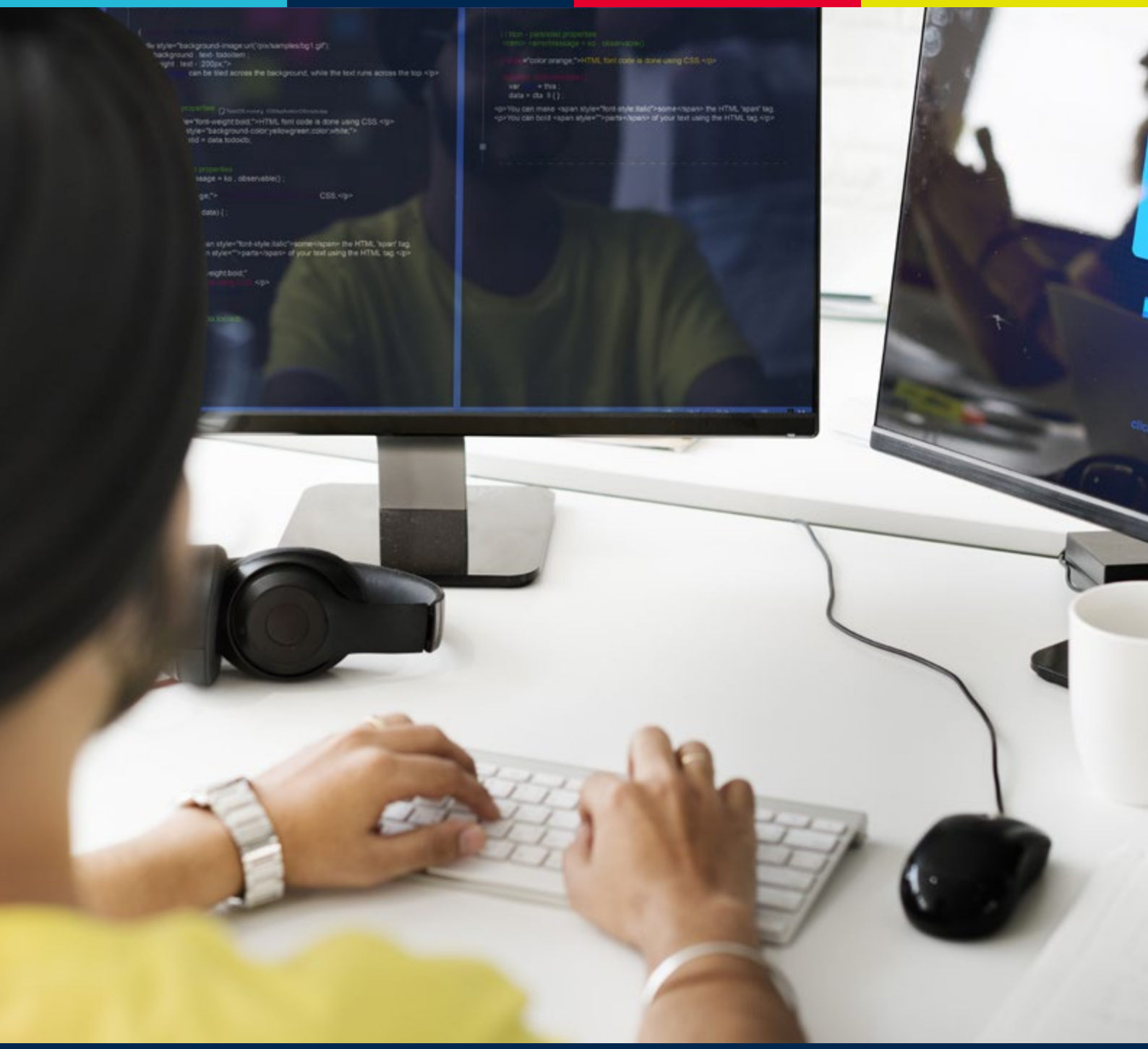


INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

UVA UNIJORGE

INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO

Texto de acordo com as normas do Novo Acordo Ortográfico da Língua Portuguesa.

AUTORIA DO CONTEÚDO
Claudio Fico Fonseca

PROJETO GRÁFICO
UVA

REVISÃO
Caroline Maganhi
Janaina Senna
Theo Cavalcanti
Lydianna Lima

DIAGRAMAÇÃO
UVA

F676 Fonseca, Claudio Fico.
Introdução à lógica de programação [recurso eletrônico] / Claudio Fico Fonseca. – Rio de Janeiro: UVA, 2020.

1 recurso digital (5061 MB)

Formato: PDF
ISBN 978-65-5700-021-2

1. Programação (Computadores). 2. Algoritmos. 3. Estruturas de dados (Computação). I. Universidade Veiga de Almeida. II. Título.

CDD – 005.1

Bibliotecária Adriana R. C. de Sá CRB 7 – 4049.
Ficha Catalográfica elaborada pelo Sistema de Bibliotecas da UVA.

SUMÁRIO

SUMÁRIO

APRESENTAÇÃO

A disciplina de **Introdução à Lógica de Programação** visa o entendimento da importância da aplicação de estruturas lógicas e sequenciais para o desenvolvimento de aplicações e pode ser usado em empresas, jogos e sistemas operacionais. Esta tecnologia, alinhada à linguagem C, permitirá o fortalecimento e o engrandecimento do negócio junto ao mercado globalizado, pois se trata de uma linguagem de uso atual e que é compartilhada em grandes softwares, jogos de ponta e/ou sistemas operacionais.

Consiste em um conjunto ordenado de ações que permitirá a execução de um determinado programa, atendendo a uma ou mais necessidades. Será trabalhada também a questão da identificação de possíveis erros de lógica de programação e como esses erros poderão ser tratados.

Dentro desse processo desenvolveremos o uso das estruturas de decisão, que auxiliarão na verificação de entrada de dados — ou seja, apenas dados válidos serão permitidos — e de uma série de condições para atender a demandas específicas.

Veremos também como o uso da estrutura de repetição permitirá a execução de uma ou mais rotinas de forma repetitiva até que uma determinada condição seja findada.

AUTOR

CLAUDIO FICO FONSECA

Graduado em Informática, Especialização em Marketing, MBA em Gestão Empresarial, MBA em Gerenciamento de Projetos, Mestre em Educação e Doutor em Gestão Educacional. Exerce na Universidade Veiga de Almeida – UVA os cargos de Diretor de Desenvolvimento Institucional, Coordenador do Curso de Sistemas da Informação – Presencial e EAD, coordenador do Curso de Análise e Desenvolvimento de Sistemas – EAD, Coordenador do Curso de Gestão da Tecnologia da Informação – EAD, Coordenador do Núcleo de Certificação Profissional e Internacional, Coordenador do MBA em Gestão Estratégica de TI. Exerceu o cargo de Diretor da Escola de Tecnologia da Informação e Inovação das Faculdades Integradas de Jacarepaguá – FIJ, o cargo de Diretor de Pós-Graduação e Extensão e Consultor de Tecnologia da Informação na Master Educacional, o cargo de Pró-Reitor Acadêmico e Diretor de Tecnologia do Grupo Educacional Anglo-Americano, o cargo de Vice-Reitor Adjunto, diretor de Certificação Profissional e gerente de TI do Centro Universitário da Cidade do Rio de Janeiro – UniverCidade. Atua como Conselheiro de Educação e Novas Tecnologias, Membro da Associação Brasileira de Educação, Membro Titular do Conselho Ibero-Americano, Avaliador Institucional e de Cursos da área de Informática e Cursos Superiores de Tecnologia do Ministério da Educação/Inep, Avaliador de Cursos do Guia do Estudante e Perito de Tecnologia da Informação da Justiça do Trabalho.

C. Lattes

UNIDADE 1

Fundamentos de lógica para a solução de problemas

INTRODUÇÃO

A partir de conhecimentos acerca dos fundamentos da programação estruturada — histórico, bibliotecas, tipos de dados e estrutura da linguagem —, serão obtidas bases sólidas para que haja o entendimento e a aptidão necessários para a estruturação e o desenvolvimento de solução no âmbito da programação.

OBJETIVO

Nesta unidade você será capaz de:

- Compreender os fundamentos da programação estruturada, o histórico, as bibliotecas, os tipos de dados e estrutura da linguagem C.

Como resolver problemas algorítmicos

Lógica de Programação é uma técnica que visa desenvolver **algoritmos** (sequências lógicas ordenadas) para atingir um conjunto de objetivos com base em regras pautadas na lógica matemática e em outras teorias clássicas oriundas da Ciência da Computação e que depois são adequadas para a Linguagem de Programação, utilizada a critério do programador para a construção do seu software.

> Um **algoritmo** nada mais é que uma **sequência não ambígua de instruções que é executada até que determinada condição seja atendida**.
>
> Especificamente, em matemática, podemos entender como a **constituição de um conjunto de processos e símbolos que os representam para que um determinado cálculo possa ser efetuado**.

Um algoritmo criado não necessariamente poderá representar um programa para ser executado em um computador, mas pode representar apenas as etapas e passos fundamentais para realizar uma determinada ação ou tarefa.

Sabemos que há diferentes tipos de algoritmos e estes poderão executar a mesma ação utilizando um conjunto distinto de instruções com mais ou menos tempo de execução. **A diferença pode estar na complexidade criada no desenvolvimento das instruções**.

O conceito teórico para um algoritmo foi formalizado no ano de 1936 pela famosa Máquina de Turing, que foi construída por Alan Turing e pelo cálculo lambda de Alonzo Church.

Dica

O conceito de algoritmo criado em 1936, durante a Segunda Guerra Mundial, pelo matemático britânico Alan Turing, foi considerado um marco na idade moderna. Após este evento, Turing ficou conhecido como o pai da ciência da computação. Para saber mais sobre esta história, indicamos o filme **O jogo da imitação**.

A fundamentação teórica para a estruturação de um **algoritmo** ou um **programa** é o argumento usado para a "busca da verdade" e suas reais fundamentações lógicas para a resolução de ações.

Um argumento é caracterizado como uma sequência de proposições, que compõem as sentenças afirmativas de uma forma ordenada e sequencial, que podem, dentro da sua estrutura, ser verdadeiras ou falsas. Uma delas é a conclusão do fato e as outras serão premissas.

As premissas sempre irão justificar a conclusão daquela situação.

O objetivo para um determinado argumento é de justificar uma afirmação ou dar propósitos plausíveis para uma certa conclusão obtida.

Dessa forma, é fundamental expor todas as hipóteses que levaram à verificação lógica como forma de convencimento, e demonstrar o correto fundamento do argumento.

Quando temos a necessidade de apontar a questão da veracidade de um argumento, as premissas serão consideradas provas fidedignas da verdade da conclusão do fato, do contrário não serão válidas. Válido será quando pudermos determinar que a conclusão é uma consequência lógica das premissas.

Importante

Vale destacar que há uma grandiosa diferença no que abrange os argumentos válidos e verdadeiros. Se em uma determinada frase, as premissas estão bem específicas e de forma que as frases tenham alguma conclusão, nem sempre teremos um resultado verdadeiro.

Podemos exemplificar uma argumentação válida com uma conclusão verdadeira:

1. **Todo golfinho é um mamífero.**
2. **Todo mamífero tem pulmões.**
3. **Logo, todo golfinho é um mamífero.**

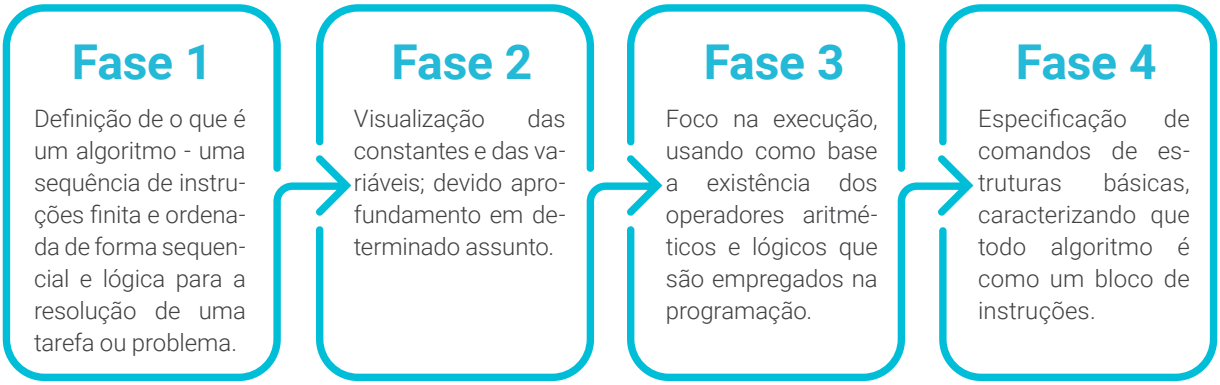
Porém, utilizando um exemplo como...

1. **Toda aranha tem quatro pernas.**
2. **Todo ser de quatro pernas tem asas.**
3. **Logo, toda aranha tem asas.**

... fica explícito que a lógica de programação foi utilizada. O referido argumento tem plena validade, mas esse não será um argumento verdadeiro, uma vez que a conclusão obtida será falsa.

Fases da Lógica de Programação

A referida metodologia possui quatro fases:



Nas questões pertinentes à lógica e à matemática, temos a **lógica proposicional ou Álgebra das Proposições**:

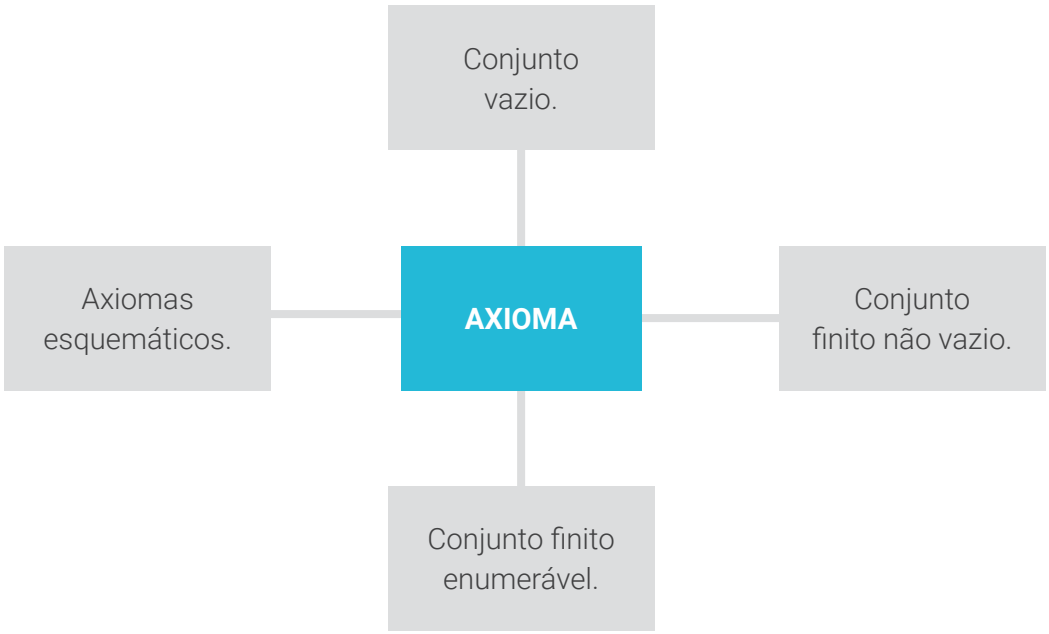
> É a maneira como um sistema formal cujas fórmulas visam representar a combinação de proposições atômicas, usando os "conectivos lógicos" e um sistema de "regras de derivação", que irão fazer com que algumas fórmulas possam ser caracterizadas como um "teorema" do sistema formal.

