Algoritmos e Programação I

Módulo 1 - Revisão de Fundamentos de Programação

Profª. Elisa de Cássia Silva Rodrigues

Sumário

Introdução

- Etapas para desenvolvimento de um programa:
  - **Análise:**
    - ★ Estudar o enunciado do problema.
    - ★ Definir dados de entrada.
    - ★ Definir o processamento.
    - ★ Definir os dados de saída.

  - **Algoritmo:**
    - ★ Sequência de instruções que realizam uma tarefa específica.
    - ★ Instruções simples, objetivas e não ambíguas.
    - ★ Note que um problema pode ser resolvido por vários algoritmos.
  - **Codificação:**
    - ★ Transformar um algoritmo em códigos de uma linguagem de programação.

Algoritmos

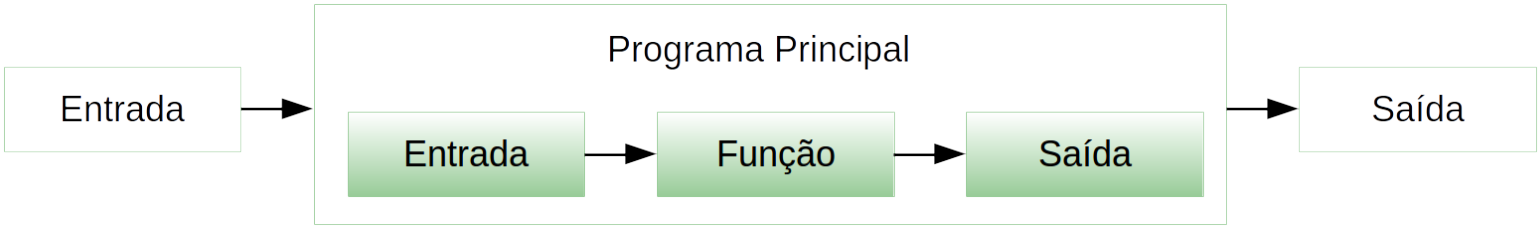
- O que é necessário para construir um algoritmo?
  - Compreender o problema.
  - Destacar pontos importantes e objetos que o compõem.
  - Definir os dados de entrada, processamento e saída.
  - Escolher um tipo de algoritmo.
  - Construir o algoritmo.
  - Testar o algoritmo realizando simulações.
- Tipos de algoritmos
  - Descrição narrativa: utiliza linguagem natural.
  - Fluxograma: utiliza símbolos gráficos predefinidos.
  - Pseudocódigo: utiliza regras predefinidas.

Algoritmos

- **Problema:** Somar dois números.
  - Dados de entrada: primeiro número (n1) e segundo número (n2)
  - Processamento: somar os números ($s = n1 + n2$).
  - Dados de saída: resultado da soma (s).
- **Exemplo:**
  - Variável $n1 = 5$
  - Variável $n2 = 4$
  - Variável $s = n1 + n2 = 9$

Programação Estruturada

- Um problema pode ser dividido em problemas menores mais fáceis de resolver (funções).



- Todo processamento pode ser realizado através de estruturas:
  - Sequenciais.
  - Condicionais.
  - Repetição.
- Exemplo de linguagem de programação estruturada:
  - Linguagem C.

Estruturas Sequenciais (Linguagem C)

- Declaração de variáveis:

Algoritmo

```
DECLARE x NUMÉRICO
```

Programa em C

```
int x;
float x;
double x;
```

- Comando de atribuição:

Algoritmo

```
x ← 1
```

Programa em C

```
x = 1;
```

Estruturas Sequenciais (Linguagem C)

- Leitura de dados (para variáveis inteiras):

Algoritmo

```
LEIA x
LEIA x, y, z
```

Programa em C

```
scanf ("%d", &x);
scanf ("%d %d %d", &x, &y, &z);
```

- Escrita de dados (para variáveis inteiras):

Algoritmo

```
ESCREVA "Mensagem"
ESCREVA x
ESCREVA "Valor = ", x
```

Programa em C

```
printf("Mensagem");
printf("%d", x);
printf("Valor = %d", x);
```

