

## Ponteiros (parte 3) Estruturas Introdução a arquivos

Baseado no material de:

Ciro Trindade (Unisantos)

Leandro C. Fernandes (ICMC-USP)

## Alocação dinâmica de memória

- A linguagem C oferece meios de requisitarmos espaços de memória em tempo de execução.
- O espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa.
  - Por isso, podemos alocar dinamicamente um espaço de memória numa função e acessá-lo em outra.
- A partir do momento que liberarmos o espaço, ele estará disponibilizado para outros usos e não podemos mais acessá-lo.
  - Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar.

2

## Vetores e alocação dinâmica

- Suponha que desejamos armazenar uma quantidade  $n$  de valores inteiros, porém essa quantidade somente é conhecida pelo usuário.
- Como proceder a reserva de memória necessária para os dados?
  - Se declarmos um vetor grande podemos incorrer em dois riscos:
    - ainda assim ser pequeno o bastante para não caber os dados
    - ou grande demais e desperdiçar memória
  - Solicitar a memória necessária assim que a quantidade for conhecida

3

## Alocação Dinâmica de Memória

- Protótipo da função `malloc`:  
**void \* malloc(size\_t n);**  
*/\* retorna um ponteiro void para n bytes de memória não iniciados. Se não há memória disponível malloc retorna NULL \*/*

4

## Funções para Alocar e Liberar memória

- A função `malloc` é usada para alocar espaço para armazenarmos valores de qualquer tipo. Por este motivo, `malloc` retorna um ponteiro genérico, para um tipo qualquer, representado por `void*`, que pode ser convertido automaticamente pela linguagem para o tipo apropriado na atribuição.
- No entanto, é comum fazermos a conversão explicitamente, utilizando o operador de molde de tipo (cast).
- Então:  

```
v = (int *) malloc(10*sizeof(int));
```

5

## Funções para Alocar e Liberar memória

- Se não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo `NULL`, definido em `stdlib.h`).
- Podemos tratar o erro na alocação do programa simplesmente verificando o valor de retorno da função `malloc`
- Ex: imprimindo mensagem e abortando o programa com a função `exit`, também definida na `stdlib`.

```
v = (int*) malloc(10*sizeof(int));  
if (v == NULL) {  
    printf("Memoria insuficiente.\n");  
    exit(1); //aborta o programa e retorna 1 para o sist. operacional  
} ...
```

6

## Alocação Dinâmica de Memória

- Protótipo da função `calloc`:

```
void * calloc(size_t n, size_t size);
```

*/\* calloc retorna um ponteiro para um array com n elementos de tamanho size cada um ou NULL se não houver memória disponível. Os elementos são iniciados em zero \*/*

7

## Alocação Dinâmica de Memória

- O ponteiro retornado por tanto pela função `malloc` quanto pela `calloc` devem ser convertido para o tipo de ponteiro que invoca a função

```
int *pi = (int *) malloc(sizeof(int));
```

*/\* aloca espaço para um inteiro \*/*

```
int *ai = (int *) calloc(n, sizeof(int));
```

*/\* aloca espaço para um array de n inteiros \*/*

- toda memória não mais utilizada deve ser liberada através da função `free()`:

```
free(ai); /* libera todo o array */
```

```
free(pi); /* libera o inteiro alocado */
```

8

## Vetores e alocação dinâmica

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

```
#include <stdlib.h>
```

```
int main () {  
    int *v, n, i;  
    printf("Qual o tamanho do vetor que deseja:");  
    scanf("%d", &n);  
    v = (int *) calloc(n, sizeof(int)); //aloca um vetor de n posições inteiras  
    for (i = 0; i < n; i++) {  
        printf("Informe o %dº elemento: ", i+1);  
        scanf("%d", &v[i]); //armazena o valor no vetor na posição i  
    }  
    for (i = 0; i < n; i++)  
        printf("%d ", v[i]);  
    free(v); //libera a memória alocada para o vetor  
    return 0;  
}
```

9

## Alocação dinâmica de matrizes

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

```
#include <stdlib.h>
```

```
float **Alocar_matriz_real(int m, int n) {  
    float **v; //ponteiro para a matriz  
    int i; //variável auxiliar  
  
    if (m < 1 || n < 1) { //verifica parâmetros  
        printf("*** Erro: Parametro invalido **\n");  
        return NULL;  
    }  
}
```

