Introdução a Ponteiros

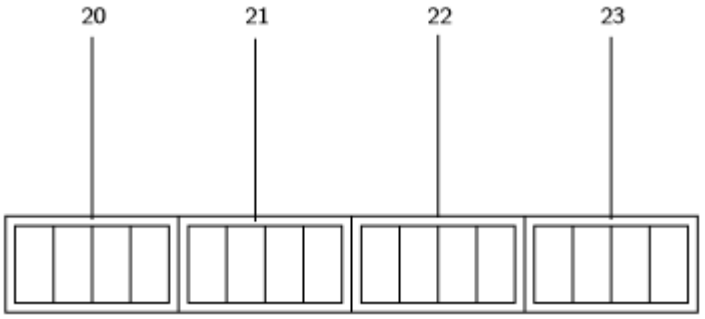
Prof. Lilian Berton

São José dos Campos, 2019

Declaração de variáveis

- Quando declaramos uma variável, reservamos um espaço na memória para ela. **Toda variável é associada a um endereço**.
- Dentro de uma variável podemos guardar informações dependendo do seu tipo.

- int n aloca 4 bytes na memória associados aos endereços 20, 21, 22 e 23 por exemplo.



Declaração de variáveis do tipo ponteiro

- Para declarar uma variável do tipo ponteiro usamos o * na frente. As variáveis ap1, ap2 e ap3 são ponteiros (guardam endereços de memória) para int, float e char, respectivamente.

- **int * ap1;**
- **float * ap2;**
- **char * ap3;**

Declaração de variáveis do tipo ponteiro

- Em C, o comando & **faz com que a variável apt guarde o endereço de n.**
- int n = 0, *apt;
- **apt =&n;**

- O operador & antes do nome da variável significa “endereço de”. Assim, o comando de atribuição acima coloca em apt o “endereço de” n.

Memória RAM

	Endereço	Conteúdo
n ⇨	15B10	0
apt ⇨	15B11	15B10
	15B12	
	...	

Uso do operador “endereço &”

- Para uma variável ponteiro receber o endereço de uma outra variável é necessário usar o operador **& (“endereço de”)**. Observe os seguintes exemplos:

- ap1 = &n; /* ap1 recebe o endereço da variável n */
- ap2 = &y; /* ap2 recebe o endereço da variável y */
- ap3 = &x; /* ap3 recebe o endereço da variável x */

- Variáveis que guardam endereços são chamadas de variáveis do tipo ponteiro, pois é como se ap1, ap2 e ap3 estivessem apontando para outras variáveis.

Uso do operador “vai para”

- Uma vez que o ponteiro está apontando para uma variável (ou seja, guardando seu endereço) é possível ter acesso a essa variável usando a variável ponteiro usando o operador ∗ **(“vai para”)**:

- ∗ ap1 = 10; /∗ vai para a gaveta que o ap1 está apontando e guarde 10 nesta gaveta∗/
- ∗ ap2 = −3.0; /∗ vai para a gaveta que o ap2 está apontando e guarde−3.0 nesta gaveta ∗/
- ∗ ap3 = −2.0; /∗ vai para a gaveta que o ap3 está apontando e guarde−2.0 nesta gaveta ∗/

- Quando uma variável ponteiro recebe um valor, esse valor vai para a variável que ele aponta e fica armazenado nesse espaço de memória.

Exemplo 1

- Em C, o comando & **faz com que a variável apt guarde o endereço de n.**
- int n = 0, *apt;
- **apt =&n;**
- ***apt = 1;**

- O operador & antes do nome da variável significa “endereço de”. Assim, o comando de atribuição acima coloca em apt o “endereço de” n.

Memória RAM

	Endereço	Conteúdo
n ⇨	15B10	0
apt ⇨	15B11	15B10
	15B12	
	...	

Memória RAM

	Endereço	Conteúdo
n ⇨	15B10	1
apt ⇨	15B11	15B10
	15B12	
	...	

