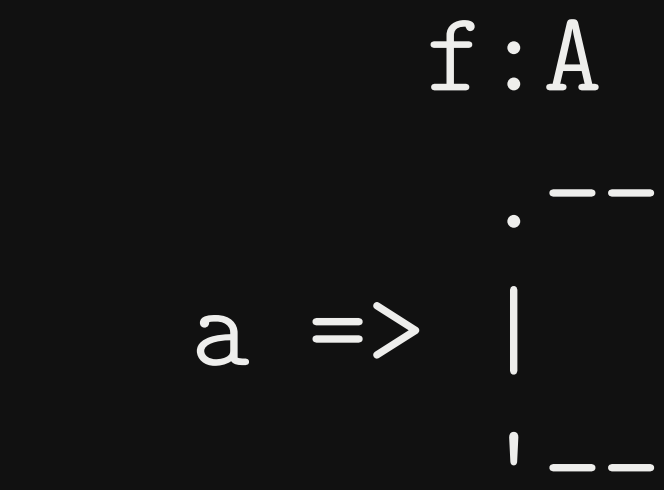
>>> Programação Orientada a Objetos (POO)

... Funções

Prof: André de Freitas Smaira

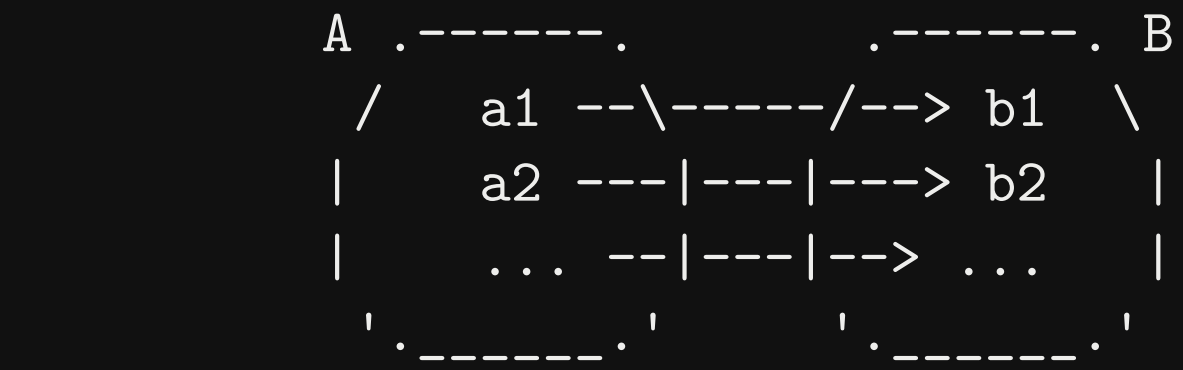
[~]$ _



Funções em programação

>>> As funções de forma geral

* Funções pegam elementos de um conjunto **A** (domínio) e transformam em elementos de um conjunto **B** (contra-domínio) de acordo com algum tipo de regra **f**.
* Um mesmo elemento de A não pode ser levado dois elementos de B.



>>> As funções na computação

* **Reutilização** de código
* Prevenção de **redundâncias**
* **Alteração** rápida
* Simplificação de **leitura**

>>> Funções em C

```
tipo nome(tipo1 arg1, tipo2 arg2, tipo3 arg3, ...)
{
    // Bloco de comandos
    return valor;
}
//Exemplo:
// Função que eleva um número b a um número x inteiro
double power(double b, int x)
{
     double  a = 1.0;
     for (int i=0; i<x; i++)
          a *= b;
     return a;
}
```

>>> O que acontece na real

* Memória
  * **Stack** (Ou Pilha): **Organizada**. Chamadas de **função**, **variáveis** declaradas estaticamente, etc. CPU gerencia
  * **Heap** (ou Monte): **Bagunçada**. **Alocação dinâmica** (**new**). Programador (ou programa) gerencia.
* Ao chamarmos **funções**: o ponto de **retorno**, suas **variáveis locais** e os **parâmetros** são colocados na **pilha**