Estruturas Condicionais (Linguagem C)

SE

```
if (condição)
{
    comando;
    ⋮
    comando;
}
```

SE ... SENÃO

```
if (condição)
{
    comandos;
}
else
{
    comandos;
}
```

Estruturas Condicionais (Linguagem C)

SE ... SENÃO SE

```
if (variável == valor1)
{
    comandos;
}
else if (variável == valor2)
{
    comandos;
else
{
    comandos;
}
```

ESCOLHA ... CASO

```
switch (variável)
{
    case valor1:
        comandos;
        break;

    case valor2:
        comandos;
        break;

    default:
        comandos;
}
```

Estruturas de Repetição (Linguagem C)

ENQUANTO

```
while (condição)
{
    comando;
    ⋮
    comando;
}
```

FAÇA ... ENQUANTO

```
do
{
    comando;
    ⋮
    comando;
} while (condição);
```

PARA

```
for (inicialização; condição; passo)
{
    comando;
    ⋮
    comando;
}
```

Exemplos (Linguagem C)

Programa para imprimir os números pares inteiros de 1 a n.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x, n;
    scanf ("%d", &n);
    x = 1;
    while(x <= n) {
        if((x % 2) == 0) {
            printf ("%d ", x);
        }
        x = x + 1;
    }
    return 0;
}
```

Exemplos (Linguagem C)

Programa para imprimir os números pares inteiros de 1 a n.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x, n;
    scanf ("%d", &n);
    for(x = 1; x <= n; x++)
    {
        if((x % 2) == 0)
        {
            printf ("%d ", x);
        }
    }
    return 0;
}
```

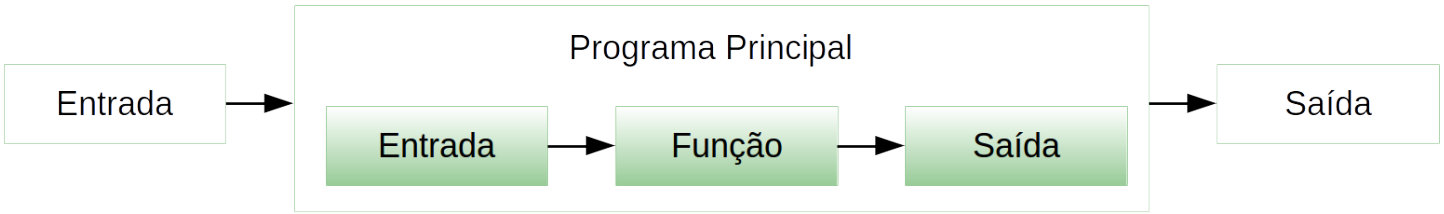
Exemplos (Linguagem C)

Programa para imprimir os números pares inteiros de 1 a n.

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x, n;
    scanf ("%d", &n);
    for(x = 2; x <= n; x+=2)
    {
        printf ("%d ", x);
    }
    return 0;
}
```

Modularização

- Blocos de instruções que realizam tarefas específicas podendo ser executados quantas vezes for necessário.
- Um problema pode ser subdividido em pequenas tarefas.



- Os programas tendem a ficar menores e mais organizados.
- O uso de funções permite a realização de desvios na execução do código (quando a função é chamada).

Modularização

- Uma função pode retornar valores ou não. Exemplo:
  - `void soma(); // sem retorno`
  - `int soma(); // retorna valor inteiro`
- Uma função pode receber parâmetros ou não. Exemplo:
  - `void soma(); // sem parâmetros`
  - `void soma(int x, int y); // parâmetros do tipo inteiro`

Exemplo de Função

Programa para imprimir a soma de dois números inteiros.

