



CEFET – RJ / Campus Maria da Graça
Centro Federal de Educação Tecnológica
Celso Suckow da Fonseca – Rio de Janeiro



Prof. Cristiano Fuschilo
Cristiano.fuschilo@cefet-rj.br

Estruturas de Dados 3º Período



BACHARELADO
Sistemas de
Informação

Struct – Registros em Linguagem C



- Em Linguagem C Registros são chamados de Estruturas e a palavra-chave é Struct.
- Registros é um conceito de programação que é implementado pelos compiladores de cada linguagem de programação.





Definindo uma Estrutura

- Uma estrutura pode ser definida de formas diferentes.
- No corpo da estrutura encontram-se os membros, ou seja, as variáveis de diversos tipos que comporão esse tipo de dado heterogêneo definido pelo usuário.
- Depois de definida uma estrutura, uma (ou mais) variável do tipo estrutura deve ser definida, para permitir a manipulação dos membros da estrutura.





Forma Geral

- A forma geral de declaração da Estrutura é:

```
1 struct etiqueta{  
2     tipo var1;  
3     tipo var2;  
4     .  
5     .  
6     .  
7     tipo varN;  
8 };
```

- Onde:
 - o nome, ou **Etiqueta (TAG)**, da estrutura é colocado logo em seguida da palavra-chave **Struct**.
 - Dentro da estrutura são definidos os tipos e os campos que a compõe!





Exemplo

- Nesse exemplo a Etiqueta da estrutura é nomeada como ALUNO.

```
1 struct aluno {  
2     int codigo;  
3     char nome[200];  
4     float nota;  
5 };
```

- A estrutura aluno tem os campos código, nome e nota, respectivamente dos tipos inteiro, string e real.
- ATENÇÃO: ainda não existe alocado em memória nenhuma variável do tipo aluno, aqui informamos ao compilador que podemos, a partir de agora, declarar variáveis do tipo aluno!



Declarando Variáveis do Tipo aluno



- Para declarar a Variável:
- Repete a palavra reservada struct e a etiqueta e depois define o nome da Variável

```
1 struct aluno {  
2     int codigo;  
3     char nome[200];  
4     float nota;  
5 };  
6 struct aluno aluno_especial, aluno_regular, aluno_ouvinte;
```



- Dessa forma foram declaradas três variáveis do tipo struct aluno, sendo: aluno_especial, aluno_regular e aluno_ouvinte.



Acessando os membros da variável do tipo Estrutura



- Para acessar os membros da estrutura, quando ela é diretamente referenciada, devemos utilizar o Ponto, que também é chamado de operador de seleção direta, veja:

```
1 aluno_especial.codigo  
2 aluno_especial.nome  
3 aluno_especial.nota
```

- Você pode atribuir valores aos membros das estruturas diretamente, e em qualquer parte do programa, conforme a seguir:

```
1 aluno_especial.codigo = 10;  
2 strcpy(aluno_especial.nome, "Manoel");  
3 aluno_especial.nota = 10.0;
```



Imprimindo os membros da Estrutura



- Você pode imprimir os membros da estrutura em qualquer parte do programa que desejar.

```
1 printf(" \n %d ", aluno_especial.codigo);
2 printf(" \n %s ", aluno_especial.nome);
3 printf(" \n %.2f ", aluno_especial.notas);
```





Obtendo Valores do Teclado

- Para obter dados do teclado devemos utilizar o `scanf`, tomando cuidado quando formos usar strings.
- Podemos obter dados do teclado em qualquer parte do programa.

```
1 printf(" Digite o código do aluno especial: ");
2 scanf("%d%c", &aluno_especial.codigo);
3 printf(" Digite o nome do aluno especial: ");
4 scanf("%s%c", &aluno_especial.nome);
5 printf(" Digite a nota do aluno especial: ");
6 scanf("%f%c", &aluno_especial.nota);
```





