AULA TEÓRICA+PRÁTICA 16

FUNÇÕES E REGISTROS

Já vimos funções com parâmetros de tipos básicos e de tipos complexos, como vetores e matrizes. Nesta aula, aprenderemos como comportam-se os registros no contexto das funções. Como veremos, um registro, diferentemente das variáveis compostas homogêneas, comporta-se como uma variável de um tipo básico qualquer quando trabalhamos com funções. Ou seja, um registro comporta-se como qualquer outra variável excluindo as compostas homogêneas, isto é, um registro é sempre um argumento passado por cópia a uma função. Se quisermos que um registro seja um argumento passado por referência a uma função, devemos explicitamente indicar essa opção usando o símbolo `&` no argumento e o símbolo `*` no parâmetro correspondente, como fazemos com variáveis de tipos primitivos.

Esta aula é baseada nas referências [9, 6].

16.1 Tipo registro

Antes de estudar diretamente o funcionamento das funções com registros, precisamos aprender algo mais sobre esses últimos. Primeiro, é importante notar que apesar das aulas anteriores mostrarem como declarar variáveis que são registros, não discutimos em detalhes o que é um tipo registro. Então, suponha que um programa precisa declarar diversas variáveis que são registros com mesmos campos. Se todas as variáveis podem ser declaradas ao mesmo tempo, não há problemas. Mas se for necessário declarar essas variáveis em diferentes pontos do programa, então temos uma situação mais complicada. Para superá-la, precisamos saber definir um nome que representa um *tipo* registro, não uma variável registro particular. A linguagem C fornece duas boas maneiras para realizar essa tarefa: declarar uma etiqueta de registro ou usar `typedef` para criar um tipo registro.

Uma **etiqueta de registro** é um nome usado para identificar um tipo particular de registro. Essa etiqueta não reserva compartimentos na memória para armazenamento dos campos e do registro, mas sim adiciona informações a uma tabela usada pelo compilador para que variáveis registros possam ser declaradas a partir dessa etiqueta. A seguir temos a declaração de uma etiqueta de registro com nome `cadastro`:

```
struct cadastro {
   int codigo;
   char nome[MAX+1];
   int fone;
};
```

Observe que o caractere ponto e vírgula (`;`) segue o caractere fecha chaves `}`. O `;` deve estar presente neste ponto para terminar a declaração da etiqueta do registro.

Uma vez que criamos a etiqueta `cadastro`, podemos usá-la para declarar variáveis:

```
struct cadastro ficha1, ficha2;
```

Infelizmente, não podemos abreviar a declaração acima removendo a palavra reservada `struct`, já que `cadastro` não é o nome de um tipo. Isso significa que sem a palavra reservada `struct` a declaração acima não tem sentido.

Também, a declaração de uma etiqueta de um registro pode ser combinada com a declaração de variáveis registros, como mostrado a seguir:

```
struct cadastro {
   int codigo;
   char nome[MAX+1];
   int fone;
} ficha1, ficha2;
```

Ou seja, acima temos a declaração da etiqueta de registro `cadastro` e a declaração de duas variáveis registros `ficha1` e `ficha2`. Todas as variáveis registros declaradas com o tipo `struct cadastro` são compatíveis entre si e, portanto, podemos atribuir uma a outra sem problemas, como mostramos abaixo:

```
struct cadastro ficha1 = {10032, "Sir Lancelot", 77165115};
struct cadastro ficha2;
ficha2 = ficha1;
```

Uma alternativa para declaração de etiquetas de registros é o uso da palavra reservada `typedef` para definir o nome de um tipo genuíno. Por exemplo, poderíamos definir um tipo com nome `Cadastro` da seguinte forma:

```
typedef struct {
   int codigo;
   char nome[MAX+1];
   int fone;
} Cadastro;
```

Observe que o nome do tipo, `Cadastro`, deve ocorrer no final, não após a palavra reservada `struct`. Além disso, como `Cadastro` é um nome de um tipo definido por `typedef`, não é permitido escrever `struct Cadastro`. Todas as variáveis do tipo `Cadastro` são compatíveis, desconsiderando onde foram declaradas. Assim, a declaração de registros com essa definição pode ser dada como abaixo:

```
Cadastro ficha1, ficha2;
```

Quando precisamos declarar um nome para um registro, podemos escolher entre declarar uma etiqueta de registro ou definir um tipo registro com a palavra reservada `typedef`. Entretanto, em geral preferimos usar etiquetas de registros por diversos motivos que serão justificados posteriormente.