```c
#include <stdio.h>
#include <stdlib.h>
int main() { // programa principal
    int x, y;
    scanf ("%d %d", &x, &y);
    soma(x,y); // chamada da função soma
    return 0;
}
void soma(int x, int y) { // função soma
    int s; // variável local
    s = x + y;
    printf ("%d", s);
}
```

Modularização

- Variáveis globais:
  - São declaradas fora do escopo das funções.
  - O valor de uma variável global pode ser utilizado por qualquer função do programa.
  - ***Desvantagem:*** *torna difícil a manuntenção do programa.*
- Variáveis locais:
  - São declaradas dentro do escopo de uma determinada função.
  - Para outra função utilizar o valor de uma variável local, é necessário passar esse valor por parâmetro para a função em questão.
  - Se a função for apenas utilizar o valor recebido, sem alterá-lo, temos:
    - ★ Passagem de parâmetros por valor.
  - Se a função for alterar o valor recebido, temos:
    - ★ Passagem de parâmetros por referência.

Modularização

- Exemplo de passagem de parâmetros por valor:
  - As variáveis `x` e `y` não podem ser alteradas dentro da função `soma`.
  - **Chamada da função:** `s = soma(x, y);`
  - **Protótipo da função:** `int soma(int a, int b);`
- Exemplo de passagem de parâmetros por referência:
  - A variável `s` pode ser alterada dentro da função `soma`.
  - **Chamada da função:** `soma(&s, x, y);`
  - **Protótipo da função:** `void soma(int *s, int a, int b);`

Ponteiros

- Uma variável refere-se a uma posição de memória que armazena um dado de determinado tipo, por exemplo, um número inteiro.
- Cada posição de memória possui um endereço.
- Algumas linguagens de programação, como a Linguagem C, permitem a manipulação direta da memória através de variáveis que armazenam endereços de memória, chamadas de ponteiros.
- Ponteiro também possui um tipo que deve ser igual ao tipo de dado armazenado na posição de memória para a qual o ponteiro aponta.
  - **Exemplo:**

```
int a; // armazena um dado do tipo inteiro
int *p; // armazena um endereço da memória que contém um inteiro
```

- Para atribuir valores ao ponteiro p, existem duas formas:
  - Operador unário & ("endereço de"): p = &a;
  - Operador unário * ("conteúdo de"): *p = 10;

Alocação de Memória

- Alocação de memória é o processo de reserva de memória para armazenamento de dados durante a execução de um programa.
- A quantidade de memória pode ser reservada automaticamente como acontece quando declaramos uma variável ou um vetor estático:
  - **Exemplo:**

