AED - Algoritmos e Estruturas de Dados

Aula 1 – Apresentação da Disciplina

Prof. Rodrigo Mafort

Aulas

- Terças:
  - 10h40 até 12h20
  - Local: Laboratório de Informática

- Sextas:
  - 09h40 até 12h20
  - Local: Sala 218

Avaliações

- Duas Provas: P1 e P2

- Dois Trabalhos:
  - Tema à definir
  - Árvores auto Ajustáveis

- Avaliação de Reposição:
  - Única para o período – conteúdo todo
  - Requisição segue as regras da UERJ

- Média Semestral: $\frac{2 * P1 + 2 * P2 + T1 + T2}{6}$

- Prova final: PF

Regras

- A presença nas aulas será contabilizada (por listas de presença)
- Frequência mínima (regimento da UERJ): 75% das aulas

- Aprovação (alunos com presença >= 75%):
  - Média Semestral >= 7.0 (aprovação direta - sem prova final)
  - (Média Semestral + PF) / 2 >= 5.0 (aprovado com prova final)

- Reprovação:
  - Presença < 75% (sem direito a prova final)
  - Média Semestral < 4.0 (sem direito a prova final)
  - (Média Semestral + PF) / 2 < 5.0

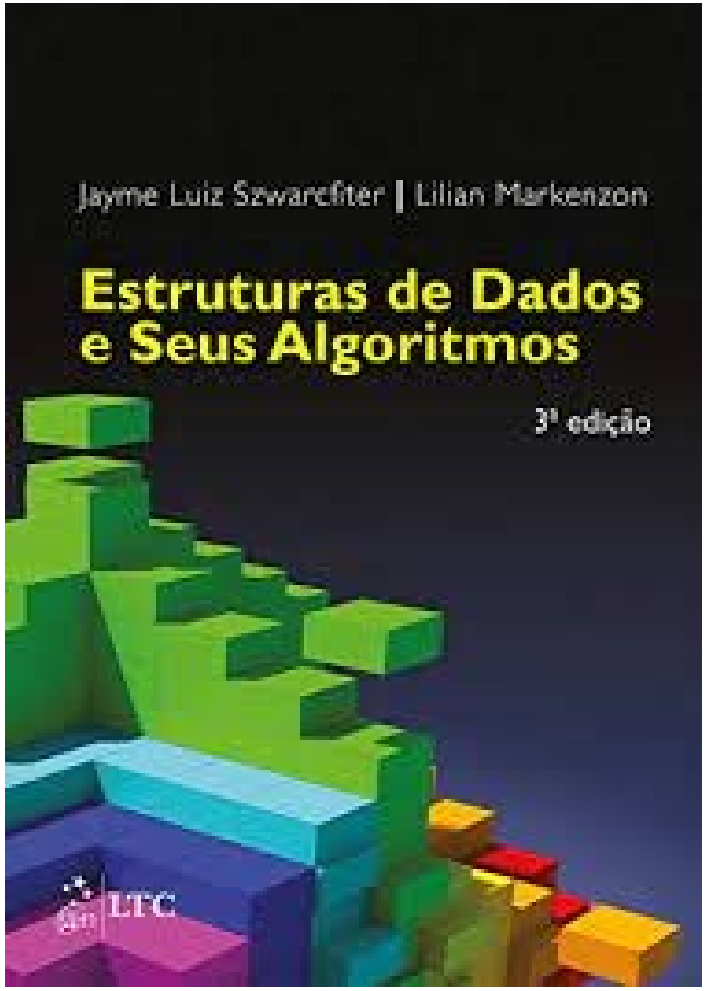
Google Classroom

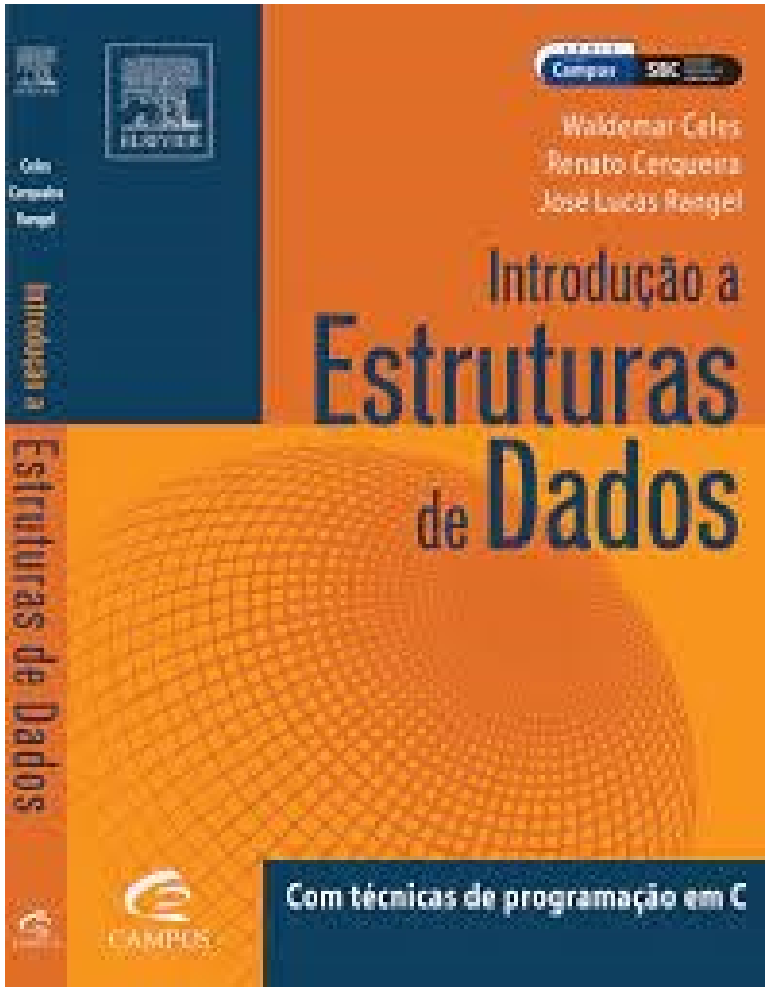
- A disciplina conta com uma página no Google Classroom.
- Link: https://bit.ly/42QdDXu

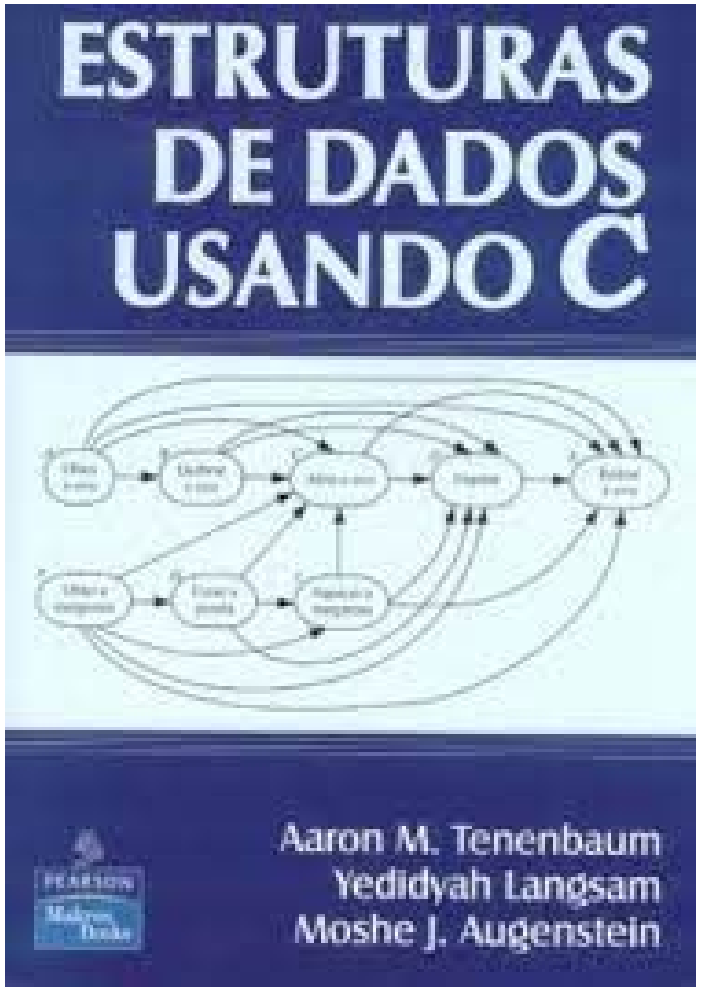
- O conteúdo e avisos importantes serão postados apenas na página da disciplina.

- Por favor, ingresse e acompanhe a disciplina.

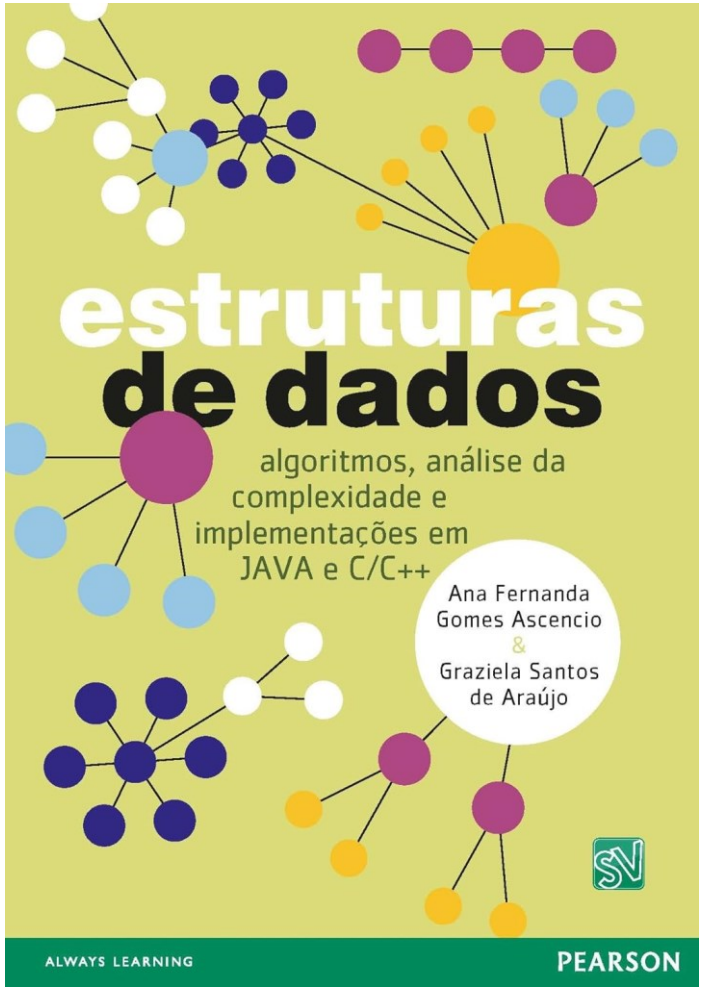
Bibliografia







Bibliografia



AED - Algoritmos e Estruturas de Dados

Tópicos Abordados

1. Programação Orientada a Objetos
    1. Conceitos
    2. Encapsulamento
    3. Abstração
    4. Herança
    5. Polimorfismo
    6. Implementação em C++

2. Complexidade Computacional (conteúdo diluído ao longo do curso)

Tópicos Abordados

3. Listas sequenciais
    1. Listas não ordenadas: Definição, Inserção, Remoção e Busca
    2. Algoritmos de ordenação
    3. Listas ordenadas: Definição, Inserção, Remoção, Busca Sequencial e Busca Binária
    4. Filas e Pilhas: Definição, Inserção e Remoção

4. Estruturas encadeadas
    1. Listas encadeadas: Definição, Inserção, Remoção e Busca
    2. Listas duplamente encadeadas: Definição, Inserção, Remoção e Busca
    3. Listas circulares: Definição, Inserção, Remoção e Busca
    4. Árvores: Definição
    5. Árvores Binárias de Busca: Definição, Inserção, Remoção e Busca
    6. Árvores Auto Ajustáveis (AVL, Rubro-negra e B)
    7. Grafos e Algoritmos em Grafos: Busca, Busca em Largura, Busca em Profundidade, Algoritmos de Caminhos Mínimos

Prog. Orientada a Objetos e Est. de Dados

- Um dos tópicos desenvolvidos nessa disciplina é a Programação Orientada a Objetos.
- Existem duas opções em relação a POO e Estruturas de Dados:
  1. Aprender POO no final do curso, sem conexão com os tópicos de estruturas de dados
  2. Integrar POO com Estruturas de Dados. Para isso:
     - Vamos começar com POO
     - As implementação das Estruturas de Dados será em C++, já usando POO (por exemplo, construindo classes para cada ED e usando classes no lugar de structs).
- Proponho usar a segunda opção nessa disciplina.

Por que estudar Estruturas de Dados?

- Suponha que você está escrevendo um sistema para controlar o cadastro de funcionários de uma empresa.

- Quantos funcionários seu sistema deverá suportar?
  - O que aconteceria se eu definisse uma lista com capacidade para 100 pessoas e a empresa contratasse mais um funcionário?
  - Por outro lado, o que aconteceria se a lista tivesse capacidade para 1.000.000 de funcionários e a empresa só contratasse 5?

- Suponha agora que a empresa tem 100.000.000 de funcionários.
  - Qual é o custo computacional para buscar o cadastro de um funcionário?
  - Busca Sequencial: Precisaríamos percorrer todo o cadastro e comparar um funcionário por vez.
  - Existe uma técnica de busca que pode reduzir o número de comparações de 100.000.000 para +/- 27 comparações.
  - Mas a lista deve estar ordenada. Vale a pena ordenar a lista?

Por que estudar Estruturas de Dados?

- Suponha que você precisa ordenar o cadastro de veículos do Detran pelo número do renavam de cada veículo. Existem 10.000.000 de veículos cadastrados.

- Você consegue pensar em como ordenar esse cadastro?
- Seu algoritmo é eficiente?

- Algoritmo 1: Até 100 trilhões de comparações
- Algoritmo 2: Aproximadamente 230 milhões de comparações

- Qual você prefere?
- Como estimar esse número?
- Você consegue fazer melhor?

Por que estudar Estruturas de Dados?

- Suponha agora que você foi contratado para ajudar no cálculo de rotas de uma empresa de transporte.

- A empresa quer saber qual é a melhor rota possível para o caminhão fazer 100 entregas. Você é capaz de encontrar essa rota?

- A empresa agora quer saber qual é o melhor caminho entre dois armazéns. Você pode encontrar esse caminho?

Por que estudar Estruturas de Dados?

- Esses sãos alguns exemplos de problemas que podemos resolver usando algoritmos e estruturas de dados.

- O estudo das estruturas de dados permite identificar a melhor ED para cada caso.

- O estudo da complexidade computacional permite estabelecer uma estimativa do “desempenho” de um algoritmo.

Como avaliar um algoritmo?

- **Ideia:** Vamos usar o tempo de execução

vs

- **Pergunta:** Essa é uma comparação justa?

Não!

Mas como podemos avaliar ou comparar um algoritmo sem depender da máquina?

Como avaliar um algoritmo?

Algoritmo 1:

Entrada: Vetor V com n elementos

```
1 para i ← 1 ... n faça
2     min ← i
3     para j ← i + 1 ... n faça
4         se V[j] < V[min] então
5             min ← j
6     aux ← V[i]
7     V[i] ← V[min]
8     V[min] ← aux
9 retorne V
```

Como avaliar um algoritmo?