Exemplo 2

```
int main () {
  int n, m;
  float y;
  char x;
```

```
int * ap1;
float * ap2;
char * ap3;
```

Declaração das variáveis ponteiros. Note o uso do asterisco para declarar ponteiros.

```
  n = 2; m = 3;
  y = 5.0; x = 's';
```

```
ap1 = &n;
ap2 = &y;
ap3 = &x;
```

Uso de Ponteiros: & significa "endereço de". Note que aqui não tem o asterisco antes da variável ponteiro.

```
* ap1 = m;
* ap2 = –5.0;
* ap3 = 'd';
```

Uso de Ponteiros: Note que aqui usa-se o asterisco antes do ponteiro. Este asterisco significa "vai para"a gaveta que o ponteiro está indicando (apontando).

```
  printf ("n = %d y = %f x = %c\n", n, y, x);
  return 0;
}
```

Quais seriam os valores de n, y e x ?

Exemplo 3

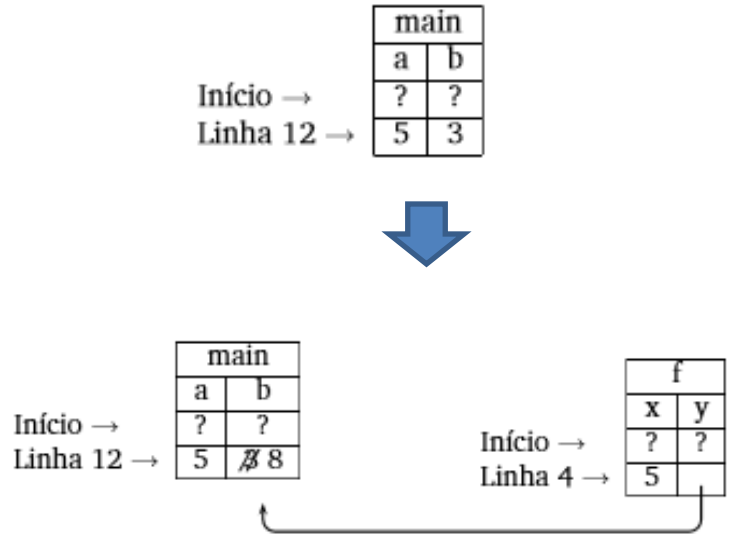
```c
#include<stdio.h>

int main() {
    int a, b;
    int *pa, *pb;

    a = 10;
    b = 11;
    pa = &a;
    pb = &b;

    //imprimindo os valores de a e b:
    printf ("%d %d\n", a, b);
    //imprimindo os endereços de a e b:
    printf ("%X %X\n",&a, &b);
    //imprimindo o conteudo de pa e pb que são os endereços de a e b
    printf ("%X %X\n",pa, pb);

    *pa = 12;
    *pb = 12;
    //imprimindo os valores de a e b:
    printf ("%d %d\n", a, b);

}
```

Resultado da impressão:

```
10 11
62FE3C 62FE38
62FE3C 62FE38
12 12
```

Variáveis como parâmetro de funções

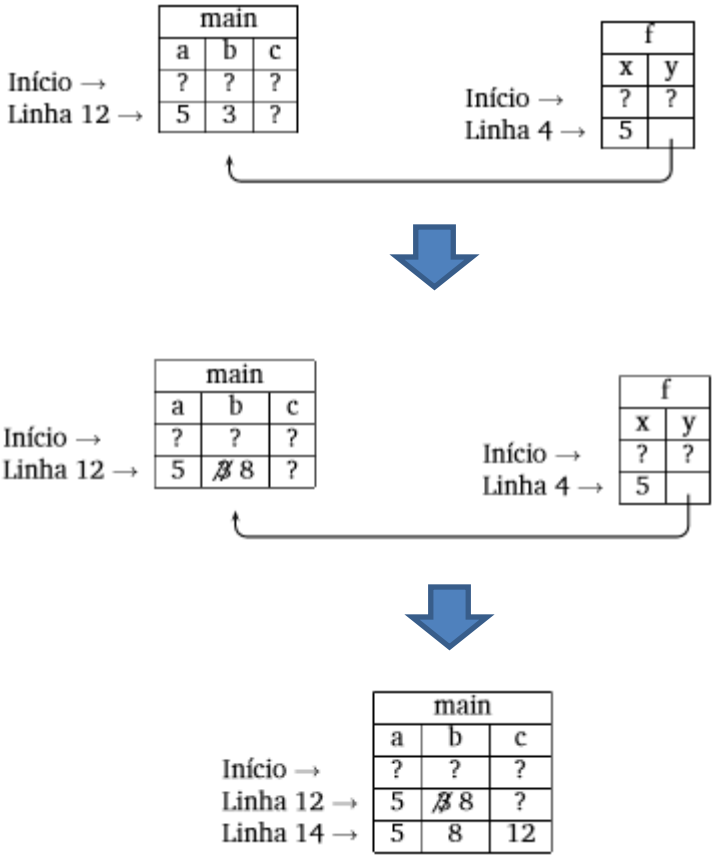
```
1   # include <stdio.h>
2
3
4   void g (int x, int *y) {
5     *y = x + 3;
6   }
7
8   int main () {
9     int a, b;
10
11
12    a = 5; b = 3;
13
14    g (a, &b);
15
16    printf ("a = %d, b = %d\n", a, b);
17
18    return 0;
19  }
```



