Estruturas

As estruturas (*structs*), também chamadas registros (*records*) são variáveis compostas pelo agrupamento de diversas variáveis e vistas/tratadas pelo programa como uma única variável. As variáveis que compõem uma estrutura podem ser simples (int, float, pointer) ou compostas (arrays e outras estruturas).

Estruturas são geralmente utilizadas para agrupar variáveis correlatas que fazem parte de um mesmo conceito lógico. Por exemplo, uma estrutura pode representar um paciente em uma aplicação médica e conter todos os dados relativos a ele ou ela.

Uma estrutura é definida através da palavra reservada "struct", da seguinte forma:

```
  struct <nome da estrutura>
  {
     type variable ;
     type variable ;
     ...
  } ;
```

Uma estrutura define um novo tipo de dados, mas não cria/aloca variáveis desse tipo. Variáveis deste novo tipo devem ser declaradas posteriormente.

A declaração e manipulação de variáveis do novo tipo é simples. Para definir o tipo, use-se a foma acima. Para acessar um componente do registro, emprega-se o operador de seleção "." (ponto). Para a estrutura **x**, **x.a** é o componente **a** daquela estrutura.

```
#include <stdio.h>
#include <string.h>

struct Paciente_t     // o sufixo _t é convenção usual para um novo tipo
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} ;

int main ()
{
   // declaração/alocação
   struct Paciente_t pac1, pac2 ;  // aloca duas variáveis de tipo Paciente_t

   // atribuição de valor aos campos idade e quarto
   pac1.idade = 53 ;
   pac1.quarto = 417 ;

   // pac1.nome é um vetor de char e pode ser usado como parâmetro
   strcpy (pac1.nome, "Homer Simpson") ;

   // na atribuição de structs, o conteúdo é copiado, campo a campo
   pac2 = pac1 ;

   // uso dos valores dos campos
   printf ("Paciente %s, %d anos, quarto %d\n", pac1.nome, pac1.idade, pac1.quarto) ;
   printf ("Paciente %s, %d anos, quarto %d\n", pac2.nome, pac2.idade, pac2.quarto) ;

   return (0) ;
}
```

Definição de tipos

A palavra reservada "typedef" permite definir (ou redefinir) um tipo de dado. Ela pode ser aplicada a qualquer tipo da linguagem C, mas é particularmente útil com *structs*, pois simplifica a declaração de variáveis e parâmetros de tipo *struct*.

Forma geral:

```
  typedef <tipo existente> <novo tipo> ;
```

Exemplo com escalares, para abreviar um nome longo: **unsigned long int** pode ser abreviado para **uint32_t**:

```
typedef unsigned long int uint32_t ; // novo tipo inteiro positivo 32 bits

uint32_t a, b ;  // aloca duas variáveis do tipo uint32_t
```

Exemplo com *structs*:

```
struct Pac_t         // definição do tipo
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} ;

typedef struct Pac_t Paciente_t ;  // struct Pac_t -> Paciente_t


Paciente_t pac1, pac2 ;  // declaração de variáveis e alocação de espaço
```

Ou redefinindo o próprio tipo "Paciente_t" ;

```
struct Paciente_t
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} ;

typedef struct Paciente_t Paciente_t ;

// declaração e alocação
Paciente_t pac1, pac2 ;
```

Ou ainda, de forma mais enxuta:

```
typedef struct Paciente_t
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} Paciente_t ;

// declaração e alocação
Paciente_t pac1, pac2 ;
```

Structs e vetores

O uso de vetores de *structs* permite criar estruturas de dados sofisticadas e com grande poder de expressão. A forma de declaração e acesso é similar ao uso convencional de vetores e *structs* visto até agora:

```
#define VECSIZE 1000
```

```
typedef struct Paciente_t
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} Paciente_t ;

// declaração e alocação
Paciente_t paciente[VECSIZE] ; // vetor com VECSIZE pacientes

// inicializa vetor
for (i=0; i<VECSIZE; i++)
{
   strcpy (paciente[i].nome, "") ;
   paciente[i].idade  = 0 ;
   paciente[i].quarto = 0 ;
}
```