16.2 Registros e passagem por cópia

O valor de um campo de um registro pode ser passado como argumento na chamada de uma função assim como fizemos com um valor associado a uma variável de um tipo básico ou a uma célula de um vetor ou matriz. Ou seja, a sentença

```
x = raizQuadrada(poligono.diagonal);
```

chama a função `raizQuadrada` com argumento `poligono.diagonal`, isto é, com o valor armazenado no campo `diagonal` do registro `poligono`.

Um registro todo pode ser argumento na chamada de funções da mesma forma que um vetor ou uma matriz podem ser, bastando listar o identificador do registro. Por exemplo, se temos um registro `tempo`, contendo os campos `hora`, `minutos` e `segundos`, então a sentença a seguir:

```
s = totalSegundos(tempo);
```

pode ser usada para chamar a função `totalSegundos` que provavelmente calcula o total de segundos de uma medida de tempo armazenada no registro `tempo`.

Diferentemente das variáveis compostas homogêneas, uma modificação realizada em qualquer campo de um registro que é um parâmetro de uma função não provoca uma modificação no conteúdo do campo correspondente que é um argumento da função. Nesse caso, a princípio, um argumento que é um registro em uma chamada de uma função tem o mesmo comportamento de um argumento que é uma variável de um tipo básico qualquer, ou seja, uma expressão que é um argumento de uma chamada de uma função e que, na verdade é ou contém um registro, é de fato um parâmetro de entrada da função, passado por cópia. Dessa forma, voltamos a repetir que somente as variáveis compostas homogêneas têm um comportamento deferenciado na linguagem C, no que ser refere a argumentos e parâmetros de funções neste caso específico.

Um exemplo com uma chamada de uma função contendo um registro como argumento é mostrado no programa 16.1.

Programa 16.1: Um programa com uma função com parâmetro do tipo registro.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss, converte essa medida em segundos e devolve esse valor */
int converteSegundos(struct marca tempo)
{
   return tempo.hh * 3600 + tempo.mm * 60 + tempo.ss;
}

/* Recebe uma medida de tempo no formato hh:mm:ss e mostra a quantida-
   de de segundos relativos à esse tempo com referência ao mesmo dia */
int main(void)
{
   int segundos;
   struct marca horario;

   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);

   segundos = converteSegundos(horario);

   printf("Passaram-se %d segundo(s) neste dia\n", segundos);

   return 0;
}
```

Devemos destacar pontos importantes neste programa. O primeiro, e talvez o mais importante, é a definição da etiqueta de registro `marca`. Em nenhuma outra oportunidade anterior havíamos desenvolvido um programa que continha uma declaração de uma etiqueta de registro. Observe ainda que a declaração da etiqueta `marca` do programa 16.1 foi feita *fora* dos corpos das funções `converteSegundos` e `main`. Em todos os programas descritos até aqui, não temos um exemplo de uma declaração de uma etiqueta de registro, ou mesmo uma declaração de uma variável de qualquer tipo, fora do corpo de qualquer função. A declaração da etiqueta `marca` realizada dessa maneira se deve ao fato que a mesma é usada nas funções `converteSegundos` e `main`, obrigando que essa declaração seja realizada *globalmente*, ou seja, fora do corpo de qualquer função. Todas as outras declarações que fizemos são declarações *locais*, isto é, estão localizadas no interior de alguma função.

Os outros pontos de destaque do programa 16.1 são todos relativos ao argumento da chamada da função `converteSegundos` e do parâmetro da mesma, descrito em sua interface. Note que essa função tem como parâmetro o registro `tempo` do tipo `struct marca`. O argumento da função, na chamada dentro da função `main` é o registro `horario` de mesmo tipo `struct marca`.

Finalmente, é importante repetir que a passagem de um registro para uma função é feita por cópia, ou seja, o parâmetro correspondente é um parâmetro de entrada e, por isso, qualquer alteração realizada em um ou mais campos do registro dentro da função não afeta o conteúdo dos campos do registro que é um argumento na sua chamada. No exemplo acima não há alterações nos campos do registro `tempo` no corpo da função `converteSegundos`, mas, caso houvesse, isso não afetaria qualquer um dos campos do registro `horario` da função `main`.

16.3 Registros e passagem por referência

Se precisamos de uma função com um parâmetro de entrada e saída, isto é, um parâmetro passado por referência, devemos fazer como usualmente fazemos com qualquer variável de um tipo básico e usar os símbolos usuais `&` e `*`. O tratamento dos campos do registro em questão tem uma peculiaridade que precisamos observar com cuidado e se refere ao uso do símbolo `*` no parâmetro passado por referência no corpo da função. Veja o programa 16.2.

Programa 16.2: Função com um registro como parâmetro de entrada e saída.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss e devolve o tempo neste registro atualizado em 1 segundo */
void adicionaSegundo(struct marca *tempo)
{
   (*tempo).ss++;
   if ((*tempo).ss == 60) {
      (*tempo).ss = 0;
      (*tempo).mm++;
      if ((*tempo).mm == 60) {
         (*tempo).mm = 0;
         (*tempo).hh++;
         if ((*tempo).hh == 24)
            (*tempo).hh = 0;
      }
   }
}

/* Recebe uma medida de tempo (hh:mm:ss) e mostra o tempo atualizado em 1s */
int main(void)
{
   struct marca horario;
   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
   adicionaSegundo(&horario);
   printf("Novo: %02d:%02d:%02d\n", horario.hh, horario.mm, horario.ss);
   return 0;
}
```

