



MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS PATO BRANCO
Diretoria de Graduação e Educação Profissional
Coordenação do Curso de Engenharia de Computação
Coordenação do Curso Superior de Tecnologia em Análise e
Desenvolvimento de Sistemas



FUNDAMENTOS DE PROGRAMAÇÃO UTILIZANDO A LINGUAGEM C

Material elaborado pelos professores:
Beatriz T. Borsoi e Omero F. Bertol

Este material visa facilitar o entendimento dos conceitos básicos de uma linguagem de programação e auxiliar no uso desses conceitos de forma a definir a solução algorítmica para problemas computacionais. É utilizada a linguagem C para exemplificar esses conceitos, mas qualquer linguagem de programação que atenda aos princípios da programação estruturada pode ser utilizada. As diferenças estarão apenas na forma de expressar instruções e estruturas da linguagem. Essas diferenças podem ser simplesmente palavras distintas para o mesmo significado ou objetivo ou símbolos utilizados, por exemplo.

Para auxiliar na representação da solução de um problema de forma que ela possa ser executada por um computador, também são apresentados conceitos de lógica de programação e de algoritmos, além dos elementos básicos de algoritmos representados na linguagem C.

PATO BRANCO
2011

LISTA DE FIGURAS

Figura 1 – Do problema ao programa	4
Figura 2 – Partes fundamentais de um algoritmo	4
Figura 3 – Símbolos gráficos para representar algoritmos por meio de fluxograma.....	13
Figura 4 – Fluxograma para o cálculo do dobro de um número	13
Figura 5 - As palavras reservadas do C ANSI.....	20
Figura 6 – Do problema ao código executável.....	22
Figura 7 – Tipos de dados	25
Figura 8 – Operadores aritméticos.....	27
Figura 11 – Operadores relacionais	29
Figura 12 – Operadores lógicos.....	29
Figura 13 – Exemplo de uso de operadores lógicos	30
Figura 14 – Exemplos de expressões abreviadas.....	32
Figura 15 – Tabela de precedência dos operadores em C.....	32
Figura 16 – Códigos de controle.....	34
Figura 17 – Exemplos de uso de códigos de controle.....	34
Figura 18 – Caracteres de formatação	35
Figura 19 - Representação esquemática das estruturas de controle.....	37
Figura 20 – Fluxograma e programa C para calcular a área de um círculo	38
Figura 21 – Fluxograma e programa C para calcular a média de duas notas.....	39
Figura 22 - Fluxograma da estrutura de decisão if	41
Figura 23 – Fluxograma e programa C para indicar se um aluno está aprovado em decorrência da sua nota.....	41
Figura 25 - Fluxograma da estrutura de decisão if else	43
Figura 26 – Fluxograma e programa C para indicar se um aluno está aprovado ou reprovado em decorrência da sua nota	44
Figura 27 – Fluxograma e programa C para indicar se aluno está aprovado ou reprovado.....	46
Figura 28 – Modelo de indentação de estruturas if else.....	47
Figura 29 - Fluxograma da estrutura de decisão switch case.....	48
Figura 30 – Fluxograma e programa em linguagem C para números por extenso.....	49
Figura 35 - Fluxograma da estrutura de repetição for	51
Figura 36 - Algoritmo para imprimir a tabuada de determinado número.....	52
Figura 31 - Fluxograma da estrutura de repetição while	56
Figura 32 - Programa para imprimir a tabuada de um número.	56
Figura 33 - Fluxograma da estrutura de repetição do while.....	58
Figura 34 – Programa para ler números até que seja informado um número positivo.	59
Figura 37 – Representação esquemática de um vetor na memória.....	62

SUMÁRIO

Introdução.....	1
1 Lógica de Programação.....	2
2 Algoritmos.....	3
2.1 Elementos básicos de um algoritmo.....	5
2.2 Formas de representação de algoritmos.....	9
2.2.1 Descrição narrativa.....	10
2.2.2 Português Estruturado.....	11
2.2.3 Fluxograma.....	12
2.2.4 Linguagens de programação.....	14
2.2.5 Método para definir um programa.....	15
3 Elementos básicos de um algoritmo em linguagem de programação C.....	15
3.1 Conceitos iniciais.....	16
3.1.1 As palavras reservadas da linguagem C.....	19
3.1.2 Comentários.....	20
3.1.3 Instruções ou comandos.....	20
3.1.4 Funções.....	20
3.1.5 O processo de compilação.....	21
3.1.6 A linguagem C é estruturada.....	23
3.1.7 Estrutura básica de um programa.....	23
3.2 Tipos de dados.....	24
3.3 Variáveis e constantes.....	25
3.4 Operadores e expressões.....	27
3.4.1 Operadores Aritméticos.....	27
3.4.2 Operadores Relacionais e Lógicos.....	29
3.4.3 Expressões matemáticas.....	31
3.5 Instruções de entrada e saída.....	33
3.6 Estruturas de controle.....	37
3.6.1 Estrutura sequencial.....	37
3.6.2 Estruturas de controle decisão.....	40
3.6.3 Estruturas de controle repetição.....	50
3.7 Estruturas de dados homogêneos.....	61
3.7.1 Vetores.....	62
3.7.2 Matrizes.....	69
3.8 Estruturas de dados heterogêneos.....	71
3.9 Funções.....	74
3.9.1 Tipos de funções.....	76
3.9.2 Protótipos de funções.....	80
3.9.3 Escopo de variáveis.....	81
3.9.4 Passagem de parâmetros por valor e por referência.....	83
3.9.5 Vetores como argumentos de funções.....	84
3.9.6 Arquivos de cabeçalho.....	85

Introdução

De maneira bastante simplificada, programar é representar a solução de um problema de forma que o computador possa entendê-la e executá-la. Ressalta-se que um problema pode referir-se a alcançar um objetivo, fazer (reconhecer uma impressão digital, por exemplo) ou resolver algo, como uma expressão aritmética. Para que o problema possa ser resolvido é necessário que uma solução para o mesmo seja conhecida, isto é, que uma solução para o problema esteja definida ou que se saiba uma maneira de resolvê-lo. Contudo, além de saber a solução, é necessário expressá-la de maneira que o computador possa entendê-la e executá-la, em linguagem e formalismo próprios. E, ainda, que existam recursos computacionais para executar a solução definida.

A linguagem falada pelos humanos (denominada linguagem natural) é bastante distinta da linguagem de máquina (denominada linguagem binária). Uma linguagem de programação é intermediária entre a linguagem natural e a de máquina. Ela permite expressar ações humanas de forma que possam ser convertidas para a linguagem binária e, assim, sejam possam ser executadas pelo computador. Embora uma linguagem de programação deva atender a um determinado formalismo e ser entendida pelos mecanismos (programas interpretadores e compiladores) que transformam as instruções em linguagem de máquina, ela é mais facilmente entendida pelas pessoas do que a linguagem de máquina.

Uma linguagem de programação permite expressar as ações que resolvem um problema de forma sistematizada, definindo um algoritmo. Um algoritmo expressa a solução de um problema como uma sequência de ações. Assim, fazer um programa de computador é expressar a solução de um problema por meio de um algoritmo que atenda ao formalismo de uma linguagem de programação.

Atualmente existem diversas linguagens de programação em uso e são várias as que estão em desuso. Algumas dessas linguagens possuem aplicabilidade bem genérica e outras, bastante restrita. A aplicabilidade se refere ao tipo de problema que elas podem resolver ou que elas possuem recursos que facilitam implementação da solução. Os problemas cuja solução é implementada computacional são bastante distintos, como: informatização de uma loja comercial, página web de notícias, informatização do chão de fábrica de uma indústria, elaboração de desenhos técnicos e processamento matemático em aplicações científicas. Assim, é coerente que nem todas as linguagens de programação possuam recursos que implementem a solução para qualquer tipo de problema.

Cada linguagem de programação possui, basicamente, um vocabulário (palavras chave, palavras reservadas) e uma forma própria de estruturar as suas instruções. Contudo, a ideia básica do significado dessas formas é semelhante entre as linguagens - pelo menos nas de uso mais geral e que são definidas a partir do mesmo paradigma conceitual.

Uma linguagem de programação pode estar mais próxima (ser mais semelhante) à linguagem natural ou à linguagem de máquina, mas um computador somente pode executar um programa em linguagem de máquina. Assim, um programa escrito em linguagem de programação (estejam as suas instruções mais próximas da linguagem natural ou não) precisa ser transformado em linguagem de máquina para que possa ser executado por um computador. Essa transformação é realizada por compiladores e interpretadores.

Um programa precisa atender ao formalismo definido pela linguagem de programação. Esse formalismo se refere à sintaxe (a forma de composição instruções e partes de um programa) e à semântica (significado dessas formas) da linguagem de programação. Apresentar esse formalismo utilizando a linguagem C é o objetivo principal deste material.

1 Lógica de Programação

Lógica se refere à técnica ou à forma de ordenar as ações que representam a resolução de um problema. A lógica para desenvolver sistemas computacionais consiste em expressar as ações, instruções, para resolver um problema na sequência e na forma em que elas devem ser executadas pelo computador. Para que essas instruções definam um programa de computador é indispensável que elas sejam definidas de acordo com a sintaxe e a semântica de uma linguagem de programação e a pragmática do problema.

A sintaxe se refere à forma de suas expressões e instruções e à organização de um programa. A sintaxe define símbolos, palavras reservadas, formas de compor as instruções e as estruturas. É o formalismo que determina como o programa é escrito de forma que possa ser compreendido pelo tradutor (compilador, interpretador) da referida linguagem.

Por exemplo, na linguagem C, a sintaxe para a divisão do número dez pelo número três é representada por $10 / 3$. E a divisão de dois números quaisquer é representada por: número / número.

Os erros de sintaxe, em geral, se referem aos erros de escrita em relação ao formalismo definido pela linguagem.

A semântica se refere ao significado das instruções e estruturas que representam a lógica que resolve o problema e são expressas de acordo com a sintaxe definida para a linguagem. Por exemplo, para dividir dois números é necessário utilizar o operador de divisão que é colocado entre dois operandos.

Em $10/3$, 10 é operando, / é o operador de divisão e 3 é operando. E o significado de número / número é a divisão de um número pelo outro.

A pragmática se refere ao contexto, à situação, que a solução do problema se aplica.

Em $10 / 3$ está sendo representada uma divisão inteira. A solução se aplica a esse contexto.

Em programação, a lógica determinada a ordem de execução das ações necessárias para alcançar determinado objetivo, que é, normalmente, colocado como resolver um problema. Para definir quais são essas ações é necessário partir do estudo do problema e chegar à construção de um algoritmo que contenha a maneira de resolução desse problema. Um algoritmo bem elaborado deve ter a solução correta para o problema e que seja genérica em termos de dados de entrada. Isto significa que mesmo que outros dados de entrada sejam informados, o algoritmo permanece funcional, desde que seja mantido o contexto e o escopo considerados na solução do problema. Por exemplo, a solução lógica definida para somar dois números reais deve ser válida e aplicável para somar dois números reais quaisquer.

Para que um computador possa resolver um problema, é preciso explicitar as ações (instruções) na sequência em que elas serão executadas. Por exemplo, para que um computador possa somar dois números é necessário definir um conjunto de instruções que:

1. ***Leia e armazene o primeiro número.***
2. ***Leia e armazene o segundo número.***

É necessário armazenar¹ esses números porque é preciso saber quais são esses

¹ Armazenar um valor significa colocá-lo na memória do computador. A memória pode ser a RAM (Random Access Memory) ou dispositivos de armazenamento de dados, como discos rígidos e dispositivos externos. Uma informação para ser armazenada e acessada na/da memória precisa estar associada a uma variável. Cada “parte” da RAM é denominada endereço de memória

números para poder somá-los. Se esses números não forem armazenados não será possível obtê-los para realizar a soma. Esses números podem ficar em espaços de memória que são identificados por nomes definidos pelo programador e denominados variáveis.

3. *Execute a operação de soma.*

A operação de soma deverá ser expressa linearmente, como uma soma de variáveis. E deve ser definida de acordo com a sintaxe e a semântica definidas pela linguagem de programação utilizada. A solução definida pode restringir-se, ser aplicada a, a adição de números reais.

4. *Mostre o resultado da soma.*

Apresentar o resultado para o usuário.

Para desenvolver um programa de computador ou um sistema computacional é necessário ter domínio do problema que esse programa irá resolver. Somente será possível implementar um algoritmo computacional para resolver um problema se existe uma solução conhecida para o mesmo. E, ainda, é importante que essa solução possa ser executada em tempo finito e com os recursos tecnológicos disponíveis.

