ESTRUTURAS DE DADOS COMPOSTAS HETEREGÊNEAS

ESTRUTURAS

Uma estrutura (`struct`, abreviatura de `structure`) é uma coleção de várias variáveis, possivelmente de tipos diferentes. Dessa forma, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

Para criar uma estrutura utiliza-se o comando `struct`. Sua forma geral é:

```
struct nome_do_tipo_da_estrutura {
  tipo_1 nome_1;
  tipo_2 nome_2;
  //...
  tipo_n nome_n;
} variáveis_estrutura;
```

O `nome_do_tipo_da_estrutura` é o nome para a estrutura. As `variáveis_estrutura` são opcionais e, são nomes de variáveis que o usuário declara e que são do tipo `nome_do_tipo_da_estrutura`.

Exemplo: Declaração de `struct`

```
struct tipo_endereco {
  char rua [50];
  int numero;
  char bairro [20];
  char cidade [30];
  char sigla_estado [3];
  long int CEP;
};

struct ficha_pessoal {
  char nome [50];
  long int telefone;
  struct tipo_endereco endereco;
} var1, var2;

struct ficha_pessoal var3, var4;
```

No exemplo acima o `struct ficha_pessoal` funciona como tipo. Observe as formas de declaração de variáveis. As variáveis **`var1`** e **`var2`** são declaradas no momento da declaração da estrutura e são **`var3`** e **`var4`** declaradas posteriormente.

Para simplificar a declaração das variáveis muitas vezes usamos uma definição de tipo.

DEFINIÇÕES DE TIPOS

Sintaxe: `typedef <type-definition> <identifier>;`

Associa um nome novo de tipo `<identifier>` a um tipo `<type-definition>`.

Exemplo:

```
typedef int INTEGER;  /* criação do tipo INTEGER */
INTEGER var2;              /* utilização do tipo */
```

Exemplo: Declaração de tipos a partir de `struct`

1)

```
struct num {
  int i;
  float f;
} x, y;

typedef struct num num;

num z, w;
```

ou

```
struct {
  int i;
  float f;
} x, y;

typedef struct {
  int i;
  float f;
} num;

num z, w;
```

2)

```
typedef struct {
  int dia, mes, ano;
} tipoData;

tipoData aniversario;
```

Exemplo: Utilização de `struct`

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

struct tipo_endereco{
  char rua [50];
  int numero;
  char bairro [20];
  char cidade [30];
  char sigla_estado [3];
  long int CEP;
};
```

```
struct ficha_pessoal {
  char nome [50];
  long int telefone;
  struct tipo_endereco endereco;
};

typedef struct ficha_pessoal fp;

int main ()
{
  fp ficha;
  strcpy (ficha.nome,"LEPEC Unesp");
  ficha.telefone = 31036080;
  strcpy (ficha.endereco.rua,"Av. Luis Edmundo Carrijo Coube");
  ficha.endereco.numero = 1401;
  strcpy (ficha.endereco.bairro,"Vargem Limpa");
  strcpy (ficha.endereco.cidade,"Bauru");
  strcpy (ficha.endereco.sigla_estado,"SP");
  ficha.endereco.CEP = 17033360;
  return 0;
}
```

Pode-se atribuir duas estruturas que sejam do mesmo tipo. Neste caso, será feita uma cópia da estrutura, campo por campo, na outra.

Exemplo:

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

struct num {
  int i;
  float f;
};

int main()
{
  struct num n1, n2;
  scanf ("%d",&n1.i);
  scanf ("%f",&n1.f);
  n2 = n1;
  printf("Valores da segunda struct: %d e %f",n2.i ,n2.f);
  return 0;
}
```

Pode-se passar para uma função uma estrutura inteira.

Exemplo:

```
void PreencheFicha (struct ficha_pessoal ficha)
{
  //...
}
```

É fácil passar a estrutura como um todo para a função. A passagem da estrutura é feita por valor. A estrutura que está sendo passada, vai ser copiada, campo por campo, em uma variável local da função `PreencheFicha`. Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função.

Exercícios:

1 Escreva uma função que receba um número inteiro que representa um intervalo de tempo medido em minutos e devolva o correspondente número de horas e minutos (por exemplo, converte 131 minutos em 2 horas e 11 minutos). Use uma `struct` como a seguinte:

```
struct hm {
   int horas;
   int minutos;
};
```

2 Defina um registro empregado para guardar os dados (nome, sobrenome, data de nascimento, RG, data de admissão, salário) de um empregado de sua empresa. Defina um vetor de empregados para armazenar todos os empregados de sua empresa.

