Fundamentos da Programação

Apontadores

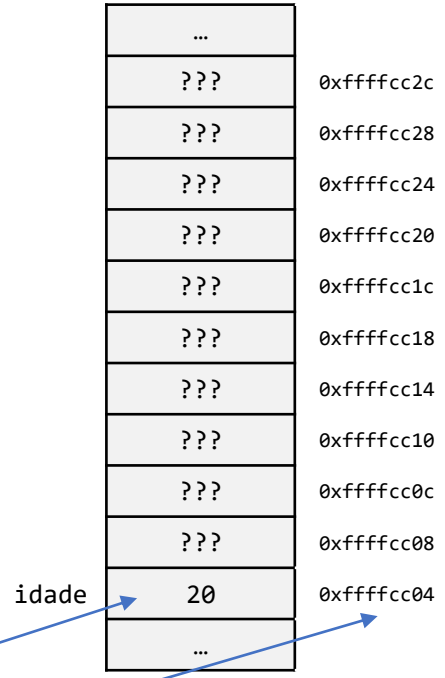
P.PORTO
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Conteúdo

- Apontadores
- Aritmética de apontadores
- Precedência dos operadores
- Apontadores e *arrays*
- Passagem de argumentos para funções
- Funções que retornam apontadores

Endereço de memória

- Uma variável é um **espaço em memória** no qual é possível guardar um **valor** de um determinado **tipo**.
- É possível saber o endereço de memória de uma variável utilizando o operador **&**.

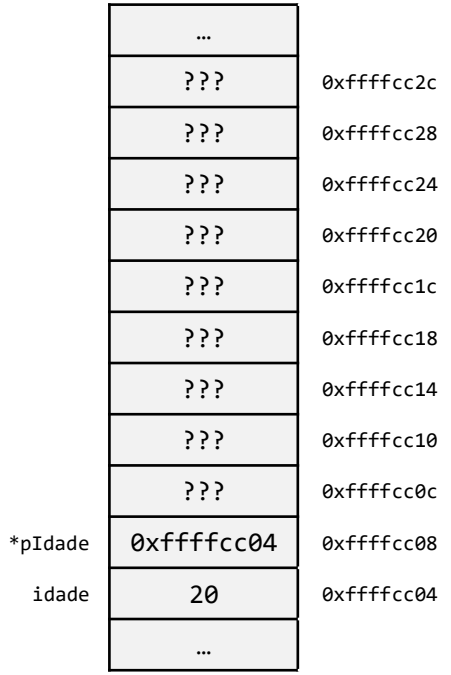



```c
#include <stdio.h>
int main() {
    int idade = 20;

    printf("\nIdade: %d", idade);
    printf("\nEndereço: %p", &idade);

    return 0;
}
```

```
Idade: 20
Endereço: 0xffffcc04
```

Apontadores

- Um **apontador** é uma variável cujo **valor** é um **endereço de memória**.
  - São declarados utilizando o caractere * antes do nome da variável.
    - Na declaração de um apontador é necessário indicar o **tipo da variável** para a qual vai apontar.
  - A atribuição de um endereço de memória a um apontador é realizada com o **&**.

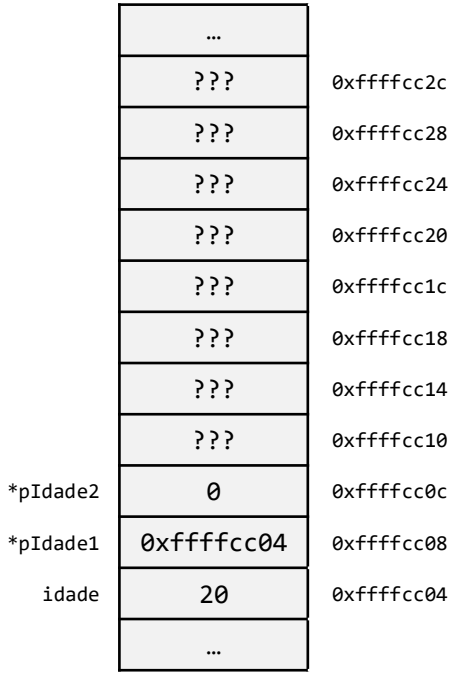



```
…
    int idade = 20, *pIdade;

    pIdade = &idade;

    printf("\nIdade: %d \nEndereço: %p", idade, &idade);
    printf("\npIdade: %p", pIdade);
…
```

```
Idade: 20
Endereço: 0xffffcc04
pIdade: 0xffffcc04
```

Apontadores

- Um apontador pode conter:
  - Um endereço de memória;
  - Constante **`NULL`** (ou **`0`**) para representar que não está a apontar para um endereço de memória (nulo).

