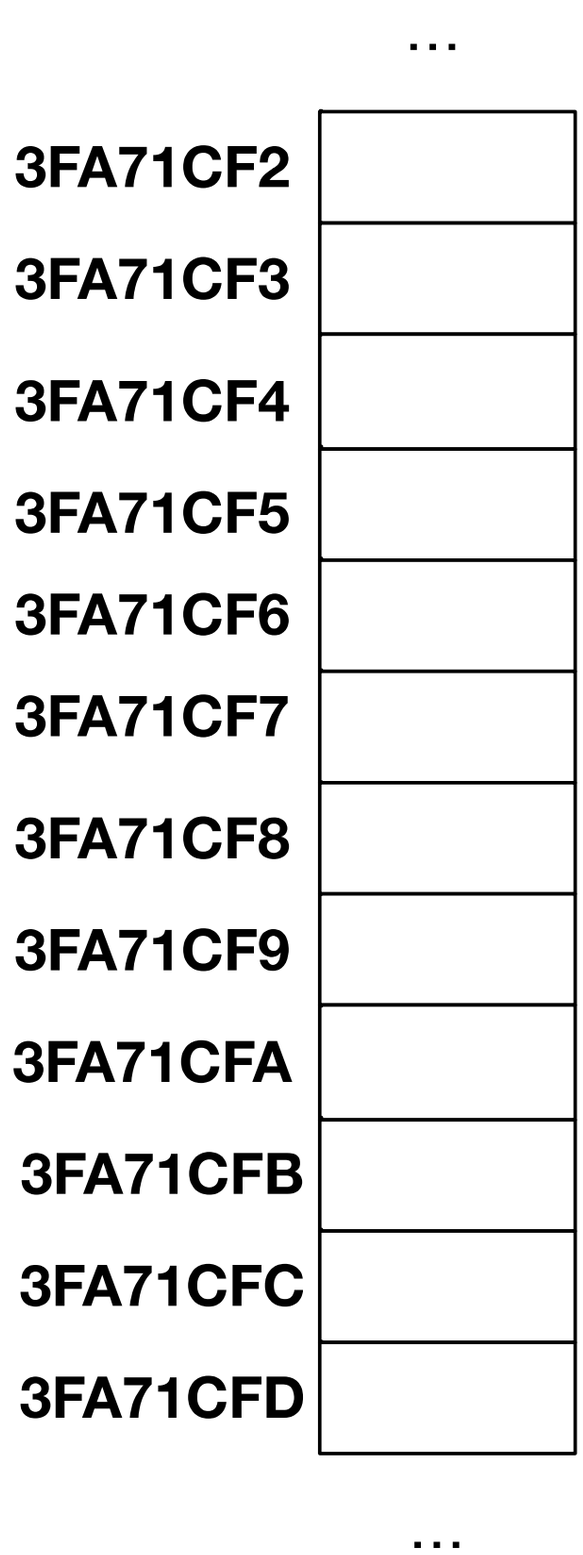
Ponteiros

Algoritmos e Programação 2
Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

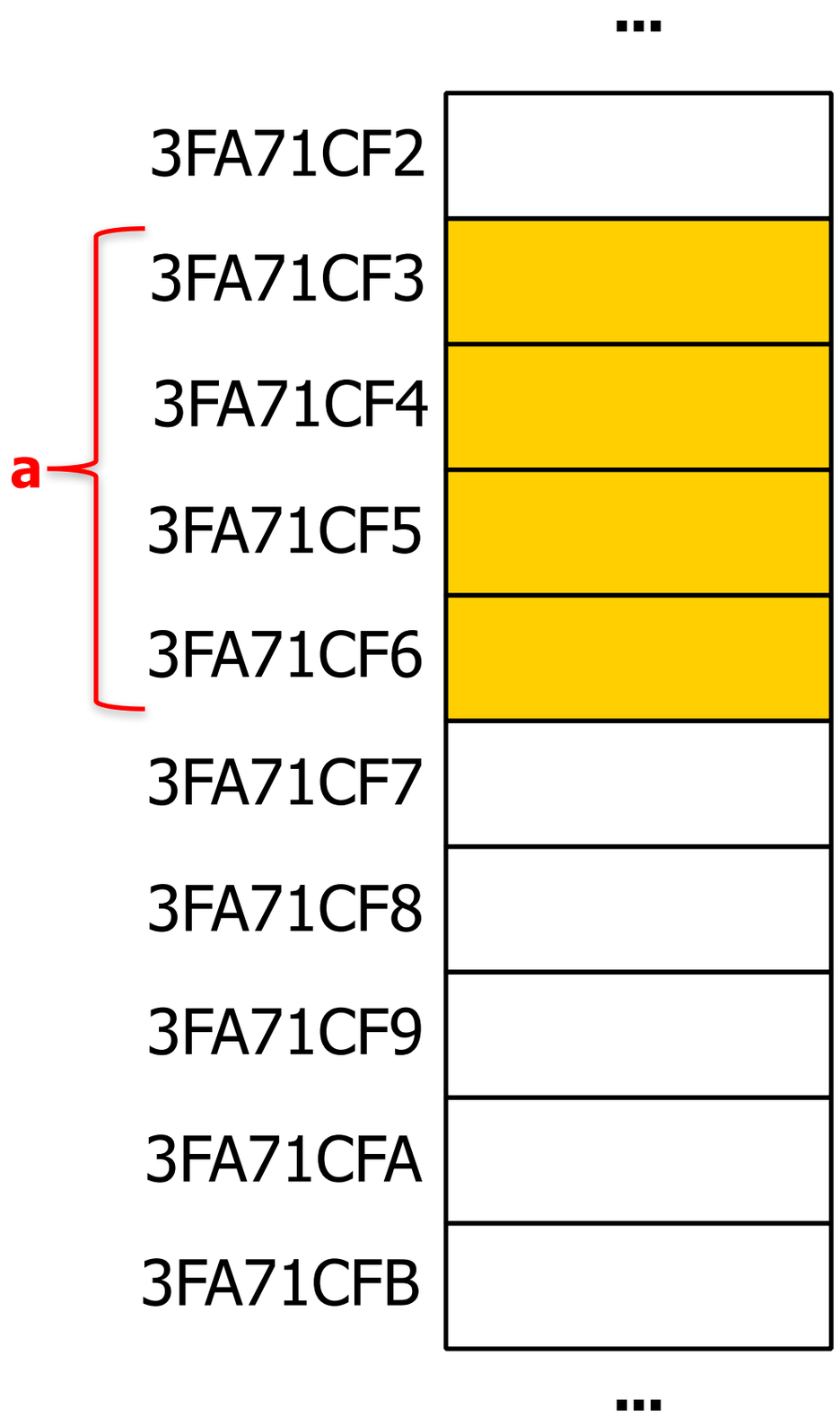
Introdução

- A memória do computador é organizada como uma tabela
- Imagine um computador hipotético onde a memória do computador é endereçada em bytes
- Cada linha da tabela possui um endereço, e em cada endereço podemos armazenar um byte
- Cada variável, ao ser criada, é associada a um endereço