Algoritmo 2:

Entrada: Vetor V com n elementos

Função `Ajustar`(V, ini, fim)

```
1  se ini < fim então
2      meio ← (ini + fim)/2
3      Ajustar(V, ini, meio)
4      Ajustar(V, meio + 1, fim)
5      Unir(V, ini, meio, fim)
6  retorne V
```

Função `Unir`$(V, ini, meio, fim)$

```
7  x ← fim − ini + 1
8  Criar vetor A com x posições
9  p1 ← ini        p2 ← fim
10 para i ← 1 ... x faça
11     se p1 ≤ meio ∧ p2 ≤ fim então
12         se V[p1] < V[p2] então A[i] ← V[p1]        p1 ← p1 + 1;
13         senão A[i] ← V[p2]        p2 ← p2 + 1;
14     senão
15         se p1] ≤ meio então A[i] ← V[p1]        p1 ← p1 + 1;
16         senão A[i] ← V[p2]        p2 ← p2 + 1;
17 para i ← 1 ... x faça  V[ini + i] ← A[i] ;
18 retorne V
```

Como avaliar um algoritmo

- O que os algoritmos fazem?
  - Ordenação de vetores
- Qual é mais eficiente em tempo de execução?
  - O segundo
- Por que?
  - Veremos durante o estudo da complexidade computacional...
- Curiosidade: O que acontece quando o computador tem a memória limitada praticamente ao tamanho do vetor?

Aviso Importante:

- Não haverá aula de AED nessa sexta-feira, dia 01/03.

- Retornamos no dia 05/03.

Revisão de Programação

Conceitos de Programação

- Vamos precisar muito dos seguintes conceitos de programação.

- É importante relembrar os seguintes tópicos:
  - Vetores
  - Estruturas (structs)
  - Funções e Procedimentos
  - Passagem de Parâmetros por valor e por referência
  - Conceitos de Ponteiros

Vetores

- Também chamado de array

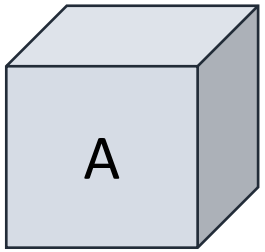
- Uma **variável compartimentada**, onde cada compartimento pode armazenar apenas um valor de um determinado tipo por vez.

- Variável → Apenas 1 valor por vez
- Vetor → Diversos compartimentos → Cada compartimento equivale a uma variável
- Cada compartimento → Um valor por vez

- **Todos os compartimentos devem ser de um mesmo tipo.**
- Vetor de Inteiro → Todos os compartimentos são inteiros

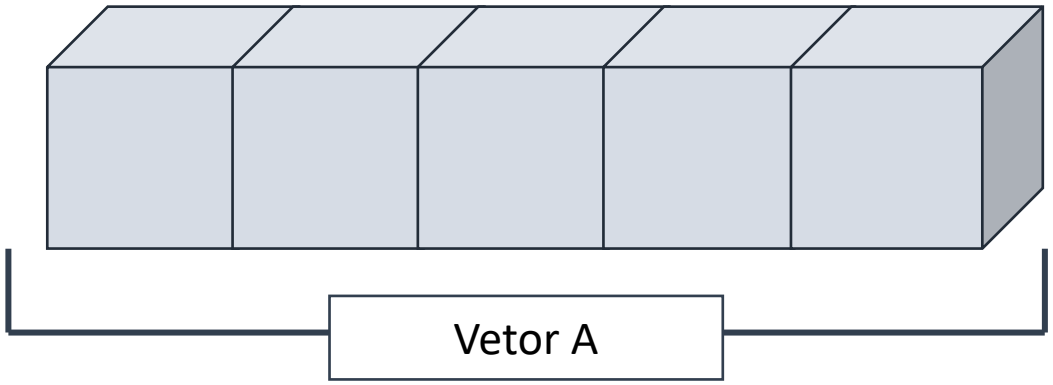
Variável vs Vetor

- Variável:



- Permite armazenar apenas 1 valor de um determinado tipo por vez
- Ao atribuir um novo valor, o anterior é perdido
- Uso: Identificado apenas pelo nome (Ex: a = 1;)

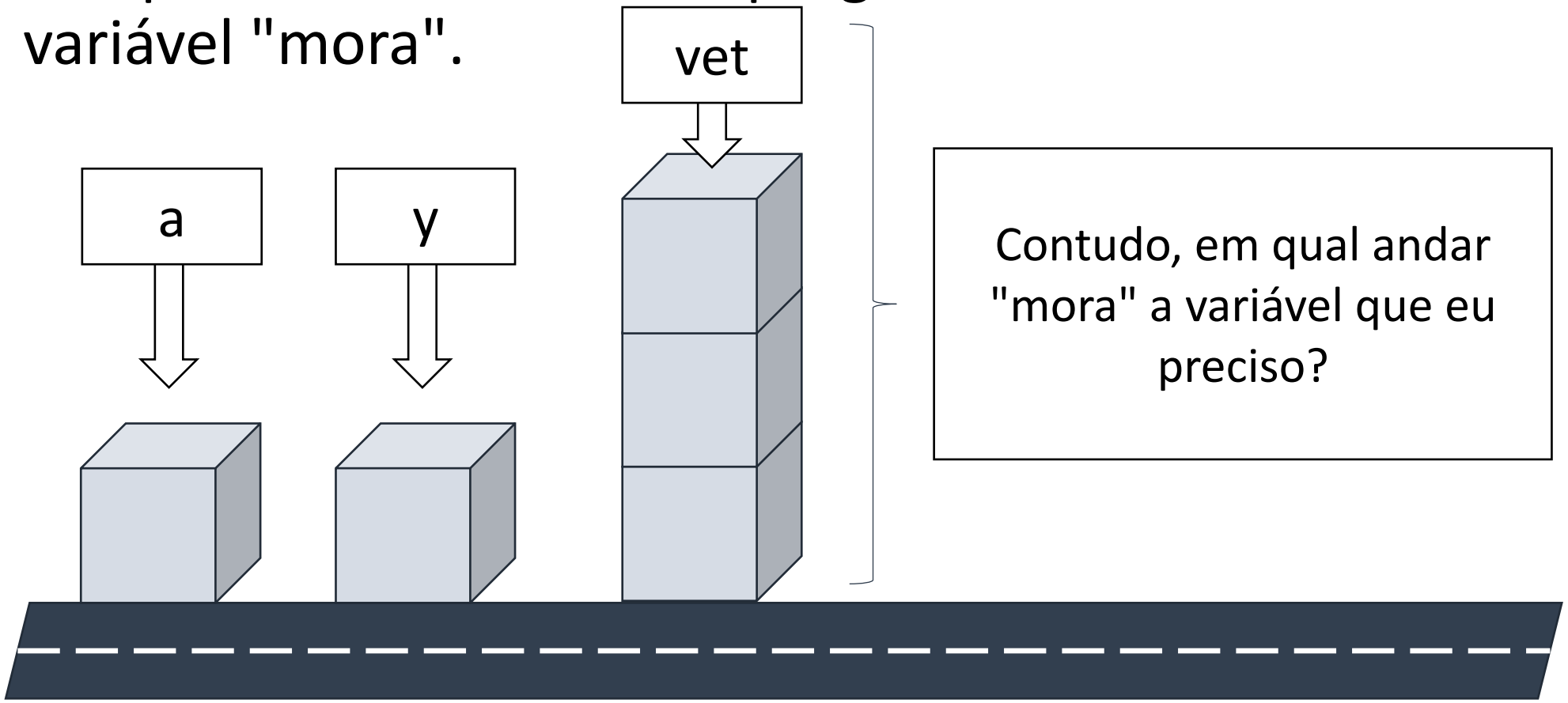
- Vetor:



- Constituído de diversos “compartimentos”
- Cada compartimento equivale a uma variável
- Desta forma, pode armazenar diversos valores
- Todos os compartimentos devem ser do mesmo tipo
- **Problema: Como especificar qual compartimento se deseja acessar?**

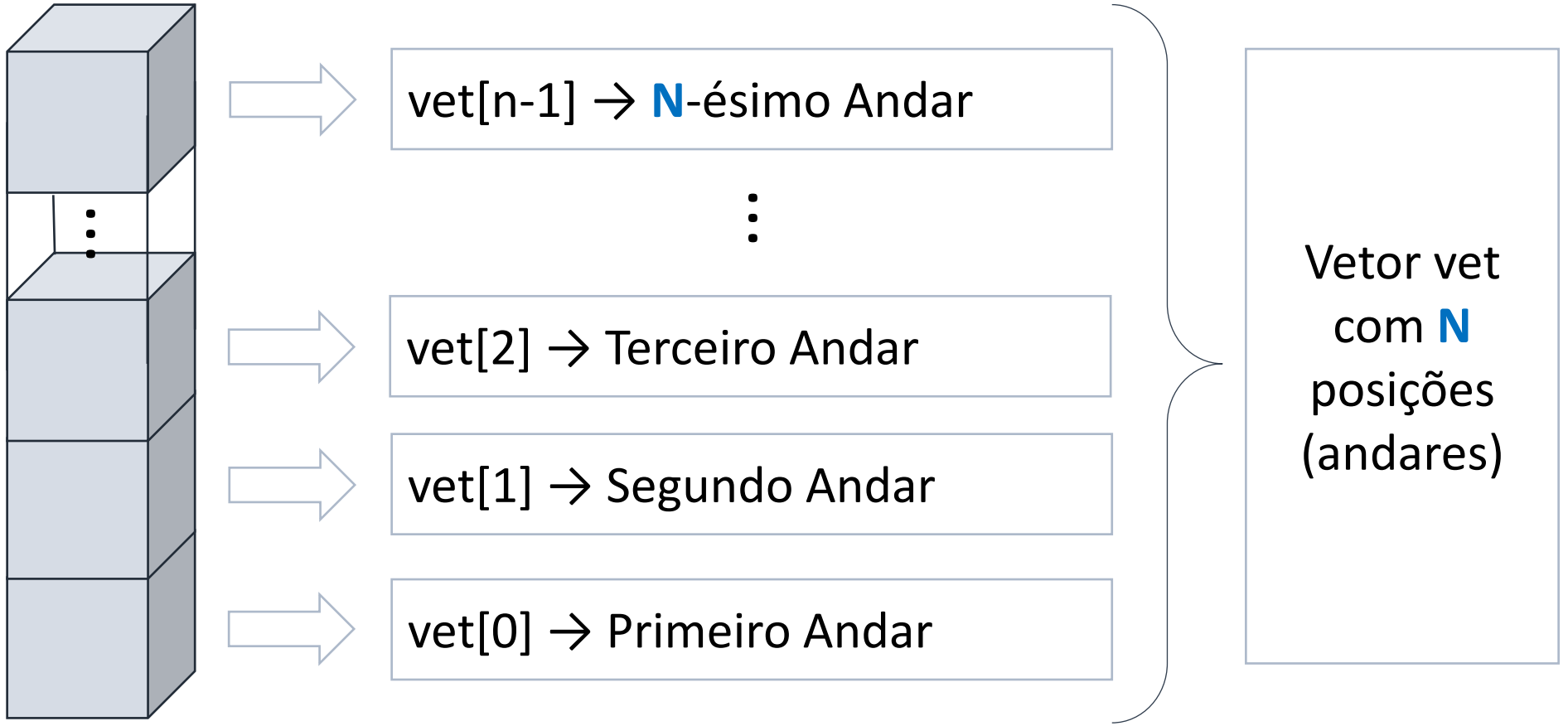
Memória

- Pode-se fazer a analogia entre a memória do computador e uma rua. O programa sabe onde cada variável "mora".



Como especificar o "andar"

- Para especificar o andar que se deseja acessar deve-se utilizar os sinais de colchete [ ]



Operador [ ]

- Serve para indicar qual posição do vetor se deseja acessar
- A primeira posição de um vetor é a posição 0. Por que?

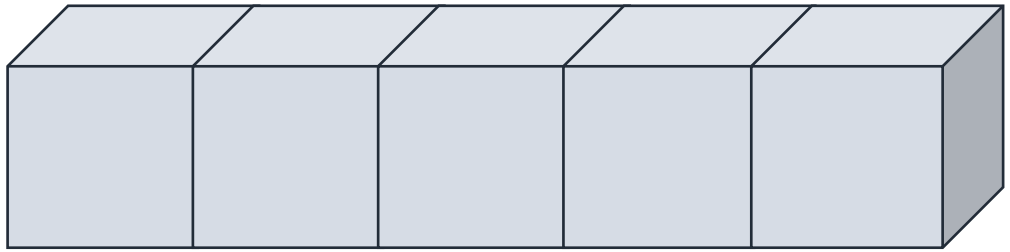
- O operador [ ] determina um **deslocamento**:
  - float vetNotas[10];
  - vetNotas[0] → Se desloque 0 posições. Equivale ao início
  - vetNotas[9] → Se desloque 9 posições. Equivale ao final
  - vetNotas[10] → Se desloque 10 posições. Extrapola o final do vetor.

  - vetNotas[**x**] → Dado o início do vetor vetNotas, se desloque **x** posições

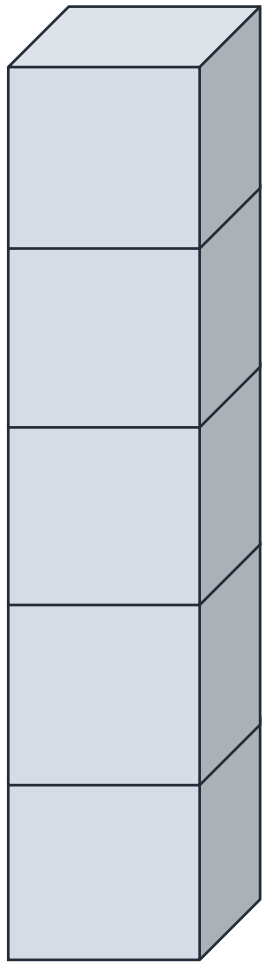
Notação gráfica

- Os vetores podem ser desenhados graficamente para exemplificar seu comportamento e função.
- Este desenho pode ser no sentido horizontal ou vertical.
- A forma de desenhar não interfere em nada no comportamento dos vetores

Horizontal:



Vertical:



Como declarar um vetor

- A definição de um vetor deve ser feita utilizando as mesmas regras aplicadas à definição de variáveis.

- Sua sintaxe é:
  - <tipo> <nome>**[<tamanho>]**;

Quantos compartimentos?

- Exemplos:
  - int vet[10];
  - float fx[100];
  - char str[20];

Como acessar um vetor

- Para acessar um vetor é necessário especificar qual posição se deseja acessar. Como foi visto, para tal são utilizados os colchetes.

- **Não é possível acessar um vetor inteiro.**
  - **notas = 0 → qual nota? qual compartimento?**

- Exemplo:

```
float notas[10];
notas[0] = 0;
notas[1] = 1;
....
notas[9] = 9;
notas[10] = 10;
```

ERRO: O vetor não tem posição 10!!! Se o prédio tem 10 andares, só podemos subir 9 lances de escada.

Tipo de Dados

- Até agora, usamos apenas tipos já existentes na linguagem
  - int, float, char, double, long ...
- Mas existem casos em que um único tipo é insuficiente para representar a informação
- Por exemplo:
  - O registro de um carro: Ele é formado por placa, renavam, chassi, cor, modelo, fabricante...
- Como armazenar todas essas informações? Várias variáveis?

Definição de Novos Tipos de Dados

- As linguagens de programação estruturadas permitem a definição de novos tipos de dados

- Esses novos tipos são **compostos** por vários campos de tipos de dados previamente definidos (int, float, ...)