Para resolver um problema é necessário analisá-lo, visando identificar a solução, os dados que serão manipulados para obter essa solução, os resultados que serão fornecidos e a forma de verificar se o problema foi resolvido. Essa é a base para transformar um problema em um algoritmo computacional que o resolva.

Analisar um problema significa compreendê-lo e identificar as possíveis soluções; definir os dados necessários para a solução que será adotada, as operações e as ações para obter a solução pretendida para o problema; e definir as saídas, que são os resultados a serem obtidos com o processamento. Os passos ou as ações para implementar a solução de um problema definem um algoritmo que representa a resolução desse problema. Esse algoritmo é implementado em uma linguagem de programação gerando um programa computacional (software).

2 Algoritmos

Um algoritmo define uma sequência de instruções que especificam a resolução de um problema. Algoritmos não estão necessariamente associados aos programas de computador. Eles podem ser definidos para finalidades diversas porque representam as ações ordenadas de maneira a resolver um problema, realizar uma tarefa ou alcançar um objetivo. Contudo, considerando que neste texto o interesse é a programação de computadores, algoritmos são tratados no contexto de programas computacionais. Assim, um algoritmo resolve um problema cuja solução é implementada em uma linguagem de programação e que é executada por um computador.

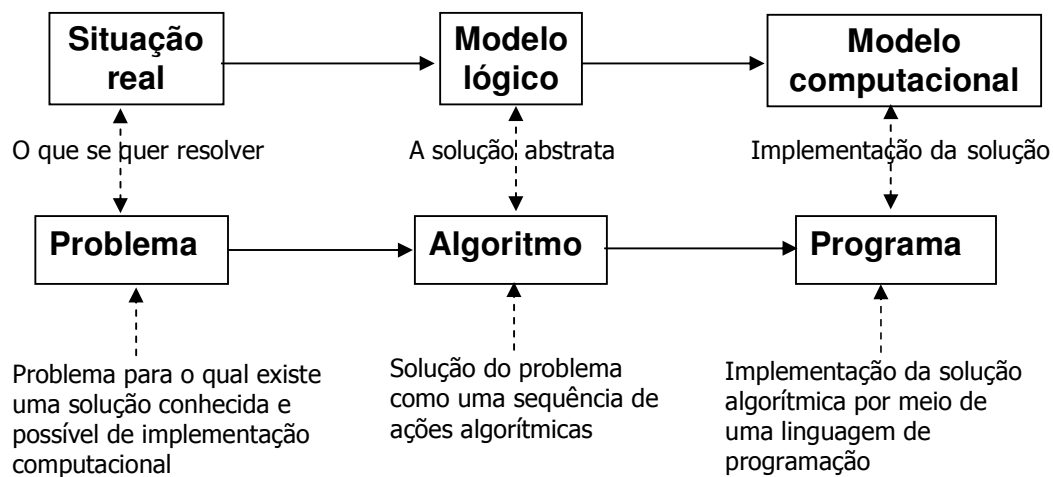
Um algoritmo, em computação, é, muitas vezes, tratado como um programa ou sistema computacional. O algoritmo está relacionado à solução de um problema e esta solução é implementada por meio de uma linguagem de programação. Assim, definir um algoritmo significa definir as ações lógicas para resolver um determinado problema. E implementar um algoritmo se refere à representação dessa solução por meio de uma linguagem de

e variáveis determinam nomes (rótulos) para manipular o conteúdo desses endereços. Ao ser atribuído um valor para uma variável, esse valor fica armazenado em um endereço de memória reservado para essa variável.
--

programação².

A resolução de um problema, cuja solução será computacionalmente implementada, deve ser definida na forma de um algoritmo. Esse algoritmo será codificado em uma linguagem de programação e, assim, transformado em um programa que pode ser executado pelo computador. A Figura 1 apresenta esse processo de maneira esquemática.

Figura 1 – Do problema ao programa

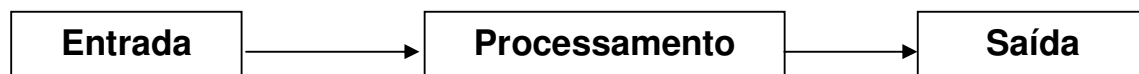


A representação da solução do problema define as ações que devem ser realizadas para resolver esse problema. No caso de programas computacionais, essas ações definem uma solução que possa ser implementada e executada por um computador.

A implementação da solução em uma linguagem de programação transforma a solução algorítmica em um formalismo de acordo com especificações sintáticas e semânticas dessa linguagem de programação. Esse formalismo possibilita que seja gerado o código (em linguagem de máquina) que é executado pelo computador. Assim, definir um programa de computador é, basicamente, expressar as instruções que resolvem um determinado problema da maneira que o computador possa entendê-las, ou seja, executá-las.

Uma forma de elaborar um algoritmo é organizar a solução do problema em três partes, conforme mostra a Figura 2.

Figura 2 – Partes fundamentais de um algoritmo



Como esquematicamente apresentado na Figura 1, um programa de computador é

²Neste texto não há separação rigorosa de significado entre algoritmo e programa. Considerando que o contexto são algoritmos computacionais, algoritmo e programa são tratados indistintamente.

Neste texto, programa, sistema ou aplicativo computacional refere-se à implementação computacional da solução de um problema. Eles são sinônimos de software.

basicamente entrada, processamento e saída.

a) Entrada - são os dados necessários para resolver o problema e que serão utilizados pelo algoritmo. Esses dados são fornecidos pelo usuário do programa, obtidos a partir de outros programas, de dispositivos externos (como sensores), de dispositivos de armazenamento de dados, dentre outros.

b) Processamento – se refere às instruções que são utilizadas para obter os resultados que resolvem o problema. O processamento representa a solução do problema e abrange a manipulação de entradas e a geração de saídas por meio de cálculos, comparações e demais operações e instruções definidos no programa.

c) Saída – a saída representa os resultados obtidos a partir do processamento e que são fornecidos como resultado da execução do programa. Uma saída pode ser composta por informações textuais, valores de constantes e pelo conteúdo de variáveis definidas no programa. São informações apresentadas na tela do computador, em relatórios impressos, interação com dispositivos externos, armazenamento de dados em banco de dados e etc.

Para organizar a solução do problema nessas três partes fundamentais (entrada, processamento e saída) é necessário:

a) Entender o problema, definindo o que o programa deve fazer, o seu objetivo, a abrangência e as limitações.

b) A partir do problema, determinar as entradas e as saídas de dados necessárias para resolver o problema. Definir o tipo de dados de entrada e as variáveis e as constantes necessárias para armazenar esses dados.

c) Determinar as ações a serem realizadas para transformar as entradas nas saídas. Para representar a solução podem ser utilizados esquemas e representações gráficas – definidos de acordo com os conceitos que fundamentam um algoritmo computacional – visando auxiliar no entendimento do problema e principalmente da sua solução.

d) Apresentar os resultados.

e) Verificar se os resultados estão corretos, avaliando se as instruções definidas resolvem o problema da maneira esperada.

Para que um algoritmo, representado de acordo com essas três partes, possa ser executado por um computador ou ser compreendido sem ambiguidade pelas pessoas, é necessário que ele seja definido de acordo com regras e elementos predefinidos. Essas regras e elementos se referem à forma de expressar as ações e os dados que elas manipulam e são definidos por uma linguagem de programação.

2.1 Elementos básicos de um algoritmo

Para representar a solução de um problema por meio de um algoritmo de forma que essa solução possa ser executada por um computador é preciso que esse algoritmo seja elaborado de acordo com as regras sintáticas e semânticas definidas por uma linguagem de programação. Existem determinados elementos que são comuns a diversas linguagens de programação e formas de representar algoritmos. Esses elementos são basicamente: variáveis, instruções, operações, expressões e estruturas de controle.

As variáveis estão relacionadas aos dados que o computador manipula. São os valores informados como entrada para um programa (valor de um produto e percentual de desconto, por exemplo), os dados gerados pelo próprio programa na execução do algoritmo (obtenção do valor com desconto do produto neste exemplo) ou os dados obtidos como saídas do programa (neste exemplo, mostrar ao usuário o valor com desconto). Instruções são as ações que o computador realizará, como escrever uma mensagem na tela. As operações e expressões

estão relacionadas aos cálculos e comparações (testes lógicos) realizados. Estruturas de controle definem a forma como as ações serão realizadas, geralmente vinculadas a testes lógicos.

Ressalta-se que o computador processa (manipula) informações ou dados em formato binário. De maneira simplificada e ilustrativa:

O **sistema binário** é composto por dois valores distintos, normalmente associado com os dígitos **0** e **1**. Por ser composto por dois dígitos distintos é denominado sistema binário ou de base 2. Para que possam ser representados todos os caracteres (letras: a ... z, A .. Z, dígitos: 0 .. 9, alguns símbolos, como: % # , . * / e outros caracteres não imprimíveis) foram definidos agrupamentos de bits, uma combinação de 8 valores binários (0s e 1s). 0 e 1 são os dígitos desse sistema, denominados **bit (binary digit)**.

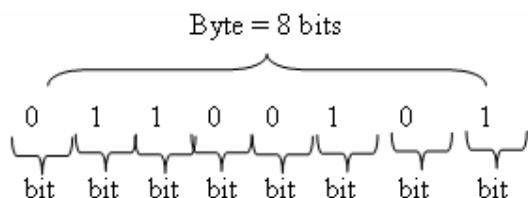
Um bit é a menor unidade de informação que pode assumir um dos dois valores possíveis no sistema binário: 0 ou 1. Um conjunto de 8 bits compõe um **Byte (binary term)**. Byte é a menor unidade manipulável de informação, isso não significa que os bits de um byte não possam ser manipulados.

Com combinações de 8 bits são obtidas 256 configurações diferentes, que representam 256 caracteres (letras maiúsculas e minúsculas, dígitos de 0 a 9 e símbolos como: # \$ %).

Byte é a unidade de medida básica e universal para a capacidade de armazenamento de informação que o computador e os seus dispositivos utilizam. Possui os seguintes múltiplos:

- Kb, Quilobyte (mil bytes) 2^{10} 1.024 bytes (1 KB)
- Mb, Megabyte (milhão de bytes) 2^{20} bytes (1MB)
- Gb, Gigabyte (bilhão de bytes) 2^{30} bytes (1GB)
- Tb, Terabyte (trilhão de bytes) 2^{40} bytes (1TB)
- Pb, Petabyte (quadrilhão de bytes) 2^{50} bytes (1PB)
- Eb, Exabyte (quintilhão de bytes) 2^{60} bytes (1EB)
- Zb, Zettabyte (sextilhão de bytes) 2^{70} bytes (1ZB)
- Yb, Yottabyte (setilhão de bytes) 2^{80} bytes (1YB)

Ilustrativamente a representação de bits compondo um byte:



Os bytes são agrupamentos de bits e, normalmente, a menor porção acessível de memória é o byte, mas esse acesso pode ser feito por meio de um conjunto de bytes (4, por exemplo) formando o que se denomina **palavra** do computador. Assim, em vez de os dados trafegarem entre os dispositivos de entrada e saída, o processador e memória e outros elementos da arquitetura de um computador, byte a byte, eles trafegam em conjunto de bytes, ou seja, por palavra.

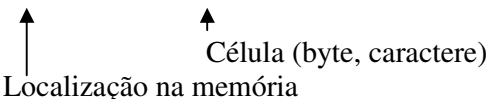
Os dados utilizados por um programa precisam ficar armazenados na memória do computador³ e eles são identificados, localizados, acessados e manipulados por meio de variáveis.

Variáveis denominam endereços de memória nos quais os dados a serem utilizados pelo programa são armazenados. Essa memória também é conhecida como memória principal do computador, memória de acesso aleatório (RAM - Random Access Memory) ou mesmo área de trabalho. A RAM é uma memória volátil porque o seu conteúdo é perdido quando cessa o fornecimento de energia, quando o computador é desligado, por exemplo. Essa memória é utilizada para armazenar instruções e dados de programas que estão em execução. Um programa precisa reservar espaço (endereços de memória) para armazenar os dados a serem manipulados (as variáveis).

A Figura 3 apresenta um esquema ilustrativo de endereços (células, posições) de memória e de conteúdo armazenado. Como o conteúdo do dado a ser armazenado nesse exemplo é caractere (um byte de tamanho), os endereços estão com incremento de um. Considerando que a representação dos endereços é hexadecimal ela é composta por 16 valores de 0 a F (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), então, $E + 1 = F$ e $F + 1 = 0$ e, assim, sucessivamente.

