Registros

Disciplina de Programação de Computadores I
Universidade Federal de Ouro Preto

Agenda

- Registros
- Sinônimos de tipos
- Alocação dinâmica de Registros
- Lista Ligada

Registros

- Um registro (struct) é a forma de agrupar variáveis em C, sejam elas de mesmo tipo ou de tipos diferentes

- As variáveis unidas em um registro se relacionam de forma a criar um contexto maior

- Exemplos de uso de registros:

    - Registro de Alunos: armazena nome, matrícula, médias, faltas, etc.

    - Registro de Pacientes: armazena nome, endereço, convênio, histórico hospitalar

Declaração do tipo registro (I)

- Um registro é declarado com a palavra-reservada **struct**

```
struct nome_tipo_registro {
        tipo_1 variavel_1;
        tipo_2 variavel_2;
        ...
        tipo_n variavel_n;
};
```

- Devemos declarar variáveis deste novo tipo assim:

```
struct nome_tipo_registro variavel_registro;
```

Declaração do tipo registro (II)

- Um registro define um novo tipo de dados com nome nome_tipo_registro que possui os campos variavel_i

- O campo variavel_i é uma variável do tipo tipo_i e será acessível a partir de uma variável do novo tipo nome_tipo_registro.

- variavel_registro é uma variável do tipo nome_tipo_registro e possui, internamente, os campos variavel_i do tipo declarado.

- A declaração de um registro pode ser feita dentro de uma sub-rotina ou fora dela.

Exemplo de declaração do tipo registro

```
struct regAluno {
        char nome[50];
        int idade;
        char sexo;
        int turma;
};
int main(void){
        struct regAluno aluno1, aluno2;
}
```

Acesso aos campos do registro

- Os campos de um registro podem ser acessados individualmente a partir de variáveis do tipo do registro da seguinte forma:

variavel_registro.variavel_i

- Cada campo variavel_registro.variavel_i se comporta como uma variável do tipo do campo, ou seja, tipo_i

- Isto significa que todas as operações válidas para variáveis do tipo tipo_i são válidas para o campo acessado por variavel_registro.variavel_i

Inicialização de Registros

- Na inicialização de registros, os campos obedecem à ordem de declaração:

```
struct contaBancaria {
        int numero;
        char idCorrentista[15];
         float saldo;
};


struct contaBancaria conta = { 1, "MG1234567", 100.0 };
```

Atribuição de registros

- Pode-se copiar o conteúdo de uma estrutura para outra utilizando uma atribuição simples.

```
struct ponto {int x; int y;};
int main() {
struct ponto p = {1,2}, q;
q.x = 3; q.y = 4;
q = p;
q.x = 5;
}
```

p		q	
x	y	x	y
1	2		
1	2	3	4
1	2	1	2
1	2	5	2

Vetores de registros

- Pode-se criar vetores de registros da mesma maneira que se criam vetores de tipos primitivos (int, float, char, double).

- É necessário definir a estrutura antes de declarar o vetor.

```
struct ponto {int x; int y;};
int main() {
    struct ponto v[2];
    v[0].x = 3; v[0].y = 4;
    v[1].x = 5; v[1].y = 6;
}
```

Registros como parâmetros de sub-rotinas

- Pode-se passar um registro como parâmetro de subrotina.

```
struct ponto {int x; int y;};
void imprimePonto( struct ponto p){
        printf(“Coordenadas (%d, %d)”, p.x, p.y);
}
int main() {
        struct Ponto p = {1,2};
   imprimePonto (p);
}
```

Sinônimos de tipos com typedef

- Permite dar novos nomes (sinônimos) a tipos de dados existentes

```
typedef int inteiros;
typedef char caracteres;
struct contabancaria {
    inteiros numero;
    caracteres idCorrentista[15];
    float saldo;
};
typedef struct contabancaria CB;

CB conta1, conta2;
```

Utilizando sinônimos junto da definição de registros

- Pode-se associar uma definição de struct a um novo tipo de dados, evitando a repetição da palavra struct pelo código:

```
typedef int inteiros;
typedef char caracteres;
```

```
typedef struct contabancaria {
    int numero;
    char idCorrentista[15];
    float saldo;
} CB;

CB conta1, conta2;
```