- Um novo tipo também pode ser formado por outros tipos já definidos (Um tipo de dados para armazenar os dados de uma pessoa, pode conter um tipo que armazena a data de nascimento)

Exemplos

- Uma data:
  - Três campos do tipo inteiro (dia, mês e ano)

- As **coordenadas** de um ponto no plano:
  - Formada por dois campos do tipo real (X e Y)

- Os dados de um círculo:
  - As **coordenadas** do centro
  - Seu raio

Novos tipos de dados

- A definição de novos tipos de dados apresenta alguns nomes diferentes na literatura
  - Estruturas (de dados)
  - Tipos
  - Registros
  - **Structs**
  - Tipo Abstrato de Dados (TAD)

Definição de novos tipos em C

- Em C, os tipos são chamados de **structs**

- Sintaxe:

```
struct <nome do novo tipo> {
        <tipo já existente> <nome do campo>;
        <tipo já existente> <nome do campo>;
        ....
} <opcional: nome da variáveis>;
```

Exemplo

- Tipo para armazenar coordenadas no plano

```
struct coordenada {
        float X;
        float Y;
};
```

Utilização dos Tipos de Dados

- Para definir variáveis de um tipo:
  - Opção 1: Criar as variáveis durante a definição do tipo:

```
struct coordenada {
        float X;
        float Y;
} A, B;
```

As variáveis A e B já são definidas no momento da construção do tipo **coordenada**.

Utilização dos Tipos de Dados

- Para definir variáveis de um tipo:
  - Opção 2: Criar as variáveis após a definição do tipo (em qualquer momento):

```
struct coordenada {
        float X;
        float Y;
};
.....
struct coordenada A, B;
```

Atenção!
Essa opção exige que se comece a definição com a palavra **struct**:
Defina duas variáveis A e B do tipo **estrutura coordenada**

Acesso aos dados dentro do tipo

```
struct coordenada {
        float X;
        float Y;
} A, B;
```

- Como acessar o valor de X **dentro** de cada coordenada?
  - Operador . (ponto): <variável do tipo>.<campo>
  - Ex:

```
A.X = 10.0;
if (A.X == B.X && A.Y == B.Y)
      printf("Os pontos A e B são iguais\n");
```

Exemplo: Definição dos Structs

```
struct coordenada {
        float X;
        float Y;
};
struct circulo {
        struct coordenada centro;
        float raio;
};
```

O centro do círculo também é uma estrutura (struct coordenada)

```
struct circulo C;
printf("Digite os dados do circulo\n");
scanf("%f %f %f", &C.centro.X, &C.centro.Y, &C.raio);

struct coordenadas P;
printf("Digite as coordenadas do Ponto\n");
scanf("%f %f", &P.X, &P.Y);
```

Comando typedef

- Permite criar um “apelido” para os tipos definidos

Sem typedef:

```
struct pessoa {
   int idade;
   float peso;
   float altura;
};
struct pessoa Joao;
struct pessoa Maria;
```

Com typedef:

```
typedef struct {
   int idade;
   float peso;
   float altura;
} pessoa;
pessoa Joao;
pessoa Maria;
```

Comando typedef

- Permite criar um “apelido” para os tipos definidos

Com typedef:

```
typedef struct {
   int idade;
   float peso;
   float altura;
} pessoa;
pessoa Joao;
pessoa Maria;
```

- Observe que o tipo "`struct pessoa`" foi batizado com o nome "`pessoa`";
- Isso possibilita se referir ao nome escolhido como um novo tipo (tal como `int`, `float`, ...)
- **Importante:** Observe que "`pessoa`" não é uma variável. "`pessoa`" é **um novo tipo de dados**.

Exemplo: Definição dos Structs

```
typedef struct {
        float X;
        float Y;
} coordenada;
typedef struct {
        coordenada centro;
        float raio;
} circulo;
```

Com o typedef coordenadas passa a ser reconhecido como um outro tipo de dados (assim como int, float, char...)

E agora podemos usar como qualquer outro tipo de dados

Inclusive como um campo dentro de outra estrutura

```
circulo C;
printf("Digite os dados do circulo\n");
scanf("%f %f %f", &C.centro.X, &C.centro.Y, &C.raio);

coordenada P;
printf("Digite as coordenadas do Ponto\n");
scanf("%f %f", &P.X, P.Y);
```

Ponteiros para estruturas (struct)

```
typedef struct {
        long long int CPF;
        int idade;
        char nome[200];
} pessoa;
```

```
Pessoa p;
p.CPF = 12345678909
p.idade = 99;
p.nome = "Fulano da Silva Sauro"
```

```
pessoa *ptP = &p;
```

```
//Para acessar os campos de p:
*ptP.CPF: Quem é o ponteiro?
                    ptP ou CPF???
```

Para evitar essa confusão, o uso de ponteiros para estruturas tem um operador diferente:

Operador ->

Ponteiros para estruturas (struct)

```
typedef struct {
        long long int CPF;
        int idade;
        char nome[200];
} pessoa;

Pessoa p;
p.CPF = 12345678909
p.idade = 99;
p.nome = "Fulano da Silva Sauro"
```

```
pessoa *ptP = &p

//Para acessar os campos de ptP:
ptP->idade = ptP->idade + 1;

//Não é necessário usar o * antes de ptP. O uso de -> já indica que é um ponteiro.
```

<ponteiro> -> <campo do struct>

Dúvidas sobre structs?

Funções e Procedimentos

- Função/Procedimento:
  - Trecho de um programa
  - Independente do programa – Pode ser usado em vários programas (printf, scanf)
  - Objetivo determinado: executa somente uma única tarefa
  - Simplifica a escrita e o entendimento do programa
  - Pode ser executada várias vezes, sem a necessidade de ser escrita várias vezes

- Exemplo:
  - Função sqrt → Raiz quadrada
  - Função pow → Potência

- Existem funções já predefinidas, contidas nas bibliotecas incluídas no código (include).
- Além destas, podemos definir diversas outras.

Retorno das Funções e Procedimentos

- Alguns trechos de código ao serem executados resultam um valor que deve ser retornado.
- Os trechos que retornam valores (o resultado da função) são chamados de Funções. Por exemplo: `pow(3,4)` retorna 3^4
- Já os trechos que não tem um valor que deve ser retornado são chamados de Procedimentos. Por exemplo: o printf não "devolve" nenhuma informação.
- **Função tem retorno. Procedimento, não.**
- O retorno de uma função deve atender à um tipo de dados.

Função

```
<tipo do retorno> <nome> ()
{
    //efetuar as operações da função
    return <resultado>; //como na main
}
```

- Calcular a raiz quadrada, calcular o número de horas entre datas, etc...

Procedimento

```
void <nome> ()
{
    //efetuar as operações da função
}
```

- **void** – indica que não haverá retorno (o retorno é vazio).
- Exemplo: Imprimir um vetor na tela, inicializar uma matriz com 0, aplicar um reajuste nos salários armazenados em um vetor

Parâmetros

- Funções e procedimentos podem receber dados necessários para o seu processamento.
- Esse dados são conhecidos como **parâmetros**.

- **Parâmetros são as entradas das funções e procedimentos.**

- Essas entradas devem ser definidas:
  - Em relação a quantidade de entradas necessárias
  - Quanto ao seu tipo
  - Quanto ao nome que será utilizado para estas entradas

Exemplo de Função

- Vamos pensar em uma função matemática

$$\mathrm{pow}(x, y) = x^y$$

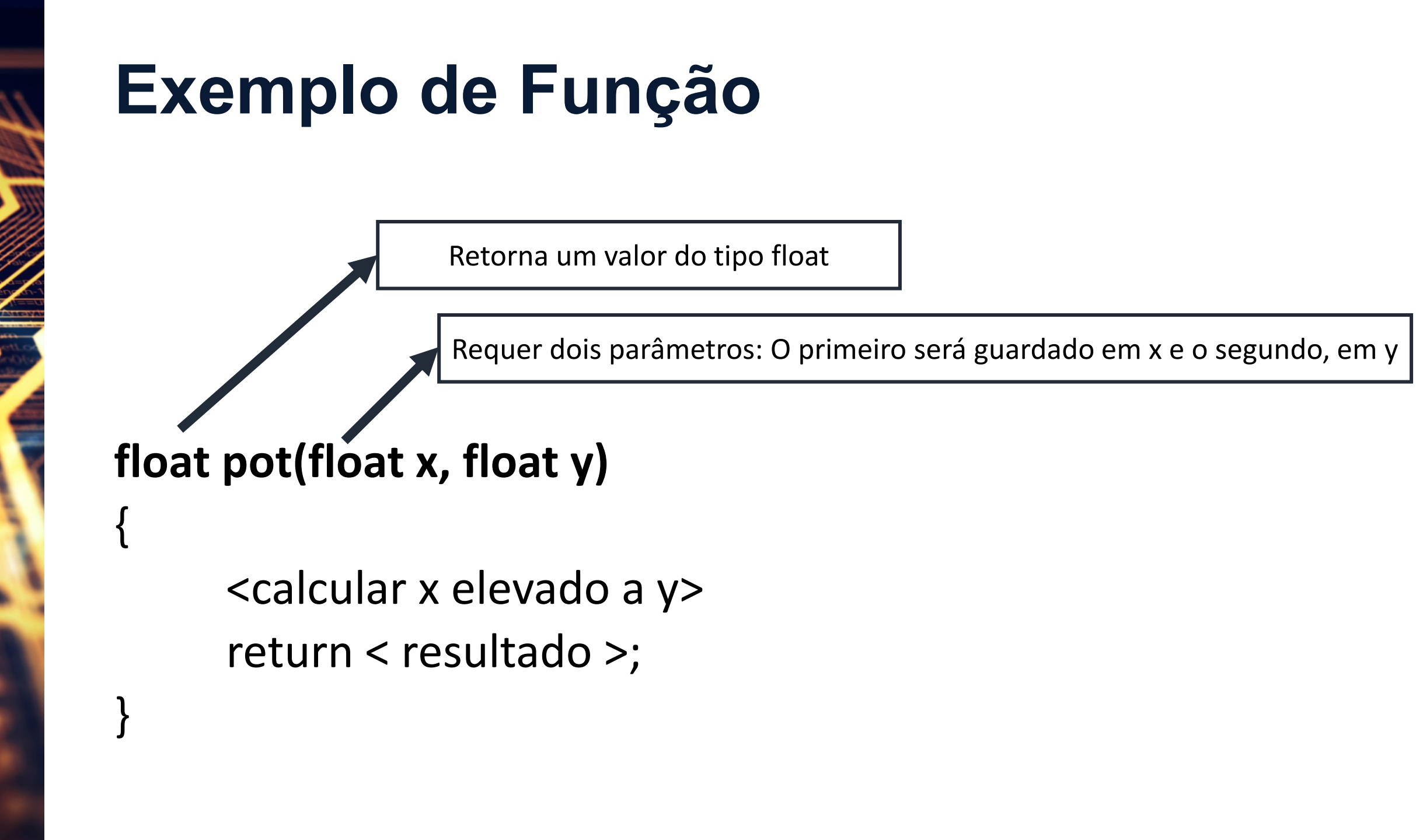
- Entradas: Números real X e Y
- Saída: Número real

- Ao calcular o resultado desta função é necessário indicar em sua chamada os valores das entradas e na ordem correta:

  $\mathrm{pow}(3,2)$ Entradas: Inteiro 3(x), Inteiro 2(y); Saída: Inteiro 9

  $\mathrm{pow}(2,3)$ Entradas: Inteiro 2(x), Inteiro 3(y); Saída: Inteiro 8

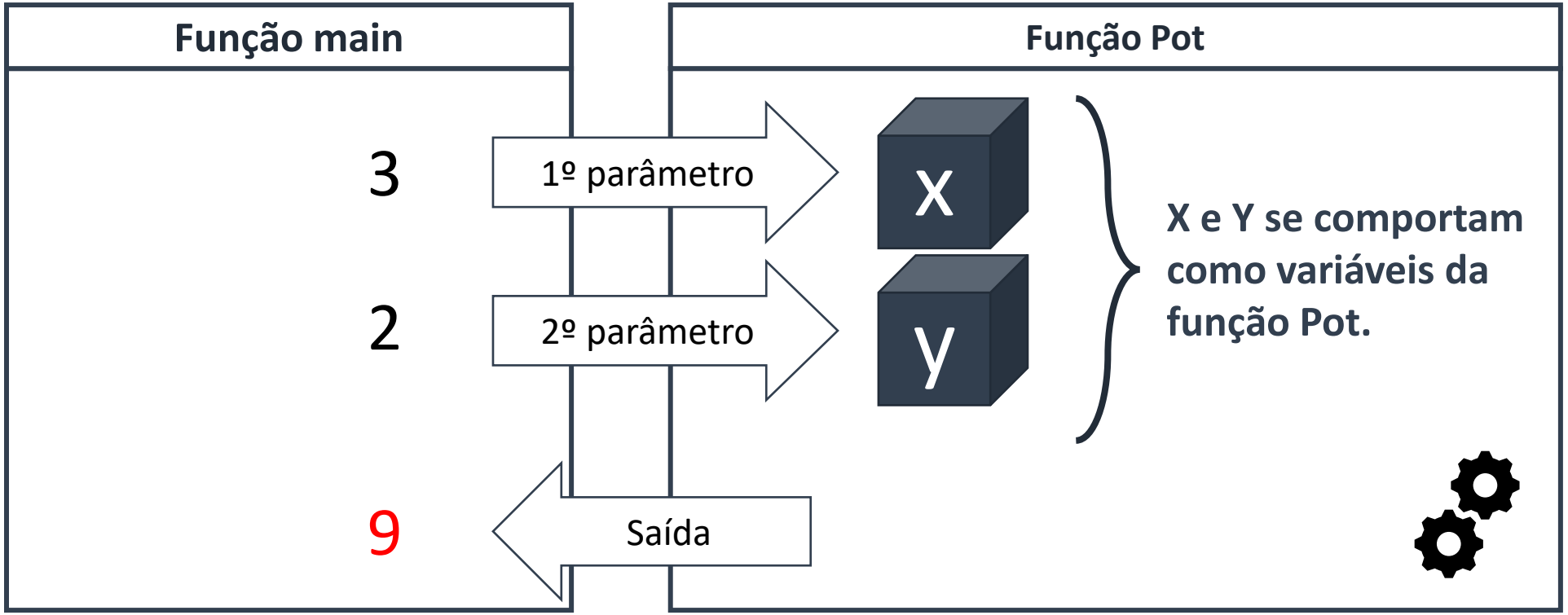
Exemplo de Função




```
float pot(float x, float y)
{
        <calcular x elevado a y>
        return < resultado >;
}
```

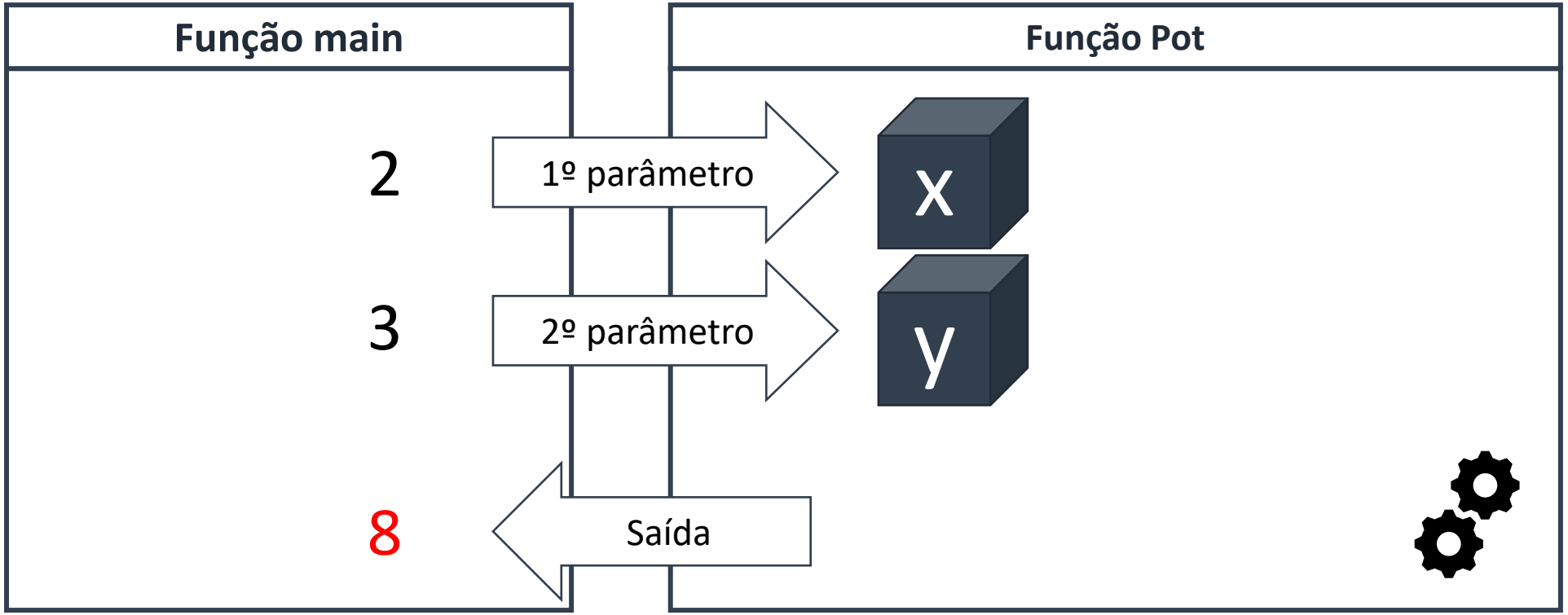
Exemplo de função

- Quando a função é executada:



Exemplo de função

- Quando a função é executada



Parâmetros: A ordem importa!

