Introdução a Técnicas de Programação

Ponteiros e alocação dinâmica

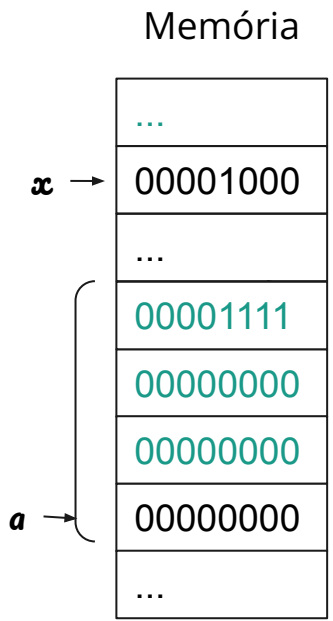
Prof. André Campos
DIMAp/UFRN

Ponteiros e endereços de memória

Como funciona a memória do computador

- Toda variável é associada a um endereço na memória
- Os tipos de variáveis definem quantos bytes elass ocupam, ou seja quantos endereços são usados

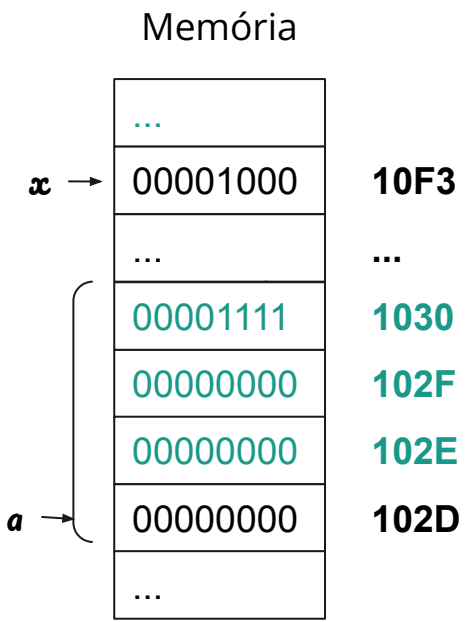
```
int a = 15; // 00000000000000000000000000001111
char x = 8; // 00001000
```



Como funciona a memória do computador

Espaços de 1 byte são “**endereçáveis**”.

- Os endereços são também valores numéricos.
- Normalmente, os endereços são representados em hexadecimal



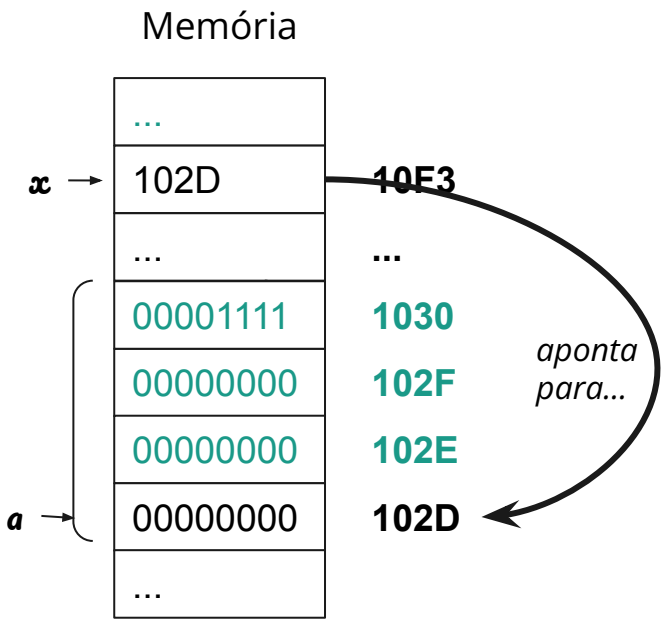
Ponteiros

Se um endereço é um valor numérico e uma variável pode guardar um valor numérico, então ela pode armazenar um endereço

