UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Armazenamento e manipulação de dados em memória

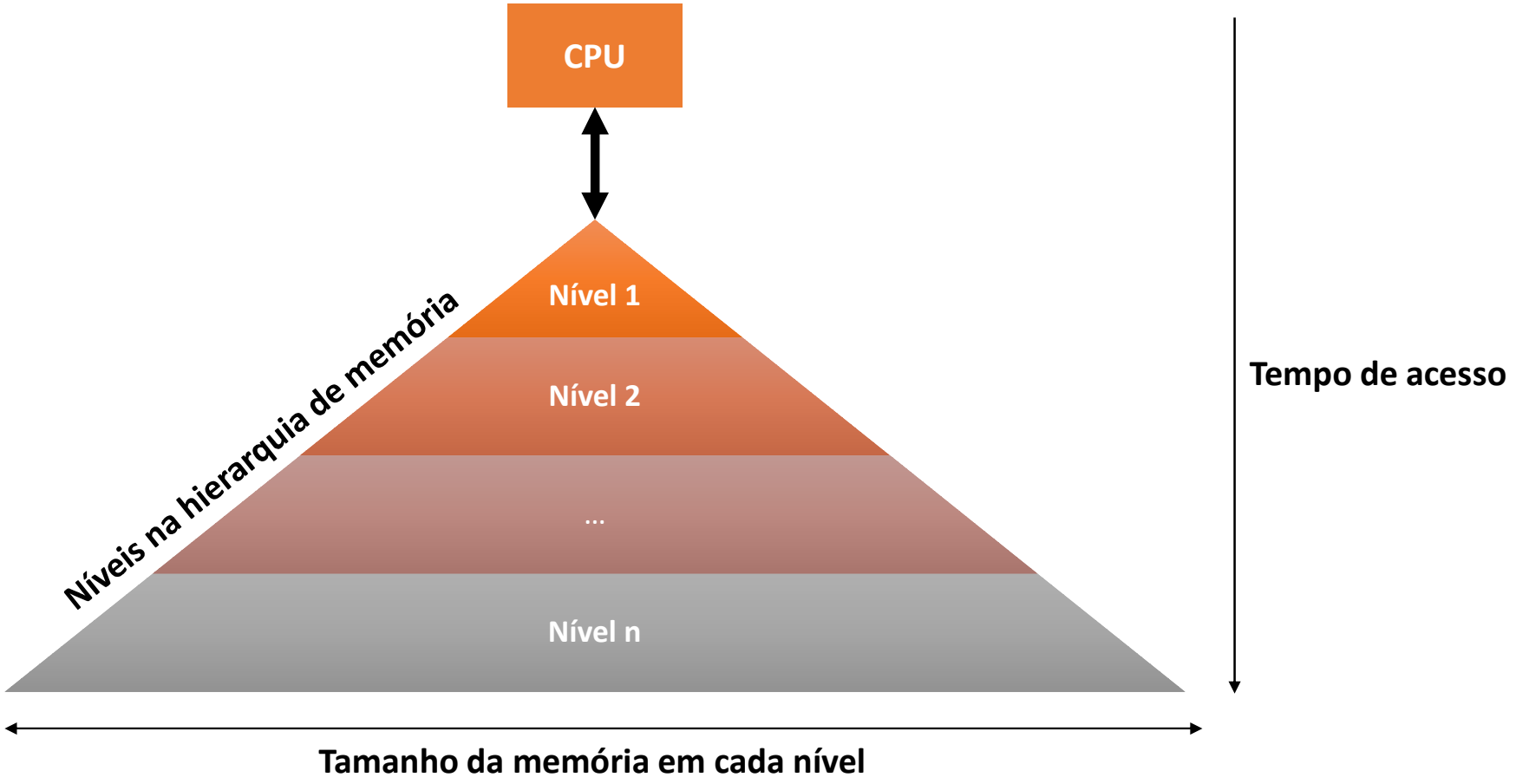
Prof. Luiz Chaimowicz
(slides adaptados do Prof. Douglas Macharet)

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

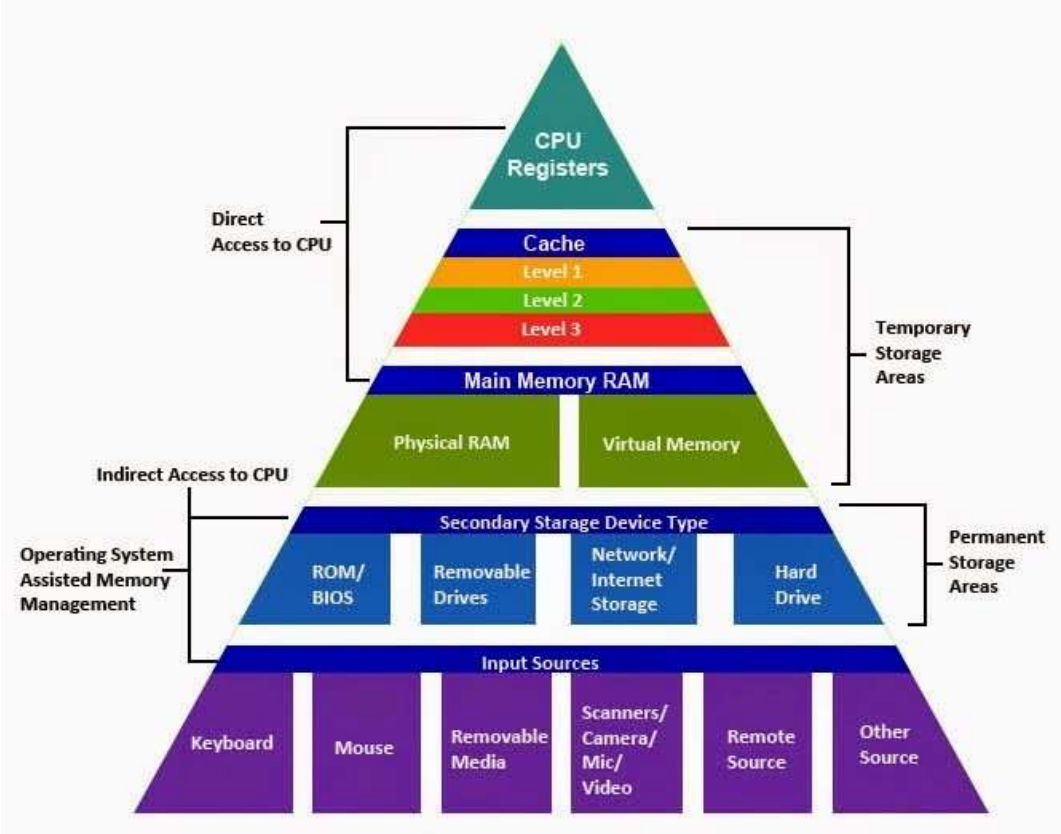
Hierarquia de memória

- Memória
  - Estrutura interna que armazena informações
- Memória principal
  - DRAM (Dynamic Random-Access Memory)
  - Armazenamento temporário
- Memória secundária
  - Tecnologias Magnéticas e Ópticas
  - Não voláteis

