COS110 – Algoritmos e Programação – Módulo 8

Priscila Machado Vieira Lima

Maio/2018

Curso : Engenharia de Controle e Automação

NCE

Universidade Federal do Rio de Janeiro

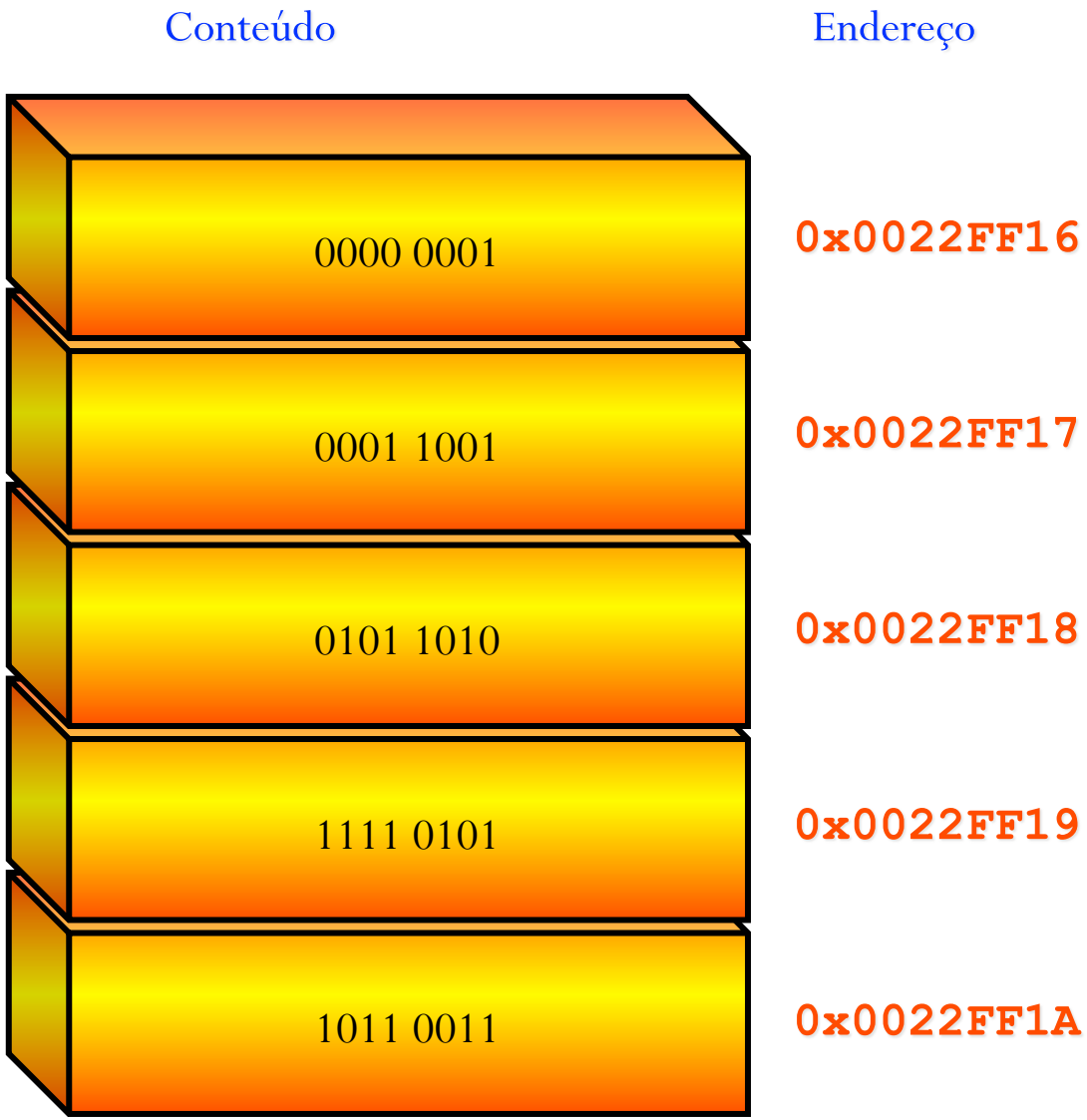
Roteiro

- **Endereçamento de Memória**
- Ponteiros: declaração e operações
- Ponteiros para Vetores e Matrizes
- Alocação Dinâmica de Memória

Endereços

- A memória de qualquer computador é uma seqüência de bytes.
- Cada byte pode armazenar um número inteiro entre 0 e 255.
- Cada byte na memória é identificado por um endereço numérico, independente do seu conteúdo.