Ponteiros são variáveis especiais que guardam endereços de memória

"Apontam para um endereço"

```
int a = 15;
int *x = &a;
```



Ponteiros

Definição de um ponteiro

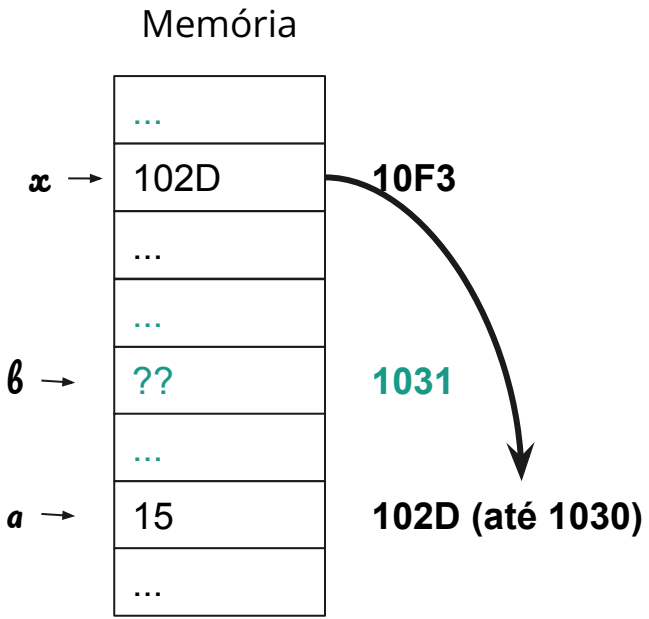
```
tipo *x;
```

Operadores especiais

`&` → retorna o endereço de uma variável

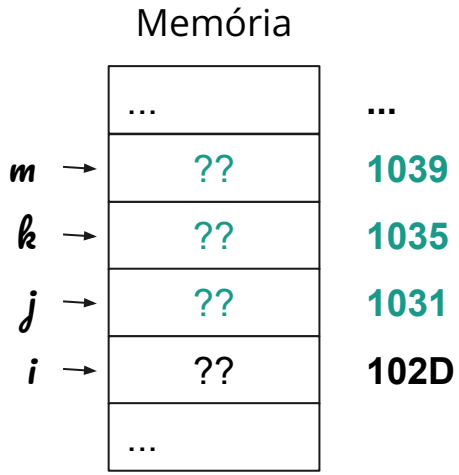
`*` → retorna o conteúdo de um endereço

```
int a = 15;
int *x = &a;
int b = *x;
```



Treinando

```
void main() {
   int i = 2;
   int j = i * i;
   int *k = &i;
   int m = *k * *k;
   *k = j * *k * m;
   i = 40;
}
```



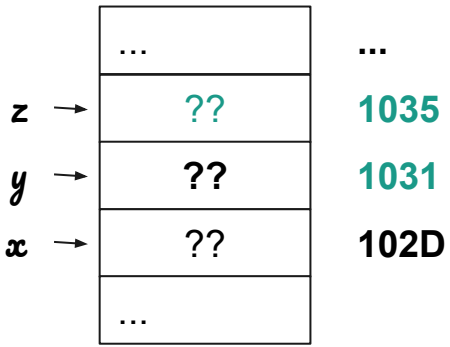
Treinando

```
void main() {
  int x = 5;
  int *y = &x;
  int z = *y;
  cout << *y;
  cout << z;

  x = 7;
  cout << *y;
  cout << z;

  *y = 2;
  cout << x;
  cout << z;
}
```

Memória



Ponteiro para “nenhum endereço”

Muitas vezes é necessário saber quando uma variável do tipo ponteiro foi inicializada, ou seja, quando ela está apontando para um **endereço válido**.

Como endereço é um valor numérico qualquer, foi definido uma constante para indicar que o ponteiro não foi inicializado (está apontando pra nada) **nullptr**.