Figura 3 – Representação esquemática de endereços de memória e conteúdo

Endereço	Conteúdo bits	ASCII correspondente	Caractere correspondente
0040280E	01100011	99	C
0040280F	01101000	104	H
00402810	01101001	105	I
00402811	01100011	99	C
00402812	01101111	111	O



Na Figura 3, o endereço representa a localização na memória em que o dado (valor da variável, a informação) está armazenado. Esse armazenamento é feito em bits, portanto o conteúdo do endereço é binário. Esse valor binário possui um valor ASCII⁴ correspondente e também um caractere que é o dado informado pelo usuário. Caracteres são letras, dígitos numéricos, símbolos gráficos e de controle definidos na tabela ASCII.

Duas ações podem ser realizadas com os dados armazenados na memória do computador: leitura e escrita. A escrita ou gravação permite armazenar valores na memória. A leitura permite recuperar esses valores, no sentido de lê-los e não de retirá-los da memória.

A declaração de uma variável em um programa é feita por meio de um nome. É como se fosse colocado um rótulo para o endereço de memória que é reservado para a variável. A

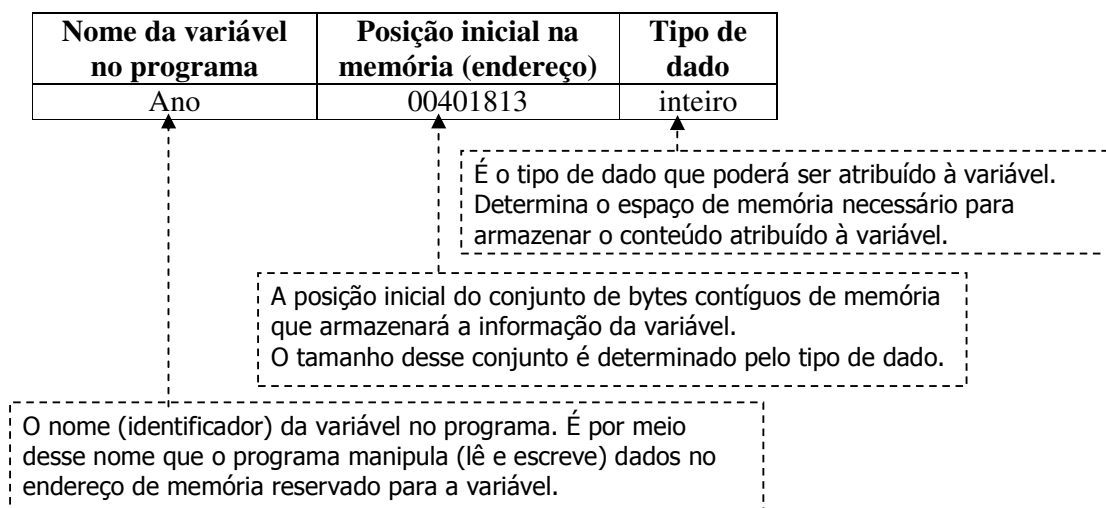
³ Quando um programa está em execução, normalmente, as entradas ficam armazenadas em variáveis na área de trabalho do computador, que é a memória RAM. Contudo, essas variáveis podem ser armazenadas em outro tipo de memória como os discos rígidos.

⁴ ASCII - American Standard Code for Information Interchange que é o código americano para intercâmbio de informações. A codificação define 256 combinações. Nessas combinações então letras maiúsculas e minúsculas, os dígitos de 0 a 9, símbolos como + e *, e outros caracteres especiais como o que indica o final de uma linha de texto.

forma de denominar uma variável, no sentido de os caracteres que são válidos para compor esse nome, depende de cada linguagem de programação. Contudo, normalmente, além do nome é preciso indicar o tipo de dado que será armazenado nesse espaço reservado e referenciado pelo nome da variável. O tipo de dado indica quanto espaço é necessário, ou seja, quantas células de memória subseqüentes serão necessárias para armazenar o referido tipo de dado.

A Figura 4 ilustra graficamente a declaração de uma variável visando exemplificar o relacionamento entre o nome da variável declarado no programa e o endereço de memória em que o dado atribuído à referida variável ficará armazenado. O computador faz referência às células de memória por meio dos seus endereços. E o programa faz referência a esses endereços por meio de nomes definidos às variáveis.

Figura 4 – Esquema de alocação de espaço de memória para variáveis



De acordo com o esquema representado na Figura 4, a posição inicial indica o início do espaço de memória reservado para a variável. A quantidade de bytes (espaço de memória) reservados depende do tipo de dado. Na representação da Figura 4 é declarada uma variável com o nome de “Ano” para armazenar dados do tipo inteiro. O espaço de memória reservado inicia no endereço “00401813” e utilizará tantas células de memória, iniciando nesse endereço, quantas foram necessárias para armazenar um valor inteiro. A quantidade de espaço (endereços memória) necessário para armazenar um valor inteiro depende do tipo de dado e da arquitetura do computador no qual o programa é executado.

De acordo com a representação da Figura 4, uma variável possui um tipo. O tipo determina a quantidade de espaço de memória que a variável precisa e que é reservada para a mesma e o conteúdo que uma variável pode armazenar. Como o tipo determina o tamanho de memória reservado, a variável fica limitada a também armazenar dados do tipo que ela foi declarada. Além disso, o tipo de uma variável é utilizado para definir compatibilidade e consistência de dados manipulados pelo programa.

Para obter os dados de entrada como os informados pelo usuário e que serão armazenados nas variáveis é preciso ter instruções de entrada definidas no programa. Essas instruções são encarregadas de obter os dados do dispositivo de entrada - como teclado, mouse, portas nas quais estão conectados equipamentos como sensores, banco de dados - e de armazená-los nas respectivas variáveis.

Um programa de computador, basicamente, manipula dados. Para manipular esses dados são realizadas instruções. Uma instrução é uma ação realizada por um programa de

computador, como ler um dado informado a partir do teclado e que será armazenado em uma variável, realizar um cálculo com esse dado e mostrar na tela do computador o conteúdo de uma variável.

Os dados armazenados em variáveis são utilizados para realizar as instruções de um programa, sejam operações matemáticas, lógicas ou relacionais, de decisão ou de repetição.

As operações matemáticas se referem às fórmulas e aos cálculos matemáticos. Para que fórmulas e cálculos sejam realizados eles precisam ser transformados em uma expressão. Por exemplo, para efetuar a seguinte fórmula:

$$\frac{12}{2} + \frac{2-5}{3}$$

É necessário transformá-la em uma equação ou expressão na forma:

$$12/2 + (2-5) / 3$$

que utilize os operadores matemáticos definidos pela linguagem, como o símbolo / para divisão e agrupamentos por parênteses em vez de um traço horizontal.

Além de realizar cálculos, para definir uma solução computacional para um problema pode ser necessário decidir se determinadas instruções serão ou não realizadas ou se serão repetidas um determinado número de vezes.

As estruturas de decisão permitem definir se um determinado conjunto de instruções será ou não realizado. Por exemplo, realizar uma divisão somente se o divisor é diferente de zero.

As estruturas de repetição permitem repetir um conjunto de instruções. Quantas vezes essas instruções serão repetidas ou as condições para que a repetição seja finalizada pode ser informada pelo usuário do programa, ser predefinida (ter sido definida de maneira fixa no programa), ser decorrente de resultados de processamento interno, dentre outros.

Por exemplo:

Ler o salário de dez funcionários. Está predefinido que serão dez leituras.

Calcular os impostos de mercadorias enquanto informado valor maior que zero para o valor da mercadoria. A condição de finalização é gerada durante a execução do programa, quando informado zero ou valor negativo para o valor da mercadoria.

Resumindo, os elementos básicos de um algoritmo se referem a:

- a) Manipular dados (ler e escrever na memória) e para isso é necessário ter variáveis que permitem ter acesso a esses dados. Lembrando que os dados podem ficar armazenados não somente na memória RAM, mas em dispositivos de armazenamento como discos rígidos.
- b) Realizar cálculos e operações matemáticas.
- c) Realizar testes lógicos.
- d) Verificar se determinado conjunto de instruções será ou não realizado.
- e) Realizar um determinado conjunto de instruções um determinado número de vezes (repeti-las).

As ações realizadas por um programa, representadas por meio de um algoritmo, visando resolver um problema, alcançar um objetivo, são denominadas instruções.

2.2 Formas de representação de algoritmos

Um algoritmo pode ser representado de diversas maneiras, incluindo descrição textual, representação gráfica e linguagem de programação. O critério usado para classificar hierarquicamente essas formas pode estar relacionado à proximidade com a linguagem natural

ou de máquina. Um algoritmo pode ser considerado mais abstrato, à medida que a forma utilizada para representá-lo se aproxima de uma linguagem computacional e, assim, mais se distancia da linguagem natural, a falada pelos humanos.

Dentre as formas de representação de algoritmos estão:

- 1) Descrição narrativa - faz uso de uma linguagem natural (falada pelos humanos), como o português, para descrever algoritmos.
- 2) Fluxograma ou diagrama de blocos – representação gráfica de instruções lógicas de determinado processamento. A ênfase está na representação da sequência em que as instruções serão realizadas.
- 3) Pseudocódigo ou português estruturado – utiliza palavras-chave pré-definidas para expressar a organização e as instruções de um programa.
- 4) Linguagem de programação – possui regras e elementos definidos que correspondem à sintaxe e à semântica que caracteriza uma linguagem de programação e permite realizar instruções que podem ser compiladas ou interpretadas e posteriormente executadas pelo computador.

Nesse texto, essas formas de representação de algoritmos serão utilizadas à medida que consideradas necessárias para facilitar o entendimento dos exemplos, sendo que a ênfase está no uso da linguagem de programação C para representar os algoritmos computacionais. Essas formas estão apresentadas nas subseções a seguir.

Transformar a solução de um problema em um algoritmo computacional pode ser uma tarefa bastante complexa. A complexidade do problema ou mesmo das instruções computacionais necessárias pode dificultar a transição direta da solução para instruções em linguagem de programação. Uma maneira de lidar com essa complexidade é definir a solução em etapas e por níveis de abstração distintos. Assim, inicia-se descrevendo a solução do problema de maneira mais próxima da linguagem humana e gradativamente passasse para o algoritmo computacional. Além disso, é importante dividir o problema e a respectiva solução em partes, definindo e implementado cada parte e agregando-as à medida que são implementadas e testadas.

Essa abordagem parte de uma representação mais abstrata, de alto nível, da solução do problema e por refinamentos sucessivos chega à sua implementação em uma linguagem de programação. Nos exemplos deste texto esse procedimento é utilizado. O nível mais alto de abstração é a identificação dos dados de entrada, das instruções básicas de processamento e dos dados de saída, de acordo com o procedimento para analisar um problema. Na sequência, a solução é representada graficamente. Dessa representação gráfica é definido um algoritmo em linguagem de programação C.

2.2.1 Descrição narrativa

A representação de algoritmos em descrição narrativa ou linguagem natural consiste no uso de frases para expressar as ações a serem realizadas.

Exemplo de algoritmo em descrição narrativa para fazer a média de quatro números:

- 1) Obter quatro números;
- 2) Somar os quatro números;
- 3) Dividir o resultado da soma dos quatro números por quatro;
- 4) Apresentar o resultado.

A principal vantagem da descrição narrativa é o uso de uma linguagem comumente utilizada, como, por exemplo, o português. Dentre as desvantagens, destacam-se a imprecisão e a extensão que o algoritmo pode alcançar. A imprecisão ocorre porque as instruções podem nem sempre ser claras ou facilmente compreendidas, podendo gerar ambiguidades. A

extensão ocorre, porque dependendo do problema é necessário muito texto para expressar as ações para resolver o problema.

A seguir está um procedimento sugerido com o objetivo de auxiliar na análise de um problema e na definição de uma solução algorítmica para o mesmo. Esse procedimento tem como base a descrição narrativa, mas visa identificar os dados de entrada, o processamento e a saída necessários para resolver um problema.

1) **Descrição do problema** - o enunciado do problema ou o quê o programa deverá fazer.

2) **Análise do problema** - a identificação da solução do problema, considerando os elementos básicos de um algoritmo.

2.1) **Identificação de dados de entrada** - os dados que o algoritmo necessita para obter a solução pretendida para o problema.

2.2) **Processamento** - as instruções, os cálculos e as comparações que precisam ser realizadas para resolver o problema.

2.3) **Definição das saídas** - as informações a serem apresentadas incluindo as obtidas a partir das instruções realizadas.

3) **Representação do algoritmo** - a representação pode ser gráfica, textual ou em uma linguagem de programação.

2.2.2 Português Estruturado

Português estruturado ou pseudocódigo é uma forma de representar algoritmos que, de certa maneira, se assemelham à representação em linguagens de programação porque utiliza palavras chaves e propõe uma estrutura à solução. São utilizados comandos para leitura e escrita, as variáveis possuem tipos e são definidas formas para sua declaração e atribuição, assim como para as estruturas de controle.