Importante

> Um cálculo será sempre exibido como um sistema formal, que irá determinar um conjunto de expressões sintáticas, um subconjunto distinto dessas expressões, e um conjunto de regras formais.

Na percepção de que o sistema formal tem como base ser um sistema lógico, as expressões devem ser, de forma imediata, interpretadas como as asserções matemáticas e as regras explícitas, naturalmente conhecidas como "regras de inferência" — normalmente as expressões impostas serão as preservadoras da verdade.

O conjunto estabelecido de axiomas poderá ser representado por:



A gramática formal irá definir as expressões e fórmulas caracterizadas pela linguagem proposta. Assim, podemos visualizar a semântica para definir o que é verdade, as suas devidas valorações e as suas interpretações.

Afinal, o que é uma proposição?

Uma proposição é uma sentença declarativa, seja ela determinada de forma afirmativa ou negativa, na qual poderemos atribuir um valor lógico:

- Verdadeiro – V.
- Falso – F

.

Vamos ver alguns exemplos?

I. Sentença declarativa afirmativa verdadeira

"A cidade de Brasília é a capital do Brasil."
Temos uma sentença declarativa expressa totalmente de forma afirmativa. Nesse caso, podemos também atribuir um valor lógico. Uma vez que a sentença é verdadeira, naturalmente o seu valor lógico será V.

II. Sentença declarativa negativa verdadeira

"O Chile não é um país pertencente ao continente europeu."
Temos uma sentença declarativa expressa na forma negativa. Contudo, podemos atribuir a ela um valor lógico V, pois se trata de uma sentença verdadeira.

III. Sentença declarativa afirmativa falsa

"6+6 = 10"
Apesar de ser uma sentença declarativa expressa na forma afirmativa, atribuímos a ela um valor lógico F, visto que a sentença é falsa.

IV. Quantificadores

"Y – 3 = 5"
Essa sentença não pode ser considerada uma proposição, pois não sabemos o conteúdo atribuído para a variável Y, ou seja, não podemos atribuir um valor lógico V ou F.

Para transformarmos a sentença em uma proposição, basta usarmos os quantificadores. Vejamos:

Para todo Y, Y pertencente aos Z (números inteiros), Y – 3 = 5.

Isso é uma proposição, pois agora podemos atribuir à sentença um valor lógico. Contudo, sabemos que é falsa, uma vez que apenas o número 8 torna a sentença verdadeira.

Sentenças	Exemplos
Interrogativas	Qual é o seu nome?
Imperativas	Venha logo aqui.
Exclamativas	Ufa!
Abertas	x > 3

Vamos conhecer agora os princípios que regem as proposições e sua representação.

I. Princípio da identidade

Uma proposição tida como **verdadeira** somente será verdadeira. E uma proposição falsa é apenas **falsa**.

II. Princípio do terceiro excluído

Proposição que ou é **verdadeira** ou **falsa**, não existindo uma terceira possibilidade de acontecimentos.

III. Princípio da não contradição

Proposição que não pode ser **verdadeira** nem **falsa** de forma simultânea.

Estruturas

Linguagens de baixo e de alto níveis

É fato que sempre existiu a caracterização das linguagens de programação em dois grandes grupos:

I. Linguagem de baixo nível.
II. Linguagem de alto nível.

I. Linguagens de baixo nível

São linguagens utilizadas para a máquina/computador (parte interna – hardware), isto é, são linguagens escritas usando as instruções do microprocessador do computador...

E conhecida como **linguagem assembly**.

VANTAGENS

- Os programas são executados com uma velocidade diferenciada de processamento (bem mais rápido).
- Os programas irão ocupar menos espaço na memória do computador.

DESVANTAGENS

- Os programas em assembly têm baixa portabilidade, isto é, um programa criado para um tipo de processador não irá servir para outro.
- Os programas em assembly não são estruturados, tornando a programação mais difícil e, com isso, também limitando o número de pessoas que conseguem programar com esta linguagem.

II. Linguagens de alto nível

As linguagens de alto nível foram criadas para que o foco seja o ser humano (criação de sistemas).

Na maior parte, utilizam uma sintaxe muito bem estruturada, permitindo que seu código-fonte seja muito mais coerente. Dessa forma, há necessidade do uso de compiladores e interpretadores para a geração de instruções do microprocessador do computador.

Vamos entender qual é a função de cada um deles?

Os **interpretadores** têm o objetivo de realizar a interpretação de cada instrução escrita no programa fonte, fazendo com que a execução de cada instrução aconteça dentro do ambiente de programação — o programa irá ser executado linha após linha, conforme é interpretado.

Os **compiladores** executam a tradução de todas as instruções desenvolvidas no programa fonte, criando de forma automática um programa executável. Esses programas executáveis são denominados de ".exe" e poderão ser levados e executados fora dos ambientes de programação.

VANTAGENS

- Por serem compiladas ou interpretadas, as linguagens de alto nível terão maior dinâmica, eficiência e portabilidade, podendo ser executadas nas mais diversas plataformas tecnológicas com pouquíssimas modificações.
- A programação será mais fácil por conta do maior ou menor grau de estruturação de suas linguagens e pela questão de se ter mais desenvolvedores ligados a esse cenário de desenvolvimento de programas.

DESVANTAGENS

- De certa forma, as rotinas geradas na linguagem de máquina são mais genéricas e, portanto, mais complexas, por isso tendem a ser mais lentas e irão ocupar mais memória do computador.

Linguagem C

A linguagem de programação C é tida como uma linguagem de alto nível genérica.

Foi desenvolvida no ano de 1972 — por programadores e para programadores — tendo como foco características de flexibilidade e de portabilidade.

A linguagem de programação C nasceu juntamente com o advento da demanda de uma linguagem estruturada e para uso constante no computador pessoal. Assim, rapidamente se tornou uma **linguagem de uso popular** entre os programadores.

A linguagem C foi usada no desenvolvimento do famoso sistema operacional **UNIX**, e hoje está sendo utilizada no desenvolvimento de novas linguagens, como as linguagens **C++** e **Java**.

I. Características da Linguagem C

A Linguagem C apresenta os seguintes destaques:

- Possui a sintaxe muito bem estruturada, segmentada, completa de códigos, funções e comandos flexíveis, permitindo que a programação seja bastante simplória e usual.
- Gera um programa executável com uma extensão ".exe", permitindo que a execução aconteça em qualquer computador.
- Existe a possibilidade tanto de compartilhamento de recursos de alto nível como de baixo nível, pois é permitido que o acesso e a programação sejam, efetivamente, executados diretamente no microprocessador do computador que está sendo utilizado.
- É uma linguagem conhecida, fácil e utilizada mundialmente por ter uma estrutura simplória, flexível e de grande portabilidade para os dias de hoje.

II. Estrutura funcional de um programa em Linguagem C

- Aplicação das diretivas de compilação e a respectiva biblioteca
  - #include <stdio.h>
- Função principal do programa
  - main();
- Declaração de rotinas
- Blocos de instruções
  - Comandos do programa
- Documentação no programa
  - Comentários nas estruturas criadas para que haja entendimento do que foi escrito no código.

III. Conjunto de caracteres

Quando desenvolvemos um programa fonte em **linguagem C**, é possível dizer que:

- É um texto que ainda não foi formatado.
- É um programa escrito em um editor de texto qualquer ou pode ser usado o ambiente IDE do respectivo compilador (Code Block, Dev C++) a ser utilizado e utilizando um o conjunto padrão de caracteres ASCII.

São considerados caracteres válidos em linguagem C:

Letras Maiúsculas	Letras Minúsculas	Números	Operadores aritméticos e símbolos	Caracteres considerados não válidos
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	a b c d e f g h i j k l m n o p q r s t u v w x y z	0 1 2 3 4 5 6 7 8 9	+ - * / = | ! &? % # { } ( ) [ ] " . , < > _ :	@ $ á é õ ç ^ ¨ ~

IV. Comentários

Os comentários na linguagem C podem ser escritos em qualquer lugar do texto para facilitar a interpretação do algoritmo. Para que seja identificado de forma válida, é preciso que o comentário:

- Tenha antes o símbolo /*, determinando o início do comentário em bloco.
- Tenha depois da informação a ser comentada o símbolo */, determinando o fim do bloco.

Saiba mais

A linguagem C permite que uma única linha seja comentada por meio do símbolo //.

V. Declarações

As declarações são partes constituídas de um programa que dão significado para um identificador:

- Permitir que haja alocação de memória.
- Definir um conteúdo inicial para uma variável.
- Definir a criação de funções.

VI. Funções

As funções irão determinar que as ações em um programa sejam executadas quando a mesma for chamada em alguma parte do programa.

A função denominada de mais importante e obrigatória é a função main() — será sempre a primeira função do programa a ser executada, mesma que haja outras funções declaradas no programa.

```
main( )
{
        <comando>;
        <comando>;
}
```

O corpo (conjunto de instruções a ser criado e executado) da função main() é delimitado pela abertura e fechamento das chaves { }.

As instruções criadas na linguagem C que estiverem escritas dentro das chaves serão executadas de forma sequencial quando a função for chamada.

VII. Diretivas de compilação

Existem instruções que são executadas no momento da compilação do programa. Essas instruções são vistas coma as diretivas de compilação.

Essas instruções escritas terão o papel de informar ao compilador:

- Quais serão as constantes simbólicas usadas no programa desenvolvido.
- Quais as bibliotecas que precisam estar declaradas no programa executável.

A diretiva **#include** irá dizer ao compilador que no processo de compilação é preciso que haja a inclusão das bibliotecas no programa.

A diretiva **#define** sinalizará ao compilador quais são as constantes simbólicas utilizadas no programa para que a devida execução aconteça sem que haja problema.

VIII. Biblioteca

Na linguagem C é possível encontrarmos uma diversidade de bibliotecas que podem ser utilizadas em diversos momentos e que, no passado, foram fragmentadas por questão de espaço de memória nos computadores.

IX. Variáveis e constantes

As varáveis e as constantes são elementos:

- Declarados dentro da função “main()”.
- Considerados mais básicos para o programa manipular.

Variáveis

Caracterizadas como informações que serão armazenadas em uma variável e alocadas em um local de memória definido pelo próprio programa.

Constantes

Informações que serão armazenadas em uma variável e alocada em um determinado espaço de memória contudo, a informação NÃO mudará durante a execução do programa.

As variáveis podem ser classificadas como locais ou globais.

Vejamos a definição de cada uma delas:

Variável local	Variável global
• É preciso que sua declaração esteja dentro da função main(). • O seu conteúdo não é visto por outra função que não seja a main().	• É preciso que sua declaração esteja entre a diretiva de compilação e a função main(). • O seu conteúdo será visto por outras funções além da main().

Importante

A linguagem C é case sensitive, ou seja, maiúsculas e minúsculas fazem toda a diferença na identificação. Se declararmos uma variável com o nome "soma" ela será diferente de "Soma", "SOMA", "SoMa" ou "sOmA".

O mesmo também se aplica para comandos e funções.

X. Tipos de dados

char	Armazena na memória valores do tipo **caractere**. Exemplo: 'M'
int	Armazena na memória **valores inteiros**. Exemplo: 28 Atenção: não armazena casa decimal.
float	Irá armazenar na memória **valores numéricos com casas decimais**. Exemplo: 10,00
double	Armazena na memória v**alores numéricos com casas decimais**, porém, comporta um tamanho maior de informações do que o float. Exemplo: 1.000.000,00
void	Armazena na memória um resultado para um tipo não definido. **Aceita qualquer valor**.

Entrada e saída de dados

Na linguagem C existem várias maneiras de fazer a leitura e a escrita de informações.

Essas operações, no processo de programação, são chamadas de operações de entrada e de saída.

- **Função de entrada de dados**

Função scanf()
Permite que o usuário realize a entrada de dados via teclado. Utiliza o "&" para fazer o endereçamento de memória. O símbolo & é quem permite que um dado seja armazenado em um determinado local de memória.

Sintaxe:
scanf ("impressão de tipo de dado", &variavel);

Exemplo:
int val;
scanf ("%d", &val);
%d → Indicativo do tipo, neste caso do tipo inteiro.
& → Operador utilizado para obter o endereço de memória da variável.
Val → Variável que receberá a informação via teclado.

- **Função de saída de dados**

Função printf()
A partir da função predefinida printf(), cujo protótipo está contido também no arquivo stdio.h, poderemos imprimir na tela informações.

Sintaxe:
printf ("Expressão");
ou
printf ("Expressão", lista de argumentos);
Exemplo:
printf ("Volta ao Mundo!");

Expressão → Mensagens que serão exibidas.

Lista de Argumentos → Pode conter identificadores de variáveis, expressões aritméticas ou lógicas e valores constantes.

XI. Estrutura de decisão

O objetivo da estrutura de decisão é realizar o direcionamento do fluxo lógico do programa para dois blocos distintos de instruções (verdadeiro ou falso), conforme uma das condições de controle for caracterizada.

Sempre haverá uma condição verdadeira ou falsa.

Pseudolinguagem	Linguagem C
se condição	if(condição){
então bloco 1	bloco 1;
senão bloco 2	}else{
fim se	bloco 2;
};	

Estrutura de repetição

A estrutura de repetição permite que seja executado, repetidamente, um bloco de instruções até que uma determinada condição de controle seja satisfeita.

Pseudolinguagem	Linguagem C
faça	do{
bloco	bloco;
até condição	}while(condição);

Representação de solução em algoritmos e os respectivos testes

Descrição narrativa

Tipo de representação que permite a utilização, na íntegra, de uma língua nativa para a realização e para a descrição das etapas visando à resolução de um problema específico.

VANTAGENS

Qualquer indivíduo pode fazer a descrição narrativa sem que, para isso, tenha conhecimentos elevados de programação.