10

## Alocação dinâmica de matrizes

```
v = (float **) calloc(m, sizeof(float *)); //aloca as linhas da matriz  
if (v == NULL) {  
    printf("*** Erro: Memoria Insuficiente ***");  
    return NULL;  
}  
for (i = 0; i < m; i++) { //aloca as colunas da matriz  
    v[i] = (float*) calloc(n, sizeof(float));  
    if (v[i] == NULL) {  
        printf("*** Erro: Memoria Insuficiente ***");  
        return NULL;  
    }  
}  
return v; //retorna o ponteiro para a matriz  
}
```

11

## Alocação dinâmica de matrizes

```
float ** liberar_matriz_real (int m, int n, float **v) {  
    int i; //variável auxiliar  
  
    if (v == NULL) return NULL;  
    if (m < 1 || n < 1) { //verifica parâmetros recebidos  
        printf("*** Erro: Parametro invalido **\n");  
        return v;  
    }  
    for (i = 0; i < m; i++) {  
        free (v[i]); //libera as linhas da matriz  
    }  
    free (v); //libera a matriz  
    return NULL; //retorna um ponteiro nulo  
}
```

12

## Alocação dinâmica de matrizes

```
void main () {
    float **mat; /* matriz a ser alocada */
    int l, c;     /* numero de linhas e colunas */
    ...
    /* outros comandos, inclusive inicialização de l e c */
    mat = Alocar_matriz_real (l, c);
    /* outros comandos utilizando mat[][] normalmente */
    if (Liberar_matriz_real (l, c, mat) != NULL) {
        printf("Erro ao desalocar a matriz!\n");
        exit(1);
    }
    ...
}
```

13

## Estruturas

14

## Estruturas

· Coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sobre um único nome

· As variáveis que fazem parte de uma estrutura são chamadas de **membros** da estrutura

Exemplo: registro da folha de pagamento de um funcionário

nome (*string*)

número do departamento (inteiro)

salário (float)

15

## Definindo um tipo estrutura

```
struct funcionario {
    char nome[60];
    int departamento;
    float salario;
};
```

16

## Declarando uma variável do tipo estrutura

· A definição da estrutura não declara nenhuma variável

· Para declarar uma variável do tipo estrutura **funcionario**, fazemos:

```
struct funcionario func;
```

· OU

```
struct funcionario {
    /* definição dos membros */
} func;
```

17

## Acessando Membros de uma Estrutura

· Para acessar um membro individual de uma estrutura usamos o operador ponto (.)

· Sintaxe:

***variável-estrutura.nome-do-membro***

· Exemplos:

```
func.departamento = 2;
```

```
printf("%.2f",func.salario);
```

```
fgets(func.nome,60,stdin);
```

18

## Inicializando Estruturas

A inicialização de estruturas é semelhante à inicialização de um vetor

Exemplo:

```
struct data {
    int dia;
    char mes[10];
    int ano;
};
```

```
struct data natal = { 25, "Dezembro", 2010 };
struct data nasc = { 14, "Março", 2000};
```

19

## Atribuição entre Estruturas

A informação contida em uma estrutura pode ser atribuída a outra estrutura do mesmo tipo usando uma simples expressão de atribuição, ou seja, você não precisa atribuir o valor de cada membro separadamente

20

## Exemplo de atribuição entre estruturas

```
#include <stdio.h>
int main() {
    struct {
        int a;
        int b;
    } x, y;
    x.a = 10;
    x.b = 20;
    y = x; // atribui uma estrutura a outra
    printf("Conteúdo de y: %d %d.\n", y.a, y.b);
    return 0;
}
```

Estrutura sem etiqueta

21

## Vetores de Estruturas

Para declarar um vetor de estruturas, você deve primeiro definir a estrutura, e depois declarar um vetor deste tipo

Exemplo:

```
struct funcionario func_array[100];
```

Cada elemento do vetor é uma estrutura do tipo **funcionario**

O nome **func\_array** não é o nome de uma estrutura e sim o nome de um vetor em que os elementos são estruturas

22

## Vetores de Estruturas

Para acessar uma estrutura específica dentro do vetor **func\_array**, use o índice no nome da variável do tipo vetor

Exemplo:

```
puts(func_array[2].nome);
```

23

## Estruturas Encaixadas

Podemos ter estruturas que contêm outras estruturas

Exemplo:

```
struct ponto {
    float x;
    float y;
};

struct circulo {
    struct ponto centro;
    float raio;
};
```

24

## Ponteiro para estruturas

■ Para declarar um ponteiro para a estrutura ponto fazemos:

■ `struct ponto * pponto;`

■ Para acessar os membros da estrutura `struct ponto` através do ponteiro `pponto` fazemos:

■ `(*pponto).x = 12.0;`

■ `pponto->x = 12.0;`

25

## Passando Estruturas para Funções

▫ Passando membros de estruturas para funções

▫ Você está passando uma variável simples

▫ Passando estruturas inteiras para funções

▫ Toda a estrutura é passada

▫ Passando estruturas por referência para uma função

▫ Ponteiro para estruturas

▫ Devolvendo uma estrutura de uma função

▫ A função deve ser do tipo estrutura

26

## Arquivos

27

## Introdução

▫ O armazenamento de dados em variáveis é temporário

▫ Os dados são perdidos quando o programa termina

▫ Os **arquivos** são utilizados para **conservação permanente dos dados**

▫ Os arquivos são armazenados em dispositivos de memória secundária, especialmente em discos

28

## Streams e Arquivos

■ O C provê um nível de abstração entre o programador e o dispositivo utilizado

■ Esta **abstração é chamada *stream* e o dispositivo real é chamado arquivo**

■ Uma *stream* é uma variável que permite ao programador enviar e receber dados de um dispositivo

29

## Arquivos

■ Tipo **FILE**

■ Definida em `stdio.h`

30

## Abrindo um arquivo

- A função **fopen()** abre um *stream* para uso e associa um arquivo a ele
- Devolve um ponteiro para o arquivo associado ou NULL
- Protótipo de **fopen()**  
**FILE \* fopen(const char \* nomearq, const char \* modo);**
  - *nomearq* é uma string com o nome do arquivo a ser aberto
  - *modo* é o modo de abertura do arquivo

31

## Modos de abertura do arquivo

- **r** abre um arquivo texto para leitura
- **w** cria um arquivo de texto para escrita. Se o arquivo existir, elimina seu conteúdo
- **a** anexa a um arquivo texto. Abre ou cria um arquivo para gravação no final do arquivo

32

## Abrindo um arquivo

- Para abrir um arquivo chamado teste, permitindo escrita, pode-se escrever:

```
FILE * fp;  
if((fp = fopen("teste.txt", "w")) == NULL)  
{  
    printf("Erro de abertura do arquivo");  
    return 1;  
}
```

Devolve NULL se  
houver um erro de  
abertura

33

## Fechando um arquivo

- A função **fclose()** fecha um *stream* que foi aberto por meio de uma chamada a **fopen()**
- Protótipo de **fclose()**  
**int fclose(FILE \* fp);**
  - Aonde *fp* é o ponteiro de arquivo devolvido por **fopen()**
  - Um valor de retorno igual a zero indica uma operação de fechamento bem-sucedida

34

## Usando feof()

- A função **feof()** indica quando o final de um arquivo binário foi de fato atingido

```
int feof(FILE * fp);
```

- Devolve verdadeiro se o final do arquivo foi atingido

35

## fprintf() e fscanf()

- Essas funções operam exatamente como **printf()** e **scanf()** exceto por operarem com arquivos
- Protótipos de **fprintf()** e **fscanf()**  
**int fprintf(FILE \* fp, const char \* controle,...);**
  - Devolve o número de caracteres escritos**int fscanf(FILE \* fp, const char \* controle,...);**
  - Devolve o número de caracteres lidos

36

## Trabalhando com strings: **fputs()** e **fgets()**

- Efetuam as operações de leitura e escrita de strings de e para um arquivo em disco
- Protótipo de **fputs()**  
`int fputs(const char * str, FILE * fp);`
- Protótipo de **fgets()**  
`char * fgets(const char * str, int tamanho, FILE * fp);`
- Devolve **NULL** se ocorrer um erro

37

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     int MAX=100, LEN=80, i, j, troca=1;
6     char texto[100][80];
7     FILE * fp;
8
9     printf("Digite uma linha vazia para sair.\n");
10    for(i = 0; i < MAX; i++) {
11        printf("%03d: ", i+1);
12        fgets(texto[i], LEN, stdin);
13        if(texto[i][0] == '\n') {
14            break;
15        }
16    }
17    // imprime todo o conteúdo da matriz em um arquivo
18    if((fp=fopen("texto.txt", "w"))==NULL) {
19        printf("erro na abertura do arquivo\n");
20    }
21    else{
22        for(j = 0; j < i; j++) {
23            fprintf(fp, "%03d: %s", j+1, texto[j]);
24        }
25        fclose(fp);
26    }
27    system("pause");
28    return 0;
29 }
```

- Exemplo:
  - Editor de textos simples
  - Arquivo resultante: texto.txt

38