

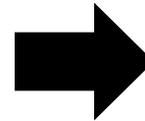
struct, union e listas ligadas

Prof. Paulo Henrique Pisani

struct

Estruturas

- Com o struct, definimos um novo tipo de dados;
- Esse tipo é uma estrutura que permite a combinação de itens de diferentes tipos de dados.



```
struct aluno {  
    int ra;  
    char nome[100];  
    char curso[20];  
};
```

Declarar variável do tipo estrutura

- Para declarar uma variável do tipo **struct aluno**:

```
struct aluno aluno1;  
struct aluno fulano1, fulano2;
```



Tipo da variável

Acesso a membros da estrutura

- Para acessar membros da estrutura, usamos o **ponto**:

```
struct aluno a1;  
a1.ra = 123;
```

```
struct aluno a2, a3;  
a2.ra = 100;  
a3.ra = 200;
```

```
scanf("%d", &a2.ra);  
scanf("%s", a2.nome);
```

```
struct aluno {  
    int ra;  
    char nome[100];  
    char curso[20];  
};
```

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

```
struct aluno {  
    int ra;  
    char nome[100];  
    char curso[20];  
};
```

} Declaração do tipo de dados (estrutura);

```
int main() {
```

```
    struct aluno p; ← Variável do tipo struct.
```

```
    scanf("%d", &p.ra);  
    scanf("%s", p.nome);  
    scanf("%s", p.curso);
```

```
    printf("RA=%d Nome=%s Curso=%s\n", p.ra, p.nome, p.curso);
```

```
    return 0;
```

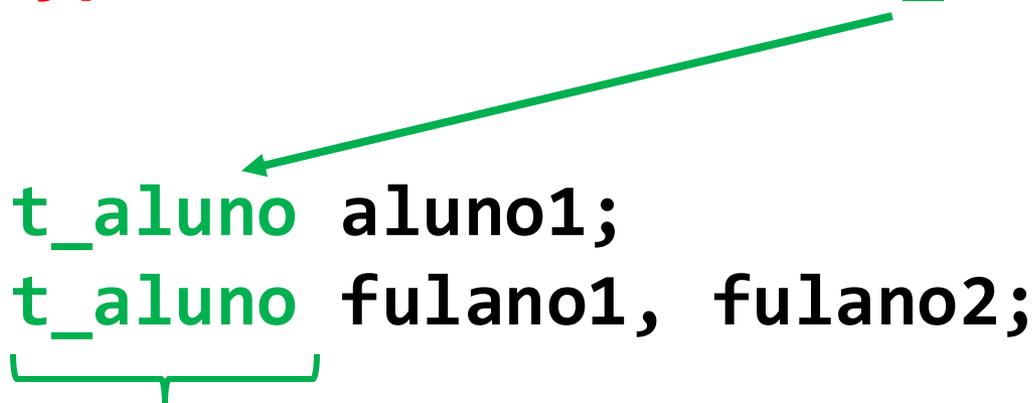
```
}
```

Declarar variável do tipo estrutura

- Podemos criar um sinônimo para o tipo de dados, e assim facilitar a declaração;

```
typedef struct aluno t_aluno;
```

```
t_aluno aluno1;  
t_aluno fulano1, fulano2;
```



Tipo da variável

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

```
struct aluno {  
    int ra;  
    char nome[100];  
    char curso[20];  
};
```

Declaração do tipo de dados (estrutura);

```
typedef struct aluno t_aluno;
```

```
int main() {
```

```
t_aluno p; ← Variável do tipo struct.
```

```
scanf("%d", &p.ra);  
scanf("%s", p.nome);  
scanf("%s", p.curso);
```

```
printf("RA=%d Nome=%s Curso=%s\n", p.ra, p.nome, p.curso);
```

```
return 0;
```

```
}
```

Vetores de estruturas

- Por exemplo: `#include <stdio.h>`

```
typedef struct aluno t_aluno;
struct aluno {
    int ra;
    double nota;
};
```

```
int main() {

    t_aluno alunos[3];

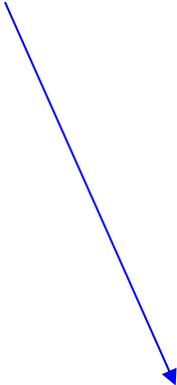
    return 0;
}
```

Alocação dinâmica de estruturas

Podemos alocar estruturas dinamicamente também!