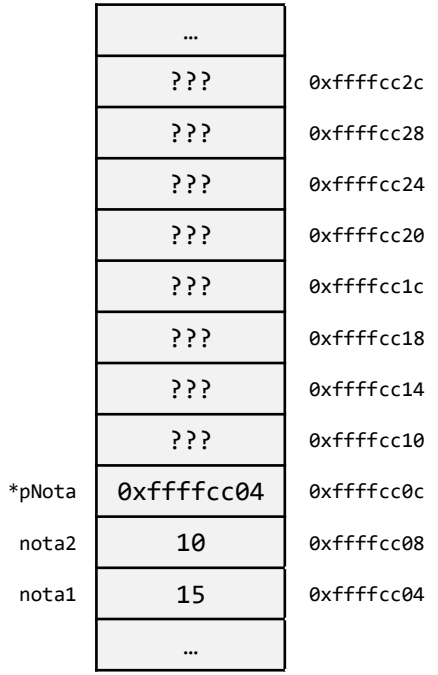



```
…
    int idade = 20, *pIdade1, *pIdade2 = NULL;

    pIdade1 = &idade;

    printf("\nIdade: %d <%p>", idade, &idade);
    printf("\npIdade1: %p", pIdade1);
    printf("\npIdade2: %p", pIdade2);
…
```

```
Idade: 20 <0xffffcc04>
pIdade1: 0xffffcc04
pIdade2: 0x0
```

Apontadores

- Para aceder ou modificar um valor apontado por um apontador utilizámos o caractere *.
  - *Valor que é apontado pelo apontador.*

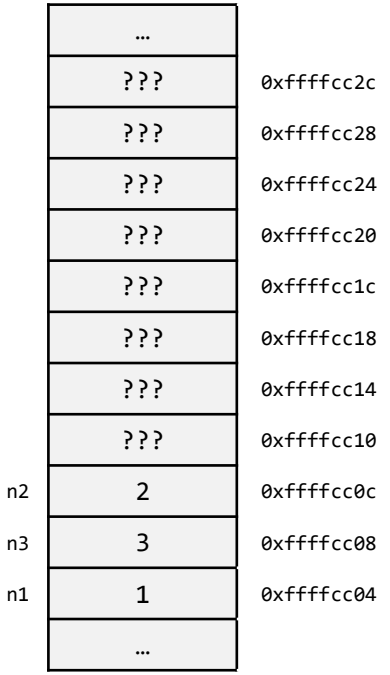



```
…
    int nota1 = 10, nota2 = 20, *pNota = NULL;
    pNota = &nota1;
    nota2 = *pNota;
    *pNota = 15;

    printf("\nnota1: %d <%p>", nota1, &nota1);
    printf("\nnota2: %d <%p>", nota2, &nota2);
    printf("\npNota: %p <%p> \nValor apontado: %d", pNota, &pNota, *pNota);
…
```

```
nota1: 15 <0xffffcc04>
nota2: 10 <0xffffcc08>
pNota: 0xffffcc04 <0xffffcc0c>
Valor apontado: 15
```

Aritmética de apontadores

- Os operadores **+**, **-**, **++**, e **--** podem ser utilizados com apontadores.
- Existe, no entanto, uma diferença com a aritmética regular:
  - o número adicionando/subtraído ao apontador é multiplicado pelo tamanho do tipo para o qual o apontador está apontando.
  - Pode-se pensar que **ptr + numero** em aritmética de apontadores é equivalente a **ptr + (numero * sizeof(*ptr))** em aritmética regular.




```
…
int n1 = 1, n2 = 2, n3 = 3;

printf("\n%p %d", &n1, *(&n1));
printf("\n%p %d", &n1 + 1, *(&n1 + 1));
printf("\n%p %d", &n1 + 2, *(&n1 + 2));
printf("\n%p %d", &n1 + 3, *(&n1 + 3));
…
```

```
0xffffcc04 1
0xffffcc08 3
0xffffcc0c 2
0xffffcc10 -13088
```

