UFABC

Ponteiros

prof. Fabrício Olivetti de França

Anteriormente em prog. estrut.

Vimos que as variáveis representando arrays em C armazenam apenas um apontador para o endereço de memória contendo os valores da array.

UFABC

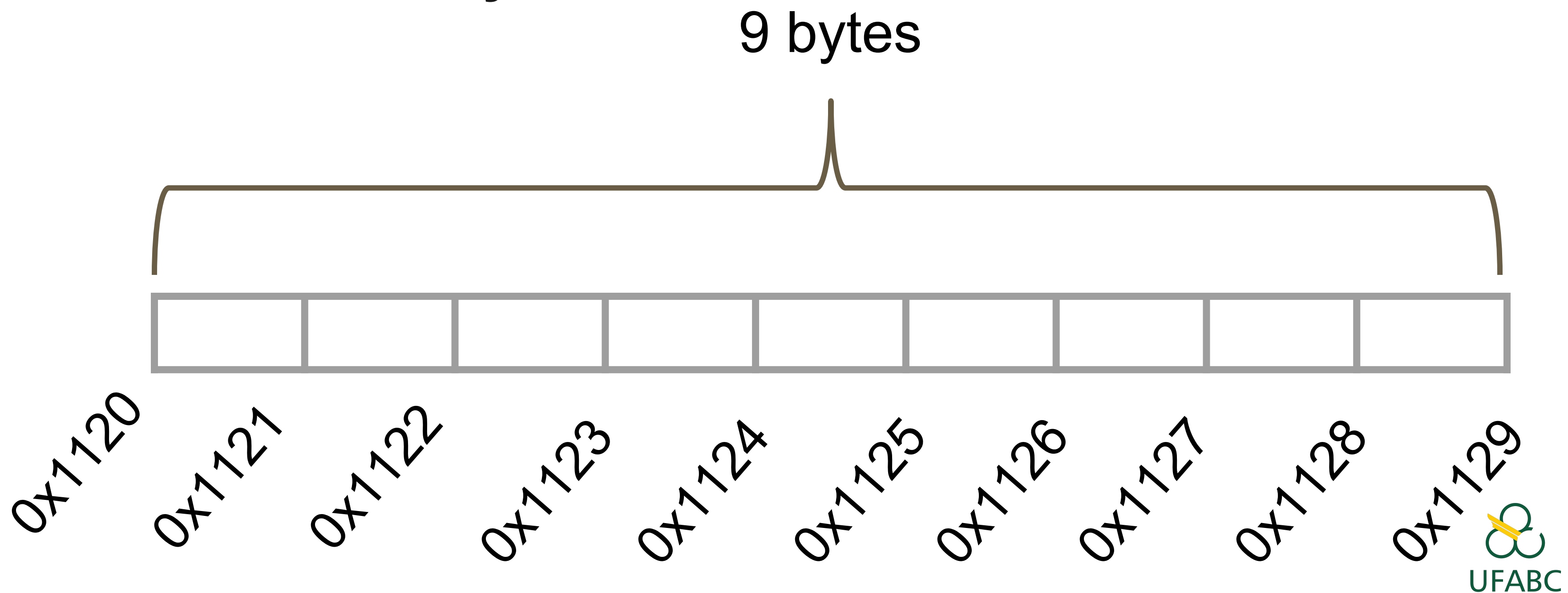
Anteriormente em prog. estrut.

Uma das características desses apontadores é que, ao passar uma array como parâmetro de uma função, o seu conteúdo era alterado dentro dela.