DESVANTAGENS

Não há como mensurar ou determinar um padrão, uma vez que cada programador pode escrever seu código-fonte da forma como bem entender, gerando imprecisão a descrição pode não ficar clara e, assim, propiciar interpretações diferentes.

Exemplo

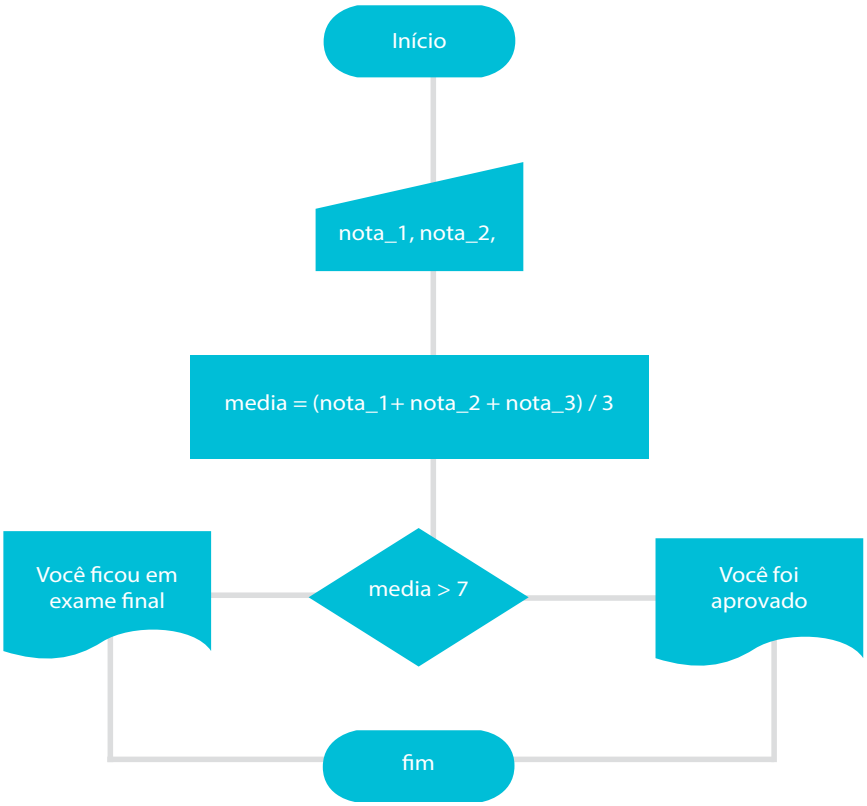
Algoritmo com uma descrição narrativa:

1. Início
2. Etapa 1: Obter os valores de nota_1, nota_2, nota_3;
3. Etapa 2: Somar os valores da etapa 1;
4. Etapa 3: Dividir o resultado obtido na etapa 2 por 3;
5. Etapa 4: Se o resultado da etapa 3 for maior ou igual a 7 então escreva
6. "Você foi aprovado", senão, escreva "Você ficou em exame final"
7. e vá para o fim do programa;
8. Fim

Fluxograma

Criado com o objetivo de eliminar as ambiguidades dos algoritmos, um fluxograma é constituído de símbolos gráficos, onde cada símbolo é representado por uma forma geométrica que implica uma ação específica, instrução ou comando a ser realizado.

Podemos verificar no diagrama a seguir que há uma denotação de um cenário intermediário à descrição narrativa e ao pseudocódigo (Portugol).



Importante

O fluxograma não traz a preocupação com detalhes de implementação do programa.

Linguagem algorítmica (pseudocódigo ou portugol)

A linguagem algorítmica foi criada com o intuito de sanar as deficiências das outras representações.

É caracterizada como uma "pseudolinguagem de programação", cujos comandos são escritos em português, mas já estão denotados em uma estrutura de uma linguagem de programação bem estruturada.

A pseudolinguagem se parece com um programa, ou seja, é escrita em uma determinada linguagem.

```
1.  algoritmo "Calcmedia"
2.  var
3.      nota_1, nota_2, nota_3,media :real;
4.  inicio
5.      leia (nota_1, nota_2, nota_3);
6.          media ← (nota_1 + nota_2 + nota_3) / 3;
7.      se  media >= 7 então
8.          escreva("Você foi aprovado");
9.      senão
10.         escreva("Você ficou de exame final");
11.     fimse
12. fimalgoritmo
```

Algoritmos de ordenação

Algoritmo criado para proporcionar a colocação dos elementos que estão em uma determinada sequência e assim respeitar uma certa ordem. O algoritmo irá realizar a ordenação das informações de forma completa ou parcial.

O objetivo maior da ordenação é facilitar a recuperação dos dados de uma determinada lista.

Alguns tipos de algoritmos para ordenação:

I. Bubble sort.
II. Selection sort.
III. Quick sort.
IV. Insertion sort.

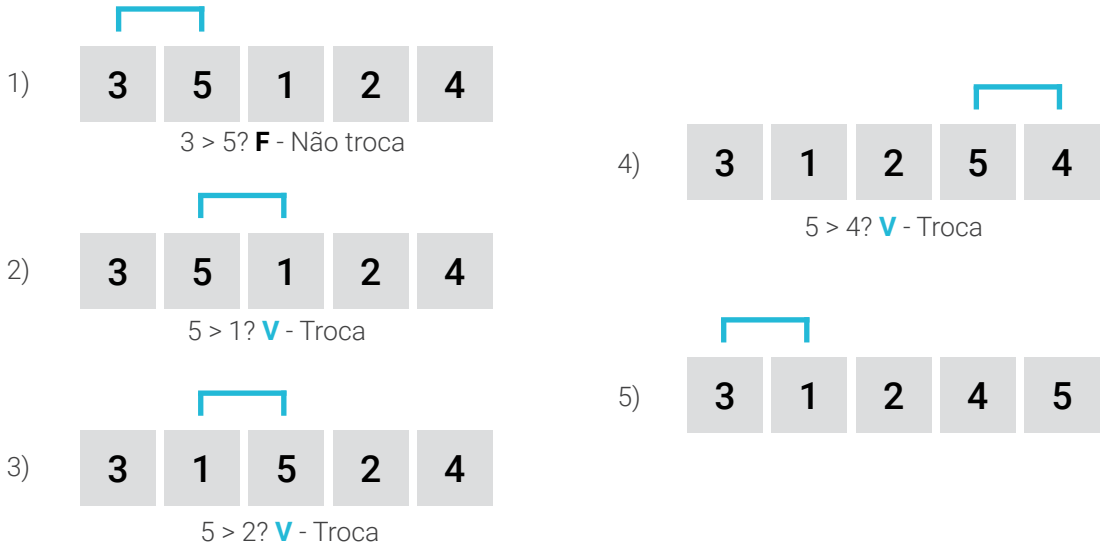
I. Bubble sort

Algoritmo mais simples e bem menos eficiente que os demais.

Neste algoritmo cada elemento existente na posição i será comparado com o elemento existente da posição i + 1. Assim, o elemento da segunda posição será comparado com o elemento da terceira posição. Se o elemento encontrado na segunda posição for maior que o elemento da terceira posição, automaticamente trocam de lugar e assim será realizado sucessivamente.

Da forma como a execução se deu, o vetor terá que ser percorrido quantas vezes for necessário, logo o algoritmo será bem ineficiente para as listas que sejam mais dimensionadas.

1ª Passagem do Bubble Sort



É verificado:

- Se o elemento 3 é maior que o elemento 5, por essa condição ser falsa, não há troca.
- Se o elemento 5 é maior que o elemento 1, por essa condição ser verdadeira, há uma troca.
- Se o elemento 5 é maior que o elemento 2, por essa condição ser verdadeira, há uma troca.
- Se o elemento 5 é maior que o elemento 4, por essa condição ser verdadeira, há uma troca.

Observação

O método retornará ao início do vetor, realizando os mesmos processos de comparações. Isso é feito até que o vetor esteja todo ordenado.

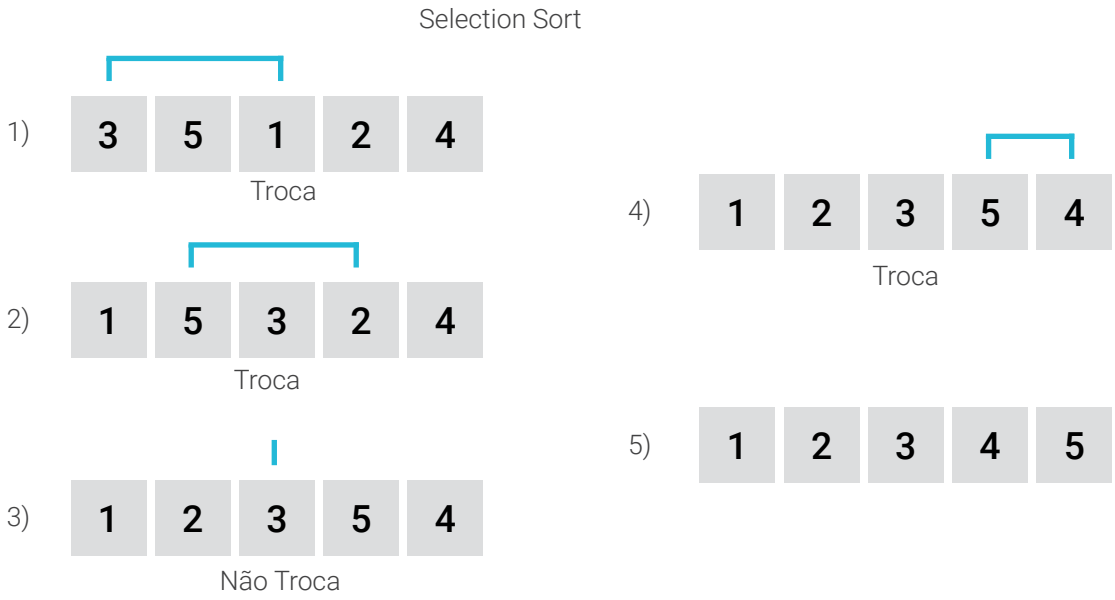
II. Selection sort

Tem como objetivo, sempre:

3. A colocação do menor elemento do vetor na primeira posição.
4. Na sequência, o segundo menor elemento na segunda posição.
5. E assim se repete, até executar os últimos dois elementos.

O algoritmo se baseia na escolha de um elemento a partir do primeiro elemento existente no vetor. Este elemento será comparado com os demais da direita para a esquerda. Quando for identificado um elemento menor, o elemento escolhido anteriormente ocupará a posição do menor. Este elemento estará próximo ao selecionado, caso não seja identificado nenhum elemento menor. Ele será colocado na posição do primeiro elemento identificado, e o próximo à sua direita vai ser o escolhido para fazer as respectivas comparações.

Este processo será executado repetidas vezes até que a lista esteja completamente ordenada.



- Neste passo, o primeiro elemento determinado foi o 3, ele foi comparado com todos os números à sua direita e o menor número encontrado foi o elemento 1, então os dois trocam de lugar.
- O mesmo processo da etapa 1 acontece, o elemento escolhido foi o 5 e o menor elemento encontrado foi o 2.
- Não foi encontrado nenhum elemento menor que 3, então ele fica na mesma posição.
- O elemento 5 foi determinado novamente e o único elemento menor que ele à sua direita é o 4, então eles trocam.
- Vetor já está ordenado.

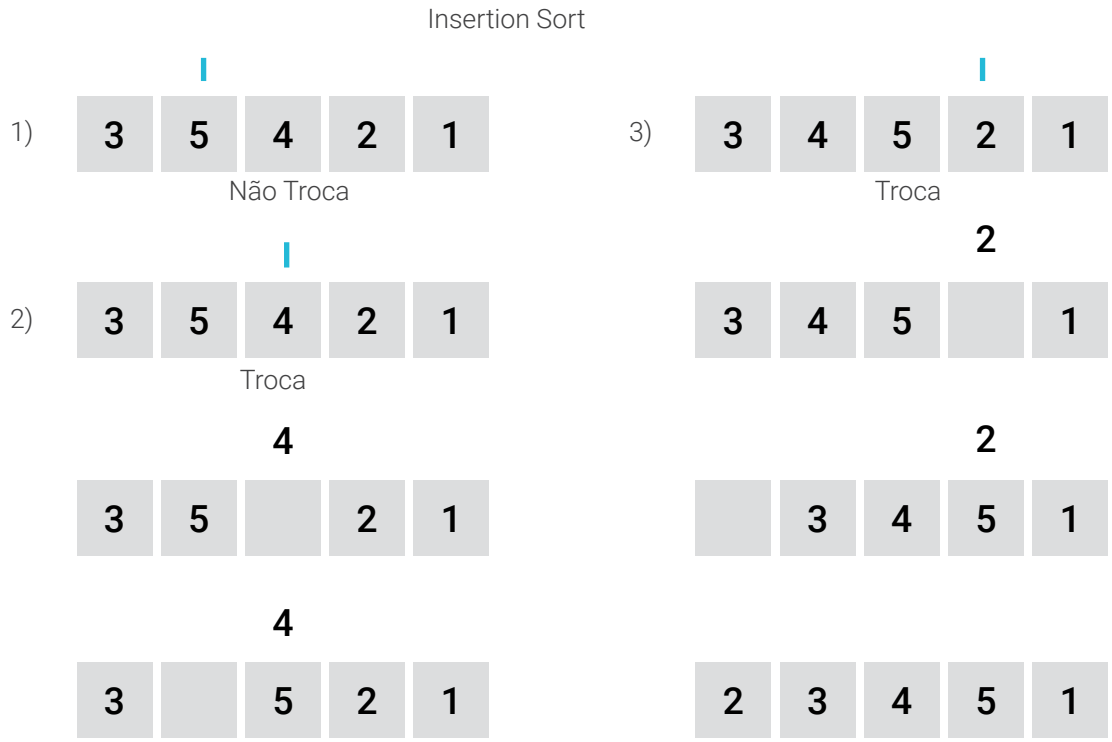
III. Insertion sort

Muito simples, eficiente e eficaz quando aplicado em pequenas listas.

A lista é varrida da esquerda para a direita e, conforme avança, vai deixando os elementos mais à esquerda ordenados.

Dica

Analogamente, o insert sort funciona como a ordenação de cartas em um jogo de baralho.



É verificado:

- Se o elemento 5 é menor que o elemento 3 — como essa condição é falsa, então não há troca dos elementos.
- Se o elemento 4 é menor que o elemento 5 e o elemento 3 — como ele só é menor que o elemento 5, então apenas os dois trocam de posição.
- Se o elemento 2 é menor que o elemento 5, que o elemento 4 e o elemento 3 — como ele é menor que o elemento 3, então o elemento 5 passa a ocupar a posição do elemento 2, o elemento 4 passa a ocupar a posição do elemento 5 e o elemento 3 passa a ocupar a posição do elemento 4, assim a posição do elemento 3 fica vazia e o elemento 2 passa para essa posição.