A chamada da função g(a,&b) envia uma **cópia do valor de a** para a função g e **o endereço de memória de b.**

Retornando mais de um valor para a main

```
1  # include <stdio.h>
2
3
4  int f (int x, int *y) {
5    *y = x + 3;
6    return *y + 4;
7  }
8
9  int main () {
10   int a, b, c;
11
12   a = 5; b = 3;
13
14   c = f (a, &b);
15
16   printf ("a = %d, b = %d, c = %d\n", a, b, c);
17
18   return 0;
19 }
```



Vetores como Parâmetro de Funções

- **O nome de um vetor dentro de parâmetro de uma função é utilizado como sendo um ponteiro para o primeiro elemento do vetor na hora de utilizar a função.**

```
float ProdutoEscalar (float u[], float v[], int n) {
  int i;
  float res = 0;
  for (i=0; i<n; i++)
  res = res + u[i] * v[i];
  return res;
}
```

Vetores como Parâmetro de Funções

- Na Linha 19, a chamada da função f faz com que o ponteiro u receba &v[0], ou seja, faz com que o ponteiro u aponte para v[0].
- Na Linha 8, temos o comando u[i]=4. Como u está apontando para v, então o comando u[i]=4 é o mesmo que fazer v[i]=4.

```
1   # define MAX 200
2
3
4   float f (float u[]) {
5     float s;
6     /* declaração da função f */
7     ...
8     u[i] = 4;
9     ...
10    return s;
11  }
12
13  int main () {
14    float a, v[MAX]; /* declaração da variável a e vetor v */
15    ...
16    /* outras coisas do programa */
17
18
19    a = f (v);    /* observe que o vetor é passado apenas pelo nome */
20
21    ...
22
23    return 0;
24  }
```

u aponta para v[0].

Na Linha 8, dentro da função f, estamos mudando o conteúdo da casa de indice i do vetor v que foi passada por parâmetro para a função .

Matrizes como Parâmetro de Funções

- **O nome de uma matriz dentro de parâmetro de uma função é utilizado como sendo um ponteiro para o primeiro elemento da matriz que está sendo usada na hora de chamar a função.**

```
# include <math.h>
float soma_diagonal (float B[300][300], int n) {
  int i;
  float r = 0;
  for (i=0; i<n; i++) {
    r = r + B[i][i];
  }
  return r;
}
```

Matrizes como Parâmetro de Funções

- Na Linha 19, a chamada da função f faz com que o ponteiro M receba &A[0][0], ou seja, faz com que o ponteiro M aponte para A[0][0].
- Na Linha 8, temos o comando M[i][j]=4. Como M está apontando para A, então o comando M[i][j]=4 é o mesmo que fazer A[i][j]=4.

M aponta para A[0][0].

```
1   # define MAX 200
2
3
4   float f (float M[MAX][MAX]) {
5     float s;
6     /* declaração da função f */
7     ...
8     M[i][j] = 4;
9     ...
10    return s;
11  }
12
13  int main () {
14    float a, A[MAX][MAX]; /* declaração da variável a e da matriz A */
15    ...
16    /* outras coisas do programa */
17
18
19    a = f (A);    /* observe que a matriz é passada apenas pelo nome */
20
21    ...
22
23    return 0;
24  }
```

Nessa Linha 8, dentro da função f, estamos mudando o conteúdo da casa de linha i e coluna j da matriz A que foi passada por parâmetro para a função .