Structs e ponteiros

Uma variável de tipo *struct* é alocada na memória da mesma forma que as demais variáveis, então é possível obter o endereço da variável usando o operador "&" e também criar ponteiros para variáveis *struct*, usando "*":

```
#include <stdio.h>
#include <string.h>

typedef struct Paciente_t
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} Paciente_t ;

int main (void)
{
   // declaração/alocação
   Paciente_t pac, *ptr ;

   // atribuição de valor aos campos
   pac.idade = 53 ;
   pac.quarto = 417 ;
   strcpy (pac.nome, "Homer Simpson") ;

   // atribuição do ponteiro
   ptr = &pac ;

   // acesso pela variável struct
   printf ("Paciente %s, %d anos, quarto %d\n",
           pac.nome, pac.idade, pac.quarto) ;

   // acesso pelo ponteiro
   printf ("Paciente %s, %d anos, quarto %d\n",
           (*ptr).nome, (*ptr).idade, (*ptr).quarto) ;

   return (0) ;
}
```

A especificação de C garante que os campos internos de um *struct* são alocados na ordem em que foram definidos, e na área alocada não há outras informações além dos próprios campos. Assim, o endereço de uma variável de tipo *struct* coincide com o endereço de seu primeiro campo.

O acesso ao campo interno do *struct* feito através do desreferenciamento do ponteiro também pode ser

feito através do operador *ponteiro->campo*. Assim, as operações abaixo são equivalentes:

```
   (*ptr).idade = 45 ;   // (1)
   ptr->idade = 45 ;     // (2)
```

Na primeira atribuição (1), o ponteiro é de-referenciado '(*ptr)' e então o membro 'idade' é acessado para atribuição.

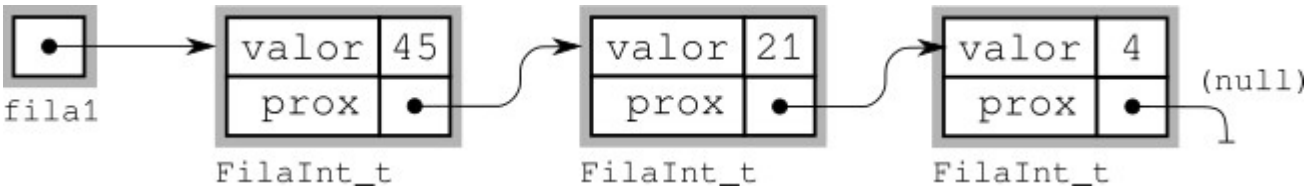
Na segunda (2), o operador "flexa" '->' é usado para acessar o membro da estrutura apontada por 'ptr'.

O uso conjunto de *structs* e ponteiros permite construir estruturas de dados complexas, como filas, pilhas, árvores e grafos. Por exemplo, podemos definir um elemento de uma fila de inteiros da seguinte forma:

```
// definição do tipo FilaInt_t
typedef struct FilaInt_t
{
   int valor ;
   struct FilaInt_t *prox ;
} FilaInt_t ;

// ponteiro para o início de uma fila
FilaInt_t *fila1 ;
```

O ponteiro "prox" dentro do *struct* permite "apontar" para outras variáveis de mesmo tipo, que por sua vez podem apontar para outras variáveis de mesmo tipo e assim sucessivamente. Isso permite criar uma fila de inteiros, com uma estrutura similar à da figura a seguir:



Structs e pontos e flexas

Tentemos clarificar os usos dos dois operadores de seleção de membros, o ponto e a flexa.

Quando o acesso se dá através de um pointer, o operador de seleção é a flexa. Quando o acessso é diretamente numa variável, o operador de seleção é o ponto.

Para acessar através de um pointer, emprega-se a flexa:

```
   ptr_to_struct->membro
```

que é equivalente a

```
   (*ptr_to_struct).membro
```

Os parenteses são necessários por causa da precedência dos operadores. É necessário de-referenciar o apontador, para então acessar o membro com o operador ponto.