UFABC

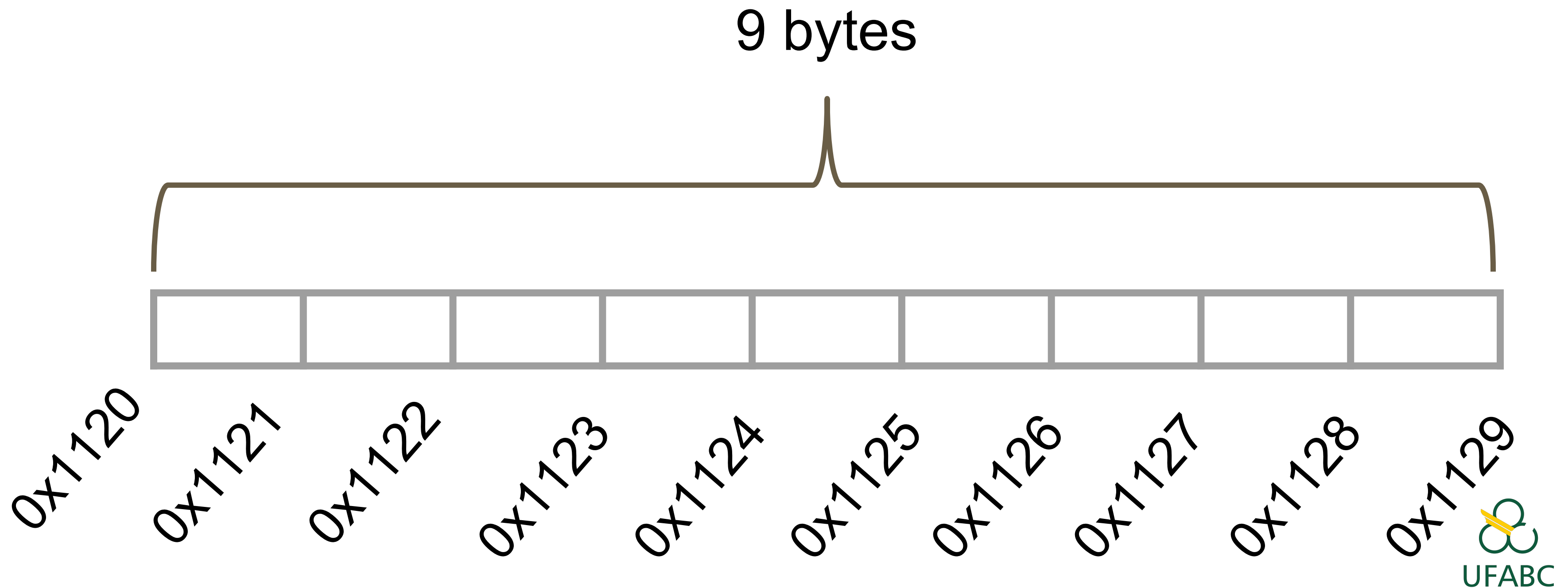
Relembrando sobre Memória

Um computador possui um vetor de posições de memória devidamente endereçadas. Cada elemento da memória armazena **um byte**.



