Linguagem de Programação I

- Ponteiros – Parte 2 -

Prof. Ulysses Santos Sousa

ulyssessousa@ifma.edu.br

Aula 09

Roteiro

- Ponteiros genéricos
- Ponteiros para estruturas
- Indireção múltipla
- Alocação estática de memória
- Alocação dinâmica de memória
- Strings como ponteiros

Ponteiros genéricos (void)

- **Ponteiros void**
  - São ponteiros de propósito geral que pode apontar para qualquer tipo de dado.
  - São utilizados em situações em que seja necessário que uma função receba ou retorne um ponteiro genérico e opere independente do tipo de dado apontado.

Ponteiros genéricos (void)

- **Declaração:**

```
void *ptr;
```

  - Observações:
    - o conceito de ponteiros void não tem nada a ver com o tipo void para funções.
    - O conteúdo da variável apontada por um ponteiro void não pode ser acessado por meio desse ponteiro.
    - É necessário criar outro ponteiro e fazer a conversão de tipo na atribuição.

Ponteiros genéricos

```
#include <stdio.h>

int main(){
   int i = 5, *pi;
   float f = 3.2, *pf;
   void *pv;

   pv = &i;
   pi = (int *)pv;   //o casting é obrigatório
   printf("%d\n", *pi);

   pv = &f;
   pf = (float *)pv;
   printf("%.2f\n", *pf);

   system("pause");
   return 0;
}
```

Ponteiros para estruturas

- **Declaração:**

```
struct Aluno *al;
```

  - Se a variável **al** armazenar o endereço de uma estrutura, podemos acessar seus membros indiretamente.

Quando utilizamos ponteiros para estrutura não podemos utilizar o operador . (ponto) para acessar os membros da estrutura.

al.nome //ERRO

Ponteiros para estruturas

- **Formas de acessar os membros de uma estrutura com ponteiros:**

```
(*al).matricula;
```

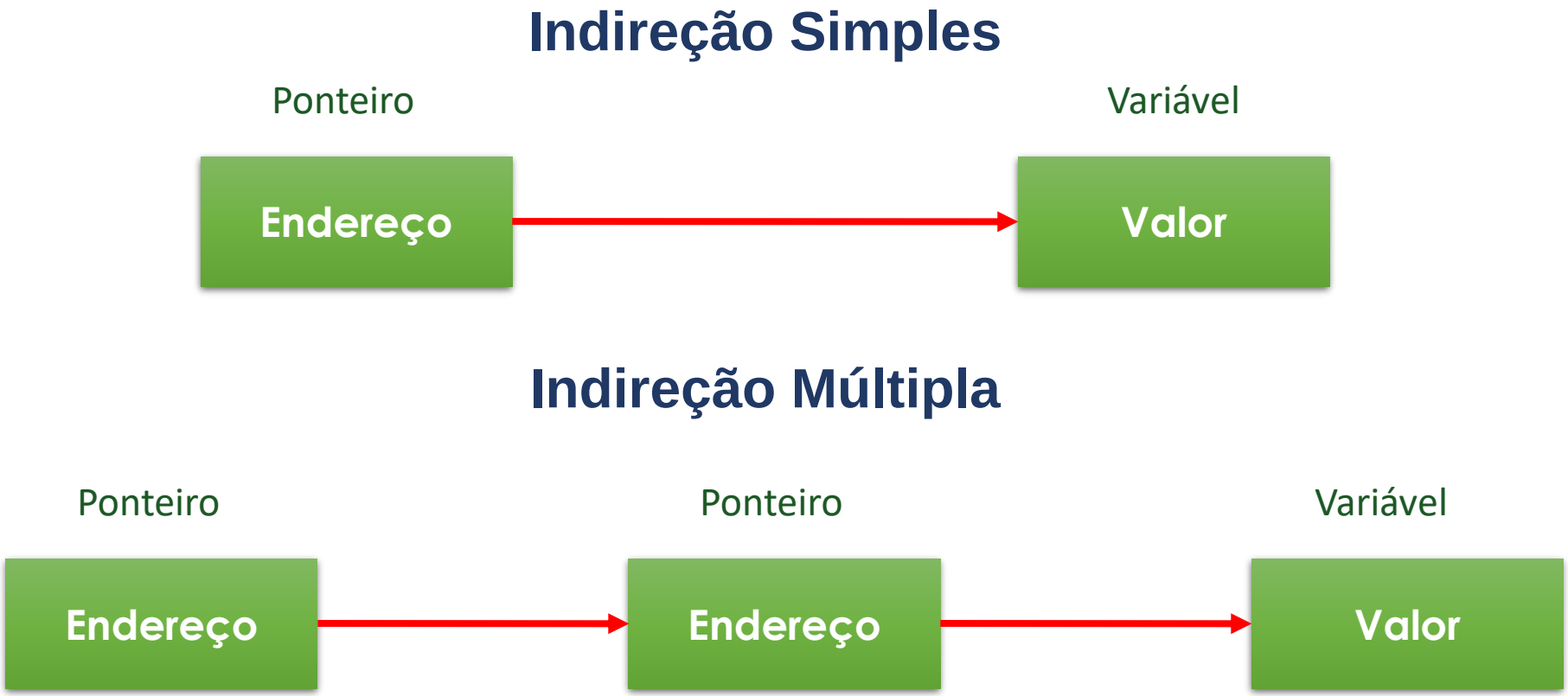
Essa é uma expressão de visualização complexa. Os parênteses são necessários, pois o operador (.) tem precedência sobre o operador (*).

```
al -> matricula;
```

Aqui, temos o uso do operador de acesso a membros (->), que opera sobre o endereço de uma variável estrutura.