Dica: sempre que declarar uma variável ponteiro ou:

1. Atribua um endereço de variável para ela (ex: `int *ptr = &a;`), ou
2. Atribua o valor “nulo” (ex: `int *ptr = nullptr;`).

Aplicações de ponteiros

- **Passagem de parâmetros por referência (passagem do endereço)**
  - Útil para alterar os valores de variáveis passadas como parâmetro
- **Variáveis vetores (arranjos) são variáveis ponteiros**
  - O bloco alocado para o vetor possui um endereço (ponteiro)
  - Variáveis que armazenam texto (strings) são vetores especiais, por isso são também ponteiros
- **Alocação dinâmica de variáveis**
  - Alocar um espaço em função, por exemplo, dos dados que o usuário fornece
- Estruturas de dados dinâmicas
  - Listas encadeadas, árvores, grafos ...
- Ponteiros de funções
  - Podemos tratar um função como uma variável qualquer, inclusive passado uma função como parâmetro para outra função

Ponteiros e linguagens de programação

Algumas linguagens evitam acesso direto à memória, apesar de tratarem internamente variáveis como referências (ponteiros), como em **Java** e **Javascript**.

Algumas linguagens permitem o acesso da memória em certas condições (tipos seguros), como em **C#**.

Outras linguagens permitem o acesso direto, como em **C**, **C++**, **Go** e **Rust**.

Ponteiros e arrays

Passagem de valores

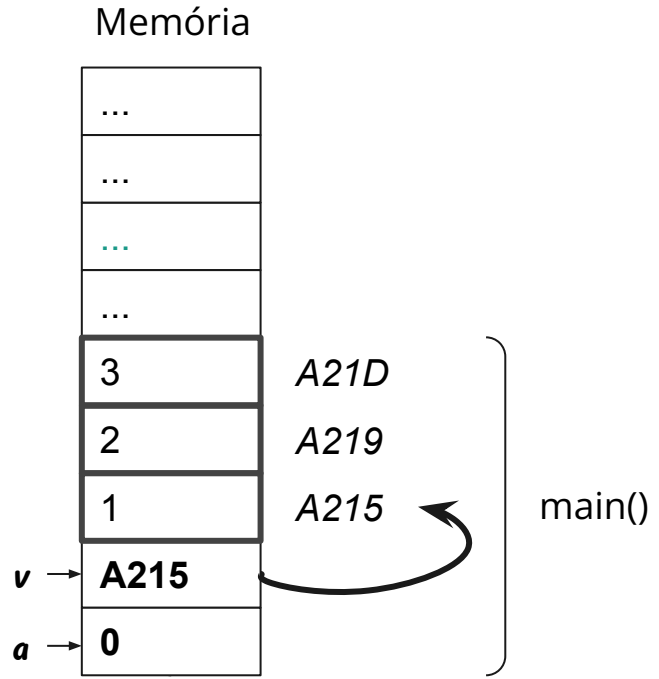
O que será impresso neste programa?

```
void func(int a, int v[3]) {
  a = 5;
  v[0] = 5;
}
int main() {
  int a = 0;
  int v[3] = { 1, 2, 3 };
  func(a, v);
  cout << a << " " << v[0];
  return 0;
}
```

Variáveis de vetores guardam endereços (são ponteiros)