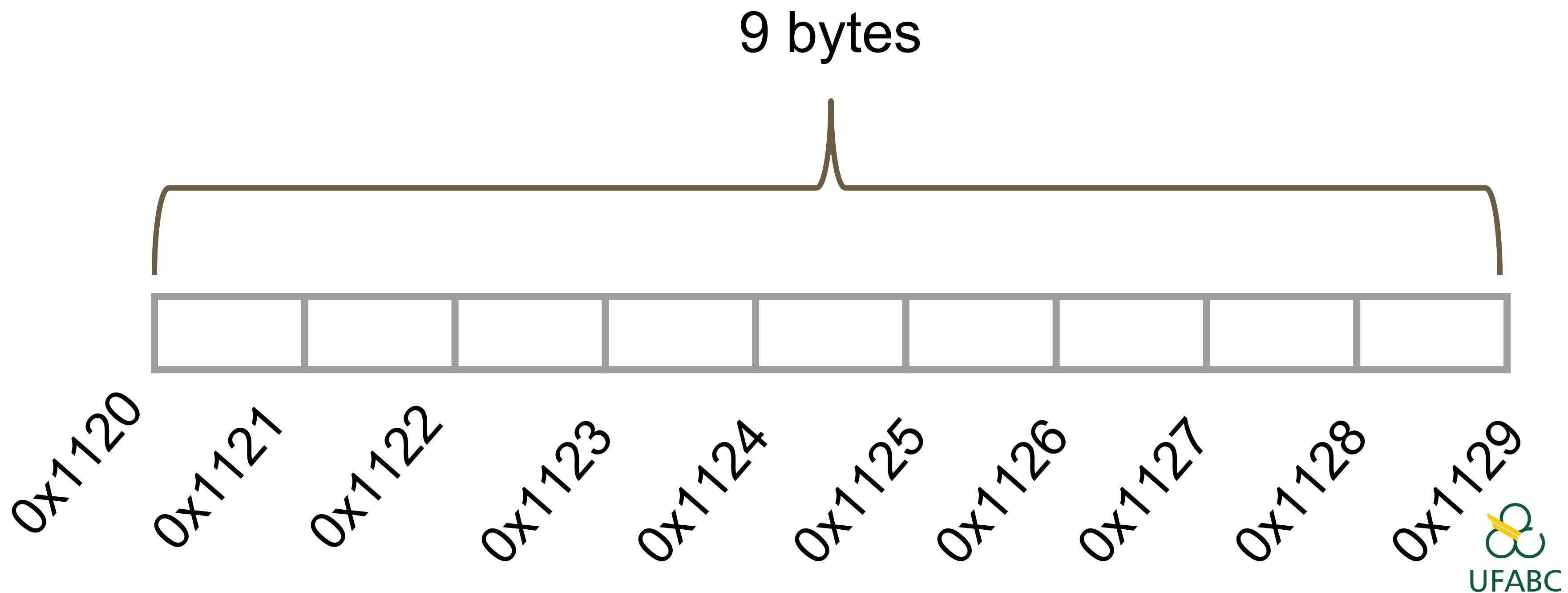
Relembrando sobre Memória

Para o computador um byte da memória pode representar um tipo **char** ou uma sequência de dois bytes um **int**.



Relembrando sobre Memória

A diferença entre os tipos na linguagem C é apenas quantos bytes consecutivos de memória serão utilizados.



UFABC

Relembrando sobre Memória

É por isso que na linguagem C podemos realizar operações entre diferentes tipos:

```
int x = 3;
char c = ‘A’;
x+c    /* == ‘D’ */
```

UFABC

Relembrando sobre Memória

E transformar um tipo em outro através do chamado **casting**.

```
int x = 65;
printf("%c", (char) x);  /* 'A'  */
```

UFABC

Relembrando sobre Memória

O casting de variáveis indica para o programa que tal variável deve ser tratada como outro tipo naquela instrução específica:

tipo1 variavel;

(tipo2) variavel;   ← casting!!

UFABC

Relembrando sobre Memória

O operador unário **&** permite obter o endereço da memória que certa variável reside:

```
int x = 10;
printf("A variável x está na posição %p da memória\n",
               &x );
```

UFABC

Ponteiros

A linguagem C permite criar apontadores, denominados **ponteiros** para quaisquer tipos de variáveis.

Os ponteiros são variáveis de 2 ou 4 bytes (dependendo do computador) que armazenam um endereço de memória.

UFABC

Ponteiros

Para declarar um ponteiro basta usar * após a declaração do tipo:

```
/* ponteiro para uma variável contendo um inteiro */
int * x;
```

UFABC

Ponteiros

Note que, ao contrário da declaração de tipo, nesse caso a variável pode apenas receber **endereço de memória**.

int * x;

x = 100;