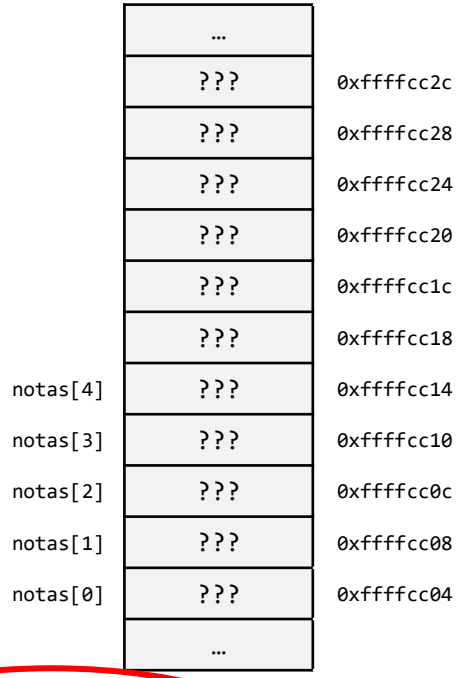
Precedência dos operadores

Operator	*Description*	*Associativity*
() **[]** **.** **->** **++ --**	Parentheses or function call Brackets or array subscript Dot or Member selection operator Arrow operator Postfix increment/decrement	left to right
++ -- **+ -** **! ~** **(type)** ***** **&** **sizeof**	Prefix increment/decrement Unary plus and minus not operator and bitwise complement type cast Indirection or dereference operator Address of operator Determine size in bytes	right to left
*** / %**	Multiplication, division and modulus	left to right
+ -	Addition and subtraction	left to right
<< >>	Bitwise left shift and right shift	left to right
< <= **> >=**	relational less than/less than equal to relational greater than/greater than or equal to	left to right
== !=	Relational equal to and not equal to	left to right
&	Bitwise AND	left to right
^	Bitwise exclusive OR	left to right
|	Bitwise inclusive OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Ternary operator	right to left
= **+= -=** ***= /=** **%= &=** **^= |=** **<<= >>=**	Assignment operator Addition/subtraction assignment Multiplication/division assignment Modulus and bitwise assignment Bitwise exclusive/inclusive OR assignment	right to left
,	Comma operator	left to right

https://overiq.com/c-programming-101/operator-precedence-and-associativity-in-c/

Apontadores e arrays

- O nome dado a um array refere-se à primeira posição do vetor.
  - **`nomeArray`** é equivalente a **`&nomeArray[0]`**

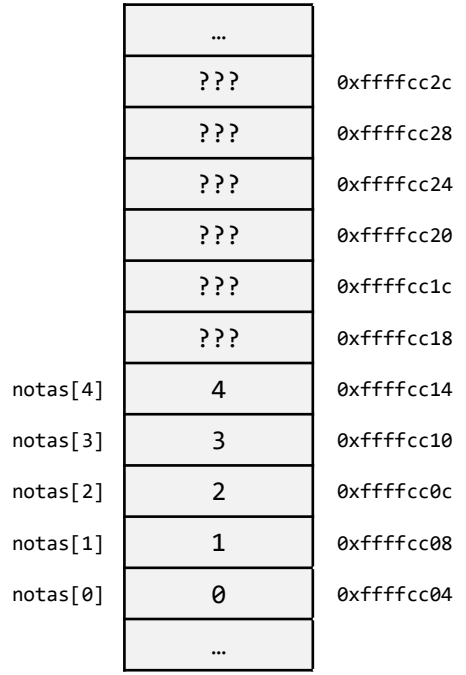



```
…
    int i, notas[5];

    printf("\nnotas: %p", notas);

    for (i = 0; i < 5; i++) {
        printf("\nnotas[%d]: %p", i, &notas[i]);
    }
…
```

```
notas: 0xffffcc04
notas[0]: 0xffffcc04
notas[1]: 0xffffcc08
notas[2]: 0xffffcc0c
notas[3]: 0xffffcc10
notas[4]: 0xffffcc14
```

Apontadores e arrays

- Se considerarmos a aritmética de apontadores as duas alternativas abaixo são equivalentes.