O mesmo processo de comparação acontece com o elemento 1, após esse processo o vetor passa a ficar ordenado.

IV. Quick sort

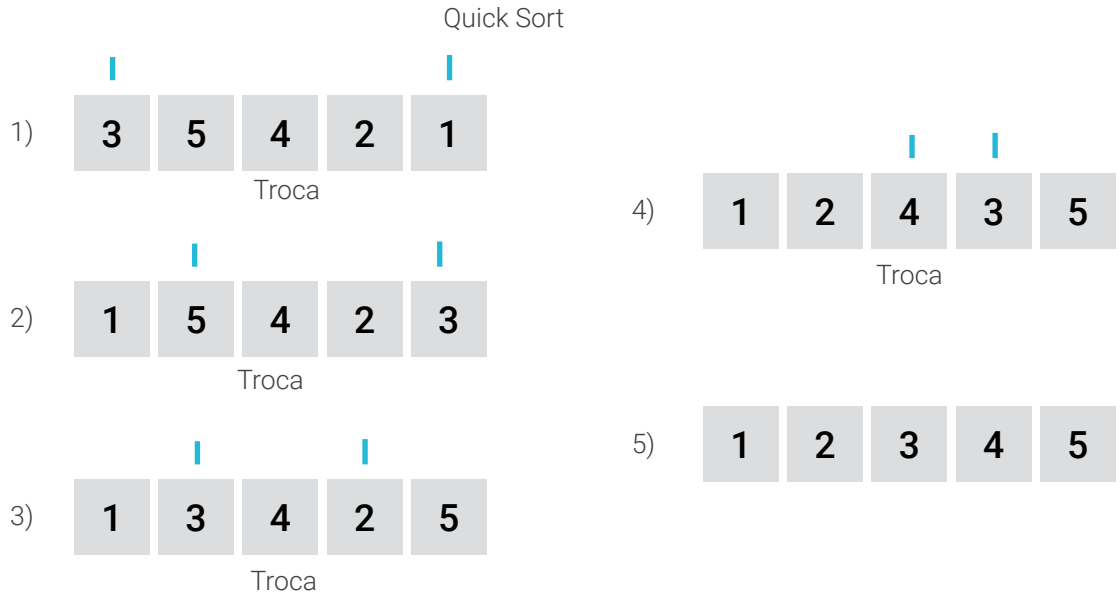
É considerado o mais eficiente para atuar com relação às ordenações por comparação.

Um determinado elemento será escolhido e chamado de pivô.

A partir de então é organizada uma lista para que:

- Todos os elementos anteriores a este pivô sejam menores que ele.
- Todos os elementos posteriores a este pivô sejam maiores que ele.
- No final de tudo, o elemento pivô já estará ocupando a sua posição final.

Os dois grupos que foram criados e foram mantidos de forma desordenada irão sofrer o mesmo processo até que a respectiva lista esteja totalmente ordenada.



- O elemento 3 foi escolhido como pivô. Nesse passo, é procurado à sua direita um elemento menor que ele para ser passado para a sua esquerda. O primeiro elemento menor encontrado foi o 1, então eles trocam de lugar.
- Agora é procurado um elemento a sua esquerda que seja maior que ele. O primeiro elemento maior encontrado foi o elemento 5, portanto eles trocam de lugar.
- O mesmo processo da etapa 1 acontece, o elemento 2 foi o menor elemento encontrado, eles trocam de lugar.
- O mesmo processo da etapa 2 acontece, o elemento 4 é o maior elemento encontrado, eles trocam de lugar.
- O vetor acima é tido como um vetor pequeno, portanto ele já foi ordenado, mas se fosse um vetor grande, ele seria dividido e recursivamente aconteceria o mesmo processo de escolha de um pivô e comparações.

MIDIATECA

Veja, na midiateca da Unidade 1, o conteúdo complementar selecionado pelo **professor sobre processo de trabalho com a linguagem C**.

NA PRÁTICA

Vamos analisar um primeiro problema e, a partir dele, permitir que as ideias fiquem mais claras.

Etapas para chegar em determinado lugar

Como um algoritmo surge? Ele surge diante da existência de um problema.

Problema: Realizar a divisão de um número por outro número. Se o resultado for satisfatório, imprimir o número encontrado pela divisão, caso seja insatisfatório, imprimir zero. Caso o divisor seja zero, imprimir o valor -1.

Início do pensamento sobre este problema:

- Possível primeira etapa: Pegar os dois números.
- Possível segunda etapa: Verificar o valor do divisor, que no nosso caso é o B.
- Possível terceira etapa: Se o divisor for zero, imprimimos -1, caso contrário continuamos.
- Possível quarta etapa: Verificar se o número encontrado após a divisão é positivo ou negativo.
- Possível quinta etapa: Imprimir o número encontrado ou imprimir o valor zero.

Resumo da Unidade 1

Nesta unidade você estudou toda a importância de se trabalhar com atenção para que não haja problemas referentes ao processo de criação de um programa. Para isso, foi fundamental a introdução sobre a linguagem de programação em C. Da mesma forma, como as dá a sua estrutura e aplicabilidade dentro dos processos de programação. Outro ponto importante foi poder contemplar os cinco tipos de algoritmos existentes que ajudam na descoberta de problemas e ordenação de valores dentro um vetor.

CONCEITO

Ao longo do material foi possível verificar a importância do uso da linguagem de programação; a importância da estrutura da linguagem de programação em C e a sua devida aplicabilidade; e a existência de vários algoritmos para testes de informações e ordenação de valores.

Referências

GUEDES, S. **Lógica de Programação Algorítmica.** São Paulo: Pearson Education do Brasil, 2015. Biblioteca Virtual.

MENDES, F. V. **Programação Avançada em C++**. São Paulo: Pearson Education do Brasil, 2006. Biblioteca Virtual.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos.** 3. ed. Rio de Janeiro: Grupo GEN - LTC, 2013. Minha Biblioteca.

UNIDADE 2

Transformando a solução algorítmica em um programa

INTRODUÇÃO

Nesta unidade vamos conhecer os fundamentos do ambiente de programação em Linguagem C e entender como se dá a criação de determinada variável e seus tipos. Desta forma, seremos capazes de verificar os tipos de dados utilizados para determinar qual informação a variável armazenará e, da mesma forma, verificar a correlação na utilização das variáveis para a realização das operações aritméticas.

OBJETIVO

Nesta unidade você será capaz de:

• Compreender e aplicar os conceitos básicos da programação estruturada com a linguagem C.

Introdução à linguagem de programação em C

A **Linguagem de Programação em C** é uma linguagem ampla, direcionada à programação estruturada e muito utilizada para análise léxica, desenvolvimento de compiladores, editores de texto, bancos de dados etc.

Curiosidade

Década de 1970

A Linguagem de Programação em C foi desenvolvida em um centro de pesquisas da empresa estadunidense **Bell Laboratories**, pelo professor Dennis Ritchie, no ano de1972.

A sua primeira utilização de grande importância foi reescrita ao Sistema Operacional UNIX, que, até então, tinha o seu código-fonte escrito em **assembly**.

Assim, o Sistema Operacional UNIX deixou de atuar apenas no laboratório e passou a ser utilizado nas universidades.

Década de 1980

O sucesso da linguagem C atinge grandes proporções, passando a existir várias versões de compiladores para a linguagem de programação em C que tiveram suas versões criadas por várias empresas, não sendo mais utilizadas apenas para o Sistema Operacional UNIX.

Contudo, essas versões eram compatíveis com vários outros sistemas operacionais existentes.

Vamos conhecer os principais objetivos da Linguagem C.

Oferecer aos programadores maior controle, eficácia e eficiência.	Facilitar a criação de programas com grandes números de linhas de código e com menos erros	Retirar a sobrecarga do compilador - cujo trabalho se complica ao ter de realizar as características complexas da linguagem

Vejamos, agora, as características da Linguagem de Programação C:

- Portabilidade.
- Modularidade.
- Compilação em separado.
- Recursos para baixo nível.
- Regularidade.
- Geração de código eficiente.
- Facilidade de uso.
- Confiabilidade.

A visão geral de um programa desenvolvido em linguagem de programação C

A forma como visualizamos o código-fonte de um programa é por meiode um ambiente **IDE**. No caso da linguagem C isto é tratado como o compilador da linguagem, como o **DEV C++** ou o **Code Block**.

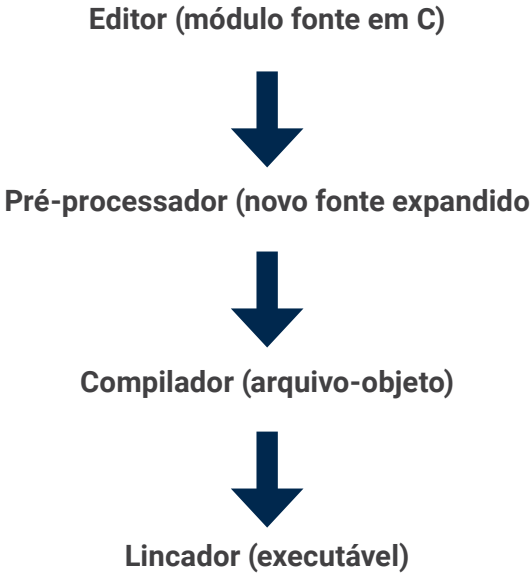
Saiba mais

O IDE é uma ferramenta de visualização dos códigos.

Dentro do ambiente de programação, este conjunto de códigos escritos é chamado de **programa-fonte** que, após o processo de compilação bem-sucedido, tornar-se-á um programa executável “.**exe**”.

Depois que o módulo-fonte em C for desenvolvido em um editor específico de textos, o programador acionará a opção do compilador – que no Sistema Operacional UNIX é chamado pelo comando "**cc**".

Essa ação desencadeará uma sequência de etapas estruturadas, realizando uma tradução e uma codificação do usuário para um formato de linguagem de nível inferior, culminado com a criação do **executável** pelo Lincador.



Compiladores e interpretadores

Os termos **compiladores** e **interpretadores** se referem à forma como um programa é executado.

Teoricamente, qualquer linguagem de programação existente poderá ser compilada e interpretada.

É fato que os respectivos interpretadores e os compiladores são tidos como programas mais robustos e dinâmicos pelo fato de poderem atuar na interpretação e compilação do código-fonte do programa.

Vamos entender a função de cada um deles.

INTERPRETADOR

Lê o código-fonte do programa, uma linha de cada vez, executando a referida instrução especificada naquela linha.

COMPILADOR

Lê o programa inteiro e faz a conversão daquela leitura em um código-objeto - que nada mais é do que uma tradução do código-fonte do programa em uma forma que o computador poderá executar as definições programadas.

O código-fonte do programa deverá estar presente toda vez que o interpretador for executado. A estrutura processual para isso é bem lenta, pois para cada execução o programa fonte será lido.

Quando mensuramos um compilador, podemos verificar que este já se comporta de forma contrária, convertendo seu programa em um código-objeto que poderá ser executado pelo seu computador.

O compilador realizará uma tradução do seu programa de uma só vez, executando seu programa pelo arquivo ".**exe**", não necessitando do programa-fonte toda vez que quiser executá-lo.

Sintaxe

As sintaxes existentes na linguagem C são constituídas de regras muito bem detalhadas para que o processo de compilação seja exitoso e tudo aquilo que foi escrito possa ser plenamente identificado e redunde em um programa executável que atenda ao objetivo proposto.

Estas regras estarão muito bem relacionadas com os seguintes propósitos:

- Tipos.
- Declarações.
- Funções.
- Expressões.

A estrutura de um programa em Linguagem C é composta por:

- Diretiva de Compilação e Biblioteca.
- Definição de Tipos de Dados.
- Protótipos de Funções.
- Funções.
- Comentários.

a. Diretiva de compilação

A diretiva de compilação deve ser iniciada com a palavra #include, sendo seguida de uma determinada biblioteca.

Exemplo

#include <stdio.h>
Diretiva de compilação utilizada para que haja um entendimento do que foi escrito no programa. Caso contrário, o desenvolvedor poderia escrever qualquer palavra para ser executada.

Tipos e declarações

Os tipos definem as respectivas propriedades dos dados manipulados em um programa.

As **declarações** expressam as referidas partes do programa:

- Dando significado a um identificador.
- Permitindo a alocação de memória.
- Definindo um conteúdo inicial.
- Definindo as funções.

b. Funções e expressões

As funções especificam as ações que determinado programa irá executar.

As determinações, as alterações de valores e as chamadas funções de I/O (input / output) serão definidas nas expressões.

As funções são entidades operacionais básicas que possuem um determinado propósito que, por sua vez, são a integração de uma ou mais funções em que cada qual irá executar a sua atividade.

Existem funções básicas que estão definidas na biblioteca stdio.h da linguagem de programação C, a saber:

- Função printf() – exibe informações na tela para o usuário.
- Função scanf() – o usuário poderá fazer entrada de dados via teclado.

O programador também poderá definir algumas novas funções em seus programas, como:

- Rotinas para cálculos.
- Leitura de arquivos textos.
- Impressão.

O programa desenvolvido na linguagem C, obrigatoriamente, terá o seu início com a execução da chamada função main() – função principal.

Observação

É obrigatória a existência da função main() no programa principal.

c. Comentários

Os **comentários** que o desenvolvedor precisar criar no programa para explicar ou definir algo que julgue importante serão colocados entre "/* e */", não sendo considerados no processo de compilação.

Cada instrução do programa em linguagem C deverá ser encerrado com ";" (ponto e vírgula) – que faz parte obrigatória da maior parte dos comandos. O ";" determina que a instrução terminará neste momento.

Exemplo

```
main() /* função obrigatória */
{
printf("Bom dia!"); /* função para exibir informação na tela */
printf("Seja bem-vindo!");
}
```

Identificadores

São nomes utilizados para se fazer uma referência a variáveis, funções, rótulos, labels e vários outros objetos que podem ser definidos pelo programador.

O primeiro caractere a ser definido é preciso que seja uma letra ou um sublinhado.

Observação

Os 32 primeiros caracteres de um identificador são de grande relevância.

A linguagem C trabalha como formato de “case sensitive” – as letras minúsculas diferem das letras maiúsculas e isso pode causar problemas no processo de complicação do programa.

Exemplo

int x; /* é diferente de **int X;*/**
Ao realizarmos a declaração de uma variável denominada “valor” ela será diferente de “Valor”, “VALOR”ou “VaLor”.

Também se aplica para comandos e funções.

Acentuação

Uma vez que trabalha com o padrão de linguagem estadunidense, o IDE do DEV C++ não enxerga nosso padrão de acentuação. Para isso, é importante aplicarmos os códigos para a transformação do entendimento para o idioma “Português”.

