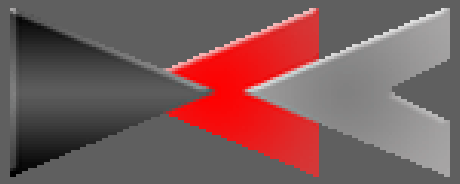
DCC107



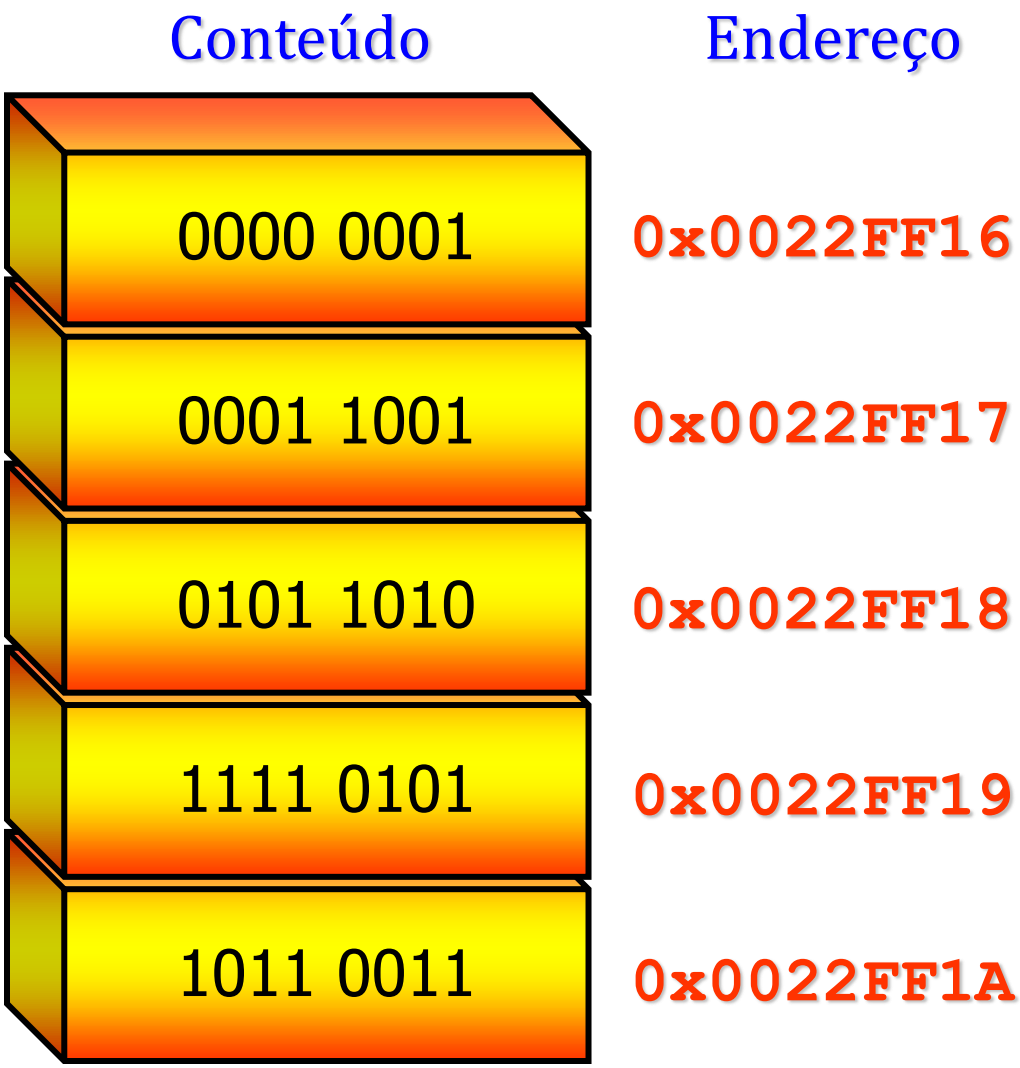
Laboratório de Programação II

Ponteiros

Endereços

- A memória de qualquer computador é uma sequência 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