- O primeiro valor informado será atribuído ao primeiro parâmetro
- O segundo valor informado será atribuído ao segundo parâmetro

.

.

.

.

O último valor informado será atribuído ao último parâmetro

Parâmetros

- Os parâmetros se comportam exatamente como uma variável.
- Só que essa variável já é previamente inicializada com o valor que foi passado para a função.
- Vale lembrar que essas variáveis (os parâmetros) só existem dentro da função ou do procedimento.
- Sendo assim... O que acontece se essas variáveis forem alteradas?
- Resposta: **Depende...**
- Existem duas formas de passar parâmetros:
  - Passagem por Valor
  - Passagem por Referência

Funções - Return

- Uma função retorna um **único** valor.

- Não existe função com 2 returns...

```
int teste()
{
   return 1; //Será executado
   //A função retorna o valor 1 e termina sua execução
   return 0; //Não será executado – a função já terminou
   printf("Teste"); //Não será executado
}
```

Onde definir suas funções: Opção 1

```
#include <stdio.h>
<aqui entram as constantes - #define >
<as estruturas devem ser definidas aqui>

<aqui entram as funções e procedimentos>

int main()
{
      <código do programa>
      return 0;
}
```

Uma função ou procedimento deve ser declarado antes de ser usado.

Desta forma, se uma função A utilizar uma outra B, B deve ser declarada antes de A.

Por isso, o programa começa com os `includes`.

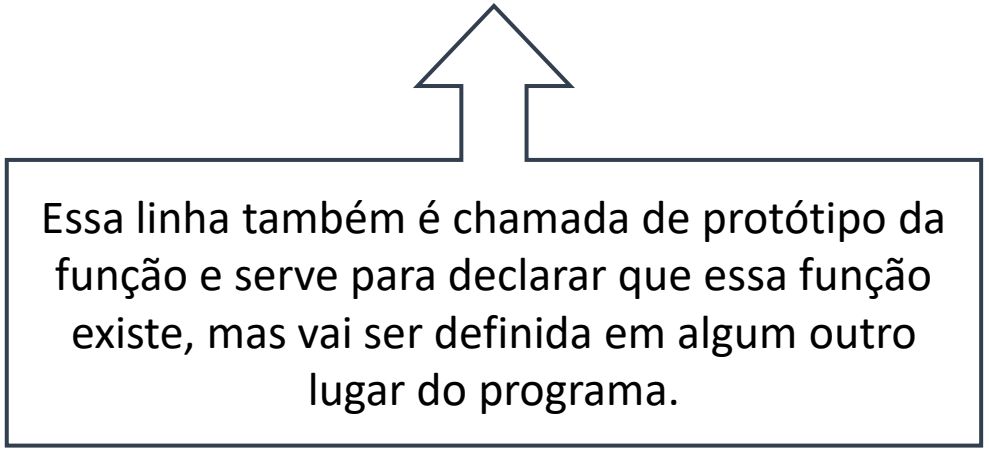
Onde definir suas funções: Opção 2

```
#include <stdio.h>
<aqui entram as constantes - #define >
<as estruturas devem ser definidas aqui>

<apresentar somente os cabeçalho das funções e procedimentos: >
<tipo de retorno> <nome da função>(<parâmetros>);

int main()
{
      <código do programa>
      return 0;
}

<as funções ou os procedimentos são definidos e escritos aqui>
```



Passagem de Parâmetros

- Existem duas formas de efetuar a passagem de parâmetros:
  - **Por Valor:** Uma cópia do valor da variável é enviado para a função
  - **Por Referência:** O endereço da variável é enviado para a função.

- Ao enviar apenas o valor da variável, o que acontece no interior da função não será repercutido à variável passada como parâmetro (ela recebeu somente uma cópia do valor)

- Contudo ao enviar o endereço, a função que recebeu o endereço da variável pode alterar seu conteúdo.

Passagem por Referência

- Para que uma função possa alterar o valor da variável passada como parâmetro é necessário efetuar algumas modificações na declaração da função e no momento de sua chamada.
- Esta modificação fará com que a função passe a receber o **endereço da variável**.
- Cada variável possui um endereço único, que representa onde, na memória do computador, ela é armazenada.
- Para implementar esta modificação são necessários dois novos operadores: & e *
- **A função scanf é um exemplo de passagem por referência.**

Operadores & e *

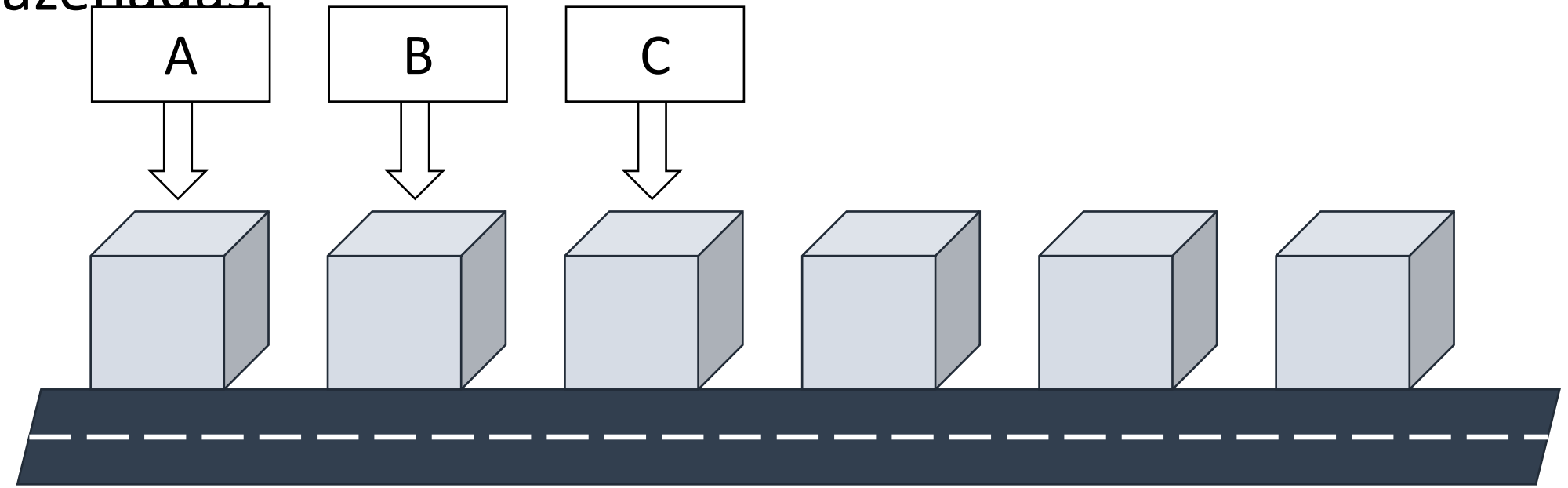
- Esses operadores permitem o acesso ao **endereço** de uma variável.
- Com o endereço de uma variável é possível fazer alterações sem nem conhecer seu nome.

- **O operador &, quando aplicado a uma variável, retorna o endereço onde ela está armazenada.**

- **O operador *, quando aplicado a um endereço, permite acessar o conteúdo lá armazenado.**

Memória

- Pode-se fazer a analogia entre a memória do computador e uma rua. O programa sabe onde cada variável “mora”.
- Supondo que em um determinado escopo 3 variáveis (A, B e C) sejam declaradas. Somente neste escopo o computador associa os nomes aos endereços onde estas estão armazenadas.



Memória

- A associação entre o nome e o endereço da variável só existe no escopo onde a variável foi definida.

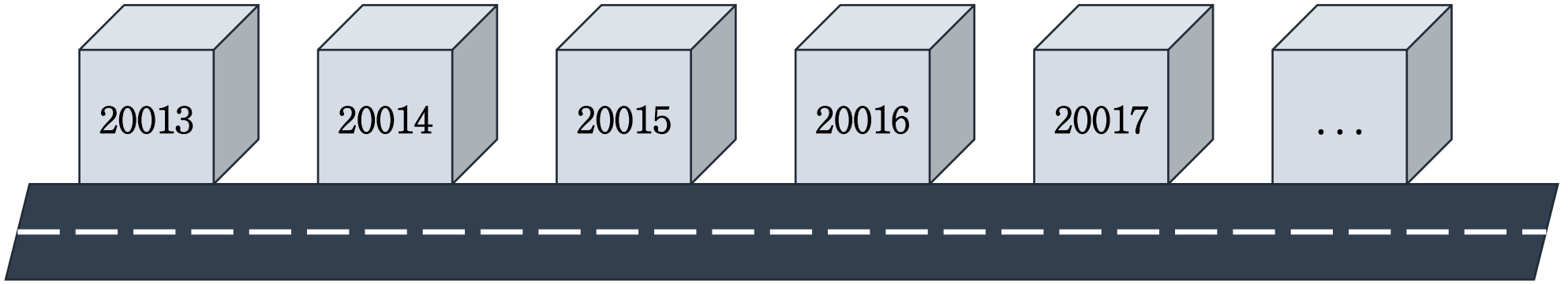
- Não confundir:
  - A variável só existe durante a execução o escopo da função!
  - O endereço pode ser acessado por outras funções chamadas **durante a execução da função** (ou procedimento) onde a variável existe.
  - Quando a função onde a variável foi definida termina, a variável é desalocada (deixa de existir)

Memória

Considerando agora outro escopo...

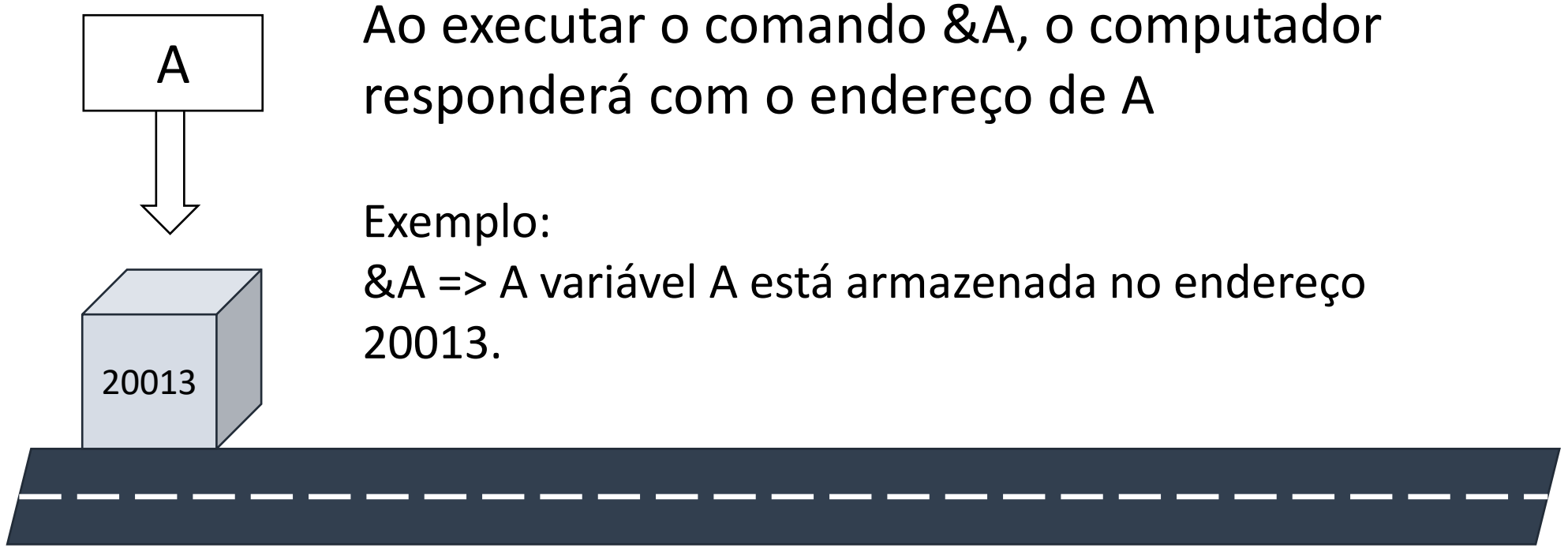
Onde está a variável A?
Não sabemos... Não estamos mais no escopo onde ela foi declarada...

Mas e se conhecermos diretamente o endereço da variável A (isto é, onde ela é guardada)?



Memória

- Como visto anteriormente, o operador & obtém o endereço da variável. Com este endereço, podemos acessar diretamente o local onde a variável é armazenada.