printf(“%d\n”, *x); ← conteúdo do endereço “100” da memória

<na verdade esse programa "trava">

UFABC

Ponteiros

```
int x;
int * y;
x = 10;
y = &x;
```

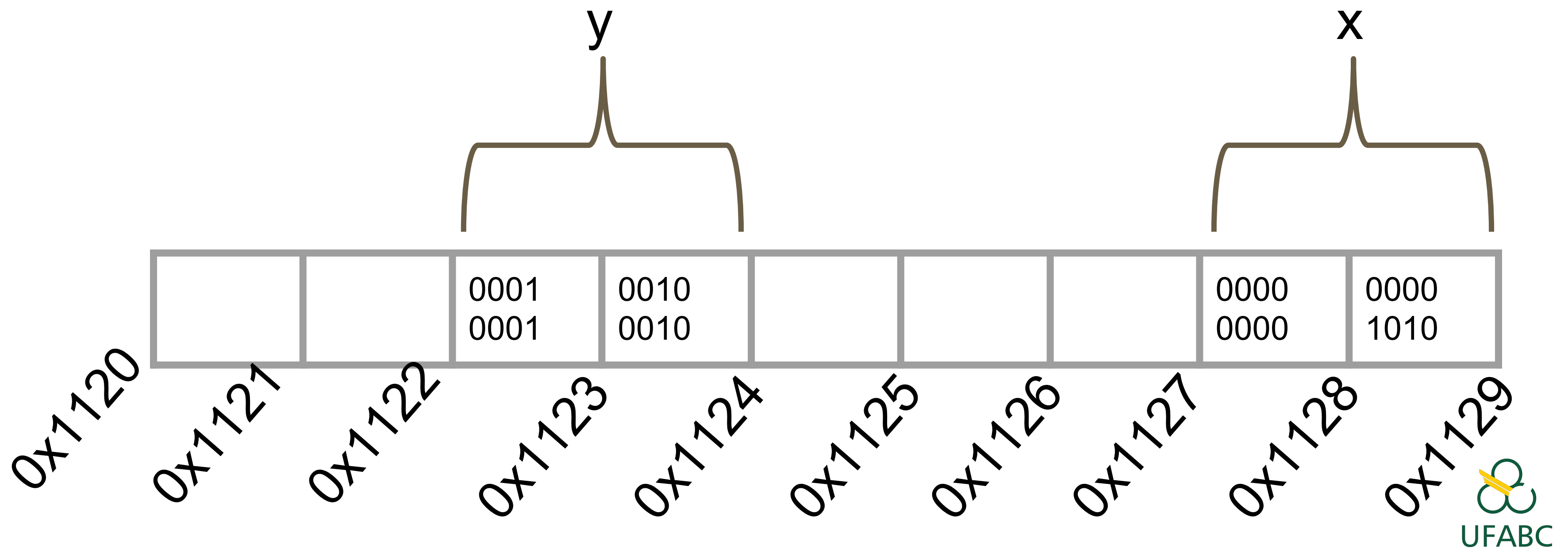
UFABC

Ponteiros

int x;

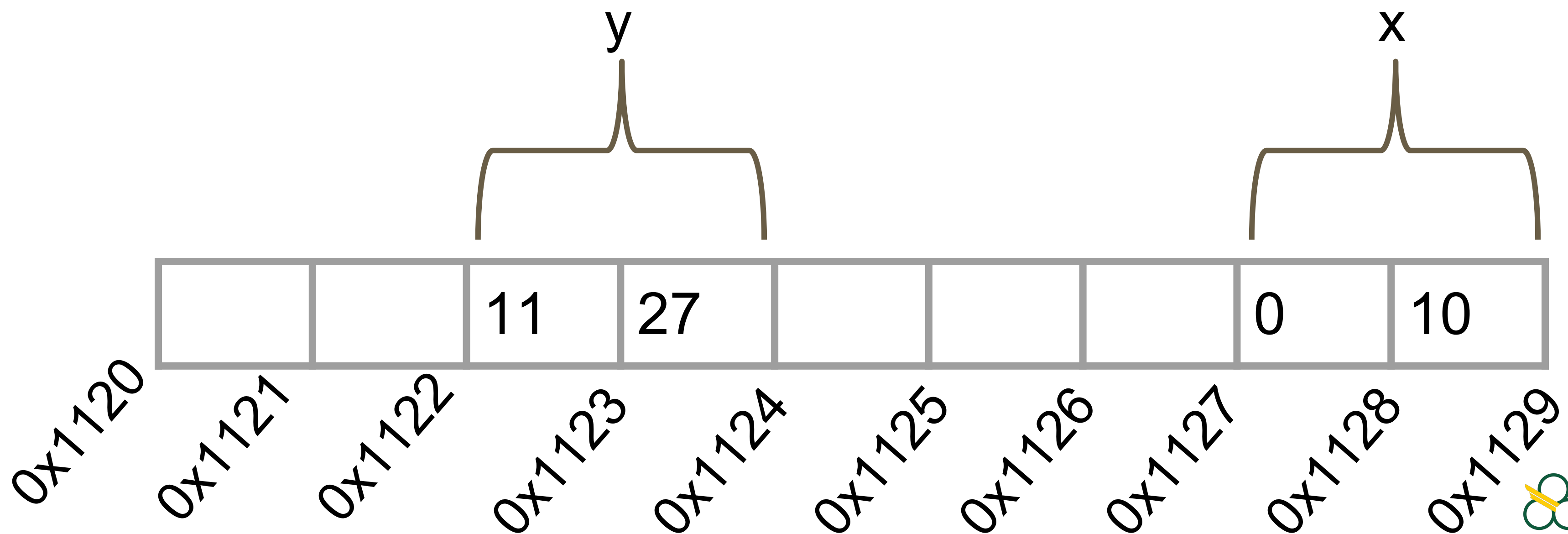
int * y;

x = 10;