Indireção Múltipla (ponteiro para ponteiro)

- Você pode ter um ponteiro apontando para outro ponteiro que aponta para o valor final.



Indireção Múltipla (ponteiro para ponteiro)

- **Observações:**
  - Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. Isso é feito colocando-se um * adicional na frente do nome da variável.
  - Para acessar o valor final apontado indiretamente por um ponteiro a um ponteiro, devemos utilizar o operador * duas vezes.

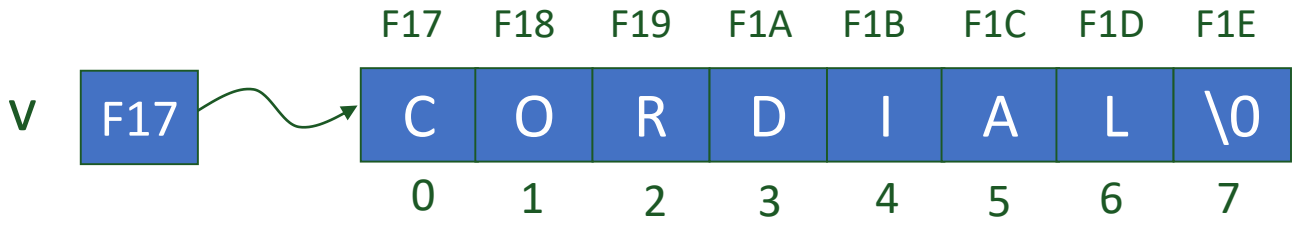
Indireção Múltipla (ponteiro para ponteiro)

```
#include <stdio.h>

int main(){
   int x = 10, *p, **q;

   p = &x;
   q = &p;

   printf("%d", **q);

   system("pause");
   return 0;
}
```

Alocação estática de memória

- Como sabemos, o nome de um vetor é um ponteiro para sua primeira posição.
- Exemplo:

```
char v[8];
```



- Portanto, uma forma alternativa de declarar o vetor é:

```
char *v;
```

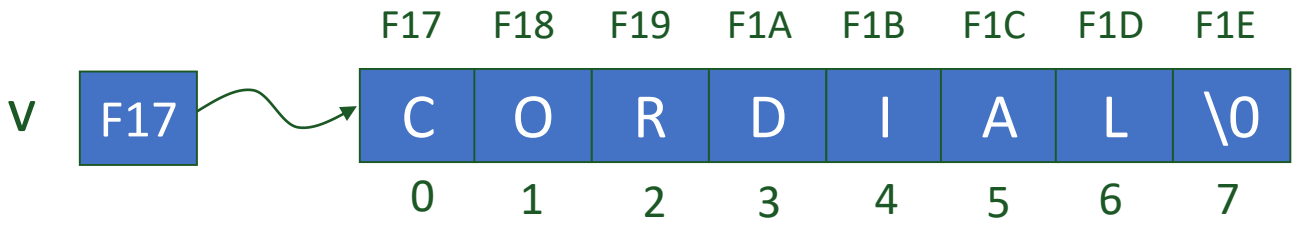
Alocação estática de memória

- **Qual a diferença entre de declarar um ponteiro e um vetor?**

```
char *v;
```

ou

```
char v[8];
```



A diferença está na alocação de memória.

Alocação estática de memória

- **Na alocação estática o compilador aloca automaticamente o espaço de memória necessário.**
- **Por exemplo:**

```
char v[8];
```

  - O compilador aloca *8 * N bytes* de memória para *v*, onde *N* corresponde ao número de bytes usado pelo compilador para armazenar o tipo ***char***.
  - Normalmente, ***N = 1 byte***, para o tipo ***char***.

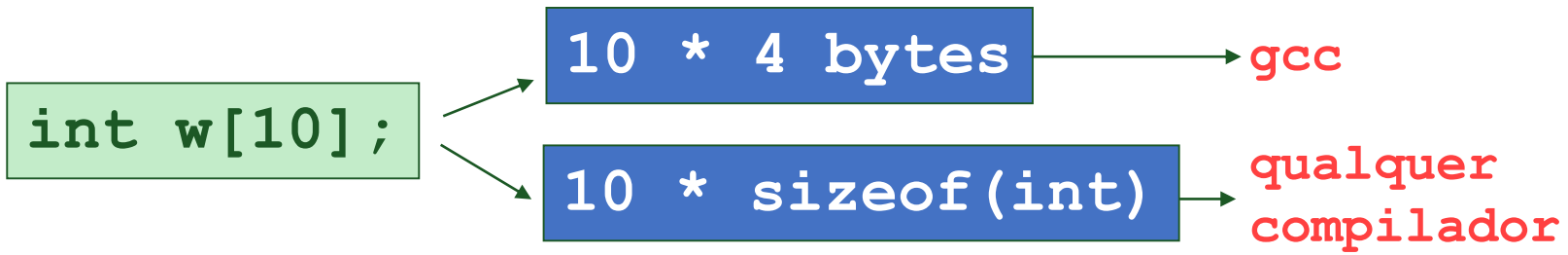
Alocação estática de memória

- Para saber o número de bytes reservados para um determinado tipo utiliza-se a função *sizeof*.