Hierarquia de memória



Hierarquia de memória



Hierarquia de memória

- Corpo humano (inspiração?)
  - Lembranças recentes
    - Memórias menores, curta duração
  - Lembranças mais antigas
    - Memórias de maior capacidade, longa duração
- Princípio da localidade
  - Temporal
  - Espacial

Hierarquia de memória
Princípio da localidade – Temporal

- Dado acessados recentemente têm mais chance de serem usados novamente do que dados usados há mais tempo
- Exemplo
  - Comandos de repetição
  - Funções
- Manter os dados e instruções usados recentemente no topo da Hierarquia (acesso mais rápido)

Hierarquia de memória
Princípio da localidade – Espacial

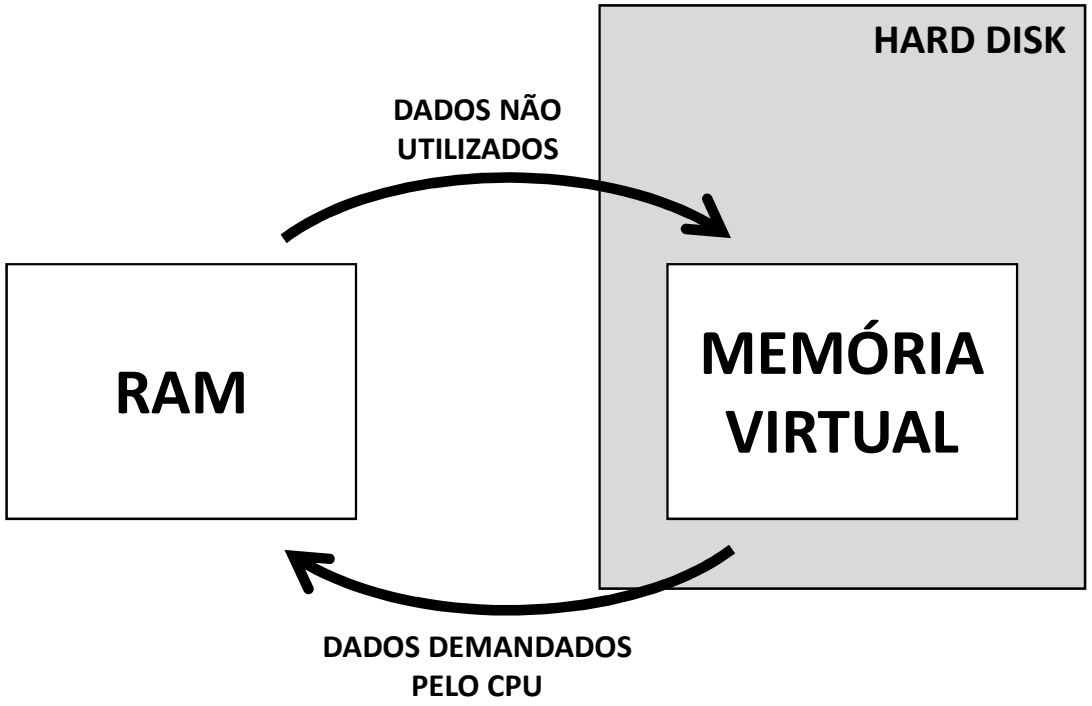
- Probabilidade de acesso maior para dados e instruções em endereços próximos àqueles acessados recentemente
- Exemplo
  - Acesso às posições de um vetor
- Variáveis são armazenadas próximas uma às outras
- Vetores e matrizes armazenados em sequência
  - Levando em consideração seus índices

Hierarquia de memória
Memória virtual

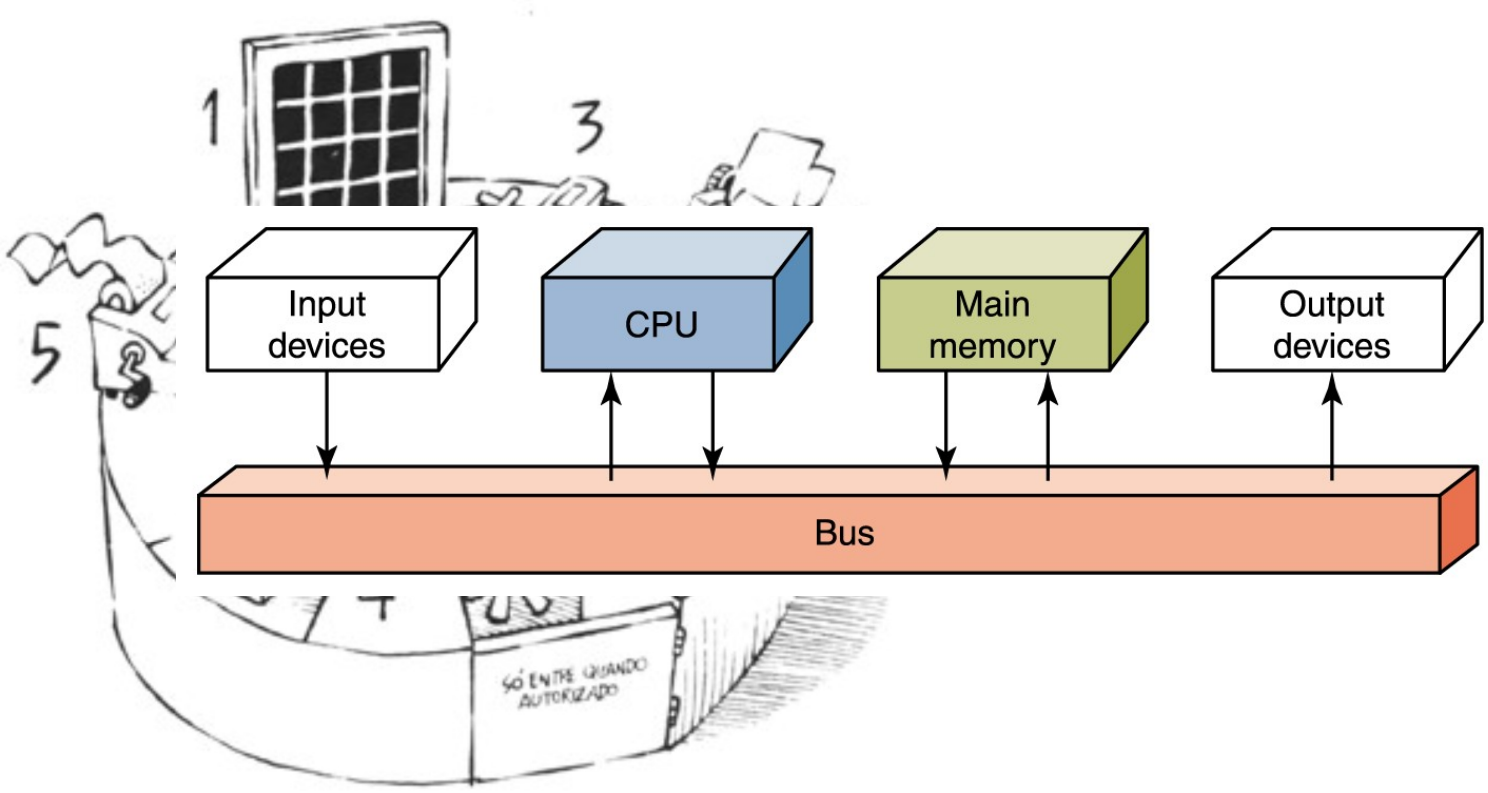
- Maior demanda da memória principal
  - Programas cada vezes maiores
  - Queda no custo não teve o mesmo ritmo
- Como resolver esse problema?
- Memória virtual
  - Memória (RAM) → Memória Secundária (HD)
  - Busca hierárquica pela informação

Hierarquia de memória

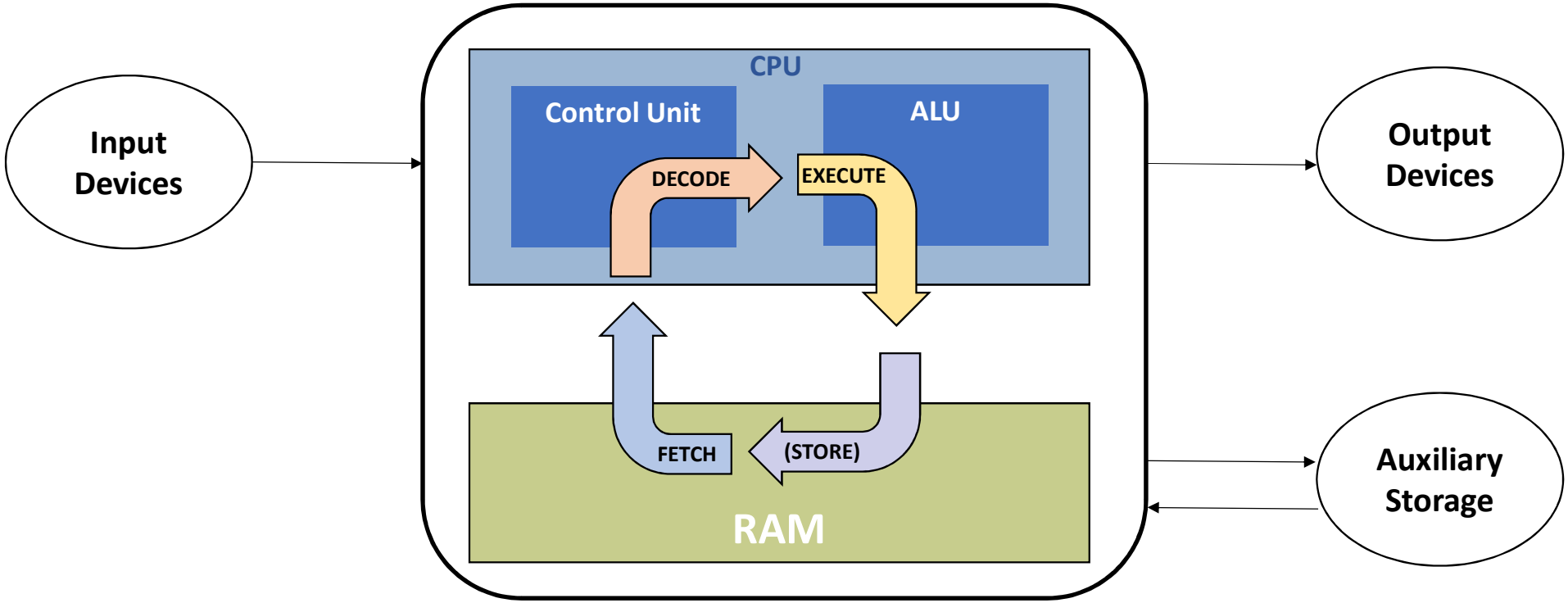
Memória virtual



Computador simplificado



Fetch-Decode-Execute



*Assunto mais aprofundado em Arquitetura de Computadores, Sistemas Operacionais, …

https://www.youtube.com/watch?v=xs5oq-i_rTc

Alocação de memória

- Segmentos da memória
  - **Código/Globais (code/data)**
    - Guarda o código compilado do programa e variáveis globais/estáticas
  - **Stack (pilha)**
    - Espaço que variáveis dentro de funções (locais) são alocadas
  - **Heap**
    - Espaço mais estável (durável) de armazenamento

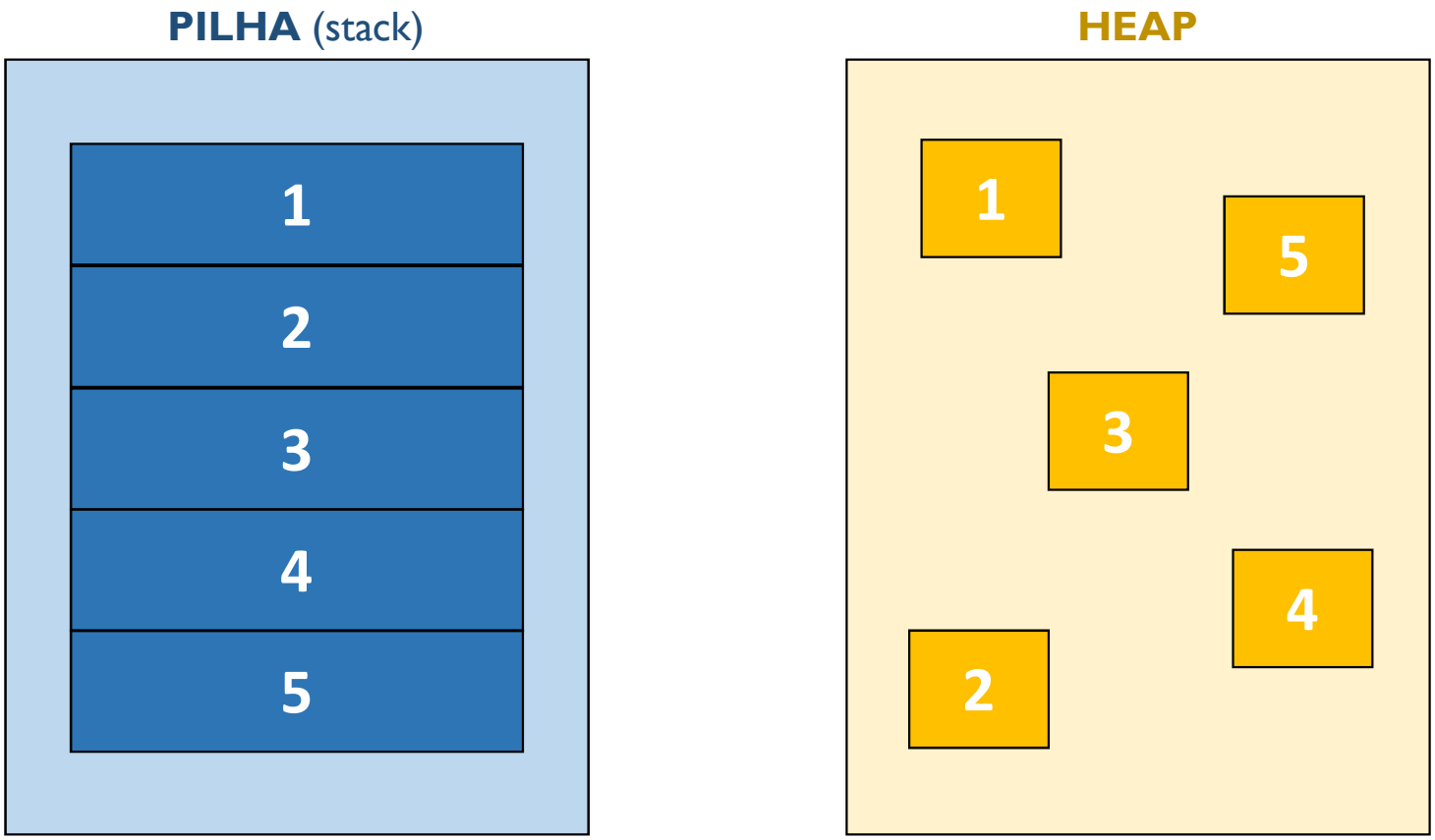
Alocação de memória
Stack (pilha)

- Porção contígua/sequencial de memória
  - Escopo de variável: incrementado toda vez que um certo método é chamado, liberado quando ele é finalizado
- LIFO (last-in-first-out)
  - Último elemento a entrar é o primeiro a sair
- Não é necessário gerenciar manualmente
- Possui tamanho fixo na execução (depende do SO)

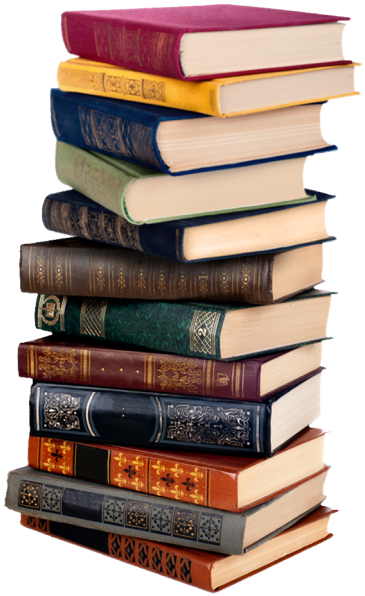
Alocação de memória
Heap

- Espaço de memória de propósito geral
- Não impõe um padrão de alocação
  - Fragmentação ao longo do tempo
- Gerenciamento **explícito**
  - Alocação/desalocação manuais!
- "Não" possui um limite de tamanho

Alocação de memória



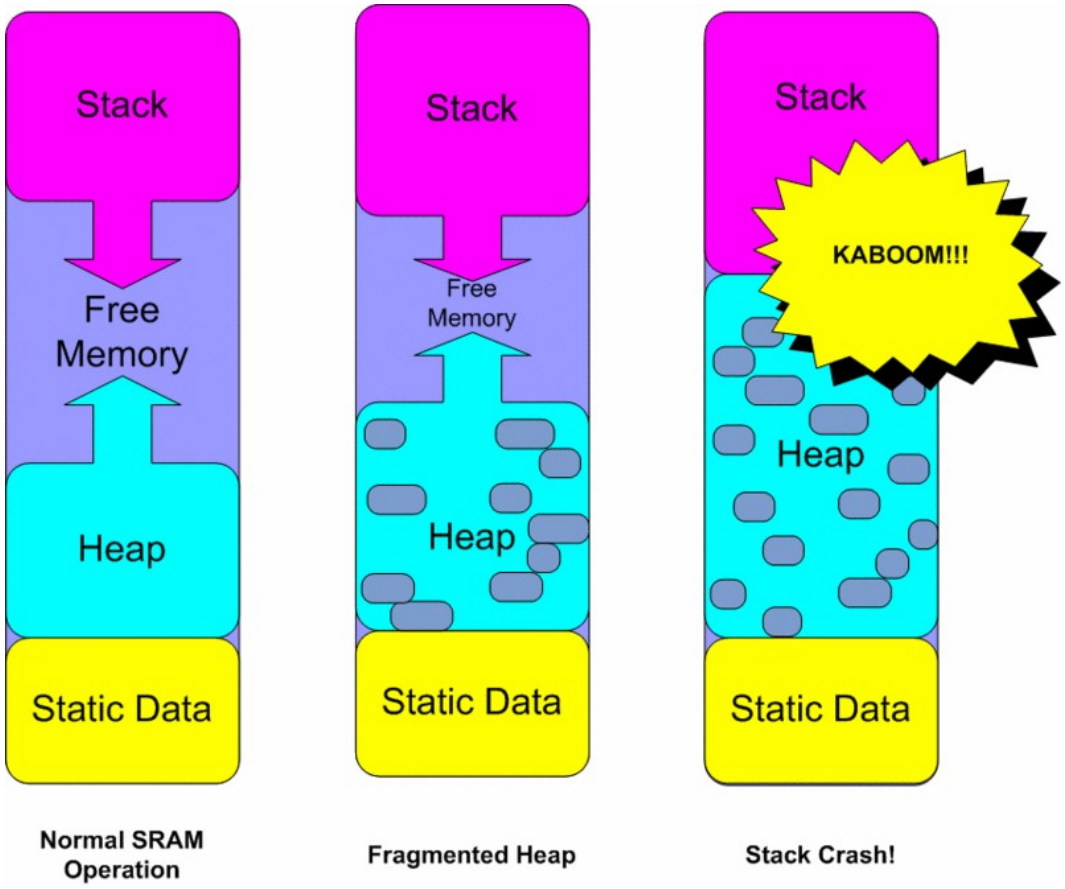
Alocação de memória



STACK

HEAP

Alocação de memória



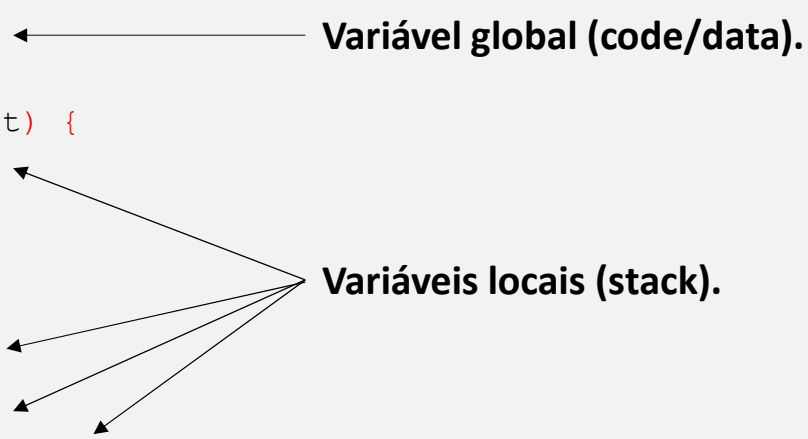
Alocação de memória

- Tipos de alocação
  - Estática → **Pilha**
  - Dinâmica → **Heap**

- Estática
  - Memória necessária é fixa/conhecida durante o desenvolvimento
  - Associada ao tipo/compilador sendo utilizado em um contexto
  - Compilador é responsável por todo o gerenciamento

Alocação de memória
Exemplo 1

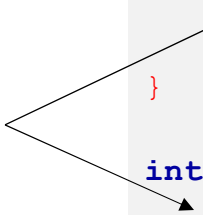
```cpp
#include <iostream>
#include <iomanip>

int valor_global = 100;

double dobrar_valor(double input) {
    double dobro = input * 2.0;
    return dobro;
}

int main() {
    int idade = 30;
    double salario = 12345.67;
    double lista[3] = {1.2, 2.3, 3.4};

    std::cout << std::fixed << std::setprecision(2) << dobrar_valor(salario);
    return 0;
}
```



Código online

Alocação de memória
Exemplo 2

As chamadas de funções também vão para a pilha!



```
#include <iostream>

int fatorial(int n) {
    if (n > 1)
        return n*fatorial(n-1);
    else return 1;
}

int main() {
    int fat1 = fatorial(3);
    std::cout << fat1 << std::endl;

    int fat2 = fatorial(10);
    std::cout << fat2 << std::endl;

    return 0;
}
```

Código online

Alocação dinâmica
Heap

- Maior controle na manipulação → Mais responsabilidade
  - Armazenamento de grandes quantidades de dados com tamanho máximo é desconhecido (não fixo) na implementação (execução)
  - Tamanho pode variar após o início da execução
  - Não estão associadas a um escopo específico!
- C/C++
  - Utilização de ponteiros
  - Manuseio da memória de maneira explícita

Ponteiros

- Armazenam um endereço de memória
- Normalmente utilizados para referenciar uma área de memória alocado dinamicamente (Stack → Heap)
- Mas na verdade podem apontar para qualquer região de memória, e portanto devem ser utilizados com atenção

Ponteiros

```
int main() {
  int var;
  int *p;

  var = 10;
  p = &var;
  *p = 20;

  return 0;
}
```

MEMÓRIA (Stack)

Endereço	Nome	Valor
0x0000		
0x0004	var	~~10~~ 20
0x0008	p	0x0004
0x000c		
0x0010		

.
.
.

Ponteiros

- **Referência**
  - **&x**
  - Endereço de memória da variável **x**

- **Deferência** (dereferência)
  - ***x**
  - Conteúdo do endereço apontado por **x**

Ponteiros

```c
int main() {
  int i;
  int *ponteiro;
  int **ppp;

  i = 10;
  ponteiro = &i;
  ppp = &ponteiro;



  return 0;
}
```

MEMÓRIA

Endereço	Nome	Valor
0x0000		
0x0004	i	10
0x0008	ponteiro	0x0004
0x000c	ppp	0x0008
0x0010		

Ponteiro para um **Ponteiro** de **Inteiro**!
Armazena o endereço de um apontador

Ponteiros
Operadores

C

- Alocação: **malloc**
- Liberação: **free**

C++

- Alocação: **new**
- Liberação: **delete**

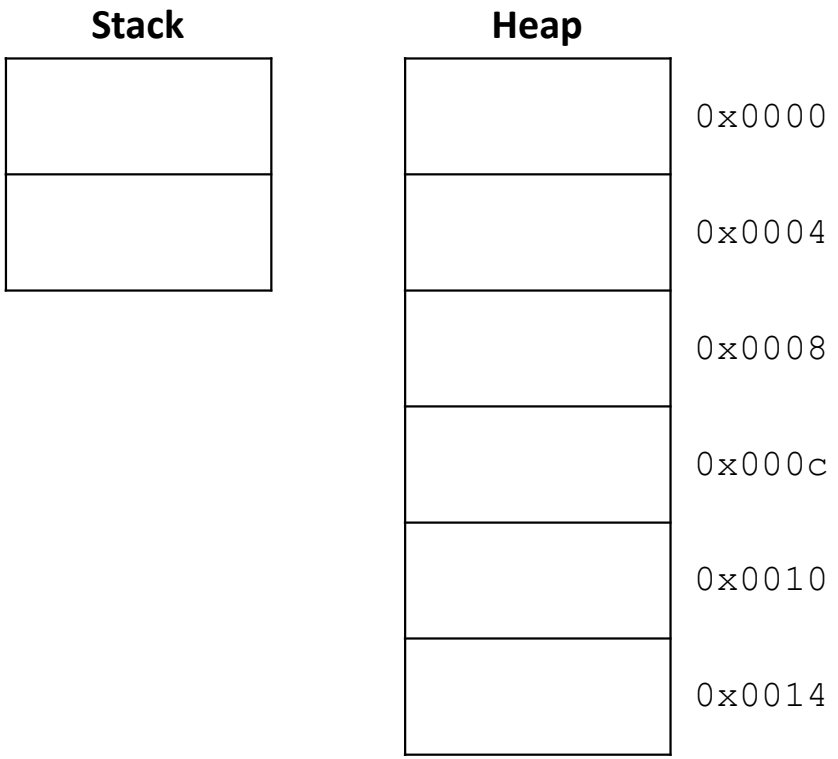
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```

- Onde cada uma dessas variáveis será alocada?
- Qual o valor (conteúdo) de 'b' ao final?
- Existe algum problema com esse código?

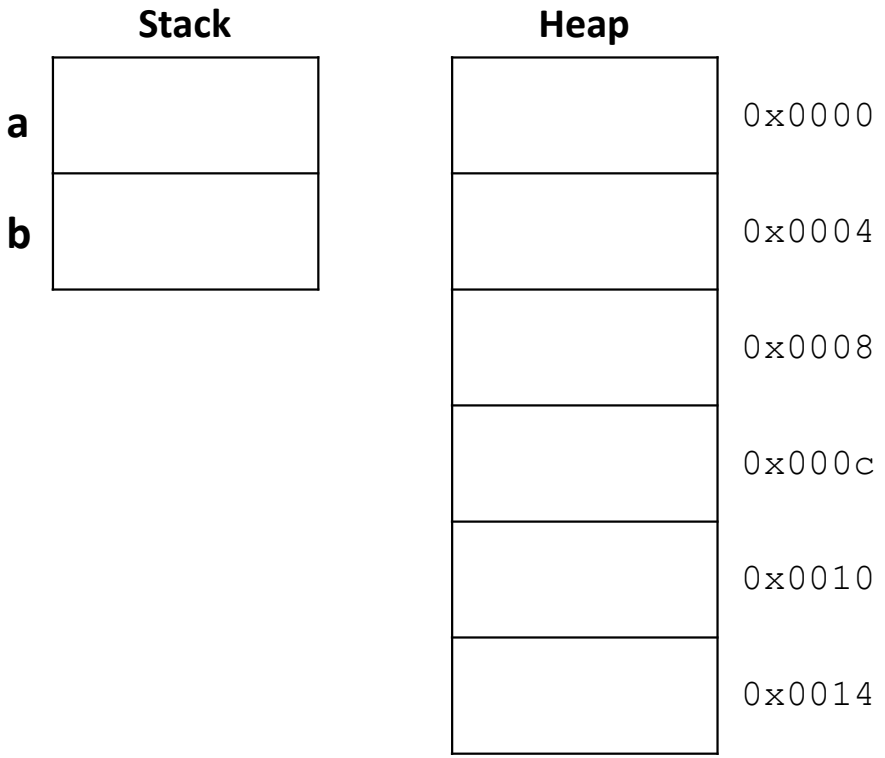
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



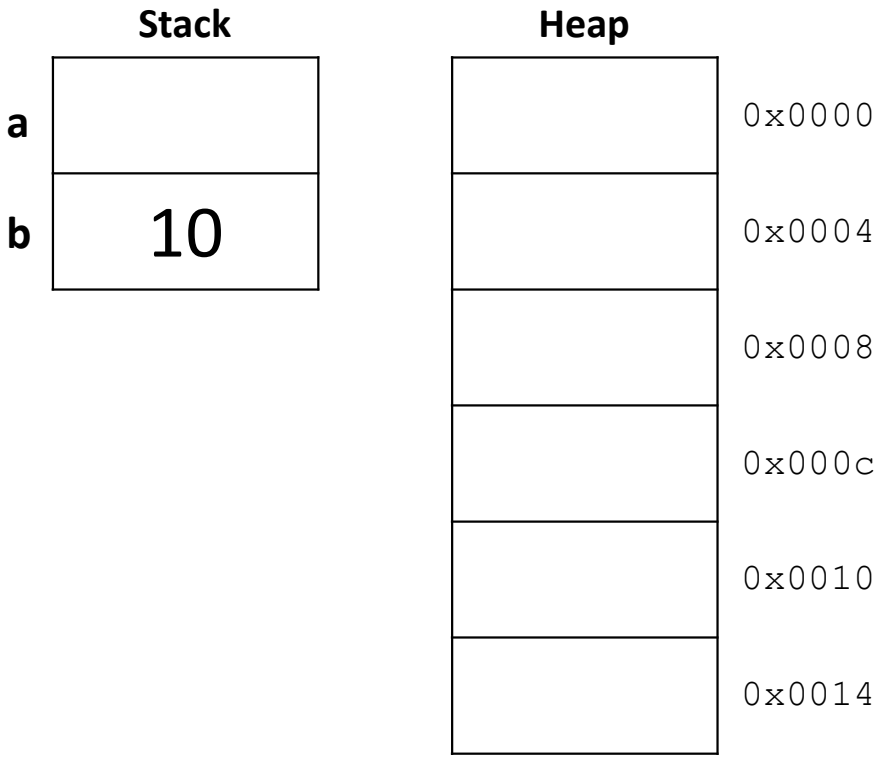
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



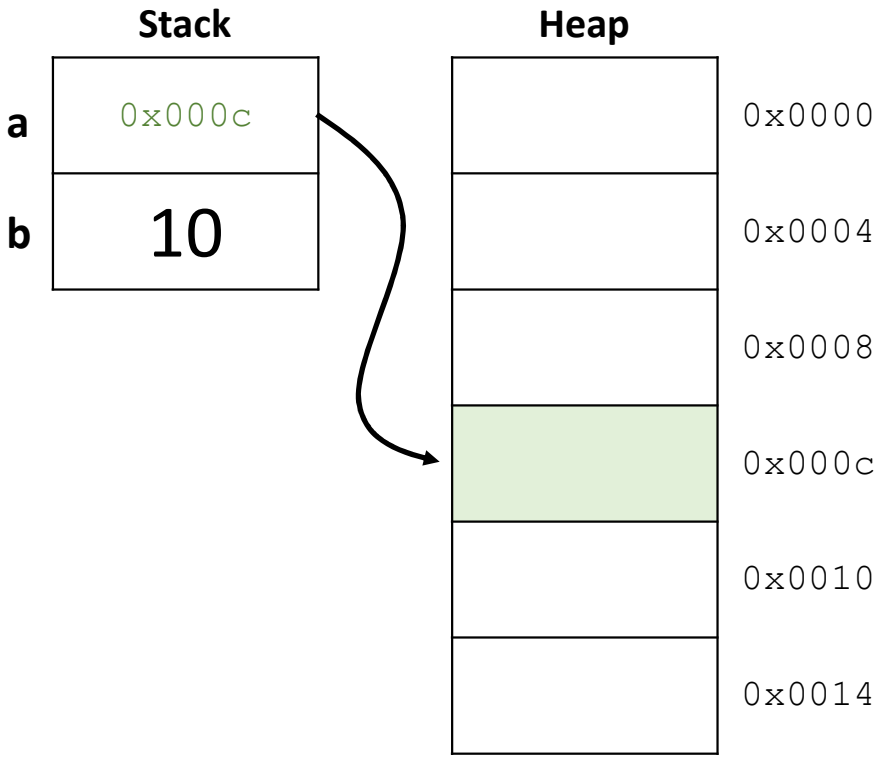
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



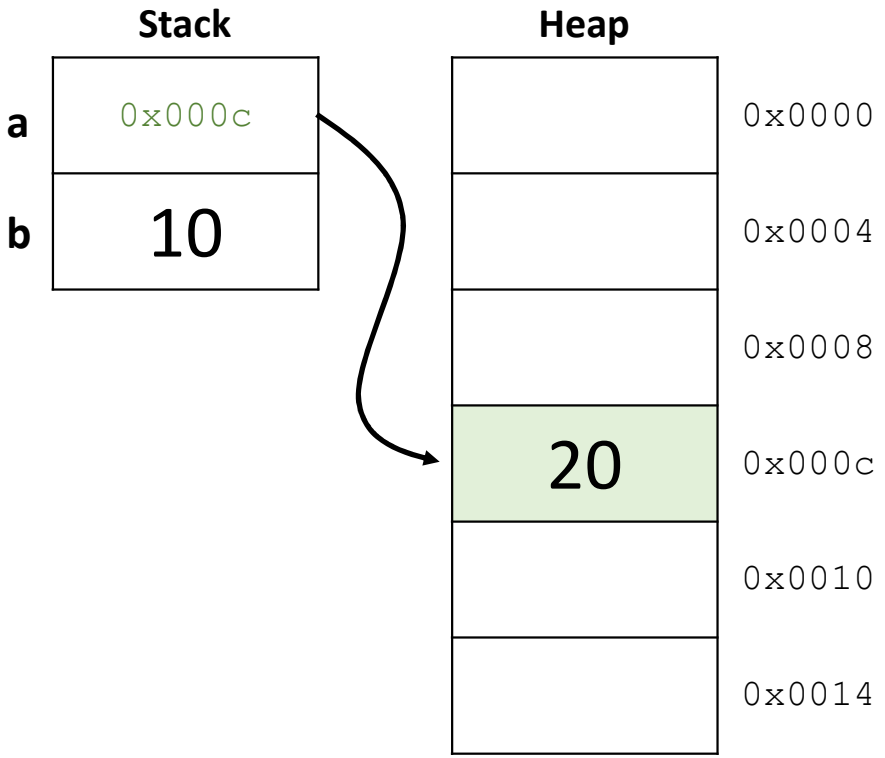
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