Ponteiros



Ponteiros

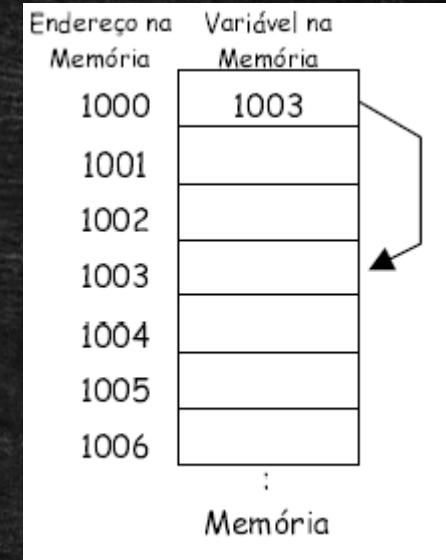


- É uma das mais poderosas estruturas de dados. Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente. O mecanismo usado para isto é o endereço da variável, sendo o ponteiro a representação simbólica de um endereço.
- Utilização dos ponteiros:
 - Manipular elementos de matrizes;
 - Receber parâmetros em funções que necessitem modificar o valor original;
 - Passar strings de uma função para outra;
 - Criar estruturas de dados dinâmicas, como pilhas, listas e árvores, onde um item deve conter referências a outro;
 - Alocar e desalocar memória do sistema.



O Funcionamento

- A memória do computador é dividida em bytes, estes bytes são numerados de 0 até o limite da memória da máquina. Estes números são chamados endereços de bytes. Um endereço é a referência que o computador usa para localizar variáveis. Toda variável ocupa uma certa localização na memória, e seu endereço é o primeiro byte ocupado por ela.
- O C oferece dois operadores para trabalharem com ponteiros. Um é o operador de endereço (**&**) que retorna o endereço de memória da variável. O segundo é o operador indireto (*****) que é o complemento de (**&**) e retorna o conteúdo da variável localizada no endereço (ponteiro) do operando, isto é, devolve o conteúdo da variável apontada pelo operando.
- Declaração
tipo ***variável;**





Declaração

- Se uma variável irá conter um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste no tipo de base, um * e o nome da variável. A forma geral é: <tipo> *<nome>; onde <tipo> é qualquer tipo válido em C e <nome> é o nome da variável ponteiro
- O tipo base define que tipo de variável o ponteiro pode apontar. Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória. No entanto, toda a aritmética de ponteiro é feita por meio de tipo base, assim é importante declarar o ponteiro corretamente.





Funcionamento

- Existem dois operadores especiais para ponteiros: * e &. O & é um operador unário* que devolve o endereço na memória do operando. Por exemplo:
`int *m, cont = 100, q;`
`m = &cont;`

É colocado em “m: o endereço da memória que contém a variável “cont”. O endereço não tem relação alguma com o valor de “cont”. O operador & pode ser imaginado como retornando “o endereço de”. Assim, o comando de atribuição anterior significa “m recebe o endereço de cont”.



* Operador *unário* é aquele que interage sobre apenas um elemento





Funcionamento

- O segundo operador de ponteiro, *****, é o complemento de **&**. É um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se “m” contém o endereço da variável “cont”,

q = *m;

coloca o valor de “cont” em “q”. Portanto, “q” terá o valor 100, o operador ***** pode ser imaginado como **“no endereço”**. Nesse caso, o comando anterior significa “q recebe o valor que está no endereço m”.



Explicando



- O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado! Isto é de suma importância!
- Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int Count=10;  
int *pt;  
pt= &Count;
```

- Criamos um inteiro **Count** com o valor 10 e um apontador para um inteiro **pt**. A expressão **&Count** nos dá o endereço de **Count**, o qual armazenamos em **pt**. Simples, não é? Repare que não alteramos o valor de **Count**, que continua valendo 10.





Explicando

- Como nós colocamos um endereço em pt, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de count usando pt. Para tanto vamos usar o operador "inverso" do operador &. É o operador *. No exemplo acima, uma vez que fizemos pt= &count a expressão *pt é equivalente ao próprio count. Isto significa que, se quisermos mudar o valor de count para 12, basta fazer *pt=12.
- Uma observação importante: apesar do símbolo ser o mesmo, o operador * (multiplicação) não é o mesmo operador que o * (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário pré-fixado.





Exemplo

```
#include <stdio.h>
int main ()
{
    int num,Valor;
    int *p;
    num=55;
    p=&num; /* Pega o endereco de num */
    Valor=*p; /* Valor e igualado a num de uma maneira indireta */
    printf ("\n\n%d\n",Valor);
    printf ("Endereco para onde o ponteiro aponta: %p\n",p);
    printf ("Valor da variavel apontada: %d\n",*p);
    return(0);
}
```





Exemplo

```
#include <stdio.h>

int main ()
{
    int num,*p;
    num=55;
    p=&num; /* Pega o endereço de num */
    printf ("\nValor inicial: %d\n",num);
    *p=100; /* Muda o valor de num de uma maneira indireta */
    printf ("\nValor final: %d\n",num);
    return(0);
}
```



Cuidados a Serem Tomados ao se Usar Ponteiros



- O principal cuidado ao se usar um ponteiro deve ser: saiba sempre para onde o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como não usar um ponteiro:

```
int main () /* Errado - Nao Execute */  
{  
    int x,*p;  
    x=13;  
    *p=x;  
    return(0);  
}
```

- Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro p pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).



Atenção – Perigo – Danger – Cuidado



- O que você aprendeu AGORA é de suma importância.
- Não siga adiante antes de entendê-la bem.





Estruturas de Dados

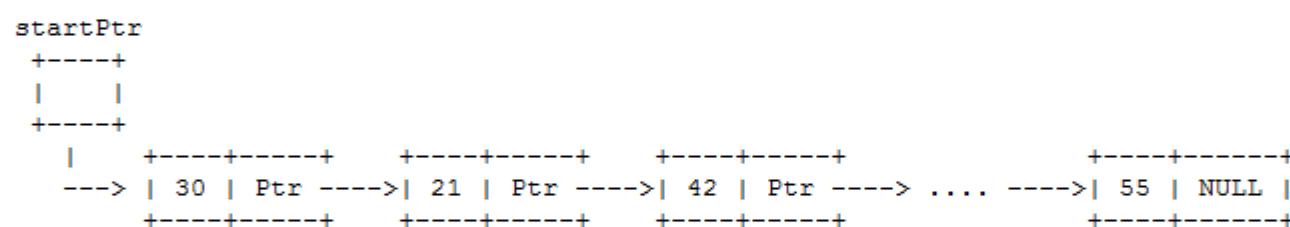
Listas Dinâmicas





Lista Encadeada Simples

- A primeira estrutura de dados a ser estudada será a **Lista Encadeada Simples**.
É chamado de **Lista Encadeada Simples** porque:
Lista : é um conjunto de elementos que seguem uma ordem.
- **Encadeada**: porque um elemento é ligado ao elemento seguinte.
- **Simples** : apenas o elemento da frente tem uma ligação com o elemento seguinte.
- Vamos ver um exemplo com um desenho:



Explicando a imagem



- O startPtr é um ponteiro que contém o endereço do início da lista.
- Ele aponta para um elemento que contém o valor 30 e outro ponteiro.
- Este outro ponteiro consequentemente, aponta para outro elemento que contém outro valor e outro ponteiro.
- A lista continua até que o ponteiro de determinado elemento não tenha um endereço em seu conteúdo, ou seja, seu conteúdo será NULL.
- Esta é uma abordagem teórica do que seria uma Lista Encadeada Simples.
- Na próxima seção iremos começar a transformar em programa este exemplo.





Transformar em código

- A primeira parte é definir como será cada elemento.
- Em C usaremos o um struct para montarmos cada elemento:

```
struct listNode {  
    int data;  
    struct listNode *nextPtr;  
}; typedef listNode ListNode;  
ListNode list;
```





O que faz este código ?

- Ele monta uma estrutura com 2 campos: Um campo que é um valor integer, outro campo que é um ponteiro para uma outra estrutura.
- Este tipo de estrutura é chamada de Estrutura Auto-referenciada, porque contém um membro que aponta para uma estrutura do mesmo tipo.
- Poderíamos modificar aquele int data por qualquer outro conteúdo que desejariamos guardar na nossa lista. É válido até mesmo conter outras estruturas no conteúdo de cada elemento.





Exemplo:

```
struct Nome {  
    char primeiro[30];  
    char segundo[30];  
};  
  
struct CadastroNode {  
    struct Nome nome;  
    int telefone;  
    int codigo;  
    struct CadastroNode *nextPtr;
```



Adicionar Elementos a Lista



- A idéia básica é o seguinte:
- Teremos um ponteiro que será o primeiro elemento da lista, o ponteiro deste primeiro elemento apontará para o segundo, assim sucessivamente até que chegue ao fim da lista, que neste caso, representado pelo ponteiro **NULL**.



Vamos observar a nossa estrutura do elemento:



```
struct listNode {  
    int data;  
    struct listNode *nextPtr;  
};
```

- Nesse código estamos definindo a variável `list` como sendo cada elemento da lista. Para "criarmos" este elemento na memória iremos aloca-lo dinamicamente com o `malloc` (em C, em C++ `new`):

```
newElement = malloc(sizeof(listNode));
```

```
newElement = new listNode();
```

- Para ligarmos este elemento ao seguinte, um possível código seria este:
`newElement.next = nextElement;`



Aqui uma função simples para adicionar um elemento na ultima posição da lista:

```
void insert(listNode **listPtr, int value) {  
    ListNode newPtr, previousPtr, currentPtr;  
  
    newPtr = malloc(sizeof(ListNode));  
  
    if (newPtr != NULL) { /* Alocado com sucesso */  
  
        newPtr->data = value;  
  
        newPtr->nextPtr = NULL;  
  
        previousPtr = NULL;  
  
        currentPtr = *listPtr;  
  
        while (currentPtr != NULL) { /* Percorrer a lista até o fim */  
  
            previousPtr = CurrentPtr;  
  
            currentPtr = CurrentPtr->nextPtr;  
  
        }  
    }  
}
```



/ Caso seja o primeiro elemento da lista, o elemento sucessor do novo apontar o apontará para a lista e a lista apontará para o novo elemento */*

if (previousPtr == NULL) {

*newPtr->nextPtr = *listPtr;*

**listPtr = newPtr;*

} / Caso NAO seja o primeiro elemento, o elemento sucessor da lista apontará para o novo e o novo apontará para o ultimo */*

else {

previousPtr->nextPtr = newPtr;

newPtr->nextPtr = CurrentPtr;

}

} else

*printf("%i não inserido, não há memoria disponivel.
" value);*

Pode-se perceber no código acima 3 partes:

- Criar espaço na memoria para o novo elemento;
- Testar a alocação;
- Posicionar o novo elemento ao fim da lista.



Remover Elementos da Lista

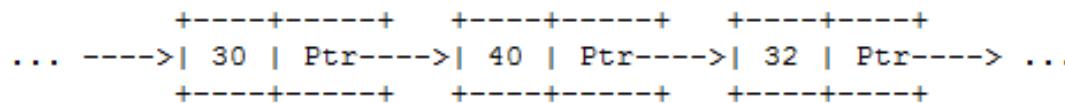


- Junto com adicionar , remover elementos é uma das funções mais importantes que uma lista deve ter. Remover um elemento consiste simplesmente em procurar na lista o elemento a ser apagado, fazer com que o elemento anterior ao ser apagado aponte para o elemento sucessor do qual será apagado e finalmente apagar o elemento da memória.

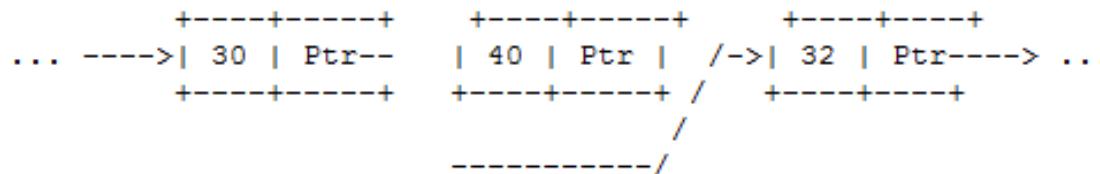


Exemplo como se procede a remoção de um elemento.

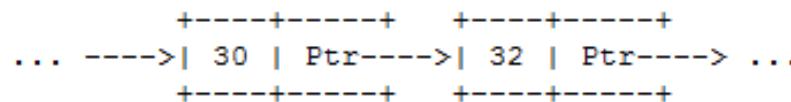
Antes:



Durante, mudando os ponteiros:



Finalmente, liberando a estrutura do elemento:



Um exemplo de código que faz isto:

```
int delete (ListNode **listPtr, int value) {  
    ListNode previousPtr, currentPtr, tempPtr;  
    if(value == (*listPtr)->data) {  
        tempPtr = *listPtr;  
        *listPtr = (*listPtr)->nextPtr;  
        /* retira a ligacao */  
        free(tempPtr);  
        return 0;  
    }  
}
```



```
else {  
    previousPtr = *listPtr;  
  
    currentPtr = (*listPtr)->nextPtr;  
  
    while(currentPtr != NULL && currentPtr->data !=  
          value) {  
        previousPtr = currentPtr;  
  
        currentPtr = currentPtr->nextPtr;  
    }  
  
    if (currentPtr != NULL) {  
        tempPtr = currentPtr;  
  
        previousPtr->nextPtr = currentPtr->nextPtr;  
  
        free(tempPtr);  
  
        return 0;  
    }  
}
```

Este código faz o seguinte:



- Primeiro, se testa para checar se o primeiro elemento já possui o valor a ser apagado. Caso não seja, ele entra num loop onde só irá parar quando econtrar o elemento, ou encontrar o fim da lista.
- Encontrando o valor, ele atribui o currentPtr ao tempPtr, para poder apagar da memória. Apos, ele será o ponteiro do elemento anterior para o seu elemento seguinte.