Vejamos um exemplo com números complexos.

```
#include <stdio.h>

typedef struct complex {
   double re ;    // parte real
   double im ;    // parte imaginária
} complex ;
```

```
void cplx_add ( complex *a, complex *b, complex *c )  // a = b + c
{
   a -> re = b -> re + c -> re ;
   a -> im = b -> im + c -> im ;
}

int main (void)
{
   complex x, y, z, *p1, *p2, *p3 ;

   // atribuição de valor aos campos
   x.re = 0.0 ;
   x.im = 0.0 ;
   y.re = 2.0 ;
   y.im = 5.0 ;
   z.re = 3.0 ;
   z.im = 1.0 ;

   p1 = &x ;
   p2 = &y ;
   p3 = &z ;

   cplx_add ( p1, p2, p3 ) ;

   // acesso pela variável struct, acessa membro com ponto
   printf ( "x.re = %f  x.im = %f\n\n", x.re, x.im ) ;

   // acesso pelo ponteiro, de-referencia o ponteiro e acessa membro com ponto
   printf ( "(*p).re = %f  (*p).im = %f\n\n", (*p1).re, (*p1).im ) ;

   // acesso pelo ponteiro, acessa membro com flexa
   printf ( "p->re = %f  p->im = %f\n", p1->re, p1->im ) ;

   return (0) ;
}
```

Vejamos mais uns exemplos, a partir da estrutura que descreve pacientes.

```
typedef struct Paciente_t
{
   char  nome[100] ;
   short idade ;
   short quarto ;
} Paciente_t ;

Paciente_t temp, *p;

p = &temp;
temp.idade  = 13 ;
temp.quarto = 307 ;
temp.nome   = "Bart Simpson" ;
```

As seguintes expressões são equivalentes:

```
temp.idade            p->idade                13
temp.nome             p->nome                 Bart Simpson
(*p).quarto           p->quarto               307
* p -> nome + 1       ( *(p -> nome) ) + 1    C     ('C' = 'B'+1)
* (p -> nome + 2)     (p -> nome) [2]         r
```

Os operadores **()**, **[]** **->** e **.** tem a mais alta precedência e associam da esquerda para a direita.

Exercícios

Escreva um programa em C para gerenciar uma relação de alunos, implementada como um vetor de *structs* de tipo "aluno_t". O programa deve ter funções para:

- ler os dados de um aluno (nome, idade, GRR, curso), retornando o *struct* preenchido com os campos lidos;

- imprimir os dados de um aluno a partir de seu *struct*;

- imprimir a relação de alunos;

- imprimir os nomes dos alunos com idade acima de 22 anos;

- ordenar a relação de alunos por idade;

- ordenar a relação de alunos por nome.

Escreva um programa em C para manipular datas e horários:

- *structs* do tipo ''data_t'' armazenam datas (dia, mês, ano);

- *structs* do tipo ''hora_t'' armazenam horários (hora, minuto, segundo);

- *structs* do tipo ''datahora_t'' armazenam datas e horários (seus campos internos são *structs* dos tipos acima);

- a função ''le_data'' lê um *struct* de tipo ''data_t'';

- a função ''le_hora'' lê um *struct* de tipo ''hora_t'';

- a função ''le_datahora'' lê um *struct* de tipo ''datahora_t'';

- as funções ''esc_data'', ''esc_hora'' e ''esc_datahora'' escrevem na tela seus tipos respectivos;

- a função ''data_dias'' retorna (''int'') o número de dias em uma data, a partir do início do calendário (1/1/1); para simplificar, desconsiderar anos bissextos e outros ajustes de calendário;

- a função ''hora_segs'' retorna (''int'') o número de segundos em um horário, a partir do início do dia (00:00:00);

- a função ''dif_data'' retorna (''int'') o número de dias entre duas datas;

- a função ''dif_hora'' retorna (''int'') o número de segundos entre dois horários;

- a função ''dif_datahora'' retorna (''datahora_t'') a diferença entre duas datas e horários.