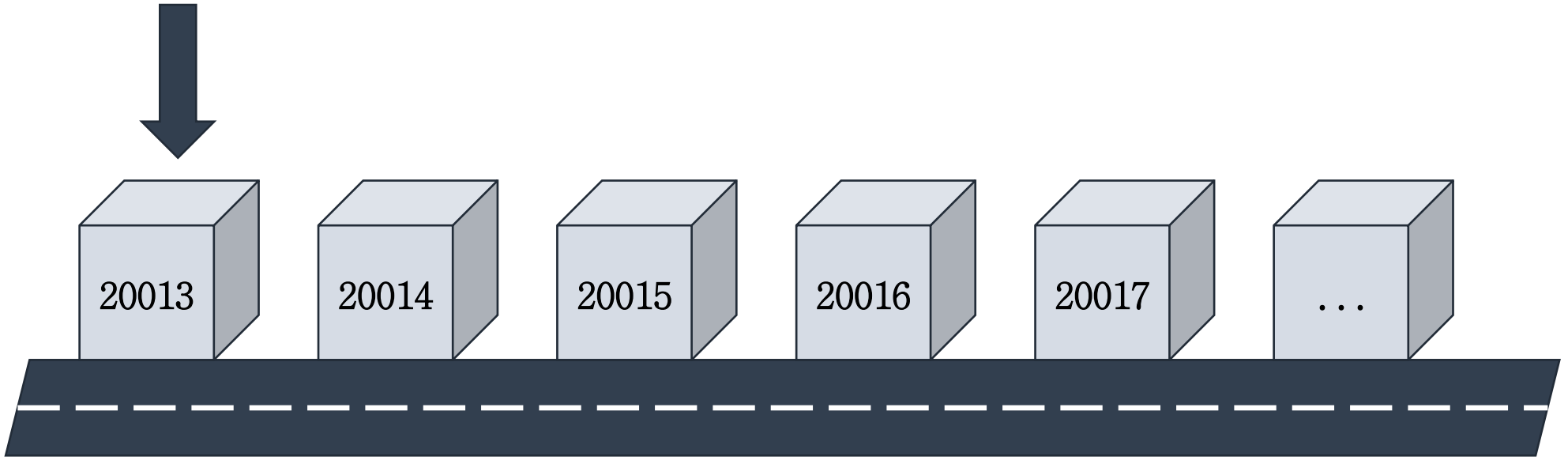
Ao executar o comando &A, o computador responderá com o endereço de A

Exemplo:
&A => A variável A está armazenada no endereço 20013.

Memória

- Onde está a variável A?
  Anteriormente, descobrimos (usando o operador &) que a variável A está armazenada no endereço 20013.

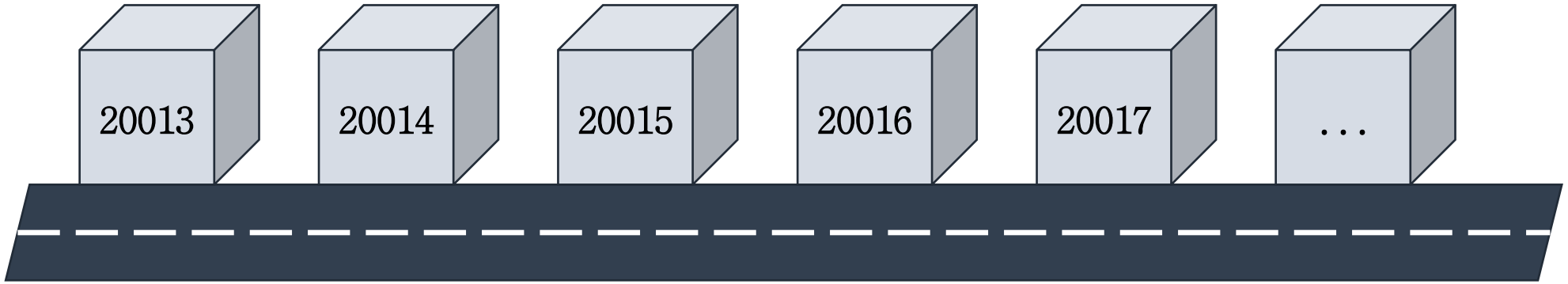
- Como acessar este endereço?



Memória

Ao executar o comando *(20013), o computador acessará o conteúdo da “casa” de número 20013.

Este operador permite **acessar (ler e escrever)** no endereço especificado.



Passagem por Referência

- Como foi dito, a passagem de parâmetros por referência implica no acesso não mais ao valor da variável, mas no **acesso direto ao seu endereço**. Para isto algumas modificações são necessárias.

- Para implementar a passagem de parâmetros por referência é necessário modificar a declaração da função (ou procedimento) para indicar que o acesso será feito diretamente ao endereço.

- Além disso, toda a referência ao parâmetro também deverá indicar que se trata de um endereço.

- O operador que permite acessar um endereço é o *.

Ponteiros

- Existe um tipo especial de dados chamado ponteiro

- **Um ponteiro aponta uma área da memória de um tipo**

- int a; //definindo uma variável inteira chamada a
- int *ptA; //definindo um ponteiro para variáveis inteiras chamado ptA

- O primeiro caso guarda valores inteiros
- O segundo armazena o endereço de uma variável inteira na memória (onde a variável é armazenada)

Passagem por Referência

```
#include <stdio.h>
void multiplicar (int *x, int y)
{
        *x = *x * y;
}
```

Na definição:

*x → O parâmetro x é um ponteiro para inteiros
(ele guarda o endereço de uma variável)

No uso:

O operador * indica o acesso ao endereço para ler ou escrever

A instrução *x = *x * y deve ser vista como os seguintes passos:

1) Acessar o endereço que x aponta e obter o valor armazenado (o inteiro)
2) Multiplicar por y (y é uma variável – veja a definição da função)
3) Guardar o resultado no endereço apontado por x

Passagem por Referência

- Além destas mudanças, também é necessário modificar a chamada da função para enviar não mais o valor da variável, mas seu endereço.
- O operador que permite obter o endereço de uma variável é o &.

```
int main()
{
    int a = 2;
    int b = 3;
    multiplicar(&a,b);
    printf("%d %d",a,b);
}
```

&a → Obtém o endereço da variável a e o envia para o procedimento.

Passagem por Referência

- Observe que a passagem de parâmetros por valor requer a realização de uma cópia do valor da variável.
- Essa cópia é enviada para a função/procedimento.
- Entretanto, a cópia de vetores, matrizes, strings e structs não é tão trivial quanto a cópia de uma variável simples.
- **Dessa forma, vetores, strings, matrizes e structs são sempre passados por referência.**

Passagem de Vetores e Matrizes

- As dimensões são passadas como parâmetros e usadas na definição dos vetores. Primeiro os tamanho, só depois o vetor ou a matriz! A definição precisa conhecer as variáveis que são referenciadas.

- <tipo retorno> <nome> (int tam, <tipo> <nome>[tam]);
- <tipo retorno> <nome> (int l, int c, <tipo> <nome>[l][c]);

- int somarVetor(int t, int vet[t]);
- void ImprimirMatriz(int linhas, int colunas, int mat[linhas][colunas]);

- Na hora de chamar a função ou procedimento: somarVetor(10,vet);

- Essa forma permite que a função/procedimento lide com qualquer tamanho de vetor ou matriz. Observe que é necessário escrever as instruções em função do tamanho passado como parâmetro.

Ponteiros

- Ponteiros são estruturas usadas para apontar/manipular diretamente a memória do computador

- Se sabemos onde uma variável está armazenada (seu endereço – usando o operador &).

- Podemos acessar diretamente esse endereço da memória e manipular a informação que está armazenada diretamente (operador *).

Ponteiros

- Observe que o acesso não depende mais de conhecer o nome da variável. Basta possuir seu endereço.

- Isso permite a alocação dinâmica de variáveis. Voltaremos a esse tema no futuro.

Ponteiros

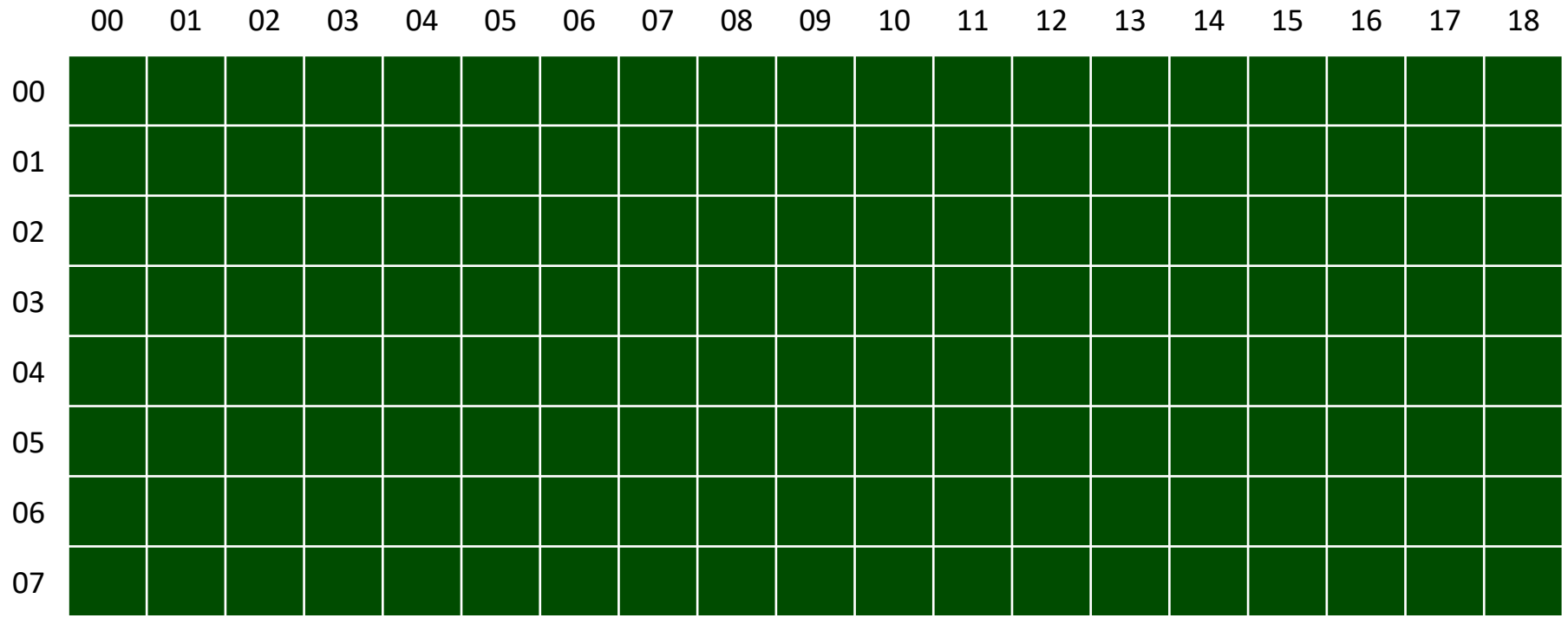
- **Um ponteiro aponta uma área da memória de um tipo**

- int *ptA; // Um ponteiro que aponta para variáveis do tipo int
- double *ptB; //Um ponteiro que aponta para variáveis do tipo double

- Importante: ptA e ptB não estão apontando para nenhuma variável no momento. Para que um ponteiro **aponte** para uma área de memória, precisamos fazer isso explicitamente:
- `int a; //Definindo uma variável: a é guardada na memória`
- `int *ptA = &a; //Agora sim ptA aponta para onde a é armazenada`

Endereçamento - Exemplo

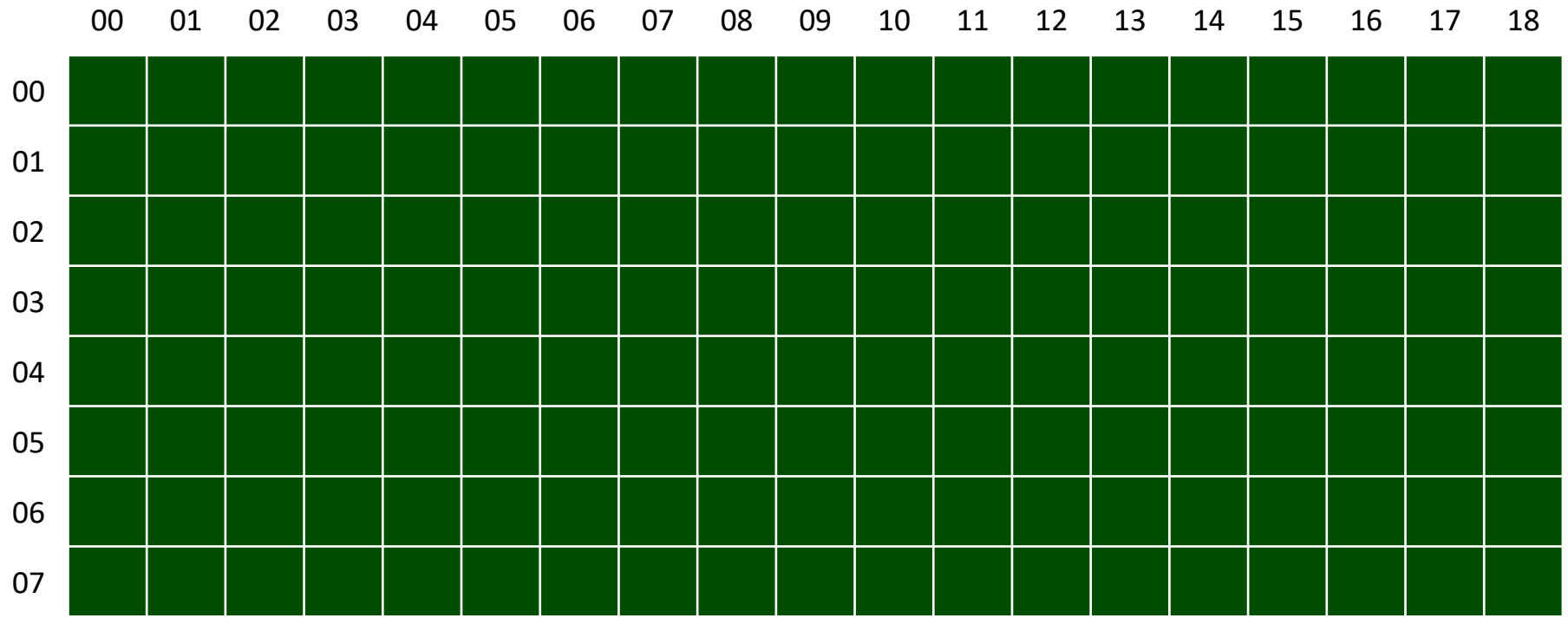
Suponha que podemos visualizar um pedaço da memória principal.



Endereçamento - Exemplo

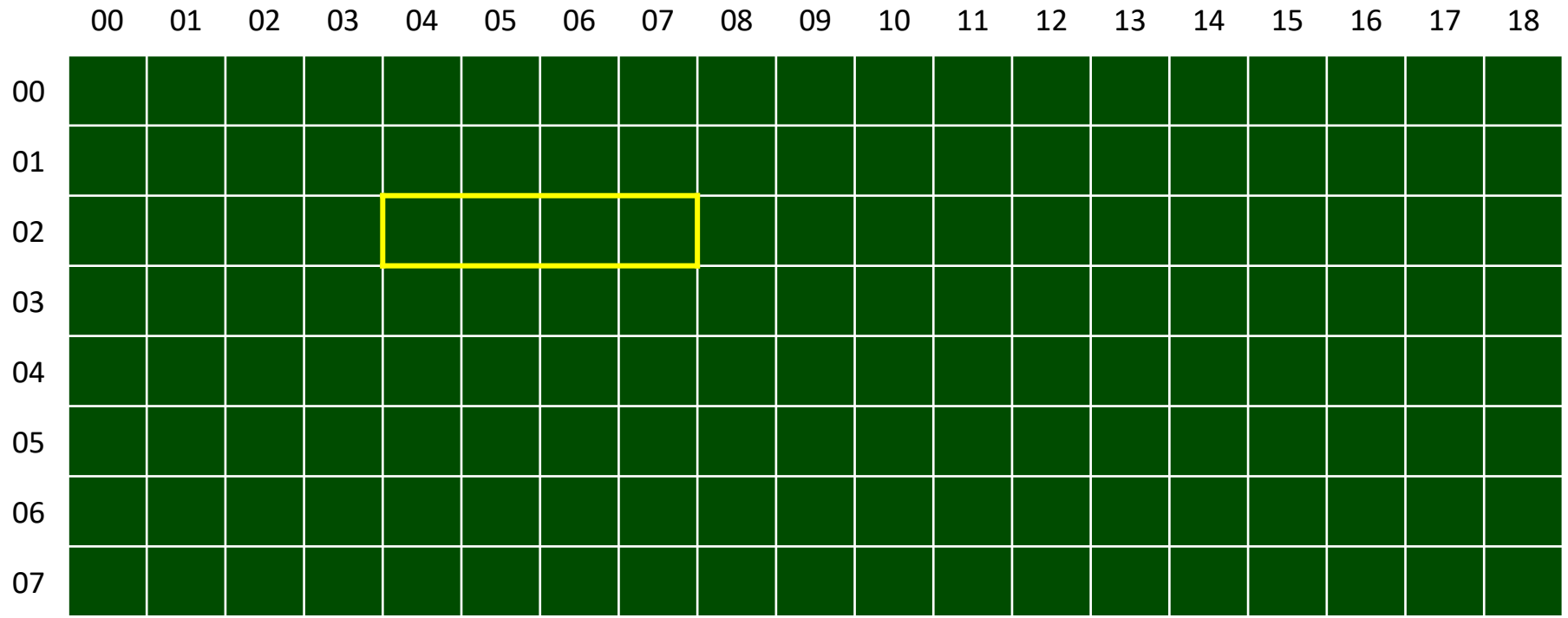
A instrução <int a;> aloca um “pedaço” da memória para guardar um inteiro

Inteiro: 4 Bytes



Endereçamento - Exemplo

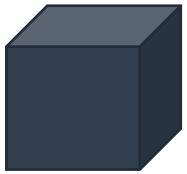
O computador verifica na memória onde ele pode armazenar os 4 bytes como um único bloco e aloca esse espaço: Endereços 1104, 1105, 1106 e 1107.