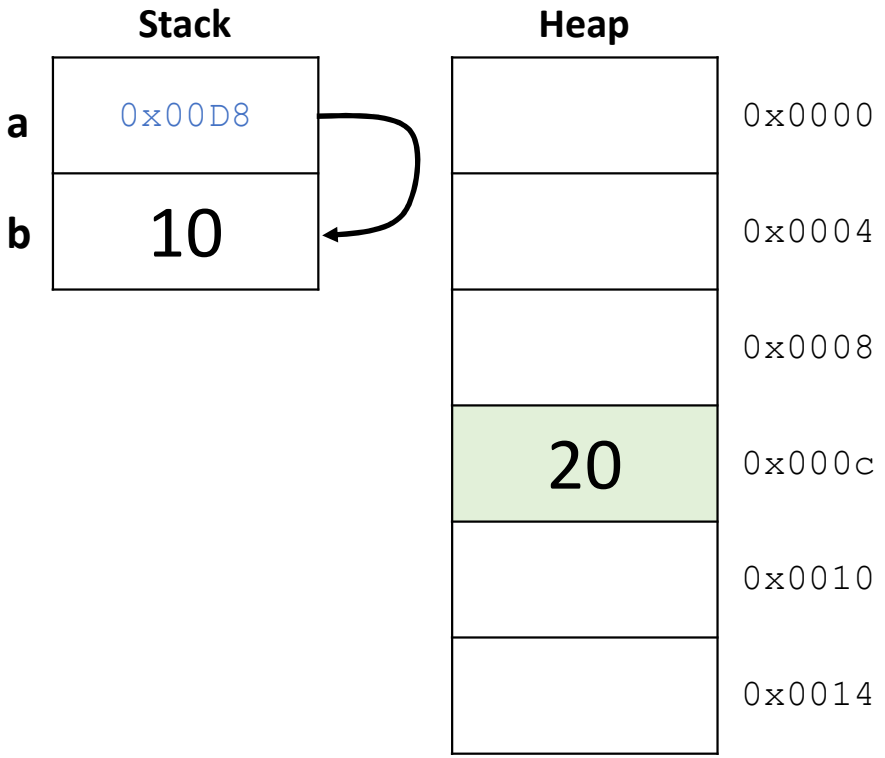
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



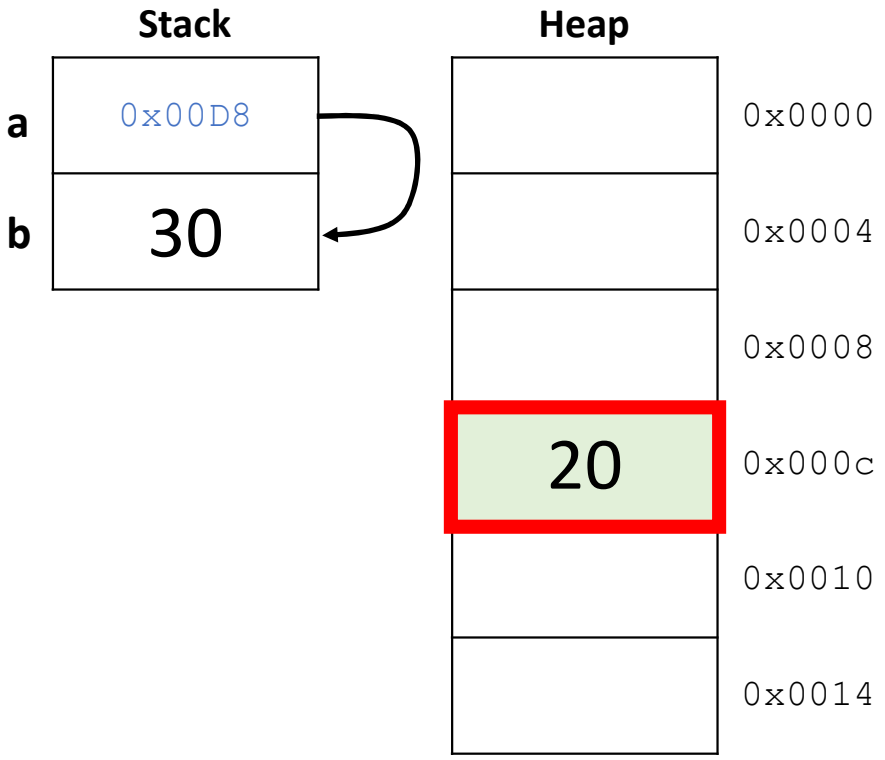
Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



Ponteiros
Exemplo 3

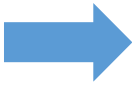
```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



Ponteiros
Exemplo 3

- Como melhorar o código?

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



```
int *a = nullptr;
int b = 10;

a = new int;
*a = 20;
delete a;
a = &b;
*a = 30;
```

Atenção para não confundir:
Aqui o ponteiro está sendo declarado (int *a)
E recebe o valor nulo. (a = nullptr)
não é equivalente a: *a = nullptr

Ponteiros nulos

- **`nullptr` (`NULL`)**
  - Constante simbólica (**`NULL = 0`**)
  - Semanticamente igual (**`nullptr`** é mais seguro)
  - Ponteiros não inicializados ou condições de erro
- Nenhum ponteiro válido possui esse valor!
- Esse valor não pode ser acessado
  - Falha de segmentação

Ponteiros nulos

Exemplo 4

