UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Armazenamento e manipulação de dados em memória (A)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

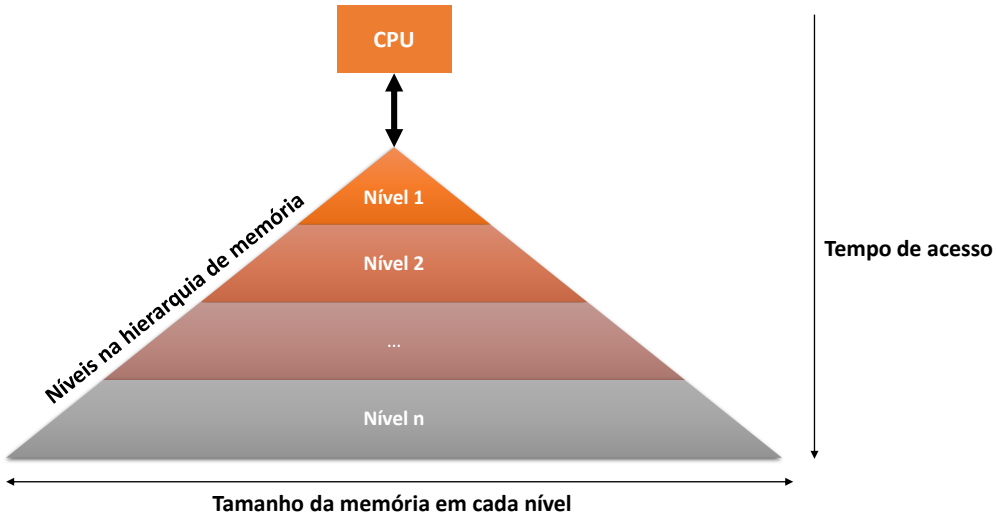
1

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

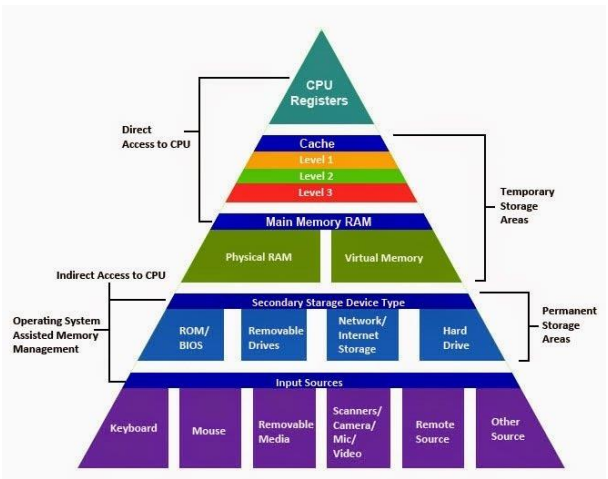
2

Hierarquia de memória



3

Hierarquia de memória



4

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)

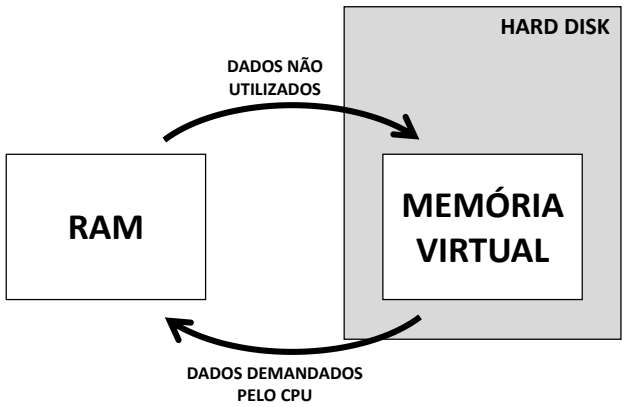
Hierarquia de memória
Princípio da localidade – Espacial

- 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

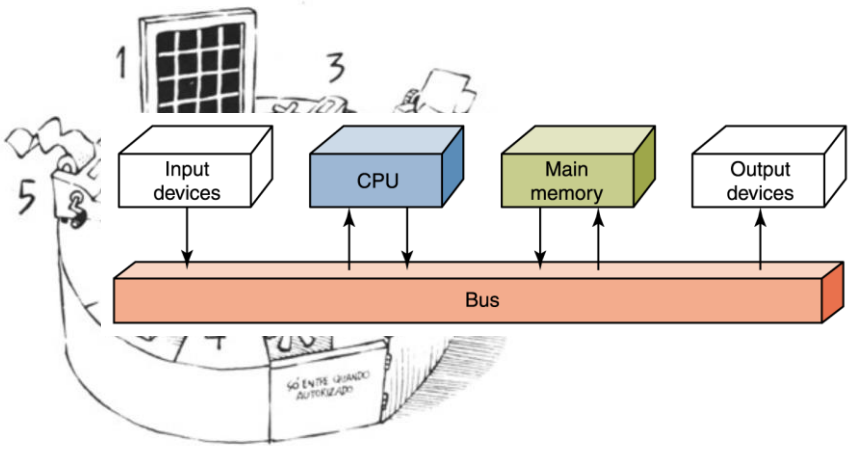
- Maior demanda da memória principal
  - Programas cada vezes maiores
  - Queda no custo não teve o mesmo ritmo
- Como resolver esse problema?
- Memória virtual
  - Memória (RAM) → Memória Secundária (HD)
  - Busca hierárquica pela informação