Se um algoritmo em português estruturado for bem definido a sua tradução para uma determinada linguagem de programação é relativamente direta. É claro que será necessário representar as instruções e as estruturas de acordo com a sintaxe e a semântica da linguagem, mas a ideia para a solução do problema permanece. Por isso, os exemplos utilizados neste texto, em geral não são representados em português estruturado, mas sim diretamente na linguagem de programação C.

Como vantagens do uso do português estruturado para representar algoritmos podem ser citadas:

a) A independência da representação e da implementação da solução. É definida apenas a solução lógica desvinculada de uma linguagem de programação específica;

b) O uso de uma linguagem conhecida, no caso, o português, como base, facilitando o entendimento;

c) A transposição relativamente facilitada da solução representada em português estruturado para linguagens de programação.

Dentre as desvantagens do uso do português estruturado para representar algoritmos estão:

a) Falta de padronização na representação do algoritmo, incluindo palavras-chave, forma das instruções e das estruturas;

b) A necessidade de entendimento de uma linguagem que não é utilizada para codificar sistemas;

c) A representação em português estruturado é facilmente traduzida para uma linguagem de programação, mas o programa para ser executado deverá ser escrito de acordo

com a sintaxe da linguagem e assim será indispensável reescrever o código atendendo ao formalismo da linguagem. Ressalta-se que existem interpretadores ou compiladores de algoritmo em português estruturado.

A forma geral da representação de um algoritmo em português estruturado é:

Nome do programa

Início

Declaração das variáveis

Conjunto de instruções

Fim

onde:

Nome do programa é um nome simbólico definido para o programa com a finalidade de identificá-lo ou distingui-lo dos demais.

Declaração de variáveis contém as variáveis usadas no algoritmo com o respectivo tipo de dado que elas podem conter.

Conjunto de instruções são os comandos ou instruções a serem executadas.

Início e Fim são, respectivamente, as palavras que delimitam o início e o término do conjunto de instruções que compõem o algoritmo.

Exemplo de algoritmo para calcular o dobro de um número:

Algoritmo Calcula_Dobro

Início

Num: inteiro

Dobro: inteiro

Leia Num

Dobro \leftarrow Num * 2

Escreva Dobro

Fim

Nesse algoritmo, são declaradas duas variáveis Num e Dobro, ou seja, são reservados espaços de memória distintos para armazenar dois valores inteiros. Num armazenará o valor informado pelo usuário. Isso é realizado pela instrução Leia Num, que obtém o valor informado e o armazena no endereço de memória reservado para a variável Num. Dobro armazenará o resultado do cálculo do dobro de Num. O processamento é realizado pela instrução que calcula o dobro de Num ($\text{Num} * 2$) e o armazena na variável Dobro. Essa instrução é representada por: $\text{Dobro} \leftarrow \text{Num} * 2$. A seta para a esquerda indica atribuição⁵. Nessa instrução, o resultado do cálculo da expressão matemática será atribuído para a variável Dobro. O resultado será apresentado para o usuário por meio da instrução Escreva Dobro.

2.2.3 Fluxograma

Um fluxograma representa a sequência em que as ações de um algoritmo devem ser

⁵Em geral, pode-se dizer que a atribuição (\leftarrow) sempre é realizada para uma variável, isto é, no lado esquerdo sempre deverá estar uma variável. No lado direito do sinal de atribuição pode estar uma constante, uma variável, uma expressão lógica ou matemática ou o retorno de uma função.



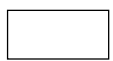
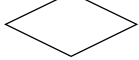


executadas. Essa sequência de ações é representada na forma de elementos gráficos relacionados. As interconexões entre as ações, que determinam a ordem de execução dessas ações, são representadas por meio de setas.

Dentre as vantagens do uso de fluxograma para representar algoritmos computacionais está a que eles são utilizados em diversas áreas do conhecimento. Outra vantagem é o uso de elementos gráficos que facilitam a visualização e o entendimento do que está sendo representado. Em relação às desvantagens do seu uso está o aumento de complexidade da representação gráfica à medida que o algoritmo se torna mais extenso ou complexo.

Para representar um algoritmo em fluxograma existem símbolos definidos para indicar o início e o fim do algoritmo, as entrada e as saída de dados e realização de cálculos, dentre outros. A representação das instruções contidas nos símbolos do fluxograma pode ser feita por meio de português estruturado.

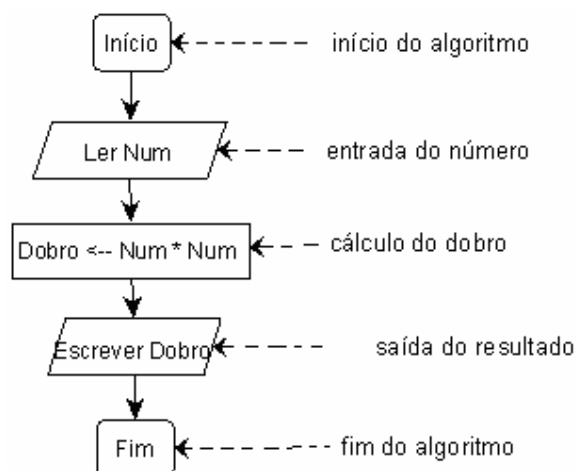
A Figura 3 contém os símbolos gráficos utilizados neste texto para representar algoritmos por meio de fluxogramas.

Figura 3 – Símbolos gráficos para representar algoritmos por meio de fluxograma

Símbolo	Descrição
	Início e fim do algoritmo.
	Entrada e saída. Receber e mostrar informações. Ler dados para armazenar em variáveis. Mostrar dados contidos em variáveis, resultados de processamento e escrever texto como saída do algoritmo.
	Realizar operações com variáveis e constantes. Executar as instruções contidas em estruturas de decisão e de repetição.
	Representa um teste lógico realizado. Possibilita sequências alternativas de instruções de acordo com o resultado desse teste.
	Utilizado para conectar fluxos como, por exemplo, saídas distintas de um teste lógico.
	Conector direcionado para conectar símbolos, indicando a sequência de execução das ações.

A Figura 4 apresenta um exemplo de fluxograma para calcular o dobro de um número. Nessa figura, as setas pontilhadas e os textos a sua direita não fazem parte da representação do algoritmo em fluxograma. Eles estão colocados como comentários, visando facilitar o entendimento das representações.

Figura 4 – Fluxograma para o cálculo do dobro de um número



O fluxograma representado na Figura 4 tem como entrada um número que é lido (possivelmente informado pelo usuário) e armazenado na variável Num. Em seguida, é calculado o dobro desse número. A variável Dobro recebe o resultado da multiplicação do conteúdo da variável Num pelo conteúdo dessa mesma variável. Em seguida é apresentado o resultado desse cálculo, que está armazenado na variável Dobro, com a instrução Escrever Dobro.

2.2.4 Linguagens de programação

Uma linguagem de programação determina a maneira de representar as instruções que compõem um algoritmo para que elas possam ser executadas por um computador. Para os objetivos deste trabalho, define-se linguagem de programação como um método que fornece a sintaxe e a semântica para representar algoritmos que representam a solução de problemas que possuem implementação computacional.

A forma de representar essas instruções pode estar mais próxima da linguagem falada pelos humanos ou mais próxima da linguagem de máquina. Uma linguagem de máquina, ou a que é diretamente executada por um computador, é bastante distinta da linguagem falada pelos humanos.

A linguagem falada pelos humanos pode ser bastante ambígua. A ambiguidade se refere à possibilidade de uma mesma frase ser interpretada distintamente, ter significado diferente, para pessoas distintas. Isso ocorre porque a frase pode não ser expressa com clareza, o seu conteúdo está inserido em um contexto ou porque cada pessoa possui conhecimentos que interferem na interpretação da mensagem recebida. O contexto e o conhecimento auxiliam a determinar significados e interpretações.

Apesar da possibilidade de ambiguidade, na linguagem humana uma frase pode ser compreendida mesmo que ela não esteja completa ou adequadamente falada ou escrita. Por exemplo, se faltar pronome, artigo ou palavras em uma frase, o seu significado pode ser compreendido, ainda que de maneira não completa.

Para um computador, as instruções representadas por meio das linguagens de programação devem ser precisas, não ambíguas, e completas na representação de cada ação que se pretende que seja realizada.

Ressalta-se, porém, que existem linguagens de programação, como, por exemplo, as voltadas para aplicações em Inteligência Artificial que possibilitam implementar certos tipos de ambiguidades e inconsistências. Essas linguagens podem utilizar lógicas não binárias. Contudo, o tratamento das ambiguidades e inconsistências é realizado pelas instruções que

compõem o programa. Mesmo nesse tipo de programa, a máquina, o computador, continua interpretando exclusivamente instruções binárias.

Como a linguagem humana e a compreendida pelo computador são bastante distintas em termos de necessidade de formalismo e de possibilidades de interpretação, as linguagens de programação estão em um nível intermediário entre a linguagem falada pelos humanos e a de máquina. Existem linguagens de programação que possuem as suas instruções expressas de maneira mais próximas à linguagem de máquina e outras que as suas instruções são representadas de maneira mais próxima da linguagem humana.

As características das linguagens de programação as tornam mais adequadas para determinadas aplicações. Essas características são recursos e funcionalidades, como, por exemplo, fornecer componentes para desenho de interface gráfica, possuir funções matemáticas avançadas pré-definidas e permitir a interação com linguagem de marcação de hipertexto. Como as linguagens de programação possuem características distintas, não são todas as linguagens que se aplicam de maneira otimizada para resolver os mesmos problemas. Por exemplo: um jogo eletrônico, um sistema de apoio à tomada de decisões, um sistema de auxílio a diagnóstico médico, uma web com notícias ou um sistema de controle de acesso a um depósito de matéria prima em uma usina atômica requerem recursos distintos para serem implementados. Esses são exemplos de sistemas que possuem características bem distintas e seria difícil para uma mesma linguagem de programação prover recursos para implementar todos esses tipos de sistema de forma facilitada. É compreensível que determinadas linguagens possuam recursos mais adequados para implementar determinados tipos de sistemas.

2.2.5 Método para definir um programa

Para definir a solução algorítmica de um problema transformando-o em um código executável pelo computador e, que, portanto, esteja de acordo com a sintaxe e a semântica de uma linguagem de programação, sugere-se o seguinte método:

- a) Entender o problema;
- b) Retirar do problema as entradas de dados necessárias;
- c) Definir as saídas necessárias;
- d) Determinar o que deve ser feito para transformar as entradas nas saídas (o algoritmo de solução do problema):
 - d.1) Determinar o tipo de dado a ser manipulado, definindo as variáveis e as constantes necessárias;
 - d.2) Definir as instruções e as estruturas de decisão e de repetição necessárias;
- e) Apresentar os resultados;
- f) Verificar se as instruções definidas resolvem o problema da maneira esperada;
 - f.1) Testar o resultado obtido.

Esse método é proposto com o objetivo de auxiliar a entender o problema, definir a solução que será representada de acordo com uma linguagem de programação e verificar o resultado. O teste de mesa é muitas vezes realizado de forma manual, determinando as saídas válidas para entradas específicas. Assim, a partir dessas entradas o programa definido deve gerar as mesmas saídas que foram obtidas pelo teste manual. A quantidade e o tipo dos testes a serem realizados estão relacionados à complexidade e ao tipo do programa.

3 Elementos básicos de um algoritmo em linguagem de programação C

A linguagem C é uma linguagem de programação estruturada⁶ e compilada. Nesta seção, a linguagem C é utilizada visando representar conceitos básicos e fundamentais de algoritmos, ou seja, os elementos que os compõem. Esses elementos são:

- a) Tipos de dados;
- b) Variáveis e constantes;
- b) Operadores aritméticos, relacionais e lógicos;
- c) Instruções de entrada e de saída;
- d) Estrutura sequencial;
- e) Estruturas de controle decisão;
- f) Estruturas de controle repetição;
- g) Estruturas de dados homogêneos;
- h) Estruturas de dados heterogêneos;
- i) Funções.

Ressalta-se que esses não são os únicos elementos de uma linguagem de programação, nem mesmo da linguagem C, e também que esses elementos não são tratados exatamente da mesma maneira por todas as linguagens de programação. Esses elementos são os considerados básicos para o paradigma de programação estruturada no contexto de aprendizado de fundamentos de programação por meio da linguagem C, que é o interesse deste material.

Para auxiliar no entendimento da linguagem C, os exemplos citados neste texto, quando considerado necessário, são complementados com outras formas de representação de algoritmos, como descrição narrativa, português estruturado e fluxograma.