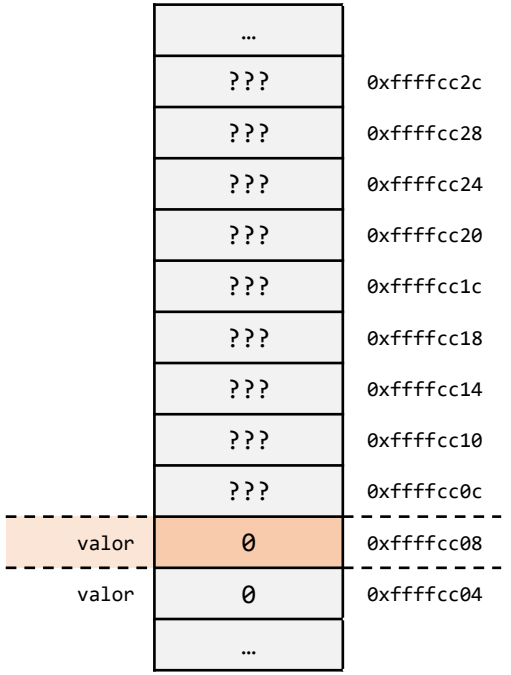

```
…
int i, notas[5];

for (i = 0; i < 5; i++) {
    notas[i] = i;
}

for (i = 0; i < 5; i++) {
    printf("%d", notas[i]);
}
…
```

```
…
int i, notas[5];

for (i = 0; i < 5; i++) {
    *(notas + i) = i;
}

for (i = 0; i < 5; i++) {
    printf(“%d", *(notas + i));
}
…
```

```
0 1 2 3 4
```

Passagem de argumentos por valor

- É atribuído o **valor** da variável passa como argumento para uma variável **local** à função.



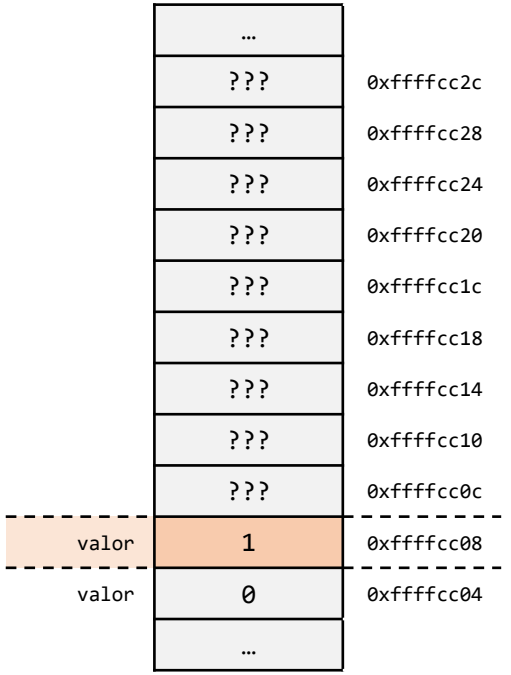

```
…
int main() {
    int valor = 0;

    incrementar(valor);

    printf("\nvalor: %d", valor);

    return 0;
}
…
```

0

```
void incrementar(int valor) {
    …
}
```

Passagem de argumentos por valor

- É atribuído o **valor** da variável passa como argumento para uma variável **local** à função.




```
…
int main() {
    int valor = 0;

    incrementar(valor);

    printf("\nvalor: %d", valor);

    return 0;
}
…
```

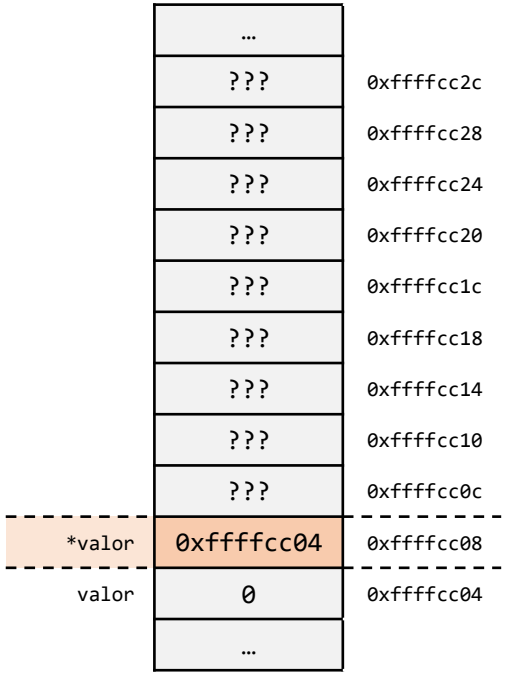
0

```
void incrementar(int valor) {
   printf("\nvalor: %d", ++valor);
}
```

```
valor: 1
valor: 0
```