```
int x; // são reservados 4 bytes de memória para armazenar um inteiro.
int V[10]; // são reservados (10*4) bytes de memória para armazenar 10
números inteiros sequencialmente na memória.
```

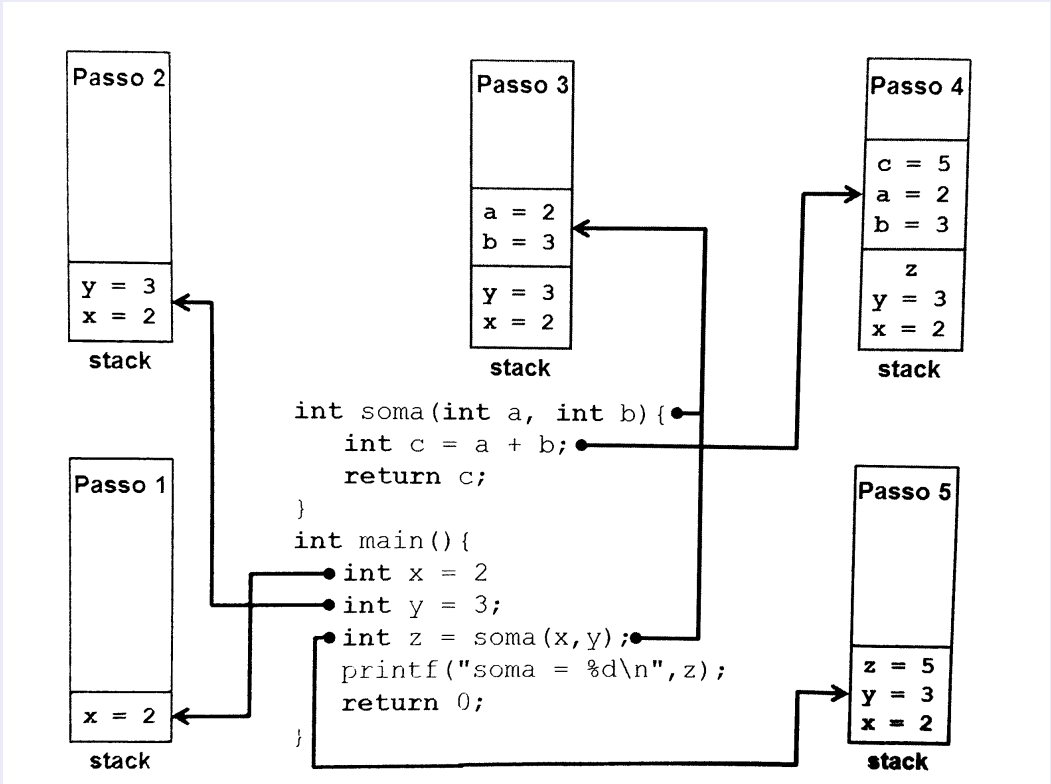
- Este processo é chamado de Alocação Estática de Memória.

Alocação Estática

- Cada tipo de variável necessita de uma quantidade de memória que é automaticamente reservada na **Pilha de Execução**.
- Vantagens:
  - Os dados ficam armazenados sequencialmente na memória (vetores).
  - Programador não precisa se preocupar em gerenciar a memória.
- Desvantagens:
  - Programador não tem controle sob o tempo de vida das variáveis.
  - Quantidade de memória utilizada pelo programa é definida previamente.
  - Espaço reservado não pode ser alterado.
  - Podem haver espaços reservados desnecessariamente.

Alocação Estática

Exemplo da pilha de execução



Alocação Estática

E quando a quantidade de memória necessária
durante a execução do programa
NÃO é previamente conhecida?

Alocação Estática

- Considere um problema onde necessita-se cadastrar o valor gasto por cada cliente de uma loja.
  - Se utilizarmos alocação estática, é preciso definir uma quantidade máxima de clientes já que não sabemos exatamente quantos clientes a loja terá.

```
float V[1000]; // máximo definido como 1000, por exemplo
```

- Problemas dessa solução:
  - Se precisar cadastrar mais de 1000 clientes o programa não servirá.
  - Se cadastrar poucos clientes haverá um desperdício de memória.

Solução: Alocação Dinâmica de Memória.

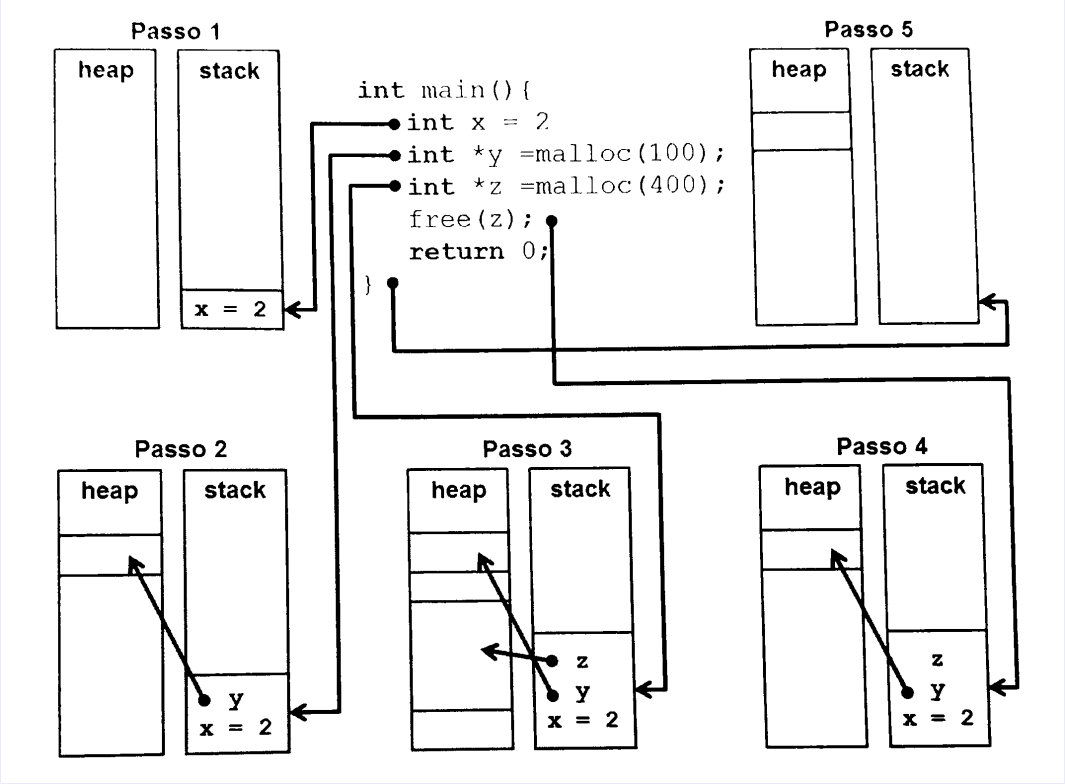
Alocação Dinâmica

- A memória é reservada dinamicamente (em tempo de execução).
- Esta reserva não é feita na **Pilha de Execução**, mas em um outra área da memória (**Heap**).
  - Na linguagem C, a alocação é feita pela função `malloc()`.
  - Os dados desta área da memória só podem ser acessados por ponteiros.
- Vantagens:
  - Variáveis não dependem do escopo.
  - Quantidade total de memória não precisa ser previamente conhecida.
  - Espaço de memória pode ser alterado durante a execução do programa.
  - Programador controla o tempo de vida das variáveis.
- Desvantagens:
  - Os dados não são necessariamente armazenados de forma sequencial.
  - A memória utilizada deve ser alocada e liberada manualmente.

Observação: esquecer de liberar a memória pode gerar falhas..

Alocação Dinâmica

Exemplo da pilha de execução



Alocação Dinâmica

- Vetores (*arrays* unidimensionais):
  - Sequência de dados de mesmo tipo armazenados automaticamente na memória (alocação estática).

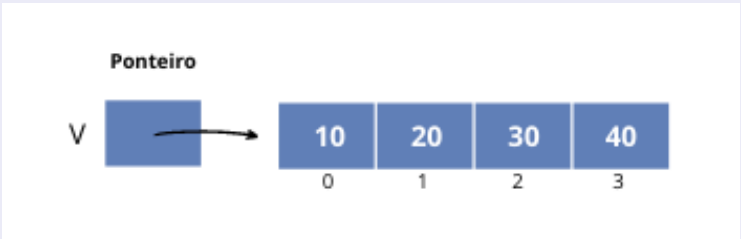
```
int V[4]; // pode-se dizer que o ponteiro V contém o endereço de V[0]
```

  - A Linguagem C também permite definir um ponteiro para acessar um bloco de memória alocado dinamicamente.

```
int *v; // após definição do ponteiro, aloca-se memória para 4 inteiros
```

  - Para isso, utiliza-se as funções da Linguagem C apresentadas a seguir.

Exemplo de um vetor de números inteiros com 4 posições



Alocação Dinâmica

- As funções da linguagem C usadas na alocação dinâmica de memória são encontradas na biblioteca `stdlib.h` [1]. São elas:
  - Operador `sizeof`.
  - Função `malloc`.
  - Função `free`.
  - Função `calloc`.
  - Função `realloc`.

[1] https://www.tutorialspoint.com/c_standard_library/stdlib_h.htm

Alocação Dinâmica

- Operador `sizeof` [1]:
  - Retorna o número de bytes necessários para alocar um único dado de determinado tipo.
  - **Sintaxe:** `sizeof(nome_do_tipo)`.
  - **Exemplos:**

    `sizeof(int)` = 4 bytes.
    `sizeof(float)` = 4 bytes.
    `sizeof(double)` = 8 bytes.
    `sizeof(char)` = 1 byte.

[1] https://www.tutorialspoint.com/cprogramming/c_sizeof_operator.htm

Alocação Dinâmica

- Função `malloc` [1]:
  - Usada para alocar memória durante a execução do programa.
  - Retorna um ponteiro que contém o endereço do início do espaço alocado na memória ou `NULL` em caso de erro (quando não tem memória suficiente para ser alocada).
    - ★ **Protótipo:** `void *malloc(unsigned int num);`
  - **Exemplos:**