As variáveis de vetores guardam o endereço do elemento inicial do vetor

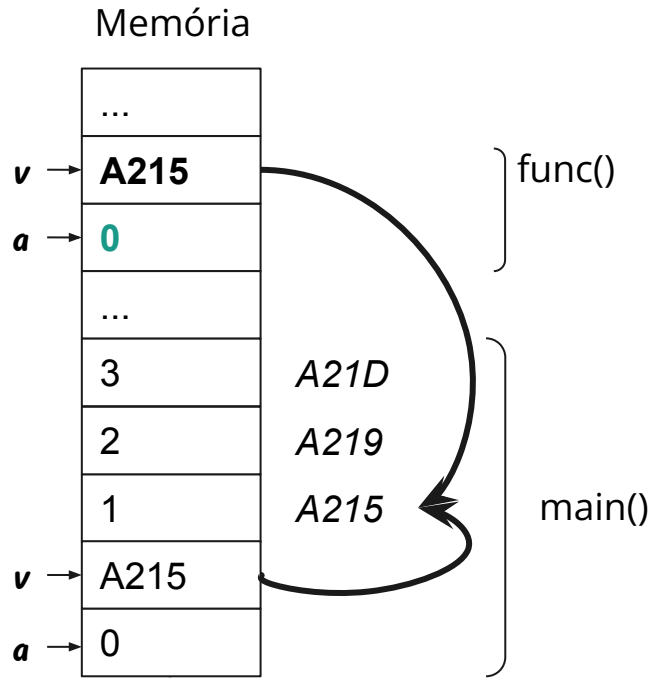
```
void func(int a, int v[3]) {
  a = 5;
  v[0] = 5;
}
int main() {
  int a = 0;
  int v[3] = { 1, 2, 3 };
  func(a, v);
  cout << a << " " << v[0];
  return 0;
}
```



Variáveis de vetores são ponteiros

As variáveis de vetores guardam o endereço do elemento inicial do vetor

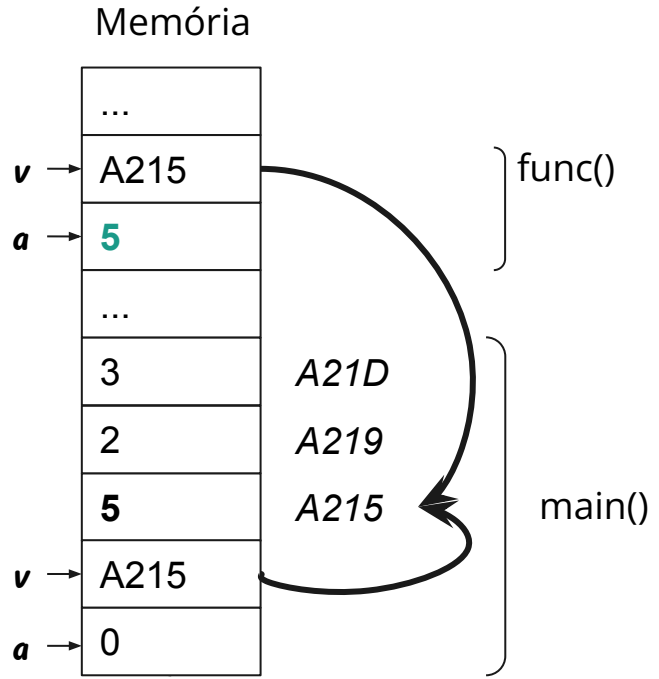
```
void func(int a, int v[3]) {
  a = 5;
  v[0] = 5;
}
int main() {
  int a = 0;
  int v[3] = { 1, 2, 3 };
  func(a, v);
  cout << a << " " << v[0];
  return 0;
}
```



Variáveis de vetores são ponteiros

As variáveis de vetores guardam o endereço do elemento inicial do vetor

```
void func(int a, int v[3]) {
  a = 5;
  v[0] = 5;
}
int main() {
  int a = 0;
  int v[3] = { 1, 2, 3 };
  func(a, v);
  cout << a << " " << v[0];
  return 0;
}
```