Uma observação importante sobre o programa 16.2 é a forma como um registro que é um parâmetro passado por referência é apresentado no corpo da função. Como o símbolo `.` que seleciona um campo de um registro tem prioridade sobre o símbolo `*`, que indica que o parâmetro foi passado por referência, os parênteses envolvendo o o símbolo `*` e o identificador do registro são essenciais para evitar erros. Dessa forma, uma expressão envolvendo, por exemplo, o campo `ss` do registro `tempo`, escrita como `*tempo.ss`, está incorreta e não realiza a seleção do campo do registro que tencionamos.

Alternativamente, podemos usar os símbolos `->` para indicar o acesso a um campo de um registro que foi passado por referência a uma função. O uso desse símbolo simplifica bastante a escrita do corpo das funções que têm parâmetros formais que são registros passados por referência. Dessa forma, o programa 16.2 pode ser reescrito como o programa 16.3. Observe que esses dois programas têm a mesma finalidade, isto é, dada uma entrada, realizam o mesmo processamento e devolvem as mesmas saídas.

Programa 16.3: Uso do símbolo `->` para registros passados por referência.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss e devolve o tempo neste registro atualizado em 1 segundo */
void adicionaSegundo(struct marca *tempo)
{
   tempo->ss++;
   if (tempo->ss == 60) {
      tempo->ss = 0;
      tempo->mm++;
      if (tempo->mm == 60) {
         tempo->mm = 0;
         tempo->hh++;
         if (tempo->hh == 24)
            tempo->hh = 0;
      }
   }
}

/* Recebe uma medida de tempo no formato hh:mm:ss
   e mostra esse tempo atualizado em 1 segundo */
int main(void)
{
   struct marca horario;