Exemplo

```
#include <stdio.h>
#include <locale.h>
main()
{
        setlocale (LC_ALL, "Portuguese");
        printf("Alô mundo!");
        printf ("E aí, você está bem? ");
        printf ("Boa Apresentação!");
}
```

Compreender o que significa uma variável e seus respectivos tipos

A variável sempre possuirá determinado local na memória do computador para que determinado dado seja armazenado.

Funciona como se fosse uma caixa guardando determinado dado que:

- Pode ser acessado em um momento do programa.
- Pode ser utilizado para a realização de um cálculo.

Toda variável precisa ter um identificador, ou seja, um nome que a identifique dentro do programa, pois por meio deste nome é que o programador poderá fazer uso dela nas suas instruções.

Além da necessidade de possuir um nome, a variável também precisará da identificação de um **tipo de dado**. O tipo de dado atribuído a uma variável determina o que ela é capaz de armazenar em termos de dados.

Variáveis e constantes

Variáveis e constantes são os elementos básicos que um programa pode manipular em tempo de execução.

variáveis	constantes
Pequenos espaços criados para alocar dados na memória de um computador - esses dados podem ser modificados durante a execução de um programa.	Pequenos espaços alocados na memória do computador para armazenar informações que **NÃO** serão modificadas no decorrer da execução do programa.

Nomes das variáveis

Existem algumas regras básicas para regular a nomenclatura das variáveis.

- O nome só poderá ter letras e dígitos.
- O caractere "_" pode ser usado, pois é tido como uma letra.
- O primeiro caractere precisa, necessariamente, ser uma letra.
- As letras minúsculas e maiúsculas são caracteres diferentes.

Importante

Palavras reservadas oriundas da linguagem C não podem, em hipótese alguma, ser usadas no nome das variáveis a serem criadas.

São palavras reservadas da linguagem C:
auto / break / case / char / const / continue / default / do / double / else / extern / for / float / short / struct / union / unsigned / void / while.

É fundamental que a escolha dos nomes das variáveis tenha um significado e indique a sua finalidade – como valor, soma, total, nome ou raio.

Variáveis locais e globais

A variável local é declarada dentro da estrutura de uma função específica, podendo ser a função main() ou outra função criada.

Observação

Somente pode ser utilizada em tempo de execução por aquela função que recebeu a declaração e mais por nenhuma outra.

Exemplo

Variável local

```
#include <stdio.h>
main ()
{
        int a; // variável declarada como LOCAL (está após a função main())
        a = 3;
}
```

A variável global é declarada após a diretiva de compilação **"#include <stdio.h"** e será usada em todas as funções que fazem parte do programa.

Exemplo

Variável global

```
#include <stdio.h>
        int a; // variável declarada como GLOBAL (está antes da função main())
main ()
{
        a = 3;
}
```

Vejamos também um exemplo de **declaração de variáveis constantes**

Exemplo

```
#include <stdio.h>
#define icms 0.18; //declaração da constante

main()
{
 float preco_produto, valor_icms; // declaração variável local
        preco_produto = 50;
        valor_icms = preco_produto * icms;
 printf("Valor de imposto a ser pago: R$ %f", valor_icms);
}
```

Tipos de dados

Os dados poderão assumir cinco tipos básicos na linguagem de programação C, a saber:

- char.
- int.
- float.
- double.
- void.

Conheça a definição de cada um deles.

char (caractere)	O valor armazenado é um caractere. Cabe ressaltar que um valor numérico, quando armazenado em um caractere, perde o seu poder de realização de cálculo. A placa de um veículo é um bom exemplo: "TGL8F92".
int (número inteiro)	Tipo de dado utilizado para a representação de informações numéricas dentro do programa.
float	Número em ponto flutuante de precisão simples caracterizado, normalmente, como números reais. São utilizados para trabalhar com valores em moedas (R$567,89), notas de uma prova (9,6) etc.
double	Número em ponto flutuante de precisão dupla utilizado para trabalhar com valores decimais que demandam um número maior de armazenamento de informações. Exemplo: R$1.000.000.000,00.
Void	Tipo de dado utilizado para denotar que um resultado quando não possuir um tipo de dado definido terá seu valor ligado ao void. Aplicações desse propósito criam um tipo vazio que poderá, posteriormente, ser modificado ao longo da execução do programa para um dos outros tipos (char, int, float ou double).

Operadores

Vejamos, agora, alguns tipos de operadores.

1. Operadores relacionais

Operadores relacionais	
=	atribuição
!=	Diferente
>	maior
<	menor
<=	menor igual
>=	maior igual
==	igual
%	resto da divisão entre números inteiros

2. Operadores lógicos

Operadores lógicos	
&&	and (e)
||	or (ou)
!	not (não)

3. Operadores aritméticos

Aritmético	Operação	Prioridade
+	Adição	5
-	Subtração	5
%	Resto da divisão	4
*	Multiplicação	3
/	Divisão	3
++	Incremento	2
--	Decremento	2
+	Manutenção do sinal	1
-	Inversão do sinal	1

4. Operadores e expressões

As expressões combinam operandos e operadores para produzir um único resultado.

Exemplo

```
a + b * c – d
```

5. Operadores de Atribuição (=)

O valor da expressão da direita é atribuído à variável da esquerda.

Exemplo

```
soma = a + b;
a = b = c = d = 10;
```

6. Operadores Aritméticos

(* , / , % , + , -)

Exemplo

```
a = 3 + 2 * (b = 7/2);  → b = 7 / 2 e  a = 3 + 2 * b
printf (“%d”, -3 + 4 * 5 – 6) → 11
```

7. Operadores aritméticos de atribuição

(+= , -= , *= , /=)

Exemplo

```
x = x + 5; → x += 5;
x = x – 5;  → x -= 5;
x = x * 5;  → x *= 5;
```

Atribuição e operações algébricas (entrada/saída)

Declarando variáveis

Vejamos como uma variável deve ser declarada em um programa de linguagem C.

É preciso definir:

- O tipo de dado.
- Atrelado ao tipo de dado, o nome da variável a ser utilizado.

É possível que na mesma linha de instrução seja colocada mais de uma variável – desde que elas sejam do mesmo tipo.

```
#include <stdio.h>
main()
{
        int val;
        double salario;
        int num, soma, total;
}
```

Atribuição de valores às variáveis

Agora, vamos entender como uma variável recebe um valor fixo dentro da função principal.

A variável **dias**, que possui o tipo de dado int, recebe em tempo de execução o valor 30.

```
#include <stdio.h>
main()
{
int dias;
dias = 30;
}
```

Porém, como uma variável recebe um valor fixo dentro da função principal?

A variável **num** – que possui o tipo de dado **int** – recebe em tempo de execução o valor 28 no mesmo momento em que é declarada.

Já a variável cep recebe o valor 22786021 após ter sido declarada.

```
#include <stdio.h>
main()
{
int num = 28, cep;
cep = 22786021;
}
```

Operações algébricas

As operações algébricas são expressões que apresentam uma combinação de letras e números, podendo também ser chamadas de **expressões literais**.

Exemplo

A = 5a+ 7b
B= (2C+6) - 1
C= 30 d + 5

Nas expressões podemos conceber que as letras são chamadas de variáveis e que cada letra poderá ser determinada por um valor numérico.

Observações prioritárias

Quando há uma operação aritmética e a existência da aplicação de **parênteses**, **colchetes** ou **chaves** (ordem de prioridade), estes serão respeitados para a realização do cálculo.

Importante

É preciso utilizar os parênteses quando formos substituir as variáveis por valores negativos.

Exemplo

1) Consideremos **B = 2C + 10**, sendo o valor de **C = 3**.
B = 2.3 + 10
B = 6 +10
B = 16

2) C é a variável da expressão; 3 é a representatividade numérica da variável; e 16 será a representatividade numérica da expressão, que é indicada pela variável B.

Na modificação do valor de C para 10, teremos:
B = 2.10 + 20
B = 20 + 10
B = 30

3) Se **C = 11**, o valor numérico da expressão de **B = 2C + 8** é igual a **30**.

4) Seja **X= 2A + 4 + B-5 e A = 4** e **B = 6**, assim:

2A+4+B
2.4+4+6-5
8+4+1
13

5) Seja Y=12- C +10 + D +3C, em que C = -3 e D = 1.
Y= 12 - (-3) + 10 + 1 +3(-3)
Y= 12 + 3 + 11 - 9
Y= 27 - 9
Y= 18

Funções matemáticas

Necessário o uso da biblioteca **math.h**

A biblioteca **math.h** oferece um vasto conjunto de funções para operações matemáticas, bem como funções logarítmicas, trigonométricas, hiperbólicas, arredondamentos e potência.

Todas as funções da biblioteca **math.h** retornam um valor do tipo **double**.

a. Potência

pow(): retorna para o usuário o valor da base elevada ao expoente.

Recebe dois argumentos do tipo **double** (pelo fato de poder retornar um grande número):

O primeiro sempre será a base.
O segundo sempre será o expoente.

Exemplo

Se quiséssemos saber o resultado da operação 410, faríamos pow (4, 10).

b. Raiz quadrada

sqrt(): retorna para o **usuário** o valor da raiz quadrada.

Recebe como argumento um tipo **double** (pelo fato de poder retornar um grande número) do qual ele deve extrair a raiz.

c. Funções trigonométricas

Função tan(), cos() e sin(): retorna os valores de um dos tipos mencionados.

Recebe como retorno, em **radianos**, o valor do tipo **float**.

Observação

1 grau = 0,017453 radianos
num é a variável que conterá a informação a ser convertida.

Sintaxes:

- tan(num).
- cos(num).
- sin(num).

d. Funções logarítmicas

Função log() e log10(): retorna o valor do logaritmo na base 2, utilizando uma variável do tipo **float**.

Observação

A função **log10()** é idêntica à função **log()**, porém retorna o valor do logaritmo na base 10.
num é a variável que conterá a informação que será convertida.

Sintaxes:

- log10(num).
- log(num).

Exemplo

Código fonte

```
#include  <stdio.h>    //biblioteca padrão
#include  <math.h>   //biblioteca para as funções matemáticas
int main()
{
        float var;
        printf("Digite um numero: ");
        scanf("%f", &var);
        printf("O Seno de %f eh %f\n", var, sin(var));
        printf("O Cosseno de %f eh %f\n", var, cos(var));
        printf("A Tangente de %f eh %f\n", var, tan(var));
        printf("O Log na base 2 de %f eh %f\n", var, log(var));
        printf("O Log na base 10 de %f eh %f\n", var, log10(var));
}
```

MIDIATECA

Para ampliar o seu conhecimento, acesse o material complementar da Unidade 2 disponível na midiateca.

NA PRÁTICA

Uma empresa possui a demanda de construção de um simples algoritmo para o cálculo exponencial. É preciso que o usuário tenha flexibilidade para informar qualquer valor para o cálculo.

```
#include <stdio.h>
#include <math.h>
Int main()
{
        double base = 0, exp = 0;
        printf ("Informe a base: ");
        scanf ("%lf", &base);
        printf ("Informe o Expoente: ");
        scanf ("%lf", &exp);
        printf (“O resultado e: %lf“, pow (base, exp));
}
```

Resumo da Unidade 2

Nesta unidade você conheceu, na íntegra, a linguagem de programação C, visando utilizá-la por meio do viés de entendimento do que é uma variável, bem como suas aplicações. Dessa forma, também pode apropriar-se de conhecimento dos tipos de dados que podem ser trabalhados ao declarar as variáveis e como utilizar estas estruturas para as devidas operações aritméticas necessárias.

CONCEITO

Importância da estrutura da linguagem de programação em C e a sua devida aplicabilidade.

Referências

GUEDES, S. **Lógica de Programação Algorítmica.** São Paulo: Pearson Education do Brasil, 2015. Biblioteca Virtual.

MENDES, F. V. **Programação Avançada em C++.** São Paulo: Pearson Education do Brasil, 2006. Biblioteca Virtual.

SZWARCFITER, J. L.; MARKEZON, L. **Estruturas de dados e seus algoritmos.** 3. ed. Rio de Janeiro: Gen - LTC, 2013. Minha Biblioteca.

UNIDADE 3

Estrutura de decisão

INTRODUÇÃO

Nesta unidade conheceremos a importância da estrutura de decisão dentro do processo de desenvolvimento de software, bem como se dá sua aplicação tendo em vista as variações que podem ser trabalhadas e a relação direta com os operadores relacionais e lógicos. Também teremos um entendimento claro da diferença de se trabalhar com o uso do **if** e do **else**, pois trata-se de performance no processo de execução da aplicação.

OBJETIVO

Nesta unidade você será capaz de:

- Compreender e aplicar os conceitos da estrutura de decisão para solucionar medidas comparativas nos programas.

O que é uma estrutura de decisão

Introdução

A **condição** (também conhecida como estrutura de decisão) na linguagem C é definida como uma expressão que pode ser **verdadeira** ou **falsa**, ou seja, só há como retornar um valor verdadeiro ou falso.

Para esse tipo de expressão também podemos utilizar a nomenclatura de **expressão lógica**.

Exemplo

(5 > 3) – expressão lógica que possui valor **verdadeiro**.
(7 < 5) – expressão lógica que possui valor **falso**.

Podemos entender que a aplicação da estrutura de decisão no desenvolvimento de uma aplicação se dá quando temos a real necessidade de fazer uma verificação. Esta verificação será validada de acordo com:

- Os tipos de dados que estão sendo trabalhados.
- A forma em que a estrutura de decisão está sendo construída.

O algoritmo quando se encontra no momento de seu desenvolvimento demandará os "Comandos de Decisão ou Desvios". Estes, naturalmente, farão parte das técnicas de programação necessárias para a condução das estruturas dos programas que não são totalmente sequenciais com as instruções de "DESVIO" e/ou "SALTO", com as quais podemos permitir que o programa aja de uma forma ou outra, conforme as decisões lógicas tomadas em função dos dados ou resultados anteriores.

Temos dentro do âmbito do desenvolvimento de um programa as principais estruturas de decisão, que, de forma generalista, são: "Se"; "Se Então", "Se Então Senão" e "Caso Selecione".

Sabemos que há diversas formas para a construção de uma sentença utilizando a estrutura de decisão.

É fato que isso também está atrelado à questão da demanda da aplicação.

Observação

Existirão diversos níveis de necessidade de entendimento de uma questão, e, com base nisso, será aplicado um nível de estrutura de decisão.