Aritmética de ponteiros

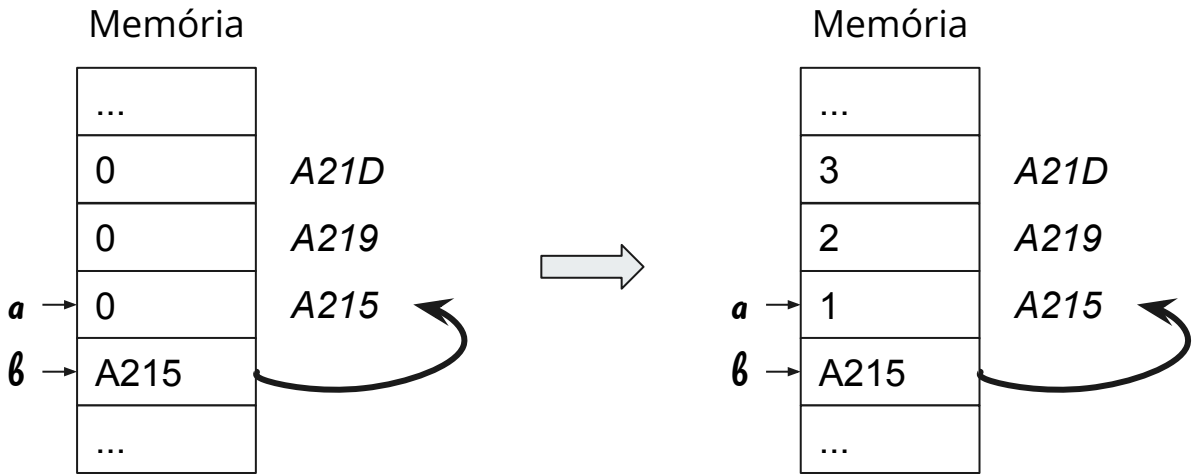
Um endereço, por ser um número, pode ser somado e subtraído

Somar um endereço de 1 resulta no “endereço seguinte”

- O “seguinte” depende do tipo de dado. O tipo `int` salta 4 valores porque usa 4 bytes.

Subtrair de 1 resulta no “endereço anterior”

```
int main() {
  int a = 0;
  int *b = &a;
  *b = 1
  *(b + 1) = 2;
  *(b + 2) = 3;
}
```

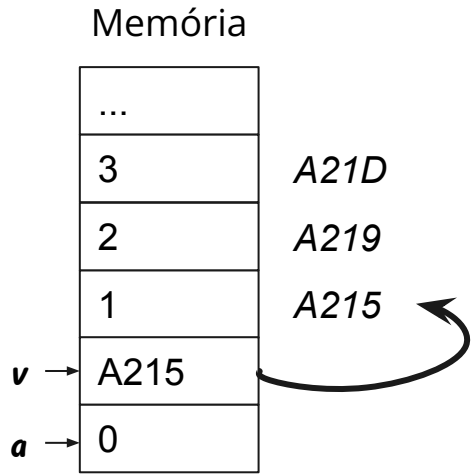


Vetores e aritmética de ponteiros

Vetores usam a soma e subtração de endereços para acessar seus dados
Acessar o valor de índice *i* de um vetor é acessar o conteúdo do endereço do vetor mais *i*. `a[i]` ⇐⇒ `*(a+i)`

OBS: Só faz sentido usar aritmética de ponteiros em vetores (blocos de memória alocados de forma contígua)

```c
int main() {
  int a = 0;
  int v[3] = { 1, 2, 3 };
  *(v + 0) = 1;   // v[0] = 1;
  *(v + 1) = 2;   // v[1] = 2;
  *(v + 2) = 3;   // v[2] = 3;
}
```

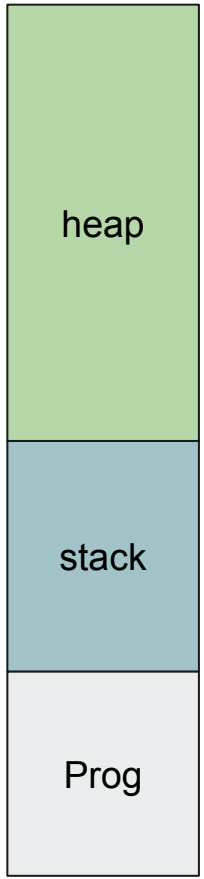


Alocação dinâmica

Memória

Quando um programa é iniciado, além do espaço para o programa na memória, o S.O. reserva dois outros espaços:

- Stack
  - Onde as variáveis são armazenadas e as rotinas empilhadas
  - Acesso rápido e eficiente (gerenciado pela CPU)
  - Não há fragmentação de espaço
  - Limitada pelo S.O.
- Heap
  - Pode ser acessado a partir de qualquer ponto do programa
  - Acesso mais lento (gerenciado pelo S.O.)
  - Pode haver fragmentação de espaço
  - Limitada pela memória do computador



Alocação estática e dinâmica

Variáveis locais são alocadas na stack.

