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, ou seja, 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.

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 dado**, mas não cria/aloca variáveis desse tipo. Variáveis desse tipo devem ser declaradas posteriormente.

A declaração e manipulação de variáveis desse novo tipo é simples:

struct1.c

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

struct paciente_t     // a extensão "_t" é uma convenção usual para um novo tipo
{
  char  nome[100] ;
  short idade ;
  short quarto ;
} ;

int main ()
{
  // declaração/alocação
  struct paciente_t pac1, pac2 ;

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

  // uso como parâmetro
  strcpy (pac1.nome, "Homer Simpson") ;

  // atribuição de structs (o conteúdo é copiado)
  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) ;
}
```

Atribuição de structs

Valores de tipo `struct` podem ser atribuídos, copiados, usados como parâmetros e retornados por funções diretamente, sem necessidade de manipular cada elemento individualmente. Em uma atribuição, por exemplo, todos os bytes contidos na estrutura de origem serão copiados na estrutura de destino.

As seguintes operações com estruturas são perfeitamente válidas (e muito úteis):

struct2.c

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

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

// imprime na tela os dados do paciente
void imprime_paciente (struct paciente_t p)
{
  printf ("Paciente %s, %d anos, quarto %d\n", p.nome, p.idade, p.quarto) ;
}

// devolve um paciente "nulo"
struct paciente_t paciente_nulo ()
{
  struct paciente_t p = {"nulo", 0, 0} ;
  return (p) ;
}

int main ()
{
  // declaração de variável com valor inicial
  struct paciente_t pac1 = { "Home Simpson", 47, 501 } ;

  // cópia de struct
  struct paciente_t pac2 = pac1 ;

  // struct como parâmetro de função (passagem por cópia)
  imprime_paciente (pac2) ;
```

```
  // struct como retorno de função
  pac2 = paciente_nulo () ;
  imprime_paciente (pac2) ;

  return (0) ;
}
```

Structs e vetores

O uso de vetores de *structs* permite criar estruturas de dados sofisticadas, 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

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

// declaração/alocação
struct 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 *:

struct-ptr.c

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

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

int main ()
{
```

```
  // declaração/alocação
  struct 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:

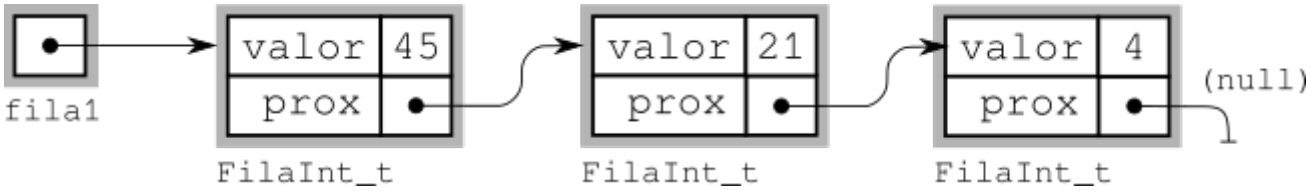
```
  (*ptr).idade = 45 ;
  ptr->idade = 45 ;
```

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:

```
struct filaInt_t
{
  int valor ;
  struct filaInt_t *prox ;
} ;

// ponteiro para o início de uma fila
struct 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:



O código que constrói essa estrutura é o seguinte (simplificado, pois não testa o sucesso do `malloc`):

```
fila1 = malloc (sizeof (struct filaInt_t)) ;

fila1->valor = 45 ;
fila1->prox  = malloc (sizeof (struct filaInt_t)) ;

fila1->prox->valor = 21 ;
fila1->prox->prox  = malloc (sizeof (struct filaInt_t)) ;

fila1->prox->prox->valor = 4 ;
fila1->prox->prox->prox  = NULL ;
```

Exercícios

1. Escrever 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:
    1. ler os dados de um aluno (nome, idade, GRR, curso), devolvendo os campos lidos como um *struct*;
    2. imprimir os dados de um aluno a partir de seu *struct*;
    3. imprimir a relação de alunos;
    4. imprimir os nomes dos alunos com idade acima de 22 anos;
    5. ordenar a relação de alunos por idade;
    6. ordenar a relação de alunos por nome.
2. escreva um programa em C para manipular datas e horários:
    1. *structs* do tipo `data_t` armazenam datas (dia, mês, ano);
    2. *structs* do tipo `hora_t` armazenam horários (hora, minuto, segundo);
    3. *structs* do tipo `datahora_t` armazenam datas e horários (seus campos internos são *structs* dos tipos acima);
    4. a função `le_data` lê um *struct* de tipo `data_t`;
    5. a função `le_hora` lê um *struct* de tipo `hora_t`;
    6. a função `le_datahora` lê um *struct* de tipo `datahora_t`;
    7. as funções `esc_data`, `esc_hora` e `esc_datahora` escrevem na tela seus tipos respectivos;
    8. 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;
    9. a função `hora_segs` retorna (`int`) o número de segundos em um horário, a partir do início do dia (00:00:00);
    10. a função `dif_data` retorna (`int`) o número de dias entre duas datas;
    11. a função `dif_hora` retorna (`int`) o número de segundos entre dois horários;
    12. a função `dif_datahora` retorna (`datahora_t`) a diferença entre duas datas e horários.