Exemplo Completo de Registro

```
typedef struct aluno {
        char nome [40];
        double nota[4];
} Aluno;

void leVetorAluno(Aluno temp[], int tam){
        for (int i = 0; i < tam; ++i){
                fgets(temp[i].nome, 40, stdin);
                for (int j = 0; j < 4; ++j){
                        char leitura[15];
                        fgets(leitura, 15, stdin);
                        double notaLida = atof(leitura);
                        temp[i].nota[j] = notaLida;
                } } }

void imprimeVetorAluno(Aluno temp[], int
tam){
        for (int i = 0; i < tam; ++i){
                printf("Aluno %d:", i+1);
                printf("\tNome: %s\n",
                        temp[i].nome);
                for (int j = 0; j < 4; ++j){
                        printf("\tNota %d = %.2lf\n",
                                j+1, temp[i].nota[j]);
                                }}}

int main(){
        Aluno alunos[2];
        leVetorAluno(alunos, 2);
        imprimeVetorAluno(alunos, 2);
        return 0; }
```

Alocação dinâmica de Registros (I)

- Para acessar os elementos de um registro, deve-se acessar o registro e, depois, os elementos;

- Os parênteses são obrigatórios porque a precedência do operador * é menor que o do operador .

```
typedef struct ponto { int x; int y; } Ponto;
Ponto* ponto_ptr ;
ponto_ptr = (Ponto*) malloc (sizeof(Ponto));
(*ponto_ptr).x = 3;
(*ponto_ptr).y = 4;
free(ponto_ptr);
```

Alocação dinâmica de Registros (II)

- O operador -> facilita o acesso aos registros, permitindo acessar diretamente o conteúdo de um campo do registro

```
typedef struct ponto { int x; int y; } Ponto;

Ponto* ponto_ptr ;

ponto_ptr = (ponto*) malloc (sizeof(Ponto));

ponto_ptr->x = 3;

ponto_ptr->y = 4;

free(ponto_ptr);
```

Exemplo de alocação dinâmica de registro

```
typedef struct aluno {
        char nome [40];
        double nota[4];
} Aluno;

void leVetorAluno(Aluno* temp, int tam){
        for (int i = 0; i < tam; ++i){
                fgets(temp[i].nome, 40, stdin);
                for (int j = 0; j < 4; ++j){
                        char leitura[15];
                        fgets(leitura, 15, stdin);
                        double notaLida = atof(leitura);
                        temp[i].nota[j] = notaLida;
                } } }

void imprimeVetorAluno(Aluno* temp, int
tam){
        for (int i = 0; i < tam; ++i){
                printf("Aluno %d:\n", i+1);
                printf("\tNome: %s", temp[i].nome);
                for (int j = 0; j < 4; ++j){
                        printf("\tNota %d = %.2lf\n",
                                j+1, temp[i].nota[j]);
                        } } }

int main(){
        Aluno* alunos =
                (Aluno*)malloc(2 * sizeof(Aluno));
        leVetorAluno(alunos, 2);
        imprimeVetorAluno(alunos, 2);
        free (alunos);
        return 0;
}
```

Lista Ligada

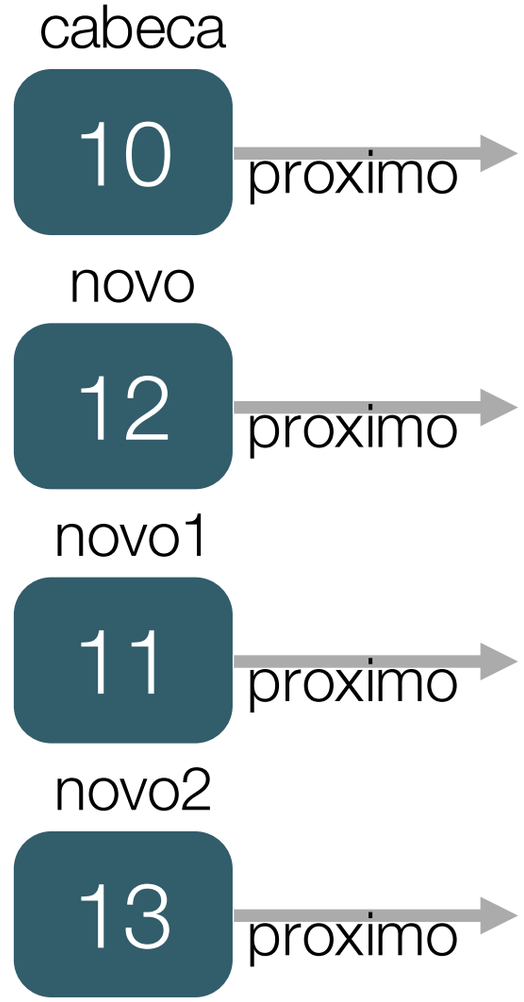
- Lista ligada é uma estrutura de dados que utiliza uma série de registros de um mesmo tipo “ligados” uns aos outros através de um ponteiro interno.
- Uma lista ligada pode ser vista como uma sequência de quadrados que possuem um dado interno e possuem uma ligação com um outro elemento do mesmo tipo deste quadrado