>>> Funções em C

```c
double power(double b, int x)
{
     double  a = 1.0;
     for (int i=0; i<x; i++)
          a *= b;
     return a;
}
```

```
   |        |
   |        |
   |        |
   |        |
   |        |
   |main()  | -> Chamada da função main
```

```
double power(double b, int x)
{
     double  a = 1.0;
     for (int i=0; i<x; i++)
          a *= b;
     return a;
}
```

```
   |        |
   |        |
   |        |
   |        |
   |power() | -> Chamada da função power "Ponto de retorno"
   |main()  |
```

```
double power(double b, int x)
{
     double  a = 1.0;
     for (int i=0; i<x; i++)
          a *= b;
     return a;
}
```

```
  |int i    | \
  |double a|  |_ Variáveis locais da função são
  |int x    |  |        jogadas na pilha
  |double b| /
  |power()  |
  |main()   |
```

```
double power(double b, int x)
{
     double  a = 1.0;
     for (int i=0; i<x; i++)
          a *= b;
     return a;
}
```

```
  |int i    | \
  |double a|  |_ Terminada a execução de power() as variáveis
  |int x    |  |             são retiradas da pilha
  |double b| /
  |power()  |
  |main()   |
```

>>> Tipos de dados em funções

* tipo de funções: os mesmos de variáveis
* Tipo especial: void (vazio) - A função não retornará nada

Ex:

```
void nada()
{
    printf("Fiz nada");
}
```

>>> Protótipos

* Ao menos **prototipada** antes de usada (Note o ponto-e-vírgula)
  Ex:

```
float prototype(float a, float b);
...
 prototype(1.0, 2.0);
```

* Os arquivos .h que incluimos **são protótipos e constantes**

>>> Passagem de argumentos

* Passar **vetores**, **matrizes**, etc = passar o **endereço**
* Outras variáveis = copia o valor
* Evitar cópia -> **ponteiro** (é possível alterar o valor)

>>> Passagem de argumentos

```c
int soma(int a, int b)
{
      // São criadas cópias locais de a e b
      return a+b; // É retornada a soma
}

void swap(int *a, int *b)
{
    // São criadas cópias para os ponteiros
    // No entanto ao acessar a memória usando o *
    // estamos alterando diretamente as variáveis
    // Apontadas por a e b
    if (*a == *b) return;
    int aux = *b;
    *b = *a;
    *a = aux;
}
```

>>> Passagem de argumentos - Vetores

```cpp
// Pega um vetor e mostra na tela
#include <iostream>
int printvec(int p[], int size)
{
    for (int i = 0; i < size; i++)
        std::cout << p[i] << std::endl;
}
int printvec2(int *p, int size)
{
    for (int i = 0; i < size; i++)
        std::cout << p[i] << std::endl;
}
int main()
{
    int p[4] = {2, 4, 6, 8};
    printvec(p, 4);
    printvec2(p, 4);
}
```

>>> Passagem de argumentos - matrizes I

```cpp
// Pega uma matriz (por referência) alocada na stack
#include <iostream>
const int n = 10;
const int m = 10;
void printmat(int M[n][m])
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            std::cout << M[i][j] << " ";
        std::cout << std::endl;
}

// Pega um ponteiro para um ponteiro para inteiro
// A matriz deve ser alocada dinamicamente neste caso
void printmat2(int **M)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            std::cout << M[i][j] << " ";
        std::cout << std::endl;
}
```

>>> Passagem de argumentos - matrizes I

```
int main()
{
    int M[n][m], **M2;
    // Aloca a matriz
    M2 = new int*[n];
    for (int i = 0; i < n; ++i)
        M2[i] = new int[m];
    // Gera a matriz identidade
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            M[i][j] = M2[i][j] = i == j;
    printmat(M);
    std::cout << std::endl;
    printmat2(M2);
    // Libera a memória da heap: Super importante!
    for (int i = 0; i < n; ++i) delete[] M2[i];
    delete[] M2;
    return 0;
}
```

>>> Escopos

* **Variáveis locais**: Definidas **dentro da função**; acessadas **apenas por essa função**. Podem ser:
  * **Automáticas**: uma nova criada **para cada execução** da função.
  * **Estáticas**: criadas apenas uma vez; **mantém valor entre execuções**
* **Variáveis globais**: são definidas **fora de qualquer função**; podem ser acessadas por **qualquer função**

```c
int x;  // Global

void f() {
    int x;  // Automática
    static int y = 0; // Estática
}
```

>>> Umas coisinahs a mais...