3.1 Conceitos iniciais

Um programa de computador é um conjunto instruções que representam o algoritmo para a resolução de um problema. Essas instruções são escritas por meio um conjunto de símbolos e palavras e de acordo com regras de estruturação semântica e sintática específicas. Esse conjunto de símbolos e de regras compõe uma linguagem de programação. E é com eles que é produzido o código, denominado fonte, de um programa.

A seguir um exemplo de programa na linguagem C, visando explicitar alguns aspectos desse formalismo. Essa é a estrutura básica para um programa executável na linguagem C.

```
#include <stdio.h>
```

```
/* exemplo básico de programa em linguagem de programação C*/
```

```
int main (void)
{
    printf ("Mostrando informações para o usuário.");
}
```

⁶ O paradigma de programação estruturada tem como base que as instruções que compõem um programa são estruturas sequenciais, de decisão ou de repetição (iteração). Porém, essas instruções podem ser implementadas em separado, ou seja, o programa pode ser modularizado. Um módulo ou função pode realizar uma funcionalidade bem específica ou conter um agrupamento de funcionalidades. Modularizar um programa tem o objetivo de facilitar a implementação da solução (divisão do problema e da solução em partes) e prover o reuso de código no mesmo programa e em programas distintos.

Esse programa ao ser executado mostrará a seguinte mensagem na tela:

Mostrando informações para o usuário.

Analisando as partes desse programa:

a) `#include <stdio.h>` - `#include <stdio.h>` indica para o compilador que deve ser incluído o arquivo de cabeçalho `stdio.h` no programa. E, assim, as instruções contidas nesse arquivo poderão ser utilizadas pelo programa. Um arquivo de cabeçalho é também conhecido como biblioteca. Esses arquivos contêm partes de programa já prontas. `stdio.h` contém funções para entrada e saída de dados (`std` (standard) = padrão; `io` (input/output) = entrada/saída. `stdio` = entrada e saída padrão). As funções de entrada e saída são utilizadas para obter dados de entrada para um programa e apresentar saídas, que podem ser informações para o usuário.

Entrada se refere a obter informações fornecidas em tempo de execução para o programa, como, por exemplo, um número para ser calculada a sua raiz quadrada. Esse número será armazenado em uma variável. O dispositivo padrão de entrada é o teclado.

Saída se refere a mostrar informações, sejam texto, conteúdo variáveis ou de constantes, acionar dispositivos, armazenar dados em banco de dados ou acessar outros aplicativos computacionais, dentre outros. O dispositivo padrão de saída é o monitor de vídeo.

De certa forma, a inclusão de bibliotecas é quase que indispensável em um programa C. Isso ocorre porque a linguagem C não possui comandos ou funções definidos internamente, ela possui apenas palavras chaves. Assim, em um programa C todas as instruções precisam ser definidas. E incluir bibliotecas permite usar instruções que já estão definidas. A inclusão de biblioteca é uma forma de reusar código.

As bibliotecas incluídas podem ter origens distintas, como ser:

- Implementadas por programadores, seja para uso comercial, particular, com fins didáticos ou não lucrativos. Todo programador pode criar as suas próprias bibliotecas.
- Disponibilizadas com o próprio compilador da linguagem.
- Obtidas por compra ou mesmo gratuitamente.

b) `/*` exemplo básico de programa em C `*/` - é um comentário. O compilador C desconsidera todo o conteúdo que estiver entre os símbolos de `/*` e `*/`. Esses comentários somente são visualizados quando da edição do código fonte do programa. Eles auxiliam no entendimento do código e facilitam a manutenção do programa.

c) `int main(void)`⁷ - `int main(void)` indica que está sendo definida uma função com o nome `main`. Um nome seguido de parênteses indica uma função na linguagem C. Todo programa executável em C devem ter uma e somente uma função `main`. É por essa função que é iniciada a execução do programa. Os arquivos de cabeçalho, que também são programas C, não possuem função `main` porque eles não são executáveis. O conteúdo de uma função é delimitado por chaves `{ }`. Chaves são utilizadas para delimitar blocos de código, seja de funções ou de estruturas de controle. Uma função delimita um conjunto de instruções (código) com objetivos ou finalidades específicas. As bibliotecas são compostas por conjuntos de funções.

d) `printf()` – `printf()` tem o objetivo de escrever informações na saída padrão que é a

⁷ Não é para todos os compiladores C que a função `main` é do tipo `int` (inteiro), ela pode ser do tipo `void`. O tipo `void` significa que a função não possui retorno. Ou seja, ela não retorna valor para a função que a chama ou para o sistema operacional. O tipo `int` da função `main()` significa que ela retorna um valor inteiro para o sistema operacional após finalizada a execução do programa.

tela do computador. Por padrão a função `printf()` está definida no arquivo `stdio.h`, ou seja, o código dessa função está nesse arquivo. Assim, para que `printf()` possa ser usada o arquivo `stdio.h` (ou outro que a contenha) deve ser incluído. Para essa função é passado um texto que é colocado entre aspas. Informação do tipo texto é conhecida como string que é uma sequência de caracteres, sejam letras, números ou símbolos. O que é passado para uma função é denominado parâmetro ou argumento. Uma função pode receber vários parâmetros e de tipos diferentes. A função `printf()` do exemplo recebe a string "Mostrando informações para o usuário." como parâmetro. A função `printf()`, no exemplo, irá apenas colocar a string que está entre aspas na tela do computador.

e) ; - o ponto e vírgula indica o final de uma instrução. Toda instrução na linguagem C é finalizada com ponto e vírgula (;). Uma instrução é uma ação que será realizada no contexto do programa implementado.

Outro exemplo de programa em linguagem C:

```
#include <stdio.h>

int main (void) /*função principal*/
{
    /* Declaração de variáveis */
    int Dias;
    float Anos;
    /* Entrada de Dados */
    printf ("Informe um valor que representa a quantidade de dias: ");
    scanf ("%d",&Dias);
    /*Processamento - conversão de dias em anos */
    Anos=Dias/365.25;
    /* Saída de Dados */
    printf ("\n%d dias equivalem a %f anos.\n",Dias,Anos);
}
```

Analisando as partes do programa:

a) Declaração de variáveis - são declaradas duas variáveis chamadas Dias e Anos. A primeira é do tipo inteiro (int) e a segunda é do tipo real ou ponto flutuante (float). Variáveis declaradas como ponto flutuante podem armazenar números com casas decimais, como, por exemplo, 5.1497. Na linguagem C, o separador decimal é o ponto. Para a variável Dias é reservado um espaço de memória suficiente para armazenar um valor inteiro. Para a variável Anos é reservado um espaço de memória para armazenar um valor real. Os inteiros compõem um subconjunto dos reais. Portanto, uma variável declarada como float pode armazenar um inteiro, o contrário não é possível e quando é pode haver a perda de dado ou de precisão.

b) `printf()` - após a declaração das variáveis é feita uma chamada à função `printf()`, que coloca uma mensagem na tela para informar ao usuário que ele precisa fornecer uma informação. O valor informado pelo usuário será armazenado na variável Dias.

c) `scanf()` - um valor fornecido pelo usuário é lido e armazenado na variável inteira Dias. Para isso é utilizada a função `scanf()`. A string "%d" diz à função `scanf()` que será lido um valor do tipo inteiro. `&Dias` significa que o valor lido deverá ser armazenado na variável Dias. O `&` indica que o valor lido será armazenado em um endereço de memória reservado para essa variável e que o valor armazenado nesse endereço é acessado por meio do seu

indicador ou nome Dias.

A função `scanf()` possui dois parâmetros: o símbolo de `%d` e `&Dias`. O símbolo `%d` indica o tipo de valor que será lido e `&` a variável que armazenará cada uma das respectivas entradas, ou mais especificamente, que o valor lido será armazenado no endereço de memória reservado para a respectiva variável. Na linguagem C, quando uma função possui dois ou mais parâmetros, eles são separados por vírgula.

d) Expressão matemática $Anos = Dias / 365.25$; – a conversão de dias em anos é realizada por meio de uma expressão matemática que atribui à variável `Anos` o conteúdo da variável `Dias` dividido por 365.25. O uso de parênteses pode ser indispensável para que uma expressão matemática fique correta ou podem ser colocados apenas para facilitar a sua interpretação.

e) Parâmetros de `printf()` - a segunda chamada à função `printf()` possui três argumentos. A string `"\n%d dias equivalem a %f anos.\n"` significa que deve ser deixada uma linha em branco (`\n`), ser mostrado o conteúdo de uma variável do tipo inteiro na tela (`%d`) que é o conteúdo da variável `Dias`, ser mostrada a frase "dias equivalem a", ser mostrado um valor float (`%f`) que é o conteúdo da variável `Anos`, ser mostrada a palavra "anos." e em seguida o cursor ser colocado no início da próxima linha (`\n`). Os outros dois parâmetros são as variáveis `Dias` e `Anos`. Essas variáveis são do tipo `int` e `float`, respectivamente, e o seu conteúdo substituirá os caracteres de formatação `%d` que indica que será mostrado um inteiro e `%f` que indica que será apresentado um valor real. O `\n` é uma constante interpretada como uma instrução ou caractere de formatação de mudança de linha. Após imprimir o conteúdo da instrução `printf()`, o cursor passará para o início da próxima linha.

Esses exemplos auxiliaram a identificar alguns elementos e conceitos da linguagem C, como função, a função `main`, parâmetros de função, declaração, atribuição, uso e tipo variáveis, funções de entrada e saída e realização de expressões matemáticas. Eles, juntamente com os demais elementos básicos de um programa, estão apresentados nas subseções a seguir.

A linguagem C é case sensitive, isto é, maiúsculas e minúsculas são tratadas de maneira distinta. Se uma variável é declarada como `dias` ela deverá ser usada dessa forma e `Dias`, `DIAS`, `DiAs` ou `DiAs`, por exemplo, são variáveis distintas. Da mesma maneira, palavras reservadas da linguagem C, como `if` e `for`, só podem ser escritos em minúsculas. É dessa forma que o compilador os conhece porque é assim que elas foram definidas. Se escritos de outra maneira eles não serão interpretados como palavras reservadas da linguagem e sim definições do programador. Ainda, palavras reservadas não podem ser utilizadas como nomes de identificadores, como variáveis. As funções que estão nos arquivos `#include` também devem ser escritas da mesma forma que foram definidas nesses arquivos.

3.1.1 As palavras reservadas da linguagem C

As linguagens de programação possuem palavras reservadas, ou palavras chave, com significados específicos e identificados pelos compiladores e/ou interpretadores de cada linguagem. A padronização ANSI da linguagem C define 32 palavras reservadas. Essas palavras são reconhecidas pelos compiladores C e a elas estão associadas a funcionalidades e ações a serem realizadas pelos programas. As palavras reservadas não podem ser usadas como nomes de identificadores, sejam variáveis, constantes ou funções. Como a linguagem C é case sensitive é possível declarar uma variável `For`, apesar de haver uma palavra reservada `for`. A

Figura 5 apresenta as palavras reservadas do C padrão ANSI.

Figura 5 - As palavras reservadas do C ANSI

<i>auto</i>	<i>break</i>	<i>case</i>	<i>char</i>	<i>const</i>	<i>continue</i>
<i>default</i>	<i>do</i>	<i>double</i>	<i>else</i>	<i>enum</i>	<i>extern</i>
<i>float</i>	<i>for</i>	<i>goto</i>	<i>if</i>	<i>int</i>	<i>long</i>
<i>register</i>	<i>return</i>	<i>short</i>	<i>signed</i>	<i>sizeof</i>	<i>static</i>
<i>struct</i>	<i>switch</i>	<i>typedef</i>	<i>union</i>	<i>unsigned</i>	<i>void</i>
<i>volatile</i>	<i>while</i>				

3.1.2 Comentários

O uso de comentários torna o código de um programa mais fácil de entender. O conteúdo dos comentários é desconsiderado no processo de compilação ou de interpretação e na execução do programa. Comentários na linguagem C podem ser de dois tipos:

a) // - duas barras indicam que é comentário o texto a partir das barras até o final da respectiva linha. O conteúdo, a partir de //, é desconsiderado pelo compilador. Se as barras estiverem no início da linha, a linha inteira é desconsiderada. Por exemplo:

```
printf ("linguagem C"); //imprime o texto entre as aspas.
```

Nesse caso, o texto linguagem C é mostrado na tela do computador.

```
//printf ("linguagem C"); imprime o texto entre as aspas.
```

Nesse caso, o conteúdo da função printf() não é mostrado na tela do computador porque a função está comentada.

b) /* */ - o que está entre os símbolos de /* e */ é comentário e, portanto, desconsiderado pelo compilador. Esses símbolos podem abranger desde um único caractere até *n* linhas. Contudo não podem ser utilizados no meio de uma palavra reservada ou de um identificador.

```
printf ("linguagem C"); /* imprime na tela do computador o texto que está entre as aspas. */
```

Nesse exemplo, o comentário ocupa mais de uma linha.

```
printf (/*parâmetros da função*/"linguagem C"); /*o texto em comentário indica o parâmetro da função*/
```

3.1.3 Instruções ou comandos

Uma instrução, também referida como comando, determina uma ação a ser realizada. As instruções devem ser escritas de acordo com a sintaxe e a semântica determinadas pela linguagem de programação.