Hierarquia de memória
Memória virtual



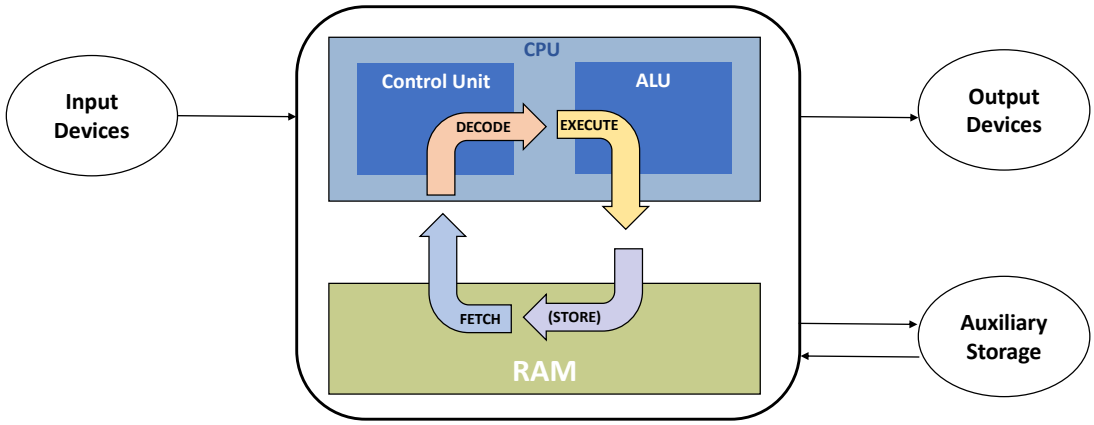
9

Computador simplificado



10

Fetch-Decode-Execute



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

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

11

UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Armazenamento e manipulação de dados em memória (B)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

12

Alocação de memória

- Segmentos da memória
  - **Código/Globais**
    - Guarda o código compilado do programa e outras variáveis
  - **Stack (pilha)**
    - Espaço que variáveis dentro de funções são alocadas
  - **Heap**
    - Espaço mais estável (durável) de armazenamento
    - Programa aloca/desaloca porções de memória do heap durante a execução

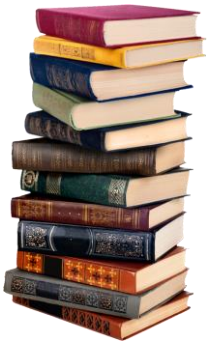
Alocação de memória
Stack (pilha)

- 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 limites no seu crescimento (linguagem)