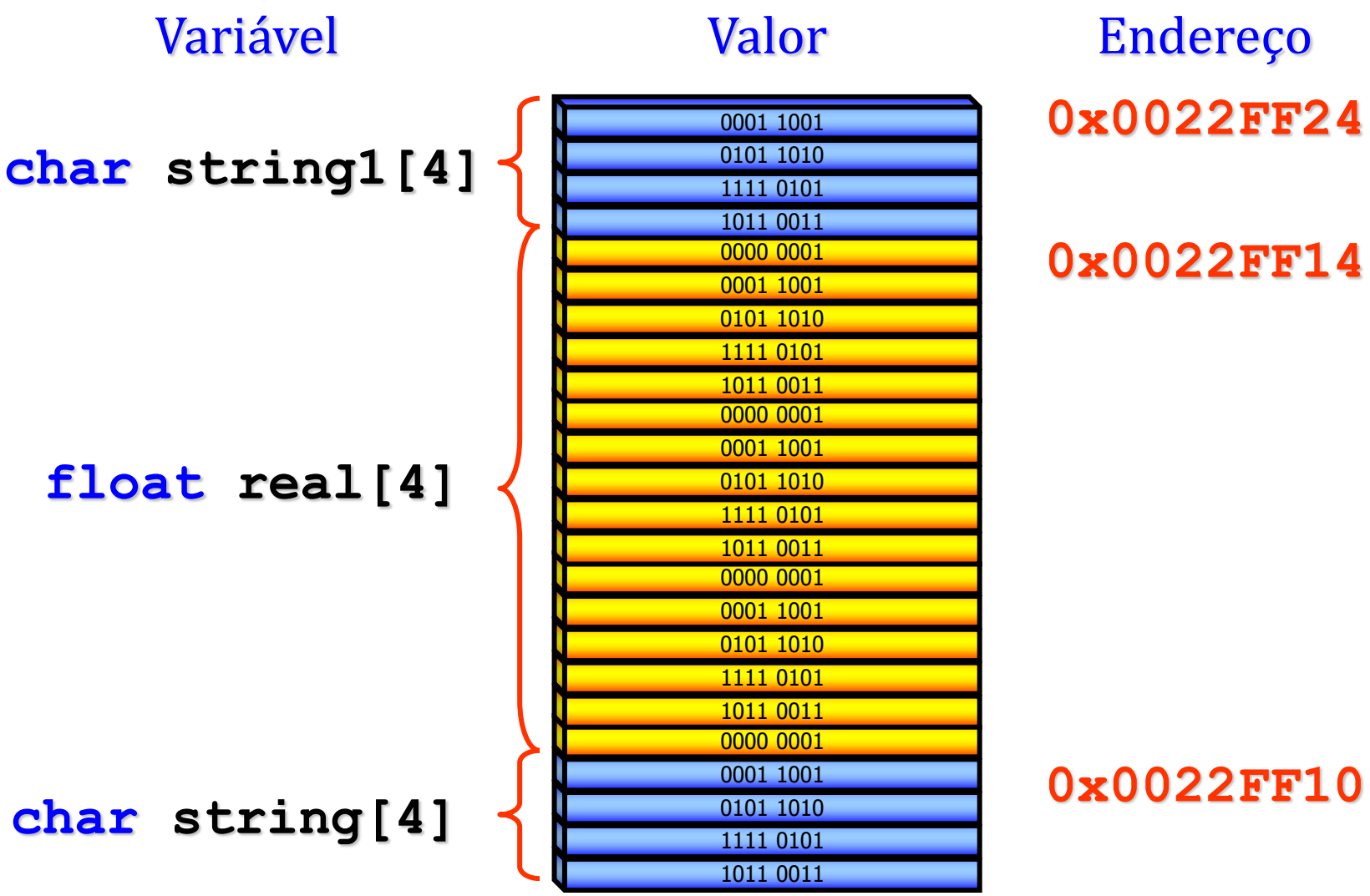

Conteúdo
Endereço
0000 0001
0x0022FF16
0001 1001
0x0022FF17
0101 1010
0x0022FF18
1111 0101
0x0022FF19
1011 0011
0x0022FF1A

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.

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 nome (x);
  - Um endereço de memória ou referência (0xbfd267c4);
  - Um valor (100).
- Para acessarmos o endereço de uma variável, utilizamos, em C, o operador **&**.

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 contém 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`** armazena um valor especial que é um endereço de memória.