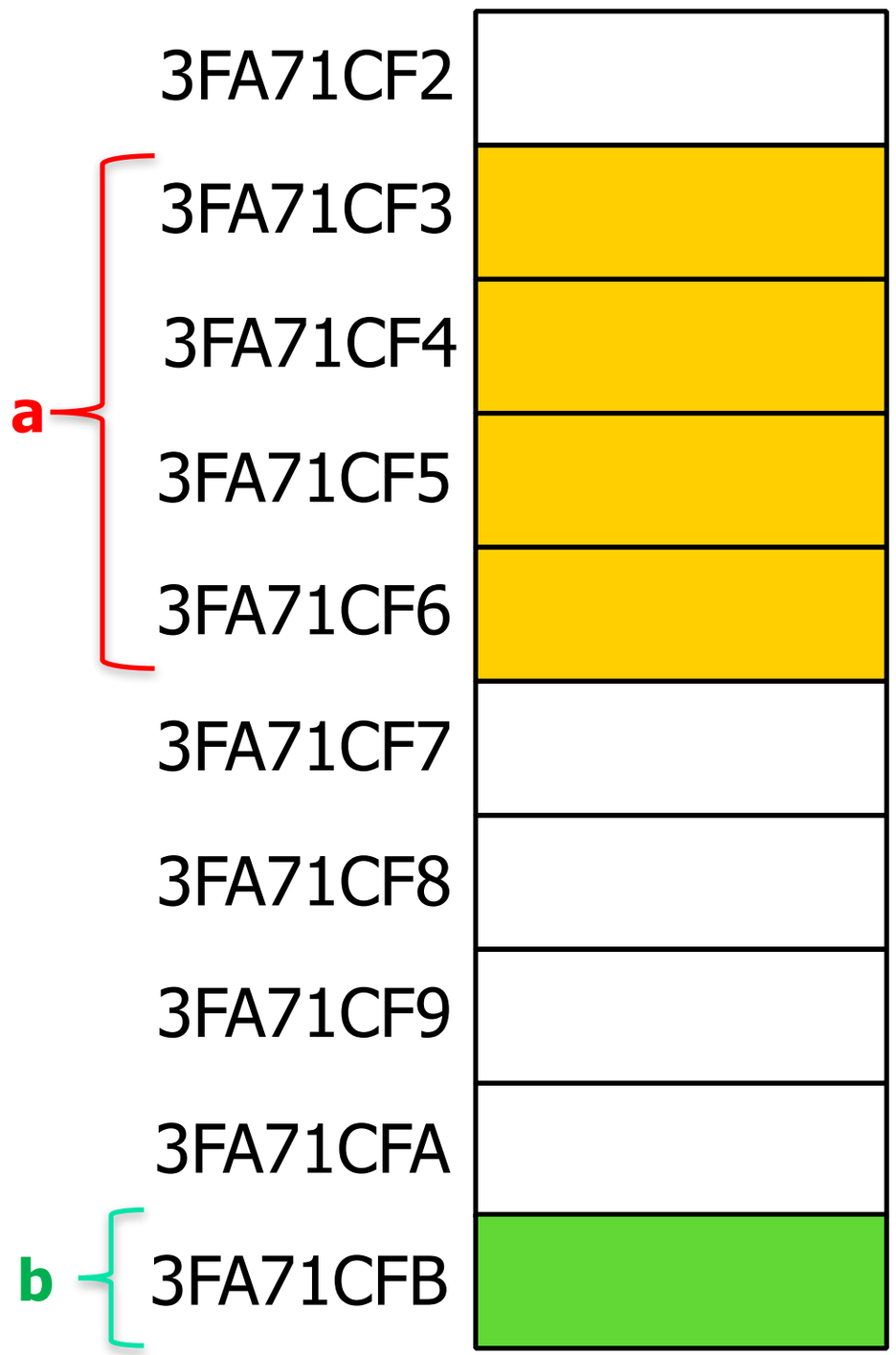
Variáveis na Memória

```
int main () {
    int a;

    return 0;
}
```



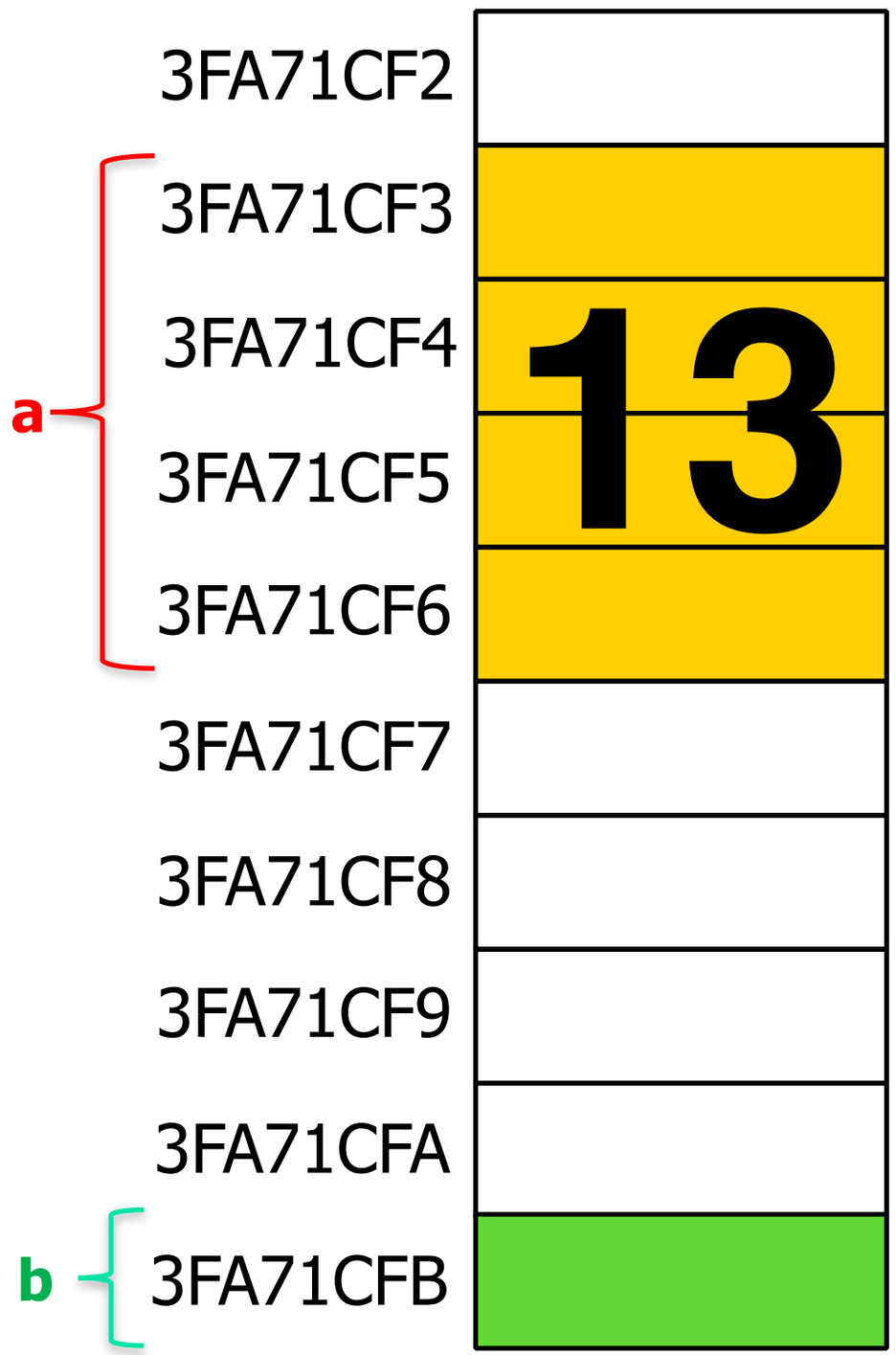
Variáveis na Memória (cont.)

```
int main () {
    int a;
    char b;

    return 0;
}
```