Dizemos que é uma **alocação estática** (fixa e pré-definida na compilação)

> **Obs:** *A memória alocada para as variáveis locais, guardadas na stack, são liberadas quando a função onde são definidas termina.*

Às vezes, é necessário:

- Alocar um tamanho desconhecido durante a compilação
- Alocar um espaço muito...muito grande
- Alocar um espaço que pode ser alterado durante a execução

Precisamos de uma **alocação dinâmica**

Exemplo de necessidade de alocação dinâmica

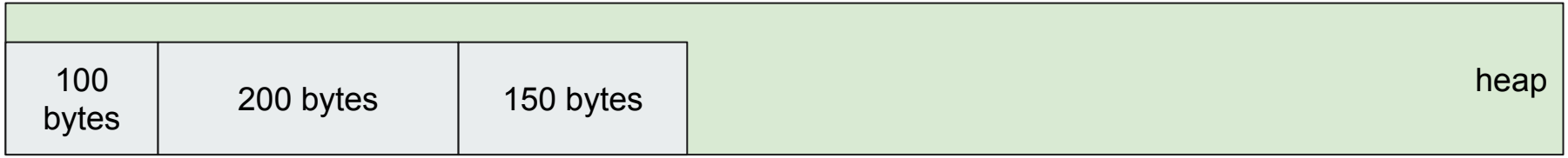
Caso exemplo

- Ler uma imagem (ex: PPM)
  - Tamanho desconhecido durante a compilação
  - Tamanho pode ser muito...muito grande
- Ampliar ou reduzir a imagem lida
  - Tamanho pode ser alterado durante a execução

Alocação dinâmica

Alocação na heap

- Definimos o tamanho do bloco de memória a ser alocado
- Liberamos o bloco quando não precisamos mais
- Precisamos guardar a referência (ponteiro) para acessar o conteúdo e liberar o bloco
- Se todo o espaço da heap for preenchido, o S.O. reserva mais espaço (limitado à memória disponível no computador)



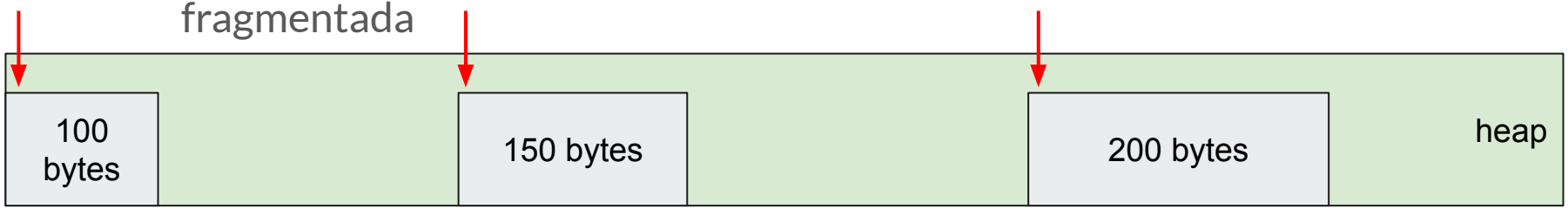
Alocação dinâmica

Acesso ao conteúdo alocado

- Precisamos guardar a referência (ponteiro) para acessar o conteúdo de um bloco alocado

Desalocação na heap

- A referência também é necessária para indicar o bloco a ser liberado
- Após várias alocações e desalocações, a memória pode ficar fragmentada

