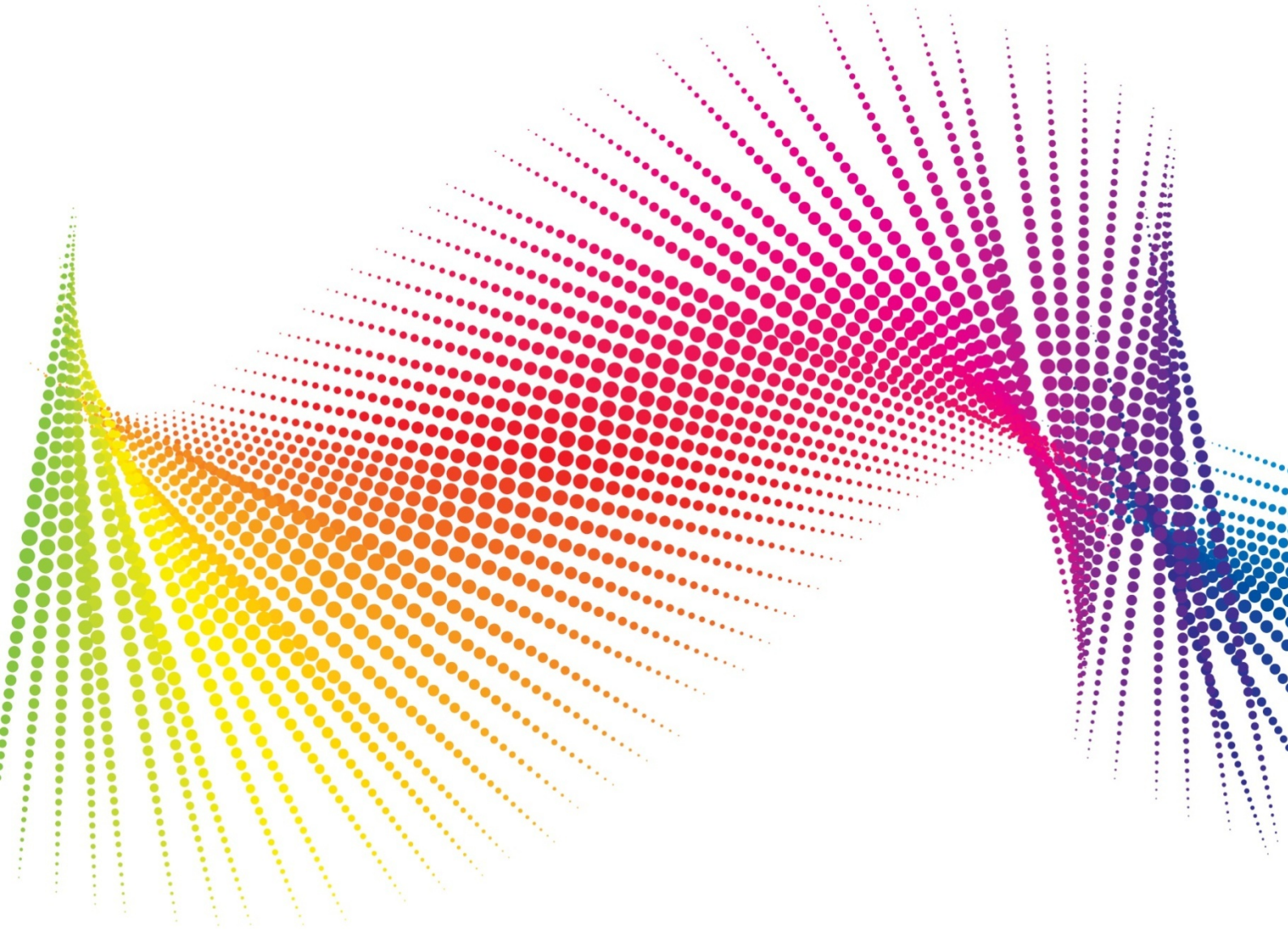
UNINOVE
Universidade Nove de Julho

Estrutura de Dados

Aula 06



Este material é parte integrante da disciplina oferecida pela UNINOVE.
O acesso às atividades, conteúdos multimídia e interativo, encontros virtuais, fóruns de discussão e a comunicação com o professor devem ser feitos diretamente no ambiente virtual de aprendizagem UNINOVE.

Uso consciente do papel.
Cause boa impressão, imprima menos.

Aula 6: Ponteiros

Objetivo: Nesta aula, continuamos a estudar os ponteiros, utilizando exemplos práticos.

Função malloc alocação dinâmica

Vimos que podemos usar o endereço de uma variável para inicializar um ponteiro, mas também podemos fazer isso usando a função **malloc**.

A função `malloc()` é usada para "pedir" ao sistema operacional um endereço livre ou um conjunto de endereços livres de memória.

O seu protótipo é:

```
void *malloc(int numero_de_bytes);
```

A função retorna um apontador genérico (`void *`) para o início da memória alocada, que deverá conter espaço suficiente para armazenar (`numero_de_bytes`) bytes. Se não for possível alocar a quantidade de memória solicitada, a função retorna o apontador `NULL`.