Endereços


Conteúdo
Endereço
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0x0022FF16
0x0022FF17
0x0022FF18
0x0022FF19
0x0022FF1A

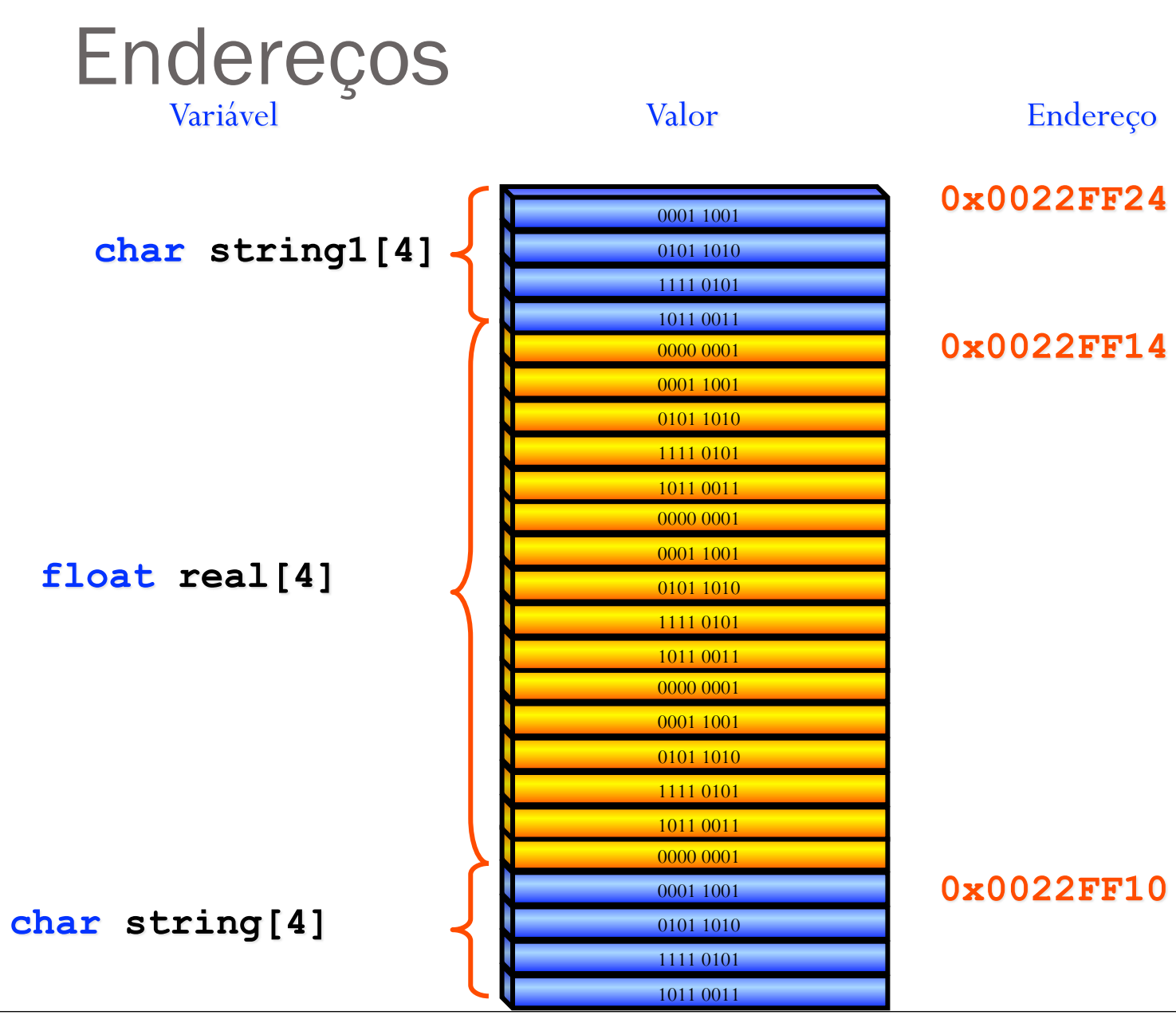
Endereços

- Cada objeto (variáveis, strings, vetores, etc.) que reside na memória do computador ocupa um certo número de bytes:
  - Inteiros: 4 bytes consecutivos
  - Caracteres: 1 byte
  - Ponto-flutuante: 4 bytes consecutivos

- Cada objeto tem um endereço.

- Na arquitetura IA-32 (Intel), o endereço é do byte menos significativo do objeto.

Endereços


Variável
Valor
Endereço
char string1[4]
float real[4]
char string[4]
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0000 0001
0001 1001
0101 1010
1111 0101
1011 0011
0x0022FF24
0x0022FF14
0x0022FF10

Endereços :: Resumo

```
int x = 100;
```

- Ao declararmos uma variável x como acima, temos associados a ela os seguintes elementos:
  - Um identificador(x)