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

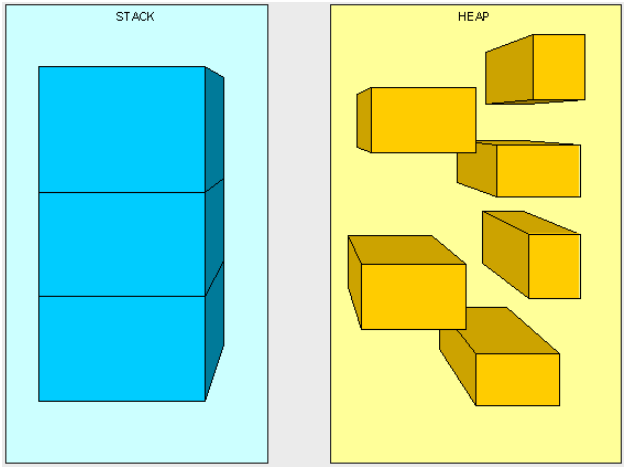
Alocação de memória



STACK

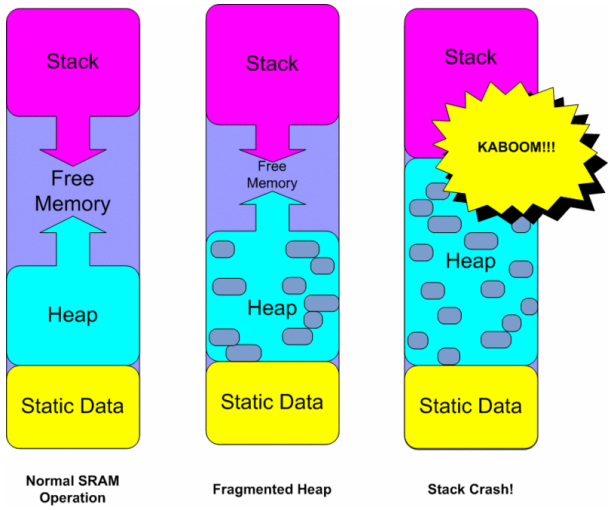
HEAP

Alocação de memória



17

Alocação de memória



18

Alocação de memória

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

- Estática
  - Acesso/manipulação sem alterar o endereço (variáveis globais)
  - Sabe-se o espaço de memória para as variáveis em *compile time*, ou seja, durante o desenvolvimento (e persiste até o final)

Alocação de memória
Exemplo 1

```
#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;
}
```

Variável global (stack).

Variáveis locais (stack).

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 cujo tamanho máximo é desconhecido na implementação (não fixo)
    - Sob demanda durante a execução (apenas quando necessário)
  - 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

- Ponteiros
  - Armazenam um endereço de memória
  - Variáveis alocadas dinamicamente (Stack → Heap)
- **Referência**
  - &x
  - Endereço de memória da variável x
- **Deferência** (dereferência)
  - *x
  - Conteúdo do endereço apontado por x

Ponteiros

```
int main() {
  int var = 100;
  int *p = &var;

  return 0;
}
```

Conteúdo

Endereço

MEMÓRIA

Nome	VALOR	ENDEREÇO
		0x0000
var	100	0x0004
p	0x0004	0x0008
		0x000c
		0x0010

Ponteiros

```
int main() {
  int i = 10;
  int *ponteiro = &i;
  int **ppp = &ponteiro;

  return 0;
}
```

Código online

Ponteiro para um **Ponteiro** de **Inteiro**!

MEMÓRIA

Nome	VALOR	ENDEREÇO
		0x0000
i	10	0x0004
ponteiro	0x0004	0x0008
ppp	0x0008	0x000c
		0x0010

. . .

UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Armazenamento e manipulação de dados em memória (C)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Ponteiros
Operadores

C	C++
• Alocação: **malloc** • Liberação: **free**	• Alocação: **new** • Liberação: **delete**

Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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

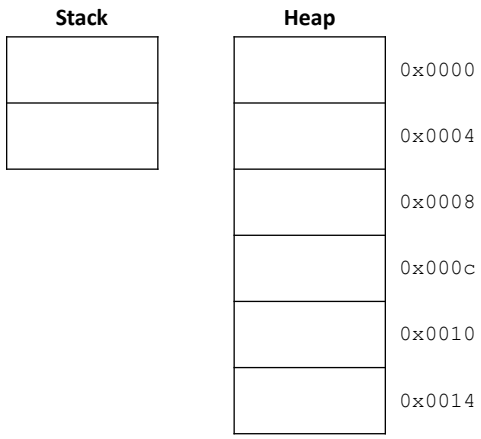
- Onde cada uma dessas variáveis será alocada?
- Qual o valor de 'a' ao final?
- Existe algum problema com esse código?

Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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

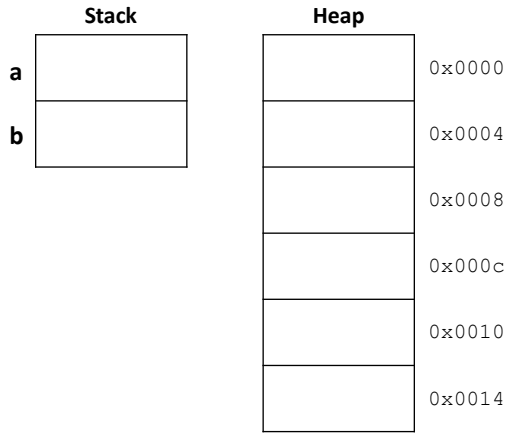


Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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

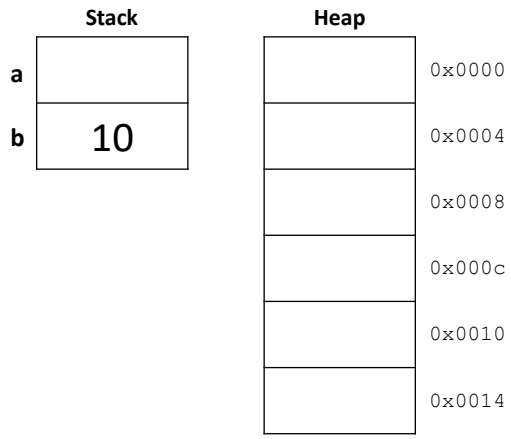


Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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