Passagem de argumentos por referência

- É atribuído o **endereço** de memória da variável passa como argumento para uma variável **local** à função.



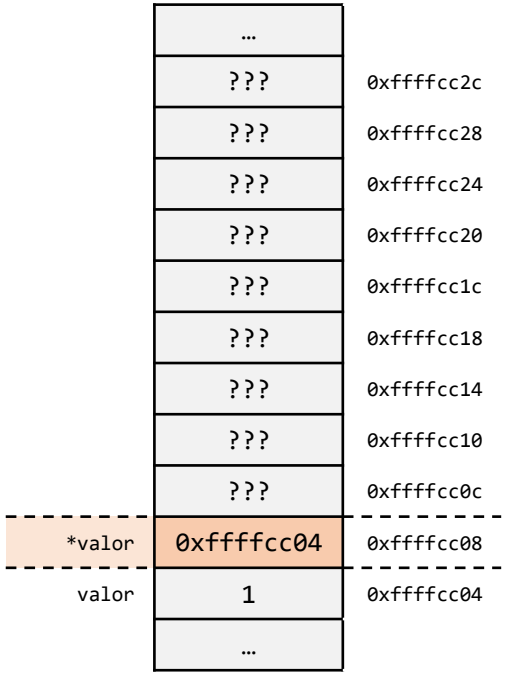

```
…
int main() {
    int valor = 0;

    incrementar(&valor);

    printf("\nvalor: %d", valor);

    return 0;
}
…
```

0xffffcc04

```
void incrementar(int *valor) {
    …
}
```

Passagem de argumentos por referência

- É atribuído o **endereço** de memória da variável passa como argumento para uma variável **local** à função.




```
…
int main() {
    int valor = 0;

    incrementar(&valor);

    printf("\nvalor: %d", valor);

    return 0;
}
…
```

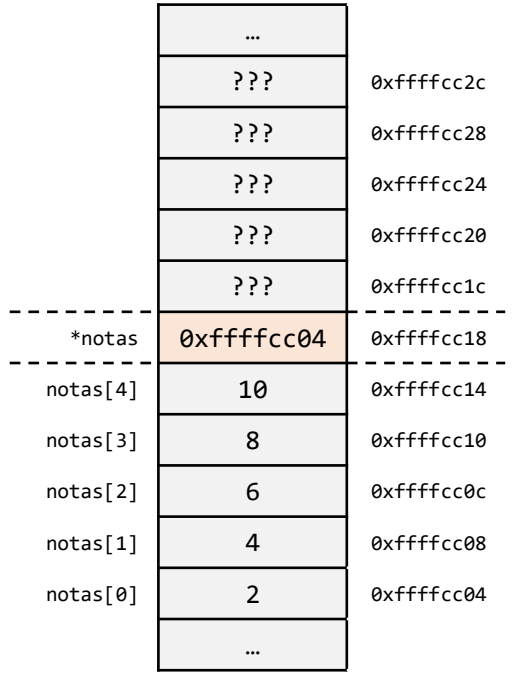
0xffffcc04

```
void incrementar(int *valor) {
   printf("\nvalor: %d", ++(*valor));
}
```

```
valor: 1
valor: 1
```

Passagem de argumentos por referência

- Os arrays são sempre passos por referência.




```
…
int main() {
   int i, notas[] = {1, 2, 3, 4, 5};
   dobrarNotas(notas);
   for (i = 0; i < TAMANHO; ++i) {
    printf("\n%d: %d", i, notas[i]);
   }
}
…
```