Operadores relacionais

Os operadores **<** e **>** **(menor e maior)**, que foram utilizados no exemplo anterior, são caracterizados como os **operadores relacionais**, pois permitem entender a relação existente entre seus dois operandos. Além desses dois, existem outros seis operadores relacionais, que serão tratados na tabela a seguir:

Operadores relacionais	
=	Atribuição
!=	Diferente
>	Maior
<	Menor
< =	Menor igual
>=	Maior igual
==	Igual
%	Resto da divisão entre números inteiros

Ao trabalharmos com estrutura de decisão, utilizamos todos os operadores listados, com exceção do primeiro — voltado apenas para a atribuição de valores para as variáveis.

Operadores lógicos

Além dos operadores relacionais, existem os chamados **operadores lógicos** — também conhecidos como **conectivos lógicos** —, cuja função é conectar duas expressões relacionais.

Operadores lógicos	
&&	And(e)
||	Or (ou)
!	Not (não)

Na tabela acima, podemos verificar os operadores lógicos da linguagem C.

O operador lógico utilizado para tratar as demandas de **Negação** deve ser usado **antes de uma expressão relacional**.

Exemplo

!(4 > 2): resultado **FALSO**
!(7 < 4): resultado **VERDADEIRO**

Precedência de operadores

!	**Operador de Negação**	Executado antes da condição.
&&	**AND Operador Lógico**	Executado depois da condição. Liga duas condições.
||	**OR Operador Lógico**	

Comando SWITCH

É uma opção bem interessante para que haja uma redução na complexidade de vários **if else encadeados**. É muito utilizado, principalmente, quando há demandas de uso em uma ou mais estruturas de menu.

O conteúdo atribuído a uma variável será comparado a um valor constante, e, caso a comparação seja verdadeira, um ou mais comandos específicos serão executados de forma sequencial.

Exemplo

Pseudocódigo com o uso do comando SWITCH:

```
Escolha (Variável)
Inicio
  Caso (numero 1):
    Instruções;

  Caso (numero 2):
    Instruções;

  Caso (numero x):
    Instruções;
Fim;
```

Algumas considerações importantes acerca do desenvolvimento de um programa em linguagem C utilizando o comando SWITCH:

- O comando SWITCH só trabalha com variáveis do parâmetro do tipo **int** ou **char**.
- O valor utilizado após a opção **case** deve ser uma **CONSTANTE**.
- A não colocação do **break** permitirá que o controle passe para o próximo **case** ao invés de interromper a verificação.
- No caso do **default** não há a necessidade de uso do **break**, pois se trata da última sentença, e não há mais nada a ser executado.

Exemplo

```
#include <stdio.h>
int op;

printf("Digite a sua opção\n");
scanf ("%d",&op);

switch (op)
{
case numero_1:
   comando_1; // executa estes comandos se o conteúdo da variável op for igual
a numero_1
    comando_2;
    break;
case numero_2:
   comando_3; // executa estes comandos se o conteúdo da variável op for igual
a numero_2
    comando_4;
    break;
default:
   comando_5; // executa estes comandos se o conteúdo da variável op não for
igual a numero_1 e numero_2
    comando_6;
}
```

Fazendo comparações

Nesse tipo de estrutura de decisão do Switch Case, o fluxo das instruções desenvolvido em uma aplicação é escolhido por causa do resultado da avaliação de uma ou mais condições existentes.

Uma condição existente é, na verdade, uma expressão lógica.

A classificação existente das estruturas de decisão é feita conforme o número de condições que devem ser verificadas para que se decida qual o melhor caminho a ser percorrido.

Há dois tipos de estrutura de decisão que podemos utilizar:

- Se → usado pelo comando **if**().
- Escolha → usado pelo **comando switch()**.

Importante

Para a utilização do **if**, os dados a serem comparados precisam ser do mesmo tipo.

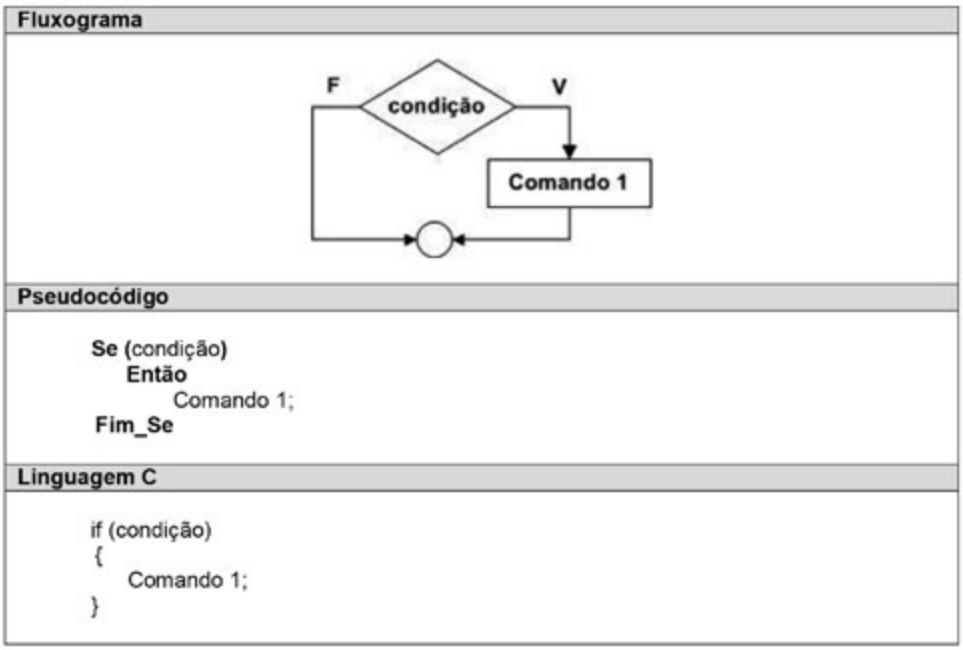
O **if** possui seus próprios parênteses, e a condição deverá sempre estar dentro deles.
if (a > b)

Nada impede que você também possa, por algum motivo de controle, usar um parêntese próprio para a condição.
if ((c < j))

a. if()

Nesta estrutura de decisão, uma única condição (expressão lógica) será avaliada:

- Caso o resultado oriundo dessa avaliação seja verdadeiro (V), então um determinado conjunto de instruções (comandos compostos) será executado.
- Caso contrário, ou seja, quando o resultado da avaliação for falso (F), nada será executado, e o fluxo de execução seguirá para o primeiro comando após a finalização da estrutura.



Fonte: docplayer.com.br

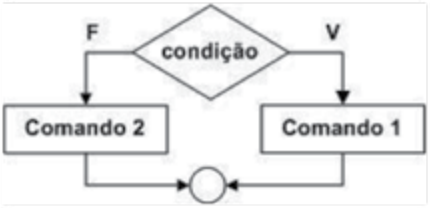
b. if() else

Nesta estrutura de decisão, uma única condição (expressão lógica) é, plenamente, avaliada:

- Se o resultado dessa avaliação for verdadeiro (V), então um determinado conjunto de instruções (comandos compostos) será executado.
- Caso contrário, ou seja, quando o resultado da avaliação for falso (F), um comando diferente será executado.

Após a execução de uma das partes (verdadeira ou falsa), o fluxo de execução seguirá para o primeiro comando após o fim da estrutura.

Fluxograma



Pseudocódigo

```
Se (condição)
    Então
        Comando 1;
    Senão
        Comando 2;
Fim_Se
```

Linguagem C

```
if (condição)
{
    Comando 1;
}
else
{
    Comando 2;
}
```

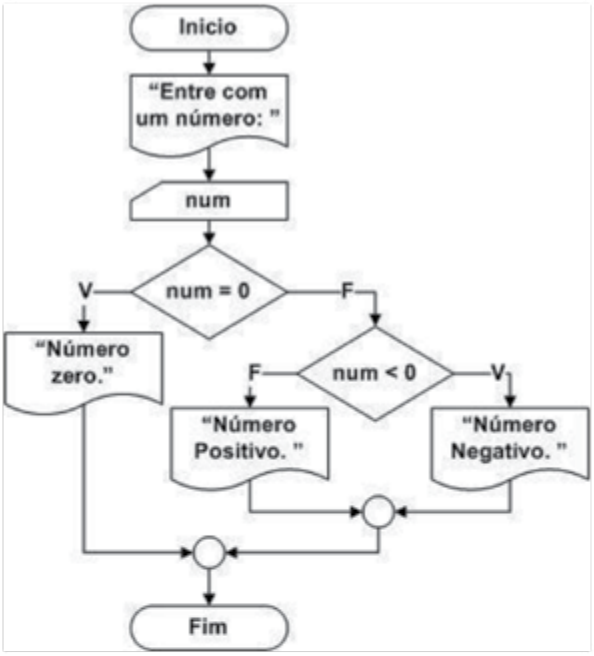
Fonte: docplayer.com.br

c. Encadeamento de if() ou if() aninhados

Permite ao desenvolvedor utilizar uma estrutura if para tratar uma situação e, dentro dessa estrutura já estabelecida, poder inserir uma nova estrutura de if para obter diversas respostas possíveis e atender à demanda.

É muito comum o desenvolvedor se deparar com esse tipo de situação no desenvolvimento de aplicações.

Fluxograma



Pseudocódigo

```
Algoritmo Encadeamento;
Var num: inteiro;
Inicio
        Escreva ("Entre com um número: ");
        Leia (num);
        Se (num = 0)
            Então
                Escreva ("Número zero.");
            Senão
                Se ( num < 0)
                  Então
                      Escreva ("Número negativo.");
                  Senão
                      Escreva ("Número positivo.");
                Fim_Se
        Fim_Se
Fim
```

Linguagem C

```
#include <stdio.h>
main()
{       int num;
        printf("Entre com um numero: ");
        scanf("%d",&num);
        if(num == 0)
        {       printf("Numero zero.\n");    }
        else
        {       if(num<0)
                {       printf("Numero negativo.\n");   }
                else
                {       printf("Numero positivo.\n");     }
        }
}
```

Fonte: docplayer.com.br

Importante

Se houver mais de uma linha de instrução que pertença ao if ou ao else, será fundamental que haja a abertura e o fechamento das chaves "{}". Caso contrário, apenas a primeira linha de instrução estará ligada ao if ou ao else. A segunda linha estará fora dessa estrutura de decisão e será executada independentemente da referida estrutura de decisão, causando um erro de lógica no programa.

Vejamos, a seguir, um exemplo de **encadeamento de if()**.

Exemplo

Programação correta

```
#include <stdio.h>
int main ()
{
        int num;
        printf ("Digite um numero: ");
        scanf ("%d", &num);
        if (num > 10)
        {
                printf ("\nO numero e maior que 10");
                printf ("\nObrigado por digitar um numero valido!");
        }
return(0);
}
```

Programação com erro

```
#include <stdio.h>
int main ()
{
        int num;
        printf ("Digite um numero: ");
        scanf ("%d", &num);
        if (num > 10)
                printf ("\nO numero e maior que 10");
                printf ("\nObrigado por digitar um numero valido!"); // mesmo estando posicionado dentro do if o printf acima não pertence ao if. Será executado independentemente de a variável num ser ou não maior que 10.
return(0);
}
```

Observação

Mesmo com uma única linha no if ou no else, é possível fazer a colocação das chaves "{}". Isso não produzirá nenhum tipo de erro para o código.

d. if() else if()

Estrutura utilizada para a aplicação de mais de uma estrutura de decisão.

É comum que o desenvolvedor tenha que verificar mais de uma condição dada à demanda surgida.

Para isso, ele aplica esta estrutura do **else if** em vez de, após o else, inserir na linha seguinte o comando if, que pertence à estrutura do else. Isso daria mais trabalho e poderia também atrapalhar o desenvolvedor na hora de fazer uma determinada revisão em seu código.

Exemplo

```
#include <stdio.h>
int main ()
{
        int num;
        printf ("Digite um numero: ");
        scanf ("%d", &num);
        if (num > 10)
                printf ("\n\nO numero e maior que 10");
        else if (num == 10)
        {
                printf ("\n\nVoce acertou!\n");
                printf ("O numero e igual a 10.");
         }
        else if (num < 10)
                        printf ("\n\nO numero e menor que 10");
return(0);
}
```

Condições compostas

Vamos tratar da questão da utilização do if com o uso de mais de uma condição — neste caso, os operadores relacionais farão uma ligação entre as **condições compostas**.

i. Uso do operador lógico E

Vejamos como se dá o uso do operador lógico E — representado pelo símbolo "&&" — em um programa desenvolvido em linguagem C.

No exemplo a seguir, para as condicionais idade **menor que 20** e sexo igual a m, será exibida na tela a informação "**Ação Permitida**", ou seja, há uma dupla verificação:

- Da variável **idade**.
- Da variável **sexo**.

Ambas as condições precisam ser verdadeiras. Caso contrário, basta que uma das condições seja falsa para que a informação exibida seja "**Ação Não Permitida**".

Exemplo

```
#include <stdio.h>
main()
{
        int idade = 0;
        char sexo;

printf ("Informe seu sexo: ");
        scanf ("%s", &sexo);

printf ("Informe sua idade: ");
        scanf ("%d", &idade);

        if ((idade < 20) && (sexo == 'm'))
        {
                printf ("Ação Permitida");
        else
                printf ("Ação Não Permitida");
        }
}
```

Vamos aumentar o grau de complexidade?

O código a seguir tratará das condicionais nas quais, sendo a idade **maior ou igual a 20 e menor** ou **igual a 50** e o sexo igual a **m**, será exibida na tela a informação "**Ação Permitida**". Isto é, há:

- Uma dupla verificação da variável idade.
- Uma verificação específica da variável sexo.

Nesse caso, ambas as condições precisam ser verdadeiras. Se uma das condições for falsa, a informação exibida será "**Ação Não Permitida**".

Observação

Repare que há o uso de parênteses para proteger a condição **((idade >= 20) && (idade <= 50))**. Com isso, ela trabalha de forma composta.

Exemplo

```
#include <stdio.h>
main()
{
        int idade = 0;
        char sexo;

printf ("Informe seu sexo: ");
        scanf ("%s", &sexo);

printf ("Informe sua idade: ");
        scanf ("%d", &idade);

if (((idade >= 20) && (idade <= 50)) && (sexo == 'm'))
                printf ("Ação Permitida");
        else
                printf ("Ação Não Permitida");
 }
```

ii. Uso do operador lógico OU

Agora, confira um exemplo de um programa desenvolvido em linguagem C no qual há o uso do operador lógico OU, representado pelo símbolo "||".

O código a seguir tratará das condicionais nas quais, a idade sendo **menor que 20** e o sexo igual a m, será exibida na tela a informação "Ação Permitida", ou seja, há uma dupla verificação da **variável idade** e da **variável sexo**. Contudo, apenas uma das condições precisa ser verdadeira — idade ou sexo. Apenas se ambas as condições forem falsas, a informação exibida será "**Ação Não Permitida**".