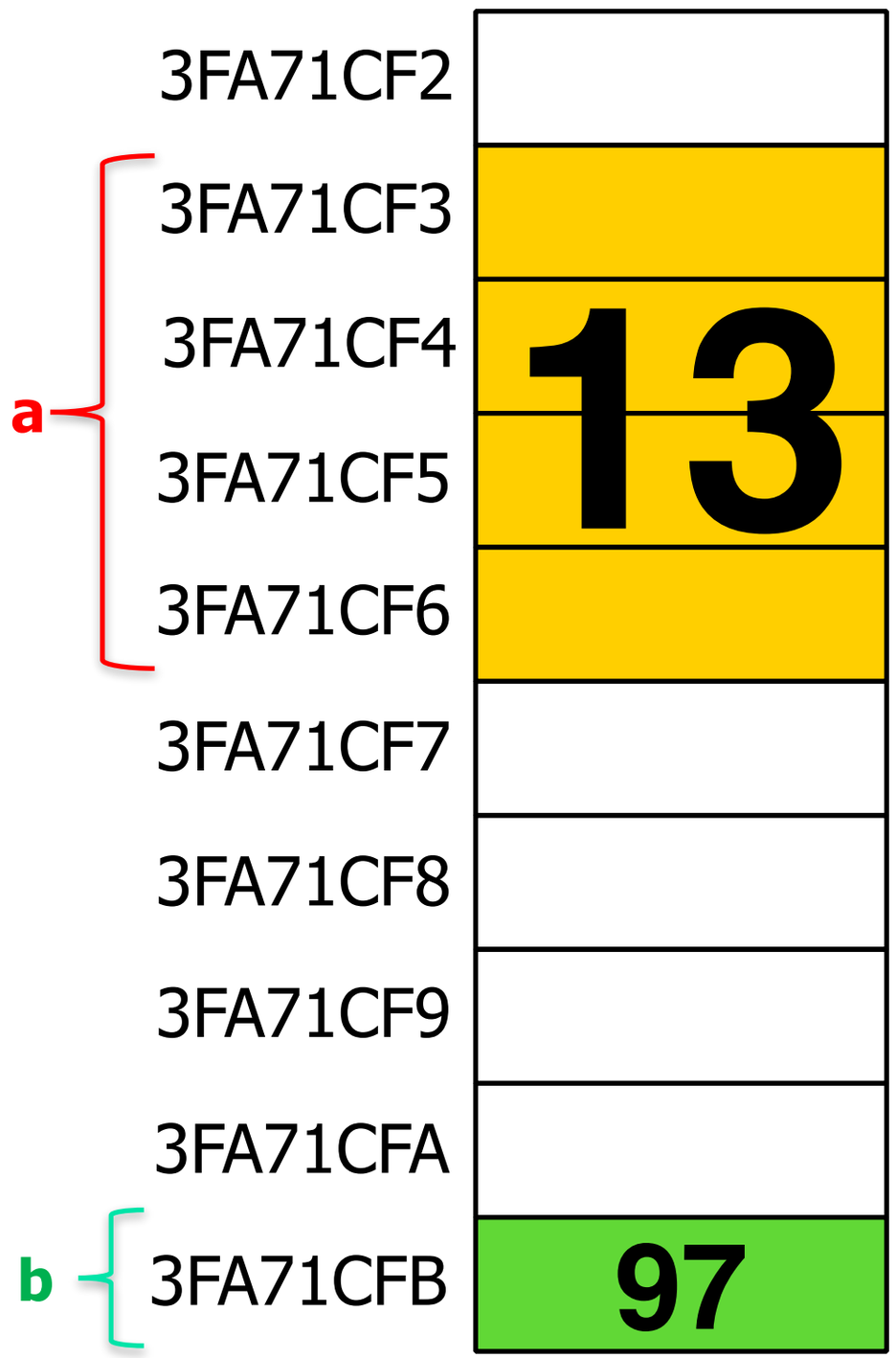
Variáveis na Memória (cont.)

```c
int main () {
    int a;
    char b;

    a = 13;

    return 0;
}
```



Variáveis na Memória (cont.)

```c
int main () {
    int a;
    char b;

    a = 13;
    b = 'a';

    return 0;
}
```



Espaço Alocado na Memória

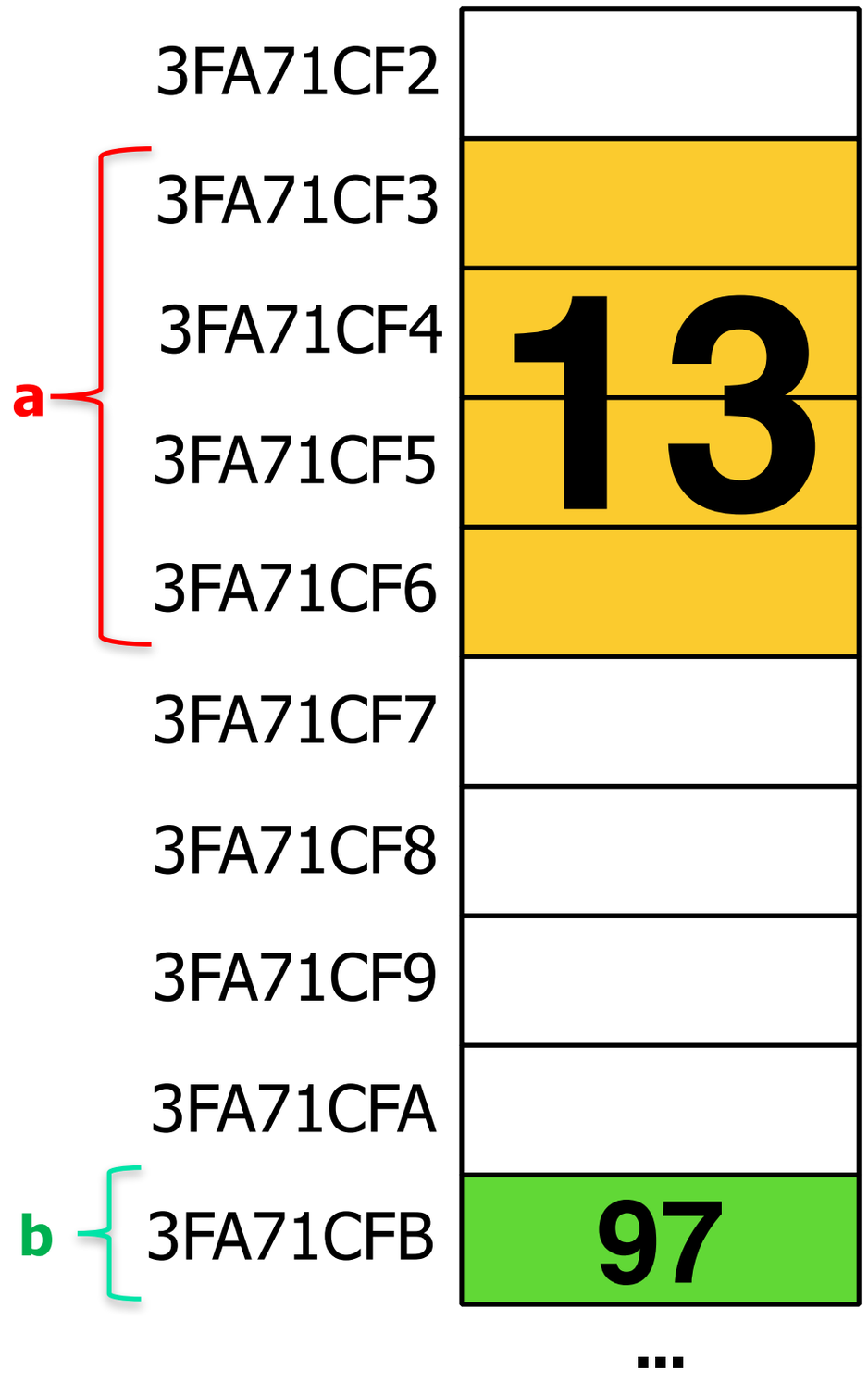
Tipo	**Bits**	**Bytes**	**Escala**
char	8	1	128 a 127
int	32	4	-2.147.483.648 a 2.147.483.647 (ambientes de 32 bits)
short	16	2	-32.765 a 32.767
long	32	4	-2.147.483.648 a 2.147.483.647
unsigned char	8	1	0 a 255
unsigned	32	4	0 a 4.294.967.295 (ambientes de 32 bits)
unsigned long	32	4	0 a 4.294.967.295
unsigned short	16	2	0 a 65.535
float	32	4	$3{,}4 \times 10^{-38}$ a $3{,}4 \times 10^{38}$
double	64	8	$1,7 \times 10^{-308}$ a $1,7 \times 10^{308}$
long double	80	10	$3,4 \times 10^{-4932}$ a $3,4 \times 10^{4932}$
void	0	0	nenhum valor

Tipos de variáveis em C.

