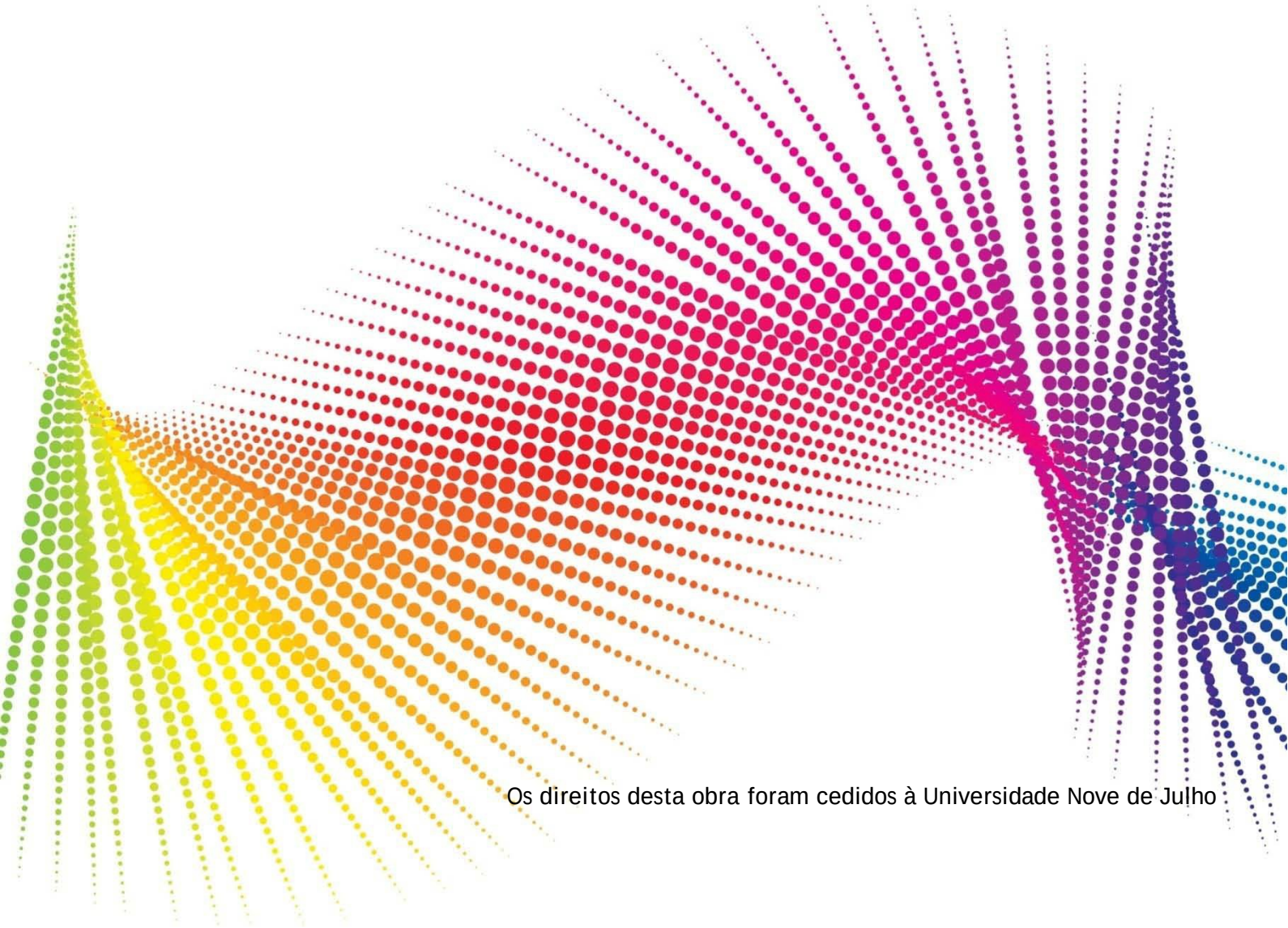
UNINOVE
Universidade Nove de Julho

Aula 19

Estrutura com Ponteiros



UNINOVE

1. Apontadores para struct

Como já estudamos, um ponteiro é uma variável que pode guardar um endereço de memória capaz de armazenar os dados do tipo a ele associado. Vejamos como funciona um apontador para uma struct:

```
struct pessoa {  char nome[20];
                 int idade;
              };

struct pessoa *app;
```

Como somente há uma variável declarada (app), para acessar os campos da struct, deve-se usar o operador seta ->:

```
app = malloc(sizeof(struct pessoa));

printf("\n\nDigite seu nome: ");
gets(app->nome);

printf("\nDigite sua idade: ");
scanf("%i",&app->idade);
```

Neste exemplo, foi declarado uma struct do tipo pessoa e um ponteiro para esta struct, app. Através da função malloc(), um endereço de memória foi alocado dinamicamente e os dados fornecidos pelo usuário foram armazenados na struct, através

do seu ponteiro, usando--se o operador -> para acessar seus campos.

No modelo tradicional de programação, sem o uso de ponteiros e alocação dinâmica, o trecho de programa acima seria assim implementado:

```
struct pessoa {  char nome[20];
                 int idade;
              };
struct pessoa pp;

…

   printf("\n\nDigite seu nome: ");
   gets(pp->nome);

   printf("\nDigite sua idade: ");
   scanf("%i",&pp->idade);
```

Compare as duas soluções. A primeira utiliza uma solução com alocação dinâmica e a segunda com alocação sequencial.