  - Um endereço de memória ou referência (0xbfd267c4)
  - Um valor (100)
- Para acessarmos o endereço de uma variável, utilizamos o operador **&**

Roteiro

- Endereçamento de Memória
- **Ponteiros: declaração e operações**
- Ponteiros para Vetores e Matrizes
- Alocação Dinâmica de Memória

Ponteiros

- Um ponteiro (apontador ou *pointer*) é um tipo especial de variável cujo valor é um endereço.

- Um ponteiro pode ter o valor especial NULL, quando não contiver nenhum endereço.

- NULL é uma constante definida na biblioteca **`stdlib.h`**.

Ponteiros

```
*var
```

- A expressão acima representa o conteúdo do endereço de memória guardado na variável **`var`**

- Ou seja, **`var`** não guarda um valor, mas sim um endereço de memória.

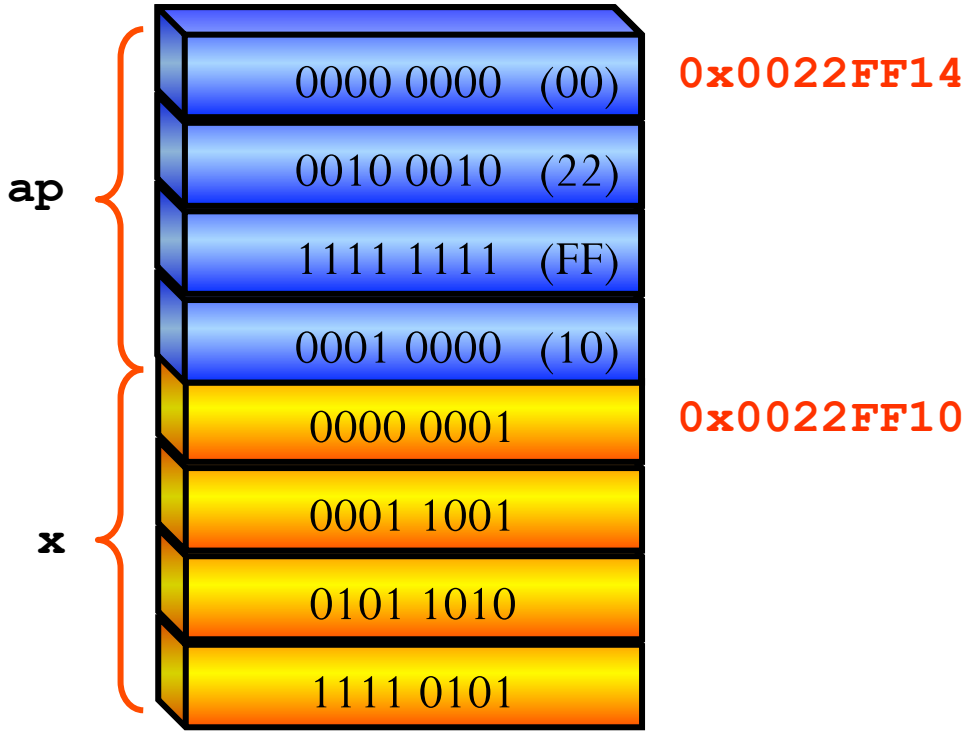
Ponteiros

```
*var
```

- O símbolo * acima é conhecido como operador de indireção.
- A operação acima é conhecida como desreferenciamento do ponteiro **`var`**.

Ponteiros :: Exemplo

```
int x;
int *ap;     // apontador para inteiros
ap = &x;     // ap aponta para x
```



Ponteiros

- Há vários tipos de ponteiros:
  - ponteiros para caracteres
  - ponteiros para inteiros
  - ponteiros para ponteiros para inteiros
  - ponteiros para vetores
  - ponteiros para estruturas

- O compilador C faz questão de saber de que tipo de ponteiro você está definindo.

Ponteiros :: Exemplo

```
int        *ap_int;      // apontador para int
char   *ap_char;     // apontador para char
float  *ap_float;    // apontador para float
double *ap_double;   // apontador para double

// apontador para apontador
int    **ap_ap_int;
```

Aritmética com Ponteiros

- Um conjunto limitado de operação aritméticas pode ser executado.

- Os ponteiros são endereços de memória. Assim, ao somar 1 a um ponteiro, você estará indo para o próximo endereço de memória do tipo de dado especificado.

Aritmética com Ponteiros

Nota:

Se v for um vetor ou um ponteiro para o primeiro elemento de um vetor, então para obter o elemento índice n desse vetor pode-se fazer v[n] ou *(v+n).

v[n] == *(v+n)

Aritmética com Ponteiros

1. Incremento

Um ponteiro pode ser incrementado como qualquer variável. No entanto, o incremento de uma unidade não significa que o endereço anteriormente armazenado no ponteiro seja incrementado em um *byte*.

Na realidade, se **ptr** é um ponteiro para um determinado tipo, quando **ptr** é incrementado, por exemplo, de uma unidade, o endereço que passa a conter é igual ao endereço anterior de **ptr** + **sizeof(tipo)** para que o ponteiro aponta, isto é, o ponteiro avança não um *byte,* mas sim a dimensão do tipo do objeto para o qual aponta.

Nota:

Um ponteiro para o tipo xyz avança sempre sizeof(xyz) *bytes* por unidade de incremento.

Aritmética com Ponteiros

2. Decremento

O decremento de ponteiros funciona da mesma forma que o incremento anteriormente apresentado.

Nota:

Um ponteiro para o tipo xyz recua sempre sizeof(xyz) *bytes* por unidade de decremento.

Aritmética com Ponteiros

3. Diferença

A operação de diferença entre dois ponteiros para elementos do mesmo tipo permite saber quantos elementos existem entre um endereço e o outro.

Por exemplo, o comprimento de uma *string* pode ser obtido através da diferença entre o endereço do caractere '\0' e o endereço do caractere original.

Nota:

A diferença entre ponteiros só pode ser realizada entre ponteiros do mesmo tipo.

Aritmética com Ponteiros

4. Comparação

É também possível a comparação de dois ponteiros para o mesmo tipo, utilizando os operadores relacionais (<, <=, >, >= , == e !=).

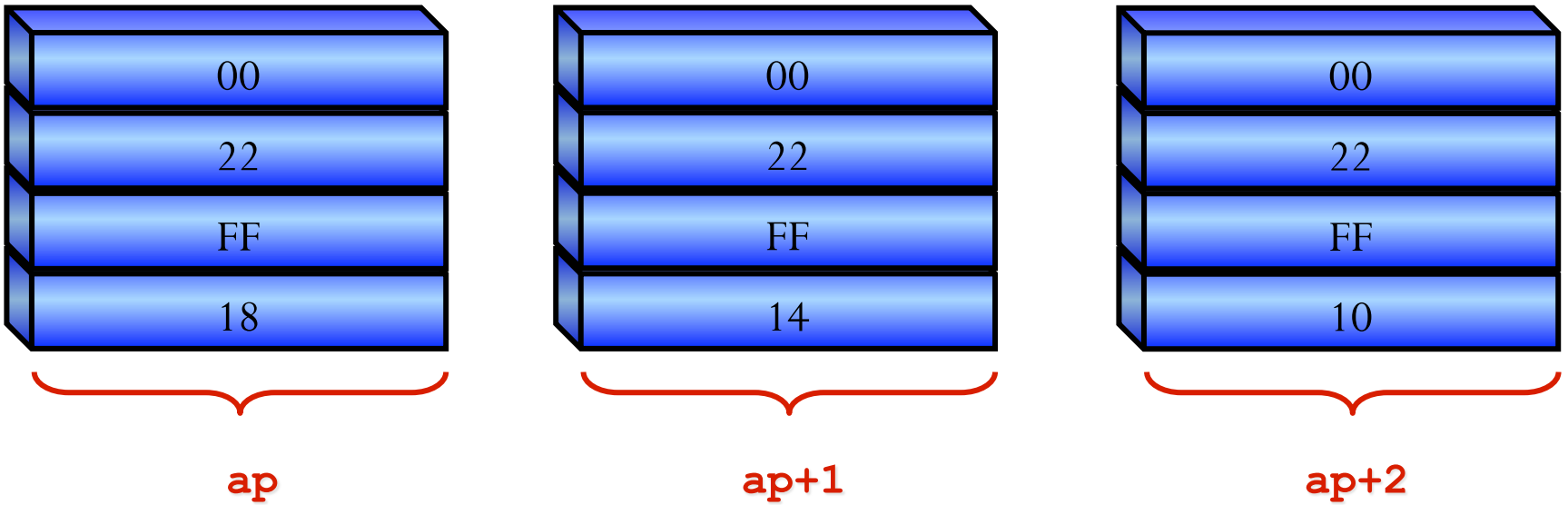
```
Ver linha 22: do exemplo prog0802.c
```

Nota:

A diferença e a comparação entre ponteiros só podem ser realizadas entre ponteiros do mesmo tipo.

Aritmética com Ponteiros

```
int    *ap;
```



Ponteiros

Um ponteiro é uma variável que contém o endereço de outra variável.

A sua declaração é feita usando o tipo da variável para a qual se quer apontar, seguido de um asterisco.

O endereço de uma variável pode ser obtido através do operador **&** (Endereço de).

Se um ponteiro **ptr** contiver o endereço de outra variável, pode-se obter o valor dessa variável através do operador * (**Apontado por**), fazendo ***ptr**.

Para evitar problemas, os ponteiros devem ser sempre iniciados com o endereço de uma variável ou com **NULL**. A constante NULL indica que o ponteiro não aponta para nenhum endereço.

Os ponteiros possuem uma aritmética própria, a qual permite realizar operações de incremento, decremento, diferença e comparação.

O nome de um vetor corresponde sempre ao endereço do primeiro elemento do vetor.

Assim, sempre que se passa um vetor para uma função apenas o endereço do primeiro elemento é enviado para ela, e não a totalidade do vetor.

Roteiro

- Endereçamento de Memória
- Ponteiros: declaração e operações
- **Ponteiros para Vetores e Matrizes**
- Alocação Dinâmica de Memória

Ponteiros e Matrizes

- O nome de uma matriz é, na verdade, um ponteiro para o primeiro elemento da matriz (endereço base)

- Assim, temos duas formas de indexar os elementos de uma matriz ou vetor:
  - Usando o operador de indexação (`v[4]`)
  - Usando aritmética de endereços (`*(ap_v + 4)`)

Ponteiros e Matrizes

Nota:

O nome de um vetor corresponde ao ENDEREÇO do seu primeiro elemento. Assim, se *s* for um vetor ***s == &s[0]***

Nota:

Os vetores são sempre passados às funções sem o &, pois o nome de um vetor é por si só um endereço.

Nota:

Se v for um vetor ou um ponteiro para o primeiro elemento de um vetor, então para obter o elemento índice n desse vetor pode-se fazer v[n] ou *(v+n).

v[n] == *(v+n)

Ponteiros e Matrizes

Nota:

Sempre que se passa um vetor para uma função, apenas o endereço original deste é efetivamente enviado para a função. É assim impossível saber, dentro de uma função, qual a dimensão dos vetores que foram passados, a menos que se envie um outro parâmetro indicador do número de elementos ou um delimitador em cada vetor. Assim, é da responsabilidade do programador garantir que os vetores enviados para as funções contêm os elementos necessários ao processamento a que serão submetidos.

Nota:

Sendo o nome de um vetor o endereço do seu primeiro elemento, poderemos com ele realizar todas as operações a que temos acesso quando manipulamos ponteiros, desde que essas operações não alterem o seu valor, pois o nome de um vetor é uma constante.

Ponteiros e Matrizes

Resumo das Operações sobre Ponteiros

Operação	Exemplo	Observações
Atribuição	ptr = &x	Podemos atribuir um valor (endereço) a um ponteiro. Se quisermos que aponte para nada podemos atribuir-lhe o valor da constante NULL.
Incremento	ptr=ptr+2	Incremento de 2*sizeof(tipo) de ptr.
Decremento	ptr=ptr-10	Decremento de 10*sizeof(tipo) de ptr.
Apontado por	*ptr	O operador asterisco permite obter o valor existente na posição cujo endereço está armazenado em ptr.
Endereço de	&ptr	Tal como qualquer outra variável, um ponteiro ocupa espaço em memória. Dessa forma podemos saber qual o endereço que um ponteiro ocupa em memória.
Diferença	ptr1 - ptr2	Permite-nos saber qual o nº de elementos entre ptr1 e ptr2.
Comparação	ptr1 > ptr2	Permite-nos verificar, por exemplo, qual a ordem de dois elementos num vetor através do valor dos seus endereços.

Ponteiros e Matrizes

- O nome de uma matriz é, na verdade, um ponteiro para o primeiro elemento da matriz (endereço base)

- Assim, temos duas formas de indexar os elementos de uma matriz ou vetor:
  - Usando o operador de indexação (`v[4]`)
  - Usando aritmética de endereços (`*(ap_v + 4)`)

Aritmética com Ponteiros

- Sempre que somar ou subtrair ponteiros, deve-se trabalhar com o tamanho do tipo de dado utilizado.

- Para isso você pode usar o operador **`sizeof()`**.

Matrizes de ponteiros

- Normalmente, são utilizadas como ponteiros para strings, pois uma string é essencialmente um ponteiro para o seu primeiro caractere.

```
void  systax_error(int num)
{
   char *erro[] = {
      "Arquivo nao pode ser aberto\n",
      "Erro de leitura\n",
      "Erro de escrita\n",
      "Falha de midia\n"};
   printf("%s", erro[num]);
}
```

Matrizes de ponteiros

- Ponteiros podem ser organizados em matrizes como qualquer outro tipo de dado.

- Nesse caso, basta observar que o operador `*` tem precedência menor que o operador de indexação `[]`.

```
int  *vet_ap[5];
char *vet_cadeias[5];
```

Ponteiro para função

- Um ponteiro para uma função contém o endereço da função na memória.
- Da mesma forma que um nome de matriz, um nome de função é o endereço na memória do começo do código que executa a tarefa da função.
- O uso mais comum de ponteiros para funções é permitir que uma função possa ser passada como parâmetro para uma outra função.

Ponteiro para função

- Ponteiros de função podem ser:
  - atribuídos a outros ponteiros,
  - passados como argumentos,
  - retornados por funções, e
  - armazenados em matrizes.