Função **inline**: Funções **pequenas** e **muito usadas**. Serão inseridas no código pelo compilador.

```cpp
#include <iostream>

inline int soma(int x, int y) {
    return x+y;
}

int main() {
    std::cout << soma(1,2) << std::endl;
    return 0;
}
```

>>> Mas não tem nada de novo em C++?

Parâmetros por referência: se alterada, a variável é alterada fora da função

```
#include <iostream>
int g(int &a) {
    a++;
    return a;
}

int main() {
    int m, n;
    n = 5;
    m = g(n);
    std::cout << m << " " << n << std::endl;
    return 0;
}
```

>>> Mas não tem nada de novo em C++?

Retorno de referência: o retorno funciona como uma variável

```
#include <iostream>
int& element(int v[], int i) {
    return v[i];
}
int main() {
    int a[10];
    for (int i = 0; i < 10; i++)
        a[i] = i;
    std::cout << element(a,3);  // Imprime: 3
    element(a,3) = 9;
    std::cout << a[3];          // Imprime: 9
}
```

>>> Mas não tem nada de novo em C++?

Valor padrão: o parâmetro assume um valor padrão se nada for fornecido

```
#include <iostream>

void func(int x = 10) {
    std::cout << "x = " << x << std::endl;
}

int main() {
    func();      // Usa o valor padrão (10)
    func(20);    // Usa o valor fornecido (20)
    return 0;
}
```

>>> Mas não tem nada de novo em C++?

Sobrecarga de nomes: funções distintas podem ter o mesmo nome, desde que tenham tipos de parâmetros distintos

```
int square(int const a) {
    return a*a;
}
float square(float const a) {
    return a*a;
}
double square(double const a) {
    return a*a;
}
int i, j; float x, y; double d;
i = square(j);
x = square(y);
d = square(2.0);
y = square(1.0);
x = square(3);
```

>>> Mas não tem nada de novo em C++?

Funções lambda: funções locais

```
int v[]{1,2,3};
int const n = 10;
int s = 0;
auto f = [&s] (int x) { s += x; };
auto g = [n] (int &x) { x += n; };
for (int i = 0; i < 3; i++)
f(v[i]);
std::cout << s << std::endl;
for (int i = 0; i < 3; i++) {
    g(v[i]);
    std::cout << v[i] << " ";
}
std::cout << std::endl;
```

Para entender recursão...
primeiro você deve entender a
recursão!

>>> O que é a recursão e funções recursivas

* Pesquise **recursão** no google.
* função **recursiva** -> chama a si mesma.
* **Algoritmo recursivo**: O problema é formado por **um ou mais dele mesmo** em menor escala
* Se o problema é pequeno, é resolvido de imediato (**Condição de parada**)
* Caso contrário reduza o problema, resolva o problema menor e retorne para o problema maior (**Recursão**)

>>> Etapas da recursão

1. Definir recursivamente o problema
2. Definir condições de parada
3. Garantir que se aproxime do fim

NUNCA PENSE EM COMO O COMPUTADOR ESTÁ TRABALHANDO

* Soluções **mais concisas**, mas **menos eficientes**
* As chamadas de função vão sendo **empilhadas na stack**
* Caso a **condição de parada não funcione** o programa pode estourar a stack: **Stack Overflow** (Também é o nome de um famoso Forum na internet sobre programação).