3 Um racional é qualquer número da forma p/q, sendo p inteiro e q inteiro não nulo. É conveniente representar um racional por um registro:

```
typedef struct {
   int p, q;
} racional;
```

Suponha que o campo q de todo racional é estritamente positivo e que o máximo divisor comum dos campos p e q é 1. Escreva:

- uma função `reduz` que receba inteiros a e b e devolva o racional que representa a/b;
- uma função `neg` que receba um racional x e devolva o racional −x;
- uma função `soma` que receba racionais x e y e devolva o racional que representa a soma de x e y;
- uma função `mult` que receba racionais x e y e devolva o racional que representa o produto de x por y;
- uma função `div` que receba racionais x e y e devolva o racional que representa o quociente de x por y;

UNIONS

Uma declaração `union` determina uma única localização de memória onde podem estar armazenadas várias variáveis diferentes.

A declaração de uma união é semelhante à declaração de uma estrutura:

```
union nome_do_tipo_da_union {
  tipo_1 nome_1;
  tipo_2 nome_2;
  //...
  tipo_n nome_n;
} variáveis_union;
```

Exemplos:

1)

```
union angulo
{
  float graus;
  float radianos;
};
```

Na `union` acima há duas variáveis (`graus` e `radianos`) que, apesar de terem nomes diferentes, ocupam o mesmo local da memória. Isto quer dizer que só gastamos o espaço equivalente a um único `float`.

2)

```
struct Ponto {
  int polar;  //  booleano
  union {
    struct {
      float x, y, z;
    };
    struct {
      float r, theta, phi;
    };
  };
};
```

O grupo de campos `x, y, z` compartilha a mesma área de memória do grupo `r, theta, phi`, assim, como `x` e `r` compartilham a mesma área de memória, se o valor de `x` for alterado o valor de `r` também será alterado. O mesmo acontece com `y` e `theta`, `z` e `phi`.

Uniões podem ser feitas também com variáveis de diferentes tipos. Neste caso, a memória alocada corresponde ao tamanho da maior variável no `union`.

Exemplo:

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

#define GRAUS 'G'
#define RAD 'R'

union angulo {
  int graus;
  float radianos;
};
```

```
int main()
{
  union angulo ang;
  char op;
  printf("\nNumeros em graus ou radianos (G/R)?: ");
  scanf("%c",&op);
  if (toupper(op) == GRAUS){
    ang.graus = 180;
    printf("\nAngulo: %d\n",ang.graus);
  }
  else
    if (toupper(op) == RAD){
      ang.radianos = 3.1415;
      printf("\nAngulo: %f\n",ang.radianos);
    }
    else printf("\nEntrada invalida!!\n");
}
```

ENUMERAÇÕES

Um tipo enumerado é uma definição de sequência de identificadores relacionados a constantes inteiras.

```
enum [<enum-type-name>]:[type] {
    <constant-names> [ = value]
    ...
} [<enum-variables>] ;
```

Por exemplo, para definirmos constantes para as estações do ano, podemos:

Usando `define`	Usando `enum`

`#define PRIMAVERA 0`
`#define VERAO 1`
`#define OUTONO 2`
`#define INVERNO 3`
	`enum Estacoes {PRIMAVERA, VERAO, OUTONO, INVERNO};`

É possível ainda explicitar os valores de cada constante:

```
enum Altura {PEQUENO = 150, MEDIO = 175, GRANDE = 200};
enum Cor {VERMELHO=100, AZUL, VERDE}; // AZUL = 101, VERDE = 102
```

Declaração de uma variável do tipo enumerado:

```
enum Altura alt;
enum Cor c;
```

Exemplo:

```
#include <stdio.h>
#define AMARELO 2
enum Cor {VERMELHO = 100, AZUL=105, VERDE=110};
```

```
int main(){
    enum Cor c = AZUL;
    printf ("%d\n",c);
    if (++c == VERDE)
      printf ("VERDE CORRETO");
    if (c == AMARELO)
      printf ("AMARELO CORRETO");
    c = 100;
    if (c > 90)
      printf ("90 CORRETO");
}
```