- Por exemplo:

```
typedef struct aluno t_aluno;  
struct aluno {  
    int ra;  
    char *nome;  
    double nota;  
};
```



```
t_aluno *a1;  
a1 = malloc(sizeof(t_aluno));
```

```
t_aluno *a2 = malloc(sizeof(t_aluno));
```

Acesso a membros de um ponteiro para estrutura

- Para acessar membros de um ponteiro para estrutura, temos duas alternativas:

1) Resolver ponteiro e acessar com o **ponto**:

```
t_aluno *a1 = malloc(sizeof(t_aluno));  
(*a1).ra = 123;
```

2) Utilizar o operador “->”:

```
t_aluno *a1 = malloc(sizeof(t_aluno));  
a1->ra = 123;
```

Uma estrutura pode referenciar a si mesma!

- Será que podemos fazer isso então?

```
struct disciplina {  
    int cod;  
    char *nome;  
    int creditos;  
    struct disciplina requisito;  
};
```

Uma estrutura pode referenciar a si mesma!

- Será que podemos fazer isso então?

```
struct disciplina {
    int cod;
    char *nome;
    int creditos;
    struct disciplina *prerequisite;
};
```

NÃO !!! Isso torna a declaração recursiva!

Uma estrutura pode referenciar a si mesma!

- Será que podemos fazer isso então?

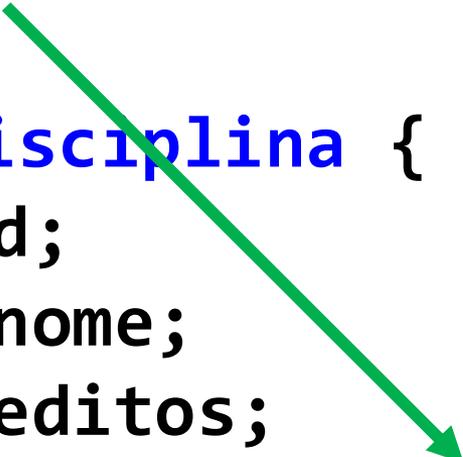
```
struct disciplina {
    int cod;
    char *nome;
    int creditos;
    struct disciplina *prerequisite;
};
```

Qual seria o tamanho de um *struct disciplina* em memória com a definição anterior? Seria infinito!

Uma estrutura pode referenciar a si mesma!

- Mas podemos referenciar usando ponteiros:

```
struct disciplina {  
    int cod;  
    char *nome;  
    int creditos;  
    struct disciplina *requisito;  
};
```



union

union

- Sintaxe similar ao struct;
- **Entretanto, o mesmo espaço de memória é compartilhado por todos os campos!**
- Portanto, alterar o valor de um membro implica em alterar os demais!

```
union ip {  
    int numero;  
    char bytes[4];  
};
```

union

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

```
union ip {  
    int numero;  
    char bytes[4];  
};
```

```
int main() {  
    union ip a;  
    a.numero = 259;  
    printf("%d %d %d %d\n",  
        a.bytes[3], a.bytes[2], a.bytes[1], a.bytes[0]);  
  
    printf("%d\n", a.numero);  
    a.bytes[1] = 2;  
    printf("%d\n", a.numero);  
    return 0;  
}
```

O que será impresso?

union

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

```
union ip {  
    int numero;  
    char bytes[4];  
};
```

```
int main() {  
    union ip a;  
    a.numero = 259;  
    printf("%d %d %d %d\n",  
           a.bytes[3], a.bytes[2], a.bytes[1], a.bytes[0]);  
  
    printf("%d\n", a.numero);  
    a.bytes[1] = 2;  
    printf("%d\n", a.numero);  
    return 0;  
}
```

```
0 0 1 3  
259  
515
```

Listas ligadas

Listas com arranjos (revisão)

- Itens dispostos em um arranjo sequencial;