- Por exemplo, no gcc:

```
a = sizeof(int);      → a = 4
b = sizeof(float);    → b = 4
c = sizeof(double);   → c = 8
```

- Ao calcular o espaço de memória de um vetor, use a função *sizeof*, pois o espaço de memória de um determinado tipo, pode variar entre compiladores.



Alocação dinâmica de memória

- **Alocação dinâmica**
  - É o processo que aloca memória em tempo de execução.
  - Utiliza-se quando não se conhece a quantidade de memória que será necessária para o armazenamento de informações.
  - Evita o desperdício de memória.

Alocação dinâmica de memória

- **Funções calloc() e malloc():**
  - calloc()
    - Requer dois parâmetros: o número de posições de memória e o tamanho em bytes de cada posição.
  - malloc()
    - Requer apenas um parâmetro: o espaço total em bytes de memória necessário.

Alocação dinâmica de memória

- **Funções calloc() e malloc():**
  - Essas funções retornam um ponteiro do tipo ***void*** para o início do espaço de memória alocada.
  - Portanto, este ponteiro deve ser convertido (type casting) para o tipo de dado desejado.

Alocação dinâmica de memória

- **Funções calloc() e malloc():**
  - Exemplo: para fazer com que *w* aponte para um espaço de memória capaz de acomodar 10 elementos do tipo ***int***, podemos escrever:

```
int *w;
w = (int *)calloc(10,sizeof(int));
```

  - Ou então:

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

Alocação dinâmica de memória

- **Qual a vantagem de declarar um vetor como ponteiro?**
  - Não é necessário, a priori, definir o número de posições de memória necessárias.

```
c = (float *)calloc(tam_vetor,sizeof(float));
```

Alocação dinâmica de memória

- **Reduzir ou aumentar a quantidade de memória alocada**
  - Isto pode ser feito através da função ***realloc()***, cujos parâmetros são um ponteiro para o início do bloco de memória e a quantidade de bytes a ser alocada.
  - Essa função retorna um ponteiro para o início do novo bloco de memória. Este ponteiro pode ser igual ao ponteiro para o bloco de memória original.
  - Caso seja diferente, a função **realloc** copia os dados armazenados no bloco de memória original para o novo bloco de memória.

Alocação dinâmica de memória

```
int main()
{
  int i;
  int * v;

  printf("vetor inicial:\n");
  v = (int *)calloc(5,sizeof(int));
  for (i = 0; i < 5; i++)
    v[i] = rand()%100;
  for (i = 0; i < 5; i++)
    printf("%3d  ",v[i]);

  printf("\n\nvetor maior:\n");
  v = (int *)realloc(v,10*sizeof(int));
  for (i = 5; i < 10; i++)
    v[i] = rand()%100;
  for (i = 0; i < 10; i++)
    printf("%3d  ",v[i]);

  printf("\n\nvetor menor:\n");
  v = (int *)realloc(v,3*sizeof(int));
  for (i = 0; i < 3; i++)
    printf("%3d  ",v[i]);

  printf("\n\n");
  system("PAUSE");
  return 0;
}
```

→ Criação do vetor.

→ Aumentando o espaço alocado.

→ Diminuindo o espaço alocado.

String como ponteiros

- **Considere o seguinte programa:**

```
void main()
{
   char *msg;
   msg = "Linguagem C\n";
   printf(msg);
   system("pause");
}
```

  - Neste caso, ***msg*** é um ponteiro para char, ou equivalentemente, um vetor de caracteres.

  - Note que não há alocação de memória chamando-se a função ***calloc*** ou a função ***malloc***.

String como ponteiros

- **Este tipo de alocação dinâmica ocorre apenas no caso de *atribuições* para *strings*.**
- **Se o valor do string não for atribuído diretamente (por exemplo, valor é lido pelo comando gets), a alocação deve ser feita usando *calloc* ou *malloc*.**

```
#define MAX_TAM  50

void main()
{
   char *nome;
   nome = (char *)calloc(MAX_TAM,sizeof(char));
   printf("Seu nome completo: ");
   gets(nome);
   printf(nome);
   system("pause");
}
```

Atividade

- **Pesquisar como é realizada a alocação dinâmica para matrizes e trazer um exemplo para próxima aula.**

Referências

- MIZRAHI, V. V. Treinamento em Linguagem C. 2ª Edição. São Paulo: Person Prentice Hall, 2008.