Exemplo

```
#include <stdio.h>
main()
{
        int idade = 0;
        char sexo;

printf ("Informe seu sexo: ");
        scanf ("%s", &sexo);

printf ("Informe sua idade: ");
        scanf ("%d", &idade);

        if ((idade < 20) || (sexo == 'm'))
        {
                printf ("Ação Permitida");
        else
                printf ("Ação Não Permitida");
        }
}
```

Aumentemos a complexidade da programação...

O código a seguir trata das condicionais nas quais, a idade sendo maior ou igual a 40 ou o sexo sendo igual a f ou a m, será exibida na tela a informação “Ação Permitida”. Isto é, há:

- Uma tripla verificação da variável idade.
- Duas verificações específicas da variável sexo.

Nesse caso, apenas uma das condições precisa ser verdadeira. Apenas se todas as condições forem falsas, a informação exibida será “Ação Não Permitida”.

Exemplo

```
#include <stdio.h>
main()
{
        int idade = 0;
        char sexo;

printf ("Informe seu sexo: ");
        scanf ("%s", &sexo);

printf ("Informe sua idade: ");
        scanf ("%d", &idade);

        if ((idade >= 40) || (sexo ==‘f’) || (sexo == ‘m’))
                printf ("Ação Permitida");
        else
                printf ("Ação Não Permitida");
 }
```

Vejamos mais um exemplo de programação utilizando o operador lógico OU.

O código a seguir trata das condicionais nas quais, a idade sendo **maior que 30**, o sexo sendo igual a **f** e a altura sendo **igual a 1,75**, a informação "**Ação Permitida**" será exibida na tela do computador, ou seja, há uma tripla verificação das variáveis idade, sexo e altura. Nesse caso, se uma das condições for verdadeira, teremos o sucesso da ação. Caso contrário, se todas as condições forem falsas (idade, sexo e altura), a informação exibida será "**Ação Não Permitida**".

Exemplo

```
#include <stdio.h>
main()
{
        int idade = 0;
        float altura;
        char sexo;

printf ("Informe seu sexo: ");
        scanf ("%s", &sexo);

printf ("Informe sua idade: ");
        scanf ("%d", &idade);


printf ("Informe sua altura: ");
        scanf ("%f", &altura);

        if ((idade > 30) && (sexo ==‘f’) && (altura == 1.75))
                printf ("Ação Permitida");
        else
                printf ("Ação Não Permitida");
 }
```

iii. Uso dos operadores lógicos E e OU

Vejamos o exemplo de um programa desenvolvido em linguagem C em que há o uso do:

- Operador lógico E, representado pelo símbolo “&&”.
- Operador lógico OU, representado pelo símbolo “||”.

O código a seguir trata das condicionais nas quais, a idade sendo **maior ou igual a 20** e **menor ou igual a 50** ou o sexo igual a **m**, será exibida na tela a informação “**Ação Permitida**”, ou seja, há uma dupla verificação da variável idade e mais uma verificação da variável sexo. Nesse caso, apenas uma das condições precisa ser verdadeira — a combinação do intervalo das idades ou sexo. Se ambas as condições forem falsas, a informação a ser exibida será “**Ação Não Permitida**”.

Observação

Repare que há um uso de parênteses para proteger a condição **((idade >= 20) && (idade <= 50))**. Com isso, ela trabalha de forma composta.

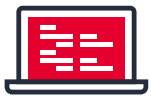
Exemplo

```
#include <stdio.h>
main()
{
        int idade = 0;
        char sexo;

printf ("Informe seu sexo: ");
        scanf ("%s", &sexo);

printf ("Informe sua idade: ");
        scanf ("%d", &idade);


if (((idade >= 20) && (idade <= 50)) || (sexo == 'm'))
                printf ("Ação Permitida");
        else
                printf ("Ação Não Permitida");
 }
```



MIDIATECA

Para ampliar seu conhecimento, acesse o material complementar da Unidade 3 disponível na midiateca.

NA PRÁTICA

A empresa de Jogos USART trabalha com o desenvolvimento de soluções tecnológicas para crianças de dois a seis anos de idade. Esse ano, a USART se associou a uma escola de Ensino Básico para criar um aplicativo capaz de identificar para a criança qual é o maior número digitado entre dois números. Assim, esta aplicação ajudará a escola nas disciplinas iniciais de matemática.

```
#include <stdio.h>
main()
{
  int x, y;
  printf("Entre com o primeiro valor: ");
  scanf("%d", &x);

  printf("Entre com o segundo valor: ");
  scanf("%d", &y);

  if ( x > y )
    printf ("O primeiro valor é maior que o segundo");
  else
    printf ("O segundo valor é maior que o primeiro");
}
```

Resumo da Unidade 3

Nesta unidade estudamos a importância das estruturas de decisão dentro do âmbito do desenvolvimento de uma aplicação e abordamos vários formatos para o trabalho com as condições, bem como o que pode acontecer quando uma estrutura de decisão não é bem definida. Dessa forma, vimos como utilizar as estruturas de decisão para as devidas operações de comparações com o uso dos operadores relacionais e lógicos.

CONCEITO

Na terceira unidade verificamos a existência de vários algoritmos para testes de informações e ordenação de valores.

Referências

GUEDES, S. **Lógica de programação algorítmica.** São Paulo: Pearson Education do Brasil, 2014. Biblioteca Virtual.

MENDES, F. V. **Programação avançada em C++.** São Paulo: Pearson Education do Brasil, 2006. Biblioteca Virtual.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos.** 3. ed. Rio de Janeiro: LTC, 2010. Minha Biblioteca.

UNIDADE 4

Estrutura de repetição

INTRODUÇÃO

Vamos conhecer os fundamentos do comando while e do comando for, visando ao uso das estruturas de repetição condicionadas a uma ou mais condições baseadas em operadores relacionais e lógicos. Além disso, abordaremos outra demanda importante: o uso assertivo das funções para que haja otimização no código do programa.

OBJETIVO

Nesta unidade você será capaz de:

- Construir estruturas de repetição para que determinado código execute até uma condição ser satisfatória, bem como a aplicação de uma ou mais funções para otimizar determinado código e permitir que ele seja repetido conforme demanda.

Quando utilizar laços

Dentro da concepção da lógica de programação, a estrutura de repetição permite executar mais de uma vez a mesma instrução ou conjunto de instruções, de acordo com uma ou mais condições ou com um contador específico.

São utilizadas para executar a repetição de ações semelhantes às executadas para todos os elementos de determinada lista de dados — ou, simplesmente, para que haja a repetição de um mesmo processamento até que determinada condição seja satisfeita.

1. Comando while

O **comando *while*** permite que o código existente dentro da estrutura de repetição seja executado por diversas vezes até que a condição seja satisfatória, ou seja, enquanto a condição estabelecida não atinge o seu cenário estabelecido.

Enquanto for verdadeira a condição, o conjunto de códigos se repete com determinado propósito a ser alcançado.

Sintaxe:

```
while (condição)
{
    <comando>;
        <comando>;
}
```

Exemplo

```
#include<stdio.h>
main()
{
int i;
i = 0;
while (i<=4) // (i <=4) é a condição. Quantas vezes a estrutura irá repetir
    {
      printf("Número %d\n", i);
          i++; // contador. A cada passagem o i é incrementado com mais 1
        }
}
```

Vejamos, a seguir, algumas questões apresentadas para o comando *while*:

A estrutura apresentada anteriormente está entre as três mais simples.

Há a repetição de um bloco de instruções enquanto uma condição for verdadeira. Caso a condição seja falsa, as instruções dentro do comando *while* não serão executadas e a execução continuará com os comandos após o término do *while*.

A repetição acontecida no while será controlada por determinada condição que verifica uma variável. Contudo, para que o *while* funcione de acordo é importante que essa variável sofra alteração dentro do *while* – o contador precisa ser incrementado ou decrementado.

Exemplo

Exemplo 1

```
#include<stdio.h>
main()
{
int i;
i = 6;
while (i > 0) // (i > 0) é a condição. Quantas vezes a estrutura irá repetir
    {
            printf("O número %d\n", i);
            i--; // contador. A cada passagem o i é decrementado com menos
1. Ou seja, o valor retroage
            }
}
```

Exemplo 2

```
#include <stdio.h>
main()
{
int i, x;
i = 1;
x = 2;
while ((i<=7) || (x < 5)) // trabalha com dois contadores i e x.
{
      printf("O valor de i eh:  %d\n", i);
      printf("O valor de x eh:  %d\n", x);
      i++;
      x++;
}
}
```

A primeira condição que atingir seu valor encerrará a execução da estrutura de repetição.

2. Comando for

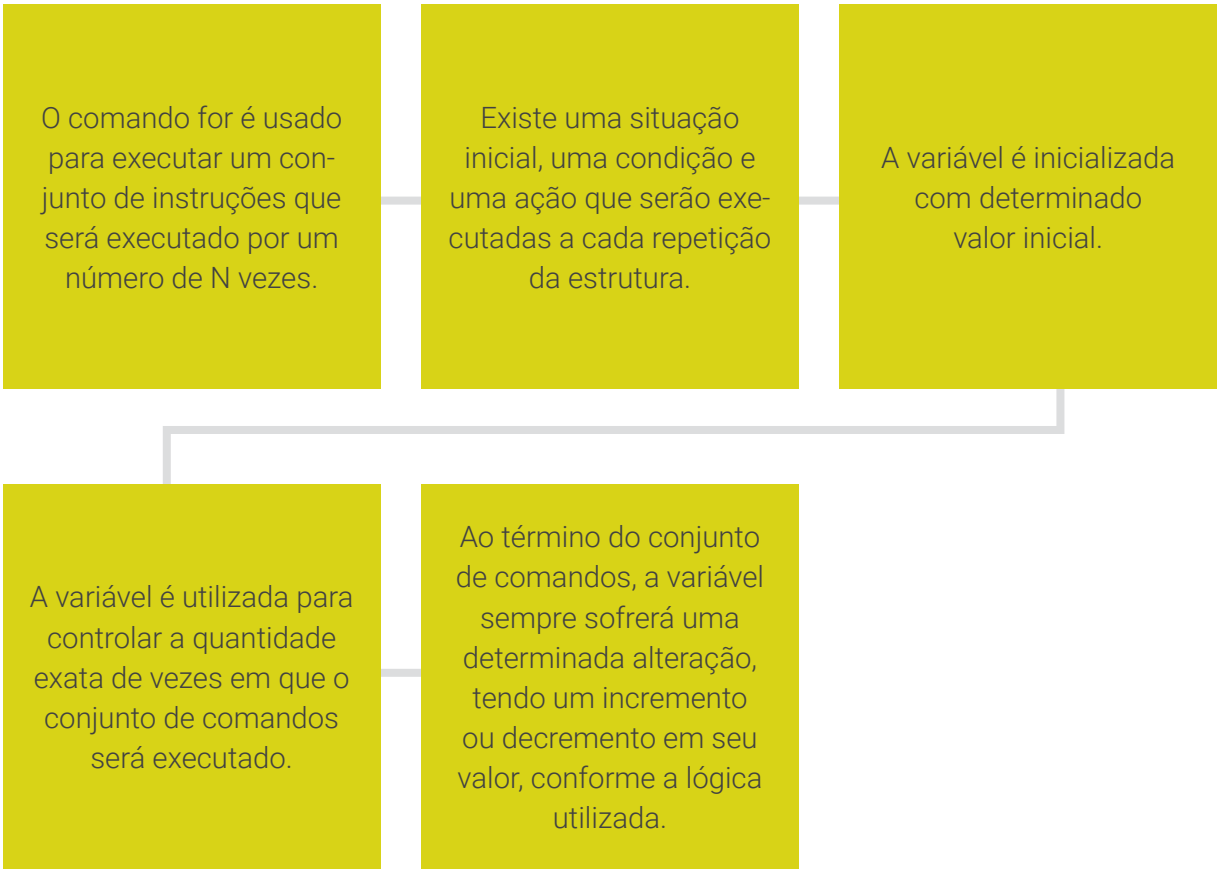
O **comando *for*** é, de alguma maneira, encontrado em todas as linguagens procedurais de programação:

- A sua condição é uma expressão de relação que testa a variável de controle do loop contra algum valor para determinar quando o loop terminará.
- O incremento define a maneira como a variável de controle do loop será alterada cada vez que o computador repetir o loop.

Sintaxe:

```
for (inicialização; condição; incremento)
{
        <comando>;
        <comando>;
        <comando>;
}
```

Vejamos, a seguir, como funciona o comando for:



Sintaxe:

```
For (inicialização; condição; incremento)
        {
<comando>;
<comando>;
```

Exemplo

```
#include<stdio.h>
main ()
{
int i, idade;
for (i=0; i<=6; i++) // controla a inicialização, a condição e o incremento
   {
        printf (“Digite uma idade”);
                scanf (“%d”, &idade);
                printf (“A idade digitada foi %d \n”);
   }
}
```

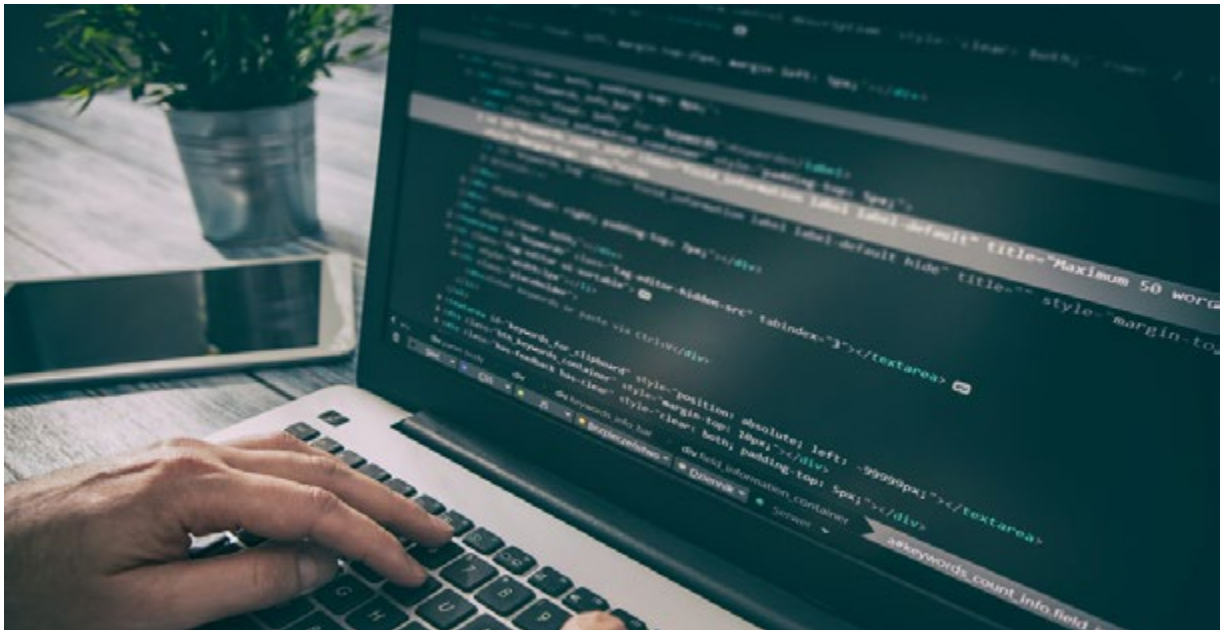
3. Diferença entre o comando while e o comando for

Dentro da estrutura de repetição do while há uma linha de código a mais por conta do incremento do contador. A quantidade total de linhas do código com o uso do *while* também é maior que a do *for* por conta da inicialização da variável de controle.

```
#include <stdio.h>
main ()
{
int i, idade;
i = 0;
while (i<=4)
   {
      printf (“Digite uma idade”);
      scanf (“%d”, &idade);
      i++;
   }
}
```

```
#include <stdio.h>
main ()
{
int i, idade;
for (i=0; i<=4; i++)
   {
      printf (“Digite uma idade”);
      scanf (“%d”, &idade);
   }
}
```

Estrutura de laços



A estrutura de repetição, quando criada, serve para tratar instruções que podem ser executadas de diferentes formas.