Alocação dinâmica de memória

- ***Usamos a biblioteca stlib.h***
- ***malloc*** (*memory allocation*) aloca um bloco de bytes consecutivos na memória RAM e devolve o endereço desse bloco.
- Para alocar um objeto que ocupa mais de um byte usamos o operador *sizeof*, que diz a quantidade de bytes desejado.

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int *v;
    int n;
    scanf("%d", &n);
    v = (int*) malloc (n * sizeof(int));
    for (int i = 0; i < n; i++)
        scanf("%d", &v[i]);
    for (int i = 0; i < n; i++)
        printf("%d ", v[i]);
}
```

Alocação dinâmica de memória

- As vezes pode ser necessário alterar, durante a execução do programa, o tamanho do bloco de bytes alocado por ***malloc***. Nesse caso podemos recorrer a função ***realloc*** para redimensionar o bloco de bytes (aumentar o diminuir).

- A função ***realloc*** recebe o endereço de um bloco alocado por *malloc* e o número de bytes que o bloco deve ter. A função aloca o novo bloco, transfere o conteúdo do bloco original e devolve o endereço do novo bloco.

```
#include<stdio.h>
#include<stdlib.h>
int main() {
	int *v;
	int n;
	scanf("%d", &n);
	v = (int*) malloc (n * sizeof(int));
	for (int i = 0; i < n; i++)
		scanf("%d", &v[i]);

	v = (int*)realloc (v, 10 * sizeof (int));
	for (int i = 5; i < 10; i++)
		scanf("%d", &v[i]);
	for (int i = 0; i < 10; i++)
		printf("%d ", v[i]);
	}
```

A função *free*

- As variáveis alocadas estaticamente dentro de uma função (variáveis locais) desaparecem assim que a execução da função termina.

- As variáveis alocadas dinamicamente continuam a existir mesmo depois que a execução da função termina.

- Para liberar a memória ocupada por essas variáveis é necessário usar a função ***free***. A qual desaloca a porção de memória alocada por *malloc.*

- A instrução *free*(ptr) avisa ao sistema que o bloco de bytes apontado por ptr está livre e disponível para reciclagem. A próxima chamada de malloc poderá tornar posse desses bytes.

Porque ponteiros são úteis?

- Ponteiros são usados em situações em que é necessário **conhecer o endereço onde está armazenada a variável** e não o seu conteúdo.

- Os ponteiros como parâmetros de função servem para modificar as variáveis que estão fora da função.

- Uma função só consegue devolver um único valor via return. Como então a função vai devolver dois valores? Resposta: usando ponteiros!

- Com ponteiros é possível alocar as posições de memória necessárias para armazenamento de vetores/matrizes somente quando o programa estiver rodando -> alocação dinâmica (conteúdo de AED1).

Exercício

1. Faça um programa que, dado um inteiro n e uma matriz quadrada de ordem n, cujos elementos são todos inteiros positivos, imprime uma tabela onde os elementos são listados em ordem decrescente, acompanhados da indicação de linha e coluna a que pertencem. Havendo repetições de elementos na matriz, a ordem é irrelevante. Uma estratégia para solucionar este problema é usar a **função maior** que devolve o maior elemento da matriz e sua respectiva posição e colocar nesta posição um inteiro negativo, digamos −1.

Exemplo: No caso da matriz $A = \begin{pmatrix} 3 & 7 & 1 \\ 1 & 2 & 8 \\ 5 & 3 & 4 \end{pmatrix}$, a saída poderia ser:

Elem	Linha	Coluna
8	1	2
7	0	1
5	2	0
4	2	2
3	0	0
3	2	1
2	1	1
1	0	2
1	1	0

Exercícios

2. Escreva um programa usando função e ponteiros que recebe vários números inteiros até que o usuário insira o caracter ‘0’ e devolve o seu primeiro dígito, seu último dígito e altera o valor de n removendo seu primeiro e último dígitos. Exemplos:

valor inicial de n	primeiro dígito	último dígito	valor final de n
732	7	2	3
14738	1	8	473
1010	1	0	1
78	7	8	0
7	7	7	0

```c
#include <stdio.h>

int main()
{
    int a;

    scanf("%i", &a);

    printf("%i", a);

    return 0;
}
```

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

int main()
{
    int a, *p;

    p = &a;

    scanf("%i", p);

    printf("%i", a);

    return 0;
}
```