   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
   adicionaSegundo(&horario);
   printf("Novo horário %02d:%02d:%02d\n", horario.hh, horario.mm, horario.ss);
   return 0;
}
```

Observe que o argumento passado por referência para a função `adicionaSegundo` é idêntico nos programas 16.2 e 16.3. Mais que isso, a interface da função `adicionaSegundo` é idêntica nos dois casos e o parâmetro é declarado como `struct marca *tempo`. No corpo desta função do programa 16.3 é que usamos o símbolo `->` como uma abreviação, ou uma simplificação, da notação usada no corpo da função `adicionaSegundo` do programa 16.2. Assim, por exemplo, a expressão `(*tempo).ss` no programa 16.2 é reescrita como `tempo->ss` no programa 16.3.

16.4 Funções que devolvem registros

Uma função pode devolver valores de tipos básicos, mas também pode devolver um valor de um tipo complexo como um registro. Observe que variáveis compostas homogêneas, pelo fato de sempre poderem ser argumentos que correspondem a parâmetros de entrada e saída, isto é, de sempre poderem ser passadas por referência, nunca são devolvidas explicitamente por uma função, através do comando `return`. O único valor de um tipo complexo que podemos devolver é um valor do tipo registro. Fazemos isso muitas vezes quando prorgramamos para solucionar grandes problemas.

No problema da seção anterior, em que é dado um horário e queremos atualizá-lo em um segundo, usamos um registro como parâmetro de entrada e saída na função `adicionaSegundo`. Por outro lado, podemos fazer com que a função apenas receba uma variável do tipo registro contendo um horário como parâmetro de entrada e devolva um registro com o horário atualizado como saída. Ou seja, teremos então um parâmetro de entrada e um parâmetro de saída que são registros distintos. Dessa forma, os programas 16.2 e 16.3 poderiam ser reescritos como no programa 16.4.

Observe que a interface da função `adicionaSegundo` é bastante diferente das definições anteriores no programa 16.4. O tipo do valor devolvido é o tipo `struct marca` – uma etiqueta de um registro –, indicando que a função devolverá um registro. O corpo dessa função também é diferente das funções anteriores e há a declaração de uma variável `atualizado` do tipo `struct marca` que ao final conterá o horário do registro `tempo`, parâmetro de entrada da função, atualizado em um segundo. Na função principal, a chamada da função `adicionaSegundo` também é diferente, já que é descrita do lado direito de uma instrução de atribuição, indicando que o registro devolvido pela função será armazenado no registro apresentado do lado esquerdo da atribuição: `novo = adicionaSegundo(agora);`.

Programa 16.4: Um programa com uma função que devolve um registro.

```
#include <stdio.h>

struct marca }
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss e devolve um registro com tempo atualizado em 1 segundo */
struct marca adicionaSegundo(struct marca tempo)
{
   struct marca atualizado;

   atualizado.ss = tempo.ss + 1;
   if (atualizado.ss == 60) {
      atualizado.ss = 0;
      atualizado.mm = tempo.mm + 1;
      if (atualizado.mm == 60) {
         atualizado.mm = 0;
         atualizado.hh = tempo.hh + 1;
         if (atualizado.hh == 24)
            atualizado.hh = 0;
      }
      else
         atualizado.hh = tempo.hh;
   }
   else {
      atualizado.mm = tempo.mm;
      atualizado.hh = tempo.hh;
   }

   return atualizado;
}

/* Recebe uma medida de tempo no formato hh:mm:ss
   e mostra esse tempo atualizado em 1 segundo */
int main(void)
{
   struct marca agora, novo;

   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &agora.hh, &agora.mm, &agora.ss);
   novo = adicionaSegundo(agora);
   printf("Novo horário %02d:%02d:%02d\n", novo.hh, novo.mm, novo.ss);

   return 0;
}
```

Exercícios

16.1 (a) Escreva uma função com a seguinte interface:

```
int bissexto(struct data d)
```

que receba uma data no formato **dd/mm/aaaa** e verifique se o ano é bissexto, devolvendo 1 em caso positivo e 0 caso contrário. Um ano é bissexto se é divisível por 4 e não por 100 ou é divisível por 400.

(b) Escreva uma função com a seguinte interface:

```
int diasMes(struct data d)
```

que receba uma data e devolva o número de dias do mês em questão.
Exemplo:

Se a função recebe a data `10/04/1992` deve devolver `30` e se recebe a data `20/02/2004` deve devolver `29`.

(c) Escreva uma função com a seguinte interface:

```
struct data diaSeguinte(struct data d)
```

que receba uma data no formato **dd/mm/aaaa** e devolva a data que representa o dia seguinte. Use as funções dos itens (a) e (b).

(d) Escreva um programa que receba uma data no formato **dd/mm/aaaa** e imprima a data que representa o dia seguinte. Use as funções dos itens anteriores.

16.2 Reescreva a função do item (c) do exercício 16.1, fazendo com que a função `diaSeguinte` tenha o registro `d` como um parâmetro de entrada e saída. Isto é, a interface da função deve ser:

```
void diaSeguinte(struct data *d)
```

16.3 (a) Escreva uma função com a seguinte interface:

```
struct tempo tempoDecorr(struct tempo t1, struct tempo t2)
```

que receba duas medidas de tempo no formato `hh:mm:ss` e calcule o tempo decorrido entre essas duas medidas de tempo. Tenha cuidado com um par de medidas de tempo que cruzam a meia noite.

(b) Escreva um programa que receba dois horários no formato **hh:mm:ss** e calcule o tempo decorrido entre esses dois horários. Use a função anterior.