```
// alocação dinâmica de um vetor com 10 números inteiros
int *v = (int*) malloc(40); // quantidade de memória é 10 * 4 bytes

// mesma alocação usando o operador sizeof
int *v = (int*) malloc(10 * sizeof(int)); // recomendado
```

[1] https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm

Alocação Dinâmica

- Função `free` [1]:
  - Usada para liberar a memória alocada dinamicamente.
  - Se a memória alocada dinamicamente não for liberada corretamente, ela fica reservada e não poderá ser usada por outros programas.
    - ★ **Protótipo:** `void free(void *ptr)`.
  - **Exemplos:**

```
// alocação dinâmica de um vetor com 10 números inteiros
int *v = (int*) malloc(10 * sizeof(int));

// liberação da memória alocada
free(v);
```

[1] https://www.tutorialspoint.com/c_standard_library/c_function_free.htm

Alocação Dinâmica

- Função `calloc` [1]:
  - Usada para alocar memória durante a execução do programa.
  - Assim como o `malloc`, retorna um ponteiro que contém o endereço do início do espaço alocado na memória ou `NULL` em caso de erro (quando não tem memória suficiente para ser alocada).
  - Diferença: inicializa todos os bits de espaço alocado com zero.
    - ★ **Protótipo:** `void *calloc(unsigned int num, unsigned int size).`
  - **Exemplo:**

```
// alocação dinâmica de um vetor com 10 números inteiros
int *v = (int*) calloc(10, sizeof(int));
```

[1] https://www.tutorialspoint.com/c_standard_library/c_function_calloc.htm

Alocação Dinâmica

- Função `realloc` [1]:
  - Usada para alocar ou realocar memória durante a execução.
  - Retorna um ponteiro que contém o endereço do início do espaço alocado na memória ou `NULL` em caso de erro (quando não tem memória suficiente para ser alocada).
    - ★ **Protótipo:** `void *realloc(void *ptr, unsigned int num);`
  - **Exemplos:**

```
// alocação dinâmica de um vetor com 10 números inteiros
int *v = (int*) malloc(10 * sizeof(int));

// realocação aumentando o tamanho de v para 100 posições
v = (int*) realloc(v, 100 * sizeof(int));
```

[1] https://www.tutorialspoint.com/c_standard_library/c_function_realloc.htm

Alocação Dinâmica

- **Observação:**
  - Se a realocação falhar, a função `realloc` retornará `NULL` e a memória alocada anteriormente será perdida. Para que isso não ocorra, pode-se utilizar um ponteiro auxiliar para a realocação (`aux`).

```
// realocação aumentando o tamanho de v para 100 posições
int *aux = (int*) realloc(v, 100 * sizeof(int));
if(aux != NULL) v = aux; // caso contrário, v continua com tamanho 10
```

- **Outros exemplos:**

```
// realloc usado como equivalente ao malloc anterior
int *v = (int*) realloc(NULL, 10 * sizeof(int));

// realloc usado como equivalente a função free
v = (int*) realloc(v, 0);
```

Alocação Dinâmica

- Matrizes (*arrays* multidimensionais):
  - Usa-se o conceito de ponteiro para ponteiro.