Conteúdo e Endereço da Variável ...

```c
int main () {
   int a;
   char b;

   a = 13;
   b = 'a';

   printf("%d\n", a);


   return 0;
}
```

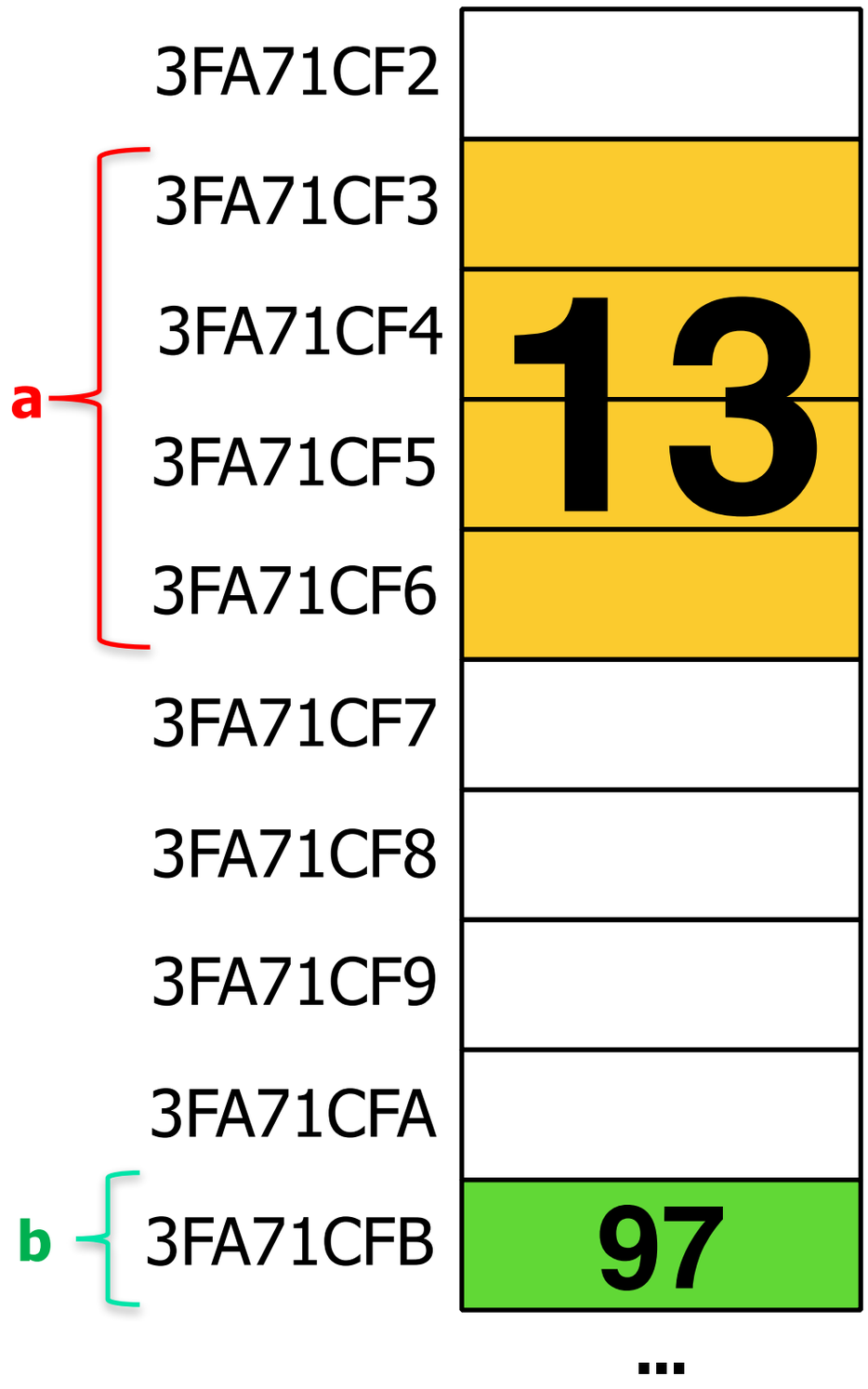
```
13
```



Conteúdo e Endereço da Variável (cont...)

```c
int main () {
   int a;
   char b;

   a = 13;
   b = 'a';

   printf("%d\n", a);
   printf("%p\n", &a);

   return 0;
}
```

```
13
3FA71CF3
```



Ponteiros

- **Ponteiro** é um tipo de variável que armazena um endereço
- Para que serve?
  - Simular chamada por referência
    - Obs.: em C++ existe uma maneira mais fácil de passar por referência
  - Alocar memória dinamicamente
  - Manipularmos estruturas de dados dinâmicas

Ponteiros: Declaração

- A declaração de um ponteiro é feita com a adição do **símbolo *** (asterisco) na declaração
- **p** é um ponteiro para **int**

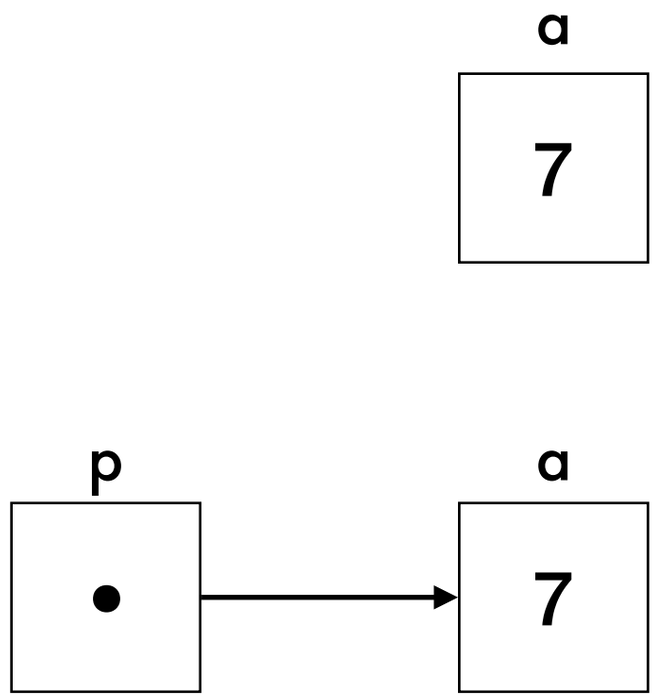
```
int main () {
    int *p, a;

    return 0;
}
```

Erro Comum de Programação 1

A notação asterisco (*) utilizada para declarar ponteiros não é distribuída para todas variáveis da declaração. Cada ponteiro deve ser declarado com um * prefixado no nome; e.g., se você deseja declarar **xPtr** e **yPtr** como ponteiros do tipo inteiro, utilize:

```
int main () {
   int *xPtr, *yPtr;

   return 0;
}
```

Referência Direta e Indireta



a diretamente
referencia uma variável
cujo valor é 7

p indiretamente
referencia uma variável
cujo valor é 7

Ponteiros: Inicialização

- Ponteiros devem ser inicializados:
  - quando são declarados ou
  - em uma expressão de atribuição
- Um ponteiro pode ser inicializado com **0**, **NULL** ou um endereço
  - Um ponteiro com o valor **0** ou **NULL** aponta para nada
  - Para obtermos endereços de variáveis utilizamos o operador **&**