>>> Exemplos: Fatorial Iterativo

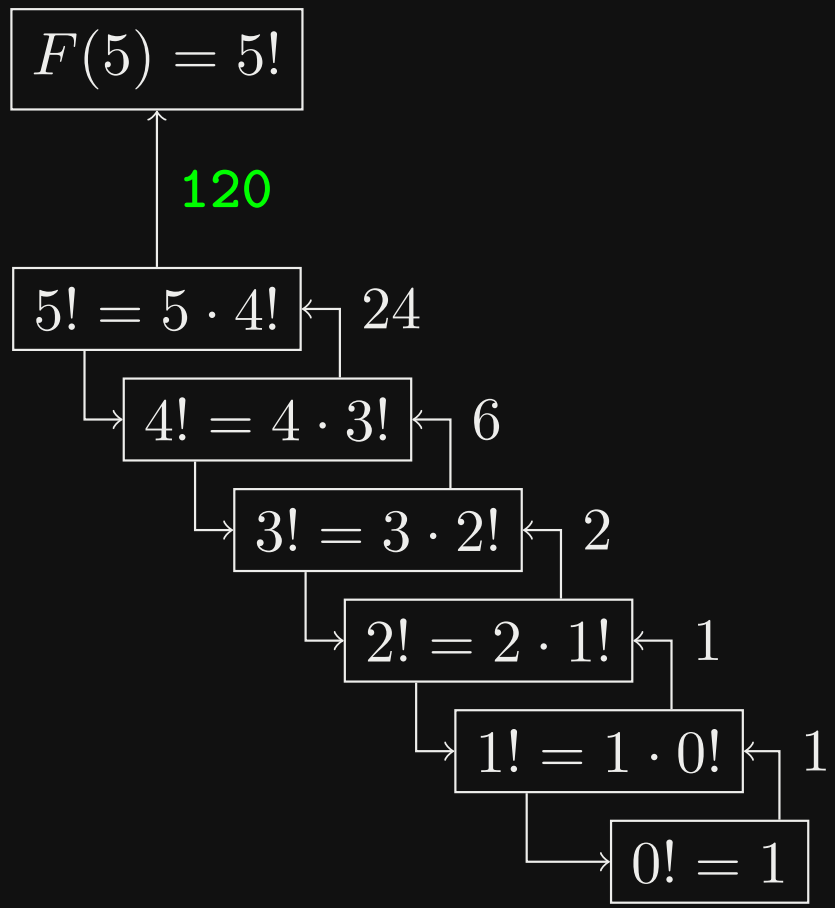
$$F(n) = n! = n \cdot (n-1) \cdots 2 \cdot 1$$

```
typedef unsigned long long int llu;

llu fat_it(int n)
{
    llu f=1;//Valor inicial
    for(int i=2; i<=n; i++)//Processo iterativo
        f *= i;
    return f;
}
```

$$F(n) = n \cdot (n-1) \cdots 2 \cdot 1 = n \cdot [(n-1) \cdots 2 \cdot 1] = n \cdot F(n-1)$$

$$F(0) = 1$$



>>> Exemplos: Fatorial Recursivo

$$F(n) = n \cdot (n-1) \cdots 2 \cdot 1 = n \cdot [(n-1) \cdots 2 \cdot 1] = n \cdot F(n-1)$$

$$F(0) = 1$$

```
typedef unsigned long long int llu;

llu fat_rec(int n)
{
    if(n==0) return 1;//Condicao de parada
    return (llu)n*fat_rec(n-1);//Recursao
}
```