Na linguagem C toda instrução termina com ; (ponto e vírgula). Não é necessário colocar uma instrução por linha, porque o final de uma instrução é indicado pelo ponto e vírgula. Contudo, colocar uma instrução por linha é uma forma de manter o código mais legível e de entendimento mais fácil.

3.1.4 Funções

Uma função é um conjunto de instruções com um objetivo específico. O cálculo da raiz quadrada de um número, a verificação se um número é par, o cálculo de juros bancários a

partir de determinada fórmula são exemplos de funções.

A sintaxe básica de uma função é:

Tipo_de_retorno Nome (parâmetros)

Tipo define o tipo de retorno da função, do resultado do seu processamento. Uma função pode retornar valores. Como, o valor da raiz quadrada do número que é passado para uma função que tem o objetivo de calcular a raiz quadrada.

Nome é o identificador da função. Esse nome é utilizado quando a função é definida e quando ela é usada.

Parâmetros são os valores que são passados para uma função. Por exemplo, um número do qual será calculada a raiz quadrada.

Exemplo:

`float Raiz (float Numero);`

A função `float Raiz (float Numero)` é utilizada para calcular a raiz quadrada de um número. Essa função pode ser utilizada da seguinte maneira:

`Resultado = Raiz (12);`

O valor retornado é do tipo `float`. Portanto, a variável `Resultado` deverá ser do tipo `float`. O nome `Raiz` é utilizado quando a função é declarada, definida, e quando ela é usada. A função espera um parâmetro `float` e é passada a constante `12`, isso é possível porque `12` que é inteiro (`int`) é um subconjunto dos reais (`float`).

A programação em linguagem C é baseada em funções. Por mais simples ou de poucas linhas que seja um programa executável em C ele terá pelo menos uma função, a `main()`. A função `main()` é obrigatória em um programa C executável. É por essa função que é iniciada a execução do programa. Se existirem outras funções, definidas no mesmo arquivo ou incluídas como arquivo de cabeçalho, elas serão chamadas a partir da função `main()`.

Um programa C que é uma biblioteca, um arquivo de cabeçalho, não possui uma função `main()`.

Um programa C executável pode ser composto por várias funções, definidas no próprio programa ou incluídas a partir de bibliotecas, mas terá, uma e somente uma função `main()`.

3.1.5 O processo de compilação

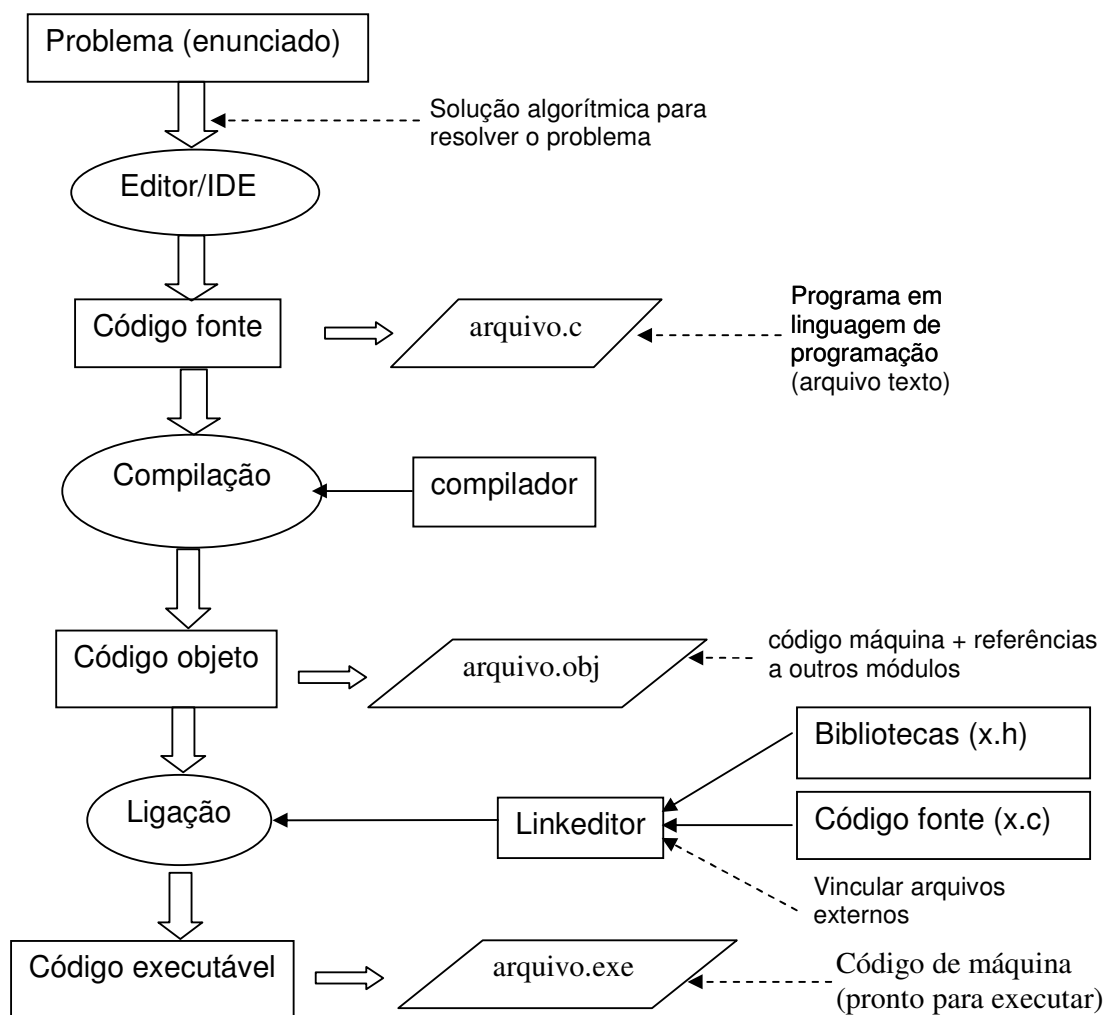
As instruções de um programa precisam ser analisadas para verificar se elas correspondem à sintaxe e à semântica da linguagem. Após essa verificação essas instruções precisam ser convertidas para uma forma de representação que possa ser executada pelo computador. Essa representação é denominada linguagem de máquina. Esse processo de verificação pode ser utilizado durante a elaboração do programa para auxiliar na identificação de erros.

O termo programa pode referir-se ao código fonte, escrito em uma linguagem de programação, ou ao arquivo que contém a forma executável desse código fonte. Os programas de computador são baseados na manipulação de informações contidas em sua memória que são referenciadas por variáveis definidas nesses programas. As instruções definem o processamento a ser realizado.

Uma linguagem permite especificar precisamente sobre quais dados um computador vai atuar, como atuar, armazená-los ou transmiti-los e quais ações devem ser tomadas sob determinadas circunstâncias.

O texto (palavras e símbolos) que é composto de acordo com as regras sintáticas e semânticas de uma linguagem de programação e utilizando suas palavras reservadas constitui o código fonte de um programa. Esse código é traduzido para código de máquina, que é executado pelo processador. A Figura 6 ilustra como esse processo ocorre na linguagem C.

Figura 6 – Do problema ao código executável



Para produzir o código fonte de um programa C é necessário ter um editor e um compilador. O editor pode ser qualquer um que produza texto não formatado, ou seja, sem caracteres especiais como marca de parágrafo e de tabulação. Para facilitar a tarefa de produção do código fonte há editores específicos, as IDEs (Integrated Development Environment) para uma ou várias linguagens. Eles reconhecem a sintaxe dos comandos dessas linguagens, facilitando a escrita correta do código. Um compilador pode estar associado ao editor e assim facilitar o processo de verificação e compilação do programa.

No processo de compilação o código fonte é analisado léxico, sintático e semanticamente. Se dessas análises não resultam erros é gerado o código que é traduzido para

a linguagem de máquina, gerando o programa executável.

O código fonte, normalmente é um arquivo texto contendo instruções de acordo com uma linguagem de programação. Esse código precisa ser convertido em binário para que possa ser executado pelo computador. O compilador verifica se os comandos, as estruturas e as declarações estão definidas de acordo as regras estabelecidas pela linguagem.

O processo de montagem traduz o código fonte de um programa em linguagem de máquina, substitui os códigos de instruções simbólicas em linguagem de programação para valores numéricos, reserva espaço na memória para as variáveis e para realizar as instruções e examina sintaticamente as instruções do código fonte.

A ligação ou linkedição é útil para reusar funções já implementadas, como as funções `printf()` e `scanf()`. O código dessas funções ou de arquivos externos é incorporado ou vinculado ao programa. Essas funções e arquivos podem estar armazenados em bibliotecas. O processo de linkedição resulta em um conjunto de códigos de máquina interligados e prontos para funcionar. Esses códigos compõem o programa executável.

O código executável é o programa depois de convertido para o formato binário. É esse código que é interpretado pelo computador, definindo as instruções a serem realizadas.

3.1.6 A linguagem C é estruturada

Um paradigma de programação está relacionado à forma de definir a solução de um problema. Estruturado ou imperativo e orientado a objetos são dois exemplos de paradigmas. Pelo paradigma estruturado (imperativo) qualquer problema pode ser resolvido por meio de estruturas sequencial, de decisão (condicional) e de repetição (iterativa). Esse paradigma procura segmentar um problema grande e/ou complexo em partes menores. O paradigma orientado a objetos compreende o problema como um conjunto de objetos que possuem características e comportamento e que interagem entre si por troca de mensagens.

A Linguagem C é estruturada e a Linguagem C++ é orientada a objetos e também estruturada. Isso porque permite a definição de programas de acordo com o paradigma estruturado.

3.1.7 Estrutura básica de um programa

Um programa implementa uma solução algorítmica para um problema. De maneira geral, por mais simples ou complexo que seja um algoritmo possui entrada, processamento e saída.

a) **Entrada**

São os valores informados ao programa e que são manipulados no processamento. Esses valores normalmente ficam armazenados em variáveis durante a execução do programa.

b) **Processamento**

São as instruções que tem o objetivo de transformar as entradas nas saídas esperadas, resolvendo o problema.

c) **Saída**

São textos (incluindo elementos gráficos) e conteúdo de constantes e de variáveis que representam o resultado do processamento.

Exemplo:

Problema:

Calcular a média aritmética de três notas.

Solução:

Entrada:

Três notas.

Processamento:

Somar as três notas.

Dividir a soma das notas por três, definindo a média.

Saída:

Média obtida.

Recomendações para a elaboração de um programa:

- a) No início do programa colocar o enunciado do problema;
- b) Comentar as partes mais complexas ou relevantes do programa;
- c) Utilizar espaços e linhas em branco para melhorar a legibilidade do programa, mas sem usá-los em demasia;
- d) Escolher nomes representativos para os identificadores, como nomes de variáveis e de funções;
- e) Colocar uma instrução por linha;
- f) Utilizar parênteses para aumentar a legibilidade das expressões e evitar erros;
- g) Utilizar indentação. Identar um código é fundamental para sua legibilidade. De forma simplificada cada bloco de código (que é delimitado por um conjunto { e }) que está dentro de um outro bloco de código deve estar deslocado um determinado número de espaços, sugere-se uma tabulação.

3.2 Tipos de dados

A entrada, o processamento e a saída de um programa baseiam-se na manipulação de dados.

A linguagem C possui cinco tipos básicos de dados. Esses tipos são utilizados na declaração de variáveis, na composição de tipos derivados ou estruturados e na definição dos parâmetros e do retorno de funções. Por exemplo, uma string, que é um tipo estruturado ou derivado é um conjunto de dados do tipo char que é um tipo básico. Os tipos básicos são:

a) **char** - é o tipo de dado para valores caractere, seja letra, número ou símbolo especial. Uma variável do tipo char pode armazenar um único caractere.

b) **int** - é o tipo de dado para valores numéricos inteiros.

c) **float** - é o tipo de dado para valores numéricos reais (ponto flutuante).

d) **double** - é o tipo de dado para valores ponto flutuante com precisão e abrangência de valor maior que o float.

e) **void** - é do tipo vazio, ou sem valor. Esse tipo é utilizado para retorno e parâmetro de função para indicar que a mesma não recebe parâmetros ou não possui retorno.