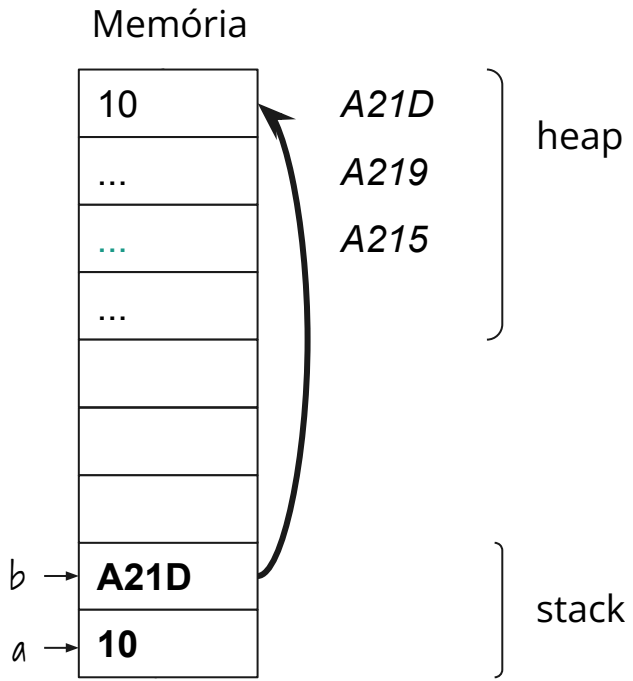


Alocação dinâmica em C++

Operador `new` aloca memória da heap

```
int main() {
  int a;
  int *b = new int;

  a = 10;
  *b = 10;
}
```

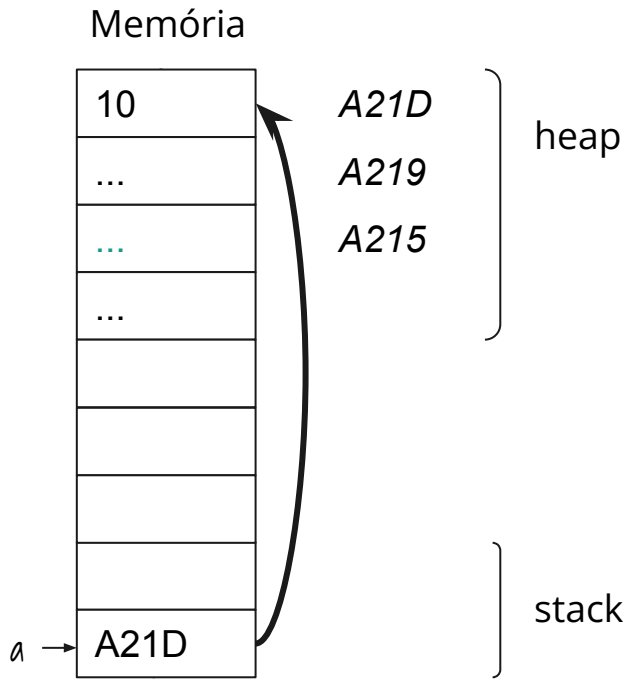


Alocação dinâmica em C++

Operador `delete` libera memória alocada anteriormente. O delete libera a memória, mas não destrói a variável do ponteiro, que pode ser usada novamente.

```
int main() {
  int *a = new int;

  *a = 10;
   …
  delete a;
}
```

