AULA 15

PONTEIROS E REGISTROS

Nesta aula trabalharemos com ponteiros e registros. Primeiro, veremos como declarar e usar ponteiros para registros. Essas tarefas são equivalentes as que já fizemos quando usamos ponteiros para números inteiros, por exemplo. Além disso, vamos adicionar também ponteiros como campos de registros. É muito comum usar registros contendo ponteiros em estruturas de dados poderosas, como listas lineares e árvores, para solução de problemas. Esta aula é baseada nas referências [2, 7].

15.1 Ponteiros para registros

Suponha que definimos uma etiqueta de registro `data` como a seguir:

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

A partir dessa definição, podemos declarar variáveis do tipo `struct data`, como abaixo:

```
struct data hoje;
```

E então, assim como fizemos com ponteiros para inteiros, caracteres e números de ponto flutuante, podemos declarar um ponteiro para o registro `data` da seguinte forma:

```
struct data *p;
```

Podemos, a partir dessa declaração, fazer uma atribuição à variável p como a seguir:

```
p = &hoje;
```

Além disso, podemos atribuir valores aos campos do registro de forma indireta, como fazemos abaixo:

```
(*p).dia = 11;
```

Essa atribuição tem o efeito de armazenar o número inteiro 11 no campo `dia` da variável `hoje`, indiretamente através do ponteiro p no entanto. Nessa atribuição, os parênteses envolvendo `*p` são necessários porque o operador `.`, de seleção de campo de um registro, tem maior prioridade que o operador `*` de indireção. É importante relembrar também que essa forma de acesso indireto aos campos de um registro pode ser substituída, e tem o mesmo efeito, pelo operador `->` como mostramos no exemplo abaixo:

```
p->dia = 11;
```

O programa 15.1 ilustra o uso de ponteiros para registros.

Programa 15.1: Uso de um ponteiro para um registro.

```
#include <stdio.h>

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

int main(void)
{
   struct data hoje, *p;

   p = &hoje;
   p->dia = 13;
   p->mes = 10;
   p->ano = 2010;
   printf("A data de hoje é %d/%d/%d\n", hoje.dia, hoje.mes, hoje.ano);

   return 0;
}
```

No programa 15.1, há a declaração de duas variáveis: um registro com identificador `hoje` e um ponteiro para registros com identificador p. Na primeira atribuição, p recebe o endereço da variável `hoje`. Observe que a variável `hoje` é do tipo `struct data`, isto é, a variável `hoje` é do mesmo tipo da variável p e, portanto, essa atribuição é válida. Em seguida, valores do tipo inteiro são armazenados na variável `hoje`, mas de forma indireta, com uso do ponteiro p. Por fim, os valores atribuídos são impressos na saída. A figura 15.1 mostra as variáveis `hoje` e p depois das atribuições realizadas durante a execução do programa 15.1.

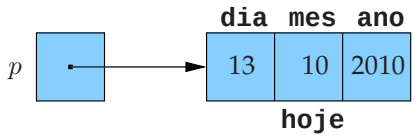

Figura 15.1: Representação do ponteiro p e do registro `hoje`.

15.2 Registros contendo ponteiros

Podemos também usar ponteiros como campos de registros. Por exemplo, podemos definir uma etiqueta de registro como abaixo:

```
struct reg_pts {
   int *pt1;
   int *pt2;
};
```

A partir dessa definição, podemos declarar variáveis (registros) do tipo `struct reg_pts` como a seguir:

```
struct reg_pts bloco;
```

Em seguida, a variável `bloco` pode ser usada como sempre fizemos. Note apenas que `bloco` não é um ponteiro, mas um registro que contém dois campos que são ponteiros. Veja o programa 15.2, que mostra o uso dessa variável.

Observe atentamente a diferença entre `(*`p`).dia` e `*reg.pt1`. No primeiro caso, p é um ponteiro para um registro e o acesso indireto a um campo do registro, via esse ponteiro, tem de ser feito com a sintaxe `(*`p`).dia`, isto é, o conteúdo do endereço contido em p é um registro e, portanto, a seleção do campo é descrita fora dos parênteses. No segundo caso, `reg` é um registro – e não um ponteiro para um registro – e como contém campos que são ponteiros, o acesso ao conteúdo dos campos é realizado através do operador de indireção `*`. Assim, `*reg.pt1` significa que queremos acessar o conteúdo do endereço apontado por `reg.pt1`. Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses, embora pudéssemos usá-los da forma `*(reg.pt1)`. A figura 15.2 ilustra essa situação.

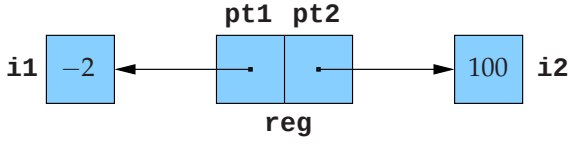

Figura 15.2: Representação do registro `reg` contendo dois campos ponteiros.

Programa 15.2: Uso de um registro que contém campos que são ponteiros.

```
#include <stdio.h>

struct pts_int {
   int *pt1;
   int *pt2;
};

int main(void)
{
   int i1, i2;
   struct pts_int reg;

   i2 = 100;
   reg.pt1 = &i1;
   reg.pt2 = &i2;
   *reg.pt1 = -2;
   printf("i1 = %d, *reg.pt1 = %d\n", i1, *reg.pt1);
   printf("i2 = %d, *reg.pt2 = %d\n", i2, *reg.pt2);

   return 0;
}
```

Exercícios

15.1 Qual a saída do programa descrito abaixo?

```
#include <stdio.h>

struct dois_valores {
   int vi;
   float vf;
};

int main(void)
{
   struct dois_valores reg1 = {53, 7.112}, reg2, *p = &reg1;

   reg2.vi = (*p).vf;
   reg2.vf = (*p).vi;
   printf("1: %d %f\n2: %d %f\n", reg1.vi, reg1.vf, reg2.vi, reg2.vf);

   return 0;
}
```

15.2 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>

struct pts {
   char *c;
   int *i;
   float *f;
};

int main(void)
{
   char caractere;
   int inteiro;
   float real;
   struct pts reg;

   reg.c = &caractere;
   reg.i = &inteiro;
   reg.f = &real;
   scanf("%c%d%f", reg.c, reg.i, reg.f);
   printf("%c\n%d\n%f\n", caractere, inteiro, real);

   return 0;
}
```

15.3 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>

struct celula {
   int valor;
   struct celula *prox;
};

int main(void)
{
   struct celula reg1, reg2, *p;

   scanf("%d%d", &reg1.valor, &reg2.valor);
   reg1.prox = &reg2;
   reg2.prox = NULL;
   for (p = &reg1; p != NULL; p = p->prox)
      printf("%d ", p->valor);
   printf("\n");

   return 0;
}
```