Ponteiros

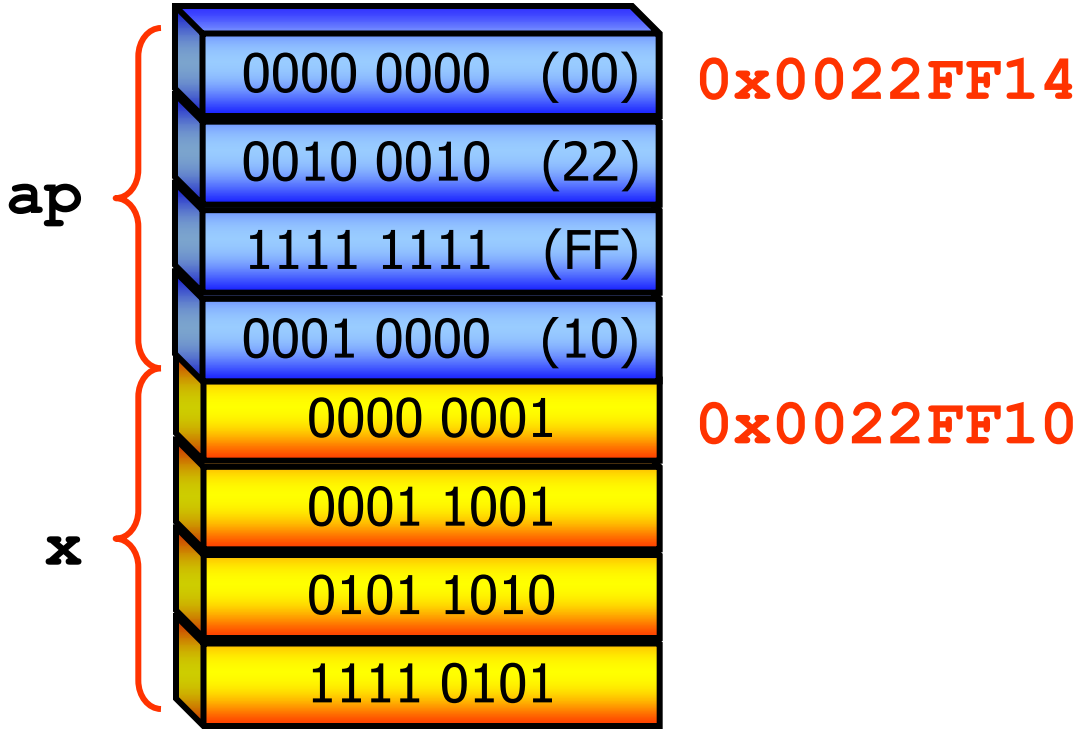

```
*var
```

- 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;
```

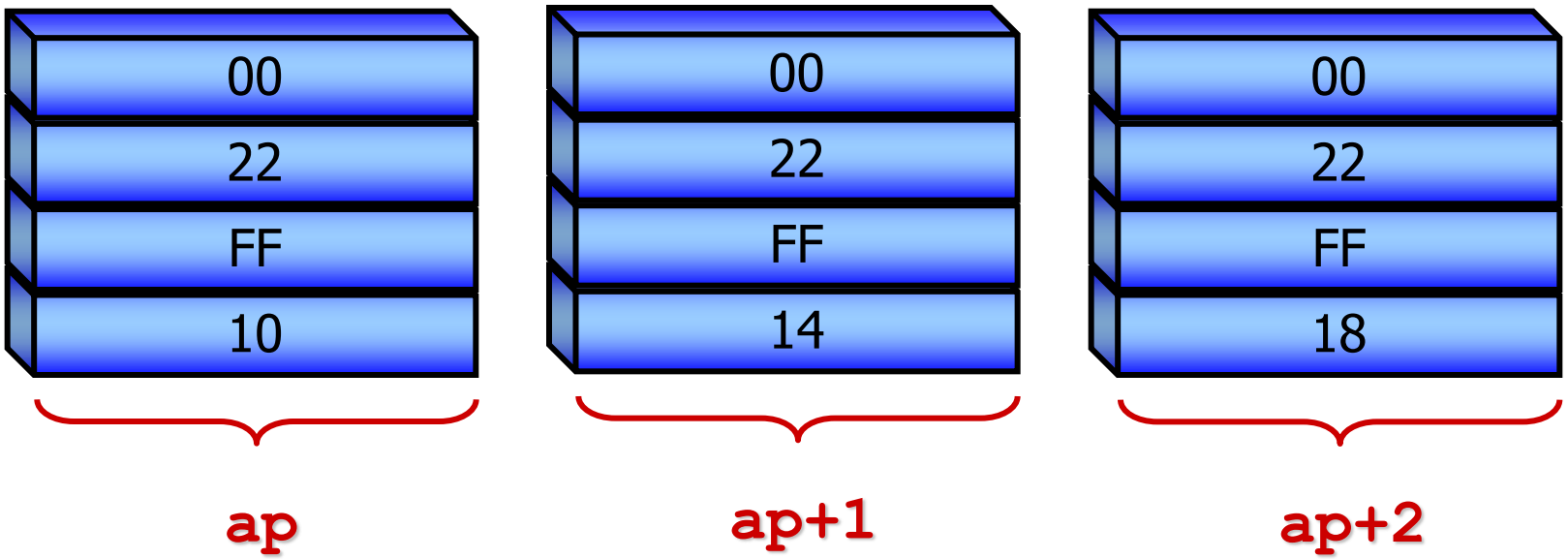
Aritmética com Ponteiros

- 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


```
int *ap;
```



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()`**.

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)`).

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];
```

Matrizes de ponteiros

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

```
void syntax_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]);
}
```

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.

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.

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.

Função `malloc()`

- Exemplo:
  - Alocando um vetor de n elementos do tipo inteiro.

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

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.

Função `malloc()`

- Convém verificar essa possibilidade antes de prosseguir.

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

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.

Função `free()`

- Exemplo:
  - Liberando espaço ocupado por um vetor de 100 inteiros.

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

Função `realloc()`

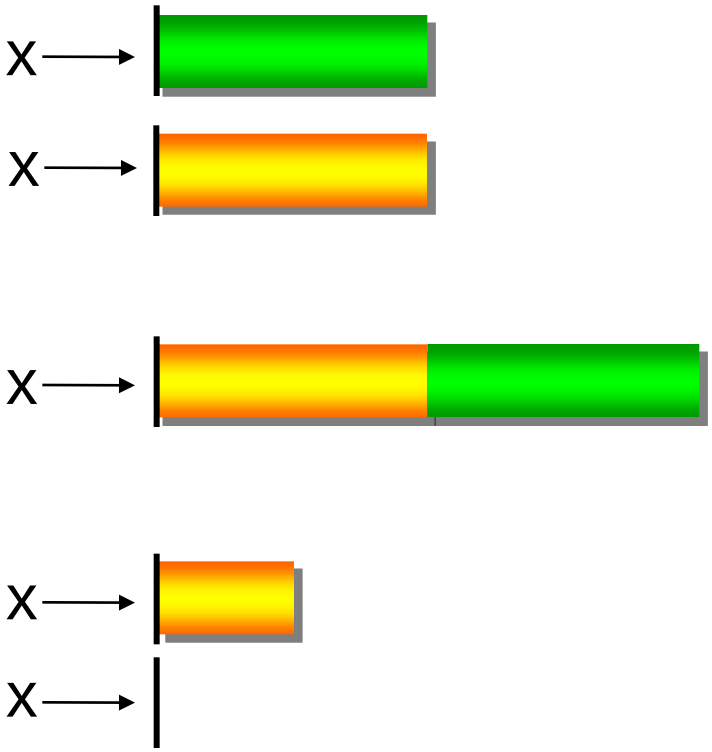
- Essa função faz um bloco já alocado crescer ou diminuir, preservando o conteúdo já existente:

```
(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);
```

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);
```



Ponteiro para estrutura

- Considere a *struct*:

```
struct ponto {
    float x;
    float y;
};
```

- Pode-se declarar a variável p:

```
struct ponto p;
```

- Podemos também declarar variáveis do tipo ponteiro para estrutura:

```
struct ponto *pp;
```

- Alocação dinâmica de estruturas:

```
struct ponto* pp;
pp = (struct ponto*) malloc(sizeof(struct ponto));
```

Ponteiro para estrutura

- O acesso aos campos dessa estrutura é feito indiretamente:

```
(*pp).x = 12.0;
(*pp).y = 8.0;
```

os parênteses são indispensáveis, pois o operador "conteúdo de" (*) tem precedência menor que o operador de acesso (.).

- C oferece outro operador de acesso que permite acessar campos a partir do ponteiro da estrutura. Este operador é composto por um traço seguido de um sinal de maior, formando uma seta (->):

```
pp->x = 12.0;
pp->y = 8.0;
```

lê – se: pp aponta ponto x
pp aponta ponto y

Exercícios - Ponteiros

1. Para cada uma das sentenças a seguintes, escreva uma instrução que realize a tarefa indicada. Admita que as variáveis de ponto flutuante **num1** e **num2** foram declaradas e que **num1** foi inicializada com o valor **7.3**.
   a) Declare a variável **fPtr** como ponteiro para um objeto do tipo **float**.
   b) Atribua o endereço da variável **num1** à variável de ponteiro **fPtr**.
   c) Imprima o valor do objeto apontado por **fPtr**.
   d) Atribua o valor do objeto apontado por **fPtr** à variável **num2**.
   e) Imprima o valor de **num2**.
   f) Imprima o endereço de **num1**. Use o especificador de conversão **%p**.
   g) Imprima o endereço armazenado em **fPtr**. Use o especificador de conversão **%p**. O valor impresso é igual ao endereço de **num1**?

Exercícios - Ponteiros

1. Respostas:
   a) float *fPtr;
   b) fPtr = &num1;
   c) printf("O valor de fPtr eh %f", *fPtr);
   d) num2 = *fPtr;
   e) printf("O valor de num2 eh %f", num2);
   f) printf("O valor de num1 eh %p", &num1);
   g) printf("O endereco armazenado em fPtr eh %p", fPtr); sim, o valor é o mesmo.

Exercícios - Ponteiros

2. Faça o que se pede:
   a) Escreva o cabeçalho de uma função chamada **exchange** que utiliza dois ponteiros para os números de ponto flutuante **x** e **y** como parâmetros e não retorna um valor;
   b) Escreva o protótipo da função do item (a).

Exercícios - Ponteiros

2. Respostas:
   a) void exchange(float *x, float *y);
   b) void exchange(float *, float *).

Exercícios - Ponteiros

3. Encontre o erro em cada um dos segmentos de programas a seguir. Admita que:

```
int *zPtr; /*zPtr faz referencia ao array z*/
int *aPtr; = NULL;
void *sPtr = NULL;
int numero, i;
int z[5] = {1, 2, 3, 4, 5};
sPtr = z;
```

a) ++zPtr;

b) /*usa o ponteiro para obter o primeiro valor do array*/
numero = zPtr;

c) /*atribui o elemento 2 do array (valor 3) a numero*/
numero = *zPtr[2];

d) /*imprime todo o array z*/
for(i = 0; i<=5; i++)
printf("%d ", zPtr[i]);

e) /*atribui o valor apontado por sPtr a numero*/
numero = *sPtr;

f) ++z;

Exercícios - Ponteiros

4. Para cada uma das sentenças a seguir, escreva uma única instrução que realize a tarefa indicada. Considere que as variáveis inteiras **valor1** e **valor2**, do tipo **long**, foram declaradas e que **valor1** foi inicializada com o valor 200000.

   a) Declare a variável **lPtr** como ponteiro de um objeto do tipo **long**.

   b) Atribua o endereço da variável **valor1** à variável de ponteiro **lPtr**.

   c) Imprima o valor do objeto apontado por **lPtr**.

   d) Atribua o valor do objeto apontado por **lPtr** à variável **valor2**.

   e) Imprima o valor de **valor2**.

   f) Imprima o endereço de **valor1**.

   g) Imprima o endereço armazenado em **lPtr**. O valor impresso é igual ao endereço de **valor1**?

Exercícios - Ponteiros

5. Fazer um programa para:
   a) declarar variáveis a, b, c, d do tipo **int**.
   b) declarar variáveis e, f, g, h do tipo **float**.
   c) declarar vetor v de 10 elementos do tipo **char**.
   d) declarar variável x do tipo **int**.
   e) criar um ponteiro apontando para o endereço de a.
   f) incrementar o ponteiro, mostrando o conteúdo do endereço apontado (em forma de número). Caso o endereço coincida com o endereço de alguma outra variável, informar o fato.