Função que retorna ponteiros

- Funções que devolvem ponteiros funcionam da mesma forma que os outros tipos de funções

- Alguns detalhes devem ser observados:
  - Ponteiros não são variáveis
  - Quando incrementados, eles apontam para o próximo endereço do tipo apontado
  - Por causa disso, o compilador deve saber o tipo apontado por cada ponteiro declarado
  - Portanto, uma função que retorna ponteiro deve declarar explicitamente qual tipo de ponteiro está retornando

Função que retorna ponteiros

```
<tipo>  *funcao(.....)
{
    ....
    return (ponteiro);
}
```

<tipo> não pode ser void, pois:

- Função deve devolver algum valor
- Ponteiro deve apontar para algum tipo de dado

Resumo sobre ponteiros 1/2

Notas Finais

É necessário prestar atenção em alguns pontos sobre ponteiros:

1. Um ponteiro é uma variável que não tem memória própria associada (apenas possui o espaço para conter um endereço), apontando normalmente para outros objetos já existentes. Funciona, mais ou menos, como um comando de televisão.
2. Embora seja possível utilizá-los como vetores, os ponteiros não possuem memória própria. Só se pode utilizar o endereçamento através de um ponteiro depois que este está apontando para algum objeto já existente.
3. Não se deve fazer cargas iniciais de objetos apontados por um ponteiro que ainda não tenha sido iniciado.

Resumo sobre ponteiros 2/2

4. Por segurança, inicie sempre os seus ponteiros. Se não souber para onde apontá-los, inicie-os com NULL.
5. Nunca se deve confundir ponteiros com vetores sem dimensão. Se não sabemos qual a dimensão de que necessitamos, como o compilador poderá saber?
6. Em uma declaração de um ponteiro com carga inicial automática

```
int *p = NULL;
```

é o ponteiro **p** que é iniciado, e não ***p**, embora a atribuição possa por vezes sugerir o contrário.

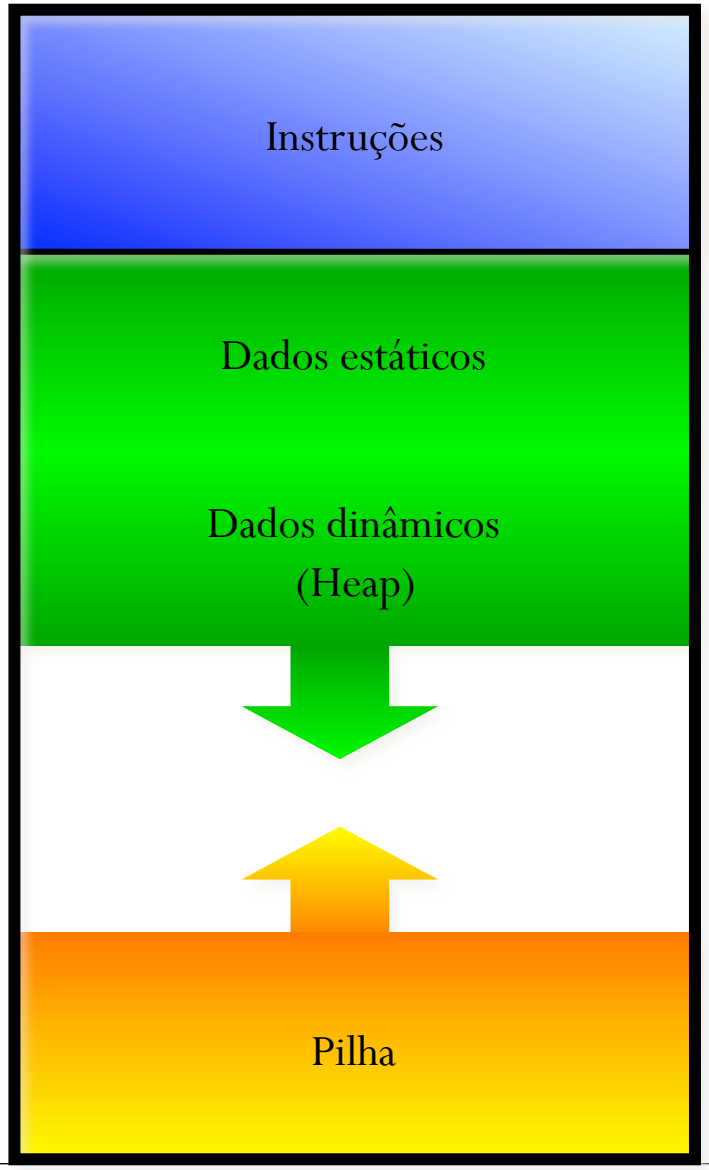
Roteiro

- Endereçamento de Memória
- Ponteiros: declaração e operações
- Ponteiros para Vetores e Matrizes
- **Alocação Dinâmica de Memória**