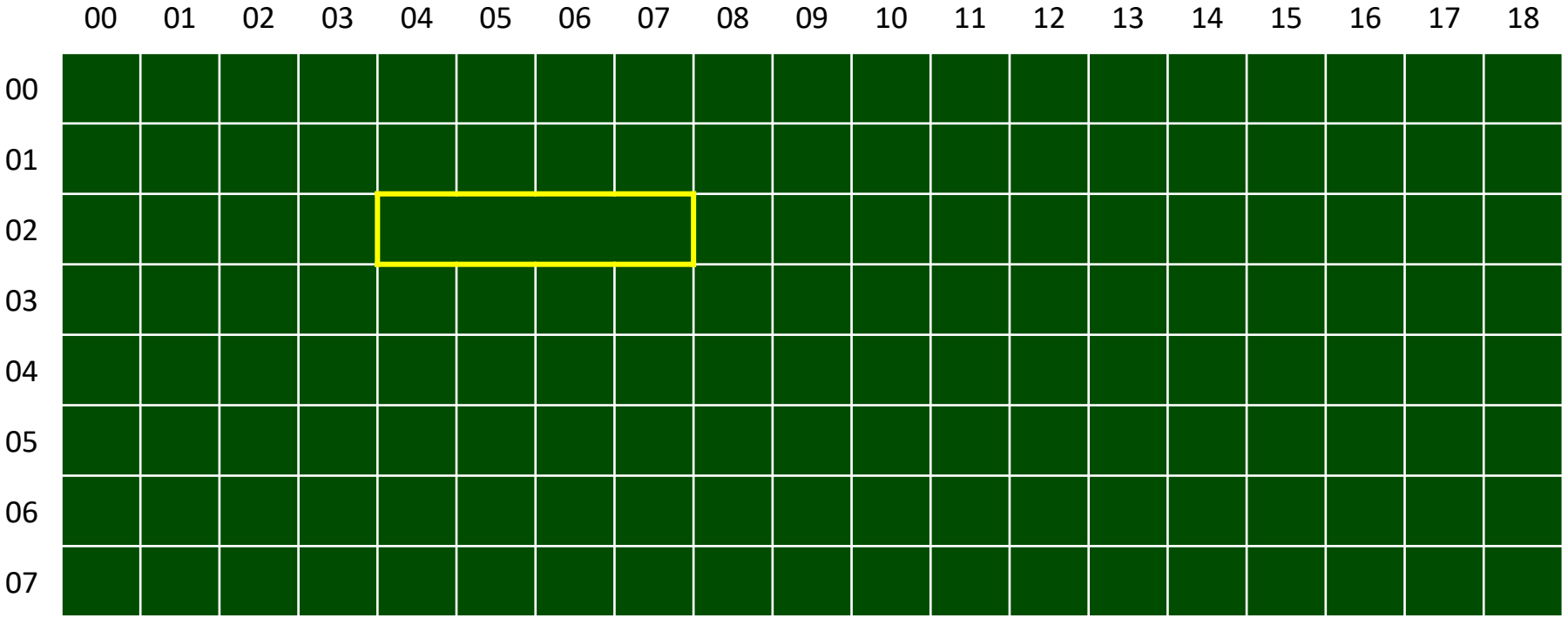
Endereçamento - Exemplo

A função que definiu o inteiro a agora sabe que os endereços 0204, 0205, 0206 e 0207 foram alocados a variável a.

a = 5; //Guardar 5 no bloco com 4 bytes iniciado em 0204



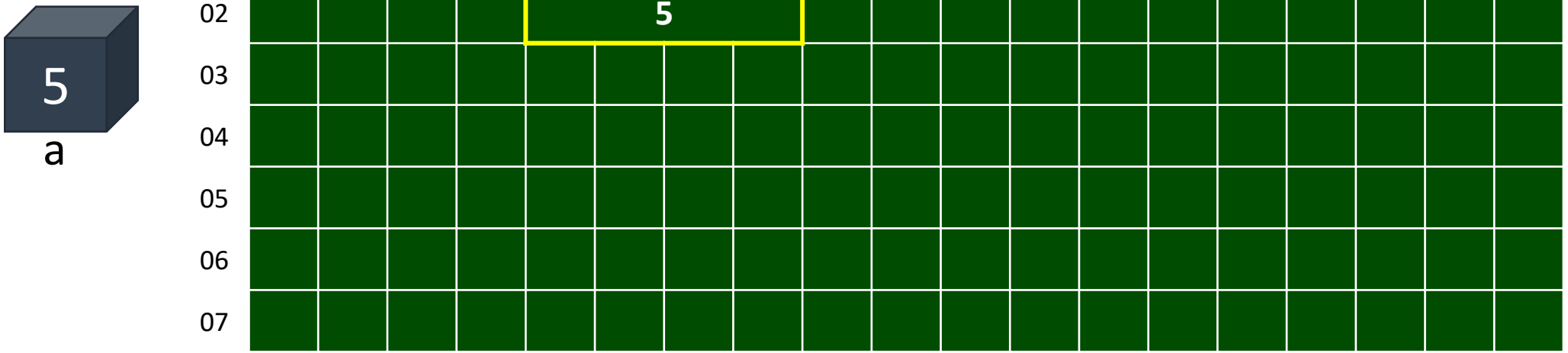
a



Endereçamento - Exemplo

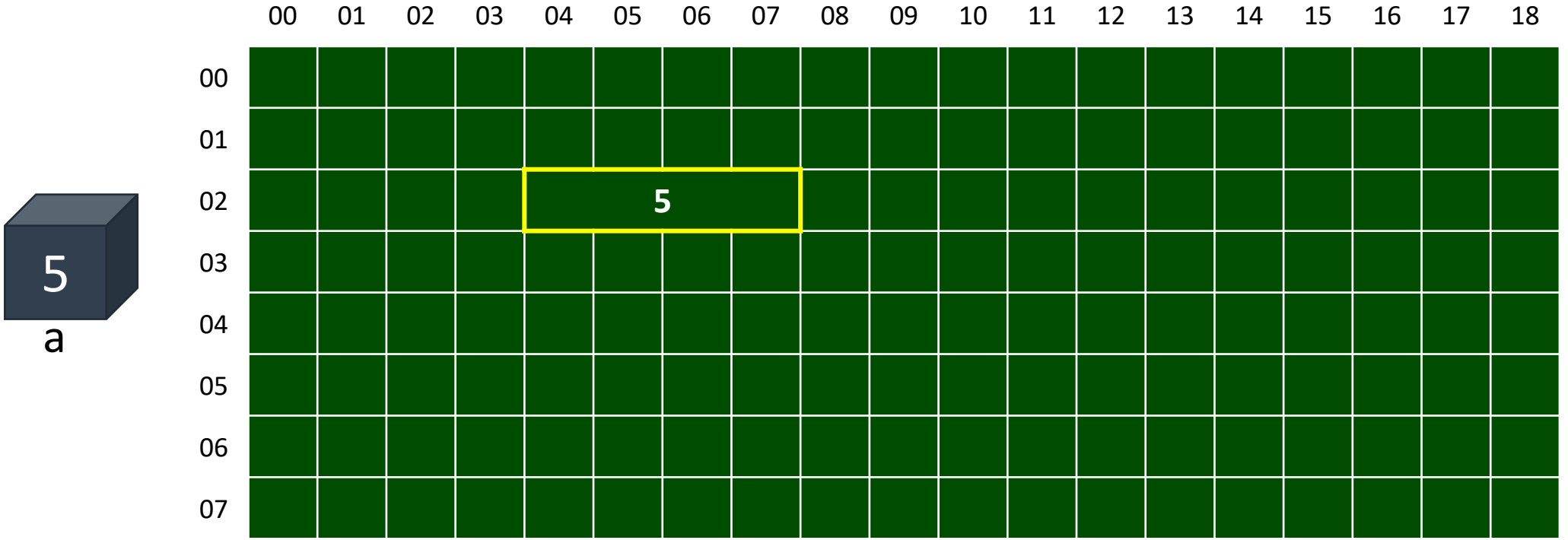
A função que definiu o inteiro a agora sabe que os endereços 0204, 0205, 0206 e 0207 foram alocados a variável a.

a = 5; //Guardar 5 no bloco com 4 bytes iniciado em 0204



Endereçamento - Exemplo

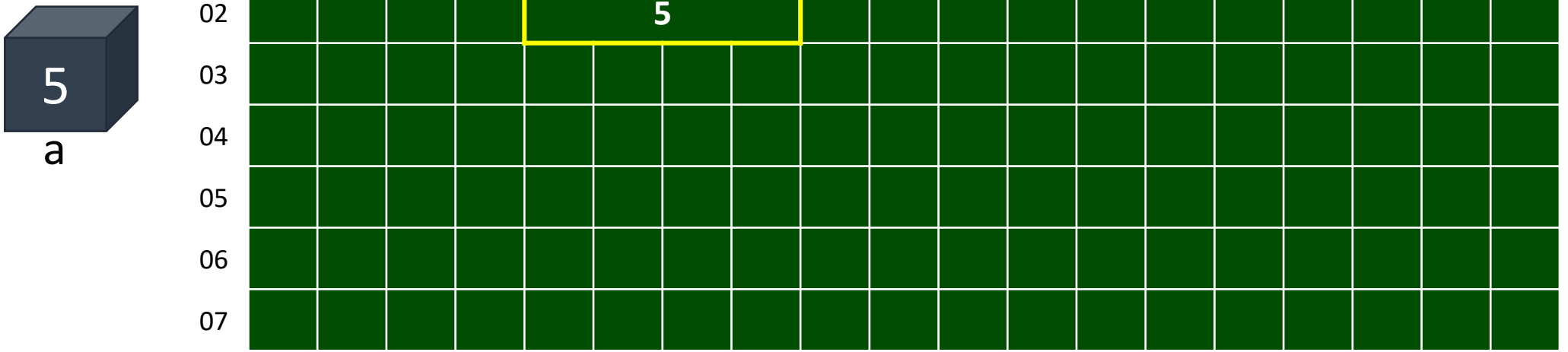
A função onde a foi definida conhece o endereço de a.
A variável a está guardada nos endereços 0204, 0205, 0206 e 0207



Endereçamento - Exemplo

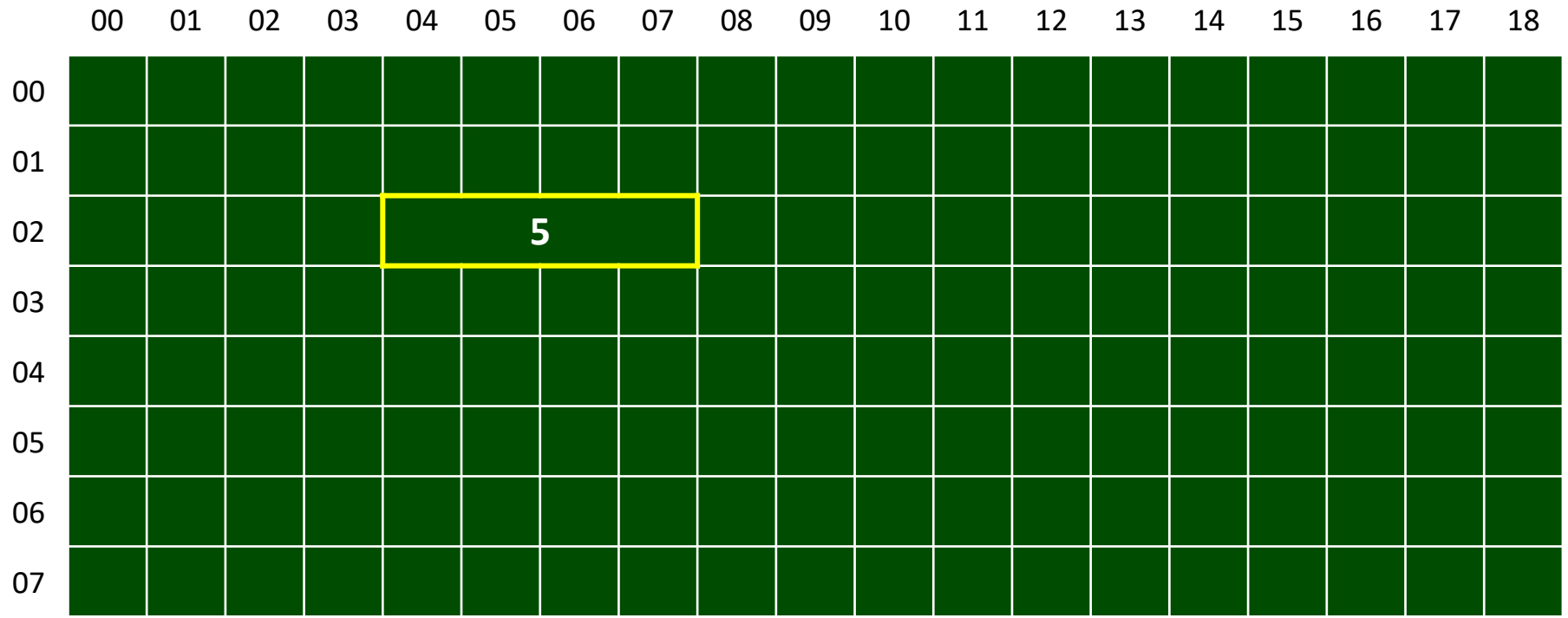
O escopo onde a foi definida conhece o endereço de a.

A variável a está guardada nos endereços 0204, 0205, 0206 e 0207



Endereçamento - Exemplo

E o que acontece quando essas função chama uma outra função (ou procedimento)?
Essa nova função não conhece o endereço de a. Como ela pode acessar seu valor?
Ponteiros!