2. Uso de `typedef`

Como já vimos, pode-se usar typedef para se definir um novo tipo de dados. Uma vez definido este novo tipo, outras variáveis deste novo tipo podem ser declaradas. O operador sizeof pode ser usado para se saber quantos bytes este novo tipo possui.

```
typedef struct {  char nome[20];
                  int idade;
               } pessoa;
pessoa pp, *app;
```

```
typedef struct {  char nome[20];
                  char fone[15];
                  long int cpf;
                  char rg[10];
                } ficha;
ficha ficha1, ficha2, fichario[100];
```

```
typedef struct nodo { char info[20];
                      struct nodo *esq, *dir;
                    } no;
no  no1, no2, *apno1, *apno2;
```

3. **Mais explicações sobre ponteiros para estruturas**

Os ponteiros guardam endereços de memória de um certo tipo de dados, o qual pode ser uma estrutura. As estruturas são capazes de armazenar diversas variáveis de tipos básicos diferentes.

```
struct  f {  char  nome[20];
             int   idade;
          } ficha;

struct f *apficha;

int tamanho;
      . . .
```

```
strcpy(ficha.nome,”Luiza”);
ficha.idade = 20;

tamanho = 20*sizeof(char) + sizeof(int);
apficha = malloc (tamanho);
strcpy(apficha->nome, ficha.nome);
apficha->idade = ficha.idade;
```

Neste caso há uma variável **ficha** e um ponteiro **apficha** declarados. Para usar a variável **ficha** é só atribuir os valores campo a campo, mas para usar o ponteiro **apficha** é necessário inicializá-lo primeiro. Isso é feito por meio da função **malloc**, que aloca espaço na memória dinamicamente, e necessita saber a quantidade de bytes do tipo de dados associado ao ponteiro. Assim, foi feito o cálculo dessa quantidade usando a variável **tamanho**.

A função **sizeof** devolve a quantidade de bytes ocupados por um tipo. No entanto, para que a função sizeof seja capaz de “contar” a quantidade de memória exata capaz de armazenar a estrutura é mais fácil criar o tipo estrutura através de **typedef**. Veja a diferença:

```
typedef struct f {  char  nome[20];
                    int   idade;
                  } tipoficha;

tipoficha ficha, *apficha;
. . .

strcpy(ficha.nome,”Luiza”);
ficha.idade = 20;

apficha = malloc (sizeof(tipoficha));
strcpy(apficha->nome, ficha.nome);
apficha->idade = ficha.idade;
```

4. Exemplos de programas C usando ponteiro para struct:

```
/*Ponteiros7.c: ilustra o uso de ponteiros para structs */
#include <stdio.h>
#include <conio.h>

struct pessoa {  char nome[20];
                 int idade;
              };

struct pessoa *app;

int main()
{  printf ("\nIlustra o uso de ponteiros para structs");

   app = malloc(sizeof(struct pessoa));

   printf("\n\nDigite seu nome: ");
   gets(app->nome);

   printf("\nDigite sua idade: ");
   scanf("%i",&app->idade);

   printf("\nDados fornecidos nome: %s idade:%i \n\n", app->nome, app-
>idade);

   printf ("\nFim do programa");
   getch();
   return 0;
}
```

```
/*typedef.c: ilustra o uso de typedef */
```

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

typedef struct  {  char nome[20];
                      int idade;
               } pessoa;

pessoa *app;

int main()
{  printf ("\nIlustra o uso de ponteiros para structs");

   app = malloc(sizeof(pessoa));

   printf("\n\nDigite seu nome: ");
   gets(app->nome);

   printf("\nDigite sua idade: ");
   scanf("%i",&app->idade);

   printf("\nDados fornecidos nome: %s idade:%i \n\n", app->nome,
app->idade);

   printf ("\nFim do programa");
   getch();
   return 0;

}
```

5. Tabela com os principais operadores de ponteiros

Esta tabela resume as principais operações que podem ser realizadas com ponteiros em C.

Operador	O que faz	Exemplos

*	Declara um ponteiro associado a um tipo	`struct pessoa { char nome[20]; int idade; };`
		
		`struct pessoa *app;`
		
		`int *apA, *apB;`
		`float *apx, *apy;`

&	Obtém o endereço de memória	`struct pessoa { char nome[20]; int idade; };`
		
		`struct pessoa *app, pp;`
		
		`int *apA, *apB, A, B;`
		
		`apA = &A;`
		`apB = &B`
		`app = &pp;`

*<ponteiro>	Apresenta o conteúdo do endereço do <ponteiro>	`int *apA, *apB, *apS, A=2, B=5;`
		
		`apA = &A;`
		`apB = &B;`

		`*apS = *apA + *apB;` `printf("\nResultado de A+B: %i",*apS);`

malloc()	Aloca dinamicamente um endereço de memória para o <ponteiro> ou um espaço de memória em bytes	`struct pessoa {  char nome[20];`
		`int idade;`
		`};`
		`struct pessoa *app;`
		`int *apA, *apB;`
		`float *apx;`
		`app = malloc(sizeof(struct pessoa));`
		`apA = malloc (sizeof(int));`
		`apx = malloc (sizeof(float));`
		`apB = **NULL**; //endereço nulo`

free();	Libera/dealoca o endereço de memória do <ponteiro>	`free(apB);` `free(app);`