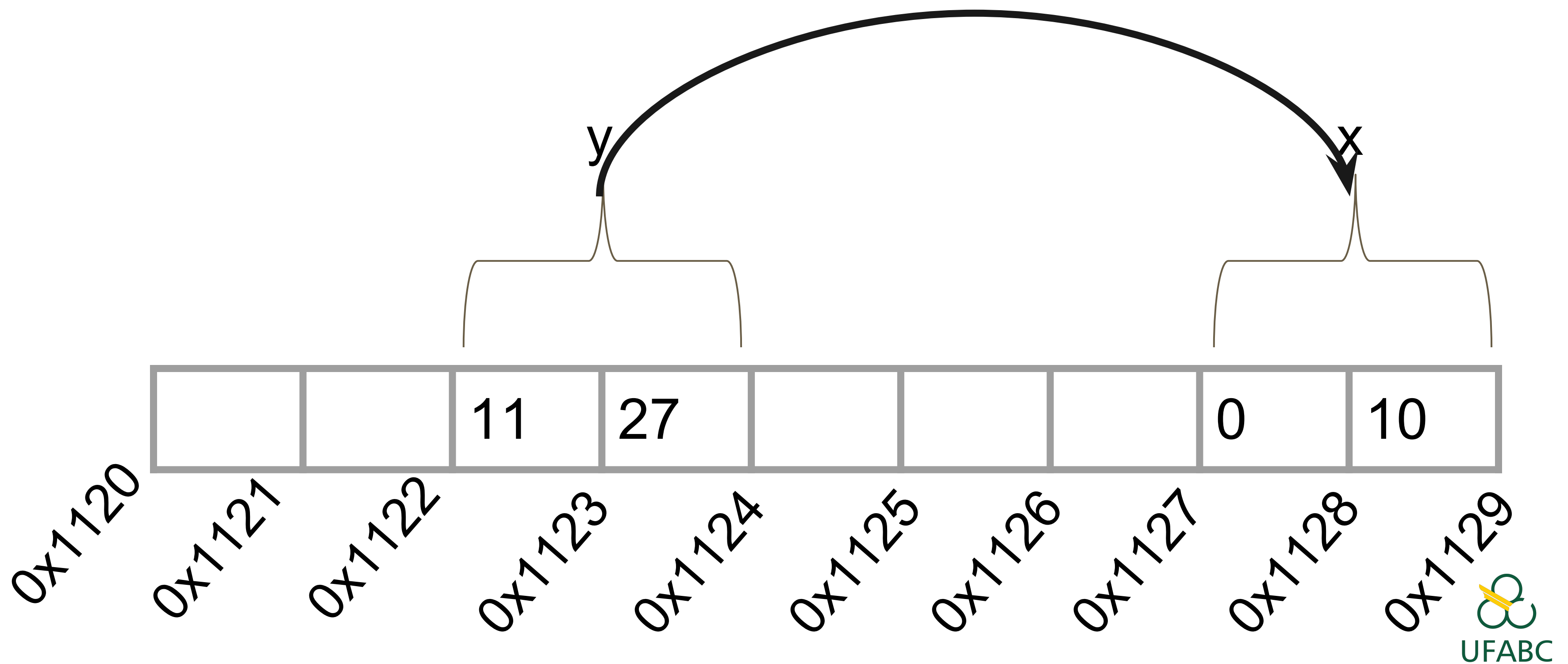
Ponteiros

```
int x;
int * y;
x = 10;
```



Ponteiros

y = &x;



Ponteiros

Para acessar o conteúdo do endereço que um ponteiro aponta, utilizamos o operador *****:

```
printf("%d\n", *y);   /* conteúdo de x → 10  */
```