Alocação dinâmica de memória

- Um programa, ao ser executado, divide a memória do computador em quatro áreas:
  - Instruções – armazena o código C compilado e montado em linguagem de máquina.
  - Pilha – nela são criadas as variáveis locais.
  - Memória estática – onde são criadas as variáveis globais e locais estáticas.
  - Heap – destinado a armazenar dados alocados dinamicamente.

Alocação dinâmica de memória



Embora seu tamanho seja desconhecido, o heap geralmente contém uma quantidade razoavelmente grande de memória livre.

Alocação dinâmica de memória

- As variáveis da pilha e da memória estática precisam ter tamanho conhecido antes do programa ser compilado.

- A alocação dinâmica de memória permite reservar espaços de memória de tamanho arbitrário e acessá-los através de apontadores.

- Desta forma, podemos escrever programas mais flexíveis, pois nem todos os tamanhos devem ser definidos aos escrever o programa.

Alocação dinâmica de memória

- A alocação e liberação desses espaços de memória é feito por duas funções da biblioteca **`stdlib.h`**:
  - **`malloc()`** : aloca um espaço de memória.
  - **`free()`** : libera um espaço de memória.

Alocação dinâmica de memória :: Função `malloc()`

- Abreviatura de *memory allocation*
- Aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco.
- Retorna um ponteiro do tipo **`void`**.
- Deve-se utilizar um cast (modelador) para transformar o ponteiro devolvido para um ponteiro do tipo desejado.

Alocação dinâmica de memória
:: Função `malloc()`

- Exemplo:

  Alocando um vetor de n elementos do tipo inteiro.

```
int *p;
p = (int*) malloc(n * sizeof(int));
```

Alocação dinâmica de memória :: Função `malloc()`

- A memória não é infinita. Se a memória do computador já estiver toda ocupada, a função malloc não consegue alocar mais espaço e devolve NULL.

- Usar um ponteiro nulo travará o seu computador na maioria dos casos.

Alocação dinâmica de memória
:: Função `malloc()`

- Convém verificar essa possibilidade antes de prosseguir.

```
ptr = (int*) malloc (1000*sizeof(int));
if (ptr == NULL)
{
    printf ("Sem memoria\n");
}
...
```

Alocação dinâmica de memória :: Função `free()`

- Libera o uso de um bloco de memória, permitindo que este espaço seja reaproveitado.

- O mesmo endereço retornado por uma chamada da função **`malloc()`** deve ser passado para a função **`free()`**.

- A determinação do tamanho do bloco a ser liberado é feita automaticamente.

Alocação dinâmica de memória
:: Função `free()`

- Exemplo: liberando espaço ocupado por um vetor de 100 inteiros

```
int *p;
p = (int*) malloc(100 * sizeof(int));
free(p);
```

Alocação dinâmica de memória :: Função `realloc()`

- Essa função faz um bloco já alocado crescer ou diminuir, preservando o conteúdo já existente:

```
(tipo*) realloc(tipo *apontador, int novo_tamanho)
```

```
int *x, i;
x = (int *) malloc(4000*sizeof(int));
for(i=0;i<4000;i++) x[i] = rand()%100;

x = (int *) realloc(x, 8000*sizeof(int));

x = (int *) realloc(x, 2000*sizeof(int));
free(x);
```

Alocação dinâmica de memória
:: Função `realloc()`

```
int *x, i;

x = (int *) malloc(4000*sizeof(int));

for(i=0;i<4000;i++) x[i] = rand()%100;


x = (int *) realloc(x, 8000*sizeof(int));


x = (int *) realloc(x, 2000*sizeof(int));

free(x);
```