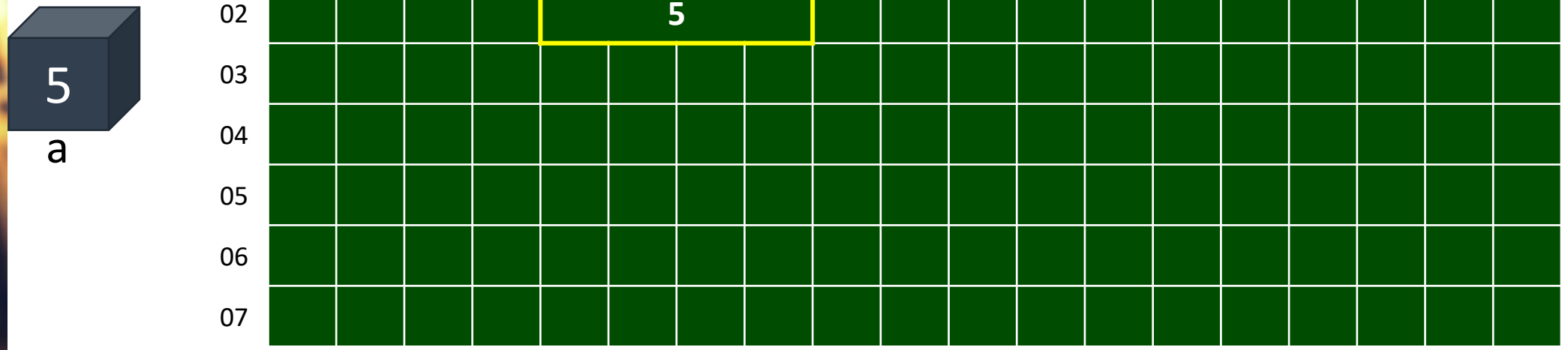


Endereçamento - Ponteiros

int a = 5;

No escopo a foi definida sabemos o endereço de a

a está alocada nos endereços 1104, 1105, 1106 e 1107



5

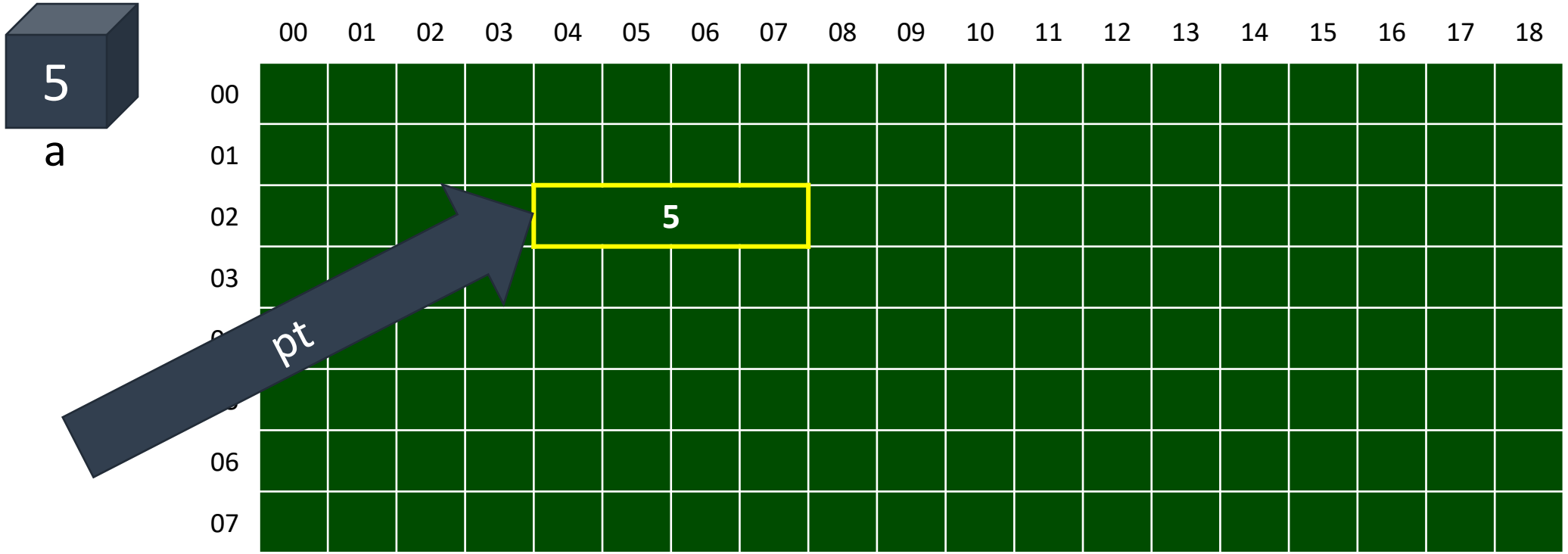
a

Endereçamento - Ponteiros

int a = 5;

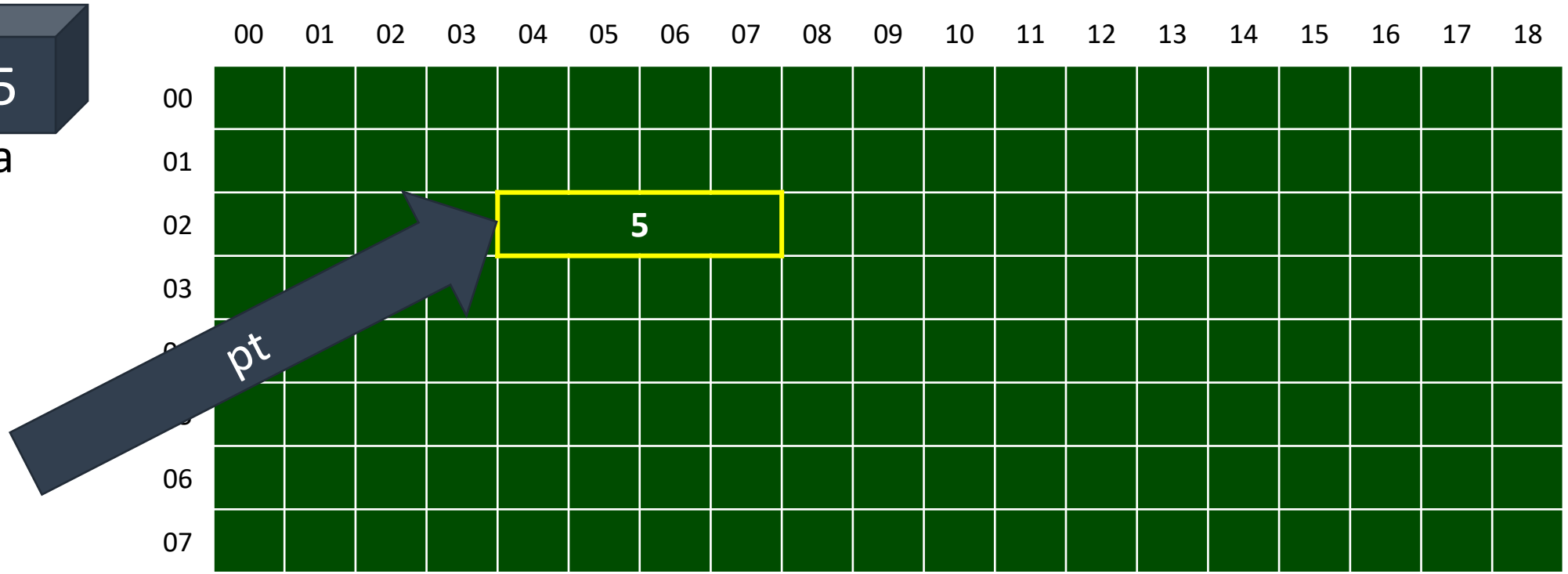
int *pt = &a;

pt agora aponta para onde a está alocada na memória



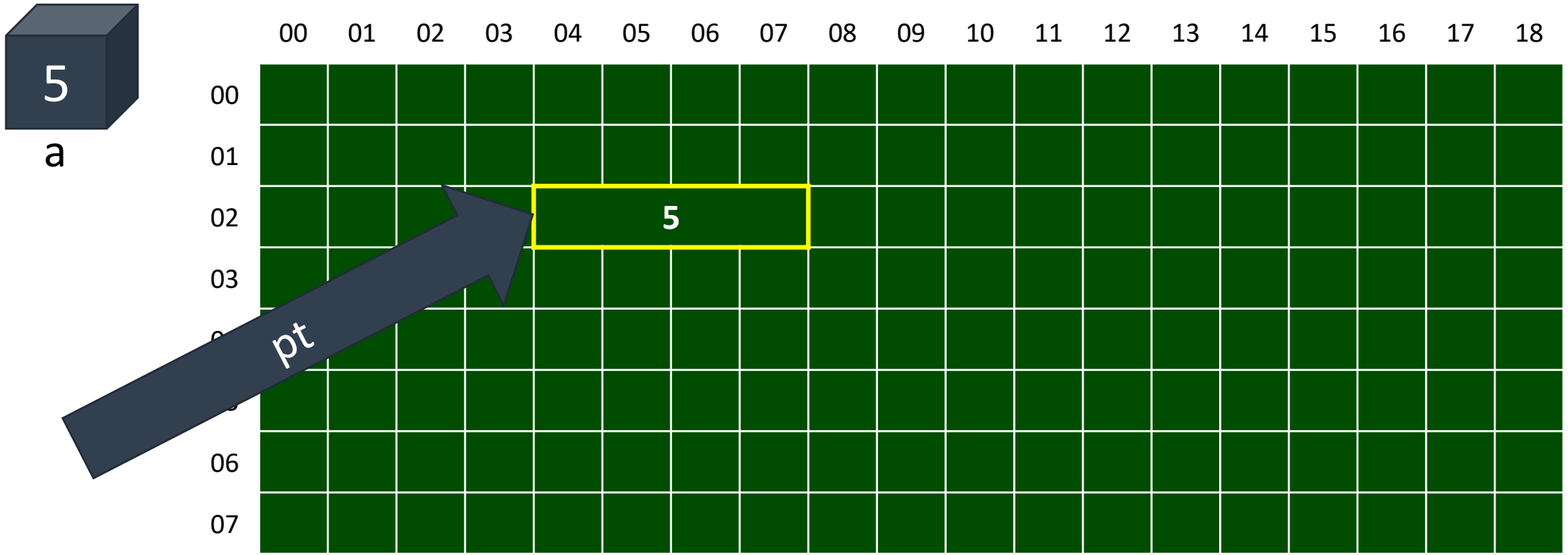
Endereçamento - Ponteiros

O ponteiro pt agora aponta para onde a está alocada na memória
Enquanto a existir, o ponteiro continua válido.



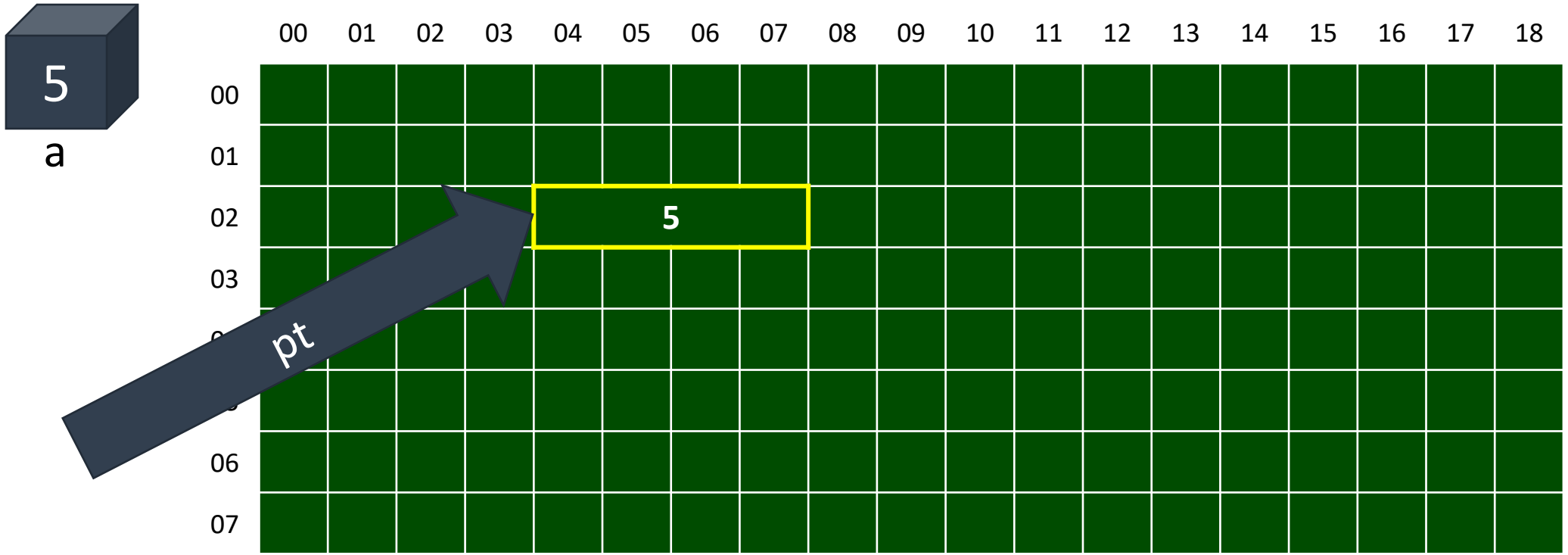
Endereçamento - Ponteiros

O ponteiro pt agora aponta para onde a está alocada na memória
Qualquer função que receba pt pode LER e ESCREVER nesse endereço



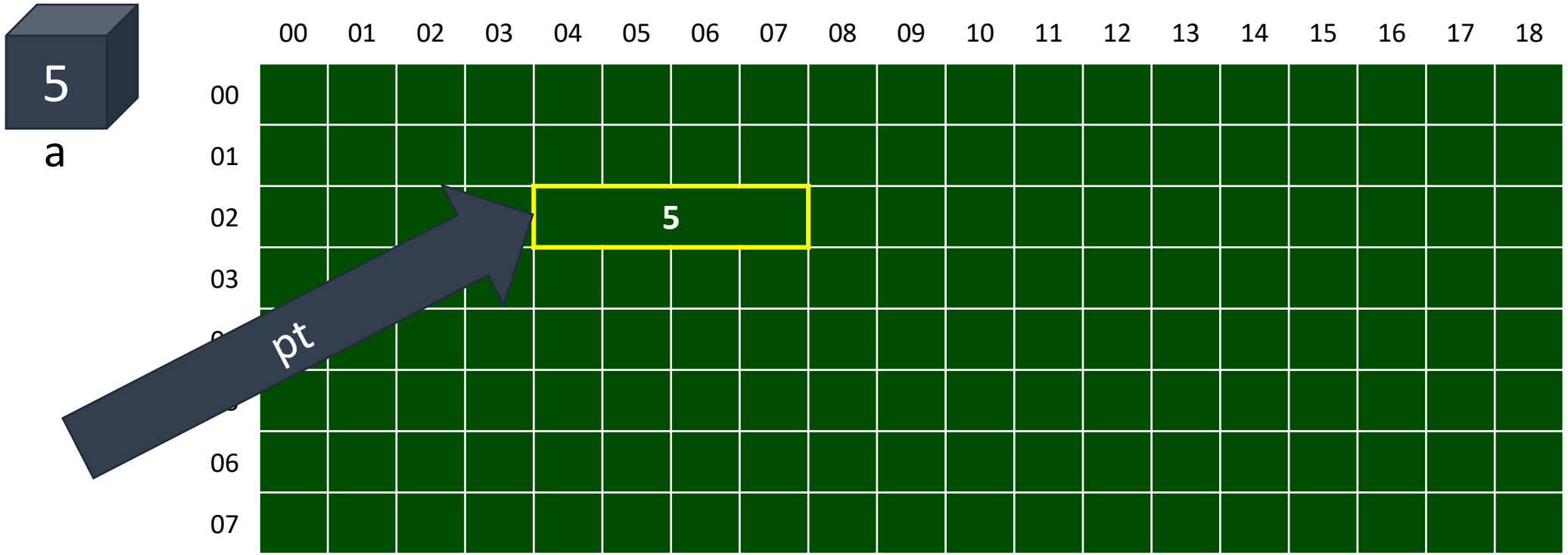
Endereçamento - Ponteiros

Ao LER a área apontada por pt: obtemos o valor da variável a
Ao ESCREVER a área apontada por pt: alteramos o valor da variável a