UFABC

Ponteiros

Os operadores aritméticos são válidos para as variáveis ponteiros:

```
int x = 10;
int * y = &x;
*y = *y + 10;   /* x agora é 20 */
y = y + 1;   /* avança sizeof(int) bytes na memória */
```

UFABC

Exemplo

```
void troca (int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

UFABC

Exemplo

A função recebe uma cópia do valor das variáveis, então ao fazer:

```
troca(a, b);
```

As variáveis **a** e **b** não terão seus valores trocados.

UFABC

Passando ponteiros como parâmetros

Se ao invés de passar uma cópia dos valores, passassemos uma cópia dos endereços de memória, poderíamos alterar o conteúdo:

```
troca(&a, &b);
```

UFABC

Passando ponteiros como parâmetros

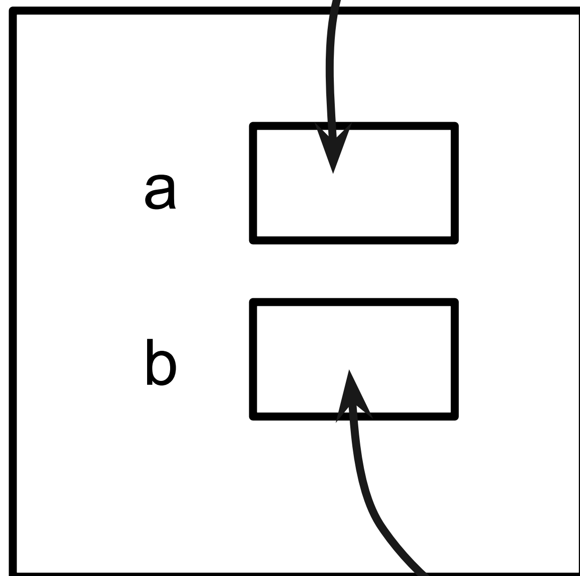
```
void troca (int * px, int * py)
{
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
```

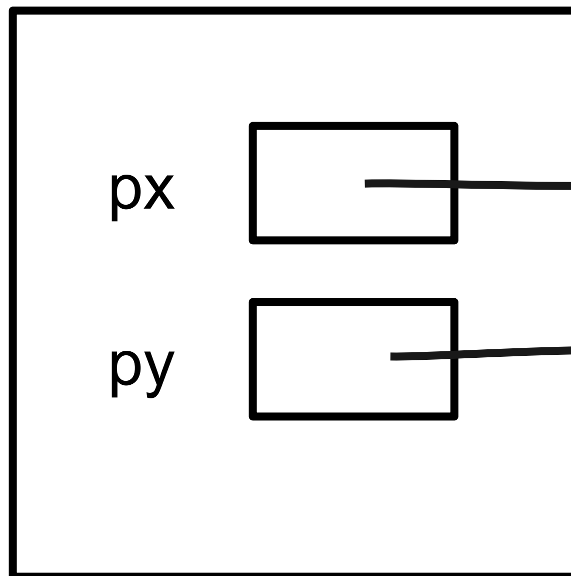
UFABC

Passando ponteiros como parâmetros

troca(&a, &b);

Função troca():





UFABC

adaptado do livro de Kernighan & Ritchie

Inicializando ponteiros

Na biblioteca **stdlib.h** define-se a constante **NULL** representando o valor nulo para ponteiros:

```
#include <stdlib.h>
int * x = NULL;    /*   não aponta o dedo pra ninguém */
```

UFABC

Exercício

Faça uma função quociente_resto() que recebe dois inteiros **x** e **y** e dois ponteiros para inteiros **q** e **r** para armazenar o quociente **q** e o resto **r**.

Faça com que a função retorne **1** em caso de sucesso e **0** em caso de erro (que erro ela pode ter?)

UFABC

Exercício

Quanto de memória essa função utiliza?

Quantas instruções ela executa?

UFABC

Vetores como ponteiros

Vimos anteriormente que uma array:

```
int x[100];
```

Reserva 100*sizeof(int) posições de memória e inicializa a variável x com o endereço da primeira posição.

UFABC

Vetores como ponteiros

Dessa forma:

x contém o endereço da memória do primeiro elemento.

(*x) é o conteúdo do primeiro elemento da array

*(x+3) é o conteúdo da posição 3 da array

UFABC

Vetores como ponteiros

Uma forma alternativa de declarar funções que recebem arrays:

```
int soma (int * v, int n);
```

UFABC

Vetores como ponteiros

```
int soma (int *v, int n)
{
    int i, total = 0;
    for (i=0; i<n;  i++) {
        total += v[i];
    }
    return total;
}
```

UFABC

Vetores como ponteiros

```
int x[100];
…
soma(x, 100); /* soma todos os elementos de x */
soma(x+5, 50);  /* soma os 50 primeiro elementos de x a partir da posição 5 */
```

UFABC

Malloc

A linguagem C permite que façamos a função de alocar memória dinamicamente para criar arrays.

Para isso utilizaremos a instrução **malloc**.

```
(void *) malloc( size_t size );
```

UFABC

Malloc

size_t - dependente da plataforma, variável para indicar tamanho em bytes.

void * - ponteiro do tipo void é utilizado para referenciar um tipo genérico, que pode ser transformado em qualquer outro tipo.

UFABC

Malloc

```
#include <stdlib.h>
#include <stdio.h>
void main ( )
{
        int * v;
        int tamanho;
        printf(“Entre com o tamanho: “);
        scanf(“%d”, &tamanho);
        v = malloc( tamanho * sizeof(int) );
}
```

UFABC

Malloc

```
v = malloc( tamanho * sizeof(int) );
```

tamanho * sizeof(int) - calcula quantos bytes tem um array com **tamanho** elementos do tipo **int**.

UFABC

Malloc

Nota1: uma vez alocado o espaço, não existe garantias do valor inicial dos elementos da array!!

Nota2: a função malloc não assegura que o espaço de memória foi realmente alocado, por isso é importante verificar se o ponteiro é diferente de NULL antes usá-lo!

UFABC

Malloc

```
v = malloc( tamanho * sizeof(int) );
printf("%d\n", v[0]); /* indeterminado */
v[0] = 10;
printf("%d\n", v[0]); /* 10 */
```

UFABC

Free

Após o uso da array e antes de terminar o programa, devemos liberar a memória para que ela seja utilizada por outros processos ou programas.

```
free(v);
```

UFABC

Array Dinâmica

```
int ** array;
```

Ela na verdade é uma array contendo arrays. Cada linha pode ter um número diferente de colunas.

UFABC

Array Dinâmica

```
int ** array = malloc(sizeof(int *)*N);
for (i=0;i<N;i++) {
    array[i] = malloc(sizeof(int)*M);
```

O segmento de memória não é contínuo, cada linha aponta para um segmento diferente.

UFABC

Array Dinâmica

```
for (i=0;i<N;i++) {
    free(array[i]);
free(array);
```

Da mesma forma devemos liberar o espaço de cada elemento da array bidimensional.

UFABC

Array Dinâmica

Nesse caso não podemos fazer *(&(array[0][0]) + i*M + j) para acessar um elemento utilizando aritmética de ponteiros.

Devemos fazer:

((array + i) + j)

UFABC

Array Dinâmica

A declaração de uma função que recebe uma array dinâmica como parâmetro é

```
funcao( int ** array );
```

UFABC

Exercício

```
void half_double (int * x, int * y)
{
    x /= 2;
    y *= 2;
}
```

UFABC

Exercício

```
int m = 10;
int n = 20;
half_double(m,n);
printf(“%d %d\n”, m, n);
```

UFABC

Exercício

1) O que a função irá imprimir?

2) Ela faz o que é esperado?

3) Como você arrumaria o erro?

UFABC

Exercício

```
int * x;
char * y;
…
```

As operações x+1 e y+1 somam o mesmo valor (1) ao endereço apontado por x e y?

UFABC

Exercício

Faça uma função chamada **copy** que recebe dois ponteiros **int**: **from** e **to** e um **int** chamado **n**.

A função deve copiar os primeiros **n** elementos de **from** para **to**.

UFABC

Exercício

O que devo alterar na função para que ela funcione com todos os tipos de variáveis?

O significado da variável **n** muda?

UFABC

Exercício

O seu chefe ouviu dizer que o laço **for** é muito mais custoso que o laço **while**. Reimplemente a função utilizando **while**.

UFABC

Exercício

Usar como condição de parada quando a variável chega em zero é mais rápido!

UFABC

Exercício

Eu quero que a função tenha apenas uma linha de instrução! (quanto menor o código mais rápido ele deve ficar né?)

UFABC

Exercício