Esses tipos básicos de dados podem ser agrupados em numéricos (int, float e double), caractere (char) e sem tipo (void).

Existem modificadores que podem ser aplicados a alguns desses tipos básicos com o objetivo de alterar o tamanho do dado que pode ser armazenado pelo respectivo tipo. Os modificadores de tipo da linguagem C são: signed, unsigned, long e short. Na Figura 7 estão os tipos de dados permitidos, a sua faixa (valores máximos e mínimos) e a aplicação dos modificadores.

A faixa valores apresentados nesta figura tem como base um compilador típico para

um computador de arquitetura de 16 bits. Em uma máquina de 32 bits, int terá 32 bits e consequentemente poderá armazenar uma faixa maior de valores. Também nesta figura está especificada a string de formatação correspondente para as funções scanf() e printf().

O modificador unsigned significa que não é aplicado sinal (positivo ou negativo) à faixa de valores e signed que o sinal é aplicado.

Figura 7 – Tipos de dados

Tipo	Quantidade de bits	Formato para scanf() e printf()	Faixa de valor	
			Início	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	16	%i	-32.768	32.767
unsigned int	16	%u	0	65.535
signed int	16	%i	-32.768	32.767
long int	32	%li	-2.147.483.648	2.147.483.647
signed long int	32	%li	-2.147.483.648	2.147.483.647
unsigned long int	32	%lu	0	4.294.967.295
float	32	%f	3.4E-38	3.4E+38
double	64	%lf	1.7E-308	1.7E+308
long double	80	%lf	3.4E-4932	3.4E+4932

Os tipos quando não indicado modificador referem-se ao signed do respectivo tipo. O tipo long double é o tipo de ponto flutuante com precisão maior. Os intervalos de ponto flutuante estão indicados em faixa de expoente, mas os números podem assumir valores tanto positivos quanto negativos.

3.3 Variáveis e constantes

Uma variável é um identificador em um programa que permite manipular espaços de memória por meio do armazenamento e da leitura de dados. Esses espaços são denominados endereços de memória. Assim, uma variável é um identificador para um endereço de memória que será utilizado pelo programa para armazenar valores e recuperá-los (ler). Durante a execução de um programa, o conteúdo dos endereços de memória é manipulado. Esses endereços são reservados e manipulados por meio de identificadores (as variáveis).

Todas as variáveis na linguagem C devem ser declaradas antes de serem usadas. A declaração se refere a reservar espaços de memória. A forma geral de declaração de variáveis é:

tipo_da_variável identificador;

Onde:

tipo_da_variável é de um dos tipos básicos de dados da linguagem C: int float, double, char, com seus modificadores. O tipo indica o tamanho do espaço de memória necessário para armazenar valores do tipo definido para a variável.

identificador é o nome da variável. Para declarar uma lista de variáveis do mesmo tipo, os nomes são separados por vírgula.

Por exemplo, a instrução:

char Caractere, Letra;

Declaram duas variáveis do tipo char (Caractere e Letra). Essas variáveis podem armazenar um caractere qualquer, como 1, a, %.

int Ano;

Declara uma variável denominada Ano do tipo inteira, isto é, pode armazenar um valor inteiro na faixa de valores determinada pela arquitetura do computador.

float Salario;

Declara uma variável denominada Salario que pode armazenar valores reais (float) ou valores inteiros, por comporem um subconjunto dos reais.

A identificação de uma variável ocorre pelo seu nome que é o identificador. Assim, valores serão armazenados e acessados no endereço de memória reservado por meio do seu identificador.

Uma variável pode ser inicializada no momento de sua declaração. A forma geral de inicialização de uma variável é:

tipo_da_variável nome_da_variável = valor;

Valor pode ser uma constante, outra variável, o resultado de uma operação matemática, o retorno de uma função.

Exemplos de inicialização de variáveis:

char Caractere='D';

long int Horas=0;

float Salario=514.19;

Na linguagem C as variáveis não são automaticamente inicializadas com algum valor padrão no momento da sua declaração. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor qualquer. Esse valor pode ser o conteúdo de alguma outra variável que ocupou anteriormente esse endereço de memória ou um valor que é resultado da energização das células de memória. Não se deve presumir que uma variável possuirá zero, espaço em branco ou qualquer outro valor como conteúdo quando da sua declaração.

Uma variável que no momento da execução do programa receberá um valor informado pelo usuário, o valor de uma constante, o valor armazenado em outra variável ou o valor resultante de uma expressão não precisa ser inicializada. Porém, variáveis que terão seus valores alterados por operações que utilizam o valor que elas já possuem armazenados, devem ser inicializadas. Por exemplo:

Soma = Soma + Valor;

*Produto = Produto * Valor;*

A variável *Soma* deve ser inicializada com 0. Ela recebe o conteúdo da variável *Valor* mais o conteúdo que a variável *Soma* possuía armazenado. Como não se sabe o valor que estará armazenado na variável *Soma*, é indispensável inicializá-la como zero. A inicialização com o valor zero é feita para que o conteúdo que está no endereço de memória reservado para a variável quando da sua declaração não influencie no resultado das operações realizadas.

A variável *Produto* deve ser inicializada com 1. Isso porque é multiplicado o conteúdo da variável *Valor* com o conteúdo da variável *Produto* e em seguida esse valor é armazenado na variável *Produto*. Como não se sabe o valor que está inicialmente armazenado na variável *produto*, é indispensável inicializá-la como 1. A inicialização com 1 é feita porque

se trata de uma multiplicação. Se a variável *Produto* fosse inicializada com 0, o resultado seria zero, e se inicializada com outro valor, o resultado da multiplicação seria alterado.

Operações que utilizam o conteúdo que já está na variável, operando com esse valor e armazenando um novo valor na variável - como as variáveis *Soma* e *Produto* no exemplo anterior – implicam necessidade de inicialização das variáveis, porque não se sabe o conteúdo do endereço de memória reservado para elas.

3.4 Operadores e expressões

Os operadores da linguagem C são os aritméticos, os relacionais e os lógicos. Eles são utilizados em expressões matemáticas e em expressões que possuem um resultado lógico.

3.4.1 Operadores Aritméticos

Os operadores aritméticos são usados na representação de expressões matemáticas. A Figura 8 contém os operadores aritméticos da linguagem C.

Figura 8 – Operadores aritméticos

Operador	Significado
+	Soma (inteira e ponto flutuante)
-	Subtração ou troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira se ambos os operandos foram inteiros e ponto flutuante se um dos operandos for (ou ambos forem) float)
%	Resto de divisão (com obrigatoriamente ambos os operando inteiros)
++	Incremento de um (inteiro e ponto flutuante)
--	Decremento de um (inteiro e ponto flutuante)

Uma operação aritmética pode ser unária ou binária.

Uma operação unária ocorre quando um operando atua sobre uma única variável, por exemplo:

```
Num = -3;
```

```
-Num;
```

Transformará o conteúdo da variável *Num* em 3.

Os operadores de incremento e de decremento são unários. Eles alteram a variável à qual estão aplicados. O que eles fazem é incrementar ou decrementar de um (1) essa variável. Assim,

```
A++;
```

```
A--;
```

equivale a:

```
A = A+1;
```

```
A = A-1;
```

Esses operadores podem ser pré-fixados ou pós-fixados. Operadores pré-fixados incrementam o valor da variável e retornam o valor da variável já incrementada. Operadores pós-fixados retornam o valor da variável sem o incremento e em seguida o valor da variável é

incrementado. Assim,

```
A=23;
B=A++;
resultam:
B=23
A=24.
e
A=23;
B=++B;
resultam:
B=24
A=24.
```

Uma operação binária requer que dois operandos sejam vinculados por um operador.

Exemplo:

a) 3.0 / 10

Representa uma divisão float entre 3 e 10. É uma divisão com resultado float porque um dos operandos é float.

b) 3/10

Representa uma divisão inteira entre 3 e 10. É uma com resultado inteiro porque ambos os operandos são inteiros.

O operador / (divisão) quando aplicado a variáveis ou constantes inteiras fornece o resultado da divisão inteira; quando aplicado a variáveis ou constantes (dividendo ou divisor ou ambos) de ponto flutuante fornece o resultado da divisão real.

O operador % fornece o resto da divisão de dois valores inteiros. Para que esse operador possa ser utilizado é obrigatório que dividendo e divisor sejam do tipo inteiro.

As seguintes declarações de variáveis e operações matemáticas:

```
int A = 17, B = 3, C, D;
float E = 17.0, F, G;
C = A / B;
D = A % B;
F = E / B;
G = A / B;
resultam:
```

C = 5, D = 2, F = 5.666666 e G = 5.000000.

Na instrução G = A / B, inicialmente é feita uma divisão inteira (porque os dois operandos são inteiros). Depois de efetuada essa divisão, o resultado é atribuído a uma variável do tipo float. Contudo, a divisão realizada é inteira (as variáveis A e B são inteiras) e posteriormente armazenada em uma variável float.

O operador de atribuição da linguagem C é o = (igual). O que ele faz é atribuir o valor da direita para a variável à esquerda. Ressalta-se que se à direita do sinal de atribuição está uma expressão e/ou chamadas a funções, primeiro essa expressão é resolvida, funções são executadas e em seguida o valor resultante é atribuído à variável.

O operador de atribuição (=) não deve ser confundido com o operador lógico de comparação (==). Por exemplo, em if (A = B), a expressão será verdadeira se B for diferente de zero. O

valor de B é atribuído para A e se ele for diferente de zero é considerado verdadeiro para o teste lógico. Nesse teste lógico não está sendo comparado o conteúdo de A com o conteúdo de B. Para haver comparação deveria ser `A == B`. Nesse teste está sendo atribuído o valor de A para B e esse valor é usado para tomar a decisão por meio do `if`.

3.4.2 Operadores Relacionais e Lógicos

Os operadores relacionais da linguagem C realizam comparações com resultado lógico entre variáveis e/ou constantes, podendo incluir expressões aritméticas. A Figura 11 apresenta os operadores relacionais.

Figura 11 – Operadores relacionais

Operador	Significado
<code>></code>	Maior do que
<code>>=</code>	Maior ou igual a
<code><</code>	Menor do que
<code><=</code>	Menor ou igual a
<code>==</code>	Igual a
<code>!=</code>	Diferente de

Na linguagem C, os operadores relacionais retornam verdadeiro (um valor diferente de zero, normalmente 1) ou falso (o valor 0).

Em um teste de comparação lógica o resultado será verdadeiro se o valor for diferente de zero. Por isso:

`X = 10;`

A instrução `if (X)` resultará verdadeiro.

`X = 0;`

A instrução `if (X)` resultará falso.

Os operadores lógicos podem ser usados para criar expressões lógicas complexas por meio da combinação de condições simples. Para fazer comparações com valores lógicos (verdadeiro e falso) a linguagem C possui os operadores apresentados na Figura 12.

Figura 12 – Operadores lógicos

Operador	Significado
<code>&&</code>	Uma expressão E (AND) é verdadeira se todas as condições forem verdadeiras.
<code> </code>	Uma expressão OU (OR) é verdadeira se pelo menos uma condição for verdadeira
<code>!</code>	Uma expressão NÃO (NOT) inverte o valor da expressão ou condição, se verdadeira inverte para falsa e vice-versa.

A linguagem C não possui um operador definido para o OU exclusivo. Uma expressão OU exclusivo (XOR) é verdadeira quando apenas uma das condições for verdadeira. Contudo, é possível obter esse operador por uma combinação de operadores E e O com a seguinte

lógica: se uma das condições é verdadeira e a outra não é ou se uma não é verdadeira e a outra é.

A Figura 13 apresenta um exemplo de tabela verdade utilizando operadores lógicos. Sendo que: X = 3 e Y = 5.

Figura 13 – Exemplo de uso de operadores lógicos

X	Y	X && Y	X Y	! X	! Y
6	6	falso	Falso	verdadeiro	Verdadeiro
6	5	falso	verdadeiro	verdadeiro	Falso
3	6	Falso	verdadeiro	falso	Verdadeiro
3	5	verdadeiro	verdadeiro	falso	Falso

O uso de parênteses pode alterar a ordem de avaliação de uma expressão que contém operadores lógicos e relacionais.

