AULA 54

APONTADORES E REGISTROS

Nesta aula trabalharemos com apontadores e registros. Primeiro, veremos como declarar e usar apontadores para registros. Essas tarefas são equivalentes as que já fizemos quando usamos apontadores para números inteiros, por exemplo. Além disso, vamos adicionar também apontadores como campos de registros. Quando registros contêm apontadores podemos usá-los como estruturas de dados poderosas tais como listas encadeadas, árvores, etc, como veremos daqui por diante.

54.1 Apontadores 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 apontadores para inteiros, caracteres e números de ponto flutuante, podemos declarar um apontador 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 apontador `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 54.1 ilustra o uso de apontadores para registros.

Programa 54.1: Uso de um apontador para um registro.

```
1  #include <stdio.h>
2  struct data {
3    int dia;
4    int mes;
5    int ano;
6  };
7  int main(void)
8  {
9    struct data hoje, *p;
10   p = &hoje;
11   p->dia = 13;
12   p->mes = 9;
13   p->ano = 2007;
14   printf("A data de hoje é %d/%d/%d\n", hoje.dia, hoje.mes, hoje.ano);
15   return 0;
16 }
```

Na linha 9 há a declaração de duas variáveis: um registro com identificador `hoje` e um apontador para registros com identificador `p`. Na linha 10, `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 apontador `p`. Por fim, os valores atribuídos são impressos na saída. A figura 54.1 mostra as variáveis `hoje` e `p` depois das atribuições realizadas durante a execução do programa 54.1.

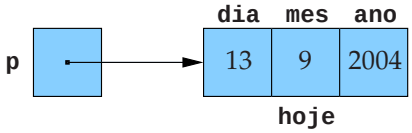

Figura 54.1: Representação do apontador `p` e do registro `hoje`.

54.2 Registros contendo apontadores

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

```
struct reg_apts {
  int *apt1;
  int *apt2;
};
```

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

```
struct reg_apts bloco;
```

Em seguida, a variável `bloco` pode ser usada como sempre fizemos. Note apenas que `bloco` não é um apontador, mas um registro que contém dois campos que são apontadores. Veja o programa 54.2, que mostra o uso dessa variável.

Observe atentamente a diferença entre `(*p).dia` e `*reg.apt1`. No primeiro caso, `p` é um apontador para um registro e o acesso indireto a um campo do registro, via esse apontador, 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 apontador para um registro – e como contém campos que são apontadores, o acesso ao conteúdo dos campos é realizado através do operador de indireção `*`. Assim, `*reg.apt1` significa que queremos acessar o conteúdo do endereço apontado por `reg.apt1`. 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.apt1)`. A figura 54.2 ilustra essa situação.

Programa 54.2: Uso de um registro que contém campos que são apontadores.

```
#include <stdio.h>
struct apts_int {
  int *apt1;
  int *apt2;
};
int main(void)
{
  int i1, i2;
  struct apts_int reg;
  i2 = 100;
  reg.apt1 = &i1;
  reg.apt2 = &i2;
  *reg.apt1 = -2;
  printf("i1 = %d, *reg.apt1 = %d\n", i1, *reg.apt1);
  printf("i2 = %d, *reg.apt2 = %d\n", i2, *reg.apt2);
  return 0;
}
```

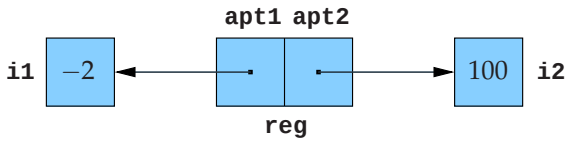

Figura 54.2: Representação do registro `reg` contendo dois campos apontadores.

Exercícios

54.1 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;
}
```

54.2 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>
struct apts {
  char *c;
  int *i;
  float *f;
};
int main(void)
{
  char caractere;
  int inteiro;
  float real;
  struct apts 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;
}
```

54.3 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;
}
```