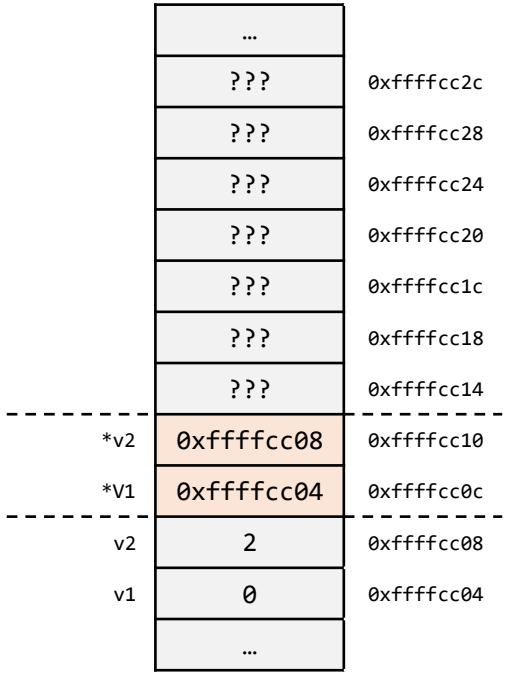
0xffffcc04

```
void dobrarNotas(int notas[]) {
   int i;
   for (i = 0; i < TAMANHO; ++i) {
      notas[i] *= 2;
   }
}
```

```
0: 2
1: 4
2: 6
3: 8
4: 10
```

Funções retornam apontadores

- É possível uma função retornar um apontador.
  - Deverá retornar:
    - Um endereço de memória
    - **NULL** (quando aplicável, ex.: exceção)




```
int main() {
    int v1 = 0, v2 = 1;
    *(getMax(&v1, &v2)) += 1;
    printf("%d %d", v1, v2);
}
```

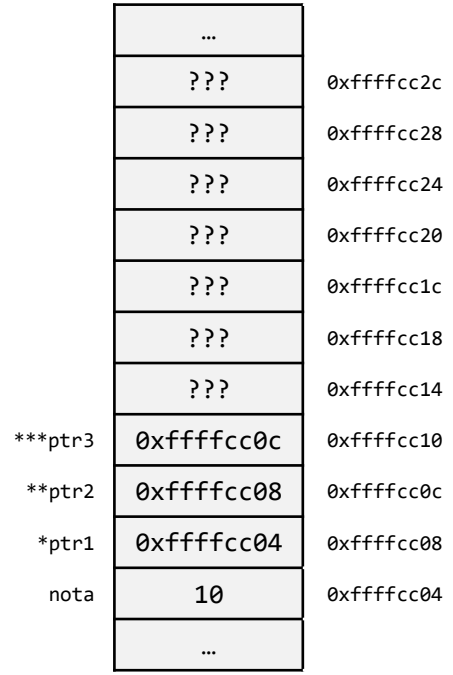
0xffffcc08

```
int *getMax(int *v1, int *v2) {
    return *v1 >= *v2 ? v1 : v2;
}
```

```
0 2
```

Apontadores de apontadores

- É possível obter o endereço de memória de um apontador através do operador **&**.
- Armazenar o endereço de memoria de um apontador, é semelhante a armazenar o endereço de uma variável utilizando o operador *.
  - Exemplo
    - `int **ptr; // apontador para o apontador ptr`




```
…
int nota = 10, *ptr1, **ptr2, ***ptr3;

ptr1 = &nota;
ptr2 = &ptr1;
ptr3 = &ptr2;

printf("\nnota:\n%d <%p>", nota, &nota);
printf("\nnotaPtr:\n%p <%p>", ptr1, &ptr1);
printf("\nnotaPtrPtr:\n%p <%p>", ptr2, &ptr2);
printf("\nnotaPtrPtrPtr:\n%p <%p>", ptr3, &ptr3);
…
```

```
nota:
10 <0xffffcc04>
ptr1:
0xffffcc04 <0xffffcc08>
ptr2:
0xffffcc08 <0xffffcc0c>
ptr3:
0xffffcc0c <0xffffcc10>
```

Leitura recomendada

- https://denniskubes.com/2017/01/24/the-5-minute-guide-to-c-pointers/
- (Capítulo 8, 9) Damas, L. Linguagem C; FCA – Editora de Informática, Lda, 1999; ISBN 9789727221561.