# Ponteiros e registros

## Aula 14

Diego Padilha Rubert

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Algoritmos e Programação II

# Conteúdo da aula

# 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;
```

# 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;
```

# Ponteiros para registros

- Assim como fazemos com outros ponteiros, podemos declarar um ponteiro para o registro `data`:

```
struct data *p;
```

- A partir daí, podemos fazer uma atribuição à variável *p*:

```
p = &hoje;
```

- Além disso, podemos atribuir valores aos campos do registro de forma indireta:

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

- Os parênteses envolvendo `*p` são necessários porque o operador `.` tem maior prioridade que o operador `*`
- Essa forma de acesso indireto aos campos de um registro pode ser substituída pelo operador `->`:

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

# Ponteiros para registros

- Assim como fazemos com outros ponteiros, podemos declarar um ponteiro para o registro `data`:

```
struct data *p;
```

- A partir daí, podemos fazer uma atribuição à variável *p*:

```
p = &hoje;
```

- Além disso, podemos atribuir valores aos campos do registro de forma indireta:

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

- Os parênteses envolvendo `*p` são necessários porque o operador `.` tem maior prioridade que o operador `*`
- Essa forma de acesso indireto aos campos de um registro pode ser substituída pelo operador `->`:

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

# Ponteiros para registros

- Assim como fazemos com outros ponteiros, podemos declarar um ponteiro para o registro `data`:

```
struct data *p;
```

- A partir daí, podemos fazer uma atribuição à variável *p*:

```
p = &hoje;
```

- Além disso, podemos atribuir valores aos campos do registro de forma indireta:

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

- Os parênteses envolvendo `*p` são necessários porque o operador `.` tem maior prioridade que o operador `*`
- Essa forma de acesso indireto aos campos de um registro pode ser substituída pelo operador `->`:

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

# Ponteiros para registros

- Assim como fazemos com outros ponteiros, podemos declarar um ponteiro para o registro `data`:

```
struct data *p;
```

- A partir daí, podemos fazer uma atribuição à variável *p*:

```
p = &hoje;
```

- Além disso, podemos atribuir valores aos campos do registro de forma indireta:

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

- Os parênteses envolvendo `*p` são necessários porque o operador `.` tem maior prioridade que o operador `*`
- Essa forma de acesso indireto aos campos de um registro pode ser substituída pelo operador `->`:

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

# Ponteiros para registros

- Assim como fazemos com outros ponteiros, podemos declarar um ponteiro para o registro `data`:

```
struct data *p;
```

- A partir daí, podemos fazer uma atribuição à variável *p*:

```
p = &hoje;
```

- Além disso, podemos atribuir valores aos campos do registro de forma indireta:

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

- Os parênteses envolvendo `*p` são necessários porque o operador `.` tem maior prioridade que o operador `*`
- Essa forma de acesso indireto aos campos de um registro pode ser substituída pelo operador `->`:

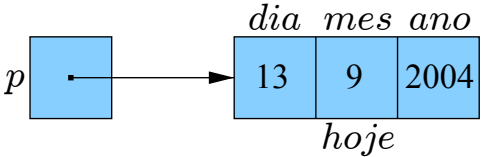
```
p->dia = 11;
```

# Ponteiros para registros

O programa abaixo ilustra o uso de ponteiros para registros.

```
#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;
}
```

# Ponteiros para registros



# Registros contendo ponteiros

- Podemos também usar ponteiros como campos de registros:

```
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.

# Registros contendo ponteiros

- Podemos também usar ponteiros como campos de registros:

```
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.

# Registros contendo ponteiros

- Podemos também usar ponteiros como campos de registros:

```
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.

# Registros contendo ponteiros

- Podemos também usar ponteiros como campos de registros:

```
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.

# Registros contendo ponteiros

```c
#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;
}
```

# Registros contendo ponteiros

- 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
- 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 `*`
- Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses em `*reg.pt1`, embora pudéssemos usá-los da forma `*(reg.pt1)`

# Registros contendo ponteiros

- 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
- 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 `*`
- Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses em `*reg.pt1`, embora pudéssemos usá-los da forma `*(reg.pt1)`

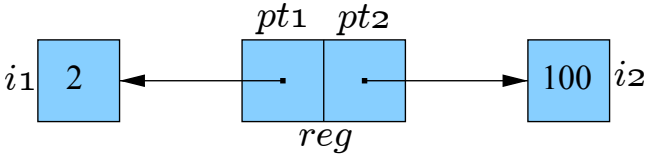
# Registros contendo ponteiros

- 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
- 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 `*`
- Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses em `*reg.pt1`, embora pudéssemos usá-los da forma `*(reg.pt1)`

# Registros contendo ponteiros

- 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
- 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 `*`
- Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses em `*reg.pt1`, embora pudéssemos usá-los da forma `*(reg.pt1)`

# Registros contendo ponteiros



# Exercícios

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;
}
```

# Exercícios

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;
}
```