Veremos, neste tópico, a utilização da estrutura de repetição por meio do uso do **comando while** e do **comando for**.

1. Comando while

Testa a condição antes de executar o laço.

Sintaxe:

```
while (condição)
{
    <comando>;
            <comando>;
}
```

Exemplo

Exemplo 1

Quando a condição atingir o valor de x = 3, a estrutura de repetição parará de executar, uma vez que a condição estabelece que a execução aconteça enquanto x <= 2.

```
#include <stdio.h>
main()
{
int x;
x = 0;
while (x <= 2) // (x <=2) é a condição. Quantas vezes a estrutura irá repetir
        {
                printf ("Número %d\n", x);
                x++; // contador. A cada passagem o i é incrementado com
mais 1
                }
}
```

Exemplo 2

Quando a condição atingir o valor de y = 0, a estrutura de repetição parará de executar, pois a condição estabelece que a execução aconteça enquanto y > 0.

```
#include<stdio.h>
main()
{
int y;
y = 7;
while (y > 0) // (y > 0) é a condição. Quantas vezes a estrutura irá repetir
        {
        printf("Número %d\n", y);
        y--; // contador. A cada passagem o i é decrementado com menos 1.
Ou   seja, o valor retroage
        }
}
```

Exemplo 3

A primeira condição que atingir seu valor — se i for igual a 5 ou se x for igual a 3 — fará com que a estrutura de repetição pare de executar.

```
#include <stdio.h>
main()
{
int i, x;
i = 0;
x = 1;
while ((i <= 4) || (x < 2))
{
                printf (“Valor de i eh:  %d\n", i);
                printf (“Valor de x eh:  %d\n", x);
                i++;
                x++;
}
}
```

Exemplo 4

A seguir, veremos uma estrutura de repetição dentro de outra.

Temos um while dentro de outro while, em que:
Para cada execução do primeiro while temos diversas execuções do segundo while.
A cada passagem do primeiro while o segundo while irá executar quatro vezes.

```
#include<stdio.h>
main()
{
int x, y;
y = 3;
while (y > 0) // (y > 0) é a condição. Quantas vezes a estrutura irá repetir
        {
                        printf ("Numero %d\n", y);
                        x = 4;
while (x > 0) // (x > 0) é a condição. Quantas vezes a estrutura irá repetir
                {
                                printf ("Numero %d\n", x);
                                x--;
                        }
y--; // contador. A cada passagem o i é decrementado com menos 1. Ou seja, o valor retroage
                }
}
```

2. Comando for

Conforme visto anteriormente, o incremento ou decremento definirão a maneira como a variável de controle do loop será alterada cada vez que o computador repetir o loop:

- O incremento aumenta um na variável de controle.
- O decremento diminui um na variável de controle.

Sintaxe:

```
for (inicialização; condição; incremento)
{
        <comando>;
        <comando>;
        <comando>;
}
```

Exemplo

Exemplo 1

A digitação da idade vai se repetir por quatro vezes, pois a variável de controle começa com valor zero. Assim, temos a execução para valores 0, 1, 2 e 3, pois a condição estabelece que x<=3.

```
#include <stdio.h>
main ()
{
int x, idade;
for (x=0; x<=3; x++) // controla a inicialização, a condição e o incremento
   {
      printf ("Digite uma idade");
            scanf ("%d", &idade);
            printf ("A idade digitada foi %d \n");
   }
}
```

Exemplo 2

A digitação da matrícula vai se repetir por três vezes, pois a variável de controle começou com o valor zero. Ainda dentro da primeira estrutura de repetição temos mais uma execução de um comando for, ou seja, este segundo comando pertence ao primeiro e será executado duas vezes. Assim, para cada matrícula digitada, será necessária a digitação de mais duas notas.

```
#include <stdio.h>
main ()
{
int x, y, matricula;
float nota;
for (x=0; x<=2; x++) // controla a inicialização, a condição e o incremento
   {
        printf (“Digite uma matricula”);
                scanf (“%d”, &matricula);

for (y=0; x<=1; y++) // controla a inicialização, a condição e o incremento
        {
                printf (“Digite a nota”);
                        scanf (“%f”, &nota);
                }
   }
}
```

Funções

A função pode ser executada repetidas vezes, bem como, dentro do seu código, pode abrigar o uso dos comandos *while* e *for*.

As funções são um conjunto de instruções constituídas em um bloco que recebe determinado nome e, por meio deste, pode ser executado.

A função será chamada pela função principal (**main()**) ou por intermédio de outra função. Isto será determinado pelo objetivo estabelecido.

Por que usar funções?

Para permitir o reaproveitamento de código já existente.	Para permitir que o código seja executado diversas vezes.	Para evitar que determinado trecho de um programa seja repetido várias vezes dentro do mesmo programa.
Para permitir a alteração de um código ou trecho de código de uma forma mais rápida sendo preciso alterar apenas dentro da função o que se deseja.	Para que os blocos do programa não fiquem grandes demais e, por consequência, mais difíceis de entender.	Para separar o programa em partes (blocos de código) a fim de que possam ser logicamente compreendidos.

Parâmetros

Os parâmetros possibilitam que se defina sobre quais dados a função irá trabalhar.

Para que haja a definição dos parâmetros em uma função, o programador deverá explicitá-los como se estivesse declarando uma variável, entre os parênteses do cabeçalho da função.

Caso haja a necessidade de declarar mais de um parâmetro, basta separá-los por vírgulas.

Exemplo

A seguir temos a função MULT que possui dois parâmetros — sendo o primeiro um int e o segundo outro int.

```
#include <stdio.h>
int MULT (int val1, int val2)  // basta separar os parâmetros por vírgula
{
  int resultado;    // declaração de variável
  resultado = val1 * val2;
  printf ("A multiplicação de %d com %d é %.d", val1, val2, resultado);
}
```

Os parâmetros da função:

- No momento de sua declaração, são denominados parâmetros formais.
- Na chamada da função, são denominados parâmetros atuais.

Os parâmetros são passados para uma função de acordo com a sua posição. Ou seja, o primeiro parâmetro atual define o valor do primeiro parâmetro formal e assim por diante.

Observação

Os nomes dos parâmetros na chamada não têm relação com os nomes dos parâmetros na definição da função.

Vejamos, a seguir, um exemplo de programação SEM passagem e recebimento de parâmetros:

Exemplo

O programa principal, após exibir determinada informação na tela para o usuário, fará a limpeza das informações. Os comandos para a execução da limpeza deverão ser desenvolvidos em uma nova função de nome "**limpa**".

```
#include <stdio.h>
#include <stdlib.h>
void limpa () // não há necessidade de recebimento de parâmetros
{
  system ("cls");
}

main()
{
   printf ("Olá!");
   limpa(); // chamada da função
}
```

Neste exemplo não houve a necessidade de passagem nem de recebimento de parâmetro pelo fato de a função executar apenas um comando de limpeza.

Assim, o comando em si não necessita de parâmetro para a sua execução.

Agora, veremos dois exemplos COM passagem e recebimento de parâmetros.

Exemplo

1. Fazer um programa no qual o usuário possa digitar dois valores e que fará a sua soma. O cálculo da soma deverá ser feito em uma nova função de nome "**soma**".

```
#include <stdio.h>
int soma (int num_1, int num_2)  // recebimento dos parâmetros oriundos da função main()
{
  int resultado;    // declaração de variáveis
  resultado = num_1 + num_2;
  printf ("A soma de %d com %d é %d", num_1, num_2, resultado);
}
main()
{
   int val_1;
   int val_2;
   val_1 = 10;
   val_2 = 12;
   soma (val_1, val_2); // chamada da função com a passagem dos parâmetros
}
```

2. Fazer um programa em que o usuário possa digitar dois valores e o programa fará a subtração desses valores. O cálculo da subtração deverá ser feito em uma nova função de nome "sub".

```
#include <stdio.h>
int sub (int num_1, int num_2)
{
  int valor;
  valor = num_1 - num_2;
  return valor; // retorna o conteúdo da variável valor para a função main()
}

main()
{
   int num_1;
   int num_2;
   int resultado;
   printf ("Digite o primeiro valor: ");
   scanf( "%d", &num_1);
   printf ("Digite o segundo valor: ");
   scanf( "%d", &num_2);
   resultado = sub (num_1, num_2);  // chamada da função soma(). Variável resultado recebe o valor da variável valor enviada pelo return da função soma().
   printf ("A subtração de %d com %d é %d", num_1, num_2, resultado);
}
```

Veremos, agora, um exemplo de aplicação de uma função na qual são usados mais de dois parâmetros — assim podemos utilizar vários parâmetros para atender a demanda.

Exemplo

Fazer um programa para criar uma função que leia cinco valores inteiros e imprima o maior e o menor valores. As entradas de dados devem ser na função principal.

```
#include <stdio.h>
int maior (int n1, int n2, int n3, int n4, int n5)
{
   int aux;
   if (n1>n2)
         aux=n1;
   else
         aux=n2;
   if (n3>aux)
         aux=n3;
   if (n4>aux)
         aux=n4;
   if (n5>aux)
         aux=n5;
   printf ("\nO MAIOR NUMERO DIGITADO FOI: %d .",aux);

   if (n1<n2)
          aux=n1;
   else
         aux=n2;
   if (n3<aux)
         aux=n3;
   if (n4<aux)
         aux=n4;
   if (n5<aux)
         aux=n5;
   printf ("\nO MENOR NUMERO DIGITADO FOI: %d .",aux);
}
```

```
main ()
{
        int x1,x2,x3,x4,x5;

printf ("Digite o Primeiro Numero: ");
        scanf ("%d", &x1);

printf ("Digite o Segundo Numero: ");
        scanf ("%d", &x2);

printf ("Digite o Terceiro Numero: ");
        scanf ("%d", &x3);

printf ("Digite o Quarto Numero: ");
        scanf ("%d", &x4);

printf ("Digite o Quinto Numero: ");
        scanf ("%d", &x5);


        maior (x1, x2, x3, x4, x5);
}
```

Para encerrarmos o conteúdo desta disciplina, veremos agora uma função na qual há retorno de informações para a função main() e o conteúdo da variável continua sendo trabalhado.

Exemplo

Fazer um programa para criar uma função que retorne um valor para identificar que tipo de número inteiro o usuário digitou. Será necessário exibir na tela a mensagem:

- Número Par, caso o usuário tenha digitado um número par na função principal.
- Número Ímpar, caso o usuário tenha digitado um número ímpar na função principal.
- Número Negativo, caso o usuário tenha digitado um número negativo na função principal.

Caso seja digitado um valor zero, este deve ser criticado e o programa deverá retornar para nova digitação.

```
#include <stdio.h>
int numeroc(int n)
{
        if (num < 0)
                return -1;
else if (num %2 ==0)
                return 0;
        else
                return 1;
}

main ()
{
int num, res;

inicio:
printf ("\nDIGITE UM NUMERO INTEIRO.\n");
scanf ("%d", &num);
if (num ==0)
{
        printf (“Numero não pode ser zero”);
        goto inicio;
}

res = numero(num); // chama a função

if (res == 0)
        printf ("\nNumero Positivo\n");
else if (res == 1)
        printf ("\nNumero Impar.\n\n");
else
        printf ("\nNumero Negativo\n\n");
}
```

MIDIATECA

Para ampliar seu conhecimento, acesse o material complementar da Unidade 4 disponível na midiateca.

NA PRÁTICA

Uma empresa de tecnologia está recrutando um programador que vai compor o quadro de funcionários para desenvolvimento de aplicações em linguagem C. Para tal, a empresa criou um processo seletivo a fim de verificar qual candidato irá criar o melhor código.

O desafio será o desenvolvimento de um algoritmo para criar uma função que retorne um valor para identificar que tipo de número inteiro o usuário digitou. Será necessário exibir na tela a mensagem:

Número Par, caso o usuário tenha digitado um número par na função principal;
Número Ímpar, caso o usuário tenha digitado um número ímpar na função principal;
Número Negativo, caso o usuário tenha digitado um número negativo na função principal.

```
#include <stdio.h>
int numero(int n)
{
        if (n < 0)
                return -1;
else if (n %2 ==0)
                return 0;
        else
                return 1;
}
```

```
main ()
{
int num = 0, res = 0;

while (num == 0)
{
printf ("\nDigite um numero inteiro: ");
scanf ("%d", &num);
}

res = numero(num); // chama a função

if (res==0)
        printf ("\nNumero Positivo\n");
else if (res==1)
        printf ("\nNumero Impar.\n\n");
else
        printf ("\nNumero Negativo\n\n");
}
```

Resumo da Unidade 4

Nesta unidade você estudou a importância de se manipular informações com base em uma estrutura de repetição, utilizando tanto o comando while como o comando for. Outro ponto de destaque foi o uso das funções que possibilitam a otimização de códigos em um programa.

CONCEITO

Uso de estruturas de repetição com os comandos while e for e o uso das funções para otimizar os códigos de um programa.

Referências

GUEDES, S. **Lógica de Programação Algorítmica.** São Paulo: Pearson Education do Brasil, 2015. Biblioteca Virtual.

MENDES, F. V. **Programação Avançada em C++.** São Paulo: Pearson Education do Brasil, 2006. Biblioteca Virtual.

SZWARCFITER, J. L.; MARKEZON, L. **Estruturas de dados e seus algoritmos.** 3. ed. Rio de Janeiro: Grupo GEN - LTC, 2013. Minha Biblioteca.

UVA

UNIJORGE