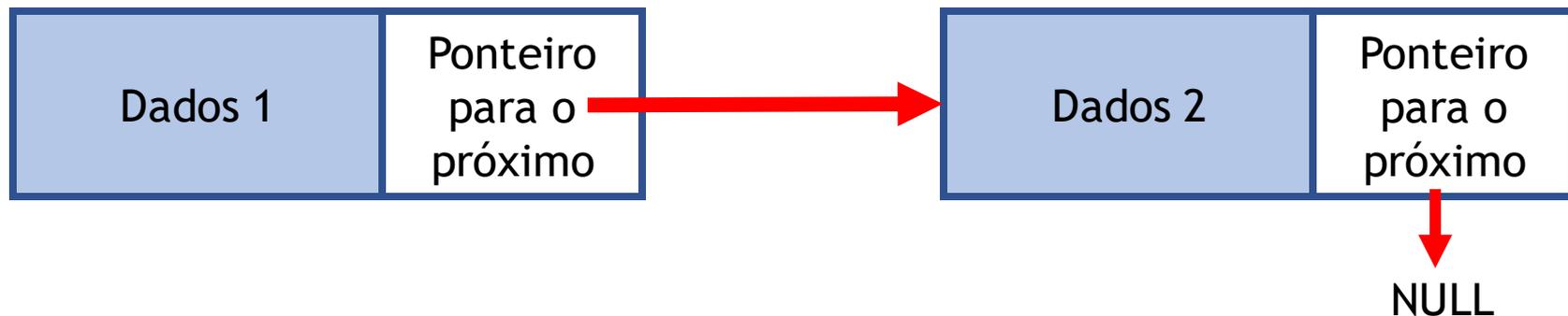


Problemas?

(pense das operações básicas: busca, inserção, remoção)

Listas ligadas/encadeadas

- Estrutura de dados que armazena os itens de forma não consecutiva na memória:
 - Usa ponteiros para “ligar” um item no próximo.



Listas ligadas/encadeadas

- Vários tipos:
 - Listas simplesmente ligadas (com e sem nó cabeça);
 - Listas duplamente ligadas (com e sem nó cabeça);
 - Listas circulares.

Listas simplesmente ligadas

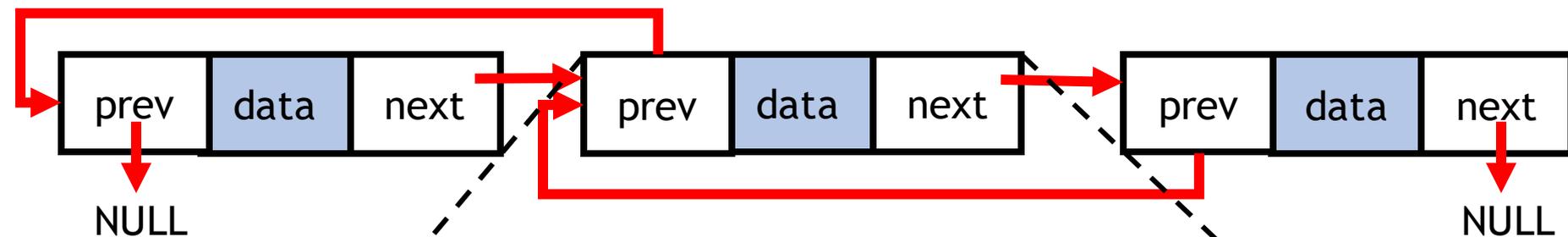
- Cada item é ligado somente ao próximo item;



```
typedef struct linked_node linked_node;
struct linked_node {
    int data;
    linked_node *next;
};
```

Listas duplamente ligadas

- Cada item é ligado ao próximo item e também ao anterior;
- **Vantagem: a lista pode ser percorrida em ambas as direções.**



```
typedef struct linked_node linked_node;
struct linked_node {
    int data;
    linked_node *prev, *next;
};
```

Referências

- Slides do Prof. Paulo H. Pisani: Programação Estruturada (Q3/2018)
- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.
- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. Introdução a Estruturas de Dados. Elsevier/Campus, 2004.