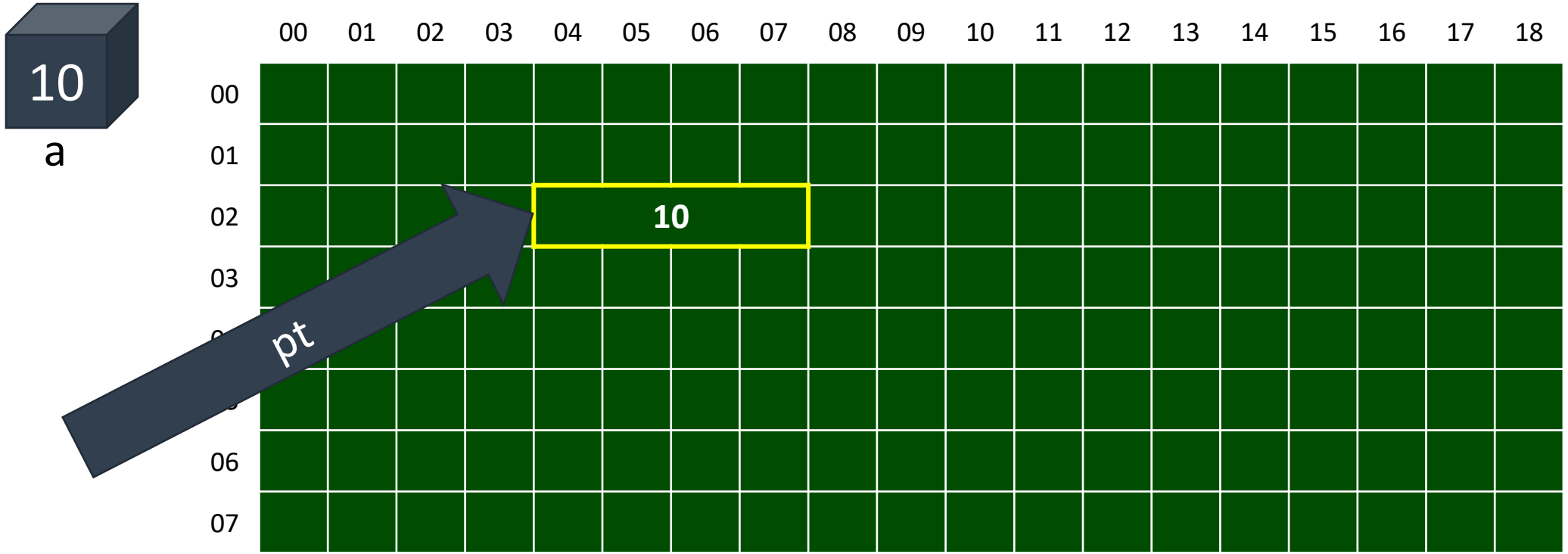
Endereçamento - Ponteiros

*pt = 10; //Siga para o endereço apontado por pt e escreva 10



Endereçamento - Ponteiros

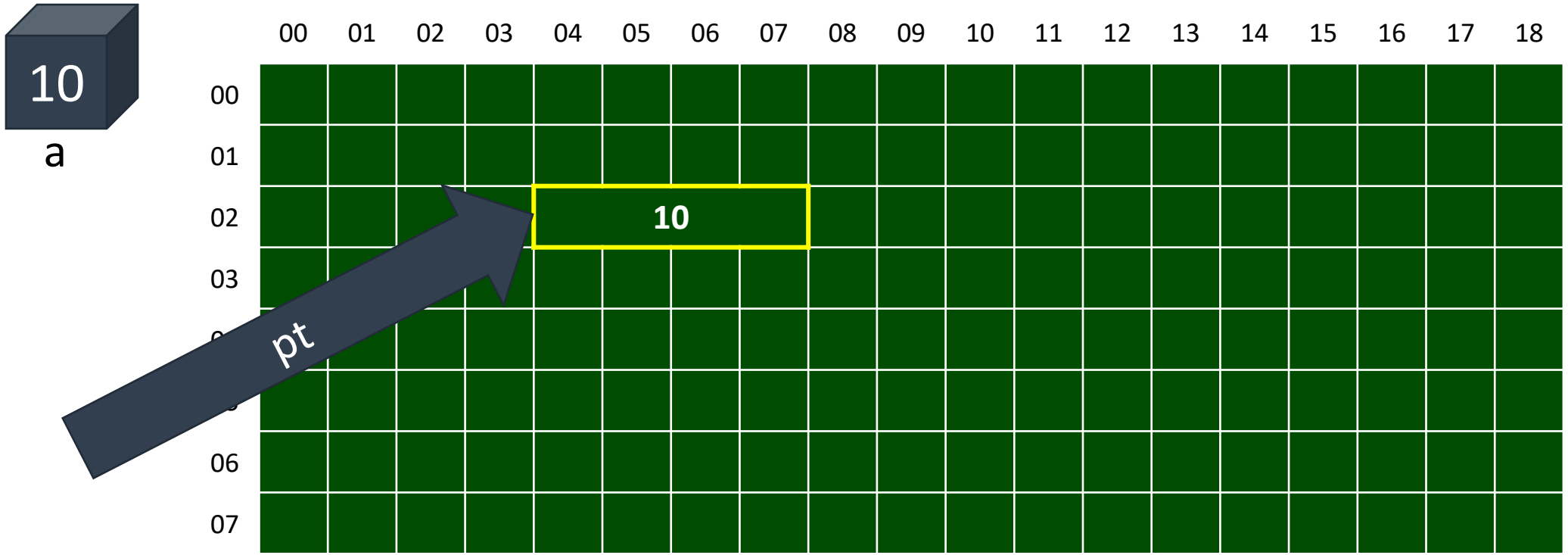
*pt = 10; //Siga para o endereço apontado por pt e escreva 10



Tipos de Ponteiros

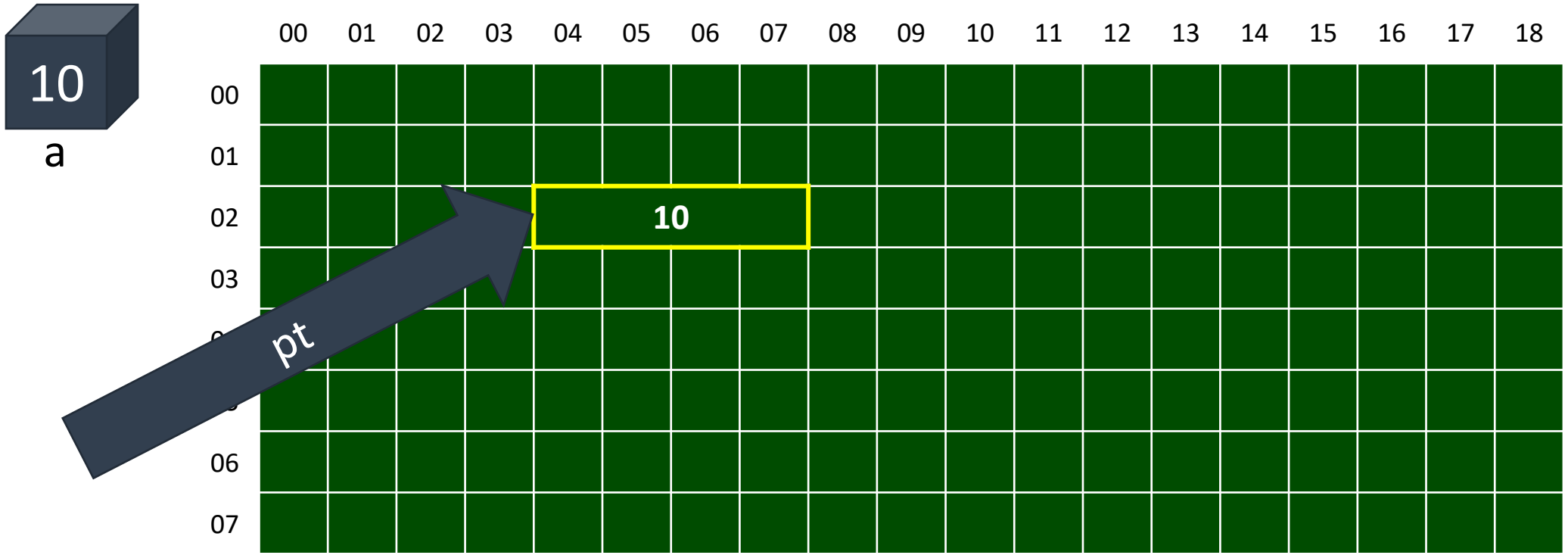
int *pt = &a; //o ponteiro pt aponta para a área da memória onde a está armazenada

Por que definimos ponteiros para inteiros e não somente ponteiros?



Tipos de Ponteiros

Cada tipo de dados consome uma quantidade diferente de bytes
Ao definir <tipo> *<nome>, já indicamos quantos bytes o ponteiro precisa referenciar.
int *pt;



Endereçamento – Na verdade

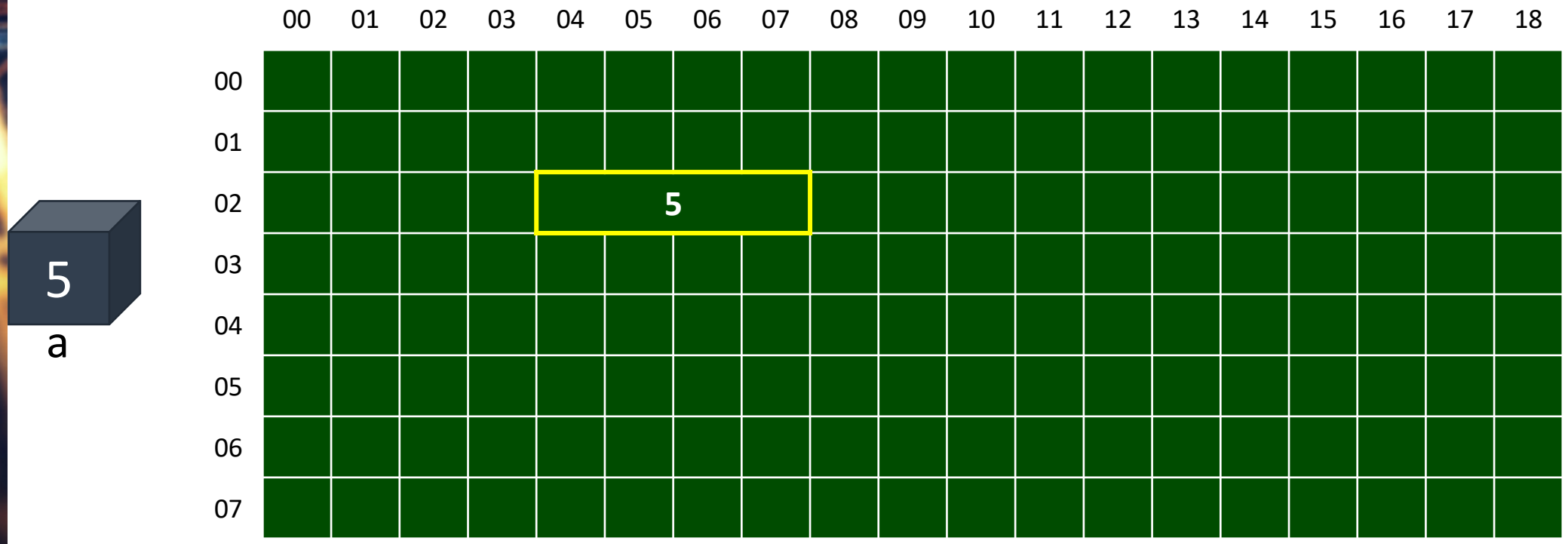
int a = 5;

Inteiro → requer 4 bytes

Onde endereço de da variável a é o primeiro endereço do bloco todo:

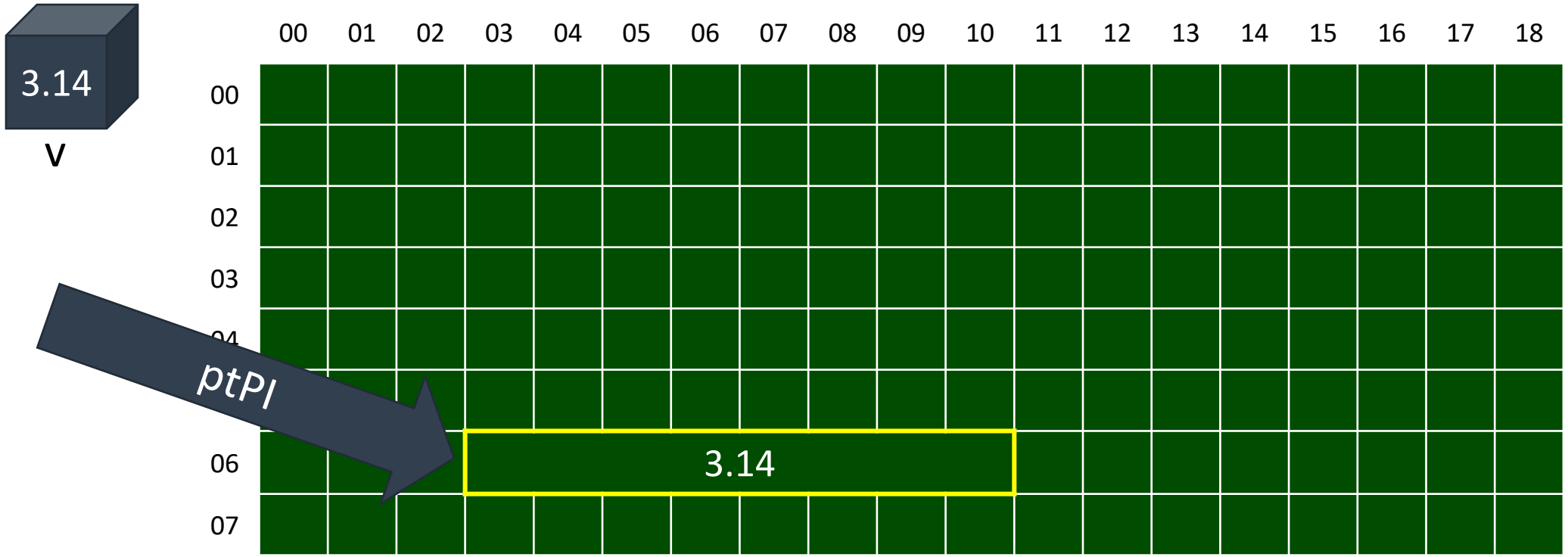
0204, 0205, 0206 e 0207 → 0204

A variável a está alocada em um bloco de 4 bytes iniciado no endereço 0204



Tipos de Ponteiros

double v = 3.14; //double ocupa 8 bytes – v foi alocado nos endereços 0603 - 0610
double *ptPi = &v;



Ponteiros: Importante

- Observe o código abaixo:

```
int* ptA;
*ptA = 10;
```

- Esse código apresenta um erro: O ponteiro foi definido, mas não aponta para lugar nenhum.
- Onde o valor 10 será armazenado?

- Logo ponteiro não aloca memória!
- A alocação de memória deve ser feito pela definição de uma variável ou através da alocação dinâmica (comando malloc).