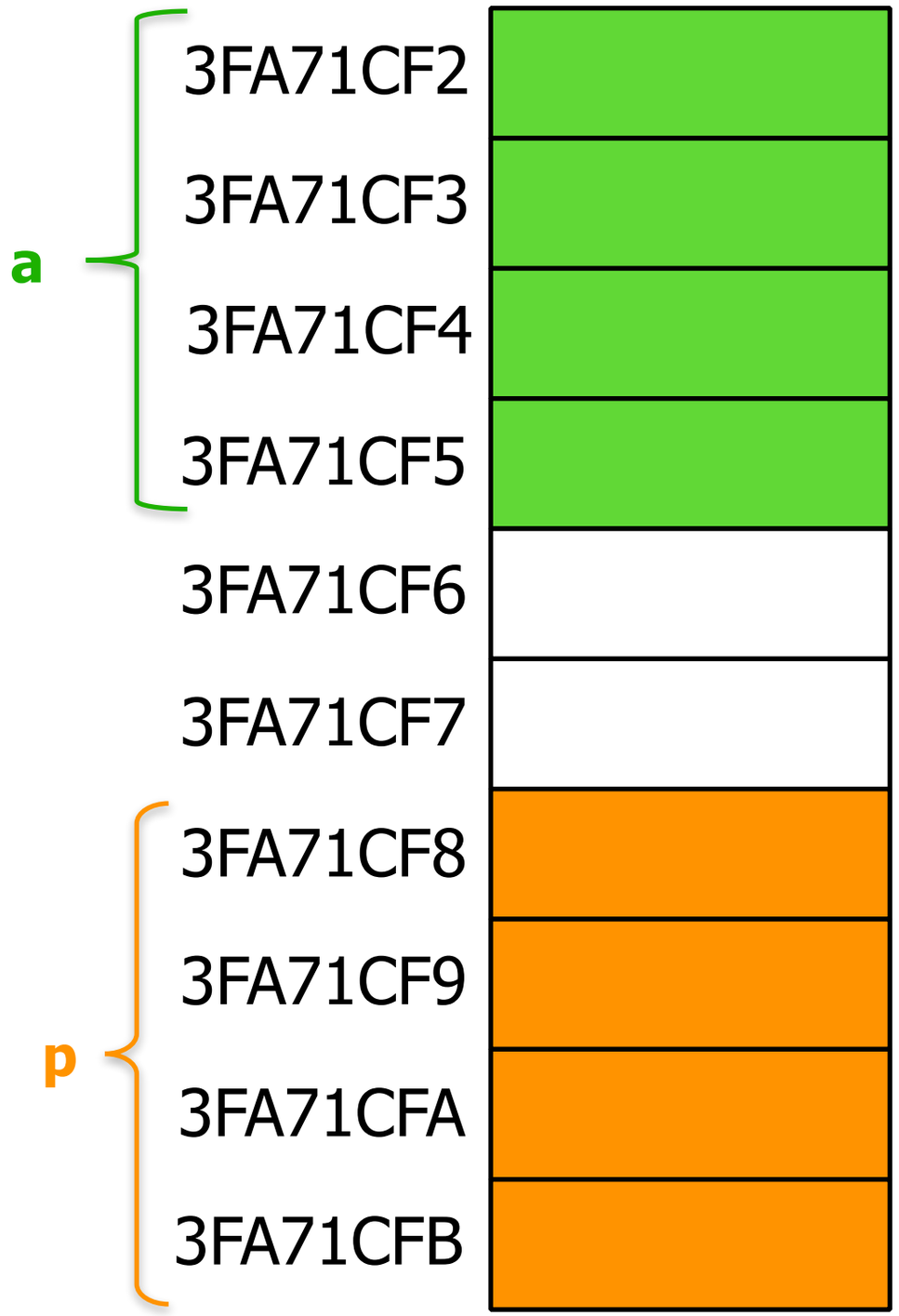
Ponteiros: Inicialização (cont.)

Assim, para fazermos o nosso ponteiro **p** receber o endereço de uma variável **a**:

```
int main () {
   int a, *p;
   a = 3;
   p = &a;
   return 0;
}
```

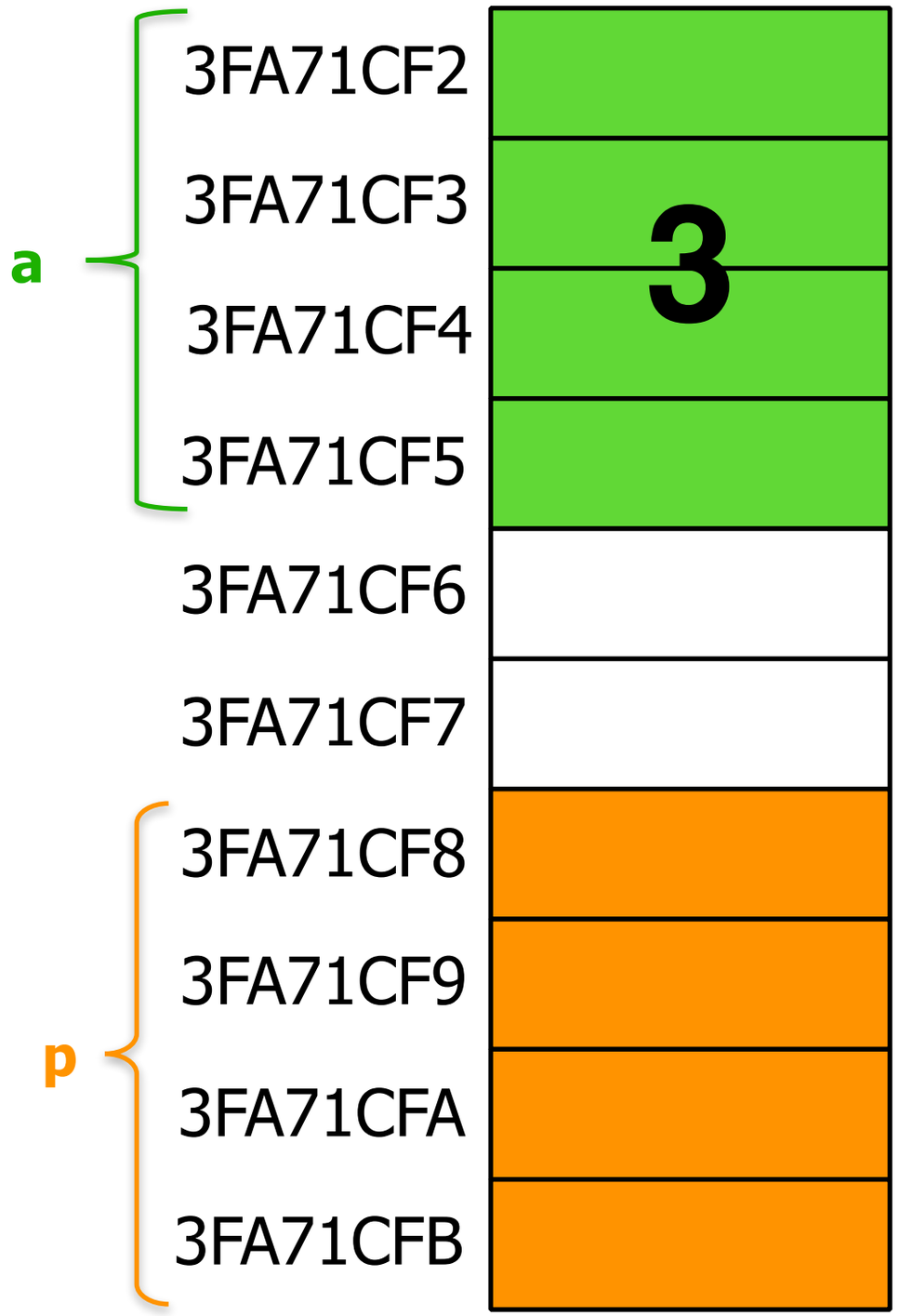
Atribuição Endereço a Ponteiro (cont...)

```
int main () {
    int a, *p;

    a = 3;
    p = &a;
    return 0;
}
```