Lista Ligada: Representação

```
struct nolista {
    struct nolista * proximo;
    int dado;
};
typedef struct nolista NoLista;
typedef struct nolista* NoListaPtr;
```

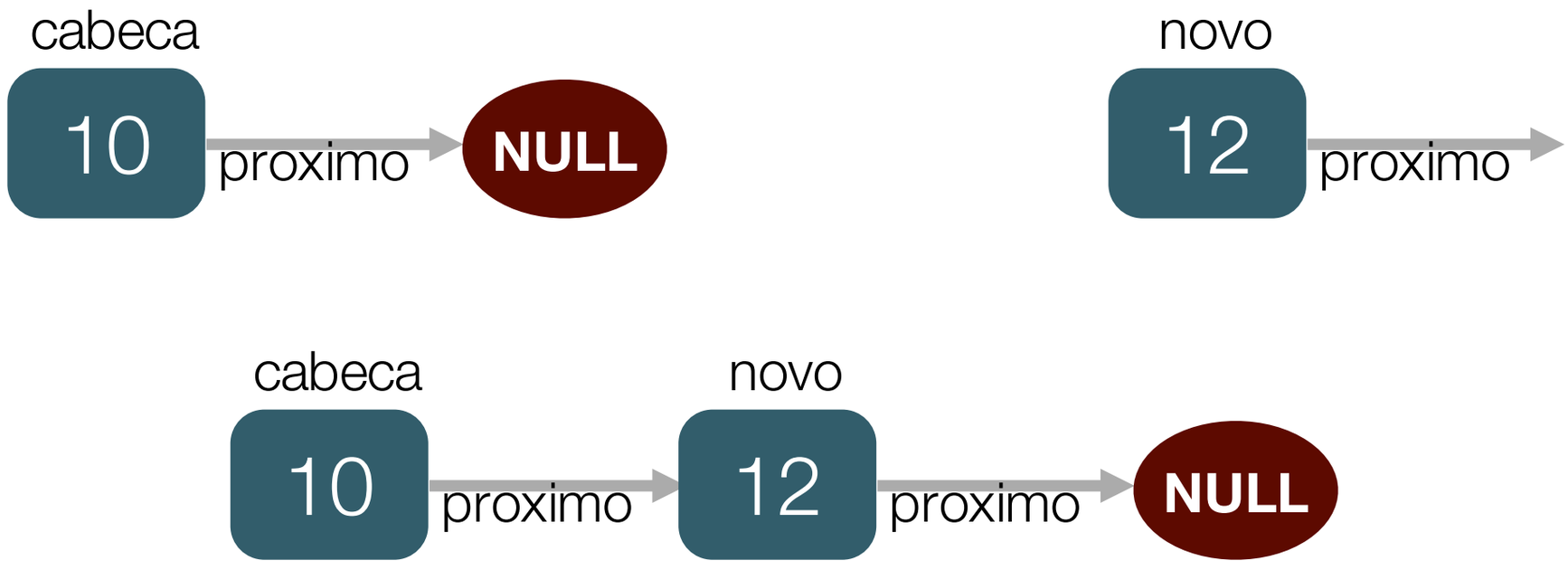
Lista Ligada: Criação de nós

```
NoListaPtr cabeca, novo, novo1, novo2;

cabeca = criaNo(10);

novo = criaNo(12);

novo1 = criaNo(11);

novo2 = criaNo(13);
```