```cpp
#include <iostream>

using namespace std;

int main() {

  int *ptr_a = nullptr;
  // ptr_a = new int;

  if (ptr_a == nullptr) {
      cout << "Memoria nao alocada!" << endl;
      exit(1);
  }

  cout << "Endereco de ptr_a: " << &ptr_a << endl;
  *ptr_a = 90;
  cout << "Conteudo de ptr_a: " << *ptr_a << endl;
  delete ptr_a;

  return 0;
}
```

Alocação dinâmica de vetores

- Criar vetores em tempo de execução
  - Só ocupar a memória quando necessário
- Ponteiro guarda o endereço da primeira posição do vetor

ATENÇÃO!

Os colchetes também devem ser usados na desalocação.

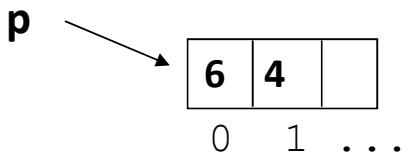
```
int main(){
   int *p = new int[10];

   p[0] = 99;

   delete[] p;

   return 0;
}
```

Código online

http://www.cplusplus.com/reference/new/operator%20delete[]/

Vetores e Apontadores

- Na verdade, em C/C++ todo vetor (mesmo alocados de forma estática) pode ser visto como um apontador.
- Pode se trabalhar usando ambas notações:
  - *p é equivalente a p[0]
  - p é equivalente a &p[0]
  - *(p + i) é equivalente a p[i]
  - considerando v um vetor alocado estaticamente, e p dinamicamente, pode-se fazer p = v, mas não v = p (v é, de certa forma, um "ponteiro constante")



Ponteiros para estruturas

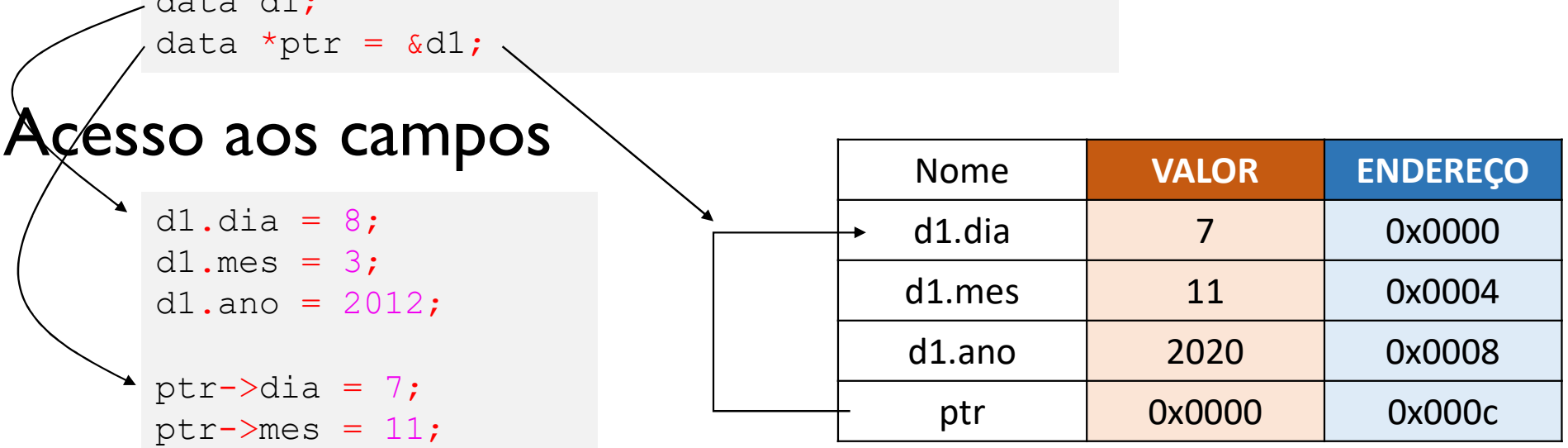
- Declaração e inicialização

```
struct data {int dia; int mes; int ano;};
data d1;
data *ptr = &d1;
```

- Acesso aos campos

```
d1.dia = 8;
d1.mes = 3;
d1.ano = 2012;

ptr->dia = 7;
ptr->mes = 11;
ptr->ano = 2020;
```

Código online



Nome	VALOR	ENDEREÇO
d1.dia	7	0x0000
d1.mes	11	0x0004
d1.ano	2020	0x0008
ptr	0x0000	0x000c

Passagem de parâmetros

- **Valor**
  - Parâmetro formal (recebido na função) é uma cópia do parâmetro real (passado na chamada)
  - Variáveis são totalmente independentes

- **Referência**
  - Parâmetro formal (recebido) é uma referência para o parâmetro real (passado)
  - Modificações refletem no parâmetro real

Passagem de parâmetros

Passagem de parâmetros
Exemplo 5 – Valor

```
#include <iostream>
using namespace std;

void addOneValue(int x) {
  x = x + 1;
}

int main() {
  int a = 0;
  cout << "Antes: " << a << endl;

  addOneValue(a);
  cout << "Depois: " << a << endl;

  return 0;
}
```

Código online

Passagem de parâmetros

Exemplo 6 – Referência

A passagem por referência usando **&** é mais segura!
Usar sempre que possível ao invés de *

```
#include <iostream>
using namespace std;

void addOneReference(int &x) {
  x = x + 1;
}

void addOnePointer(int *x) {
  *x = (*x) + 1;
}

int main() {
  int a = 0;
  cout << "Antes: " << a << endl;
  addOneReference(a);
  addOnePointer(&a);
  cout << "Depois: " << a << endl;
  return 0;
}
```

Essa referência NÃO pode ser nul

Esse ponteiro PODE ser null!

Código online

Passagem de parâmetros
Uso de **const**

- O que ocorre nesse código?
  - Passagem de uma estrutura grande por valor
  - Cópia! Custo desnecessário!
- Solução:
  - Passagem por referência
  - Risco: alteração do conteúdo
    - Solução: parâmetro como **const**

```
#include <iostream>
using namespace std;

const int TAM = 10;

struct Grande {int num; int dados[TAM];};

void ImprimeDados(Grande x) {
  cout << x.num << endl;
  for(int i=0; i<TAM; i++)
     cout << x.dados[i] << endl;
}
int main() {
  struct Grande a;

  a.num = 213;
  for(int i=0; i<TAM; i++)
    a.dados[i] = i;

  ImprimeDados(a);

  return 0;
}
```

Passagem de parâmetros
Uso de **const**

Qualquer alteração na variável declarada como *const* gera um erro de complilação!

```
#include <iostream>
using namespace std;

const int TAM = 10;

struct Grande {int num; int dados[TAM];};

void ImprimeDados(const Grande &x) {
  cout << x.num << endl;
  for(int i=0; i<TAM; i++)
     cout << x.dados[i] << endl;
}
int main() {
  struct Grande a;

  a.num = 213;
  for(int i=0; i<TAM; i++)
    a.dados[i] = i;

  ImprimeDados(a);

  return 0;
}
```

https://www.delftstack.com/howto/cpp/const-reference-vs-normal-parameter-passing-in-cpp/

Considerações finais

Erros comuns

- Tentar acessar o conteúdo de uma posição de memória sem essa ter sido alocada anteriormente
  - Ou após já ter sido desalocada
- Copiar o valor do ponteiro e não o valor da variável apontada
  - Endereço != Conteúdo
- Esquecer de desalocar memória
  - Escopo: desalocada ao fim do programa ou da função
  - Pode ser um problema em loops