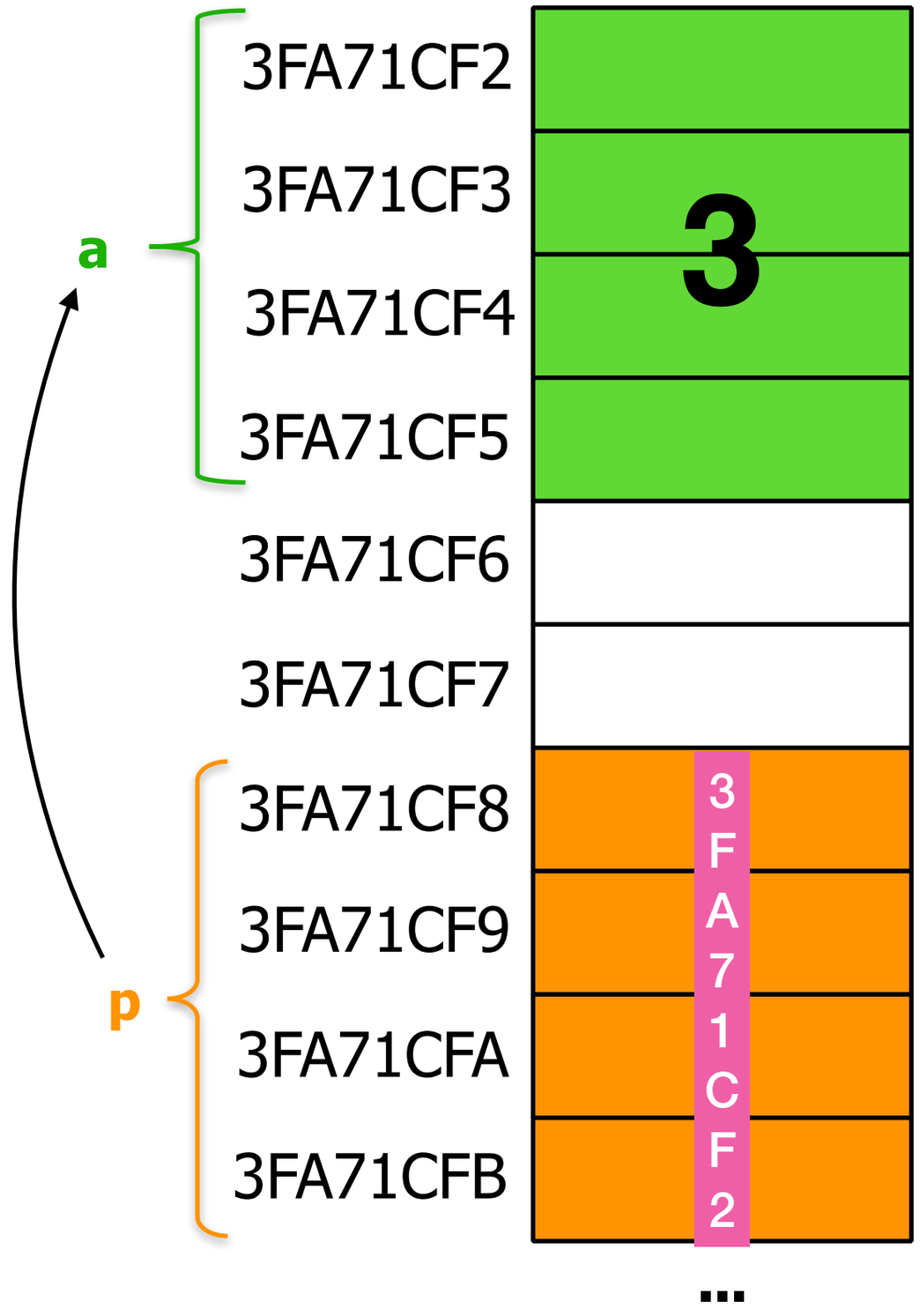
Atribuição Endereço a Ponteiro (cont...)

```
int main () {
   int a, *p;

   a = 3;
   p = &a;
   return 0;
}
```



Atribuição Endereço a Ponteiro (cont…)

```c
int main () {
    int a, *p;

    a = 3;
    p = &a;
    return 0;
}
```



Dizemos que o ponteiro **p** aponta para **a**.

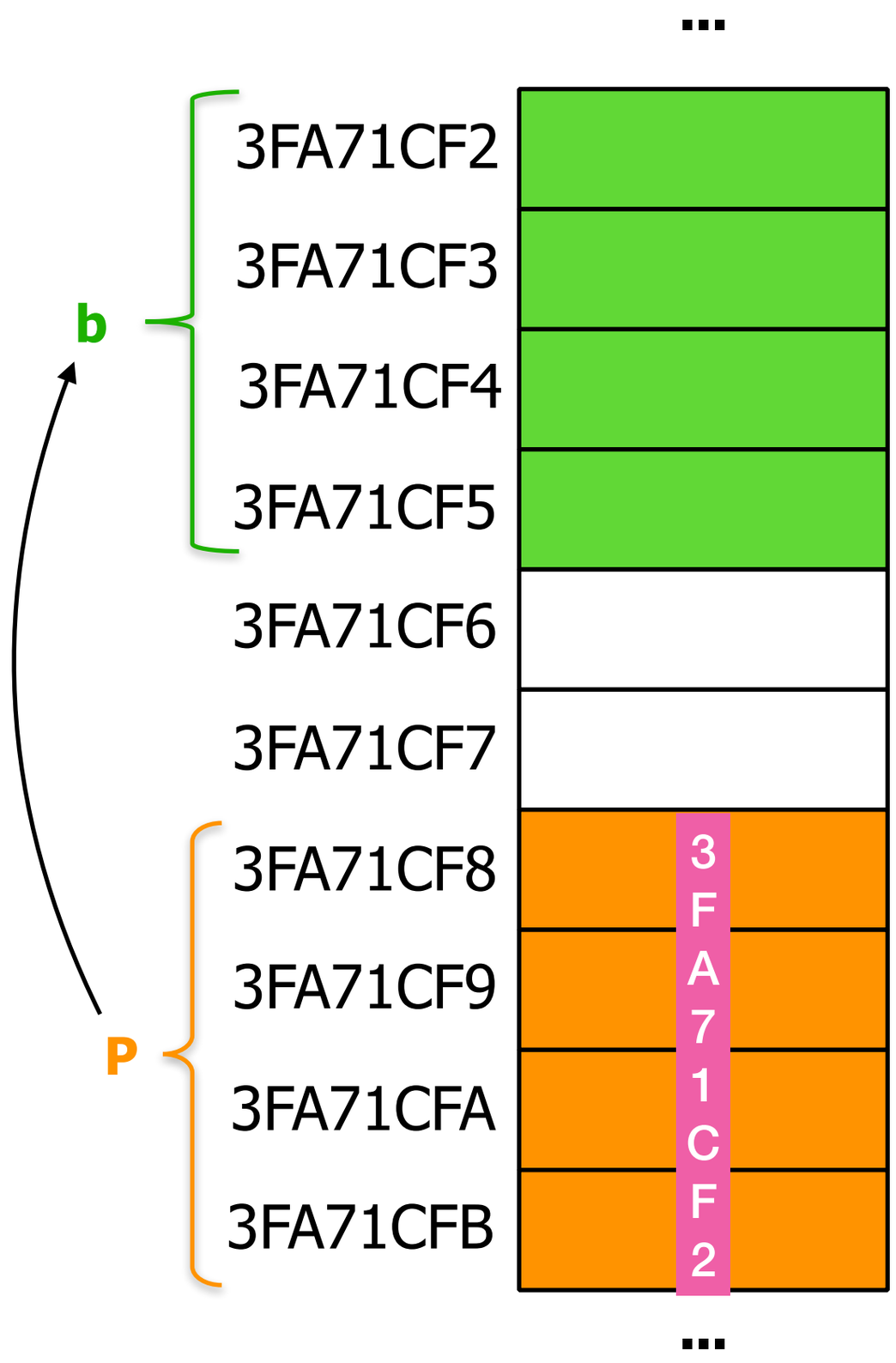
Operador de Indireção (*)

- O **operador unário *** é conhecido como operador de indireção, e retorna o valor de um objeto no qual o ponteiro aponta
  - Retorna um sinônimo para o objeto no qual o seu operando (ponteiro) aponta
- * e & são inversos
  - Cancelam um ao outro

Conteúdo Ponteiro

Diferente do que ocorre com atribuição de variáveis simples ..