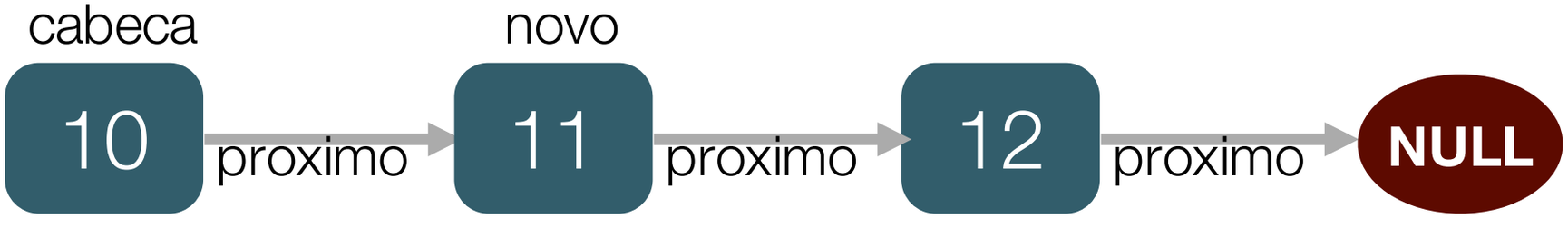
Lista Ligada: Adiciona nó na cabeca

```
void adicionaNaCabeca(NoListaPtr cabeca, NoListaPtr novo){
    if ( cabeca != NULL ) {
        novo->proximo = cabeca->proximo;
        cabeca->proximo = novo;
    }
}
```



Lista Ligada: Adiciona nó em ordem crescente (I)

```
void adicionaOrdemCrescente(NoListaPtr cabeca, NoListaPtr novo){
    while ( cabeca != NULL && cabeca->proximo != NULL
            && novo->dado > cabeca->proximo->dado )
        cabeca = cabeca->proximo;

    novo->proximo = cabeca->proximo;
    cabeca->proximo = novo;
}
```



Lista Ligada: Adiciona nó em ordem crescente (II)

```
void adicionaOrdemCrescente(NoListaPtr cabeca, NoListaPtr novo){
    while ( cabeca != NULL && cabeca->proximo != NULL
            && novo->dado > cabeca->proximo->dado )
        cabeca = cabeca->proximo;

    novo->proximo = cabeca->proximo;
        cabeca->proximo = novo;
    }
```

Lista Ligada: Impressão

```
void imprimeLista(NoListaPtr cabeca){
    printf("Nós da lista:\n");
    while(cabeca != NULL){
        printf("Nó %d\n", cabeca->dado);
        cabeca = cabeca->proximo;
    }
    printf("\n");
}
```

Lista Ligada: Remoção (I)

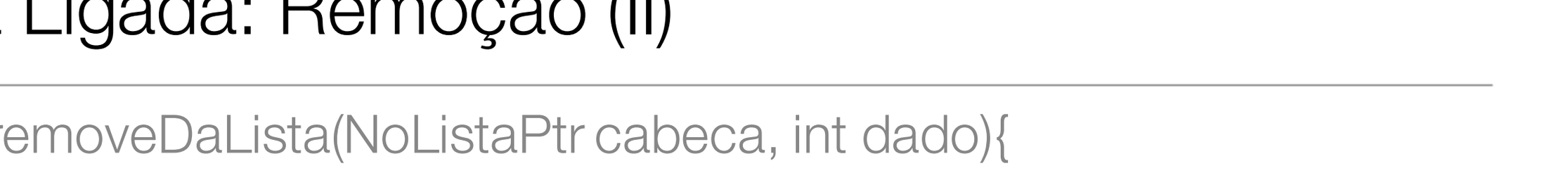
```
void removeDaLista(NoListaPtr cabeca, int dado){
    if(cabeca->dado == dado){
        NoListaPtr lixo = cabeca;
        cabeca = cabeca->proximo;
        free(lixo);
    }else{...}
}
```

dado = 10

Lista Ligada: Remoção (II)

```
void removeDaLista(NoListaPtr cabeca, int dado){
    if(cabeca->dado == dado){...}else{
        while(cabeca != NULL && cabeca->proximo != NULL
                && cabeca->proximo->dado != dado)
            cabeca = cabeca->proximo;
        if(cabeca != NULL && cabeca->proximo != NULL){
            NoListaPtr lixo = cabeca->proximo;
            cabeca->proximo = lixo->proximo;
            free(lixo);
        }
    }
}
```

dado = 12



Referências Bibliográficas

- Material de aula do Prof. Ricardo Anido, da UNICAMP: http://www.ic.unicamp.br/~ranido/mc102/

- Material de aula da Profa. Virgínia F. Mota: https://sites.google.com/site/virginiaferm/home/disciplinas

- DEITEL, P; DEITEL, H. *C How to Program*. 6a Ed. Pearson, 2010.