
Algoritmos
Aplicações práticas
na Linguagem C
Th

Sumário

Introdução

Seja muito bem-vindo(a) ao curso Algoritmos e suas aplicações na linguagem C. Se está lendo este material, provavelmente você o conheceu pelo meu canal do YouTube, ou por meio de alguma divulgação. Como alguém que já percorreu este caminho e estudou os conceitos aqui abordados, recomendo fortemente que você se dedique a resolver todos os exercícios propostos ao longo de cada conteúdo estudado, mesmo que se depare com alguma dificuldade, o que eventualmente pode acontecer. Esse esforço não só ajudará a consolidar sua compreensão, mas também a construir uma bagagem de aprendizado e dominar a maioria dos conceitos fundamentais, essenciais para o seu desenvolvimento no campo da computação. Lembre-se: a prática constante é o que transforma conhecimento em habilidade.

Neste curso, abordaremos alguns conceitos essenciais da computação para facilitar a compreensão dos conteúdos apresentados. Entre os tópicos explorados, destacam-se as diferentes estruturas utilizadas no desenvolvimento de programas e na escrita de código, a importância de realizar uma análise detalhada do problema, a necessidade de garantir clareza e legibilidade no código, além de documentar adequadamente o processo de desenvolvimento. Também discutiremos a interpretação lógica e matemática como ferramenta-chave na resolução de problemas, aplicando a estruturação de algoritmos de forma eficiente e organizada. Com esses fundamentos, você estará mais bem preparado(a) para enfrentar os desafios da programação.

“Algoritmos e suas aplicações na linguagem C” foi desenvolvido para atingir um público amplo que se interesse pelo assunto, utilizando exemplos simplificados e claros de conceitos fundamentais que servirão de base para a resolução de problemas mais complexos no futuro. Portanto, mesmo que você não tenha nenhum conhecimento prévio sobre programação ou nunca tenha tido contato direto com essa área, este conteúdo foi pensado especialmente para você! Minha abordagem será sempre focada na resolução passo a passo, para que você se familiarize com os conceitos de forma acessível e didática, podendo progredir posteriormente por conta própria nos seus aprendizados.

Desde já, desejo-lhe ótimos estudos e que esta apostila seja a porta de entrada para seus primeiros passos no mundo da computação. E é claro, sinta-se à vontade para explorar os tópicos abordados aqui em outras fontes, como na Internet, livros e cursos, pois conhecimento nunca é demais! Aprofundar-se em cada tema garantirá que você construa um entendimento sólido e esteja melhor preparado(a) para os desafios futuros.

Conceitos básicos computacionais

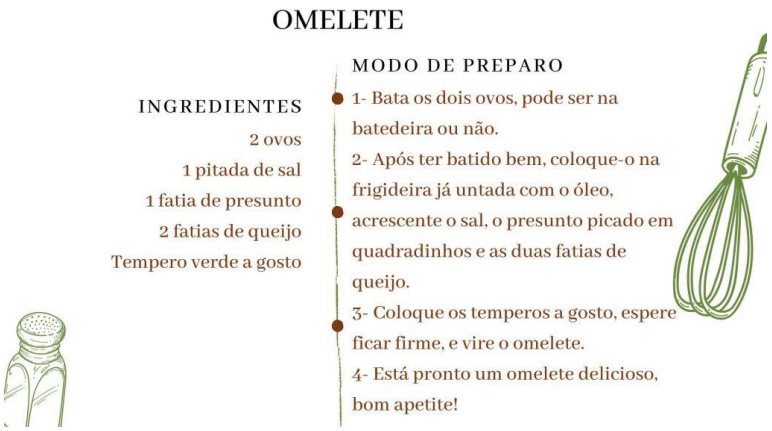
Antes de mais nada, é fundamental abordarmos os principais conceitos introdutórios da computação. Compreender essas noções básicas é essencial para garantir que, ao nos aprofundarmos em temas mais avançados, você tenha uma base sólida para interpretar e aplicar os novos conhecimentos de forma eficaz. Nesta seção, iremos explorar os fundamentos que servirão como alicerces para seu aprendizado, como a estrutura de dados, a lógica de programação, e os princípios dos algoritmos. Essa preparação não só facilitará sua compreensão ao longo do curso, mas também permitirá que você desenvolva habilidades críticas para enfrentar desafios mais complexos no futuro. Assim, você estará pronto(a) para mergulhar nos tópicos mais detalhados e avançados que iremos abordar adiante.

Definindo…

Algoritmos: De forma simplificada, algoritmos nada mais são do que instruções que fornecemos para que nosso computador realize determinadas tarefas, seguindo sempre uma sequência pré-definida de etapas.

Vamos trazer um exemplo para nosso dia a dia, tornando mais fácil de visualizar: Supondo que você chegou agora em uma cidade como turista e não faz ideia de como chegar a determinado lugar, ignorando a existência de uber ou GPS, o mais provável a se fazer é pedir informações para algum habitante local, a fim de que esclareça qual o caminho escolher. Entendendo isso, um algoritmo nessa situação poderia ser simplesmente a sequência de instruções ordenadas dadas pela pessoa a você: “Bom, primeiramente você pode seguir reto por esta rua, chegando até o final dela, vire a direita e você chegará ao seu destino”. Seguindo etapa por etapa, chega-se ao resultado final: seu destino desejado.

Além do exemplo de navegação em uma nova cidade, sendo um turista, como fornecido acima, também temos um dos algoritmos mais comuns que utilizamos em nosso dia a dia, que é basicamente a receita de algum determinado alimento.



Fonte: acervo pessoal de Ana Cristina Diniz

Note que também no caso da receita, a ordem e a sequência das etapas foram cruciais para que o resultado final estivesse conforme o esperado. O sucesso de um prato depende diretamente da forma como os ingredientes são adicionados, do tempo de cozimento e das técnicas de preparo utilizadas. Por exemplo, se você adicionar um ingrediente no momento errado ou pular uma etapa, o resultado pode ser completamente diferente do que você imaginava. Isso demonstra claramente que, assim como um algoritmo, uma receita é uma sequência de instruções que deve ser seguida cuidadosamente para alcançar um resultado específico.

Linguagem de programação: Uma linguagem de programação é um conjunto de regras sintáticas e semânticas que permite aos programadores escrever instruções que um computador pode compreender e executar. Essas linguagens são usadas para criar softwares, scripts e programas que executam determinadas tarefas. Existem diversos tipos de linguagens de programação, cada uma com suas próprias características, usos e normas, como C (abordada neste curso), Python, Java e JavaScript, que variam em complexidade, funcionalidade e aplicações práticas.

Em geral, as linguagens de programação podem ser classificadas em baixo nível, as quais são mais próximas da linguagem de máquina, sendo muito utilizado os conceitos de binário e hexadecimal, e geralmente requerem um conhecimento mais aprofundado sobre a conexão entre ela e o hardware. Uma das mais conhecidas chama-se Assembly, interpretada por microprocessadores e micro controladores.

Temos também as linguagens de programação classificadas em alto nível, que são mais próximas da linguagem humana e oferecem uma sintaxe mais intuitiva e legível. Essas linguagens não se centram em muitos detalhes técnicos do hardware, permitindo que

os programadores foquem mais na lógica do programa a ser desenvolvido em vez da manipulação direta da comunicação com a máquina. Os exemplos mais populares de linguagens de alto nível incluem Python, Javascript, C++ e PHP

Variável: Durante nosso curso, mencionaremos o conceito de variável inúmeras vezes, pois é um dos conceitos fundamentais da programação. Antes de explorarmos seus usos propriamente, é importante entender o conceito básico de uma variável.

De forma resumida, uma variável é um espaço na memória RAM do computador que nós programadores, solicitamos ao sistema operacional para armazenar informações temporariamente enquanto o programa está em execução. E é neste espaço que guardamos valores de diferentes tipos, como números (inteiros ou decimais), caracteres, textos (strings), ou outros dados mais complexos.

Podemos imaginar uma variável como uma caixa, onde podemos armazenar um item específico. Cada caixa tem um nome ou identificador único que usamos para localizá-la na memória, e cada caixa só pode guardar um tipo de item por vez, como inteiros, caracteres, ou booleanos, dependendo do tipo de dado que definimos ao declarar a variável.

Tipos de dados: Como mencionado anteriormente, existem diversos tipos de dados que podem ser armazenados em variáveis, e essa classificação define quais informações uma variável pode conter. Neste tópico, abordarei apenas os tipos principais para a sua compreensão inicial na linguagem C. São eles:

- char: Caracteres e símbolos de modo geral.
  Ex: ‘A’, ‘1’, ‘$’, ‘-’, etc.
- int: Números inteiros, sem parte decimal.
  Ex: 1, 25, -15304, etc.
- float: Números decimais de ponto flutuante (precisão simples: 6 casas decimais).
  Ex: 0.5, 23.5, -503.34, etc.
- double: Números decimais de ponto flutuante (dupla precisão: 15 casas decimais).
  Ex: 3.14159, 940.390402, 10344.32390390423415, etc.

IDEs: A sigla IDE (do inglês Integrated Development Environment, ou Ambiente de Desenvolvimento Integrado) refere-se a um conjunto de ferramentas e funcionalidades projetadas para auxiliar o programador na escrita de código durante o desenvolvimento de

software. Uma IDE oferece recursos como detecção de problemas sintáticos, identificação de bugs, sugestões de estruturas, e autopreenchimento de nomes de funções da linguagem de programação escolhida.

De forma simplificada, podemos entender uma IDE como um "bloco de notas" avançado, desenvolvido especificamente para uma linguagem de programação. Ela se adapta aos padrões e normas dessa linguagem e, geralmente, inclui um compilador, permitindo a execução do código diretamente na máquina do programador para testes e depuração.

Atualmente, há uma variedade de IDEs gratuitas disponíveis, tanto para desktop quanto para web. A seguir, apresentarei as principais e mais utilizadas para programação em linguagem C, para que você possa escolher aquela que mais lhe agrada: Code::Blocks, Geany, Dev-C++, Visual Studio Code e GDB (online).

É importante ressaltar que, independentemente do ambiente de desenvolvimento escolhido, o processo de aprendizado será semelhante, pois a linguagem de programação e as metodologias utilizadas permanecem consistentes, embora alguns ambientes possam estar mais desatualizados que outros. Portanto, pesquise e escolha aquele com o qual você mais se identifica e se adapta, garantindo que o aprendizado seja mais agradável e atenda às suas necessidades.

Compilador/interpretador: Compiladores e interpretadores podem ser comparados a tradutores que permitem a comunicação entre duas pessoas que falam línguas diferentes. Imagine que duas pessoas precisam se comunicar, mas cada uma fala um idioma distinto. Para que elas possam se entender, seria necessário um tradutor para converter

as falas de uma pessoa para a língua da outra. No mundo da programação, os compiladores e interpretadores assumem esse papel, traduzindo o código escrito em uma linguagem de programação de alto nível, compreensível para humanos, para a linguagem de máquina, entendida pelo computador, permitindo que o computador execute as instruções fornecidas.

Entretanto, existem algumas diferenças entre o compilador e interpretador, sendo que um compilador funciona como um tradutor que traduz todo o conteúdo de uma única vez. Ele pega o código-fonte inteiro, analisa e o converte tudo para linguagem de máquina, gerando um arquivo executável (.exe) que pode ser executado pelo sistema operacional. Depois de compilado, o código não precisa mais do compilador para ser executado, sendo necessário apenas o arquivo gerado. Esse processo tende a ser mais otimizado e eficiente no momento da execução, mas podendo demorar mais durante a compilação, uma vez que tudo é traduzido de uma só vez. C, C++ e Java são exemplos de linguagens que se usam compiladores na hora da execução do código.

Já o interpretador trabalha de forma interativa, etapa por etapa, traduzindo o código escrito linha por linha, executando cada instrução de cada vez, logo após que é interpretada. A principal diferença é que o interpretador não gera um arquivo executável, o que significa que toda vez que o programa rodar, o interpretador precisa estar presente para realizar a tradução em tempo real novamente. Isso geralmente resulta em uma execução mais lenta do programa. Linguagens como Python e JavaScript utilizam interpretadores.

Software: Um Software pode ser entendido como um conjunto organizado de algoritmos escritos em uma determinada linguagem de programação a fim de orientar o computador a desempenhar tarefas específicas e atender determinadas necessidades. Vamos entender Software como se fosse vários algoritmos sintetizados em algo maior, gerando um produto final. Esse produto pode ser uma aplicação desktop, instalável nas máquinas através de standalones ou setups, um aplicativo web, rodando por meio de um servidor físico ou na nuvem, um jogo, instalado a partir de arquivos locais, ou qualquer outro tipo de software. A eficiência e a eficácia desse produto dependem da qualidade e formas de otimização dos algoritmos implementados e de como os mesmos foram integrados.

Sistema Operacional: O sistema operacional pode ser entendido como um conjunto de softwares que gerenciam e coordenam o uso dos recursos do sistema, como o hardware, a memória, os dispositivos de entrada e saída, e os processos em execução.

Ele serve como uma ponte entre o usuário e o hardware do computador, garantindo

que os programas sejam executados de forma otimizada e segura, evitando bugs.

Exercícios propostos - Conceitos básicos computacionais

01) Explique com suas palavras a diferença entre um algoritmo e uma linguagem de programação.

02) Defina variável e explique se é possível armazenar mais de um tipo de valor na mesma.

03) Determine se os tipos de dados a seguir conferem, com relação aos tipos de dados da linguagem C, determinando V (verdadeiro) ou F (falso):
a) int = números inteiros;
b) char = caracteres unicos e textos;
c) double = números decimais e inteiros;
d) float = números inteiros;

04) Diferencie compilador de interpretador e exemplifique linguagens de programação usadas por ambos.

05) Defina o que é um software e dê exemplos conhecidos por você.

Estruturas sequenciais

Neste capítulo iremos nos concentrar na aplicação prática de conhecimentos amplamente utilizados na programação, atrelados aos conceitos fundamentais do capítulo anterior. Focaremos nas primeiras estruturas empregadas na codificação de programas e algoritmos, as estruturas sequenciais.

Inicialmente, vamos definir estruturas sequenciais como uma forma de escrever o código na qual uma determinada ordem de execução é seguida em nosso programa, acontecendo step by step (etapa por etapa). Esse tipo de escrita do código não permite que tomemos decisões que alteram o fluxo de execução do nosso código, por esse motivo se chama sequencial, que também pode ser entendido como linear ou contínuo.

Em algoritmos, geralmente escrevemos um código utilizando o portugol, uma linguagem de programação algorítmica escrita em pseudocódigo, alinhada com a língua portuguesa. O portugol é geralmente uma forma mais simples e didática de visualizar a resolução de um problema. Entretanto, neste curso estarei disponibilizando algoritmos em pseudocódigo (inglês) e linguagem C, a fim de reforçar a compreensão de ambos, conforme mencionado anteriormente.

Conceitos fundamentais – Estruturas sequenciais

Bibliotecas: As bibliotecas na programação são um conjunto de funções que possuem seus próprios códigos que resolvem o problema específico delas. No geral, usamos diversas bibliotecas para o bom funcionamento do nosso programa, permitindo usarmos ferramentas adicionais e facilitar o nosso caminho durante a escrita do código.

Além de economizar tempo e esforço ao evitar a necessidade de escrever código do zero, as bibliotecas também ajudam a organizar o código, tornando-o mais legível e fácil de manter, de forma mais sintetizada. Com funções agrupadas em bibliotecas, o programador pode entender rapidamente o propósito de cada parte do código, melhorando a clareza do projeto.

Muitas bibliotecas oferecem ferramentas e algoritmos mais complexos que podem ser utilizados sem a necessidade de compreendê-los completamente ou visualizar o código na íntegra. Essa abstração permite que os programadores se concentrem na lógica de seus próprios programas, utilizando funcionalidades avançadas que já foram testadas e otimizadas. Para utilizar essas bibliotecas em C, é necessário incluí-las no código-fonte usando o prefixo #include, seguido da sintaxe <nome_da_biblioteca.h>. Neste primeiro momento, teremos algumas bibliotecas importantes que irão permitir que nosso código

funcione corretamente, em linguagem C:

- <stdio.h>: Biblioteca do sistema padrão para controle da entrada e saída de dados do nosso programa.
- <stdlib.h>: Biblioteca diretiva de compilação, contendo funções do sistema.
- <math.h>: Biblioteca relacionada às funções matemáticas para nosso programa.

Funções: Funções podem ser definidas como um bloco de código que resolve um determinado problema. Elas desempenham um papel fundamental na programação, permitindo a modularização e a sua reutilização no código várias vezes. As funções apresentam, de modo geral, algumas características básicas, como:

- Tipo de função:
  - int, float ou double: retornará valores numéricos.
  - char: retornará um caractere.
  - void: a função não retorna nenhum valor.
- Identificador da função: Cada função deve apresentar um identificador ou nome único.
- Parâmetros: Trata-se de valores recebidos de entrada pela função, que poderão ser usados internamente no código da mesma.

Todavia, neste curso, vamos tratar apenas da função principal do código, que pode ser nomeada conforme o algoritmo construído em Pseudocódigo ou como int main() na linguagem C. Ambas as definições são seguidas por chaves {}, onde o código será inserido.

Ex:

```
// Inicializando o programa principal em:
// Pseudocódigo
Algorithm: nomeAlgoritmo // Inicializando o algoritmo
   Comando 1
   Comando 2
   Comando 3
      ...
end nomeAlgoritmo // Indica o fim do algoritmo
```

```
// Linguagem C
#include <stdio.h> // Biblioteca padrão de comandos de entrada/saída

int main() // Inicializando a função principal do programa
{
    Comando 1;
    Comando 2;
    Comando 3;
        ...
    return 0; // Retorno de valor, indicando que o programa acabou
}
```

Obs: Temos algumas funções em linguagem C importantes que poderão ser usadas em nossos programas para realização de cálculos matemáticos, funcionando juntamente com a inclusão da biblioteca <math.h>, citada anteriormente. São elas:

- pow(variável ou número, expoente): Eleva um determinado número ou conteúdo presente em uma variável a um determinado expoente.
  Ex: pow(2, 5) = 2^5 = 32
- sqrt(variável ou número): Calcula a raiz quadrada de um número ou conteúdo presente em uma variável
  Ex: sqrt(9) = $\sqrt{9}$ = 3

Comentários (// ou //):** Comentários são anotações feitas pelos programadores com o objetivo de explicar trechos específicos de um código. Eles são ignorados pelo compilador, o que significa que não afetam o comportamento ou a execução do programa.

Ex:

```
// Isso é um comentário (Comentário de linha única)
/* Isso também é um comentário */ (Comentário de linha única)
/*
        Esse
        é o comentário
        que abrange várias linhas
```

*/ (Bloco de comentário)

Os comentários são extremamente importantes para a legibilidade do código, facilitando a compreensão por outros programadores, ou pelo próprio autor do código em um momento futuro. Eles ajudam a tornar o código mais claro e organizado, servindo como uma ferramenta de documentação, especialmente em partes mais complexas, onde é fundamental descrever o raciocínio por trás da implementação.

Variáveis: Revisando, variáveis são espaços alocados na memória RAM utilizados para armazenar diferentes tipos de valores ou dados durante a execução de um programa. Ao declarar uma variável, o programador indica que o sistema operacional deve reservar uma parte da memória para armazenar um valor que poderá ser modificado ao longo do tempo. O tamanho desse espaço varia de acordo com o tipo de dado que a variável vai armazenar (por exemplo, caracteres, inteiros ou números decimais).

Ex:

```
//Declarando variáveis
// Pseudocódigo
declare:
    idade: integer
    altura: real

// Linguagem C
int idade;
float altura;
```

Neste caso, declaramos uma variável de identificador/nome *idade* do tipo inteiro (*int*), ou seja, ela só poderá armazenar valores numéricos inteiros (sem parte decimal), no caso valores para a idade de alguém.

Declaramos também uma variável chamada “*altura*” do tipo real (*float*), dessa forma, ela só poderá guardar números com dígitos decimais.

A declaração de variáveis é um conceito fundamental da programação, pois utilizamos variáveis constantemente para manipular dados de diversos tipos e realizar operações de diferentes formas.

Constantes: Diferente das variáveis, cujo valor pode ser alterado ao longo da execução de um programa, as constantes são valores fixos que, uma vez definidos, não podem ser modificados. Elas são imutáveis durante a execução do código e só podem ser alteradas diretamente no código-fonte, antes da compilação.

As constantes são especialmente úteis para representar valores que não devem mudar ao longo do tempo, como constantes matemáticas, limites máximos, strings (sequências de caracteres) ou valores fixos que fazem parte da lógica do programa.

Além disso, o uso de constantes pode contribuir para a redução de erros durante operações, por exemplo, uma vez que a alteração de um valor fixo precisa ser feita em um único lugar, minimizando o risco de mudanças indesejadas.

Ex:

```
//Declarando constantes
// Pseudocódigo
declare:
    const real X <- 15.234
    const real PI <- 3.14159

// Linguagem C
const float X = 15.234;
const double PI = 3.14159;
```

Durante a declaração de constantes, seguimos algumas convenções para padronizar e facilitar a identificação das mesmas, como geralmente declará-las com letras maiúsculas. No exemplo acima, utilizamos o prefixo *const* para indicar que estamos declarando uma constante. A constante *X*, do tipo *float*, recebe o valor de 15.234.

Além disso, declaramos a constante *PI*, que segue o mesmo padrão, mas é do tipo double, o que proporciona maior precisão em cálculos que envolvem esse valor. Usar constantes como *PI* em um programa não apenas melhora a legibilidade, mas também garante que o valor não será alterado acidentalmente, evitando erros em cálculos que dependem de valores imutáveis.

Comando de atribuição (<- ou =): Um comando de atribuição é responsável por atribuir/dar um valor a uma variável ou constante. Podemos compará-lo a armazenar um item em uma caixa organizada, onde a caixa representa a variável ou constante, e o item

é o valor que estamos armazenando. Assim como uma caixa só pode conter itens de um determinado tipo, uma variável deve ser do mesmo tipo que o valor que estamos atribuindo a ela.

Por exemplo, temos uma caixa do tipo *inteiro*, ela só pode receber valores desse mesmo tipo, no caso números inteiros. Essa organização garante que o programa funcione corretamente e que os dados possam ser manipulados da forma correta.

Ex:
//Comando de atribuição
x <- 120; (Pseudocódigo)
y = 5; (Linguagem C)

Nesses casos, estamos determinando um valor que cada variável irá receber, onde no exemplo *x recebe* 120 e *y recebe* 5. Vale ressaltar que assim como no segundo exemplo, podemos atribuir um valor à alguma variável/constante no momento em que já estamos a declarando. Portanto, lê-se: “x recebe 120” ou “y recebe 5”.

Note que no exemplo acima foram utilizados os símbolos <- e =. No entanto, em Pseudocódigo, usamos a seta (<-) para representar o comando de atribuição, que indica que um valor ou expressão será armazenado em uma variável. Já em linguagens de programação mais amplamente utilizadas, como C, Java ou Python, o sinal de igualdade (=) é utilizado para esse mesmo propósito. Apesar da diferença nos símbolos, o conceito permanece o mesmo: atribuir um determinado valor a uma variável.

Comando de entrada e saída: Durante o processo de codificação, é comum desejarmos que os valores atribuídos a uma variável ou constante sejam inseridos diretamente pelo usuário. Para atender a essa necessidade, utilizamos o comando de entrada, que permite a leitura de valores por meio do teclado. Esse comando armazena as informações digitadas na variável ou constante especificada, possibilitando a interação do usuário com o programa.

Ex:
//Comando de entrada
read x; (Pseudocódigo)
scanf(“%d”, &x); (Linguagem C)

```
//Comando de saída
write x; (Pseudocódigo)
printf(“%d”, x); (Linguagem C)
```

Obs: Dentro das aspas (") nas funções printf e scanf, encontramos os chamados especificadores de formato, esses indicam a quantidade e o tipo de valores aos quais nos referimos ao especificarmos quais variáveis serão lidas ou impressas. Temos também, após a vírgula os chamados argumentos, responsável por representar qual deve ser o valor a ser usado de referência pelas funções. Cada argumento é representado por um especificador de formato, como %d para inteiros, %f para números de ponto flutuante, e assim por diante. Esses especificadores ajudam a função a determinar como manipular os dados das variáveis associadas. Vale ressaltar que para pularmos uma linha na impressão, utilizamos o caractere \n dentro das aspas da função printf, que indica uma nova linha na impressão (new line).

Note que a maneira como realizamos a leitura de valores pode variar um pouco, dependendo da estrutura da linguagem que escolhemos para programar. No caso da linguagem C, além de solicitar que o programa leia dados utilizando a função scanf(), também é necessário informar o tipo de dado a ser lido.

Como discutido anteriormente, existem diferentes tipos de dados na linguagem C. Durante a leitura de cada um, utilizamos os chamados especificadores de formato, como %c, %d, %f, %lf, entre outros. Esses especificadores informam à linguagem qual tipo de dado será lido no momento da execução do código, assegurando que o valor inserido pelo usuário seja armazenado corretamente na variável apropriada, indicada pelo operador &, que representa o endereço da variável em questão.

Já no comando de saída, imprimimos/exibimos na tela do computador o valor contido na variável em questão. Nesse caso, não utilizamos o (&) na linguagem C, uma vez que queremos imprimir o conteúdo da variável e não o seu endereço de memória. Podemos também, para os números decimais, especificar quantas casas decimais queremos que o programa imprima, utilizando a notação: %.5lf (5 casas decimais), %.2f (2 casas decimais), etc.

Especificadores de formato – Linguagem C

%c	Representa um caractere único (char).
%d	Representa um número inteiro (int).
%ld	Representa um número inteiro de 32 bits de capacidade (long int)
%lld	Representa um número inteiro de 64 bits de capacidade (long long int)
%f	Representa um número de ponto flutuante de precisão simples (float).
%lf	Representa um número de ponto flutuante de precisão dupla (double).

Operadores: Na programação, assim como na matemática, os operadores básicos que utilizamos são adição, subtração, multiplicação e divisão. Esses operadores seguem a mesma ordem de precedência utilizada em cálculos matemáticos, garantindo que as operações sejam realizadas corretamente e de forma previsível. Vejamos alguns exemplos de como utilizá-los em nossos programas:

- Adição (+): Realiza a operação de soma entre dois valores.
  Ex: r <- x+y (Pseudocódigo) ou r = x+y (Linguagem C);
- Subtração (-): Realiza a operação de diferença entre dois valores.
  Ex: r <- x-y (Pseudocódigo) ou r = x-y (Linguagem C);
- Multiplicação (*): Realiza a operação de multiplicação entre dois valores.
  Ex: r <- x*y (Pseudocódigo) ou r = x*y (Linguagem C);
- Divisão (/ ou div): Realiza a operação de divisão entre dois valores.
  Ex: r <- x div y (Pseudocódigo) ou r = x/y (Linguagem C);
- Resto da divisão (% ou mod): Realiza a operação do resto da divisão entre dois valores.
  Ex: r <- x mod y (Pseudocódigo) ou r = x%y (Linguagem C);

Nesse contexto, variáveis e constantes atuam como operandos, sendo diretamente associadas aos operadores mencionados anteriormente. A interação entre operadores e

operandos é fundamental para a manipulação de diversos dados, permitindo realizar cálculos e atribuições nas diversas linguagens de programação.

Exercícios propostos – Estruturas sequenciais

01) Desenvolva um algoritmo que leia dois valores numéricos fornecidos pelo usuário e armazene-os em duas variáveis, a e b. Em seguida, o algoritmo deve calcular a soma desses valores, armazenando o resultado em uma terceira variável x. Por fim, imprima o valor de x.

Exemplo de caso de teste

Entrada	Saída
3 7	10

02) Crie um programa que calcule a média ponderada de um aluno, considerando que a primeira prova (P1) tem peso de 60% e a segunda prova (P2) tem peso de 40%. O algoritmo deve solicitar ao usuário as notas das duas provas e, em seguida, calcular e exibir a média final com uma casa decimal após a vírgula.

Exemplo de caso de teste

Entrada	Saída
6.4 8.3	7.16

03) Faça um algoritmo que calcule o valor da hipotenusa de um triângulo retângulo, onde é dado o valor do cateto oposto. Entretanto o valor do outro cateto deve ser fornecido pelo usuário através da entrada do teclado. Utilize o Teorema de Pitágoras para realizar o cálculo da hipotenusa e exiba o resultado ao final com três dígitos após a vírgula.
Dados: cateto oposto = 3.75

Exemplo de caso de teste

Entrada	Saída
4.0	5.483

04) Escreva o código de um programa que calcule, a partir dos anos dados, quantos meses, dias, horas, e segundos uma pessoa viveu e imprima o resultado ao final.
Dados: ano = 365 dias

Exemplo de caso de teste

Entrada	Saída
35	420 meses, 12775 dias, 306600 horas, 1103760000 segundos

05) Identifique abaixo quais expressões são verdadeiras (V) ou falsas (F):

a) int x = 3;

b) y = y+1;

c) int x+3;

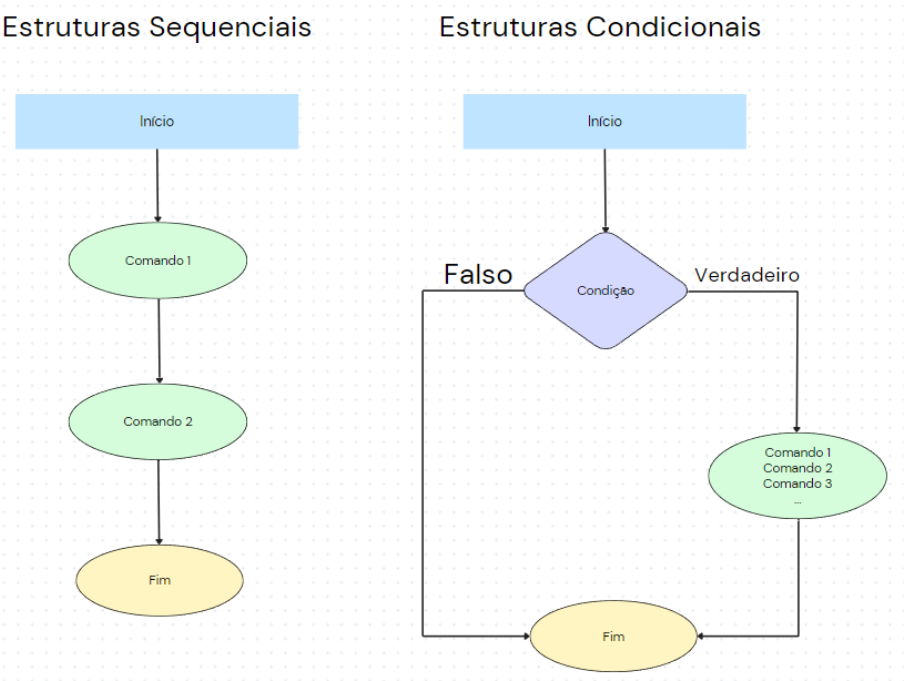
d) int A = 3, A é uma constante

e) printf("%.f", x);

f) 1 + ‘1’ = 2;

Estruturas condicionais

Neste capítulo, vamos introduzir as estruturas condicionais, elementos fundamentais no processo de escrita de código. Diferentemente das estruturas sequenciais, que seguem um fluxo linear de execução, as estruturas condicionais permitem que o programa tome decisões, alterando o fluxo com base no valor-verdade de uma condição.



As estruturas condicionais mais comuns nas linguagens de programação são o if (se) e o else (se não). Elas permitem ao programador estabelecer condições específicas que, quando atendidas, mudam a sequência de execução do programa.

Principais estruturas

If e else: Para começar, o comando if verifica se uma determinada condição é verdadeira. Se for, o bloco de código associado a essa condição será executado; caso contrário, o programa seguirá para a próxima instrução ou para o bloco alternativo definido pelo else.

Em alguns casos, podemos simplesmente optar por não utilizar o bloco else. Se quisermos apenas verificar uma condição e, independentemente do resultado, continuar a execução normal do programa com as instruções subsequentes, o bloco else pode ser inútil. No entanto, se houver a necessidade de executar comandos específicos quando a

condição do if não for satisfeita, o uso do else torna-se necessário. Dessa forma, o bloco else pode garantir que um conjunto alternativo de comandos seja executado antes que o programa continue seu fluxo normal.

Vejamos como escrever as estruturas condicionais if e else

```
if(condição)
{
    // Se a condição acima for verdadeira, então executa todos os códigos dentro
    das chaves do if

    Código 1;
    Código 2;
    Código 3;
        ...
}
else
{
    // Se a condição for falsa, então executa todos os códigos dentro das chaves do
    else
    Comando 4;
    Comando 5;
    Comando 6;
        ...
}
```

Uma estrutura bastante utilizada nas estruturas condicionais if e else é o aninhamento de condições, uma técnica em programação onde uma estrutura condicional (como um if) é colocada dentro de outra. Essa abordagem é especialmente útil quando precisamos avaliar várias condições que dependem umas das outras para que a resolução final tenha um valor esperado, permitindo uma lógica de decisão mais complexa.

Porém, aninhar muitas condições pode levar a um código de mais difícil legibilidade e compreensão. Isso é conhecido como "aninhamento profundo" (deep nesting), que deve ser evitado sempre que possível. Além disso, códigos muito aninhados podem se tornar difíceis de manter e modificar, especialmente se as condições mudarem ao longo do tempo.

Vejamos o uso de estruturas condicionais aninhadas:

```
if(condição 1)
{
   if(condição 2)
   {
      if(condição 3)
      {
         //Todas as condições são verdadeiras
         Comando 1;
            ...
      }
      else
      {
         //Ambas condições 1 e 2 são verdadeiras, mas a 3 não
         Comando 2;
            ...
      }
   }
}
```

Switch/case: A estrutura switch é uma forma de controle de fluxo de execução que permite executar diferentes partes do código com base no valor de uma variável ou expressão. É uma alternativa mais clara e organizada em relação ao uso de múltiplas instruções if-else, especialmente quando se tem várias condições a serem verificadas sob uma mesma variável.

A sintaxe do switch é bastante simples e direta. Primeiro, uma expressão é avaliada, e o resultado é comparado com os valores de cada case. Cada case é associado a um valor constante e, se a expressão corresponder a algum desses valores, o bloco de código correspondente a esse será executado. Para interromper a ordem de execução do do switch, utilizamos o comando break, evitando que o fluxo continue para os próximos casos.

Utilizamos como boa prática como um último caso do switch o bloco “default” (opcional), este serve como uma forma de garantir que, caso nenhuma condição anterior seja

verdadeira, ele será então executado. Ele funciona semelhante à estrutura else, da estrutura if-else.

Aqui está o uso da estrutura switch:

```
switch(nomeVariavel)
{
    case 1:
      Comando 1;
          ...
      break;
    case 2:
      Comando 2;
          ...
      break;
    default:
      Comando 3;
          ...
      break;
}
```

Estrutura condicional ternária: Além das formas convencionais (if/else) de representar as estruturas condicionais, temos também o operador condicional ternário em linguagem C, caracterizado pela seguinte forma: Nessa situação, temos uma condição inicial, antes do sinal de interrogação (?). Caso a mesma for verdadeira, executamos o código 1, antes dos dois pontos (:), caso contrário, executamos o código 2, depois dos dois pontos;

```
// Ex:
// Estrutura condicional ternária
(condição) ? código1 : código 2;
```

“Se p, então q, se não, então r” (Sentença condicional simples).

Operadores relacionais: Operadores relacionais são utilizados em linguagens de programação para comparar dois valores. O resultado de uma comparação feita

por um operador relacional é sempre um valor lógico/booleano, ou seja, verdadeiro (true) ou falso (false). Os principais operadores relacionais são:

- Igual (==): Verifica se dois valores são iguais.
  Ex: a==b retorna verdadeiro se a for igual a b.
- Diferente (!=): Verifica se dois valores são diferentes.
  Ex: a!=b retorna verdadeiro se a for diferente de b.
- Maior que (>): Verifica se o valor à esquerda é maior que o valor à direita.
  Ex: a>b retorna verdadeiro se a for maior que b.
- Menor que (<): Verifica se o valor à esquerda é menor que o valor à direita.
  Ex: a<b retorna verdadeiro se a for menor que b.
- Maior ou igual (>=): Verifica se o valor à esquerda é maior ou igual ao valor à direita.
  Ex: a>=b retorna verdadeiro se a for maior ou igual a b.
- Menor ou igual (<=): Verifica se o valor à esquerda é menor ou igual ao valor à direita.
  Ex: a<=b retorna verdadeiro se a for menor ou igual a b.

Operadores lógicos: Operadores lógicos são usados para combinar expressões lógicas/booleanas ou realizar operações lógicas em condições. Eles retornam um valor lógico, ou seja, verdadeiro (true) ou falso (false), e são essenciais em estruturas condicionais mais complexas, permitindo a combinação de múltiplas condições. Os principais operadores lógicos são:

- E lógico (&&): Retorna verdadeiro apenas se ambas condições forem verdadeiras.
  Ex: (a>b) && (a>c) retorna verdadeiro apenas se a for maior que b e a for maior que c.
- Ou lógico (||): Retorna verdadeiro pelo menos uma das condições for verdadeira.
  Ex: (a>b) || (a>c) retorna verdadeiro se a for maior que b ou a for maior que c ou ambos os casos.
- Negação lógica (!): Inverte o valor verdade de uma condição.
  Ex: !(a>b) retorna verdadeiro se a não for maior do que b

A tabela verdade pode ser um exemplo de usos desses operadores...

P	Q	$P \wedge Q$	$P \vee Q$	$P \veebar Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
V	V	V	V	F	V	V
V	F	F	V	V	F	F
F	V	F	V	V	V	F
F	F	F	F	F	V	V

Fonte: Marcelo Eustáquio

Vejamos um exemplo de uso da estrutura condicional, com um programa que lerá dois valores inteiros para duas variáveis a e b e logo em seguida fará a verificação se o conteúdo da variável a é maior que o conteúdo da variável b:

```
// Pseudocódigo
Algorithm: maiorNumero
    //Declarando variáveis
    declare:
        a, b: integer

    //Lendo valores para a e b
    read a, b;


 //Verificando se a é maior que b
    if(a>b)
    {
        //Se for maior, então imprima a (valor contido na variável) é maior que b (valor contido na variável)
        write(a, "é maior que ", b);
    }
    else
    {
        //Se não for maior, então imprima b (valor contido na variável) é maior que a (valor contido na variável)
        write(b, "é maior que ", a);
    }
end maiorNumero
```

```
//Linguagem C
int main()
{
    //Declarando variáveis
    int a, b;

    //Lendo valores para a e b
    scanf("%d%d", &a, &b);

    //Verificando se a é maior que b
    if(a>b)
    {
        //Se for maior, então imprima a (valor contido na variável) é maior que b (valor contido na variável)
        printf("%d é maior do que %d.", a, b);
    }
    else
    {
        //Se não for maior, então imprima b (valor contido na variável) é maior que a (valor contido na variável)
        printf("%d é menor do que %d.", a, b);
    }
    return 0;
}
```

Exercícios propostos – Estruturas condicionais

01) Escreva um algoritmo que resolva a equação do segundo grau da forma geral: $a^2 + bx + c = 0$, em que os coeficientes a, b e c são fornecidos pelo usuário. O programa deve calcular as raízes x' e x" utilizando a fórmula de bhaskara. Se o valor de Δ for negativo, o programa deve exibir a mensagem "Solução inválida", inicando assim que não há raízes reais. Entretanto, caso contrário, imprima x' e x".

Exemplo de caso de teste

Entrada	Saída
1 -6 5	x’ = 5 x” = 1

02) Crie um programa que leia três números inteiros e determine qual é o maior entre eles.

Exemplo de caso de teste

Entrada	Saída
12 25 8	25

03) Crie um programa que leia três números inteiros e ordene-os em ordem crescente

Exemplo de caso de teste

Entrada	Saída
4 38 12	4, 12, 38

04) Faça um algoritmo que leia duas notas de um aluno (n1 e n2) e informe se ele foi aprovado, reprovado ou está em recuperação. Considere que para uma aluno ser aprovado, sua média tem que ser maior ou superior a 6.0. Caso contrário, se o aluno estiver uma média entre 4.0 e 5.9, o aluno está de recuperação. Entretanto, caso a nota seja inferior a 4.0, então o aluno está reprovado.

Exemplo de caso de teste

Entrada	Saída
6.5 4.3	“reprovado"

05) Desenvolva um código que calcule o imposto de renda de uma pessoa com base em seu salário, de acordo com a tabela abaixo:

Salário	Percentual de imposto (%)
Até R$ 1903,98	Isento
De R$ 1903,99 até R$ 2826,65	7,5%
De R$ 2826,66 até R$ 3751,05	15%

De R$ 3751,06 até R$ 4664,68	22,5%
Acima de R$ 4664,68	27,5%

Exemplo de caso de teste

Entrada	Saída
3759.39	“Imposto a ser pago: R$ 845.86”

06) Uma loja solicitou a criação de um código que apresente quatro opções de produtos ao cliente e pergunte se ele gostaria de comprar algum deles. Se o cliente optar por adquirir o produto atual, será necessário solicitar a quantidade desejada. Os produtos escolhidos serão então adicionados ao carrinho de compras. Ao final do processo, o sistema deverá calcular e exibir o valor total a ser pago pelo cliente, com base nas opções e preços a seguir:

Produto	Preço
1	R$ 10,00
2	R$ 20,00
3	R$ 50,00
4	R$ 100,00

Exemplo de caso de teste

Entrada	Saída
1 2 0 0 0	“Valor total: R$ 20.00”

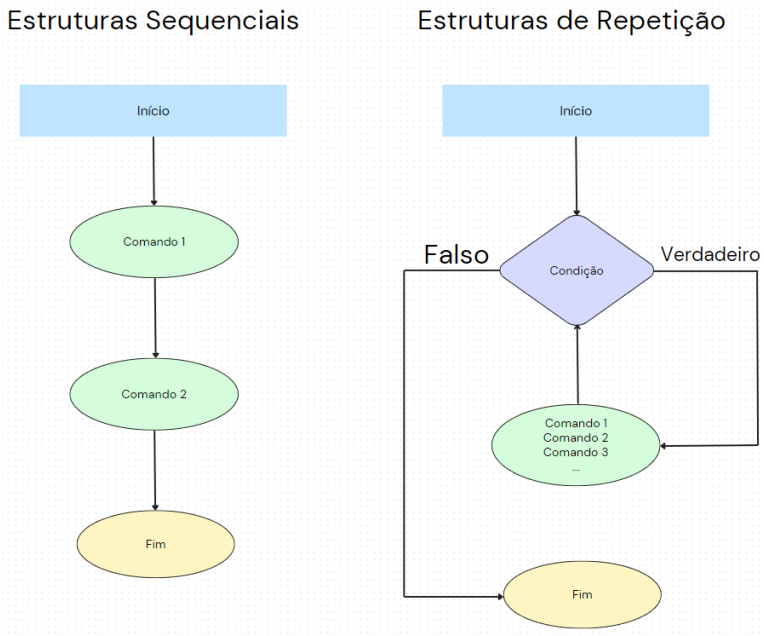
Estruturas de repetição

Nesta parte da nossa apostila, iremos dar início ao estudo das estruturas de repetição, artifícios fundamentais na maioria dos programas que iremos fazer a partir de agora. É de extrema importância ter entendido todos os conteúdos anteriores para darmos continuidade e que o aprendizado continue fluido daqui para a frente.

Para começarmos, estruturas ou laços de repetição são basicamente mecanismos que nos permite rodar uma ou mais instruções idênticas várias vezes. Dessa forma, isso facilita muito nossas vidas quando o assunto é tornar o código mais curto e eficiente, para trechos de códigos que executam as mesmas coisas mais de uma vez.

Vamos supor que queremos ler as notas de 30 alunos. Podemos simplesmente adicionar 30 vezes em nosso programa uma linha com o comando scanf, entretanto isso seria extremamente inviável. Por quê não apenas utilizar de uma estrutura que repita esse mesmo código o número de vezes que desejarmos? Pois bem, com essa finalidade surgem os laços de repetição.

Assim como as estruturas condicionais, os laços de repetição também rompem o fluxo de execução padrão quando um programa é executado, tornando-o uma repetição enquanto uma determinada condição é executada



No geral, todas estruturas de repetição apresentam uma condição de continuidade, isto é, enquanto essa condição se torna verdadeira, o programa entrará na estrutura da repetição, executando todos os comandos escritos. Entretanto, uma vez que a mesma

condição for falsa, o laço é interrompido, continuando o fluxo de execução normal com os próximos comandos abaixo dele.

Principais estruturas

for: A primeira estrutura de repetição que iremos estudar é o for (ou “para”), a qual possui três parâmetros a serem passados em sua função.

Utilizamos o for, na maioria das vezes, quando sabemos quantas vezes um código deve ser rodado.

Vejamos a sintaxe da estrutura *for* a seguir

```
//Pseudocódigo
for i<-0 to num step 1 do
   Comando 1;
   Comando 2;
   Comando 3;
        ...

//Linguagem C
for(inicialização; condição de continuidade; incremento/decremento)
{
   Comando 1;
   Comando 2;
   Comando 3;
        ...
}
```

- inicialização: A inicialização ocorre uma única vez, antes do laço começar. Ela serve para inicializar uma ou mais variáveis que serão utilizadas no controle do laço.

  Na maioria dos casos, a variável de controle do laço é um contador (i, j, k...) que começa de um determinado valor inicial. Para inicializarmos uma variável no laço, podemos declará-la dentro dessa estrutura ou fora, ficando int i=0 ou i=0 (caso já tenha sido declarada anteriormente).
- condição de continuidade: A condição de continuidade é verificada cada vez antes de entrar no laço. Somente se a condição apresentada for verdadeira, o bloco de

código dentro do laço é executado. Caso contrário, o laço é imediatamente interrompido e a execução do programa continua normalmente após o bloco de repetição.

Geralmente, quando temos um contador, a condição de continuidade é definida pelo número de vezes que queremos que o código seja repetido. Por exemplo, caso queremos que um código rode 10 vezes, teremos i<10 (com i inicializando com 0, pois de 0 a 10 temos 11 números, mas queremos ir até 10, então quando chegar no 10 ele não entra mais no loop) ou i<=10 (com i inicializando com 1, pois de 1 a 10 temos 10 números, então devemos chegar até o 10, caso o i for maior que o 10, ele sai da repetição).

- incremento/decremento: São alterações que as variáveis do laço recebem, as quais podem ser incrementos nos valores que cada uma guarda ou decrementos.

  É possível descrever um incremento como nomeDaVariavel+=1 (incrementa em 1 no valor armazenado na variável) ou decremento como nomeDaVariavel-=1 (decrementa em 1 no valor armazenado na variável).

  A fim de padronizar nossa escrita de código, o incremento e decremento será escrito como nomeDaVariavel++ e nomeDaVariavel--, respectivamente, que significam as mesma operação, porém de forma mais simplificada. Portanto, fica ao seu critério qual opção utilizar, pois ambas representam a mesma coisa.

while: Além da estrutura de repetição for, existe também a estrutura while (ou "enquanto"), que permite a execução do código contido nela enquanto uma condição específica for verdadeira.

Utilizamos o while principalmente quando não temos certeza de quantas vezes as instruções devem ser repetidas.

Vejamos a sintaxe da estrutura *while* a seguir:

```
//Pseudocódigo e Linguagem C são escritos basicamente da mesma forma
while(condição de continuidade)
{
   Comando 1;
   Comando 2;
   Comando 3;
      ...
```

```
}
```

Observe que, neste caso, não temos outros parâmetros a serem definidos além da condição de continuidade, já que o loop será executado um número indeterminado de vezes.

Na estrutura de repetição while, temos uma técnica de programação muito útil chamada “flag”. Uma flag define um ponto de parada de um código sem sabermos o número de vezes que o mesmo repete. Utilizamos um sinalizador (flag), que indica que devemos sair da repetição caso o encontremos, ou seja, é o ponto de parada. Caso contrário, permanecemos no laço de repetição.

Para criar uma flag, declaramos uma variável, geralmente inteira ou booleana, com um valor iniciado igual ao da condição de continuidade do laço de repetição while.

```
//Flag – Linguagem C
//Ex1:
scanf(“%d“, &num); //Lendo o valor de num
while(num!=0) //Enquanto o número digitado for diferente de 0, o código executa
{
   Comando 1;
   Comando 2;
   Comando 3;
        ...
   scanf(“%d“, &num); //Lendo novamente o valor de num
}

//Ex2:
flag = 1;
while(flag!=0)
{
   if(condição)
   {
      Comando 1;
      Comando 2;
      Comando 3;
           ...
```

```
    }
    else
    {
        flag = 0;
    }
}
```

do while: Por fim, a útlima estrutura de repetição mais importante para conhecermos é a do while (ou "faça enquanto"). Esse laço de repetição se assemelha bastante com a estrutura while, porém com uma diferença crucial.

Usamos a estrutura do while quando queremos que o código sempre execute uma primeira vez, independente da condição que for informada, para apenas depois dessa, termos as verificações da condição. Portanto, a execução do bloco de código pela primeira vez é intrínseca.

Vejamos a sintaxe da estrutura do *do while* a seguir:

```
//Pseudocódigo e Linguagem C são escritos basicamente da mesma forma
do
{
    Código 1;
    Código 2;
    Código 3;
        ...
}while(condição de continuidade);
```

Conhecidas as principais estruturas de repetição *for*, *while* e *do while*, vamos para um exemplo prático de código utilizando cada uma:

Nosso programa terá que ler as notas de 30 alunos e verificar se a nota do aluno atual está na média. Caso a nota seja maior ou igual a 6.0, imprimimos *"o aluno está na média",* caso contrário, imprimimos *"o aluno está abaixo da média"*.

```
//Estruturas de repetição – Linguagem C
// Estrutura for
for(int i=0; i<30; i++)
{
```

```
    scanf("%f", &nota);
    if(nota>=6.0)
    {
        printf("O aluno esta na media.");
    }
    else
    {
        printf("O aluno esta abaixo da media.");
    }
}

// Estrutura while
int i = 0;
while(i<30)
{
    scanf("%f", &nota);
    if(nota>=6.0)
    {
        printf("O aluno esta na media.");
    }
    else
    {
        printf("O aluno esta abaixo da media.");
    }
    i++;
}

// Estrutura do while
i = 0;
do
{
    scanf("%f", &nota);
    if(nota>=6.0)
    {
        printf("O aluno esta na media.");
```

```
    }
    else
    {
        printf("O aluno esta abaixo da media.");
    }
    i++;
}while(i<30);
```

Assim como nas estruturas condicionais, as estruturas de repetição também podem ser aninhadas. O aninhamento de laços permite que um laço seja executado dentro de outro, criando uma relação de dependência entre si. O laço mais externo controla o número total de iterações e aguarda a execução completa dos laços mais internos a cada iteração. Somente após a conclusão de todas as iterações dos laços interno é que o laço externo prossegue para sua próxima iteração.

Por exemplo, temos:

```
// Estrutura aninhada usando for – Linguagem C
for(int i=0; i<5; i++)
{
    for(int j=0; j<5; j++)
    {
        printf("%d", j);
    }
    printf(“\n”);
}
```

Saída:

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

Em resumo, o aninhamento de laços é uma ferramenta bastante útil que permite executar operações mais complexas, possibilitando a manipulação de estruturas de dados multidimensionais, como matrizes, ou a realização de processos repetitivos de forma eficiente e organizada. No entanto, aninhar laços pode ser de certa forma uma solução que reduz o desempenho de nosso programa, aumentando sua complexidade.

Exemplo - Fatorial: Vamos então resolver um exercício que, dado um número inteiro, calcule o fatorial do mesmo, com os comentários adequados para cada parte do código:

```
//Calculando o fatorial de um número – Linguagem C
//Linguagem C
#include <stdio.h>

int main()
{
    //Declarando variáveis
    int num;
    long long int fat = 1;

    /*
        obs: se inicia o fatorial com 1 pois caso o número escolhido pelo usuário for
        0 ou 1, não entraremos no bloco da repetição pois a inicialização, no caso o
        valor inteiro 2, não é menor ou igual ao número (0 ou 1), logo teremos todo o
        bloco da repetição ignorado, tendo o fatorial resultando em 1 apenas.
    */

    //Lendo o número a ser calculado seu fatorial
    scanf("%d", &num);

    //Caso i<= num, ele executará o que há dentro da repetição e incrementará em 1
    a variável i
    for(int i=2; i<=num; i++)
```

```
    {
        //Calculando o fatorial do número
        fat = fat*i;
    }
    //Imprimindo o resultado do fatorial do número
    printf("%d", fat);
    return 0;
}
```

A fim de facilitar nossa escrita, podemos reescrever o cálculo do fatorial da seguinte forma: fat *= i, que significa a mesma coisa de fat = fat*i, ou seja, é o valor já acumulado calculado anteriormente multiplicado pelo i incrementado.

Exercícios propostos – Estruturas de repetição

01) Desenvolva um algoritmo que, lido um N (número inteiro), calcule a soma dos números de 1 até o N.

Exemplo de caso de teste

Entrada	Saída
5	15

02) Um criador de gado enfrenta dificuldades para identificar qual dos seus bois apresenta o menor e o maior peso. Você foi contratado por ele para desenvolver um programa que, dada uma sequência indefinida de bois, determine o peso do boi mais leve e do boi mais pesado registrados.
Dados: A entrada de bois se encerra quando o peso do boi atual for zero

Exemplo de caso de teste

Entrada	Saída
300	Menor peso: 160 kg

160 430 0	Maior peso: 430 kg

03) Codifique uma solução para o seguinte problema: Dada uma turma de um certo número de alunos, lido através do teclado, leia a matrícula de cada aluno da turma, juntamente as notas n1 e n2, correspondentes às suas provas. Determine se o aluno está de recuperação ou foi aprovado, considerando que um aluno aprovado deve ter média igual ou maior que 6.0. Por fim, determine quantos alunos da turma foram reprovados e aprovados e qual foi a maior e a menor média da turma. A leitura de dados deve parar quando a matrícula do aluno for nula.

Exemplo de caso de teste

Entrada	Saída
12345 7.0 8.0 23456 5.5 6.0 34567 6.5 5.0 45678 8.0 9.0 56789 4.0 5.0	Total de alunos aprovados: 3 Total de alunos reprovados: 2 Maior média: 8.5 Menor média: 4.5

04) Crie um programa que leia uma número e verifique quantos números são pares de 1 até esse número.

Exemplo de caso de teste

Entrada	Saída
10	5

05) Elabore um programa em C que solicite ao usuário a entrada de um número inteiro. Em seguida, calcule e exiba a tabuada de multiplicação desse número, variando de 1 a 10.

Exemplo de caso de teste

Entrada	Saída
9	9 x 1 = 9 9 x 2 = 18 9 x 3 = 27 9 x 4 = 36 9 x 5 = 45 9 x 6 = 54 9 x 7 = 63 9 x 8 = 72 9 x 9 = 81 9 x 10 = 90

Estrutura de dados homogênea unidimensional

A partir deste momento, após estudarmos as estruturas de repetição, iremos abordar o tema: Estrutura de dados homogênea unidimensional.

Essa estrutura também pode ser chamada de vetor ou array, em inglês, que nada mais é que um conjunto de elementos do mesmo tipo organizados em uma única dimensão.

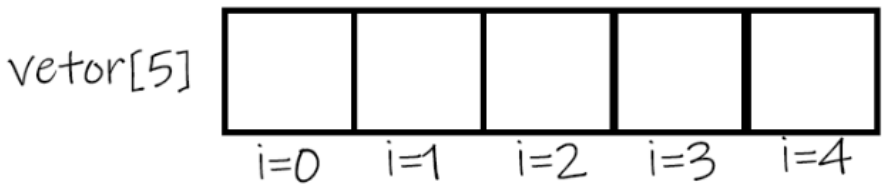
Os elementos nessas estruturas precisam ser sempre do mesmo tipo (caractere, número inteiro ou número decimal), onde cada elemento é localizado sequencialmente na memória. É importante ressaltar que um vetor, neste momento, tem tamanho fixo, ou seja, declaramos o tamanho dessa estrutura e ele permanece inalterado.

Cada vetor possui um tamanho, definido entre colchetes “[]” em sua declaração, esse tamanho representa quantos espaços serão alocados para que ele possa armazenar valores dentro dos mesmos.

Vamos supor que queremos um conjunto de 30 notas, de cada aluno de uma turma. Qual seria a solução mais viável para esse problema? Criar 30 variáveis para cada nota de cada aluno dessa turma? Não! Podemos utilizar um vetor, por exemplo, notas_algoritmos[30], o qual o mesmo será capaz de armazenar até 30 notas dos alunos da disciplina de algoritmos.

Cada elemento de um array pode ser acessado por um índice, que nada mais é do que uma variável inteira representando a posição do elemento dentro do array. Esses índices são, em geral, nomeados como i, j, k, etc., e sempre começam em 0, o que significa que a primeira posição do array é de índice 0. Portanto, para um vetor com 5 elementos, o último índice será 4, já que os índices variam de 0 a 4, totalizando 5 posições. Portanto, se quisermos acessar o elemento n de um vetor, sua posição será n-1.

Vejamos a representação visual de um vetor para facilitar o entendimento:



No exemplo acima temos:

Elemento 1 -> i = 0

Elemento 2 -> i = 1

Elemento 3 -> i = 2

Elemento 4 -> i = 3

Elemento 5 -> i = 4

Essa forma de indexação facilita o acesso direto aos elementos que desejamos, tornando os vetores ideais para cenários em que é necessário percorrer elemento por elemento ou acessar dados de forma rápida e eficiente. Entretanto, é extremamente importante saber se estamos lidando de fato com uma posição existente no vetor ou estamos tentando acessar uma memória que não foi alocada para este, o que pode gerar um comportamento indesejado por estamos acessando algo temporário da memória.

O tamanho de um vetor geralmente é definido no momento da sua declaração, o que significa que o número de elementos que ele pode armazenar é fixo durante a execução do programa, embora algumas linguagens permitam a criação de vetores dinâmicos que podem crescer ou diminuir conforme a necessidade, mas não iremos nos aprofundar em alocação dinâmica nesta apostila.

Declarando um vetor: Para declarar um vetor, iremos utilizar a mesma ideia da declaração de variáveis, indicando seu tipo de dado (char, int, float, double, ...) e seu identificador.

```
//Declarando um vetor
int vet1[5]; //Vetor do tipo inteiro, com 5 posições reservadas na memória RAM
float vet2[10]; //Vetor do tipo float, com 10 posições reservadas na memória RAM
```

É importante saber que também é possível declarar um vetor, inicializando em cada posição do mesmo um elemento correspondente ao seu tipo, por exemplo:

```
//Declarando um vetor e inicializando-o com elementos
int vet1[5] = {10, 20, 30, 40, 50};
```

Nesse caso, a primeira posição receberá o valor 10, a segunda 20 e assim por diante.

Por fim, podemos ainda declarar um vetor nulo, o qual todas suas posições inicialmente possuem zero como valor.

```
//Declarando um vetor nulo
int vet1[5] = {0};
```

Nesse caso, o número zero será atribuído à todas as posições.

Acessando uma posição do vetor: Para acessarmos uma posição de um vetor, podemos fazer isso de duas maneiras: diretamente, utilizando uma constante numérica inteira, ou por meio de uma variável numérica inteira que represente o índice.

No primeiro caso, usamos o número da posição diretamente, por exemplo:

```
//Atribuindo 5 à sexta posição do vetor
//Usando uma constante numérica
vet[número da posição] = 5;

//Usando uma variável numérica inteira
i = 5;
vet[i] = 5;
```

No primeiro caso, o valor 5 será atribuído à sexta posição do vetor (já que os índices começam em zero). No segundo caso, o valor 5 também será armazenado na posição indicada pela variável i.

Isso oferece flexibilidade para acessar diferentes posições de um vetor durante a execução de um programa, a qual podemos percorrer um vetor como veremos a seguir.

Entretanto, em ambos casos é importante sempre certificar-se de que estamos de fato acessando uma posição válida do vetor, para evitar acesso fora dos limites (out-of-bounds access).

Comando de entrada e saída com vetores: Assim como as variáveis simples, a leitura e impressão de valores de um vetor funciona de forma semelhante. Na linguagem C, usamos as mesmas funções scanf e printf para ler e exibir valores, respectivamente. A única diferença é que precisamos especificar a posição (índice) do vetor que desejamos acessar. Vejamos alguns exemplos:

```
//Lendo um valor para uma posição do vetor
//Pseudocódigo
read vet1[5]
read vet2[i]

//Linguagem C
scanf(“%d”, &vet1[5]);
scanf(“%d”, &vet1[i]);

//Imprimindo o valor de uma posição de um vetor
//Pseudocódigo
write vet[5]
write vet[i]

//Linguagem C
printf(“%d”, vet1[5]);
printf(“%d”, vet2[i]);
```

Observe que tanto na leitura quanto na impressão, o acesso ao vetor é feito com base no índice, que indica a posição que desejamos manipular.

Percorrendo um vetor: Neste tópico, iremos aprender a percorrer um vetor utilizando as estruturas de repetição estudadas anteriormente. Vamos supor que queremos percorrer um vetor linearmente, uma posição seguida da outra:

Primeiramente, vamos definir o índice (como i, j, k, etc) do vetor sendo o valor inicializado no laço da repetição como primeiro argumento da estrutura. Nesse caso, inicia-se o mesmo com o valor zero, uma vez que a primeira posição de um vetor, como vemos, é sempre zero.

Em seguida, definimos até qual posição do vetor iremos percorrer. Se quisermos percorrer até a última, por exemplo, precisamos que o índice vá apenas até a posição n-1 do vetor, pois a contagem de posições se inicia em zero, não em um como estamos acostumados. Por exemplo, queremos percorrer um vetor de cinco posições, a condição para continuarmos a repetição é que o índice deve ser menor que o cinco, totalizando cinco repetições no total, uma vez que ele vai apenas até o índice quatro.

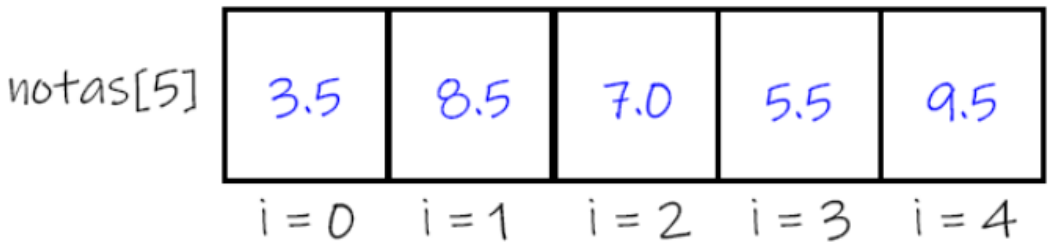
Por fim, a cada iteração é acrescido uma unidade no valor do contador, para que na próxima rodada já tenhamos a nova posição do vetor a ser acessada.

Para representar melhor como isso funciona, vamos observar um exemplo prático. Suponha que temos um vetor que armazena as notas de cinco alunos em uma determinada disciplina. O vetor será declarado como float notas[5];. Agora, para preenchermos este vetor com as notas 5 notas, utilizamos um laço de repetição que percorre cada índice, permitindo que o usuário insira os valores correspondentes a cada aluno por meio do teclado, vejamos:

```
//Percorrendo um vetor e lendo as 5 notas
//Pseudocódigo
for i<-0 to 5 step 1 do
    read notas[i]

//Linguagem C
for(int i=0; i<5; i++)
{
    scanf("%f", &notas[i]);
}
```

Portanto, no final quando o usuário digitar corretamente as notas, teremos um vetor como cada uma armazenada em sua devida posição, como por exemplo:



Exemplo – Ordenação de vetor: Vamos então fazer um exercício que, dado um vetor de números inteiros (não ordenado), ordene-o em ordem crescente.

```
//Ordenação de vetor
//Pseudocódigo
for i<-0 to (tam-1) step 1 do
```

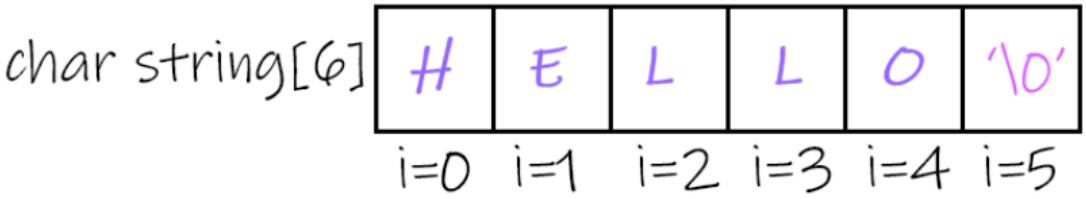
```
{
    pos_menor <- i;
    for i<-i+1 to (tam) step 1 do
    {
        if(A[j]<A[pos_menor])
        {
            pos_menor <- j;
        }
    }
    aux <- A[i];
    A[i] <- A[pos_menor];
    A[pos_menor] <- aux;
}

//Linguagem C
for(int i=0; i<tam-1; i++)
{
    pos_menor = i;
    for(int j=i+1; j<tam; j++)
    {
        if(A[j]<A[pos_menor])
        {
            pos_menor = j;
        }
    }
    aux = A[i];
    A[i] = A[pos_menor];
    A[pos_menor] = aux;
}
```

Sequência de caracteres (String): Durante todo o curso, estudamos os principais tipos de variáveis existentes, neles vimos que podemos criar uma variável do tipo char, que armazena dados do tipo caractere. Entretanto, nosso conhecimento estava limitado

apenas a um caractere único, ou seja, caso quiséssemos criar uma frase iríamos precisar de outro conceito: o vetor de caracteres (string).

Uma string é uma sequência de caracteres que pode incluir letras, números, espaços e símbolos, ou seja os caracteres em geral como já vimos, onde cada espaço do nosso vetor será armazenado por um único caractere. Na linguagem C, strings são representadas como arrays de caracteres, onde o último caractere deve ser um caractere nulo (\0) para indicar o fim da string. Essa característica é fundamental, pois permite que funções que manipulam strings saibam onde a sequência termina. Dessa forma, vamos visualizar uma string dessa maneira, por exemplo:



- **Linguagem C – String:**

  Antes de estudarmos strings em linguagem C, é extremamente recomendado a inserção da biblioteca string.h em seu programa, para todas as funções funcionarem corretamente.

  Obs: Uma string possui especificador de formato %s na leitura e impressão da mesma.

  - Declarando uma string: Para declarar uma string, utilizamos da mesma técnica já conhecida para os vetores, entretanto, nos atentando que se o tamanho da palavra for n, teremos que ter n+1 espaços disponíveis para ela, uma vez que temos um caractere extra que indica o seu fim, no caso o próprio (\0).

```
//Declarando uma string (sem inicializa-la como caracteres)
char str[5];

//Declarando a string HELLO, acima (inicializando com caracteres)
char str1[6] = {'H', 'E', 'L', 'L', 'O', '\0'};
char str2[6] = "HELLO";
```

- Leitura de uma string: Diferente dos vetores numéricos, não é preciso utilizar o índice de cada posição para ler a string de uma única vez, pois já se trata do próprio endereço. Logo, o que for digitado já será inserido linearmente, posição por posição do vetor.

  Entretanto, quando o assunto é ler uma string, existem diversas formas de fazer isso, irei abordar as principais, que são:

  - *Função scanf*: É possível ler uma string utilizando scanf, entretanto há duas maneiras específicas para isso.

    1. Leitura utilizando o especificador de formato %s e vetor como argumento: Dessa forma, é possível ler apenas uma palavra e armazená-la na string. Ou seja, tudo o que for digitado depois do espaço será perdido. É recomendada a leitura dessa forma apenas quando lemos uma única palavra.

       ```
       //Leitura utilizando scanf e %s
       scanf("%s", str);
       ```

    2. Leitura utilizando o especificador de formato %[^\n]s e vetor como argumento: Dessa forma, é possível ler uma linha inteira, até que o usuário aperte enter, que significa que foi encontrado o caractere (\n) que indica nova linha.

       ```
       //Leitura utilizando scanf e %[^\n]s
       scanf("%[^\n]s", str);
       ```

       Entratanto, essa forma não é tão indicada pois o scanf é uma função que está mais propensa a erros quando se trata de longas entradas do teclado.

  - *Função fgets*: Essa função é capaz de ler uma string inteira, até ser encontrado um caractere de nova linha (\n), no caso se o usuário digitar enter. Diferente da função scanf, a função fgets é a mais segura ao fazer a leitura de uma string, uma vez que o tamanho da string

deve ser sempre definido, o que evita problemas como estouro de buffer, por exemplo.

```
//Leitura utilizando fgets
fgets(nome da variável, tamanho da string, método de entrada);
```

A função fgets possui alguns parâmetros a serem passados, que são eles:
nome da variável: descreve o nome do vetor declarado no programa que será armazenada a leitura
tamanho da string: descreve o tamanho que a string tem, assim, evita estouro de buffer, garantindo que a leitura seja feita até, no máximo, o tamanho definido.
método de entrada: descreve como serão inseridos os caracteres para armazenamento no vetor, no nosso caso como não estamos tratando de outras rotinas, utilizamos stdin, o método de entrada por meio do teclado.

```
//Exemplo usando fgets
fgets(str, 100, stdin);
```

- Impressão de uma string: Para imprimir uma string, não vamos precisar de nada de novo, pois somente como nosso bom e velho printf já somos satisfeitos.

```
//Imprimindo uma string com printf
printf("%s", str);
```

- Principais funções de string.h para manipular strings: Diversas funções existem para que possamos manipular nossas strings, facilitando nosso trabalho e simplificando nossa lógica e estruturação do código. São elas:
  - *Função strlen:* Retorna o comprimento de uma string, incluindo o (\0)

```
//Atribuindo o comprimento de uma string a uma variavel
int a = strlen(string);
```

Portanto, supondo que no exemplo acima o usuário digitou “arvore” como string, o valor armazenado pela variável *tam* será 7.

- *Função strcpy:* Atribui o conteúdo de uma string a outra

```
//Sintaxe
strcpy(string de destino, string de origem);
```

Logo, supondo que quero copiar o conteúdo da string *str1* para outro vetor de caracteres *str2*, teremos:

```
//Copiando str1 para str2
char str1[100] = “Copy that!“;
char str2[100];
strcpy(str2, str1);
printf(“%s”, str2);
```

Saída: Copy that!

- *Função strcat:* Concatena duas strings

```
//Sintaxe
strcat(string que receberá a outra, string que irá se juntar)
```

Dessa forma, caso eu queira concatenar uma string *str1* com *str2, fica da seguinte maneira:*

```
//Concatenando str1 com str2
char str1[100] = “Hello, “;
char str2[100] = “world!”;
strcat(str1, str2);
```

```
printf(“%s”, str1);
```

Saída: Hello, world!

- *Função strcmp:* Compara um conteúdo de uma string com o conteúdo de outra. (Caso forem iguais, retorna 0)

```
//Sintaxe
strcmp(string 1, string 2);
```

Desse modo, vamos supor que tenho uma string *str1* e quero verificar se ela é igual à *str2*, então teremos:

```
//Verificando se str1 é igual a str2
char str1[100] = “ABC“;
char str2[100] = “CBA”;
if(strcmp(str1, str2)==0)
{
    printf(“São iguais!”);
}
else
{
    printf(“São diferentes!”);
}
```

Saída: São diferentes!

Exercícios propostos – Estrutura de dados homogênea unidimensional

01) Uma empresa pediu que você desenvolvesse um algoritmo que, dado um conjunto de senhas numéricas, o usuário de sua plataforma tentará fazer Login em sua plataforma com uma senha. O seu programa deve procurar na lista de senhas dessa empresa a senha digitada pelo usuário. Caso a mesma exista, você retorna ao usuário que o Login foi realizado com sucesso, caso contrário retorne que a senha digitada é inválida.
Dados: 50 senhas cadastradas na plataforma: 96, 204, 470, 575, 585, 593, 605, 681, 730, 806, 853, 880, 930, 1103, 1163, 1177, 1180, 1195, 1204, 1222, 1335, 1379, 1409, 1487, 1660, 1693, 1778, 1979, 2031, 2409, 2551, 2581, 2690, 2774, 2886, 3086, 3293, 3570, 3886, 3925, 4010, 4332, 4359, 4405, 4499, 4561, 4768, 4787, 4840, 4933.

Exemplo de caso de teste

Entrada	Saída
3086	Login realizado com sucesso!

02) Desenvolva um programa que leia uma string fornecida pelo usuário e, em seguida, exiba a string invertida. O programa deve manipular diretamente o vetor de caracteres para realizar a inversão.

Exemplo de caso de teste

Entrada	Saída
computador	rodatupmoc

03) Crie um código de um programa que leia dois vetores A e B de números inteiros, com 5 espaços ambos, armazene a soma deles em um terceiro vetor C e o imprima.

Exemplo de caso de teste

Entrada	Saída
10 30 20 39 85 11 30 49 81 5	21 60 69 120 90

04) Faça um programa que, lido um vetor de números inteiros de tamanho definido pelo próprio usuário, determine qual o menor número contido nele e sua posição no mesmo.

Exemplo de caso de teste

Entrada	Saída
10 1 32 4 12 -5 64 -7 3 40 10	Menor valor: -7 Posição: 6

05) Faça um programa que receba um nome completo do usuário e retorne a abreviatura deste nome. Não se devem abreviar palavras com dois caracteres ou menos, tais como as preposições: do, de, etc. A abreviatura deve vir separada por pontos. Ex: Paulo Jose de Almeida Prado. Abreviatura: P. J. de A. P

Exemplo de caso de teste

Entrada	Saída
Paulo Jose de Almeida Prado	P. J. de A. P

Estrutura de dados homogênea bidimensional

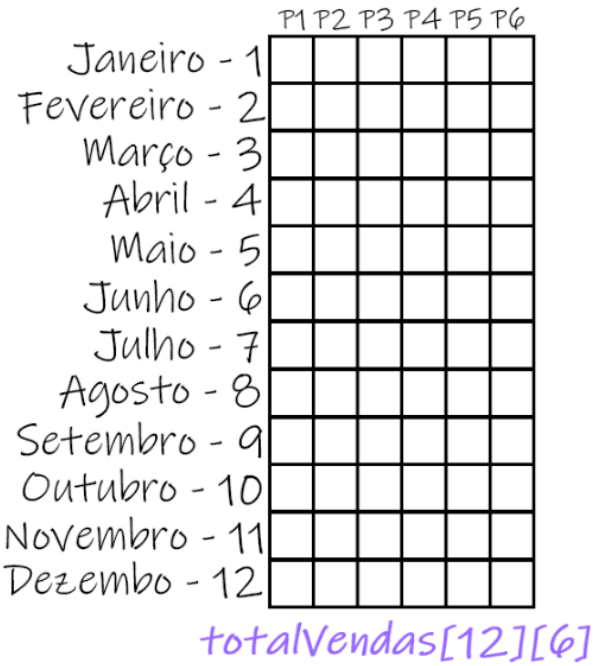
Neste capítulo iremos estudar nossa última estrutura de dados desse curso de algoritmo: a estrutura de dados homogênea bidimensional. Geralmente nos referimos a essa estrutura como matriz, o que será explicado a seguir.

Diferentemente dos vetores, que possuíam apenas uma direção, ou seja, uma única linha com elementos lineares, as matrizes na estrutura de dados homogênea bidimensional é caracterizada por possuir linhas e colunas, semelhante ao modelo matemático estudado no Ensino Médio.

Entretanto, como se trata de uma estrutura homogênea, nossos elementos contidos nela serão todos do mesmo tipo, assim como vimos nas estruturas unidimensionais e seu tamanho será fixo, pelo menos neste momento de nosso estudo, uma vez que não será abordado conceitos e estudos sobre alocação dinâmica.

Uma matriz possui um tamanho definido entre [][], por linhas e colunas, respectivamente, onde o primeiro parêntese representa o número de linhas e o segundo o número de colunas. Ou seja, o total de espaços alocados será a multiplicação do número de linhas pelo número de colunas. Por exemplo: queremos uma matriz com nove espaços alocados, então teremos que ter três linhas e três colunas declaradas para satisfazer essa necessidade.

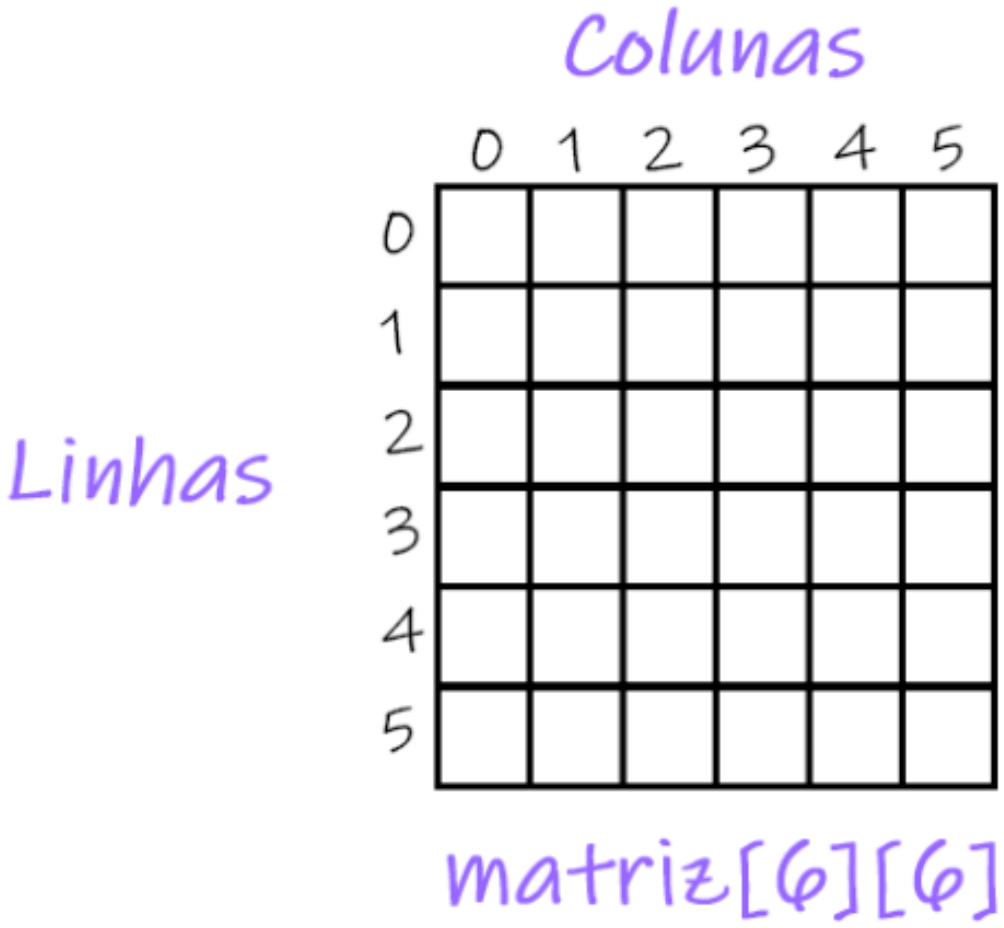
Um exemplo prático de matrizes que iremos abordar seria uma loja de produtos de moda (roupas, calçados etc), onde temos os 12 meses do ano e em cada mês o total de vendas de um certo produto. Para facilitar a visualização, teremos uma matriz parecida com isso:



Nesse exemplo, as linhas representam os meses de trabalho da empresa e as colunas representam os tipos de produtos vendidos por ela. Em cada espaço deve ser armazenado o número de vendas mensais desses produtos. Dessa forma, temos uma matriz 12x6, resultando em 72 espaços disponíveis para armazenar valores.

Semelhante aos vetores, cada elemento de uma matriz pode ser acessado diretamente através de dois índices: um para linha e outro para a coluna do elemento em questão, os mesmos nada mais são do que uma variável inteira representando a posição do elemento dentro da matriz. Por convenção, utilizamos i para representar o índice das linhas e j para representar o índice das colunas. É importante lembrar que ambos sempre começam em 0, o que significa que a primeira posição da matriz é de índice [0][0] e o último seria o de índice [num_linhas-1][num_colunas-1]. Dessa forma, para uma matriz com 4 elementos, ou seja matriz[2][2], o último índice para as linhas seria um, já que vai de 0 a 1, totalizando duas posições, e o último índice para as colunas de igual forma sendo um.

De forma geral, a representação de uma matriz na computação é feita da seguinte forma:



Dessa forma, temos:

Linha 0

Elemento 1 -> i = 0, j = 0

Elemento 2 -> i = 0, j = 1

Elemento 3 -> i = 0, j = 2

Elemento 4 -> i = 0, j = 3

Elemento 5 -> i = 0, j = 4

Elemento 6 -> i = 0, j = 5

...

Linha 5

Elemento 31 -> i = 5, j = 0

Elemento 32 -> i = 5, j = 1

Elemento 33 -> i = 5, j = 2

Elemento 34 -> i = 5, j = 3

Elemento 35-> i = 5 j = 4

Elemento 36-> i = 5, j = 5

Essa abordagem de indexação permite o acesso direto aos elementos desejados, para isso, é crucial verificar se estamos realmente acessando uma posição válida dentro da matriz, em vez de tentar acessar uma área da memória que não foi alocada, o que pode resultar em comportamentos indesejados ao acessarmos dados temporários da memória.

Declarando uma matriz: Para declarar uma matriz, iremos utilizar a mesma ideia da declaração de variáveis e de vetores, indicando seu tipo de dado (char, int, float, double, ...), seu identificador e os índices desejados. Como nosso caso é uma matriz, precisaremos de dois índices: linha e coluna.

```
//Declarando um vetor
int mat1[3][3]; //Matriz do tipo inteiro, com três linhas e três colunas
float mat2[2][5]; //Matriz do tipo float, com duas linhas e cinco colunas
```

Assim como os vetores, também é possível declarar uma matriz, inicializando em cada posição dela com um elemento correspondente ao seu tipo, por exemplo:

```
//Declarando uma matriz e inicializando-a com elementos
int mat1[3][3] = {{10, 20, 30}, {40, 50, 60}, {70, 80, 90}};
```

Para cada linha abrimos chaves e inserimos os elementos correspondentes às colunas da mesma. Como cada linha do exemplo acima possui três colunas, temos três elementos por linha.

Por fim, podemos ainda declarar uma matriz nula, o qual todas suas posições inicialmente possuem zero como valor.

```
//Declarando uma matriz nula
int mat[5][5] = {0};
```

Nesse exemplo, todas as posições da matriz inicializarão com o valor zero.

Acessando uma posição da matriz: Podemos acessar uma posição de uma matriz através da indicação dos índices de linha e coluna do elemento que desejamos. Assim como nos vetores, podemos fazer isso de duas formas: diretamente, utilizando números inteiros que representarão a linha e coluna do elemento, respectivamente, ou por meio de uma variável numérica inteira que represente esses índices.

Como exemplo para o primeiro caso, usamos o número da linha e coluna diretamente, desta forma:

```
//Atribuindo 5 ao elemento que está na linha X e coluna Y da matriz
//Usando constantes numéricas
mat[número da linha][número da coluna] = 5;

//Usando variáveis numéricas inteiras
i = número da linha;
j = número da coluna;
mat[i][j] = 5;
```

No primeiro caso, o valor 5 será atribuído por meio da indicação em relação à sua posição na matriz, fazendo dessa forma a partir de constantes numéricas referentes ao

número da própria linha e coluna correspondentes. No outro exemplo, o valor 5 também será guardado na posição indicada pela variável i (linha) e j (coluna) do espaço em questão.

Isso proporciona flexibilidade para acessar diferentes posições de uma matriz durante a execução do programa, permitindo percorrê-la, como será demonstrado a seguir.

Entretanto, também é importante certificar-se de que estamos de fato acessando uma posição válida da matriz, ou seja, que foi declarada anteriormente, pois dessa forma evita-se acessar espaços que não foram alocados na memória RAM, causando comportamentos indesejados durante a execução do programa

Comando de entrada e saída com matrizes: Os comandos de leitura e impressão de valores contidos em uma matriz funciona da mesma forma que os vetores, utilizando dos comandos padrão para realizar esses comandos, como na linguagem C, que são usadas as funções scanf e printf para ler e exibir valores, respectivamente. Entretanto, nesse caso precisamos especificar a posição do elemento desejado por meio do número da linha e coluna do elemento da nossa matriz desejado. Vamos ver os seguintes exemplos:

```
//Lendo um valor para uma posição da matriz
//Pseudocódigo
read mat1[3][2]
read mat2[i][j]

//Linguagem C
scanf(“%d”, &mat1[5][3]);
scanf(“%d”, &mat2[i][j]);

//Imprimindo o valor de uma posição de um vetor
//Pseudocódigo
write mat1[5][1]
write mat2[i][j]

//Linguagem C
printf(“%d”, mat1[0][2]);
printf(“%d”, mat2[i][j]);
```

Note que em ambos os casos (leitura e impressão), o acesso a posição desejada da matriz é realizado por meio dos índices de linhas e colunas do elemento em questão, indicando qual posição especificamente queremos manipular.

Percorrendo uma matriz: Percorrer uma matriz é um processo que utiliza laços de repetição de maneira semelhante ao que fazemos com vetores, mas com uma diferença importante: ao invés de um único laço para percorrer as posições, precisamos de dois laços aninhados, um para as linhas e outro para as colunas.

Para iniciar, o primeiro laço, que chamamos de loop mais externo, é responsável por iterar cada linha da matriz. Ele começa no índice 0, já que a contagem de uma matriz, assim como em um vetor, começa do zero. Isso significa que a primeira linha tem índice 0, a segunda linha tem índice 1, e assim por diante.

Depois, definimos a condição de continuidade do laço externo de acordo com o número total de linhas da matriz. Se a matriz tem, por exemplo, 5 linhas, o laço deve rodar enquanto o índice das linhas for menor que 5. Assim, ele vai percorrer as linhas de 0 a 4, totalizando cinco iterações. A cada passagem pelo laço externo, o índice da linha é incrementado, permitindo que avancemos para a próxima linha.

Dentro desse laço, colocamos outro laço de repetição, chamado de laço interno, que será responsável por iterar pelas colunas da linha atual. Esse laço começa percorrendo as colunas da linha em que o laço externo está posicionado, também começando do índice 0. Ele se comporta da mesma forma que o laço externo: percorre as colunas da matriz da primeira até a última, incrementando o índice a cada iteração. Note que nesse caso, fixamos a linha atual que estamos para percorrer cada coluna dela, para somente depois disso avançar para a linha seguinte.

Para representar melhor como isso funciona, vamos observar um exemplo prático no código implementando essa estrutura a seguir:

```
//Percorrendo uma matriz 3x4
//Pseudocódigo
for i<-0 to 3 step 1 do
   for j<-0 to 4 step 1 do
      read mat[i][j]
```

```
//Linguagem C
for(int i=0; i<3; i++)
{
    for(int j=0; j<4; j++)
    {
        scanf("%d", &mat[i][j]);
    }
}
```

Dessa forma, o exemplo acima faz o usuário inserir os 12 elementos da matriz, elemento por elemento de cada linha.

Exemplo – Loja de roupas: Tendo compreendido os conceitos fundamentais sobre matrizes, vamos fazer um programa que atenda nosso problema envolvendo matrizes visto brevemente antes. Primeiramente, uma loja possui os seguintes produtos à venda, cujos preços serão informados individualmente para cada tipo de produto: camisetas, calças e moletons. Queremos que, para cada mês do ano, onde é dado o total de vendas do mesmo, determine o total de venda para um ano (12 meses) e o valor total em R$ arrecadado pela loja. Vejamos um código que implemente uma solução para esse problema em linguagem C:

```
//Programa para a loja
#include <stdio.h>

int main()
{
    //Declarando variáveis
    int vendasAno[12][3], vendasTotais[3] = {0};
    double p1, p2, p3, valorTotal = 0;

    //Lendo os preços para cada tipo de produto da loja
    printf("Preco das camisetas: ");
    scanf("%lf", &p1);
    printf("Preco das calcas: ");
```

```
    scanf("%lf", &p2);
    printf("Preco dos moletons: ");
    scanf("%lf", &p3);

    //Iniciando a repetição para a leitura das vendas dos produtos por mês
    for(int i=0; i<12; i++)
    {
        //Mês atual
        printf("\nMes %d\n", i+1);

        //Para as camisetas
        printf("Camisetas vendidas: ");
        scanf("%d", &vendasAno[i][0]);
        vendasTotais[0] += vendasAno[i][0];
        valorTotal += vendasAno[i][0]*p1;

        //Para as calças
        printf("Calcas vendidas: ");
        scanf("%d", &vendasAno[i][1]);
        vendasTotais[1] += vendasAno[i][1];
        valorTotal += vendasAno[i][1]*p2;

        //Para os moletons
        printf("Moletons vendidos: ");
        scanf("%d", &vendasAno[i][2]);
        vendasTotais[2] += vendasAno[i][2];
        valorTotal += vendasAno[i][2]*p3;
    }
    //Imprimindo os resultados obtidos em um ano
    printf("\nTotal de vendas no ano:\n");
    printf("Camisetas: %d\n", vendasTotais[0]);
    printf("Calcas: %d\n", vendasTotais[1]);
    printf("Moletons: %d\n", vendasTotais[2]);
    printf("Valor total arrecadado no ano: R$ %.2lf\n", valorTotal);
}
```

Exercícios propostos – Estrutura de dados homogênea bidimensional

01) Desenvolva um algoritmo que multiplique duas matrizes, A e B, armazenando o resultado em uma terceira matriz, C. O algoritmo deve inicialmente verificar se o número de colunas da matriz A é igual ao número de linhas da matriz B. Caso essa condição não seja satisfeita, o programa deve exibir a mensagem "Solução inválida!" e encerrar. Se a condição for atendida, o programa deve proceder com a leitura dos elementos de ambas as matrizes e, em seguida, realizar o cálculo da multiplicação, armazenando os resultados na matriz C.

Exemplo de caso de teste

Entrada	Saída
3 2 2 3 1 2 3 4 5 6 7 8 9 10 11 12	27 30 33 61 68 75 95 106 117

02) Crie um algoritmo que leia uma matriz A de ordem 4x4 e calcule sua transposta, armazenando o resultado em uma nova matriz B. Imprima a matriz transposta.

Exemplo de caso de teste

Entrada	Saída
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160	10 50 90 130 20 60 100 140 30 70 110 150 40 80 120 160

03) Escreva um algoritmo que leia os elementos de uma matriz 2x3 e encontre o maior valor presente nessa matriz. Imprima o valor do maior elemento e sua posição na matriz (linha e coluna).

Exemplo de caso de teste

Entrada	Saída
1 53 320 530 3 724	724

04) Codifique uma solução de um algoritmo que leia uma matriz quadrada de ordem 4x4 e imprima os elementos da diagonal principal e da diagonal secundária.

Exemplo de caso de teste

Entrada	Saída
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	Diagonal principal: 1 6 11 16 Diagonal secundária: 4 7 10 13

05) Crie um código que leia uma matriz quadrada 4x4 e calcule a soma dos elementos que estão acima da diagonal principal. Imprima o valor da soma ao final.

Exemplo de caso de teste

Entrada	Saída
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	S = 36

Conclusão e agradecimentos

E bom... chegamos ao final deste curso introdutório de algoritmos vinculados à linguagem C, é com grande satisfação ter a oportunidade de escrever meu primeiro material sobre o assunto. Espero que todo conteúdo fornecido durante essa jornada tenha sido de alguma forma proveitoso, desde a introdução aos conceitos básicos até a exploração de estruturas e modelos mais avançados. Meu objetivo foi fazer com que, até mesmo quem nunca programou na vida, dê seus primeiros passos no mundo da programação e possa caminhar com suas próprias pernas de agora em diante. Como mencionei no início, sempre procure novos desafios e maneiras de solucionar seus problemas, vá além do que foi fornecido aqui. Entender como os algoritmos funcionam é fundamental para uma compreensão e maior aprendizado da lógica da programação no geral.

Espero que os exemplos práticos e exercícios propostos tenham instigado seu pensamento crítico e sua capacidade de resolver problemas. Lembre-se de que a programação é uma habilidade em constante evolução, e o aprendizado nunca termina. Continue praticando, experimentando e, acima de tudo, se divertindo enquanto explora novos conceitos e tecnologias.