Então, se desejarmos alocar espaço para 100 inteiros, podemos usar:

```
int *pti;
pti = (int *) malloc(100*sizeof(int));
```

Note o uso de um *cast* (`int *`) na saída da função e também o uso do operador `sizeof` para determinar, em bytes, o tamanho de um inteiro. O *cast* para o tipo do apontador é opcional, mas é importante para assegurar que a aritmética com os apontadores seja efetuada de modo correto.

O uso do operador `sizeof` é indispensável, mesmo que se saiba o tamanho do tipo de dados, para tornar o código independente da máquina em que for compilado, ou seja, aumentar sua portabilidade. Este operador é utilizado para

determinar o tamanho (em bytes) de qualquer tipo ou variável, mas podemos, é claro, alocar espaço por um único inteiro, como em:

```
pti = malloc(sizeof(int);
```

Alocação e exclusão de alocação de endereços de memória

Para atribuir um valor a um ponteiro recém-criado é necessário ter um endereço da memória principal (RAM). Vimos que este endereço pode ser obtido do endereço de uma variável (não ponteiro) declarada ou ser solicitado um novo endereço de memória através da função `malloc()`, que aloca dinamicamente (em tempo de execução) um endereço de memória capaz de armazenar o conteúdo do tipo para o qual o ponteiro está associado. Para que este espaço na memória seja liberado (dealocado) após seu uso, existe a função `free()`. Além disso, caso se queira atribuir um valor nulo a um ponteiro, usa-se a constante `NULL`.

Exemplo de programa que aloca dinamicamente endereços de memória

```
//Ponteiros4.c: ilustra o uso de alocação e dealocação de ponteiros

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

int     A,B;
int    *apS,*apA,*apB;

int main()
{
printf("\nDigite um numero inteiro A: ");
scanf("%i", &A);
printf("\nDigite um numero inteiro B: ");
scanf("%i",&B);

apS = malloc (sizeof(int));
apA = &A;
apB = &B;

*apS = *apA + *apB;  //isto seria equivalente a fazer *apS = A + B;

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

```
free(apS);

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

Operações com ponteiros

Depois que aprendemos a usar os dois operadores **&** (endereço de memória de) e ***** (conteúdo do endereço de memória), fica fácil entender operações com ponteiros.

Atribuição

A primeira operação, e mais simples, é igualar dois ponteiros. Se existem dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1 = p2**. Repare que este comando faz com que **p1** aponte para o mesmo endereço de memória que **p2** aponta.

Se desejarmos que a variável apontada por **p1** tenha o mesmo conteúdo da variável apontada por **p2**, devemos fazer ***p1 = *p2**.

As únicas operações aritméticas possíveis entre ponteiros e inteiros são a subtração e a adição. Isso pode ser feito com os operadores unários **++** e **–** e com os operadores binários **+** e **-**.

Incremento e Decremento

Quando incrementamos um ponteiro, ele passa a apontar para o próximo endereço de memória a partir do atual. Isto é, se existe um ponteiro para um inteiro e este é incrementado, ele passa a apontar para o próximo endereço de memória, avançando o número de bytes necessários para a alocação de um tipo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo do ponteiro.

- Se incrementamos um ponteiro **char***, ele avança 1 byte na memória.
- Se incrementamos um ponteiro **int***, ele avança 2 bytes na memória.
- Se incrementamos um ponteiro **double***, ele avança 8 bytes na memória.

Estamos considerando que o tipo **char*** ocupa 1 byte, que o tipo **int*** ocupa 2 bytes e que o tipo **double*** ocupa 8 bytes.

O decremento funciona de forma semelhante.

Supondo que **p** é um ponteiro, as operações de incremento e decremento são escritas como:

```
p++;
p--;
```

É importante ressaltar que estas operações são sobre ponteiros (ou endereços de memória) e não operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, fazemos:

```
(*p)++;
```

Soma e Subtração

Outras operações aritméticas úteis são a soma e a subtração de inteiros com ponteiros. Suponha que se queira incrementar um ponteiro **p** em 15. Basta fazer:

`p=p+15;` ou `p+=15;`

O resultado desta operação é que **p** vai avançar 15 posições de memória.

E se desejamos usar o conteúdo do ponteiro **p** 15 posições adiante, fazemos:

```
*(p+15);
```

A subtração funciona da mesma maneira.

`p=p-5;` ou `p+=5;`

Neste caso, desejamos usar o conteúdo do ponteiro **p** há 5 posições anteriores.

Apresentação de endereços de memória

Já sabemos que para apresentarmos um endereço de memória (ou um ponteiro) usamos o formato %p.

Você também já deve ter notado que os endereços de memória são apresentados numa mistura de números e letras, não é? Isso porque estes endereços estão na base 16 ou hexadecimal.

Agora acesse o AVA e assista à animação para conhecer mais sobre os endereços de memória em hexadecimal. Ela faz parte da sequência desta aula e portanto é essencial para a sua aprendizagem.

Depois, resolva os exercícios e verifique seu conhecimento. Caso fique alguma dúvida, leve a questão ao Fórum e divida com seus colegas e professor.

REFERÊNCIAS

UNIVERSIDADE FEDERAL DE MINAS GERAIS. *Curso de C:* como funcionam os ponteiros. 2005. Disponível em: <http://www.ead.cpdee.ufmg.br/cursos/C/aulas/c610.html>. Acesso em: 01 mai. 2012.

SCHILDT, H. *C Completo e total*. 3. ed. São Paulo: Makron Books, 1997.