamento de inteiros de 2 bytes em arquivo binário.

```
/* União */
union contents
{
  unsigned short value;
  unsigned char  two_bytes[2];
};

/* Protótipo da Função */
unsigned short rec(union contents *d, FILE *arq);

/* Função Principal */
main()
{
  FILE *arq_dat;

  union contents wr;

  wr.value = 1050; /* exemplo de dado. Gravará 1A04h em rec.dat */

  if(!rec(&wr,arq_dat))
  {
    printf("Erro ao gerar arquivo.\n");
    system("pause");
  }

  printf("Sucesso!\n");

  system("PAUSE");
  return 0;

} /* end main */


/* Desenvolvimento da Função rec */
unsigned short rec(union contents *d, FILE *arq)
{
  if((arq=fopen("rec.dat","wb"))==NULL)
    return 0;

  fputc(d->two_bytes[0],arq);
  fputc(d->two_bytes[1],arq);
  fclose(arq);
  return 1;

} /* end rec */
```

Box 19 - A função rec grava um inteiro separado em 2 bytes no arquivo rec.dat.

Analise cuidadosamente o código do Box 19. Temos uma união com um inteiro de 2 bytes (tipo *short*) e um vetor de 2 bytes (tipo *char*).

enums

Outro recurso bastante aplicável em C são as enumerações. Elas foram acrescentadas pelo padrão ANSI e têm como palavra reservada *enum*. Uma enumeração basicamente é um conjunto de constantes inteiras, onde cada símbolo declarado representará um valor inteiro. Podemos conferir como declarar uma enumeração no Box 20.

```
enum carros
{
  ford,
  honda,
  volks,
  renault

} conc;
```

Box 20 - Declarando uma enumeração.

No caso da enumeração do Box 20, *ford* será 0, *honda* será 1, *volks* será 2, e *renault* será 3, simples assim. A linha do Box 21 imprimirá 0 e 3 na tela.

```
  printf("%d %d\n\n",ford,renault);
```

Box 21 - Imprimirá 0 e 3 na tela.

Você pode utilizar a variável de enumeração para verificar as palavras nela contida, um exemplo com *switch* pode ser visto no Box 22.

```
switch(conc)
{
  case ford:    printf(“ford\n”);  break;
  case honda:   printf(“honda\n”); break;
  case volks:   printf(“volks\n”); break;
  case renault: printf(“renault\n”);

}
```

Box 22 - Teste de conc com switch.

Você poderá descontinuar a sequência de uma enumeração, vamos supor que deseje que volks e renault sejam referentes aos números 300 e 301. Declare a enumeração conforme o Box 23.

```
enum carros
{
  ford,
  honda,
  volks = 300,
  renault

} conc;
```

Box 23 - Descontinuando a enumeração.

typedef

Um método para definir novos tipos na Linguagem C é *typedef*. Você pode utilizar em conjunto com estruturas, uniões, enumerações ou mesmo tipos mais simples do C. O exemplo do Box 24 é bem intuitivo, basicamente definimos um tipo chamado "reais" para declarar variáveis do tipo *float*.

```
typedef  float  reais;

main()
{
  reais val;  /* declara uma variável do tipo float */

  val = 51.3;

  printf("%.1f\n",val); /* imprime 51.3 */

}
```

Box 24 - Utilizando typedef para definir novo tipo de float.

Vale destacar que você não criou um novo tipo de dados, apenas definiu um novo nome para se referir a um tipo já existente. Este processo pode auxiliar na portabilidade entre máquinas pois bastará alterar o tipo em *typedef* para normalizar todas as variáveis existentes em seu código.

Veja um exemplo de uso de *typedef* com estruturas no Box 25.

```
typedef struct pais
{
  char brasil;
  char alemanha;
  int  japao;

} pais;

pais country; /* declara a variável country para estrutura */
```

Box 25 - Utilizando typedef com estruturas.

Pelo fato da aula ter tratado de um assunto muito extenso, não teremos um exercício proposto, no entanto o exercício sugerido é para vocês revisarem todos os conceitos apresentados nesta aula e testarem no compilador.

Como resumo de tudo que aprendemos na aula de hoje, vamos apresentar em detalhes um programa que poderá servir de base para um software de cadastro de clientes, ou outros tipos de cadastro.

Bibliografia:

DAMAS, Luís; Linguagem C, décima edição.

Disponível em: https://amzn.to/3nGdIbN

SCHILDT, Herbert; C Completo e Total, terceira edição.

Disponível em: https://amzn.to/3xxi3l8