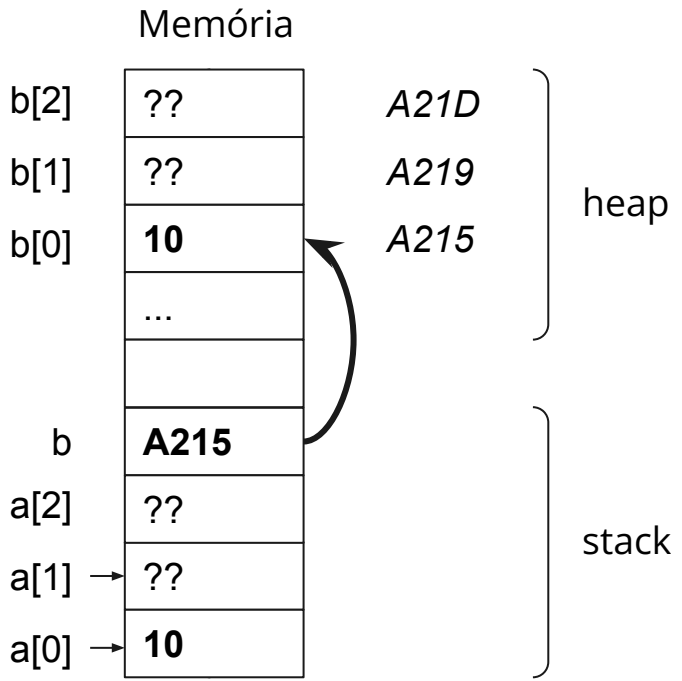


Alocação de arrays dinâmicos

Uso de [ ] para definir o tamanho do array.

```
int main() {
  int a[3];
  int *b = new int[3];

  a[0] = 10;
  b[0] = 10;
}
```



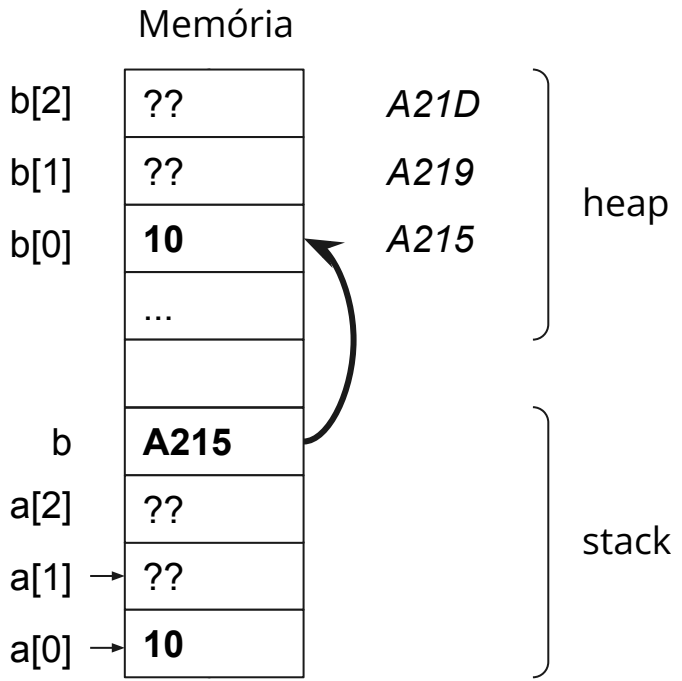
Alocação de arrays dinâmicos

Operador delete[] é usado para liberar um array dinâmico.

```
int main() {
  int a[3];
  int *b = new int[3];

  a[0] = 10;
  b[0] = 10;

  delete[] b;
}
```



Alocação de matrizes

Matrizes são “arrays de arrays”, portanto:

- A variável deve ser “ponteiro de ponteiro”
- A alocação e liberação da memória requer um laço.

Alocação

```
int **matriz = new int*[lin];
for(int i = 0; i < lin; i++) {
  matriz[i] = new int[col];
}
```

Liberação

```
for(int i = 0; i < lin; i++) {
  delete[] matriz[i];
}
delete[] matriz;
```

Prática

1. Implemente uma função que soma dois vetores (sequências de inteiros). A função deve retornar uma nova sequência alocada dinamicamente.
2. Implemente uma função para criar uma matriz identidade cujo tamanho é passado por parâmetro. A função deve retornar a matriz criada dinamicamente.
3. Implemente uma função para desalocar uma matriz alocada dinamicamente.