```
int *v; // matriz com 1 dimensão (vetor)
int **p; // matriz com 2 dimensões
int ***d; // matriz com 3 dimensões
```

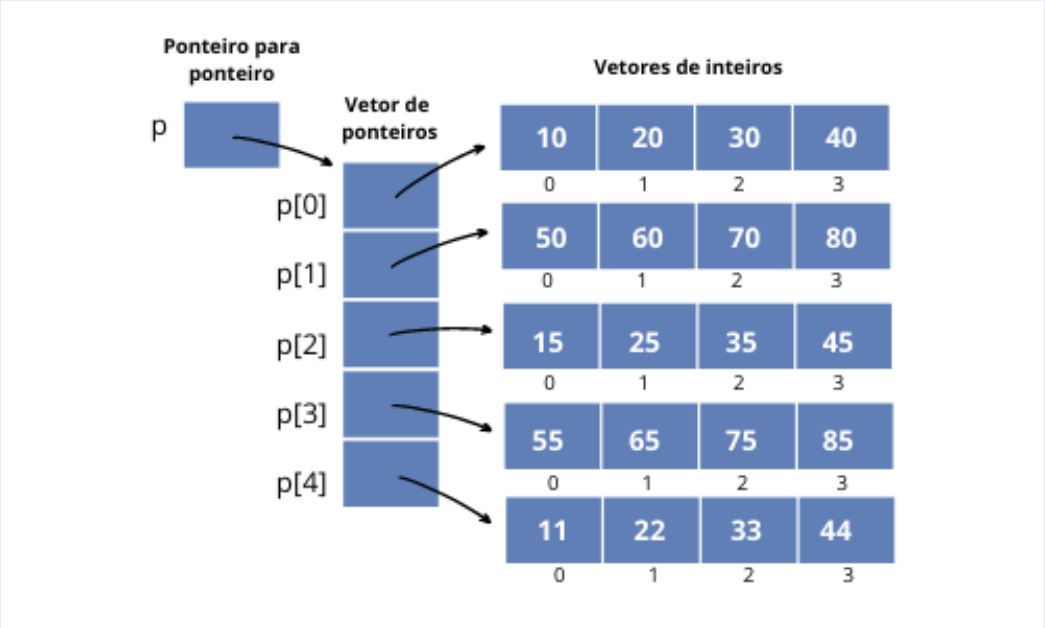
  - **Exemplo:**

```
// alocação dinâmica de uma matriz (m x n) de inteiros
int **p = (int**) malloc(m * sizeof(int*)); // vetor de ponteiros

// alocação dos vetores referentes a cada linha da matriz
for(int i=0; i<m; i++){
    p[i] = (int*) malloc(n * sizeof(int)); // vetor de inteiros
}
```

Alocação Dinâmica

Exemplo de uma matriz bidimensional de números inteiros com 5 linhas e 4 colunas



Alocação Dinâmica

- Matrizes (*arrays* multidimensionais):
  - Note que primeiro foi criado um vetor de ponteiros `p` (ponteiro para ponteiro) que representa a matriz bidimensional.
    - ⋆ Cada ponteiro `p[i]` desse vetor aponta para a posição 0 de um vetor de inteiros (alocado posteriormente) que representa a linha `i` da matriz.
  - Para liberação da memória, essa ordem deve ser inversa (antes de liberar `p`, deve-se liberar os ponteiros `p[i]`).
  - **Exemplo:**

```
// liberação de memória dos vetores de inteiros
for(int i=0; i<m; i++){
     free(p[i]);
}

// liberação de memória da matriz (vetor de ponteiros)
free(p);
```

Referências Bibliográficas

1. ASCÊNCIO, A. F. G.; CAMPOS, E. A. V. ***Fundamentos da Programação de Computadores***. 2012.

2. BACKES, A. ***Linguagem C: Completa e Descomplicada***. 2013.
    1. Vídeo aulas:
    `https://programacaodescomplicada.wordpress.com/indice/linguagem-c/`

3. XAVIER, E. C. ***Material Didático de MC102 (IC/UNICAMP)***.

    Aula 20 (Ponteiros II):

    https://www.ic.unicamp.br/~eduardo/material_mc102/aula20.pdf.

    Aula 21 (Ponteiros III):

    https://www.ic.unicamp.br/~eduardo/material_mc102/aula21.pdf.