>>> Exemplos: Fibonacci

$$F(1) = F(0) = 1$$

$$F(n > 1) = F(n - 1) + F(n - 2)$$

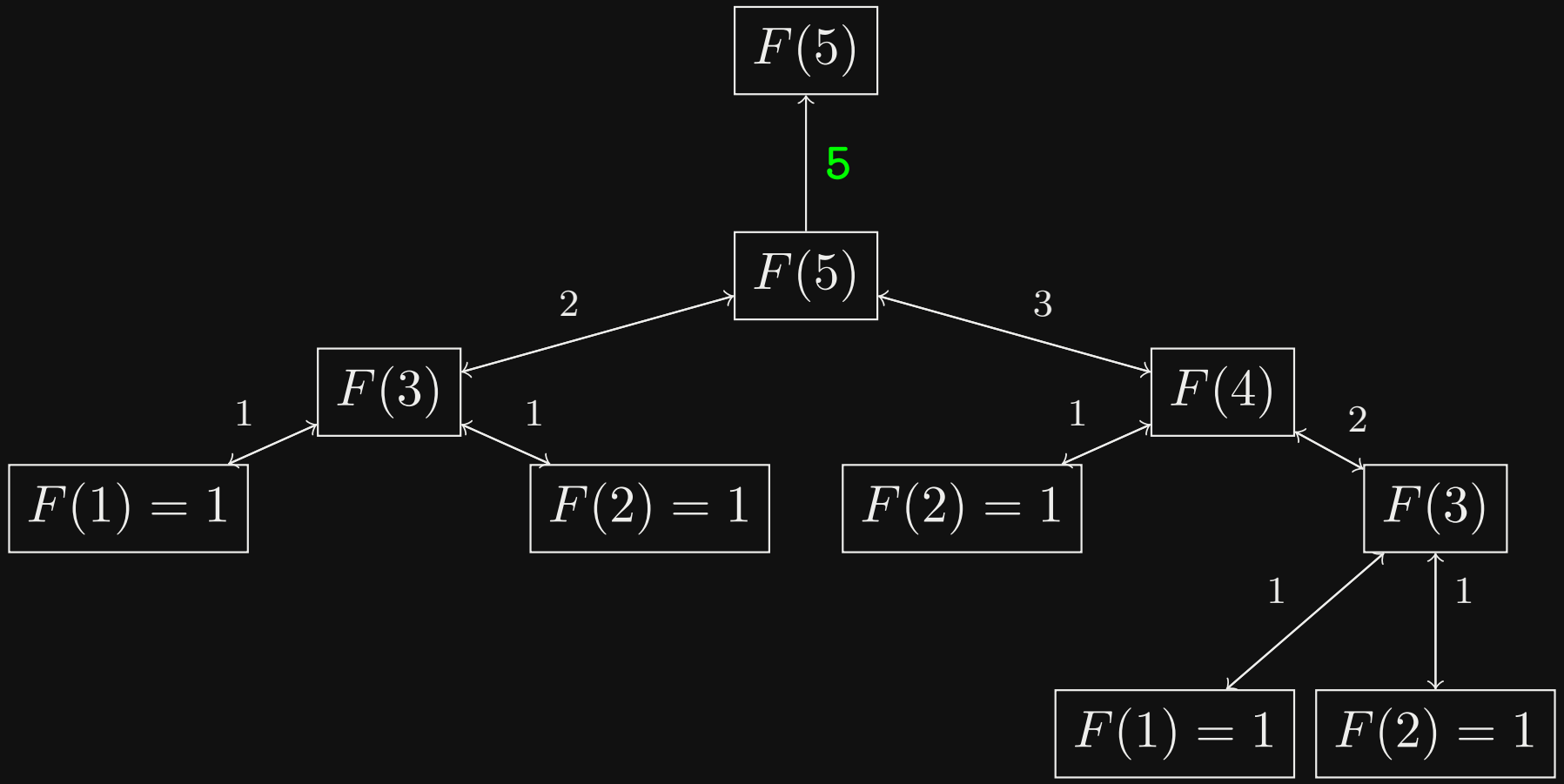
>>> Exemplos: Fibonacci Iterativo

```cpp
typedef unsigned long long int llu;

//(30,0m0.008s),(40,0m0.014s),(50,0m0.015s)
llu fib_it(int n)
{
    llu a=1,b=1;//Valores iniciais
    for(int i=2; i<=n; i++)//Processo iterativo
    {
        llu aux = a;
        a += b;
        b = aux;
    }
    return a;
}
```