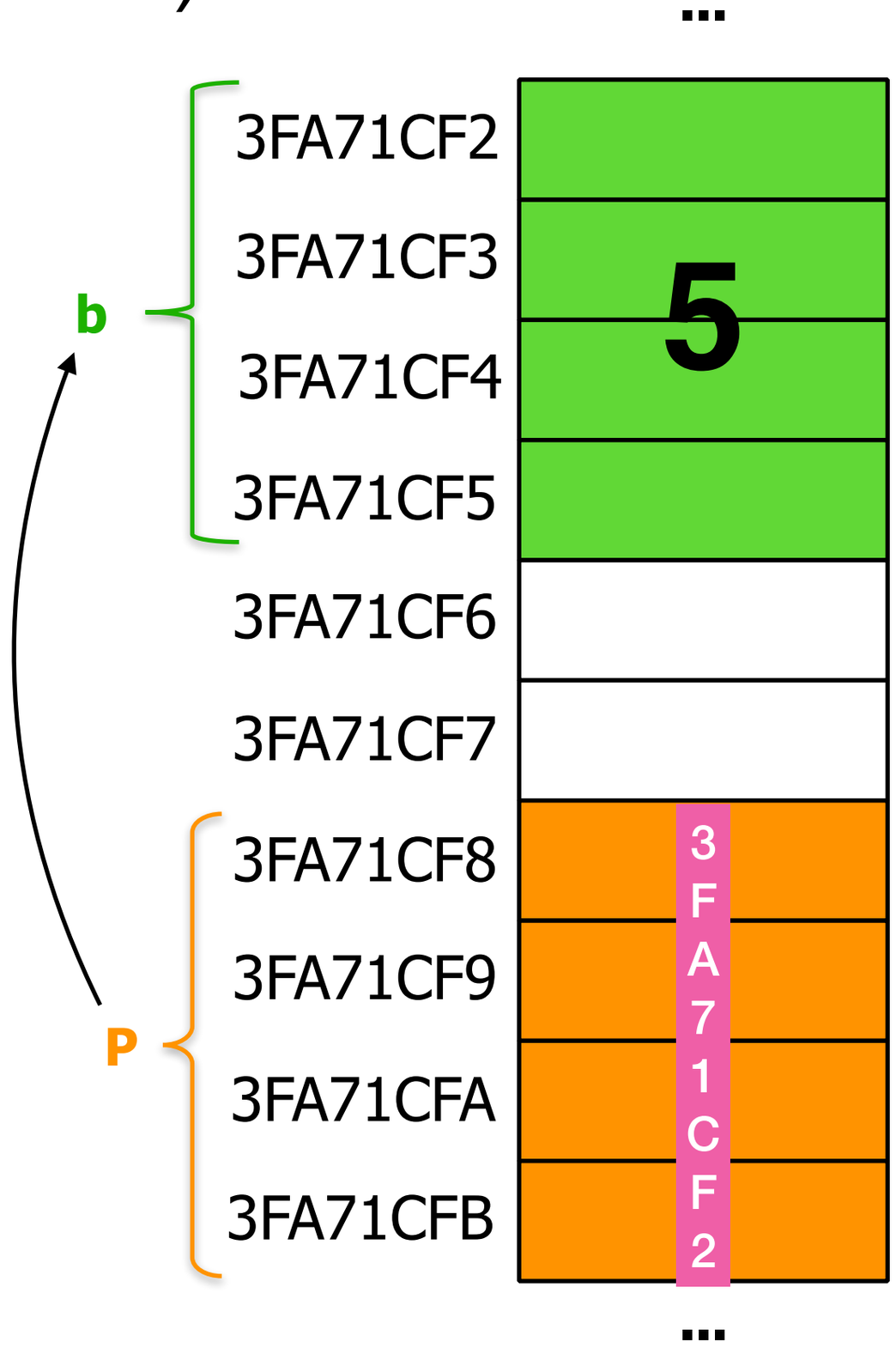
```
int main () {
   int a, *p;

   p = &a;
   a = 5;
   *p = 3;
   return 0;
}
```



Conteúdo Ponteiro (cont.)

Diferente do que ocorre com atribuição de variáveis simples ..

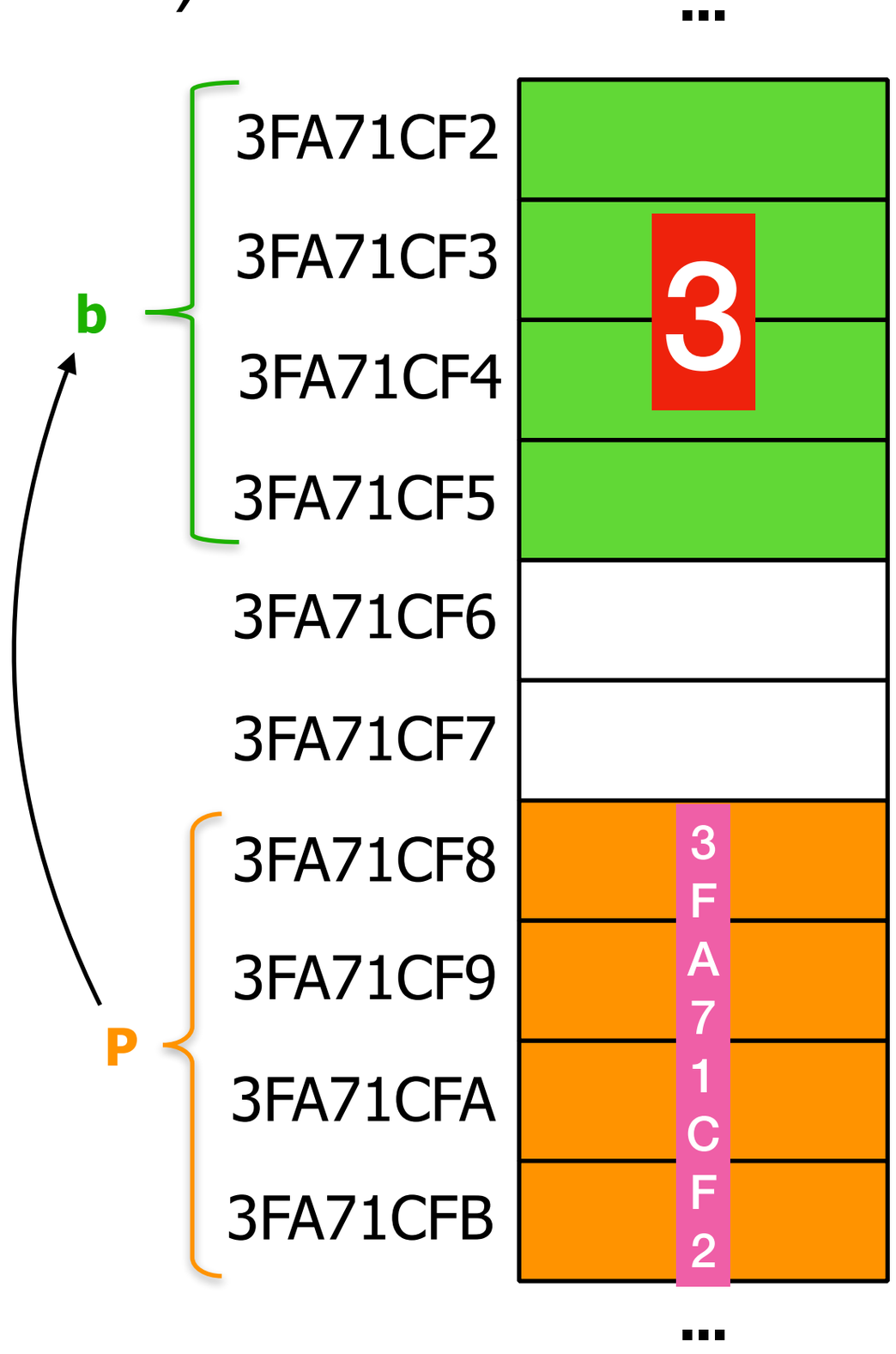
```
int main () {
   int a, *p;

   p = &a;
   a = 5;
   *p = 3;
   return 0;
}
```



Conteúdo Ponteiro (cont.)

Diferente do que ocorre com atribuição de variáveis simples ..

```
int main () {
   int a, *p;

   p = &a;
   a = 5;
   *p = 3;
   return 0;
}
```



Erro Comum de Programação 2

Referenciar um ponteiro que não foi inicializado adequadamente ou que não foi atribuído para apontar para um local específico na memória é um erro. Isto pode causar um erro fatal em tempo de execução, ou pode acidentalmente modificar importantes dados e permitir que o programa execute até o fim com resultados incorretos.

Erro Comum de Programação 3

Utilizar o operador de indireção em uma variável não ponteiro é um erro sintático.

Erro Comum de Programação 4

Desreferenciar (aplicar o operador de indireção) em um ponteiro que possui o valor **0** (**NULL**) gera um erro fatal em tempo de execução.

Operadores & e *

```c
#include <stdio.h>

int main ()
{
   int a, *aPtr;

   a = 7;
   aPtr = &a;

   printf("O endereco de a eh %p\n"
        "\nO valor de aPtr eh %p", &a, aPtr);

   printf("\n\nO valor de a eh %d"
        "\nO valor de *aPtr eh %d", a, *aPtr);

   printf("\n\nMostrando que * e & sao complementos"
        "\n&*aPtr = %p"
        "\n*&aPtr = %p\n", &*aPtr, *&aPtr);

   return 0;
}
```

Se aPtr aponta para a, então &a e aPtr tem o mesmo valor.

a e *aPtr tem o mesmo valor

&*aPtr e *&aPtr tem o mesmo valor

Operadores & e * (cont.)

```
O endereco de a e 0x16d84b2c8
O valor de aPtr e 0x16d84b2c8

O valor de a e 7
O valor de *aPtr e 7

Mostrando que * e & sao complementos
&*aPtr = 0x16d84b2c8
*&aPtr = 0x16d84b2c8
```

Alocação de Memória

Alocação Estática

- As declarações abaixo alocam espaço de memória para diversas variáveis

```
char c;
int i;
int v[10];
```

- A **alocação é estática**, ou seja, acontece antes que o programa comece a ser executado

Alocação Dinâmica

- Uma outra opção é ao invés do ponteiro apontar para o endereço de uma outra variável, **alocarmos a nossa própria memória** para o ponteiro
- Isso é possível usando dos comandos **malloc** (para criar a memória) e **free** (para liberar a memória)
- Estes comandos presentes na biblioteca **stdlib**

Exemplo Malloc e Free

```c
/* malloc example: random string generator*/
#include <stdio.h>      /* printf, scanf */
#include <stdlib.h>     /* malloc, free */
#include <time.h>

int main () {
    int i,n;
    char * buffer;

    srand(time(NULL));

    printf("Qual o tamanho da string que deseja? ");
    scanf("%d", &i);

    buffer = (char *) malloc(sizeof(char)*(i+1));
    if (buffer == NULL)
        exit (1);

    for (n = 0; n < i; n++)
        buffer[n] = rand()%26+'a';
    buffer[i]='\0';

    printf("Random string: %s\n", buffer);
    free(buffer);
    return 0;
}
```

Aloca uma área de memória e retorna o endereço

A memória alocada dinamicamente deve ser SEMPRE liberada explicitamente antes de o programa ser finalizado.

Exemplo Malloc e Free: Saída

```
./a.out
Qual o tamanho da string que deseja? 10
Random string: lrfkqyuqfj
```

Casting

- Ponteiros do mesmo tipo podem ser atribuídos uns aos outros
  - Se não for do mesmo tipo, um operador **cast** deve ser utilizado
  - Exceção: ponteiro para **void** (tipo **void ***)
  - Nenhum **cast** necessário para converter um ponteiro para **void pointer**

```
(type_name) expression
```

Exemplo: Alocação dinâmica

- Escreva um programa que dinamicamente aloca uma região de memória suficiente para armazenar *n* inteiros. Então, solicite ao usuário entrar com *n* inteiros e armazene-os no novo local de memória alocado. Finalmente, imprima os valores inteiros na ordem inversa.

Exemplo: Alocação dinâmica

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int i, *v, n;

  printf("Entre com a quantidade de inteiros a serem lidos: ");
  scanf("%d", &n);

  // alocando memoria dinamicamente
  v = (int *) malloc(sizeof(int)*n);

  // lendo n inteiros
  printf("Entre com %d inteiros: ", n);
  for(i = 0; i < n; i++)
    scanf("%d", &v[i]);

  // imprimindo na ordem inversa da leitura
  printf("Valores impressos na ordem inversa\n");
  for(i = n-1; i >= 0; i--)
    printf("%d ", v[i]);

  free(v);
  return 0;
}
```

Passagem de Parâmetros

Passagem de Parâmetros

Há três maneiras em C++ de passar argumentos para uma função:

1. Passagem por valor
2. Passagem por referência com argumentos de referência (somente em C++)
3. Passagem por referência com argumentos de ponteiro

*pass by **reference***

cup =

```
fillCup(      )
```

*pass by **value***

cup =

```
fillCup(      )
```

www.penjee.com

Passagem por Valor

- É a forma mais comum de passagem de parâmetros
- Uma cópia do valor do argumento é feita e passada para a função chamada
- Alterações na cópia não afetam valor variável original

Outline

fig07_06.c

```c
/* Fig. 7.6: fig07_06.c
   Cube a variable using call-by-value */
#include <stdio.h>

int cubeByValue( int n ); /* prototype */

int main( void )
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass number by value to cubeByValue */
   number = cubeByValue( number );

   printf( "\nThe new value of number is %d\n", number );

   return 0; /* indicates successful termination */

} /* end main */

/* calculate and return cube of integer argument */
int cubeByValue( int n )
{
   return n * n * n;   /* cube local variable n and return result */

} /* end function cubeByValue */
```

```
The original value of number is 5
The new value of number is 125
```

Passagem por Referência

- Há situações que desejamos criar uma função que retorne mais de 2 variáveis
- As funções que vimos até agora são capazes somente de retornar um único valor

Passagem por Referência com Argumentos de Referência

- Um parâmetro de referência é um *alias* para seu argumento correspondente em uma chamada de função
- Adicione um ‘e comercial’ (**&**) depois do tipo do parâmetro no protótipo de função
- Na chamada de função, simplesmente mencione a variável por nome para passá-la por referência

```c
#include <stdio.h>

int squareByValue(int n);
void squareByReference(int &n);

int main() {
    int x = 2;
    int z = 4;

    // demonstra squareByValue
    printf("x = %d antes de squareByValue\n", x);
    printf("Valor retornado por squareByValue: %d\n", squareByValue(x));
    printf("x = %d depois squareByValue\n\n", x);

    // demonstra squareByReference
    printf("z = %d antes de squareByReference\n", z);
    squareByReference(z);
    printf("z = %d depois squareByReference\n", z);

    return 0;
}

int squareByValue(int n) {
    return n = n * n;
}

void squareByReference(int &nRef) {
    nRef = nRef * nRef;
}
```

```
x = 2 antes de squareByValue
Valor retornado por squareByValue: 4
x = 2 depois squareByValue

z = 4 antes de squareByReference
z = 16 depois squareByReference
```

Passagem por Referência com Argumentos de Ponteiro

- Em C++, os programadores podem utilizar ponteiros e o operador de indireção (*) para realizar a passagem por referência
  - Exatamente como a passagem por referência é feita em programas C, porque o C não tem referências

Outline

fig07_07.c

```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call by reference with a pointer argument */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* prototype */
7
8  int main( void )
9  {
10    int number = 5; /* initialize number */
11
12    printf( "The original value of number is %d", number );
13
14    /* pass address of number to cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nThe new value of number is %d\n", number );
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
22
23 /* calculate cube of *nPtr; modifies variable number
24 void cubeByReference( int *nPtr )
25 {
26    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
27 } /* end function cubeByReference */
```

Protótipo da função recebe como argumento um ponteiro

Função **cubeByReference** é passado um endereço, que pode ser o valor de uma variável ponteiro

Neste programa, ***nPtr** é **number**, assim esta expressão modifica o valor de **number**.

```
The original value of number is 5
The new value of number is 125
```

Referências

- DEITEL, C Como Programar 6ª Edição. Pearson;
- FEOFILOFF, P. Algoritmos em Linguagem C, 1. ed. Rio de Janeiro: Elsevier, 2008;
- PIVA, D.J. et al. Algoritmos e programação de computadores. Rio de Janeiro: Elsevier, 2012;
- WIRTH, Niklaus. Algoritmos e estrutura de dados. Rio de Janeiro, RJ: LTC, 2009. 255p;