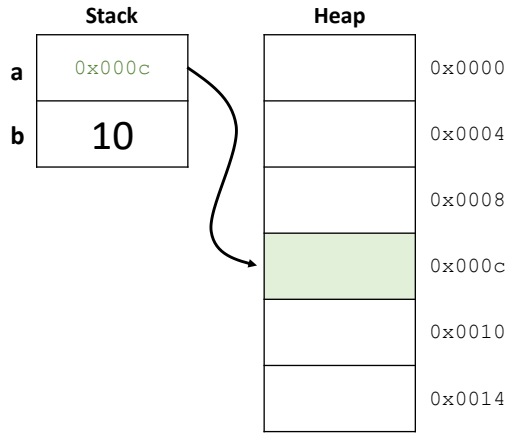


Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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

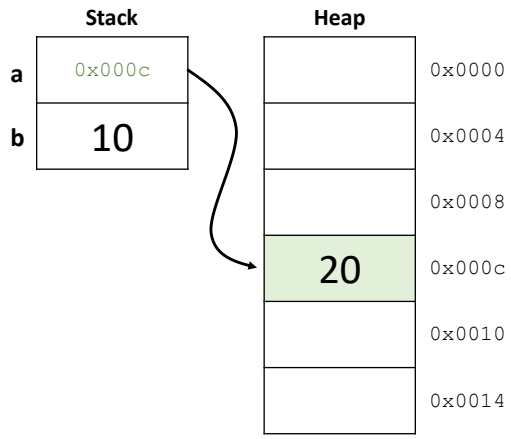


Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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



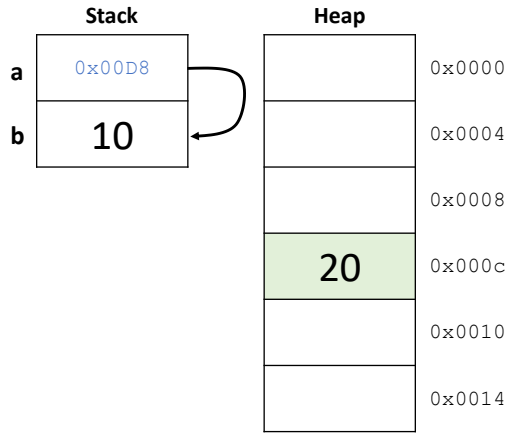
33

Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

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



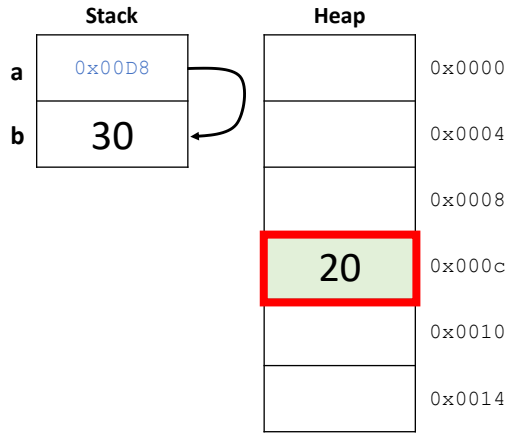
34

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;
```

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[]/

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

```
#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;
}
```

Código online

Ponteiros para `void`

- Não existe um objeto do tipo **`void`**
- Valor utilizado com um coringa
  - Pode apontar para qualquer tipo de variável

```
int i = 10;
int *int_ptr;
void *void_ptr;
double *double_ptr;

int_ptr = &i;
void_ptr = int_ptr;        // OK
double_ptr = int_ptr;      // !OK
double_ptr = void_ptr;     // OK
```

Nome	VALOR	ENDEREÇO
i	10	0x0000
int_ptr	0x0000	0x0004
void_ptr	0x0000	0x0008
double_prt	0x0000	0x000c

Ponteiros para estruturas

- Declaração e inicialização

```
struct data {int dia; int mes; int ano;};
struct data d1;
struct data *ptr = &d1;
int i = 0;
```

- 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
i	0	0x0010

Passagem de parâmetros

- **Valor**
  - Parâmetro formal (recebido na função) é uma cópia do parâmetro real (passado na chamada)
  - Variáveis são totalmente independentes
- **Referência (ponteiro)**
  - Parâmetro formal (recebido) é uma referência para o parâmetro real (passado)
  - Modificações refletem no parâmetro real

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

```
#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 null!

Esse ponteiro PODE ser null!

Código online

Passagem de parâmetros

Passagem de parâmetros
Boas práticas

```
void function(int &i) {
  int j = 10;
  i = &j;
}
```

- Erro de compilação!

```
void function(int *i) {
  int j = 10;
  i = &j;
}
```

- Compila, mas ótima fonte de bugs!

- Use referências (&) sempre que possível
- Use ponteiros (*) quando for necessário

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