```
void copy (void * from, void * to, size_t n)
{
	while(n--) *to++ = *from++;
}
```

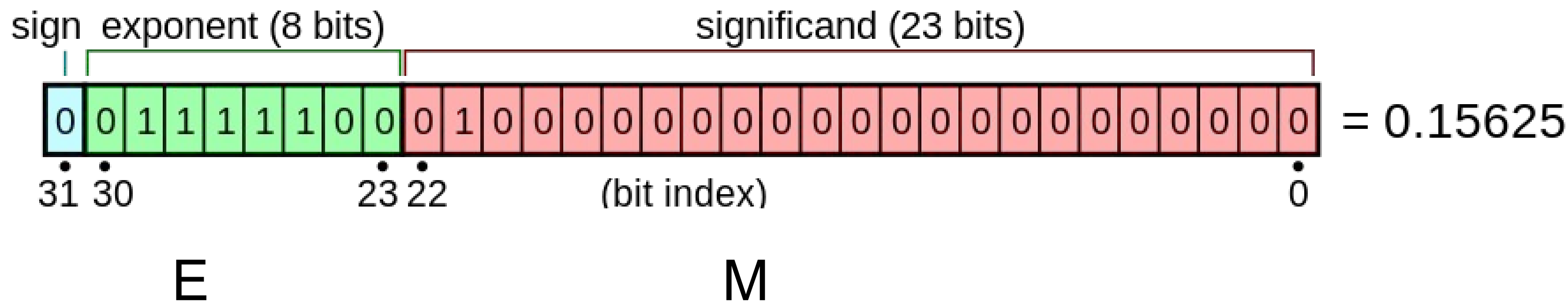
UFABC

UFABC

Curiosidade

Calculando $\log_2(x)$

Dado uma variável do tipo **float** com valor entre [1,2], temos que:



https://en.wikipedia.org/wiki/File:Float_w_significand_2.svg

Calculando $\log_2(x)$

Desconsiderando o sinal, o número em ponto flutuante é representado por:

$$x = 2^{E-127} (1 + M.2^{-23})$$

UFABC

Calculando $\log_2(x)$

Logo:

$$\log_2(x) = (E-127) + \log_2(1 + M.2^{-23})$$

UFABC

Calculando $\log_2(x)$

Temos que:

$$0 <= M.2^{-23} < 1$$

UFABC

Calculando $\log_2(x)$

E nessa situação temos que:

$$\log_2(1 + M.2^{-23}) \sim M.2^{-23} + eps$$

UFABC

Calculando $\log_2(x)$

Logo:

$$\log_2(x) = (E-127) + M.2^{-23} + eps$$

UFABC

Calculando $\log_2(x)$

Se convertermos x para int os 32 bits representarão:

$$(int)\ x = E.2^{23} + M$$

UFABC

Calculando $\log_2(x)$

$(int)\ x = E.2^{23} + M$

$= 2^{23} (E + M.2^{-23})$

$= 2^{23} (E - 127 + 127 + M.2^{-23} + eps - eps)$

$= 2^{23} (\mathbf{E - 127 + M.2^{-23} + eps} + 127 - eps)$

UFABC

Calculando $\log_2(x)$

(int) $x = E.2^{23} + M$

$= 2^{23}$ (**$E - 127 + M.2^{-23}$ + eps +** 127 - eps)

$= 2^{23}(\log(x) + 127 - eps)$

$= 2^{23}\log(x) + 2^{23}(127 - eps)$

UFABC

Calculando $\log_2(x)$

```
float x = …;
long ix = * ( long * ) &x;
log(x) = ( (float) ix )/8388608.0 - 127 + 0.0430357;
```

eps ótimo = 0.0430357

(historicamente foi encontrado por tentativa e erro)

UFABC