>>> Recursão

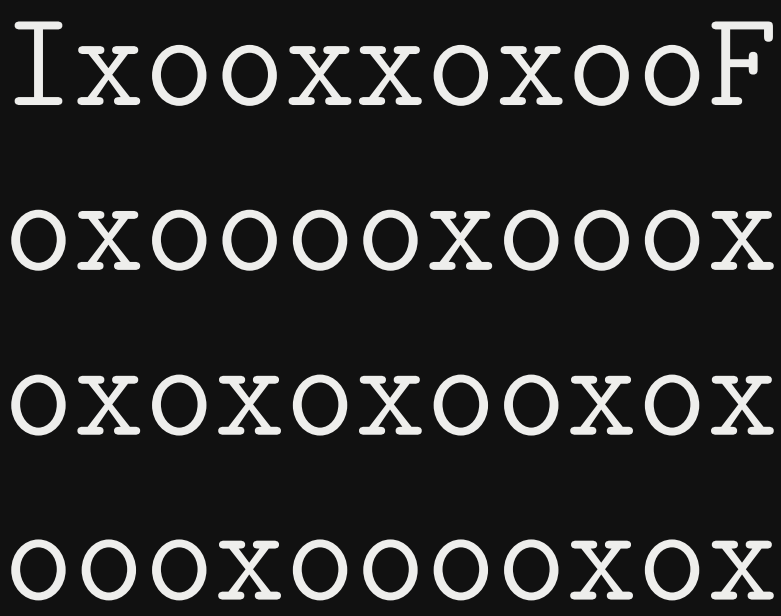
Fibonacci: $F(n) = F(n-1) + F(n-2)$



>>> Exemplos: Fibonacci Recursivo

```
typedef unsigned long long int llu;

//IT:  (30,0m0.008s),(40,0m0.014s),(50,0m0.015s)
//REC: (30,0m0.022s),(40,0m0.883s),(50,1m38.047s)
llu fib_rec(int n)
{
    if(n<2) return 1;//Condicao de parada
    return fib_rec(n-1)+fib_rec(n-2);//Recursao
}
```


IxooxxoxooF
oxoooooxooox
oxoxoxooxox
oooxooooxox

>>> Exemplos: Exploração Iterativo

```cpp
#include<iostream>
#define N 11
void exp_it(int i0, int j0, int iF, int jF)
{
    int pilha[1000],k=0;
    pilha[k++] = N*i0+j0;
    while(k>0)
    {
        int aux = pilha[k--];
        int i = aux/N, j = aux%N;
        if(i==iF && j==jF)
        {
            std::cout << "CHEGAMOS!" << std::endl;
            return;
        }
        if(tab[i][j] == 'x') continue;
        tab[i][j] = 'x';
        if(i-1>=0 && tab[i-1][j]=='o') pilha[k++] = N*(i-1)+j;
        if(j-1>=0 && tab[i][j-1]=='o') pilha[k++] = N*i+j-1;
        if(i+1<N && tab[i+1][j]=='o') pilha[k++] = N*(i+1)+j;
        if(j+1<N && tab[i][j+1]=='o') pilha[k++] = N*i+j+1;
    }
}
```

>>> Exemplos: Exploração Recursivo

```cpp
#include<iostream>
#define N 11
void exp_rec(int i, int j, int iF, int jF)
{
    if(i<0 || j<0 || i>=N || j>=N) return;//Condicao de parada
    if(tab[i][j]=='x')  return;//Condicao de parada
    tab[i][j] = 'x';    //Evita loop infinito
    if(i==iF && j==jF)  //Condicao de parada
    {
        std::cout << "CHEGAMOS!\n";
        return;
    }
    exp_rec(i-1,j);    //Recursao
    exp_rec(i+1,j);
    exp_rec(i,j-1);
    exp_rec(i,j+1);
}
```

>>> Referências e Leitura Recomendada I

* Aulas do **Grupo Maratona IFSC** (Ian Giestas Pauli e eu)
* Apostila e Aulas do **Gonzalo Travieso** (IFSC/USP)
* Tensores (Generalização de matrizes e vetores)
  https://pt.wikipedia.org/wiki/Tensor
* Como passar matrizes como parâmetro
  https://www.geeksforgeeks.org/pass-2d-array-parameter-c/
* "Pontas soltas"(dangling pointers)
  https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf