Prof. MSc. Raphael Gomes (raphael@ifg.edu.br)

Variáveis Compostas Heterogêneas – Structs em C

Um registro (= *record*) é uma coleção de várias variáveis, possivelmente de tipos diferentes. Na linguagem C, registros são conhecidos como ***structs*** (abreviatura de *structures*).

1.1 – Definição e manipulação de structs

O exemplo abaixo declara um registro x com três campos (ou membros) inteiros:

```
struct {
     int dia;
     int mes;
     int ano;
  };
```

É uma boa ideia dar um nome ao tipo de registro. No nosso exemplo, ***data*** parece um nome apropriado:

```
struct data {
     int dia;
     int mes;
     int ano;
  };
  struct data x;  /* um registro x do tipo dma */
  struct data y;  /* um registro y do tipo dma */
```

É fácil atribuir valores aos campos de um registro:

```
 x.dia = 31;
 x.mes = 8;
 x.ano = 1998;
```

Exemplo:
A função abaixo recebe a data de início de um evento e a duração do evento em dias. Ela devolve a data de fim do evento.

```
struct data fim_evento (struct data datainicio, int duracao) {
   struct data datafim;
   . . .
   . . .
   datafim.dia = ...
   datafim.mes = ...
   datafim.ano = ...
   return datafim;
}
```

O código foi omitido porque é um tanto enfadonho: deve levar em conta a existência de meses com 31 dias, de meses com 30 dias, com 29 dias etc.
Eis como essa função poderia ser usada:

```
int main (void) {
   struct data a, b;
   int d;
   scanf ("%d %d %d", &a.dia, &a.mes, &a.ano);
   scanf ("%d", &d);
   b = fim_evento (a, d);
   printf ("%d %d %d\n", b.dia, b.mes, b.ano);
   return EXIT_SUCCESS;
}
```

1.2 – Structs e ponteiros

Cada registro tem um endereço na memória do computador. (Você pode imaginar que o endereço de um registro é o endereço de seu primeiro campo, mas essa detalhe é irrelevante.) É muito comum usar um ponteiro para guardar o endereço de um registro. Dizemos que um tal ponteiro aponta para o registro. Por exemplo,

```
struct data *p;   /* p é um ponteiro para registros dma */
struct data  x;
p = &x;           /* agora p aponta para x */
(*p).dia = 31;    /* mesmo efeito que x.dia = 31 */
```

[Cuidado! A expressão `*p.dia`, que equivale a `*(p.dia)`, tem significado muito diferente de `(*p).dia`.] A expressão `p->mes` é uma abreviatura muito útil para a expressão `(*p).mes`:

```
p->mes = 8;       /* mesmo efeito que (*p).mes = 8 */
p->ano = 1998;
```

Registros podem ser tratados como um novo tipo-de-dados. Por exemplo,

```
typedef struct data dma;
dma  x;
dma *p;
p = &x;
```

1.3 – Mais sobre typedef

Em C e C++ podemos redefinir um tipo de dado dando-lhe um novo nome.
Essa forma de programação ajuda em dois sentidos: 1º. Fica mais simples entender para que serve tal tipo de dado; 2º. É a única forma de conseguirmos referenciar uma estrutura de dados dentro de outra (struct dentro de struct).
Para redefinirmos o nome de um tipo de dado usamos o comando `typedef`, que provém de *type definition*.
`Typedef` faz o compilador assumir que o novo nome é um certo tipo de dado, e então, passamos a usar o novo nome da mesma forma que usaríamos o antigo.
Por exemplo, podemos definir que, ao invés de usarmos `int`, agora usaremos `inteiro` ou, ao invés de usarmos `float`, usaremos `decimal`.
`Typedef` deve sempre vir antes de qualquer programação que envolva procedimentos (protótipo de funções, funções, função main, structs, etc.) e sua sintase base é:

```
typedef nome_antigo nome_novo;
```

Dessa forma, simplesmente, definiríamos o exemplo acima como:

```
typedef int inteiro;
typedef float decimal;
```

Se fossemos utilizar essas novas definições em um programa ficaria assim:

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

```
typedef int inteiro;
typedef float decimal;

int main (){
      inteiro x = 1;
      decimal y = 1.5;
      printf("X=%d \n Y=%f \n");

      system ("pause");
      return EXIT_SUCCESS;
}
```

Nota importante: O uso de `typedef` para redefinir nomes de tipos primitivos (como `int`, `float`, `char`) é altamente desencorajado por proporcionar uma menor legibilidade do código. Portanto, devemos utilizar `typedef` apenas em momentos oportunos (como por exemplo, definir o nome de uma estrutura de dados complexa - struct).

DEFININDO NOMES PARA ESTRUTURAS DE DADOS

Uma vantagem muito grande que `typedef` nos proporciona é definir um nome para nossa estrutura de dados (struct).
Graças a isso, somos capazes de auto-referenciar a estrutura, ou seja, colocar um tipo de dado struct dentro de outro struct.
Podemos definir o nome de uma estrutura de dados (struct) de duas maneiras: Definindo o nome da estrutura e só depois definir a estrutura; ou definir a estrutura ao mesmo tempo que define o nome.
Da primeira forma seria o mesmo que fizéssemos isso:

```
typedef struct estrutura1 MinhaEstrutura;
struct estrutura1 {
      int var1;
      float var2;
};
```

Da segunda forma seria o mesmo que fizéssemos isso:

```
typedef struct estrutura1 {
      int var1;
      float var2;
} MinhaEstrutura;
```

Agora, que já possuímos uma estrutura de dados definida com um nome, nós podemos utilizá-la dentro de outra estrutura de dados. Por exemplo:

```
typedef struct estrutura1 MinhaEstrutura;
struct estrutura1 {
      int var1;
      float var2;
};

struct estrutura2 {
      int var3;
      MinhaEstrutura var4;
};
```

Como podemos perceber, somos capazes de usar `MinhaEstrutura` (que na verdade é o `struct estrutura1`) dentro do `struct estrutura2` perfeitamente, sem maiores problemas. O que seria impossível de ser feito se declarássemos dentro de `estrutura2` o `var4` como sendo `estrutura1`.

Referências

- Paulo Feofiloff, Programação em C. Disponível em: http://www.ime.usp.br/~pf/algoritmos/
- http://www.tiexpert.net/programacao/c/typedef.php