Exemplo:

Var1 = 2;

Var2 = 3;

Var3 = 5;

Uma expressão composta com essas três variáveis, utilizando operadores lógicos e parênteses:

if (Var1 > 5 && (Var2 = 5 || Var3 = 5))

1º é verificado internamente aos parênteses

Var2 = 5		Var3 = 5
F		V
		V

2º o restante da expressão é verificado

Var1 > 5	&&	(Var2 = 5 OU Var3 = 5)
Var1 > 5	&&	V
F	&&	V
		F

A mesma expressão sem parênteses:

if (Var1 > 5 && Var2 = 5 || Var3 = 5)

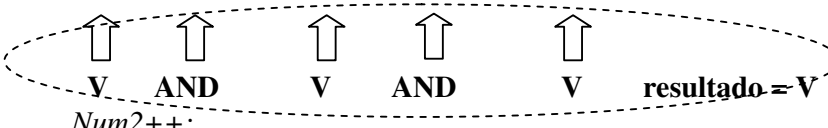
a expressão é verificada linearmente da esquerda para a direita.

Var1 > 5	&&	Var2 = 5		Var3 = 5
F	&&	F		V
		F		V
		V		

Por meio desse exemplo é possível verificar que o uso de parênteses pode definir um resultado diferente para uma expressão lógica. O mesmo ocorre com expressões aritméticas.

No programa a seguir, a seguir a operação Num2++ será executada, pois o resultado da expressão lógica é verdadeiro:

```
#include <stdio.h>

int main(void)
{
    int Num1 = 5, Num2 = 7;
    if ( (Num1 > 3) && (Num2 <= 7) && ( Num1 != Num2) )
    {
        
        Num2++;
    }
    printf("O valor na variável Num2 é %d", Num2);
}
```

A execução desse programa resultará na impressão de *O valor na variável Num2 é 8*. A instrução Num2++ incrementou 1 ao conteúdo da variável Num2, que era 7.

O programa a seguir, imprime na tela os números pares entre 1 e 100, apesar da variação de Num1 ocorrer de 1 em 1. Isso ocorre porque o teste lógico realizado pelo if somente imprimirá o número se o resto da divisão for zero.

```
/* Imprime os números pares entre 1 e 100. */
#include <stdio.h>

int main(void)
{
    int Num1;
    for(Num1=1; Num1<=100; Num1++)
    {
        if( Num1 % 2 == 0 ) //Se o resto da divisão de Num1 por 2 for igual a 0
        {
            printf("%d ", Num1);
        }
    }
}
```

3.4.3 Expressões matemáticas

Expressões matemáticas combinam valores e operadores. Os valores podem ser provenientes de variáveis, do retorno de funções ou constantes. Na composição de uma expressão é necessário considerar a ordem de execução dos operadores, de acordo com a precedência estabelecida pela linguagem C (Figura 15). Essa ordem pode ser alterada utilizando parênteses.

Exemplos de expressões:

Anos=Dias / 365.25;

O conteúdo da variável Dias é dividido por 365.25 e armazenado na variável Anos. É necessário que a variável Anos seja declarada como float para que o resultado real dessa expressão seja armazenado na variável Anos.

Pessoas = Pessoas+3;

Ao conteúdo da variável Pessoas é acrescentado 3 e o valor obtido é armazenado novamente na variável Pessoas.

*Resultado = Dias * ValorDiario + TotalGeral / ValorParcial;*

Essa expressão é resolvida de acordo com a prioridade dos operadores. Ressalta-se que se TotalGeral e ValorParcial forem ambos do tipo inteiro, o resultado dessa divisão será inteiro.

*Resultado = A + B * (C - D) / (E + F);*

Nessa expressão, as operações entre parênteses têm prioridade mais alta, são realizadas primeiro.

Resultado = sqrt(10) + sqrt(Anos + Pessoas);

sqrt() é uma função que recebe um valor (constante ou conteúdo de uma variável), calcula a raiz quadrada e retorna esse valor que é utilizada na expressão matemática que após resolvida tem o valor obtido armazenado na variável Resultado.

Expressões matemáticas que tem como objetivo alterar o conteúdo de uma única variável pelo acréscimo ou redução de um valor podem ser representadas de maneira abreviada. A Figura 14 exemplifica essas expressões.

Figura 14 – Exemplos de expressões abreviadas

Expressão	Expressão abreviada
Var = Var + 2;	Var += 2;
Var = Var - 2;	Var -= 2;
Var = Var * 2;	Var *= 2;
Var = Var / 2;	Var /= 2;
Var = Var + 1;	Var ++;
Var = Var -1;	Var --;

A resolução das expressões é realizada de acordo com regras de precedência de operadores. A Figura 15 apresenta a tabela de precedência dos operadores em C.

Figura 15 – Tabela de precedência dos operadores em C

Maior precedência	() [] →
	! ++ -- -(unário) (type cast)
	* / %
	+ -
	== !=
	!
	&&
	? ;
	= += -= *= /= %=
Menor precedência	,

Parênteses podem ser utilizados para tornar expressões mais legíveis e para alterar a ordem de precedência dos operadores.

Um modelador de tipo (type cast) é utilizado para transformar um tipo de dado em outro. Essa transformação não altera o valor armazenado na variável. A alteração ocorre apenas no uso do conteúdo da variável. Um modelador para inteiro aplicado a um tipo de dado float faz com que seja considerada apenas a parte inteira desse valor. Um modelador float aplicado a um tipo de dado int transforma-o em float. De maneira ilustrativa, é como se fosse acrescentado .0 no final do número.

A forma geral do modelador de tipo é:

(tipo) variável ou constante

Onde:

(tipo) indica o tipo que o conteúdo da variável ou a constante deve ser transformado.

Variável contém o dado a ser transformado.

Constante é o dado a ser transformado.

Exemplo:

```
#include <stdio.h>
```

```
int main(void)
{
    int Num1 = 10;
    float Num2;
    Num2 = (float) Num1/7; /* o modelador (float) força a transformação de inteiro
em real */
    printf("%d / 7 = %f",Num2);
}
```

Na instrução (float) Num1/7, inicialmente Num1, que é inteiro, é transformado em float, resultando em 10.0 e em seguida é realizada a operação de divisão. Como Num1 foi transformado em float antes da divisão, é realizada uma divisão float. O mesmo ocorreria, com Num1 / (float) 7, porque antes da divisão, 7 é transformado em float, ficando 7.0. Também é válido Num1 / 7.0, ou seja, transformar a constante inteira 7 em constante ponto flutuante 7.0.

Nesse exemplo, se não houvesse o uso de modelador, seria realizada uma divisão inteira entre 10 e 7. O resultado seria 1 (um) e este seria posteriormente convertido para float, como 1.000000. Também resultaria 1.000000, com a expressão (float) (Num1/7), porque primeiro seria realizada a divisão inteira, pois Num1 e 7 são inteiros, e posteriormente o resultado seria convertido para float, resultando 1.000000.

3.5 Instruções de entrada e saída

Entrada e saída se referem aos dados que o programa recebe durante a sua execução e aos resultados do seu processamento que são apresentados aos usuários, acionam dispositivos, são armazenados em banco de dados ou transmitidos para outros sistemas, dentre outros.

Existem diversas funções disponibilizadas sob a forma de bibliotecas para a entrada e a saída de dados na linguagem C. Dentre essas funções estão printf(), scanf() e gets() que

serão apresentadas neste texto. A função `printf()` é utilizada para saída. A função `scanf()` é utilizada para obter entradas de dados dos tipos inteiro, ponto flutuante e caractere e a função `gets()` para obter dados do tipo texto (string).

A função `printf()` tem a seguinte forma geral:

```
printf(expressão_de_controle, lista_de_argumentos);
```

onde:

expressão de controle é composta por: caracteres (texto) que serão mostrados literalmente na tela; as posições nas quais será colocado o conteúdo de variáveis, constantes, o retorno de funções que é indicado por caracteres de controle e o resultado de expressões aritméticas e lógicas; e caracteres de formatação.

lista de argumentos contém as variáveis, as constantes, o retorno de funções e o resultado de expressões aritméticas e lógicas que substituirão os caracteres de controle. Os argumentos são separados por vírgula.

Os caracteres de controle usam a notação `%`. Essa notação é seguida do tipo de dado e indica onde na expressão de controle que serão substituídos pelos argumentos da lista de argumentos, na ordem colocada nessa lista. Para cada caractere de controle é necessário ter um argumento na lista de argumentos do tipo indicado por esses caracteres.

Os caracteres de formatação indicam a formatação na apresentação da expressão de controle como, por exemplo, nova linha e a marca de tabulação.

A Figura 16 apresenta algumas dos códigos de controle utilizadas pelas funções `printf()` e `scanf()`.

Figura 16 – Códigos de controle

Formato	Significado
<code>%c</code>	Caractere (char)
<code>%d</code>	Número inteiro decimal (int)
<code>%i</code>	O mesmo que <code>%d</code>
<code>%e</code>	Número em notação científica com o "e" minúsculo
<code>%E</code>	Número em notação científica com o "E" maiúsculo
<code>%f</code>	Número ponto flutuante decimal
<code>%o</code>	Número octal
<code>%s</code>	String
<code>%u</code>	Número decimal sem sinal (unsigned)
<code>%x</code>	Número hexadecimal com letras minúsculas
<code>%X</code>	Número hexadecimal com letras maiúsculas
<code>%%</code>	Imprime o caractere <code>%</code>
<code>%p</code>	Ponteiro

A Figura 17 apresenta exemplos de uso de códigos de controle na função `printf()`.

Figura 17 – Exemplos de uso de códigos de controle

Instrução	É apresentado
<code>printf ("Teste %% %")</code>	Teste % %
<code>printf ("Exemplo de %s ", "texto")</code>	Exemplo de texto
<code>printf ("%s%d%", "Aumento de ", 10)</code>	Aumento de 10%
<code>printf ("%f", 40.343032)</code>	40.343
<code>printf ("Caractere %c. Inteiro %d", 'C', 123)</code>	Caractere C. Inteiro 123

Os caracteres de formatação são utilizados pela função `printf()` com o objetivo de formatar o conteúdo da expressão de controle e imprimir caracteres que são utilizados como parte da sintaxe da função `printf()` ou de caracteres de formatação. A Figura 18 apresenta caracteres de formatação.

Figura 18 – Caracteres de formatação

Caractere	Significado
<code>\n</code>	nova linha
<code>\r</code>	enter
<code>\t</code>	tabulação (tab)
<code>\b</code>	retrocesso
<code>\"</code>	aspas
<code>\\</code>	barra

Exemplo de uso de caracteres de formatação:

```
#include <stdio.h>

int main(void)
{
    char Caractere;
    Caractere='A';
    printf("Uma quebra de linha\n %c e uma linha em branco no final\n",Caractere);
}
```

Nesse programa, é impresso na tela do computador *Uma quebra de linha*, na linha seguinte é escrito *A e uma linha em branco no final* e o cursor ficará posicionado no início da próxima linha. Os caracteres `\n` colocam uma quebra de linha no lugar em que eles foram definidos na expressão de controle.

A função `scanf()` é utilizada para leitura de valores informados pelo usuário e que são armazenados em variáveis.

O formato geral da função `scanf()` é:

scanf (expressão_de_controle,lista_de_argumentos);

onde:

expressão de controle é composta por caracteres (texto) que serão mostrados literalmente na tela e caracteres de formatação.

lista de argumentos são as variáveis que armazenarão os valores informados como entrada. Para cada entrada deve ser indicado o respectivo tipo, na ordem das variáveis que irão armazená-los na expressão de controle. Os argumentos são separados por vírgula. É indispensável colocar o `&` antes de cada uma das variáveis da lista de argumentos, exceto para

as variáveis declaradas como ponteiro, incluindo variáveis do tipo string, que são ponteiros.

Exemplos:

```
#include <stdio.h>
```

```
int main(void)
{
    char Letra;
    printf("Informe um caractere: ");
    scanf("%c",&Letra);
    printf("O caractere informado é %c",Letra);
}
```

A instrução *char Letra;* declara uma variável do tipo char que poderá armazenar um caractere, seja letra, número ou símbolo especial. A instrução *scanf("%c",&Letra);* indica que será lido um dado do tipo caractere (%c) e ele será armazenado no endereço de memória reservado para a variável Letra. O operador & indica que está sendo passado um ponteiro (indicador) para o endereço de memória reservado para a respectiva variável.

```
#include <stdio.h>
```

```
int main(void)
{
    int Dia;
    int Mes;
    int Ano;
    printf("Informe uma data no formato dd/mm/aaaa: ");
    scanf("%d/%d/%d",&Dia,&Mes,&Ano);
    printf("Data informada no formato mm/dd/aaaa: ",Mes,Dia,Ano);
}
```

A função *gets()*, que normalmente está na biblioteca *stdio.h*, é utilizada para a leitura de strings (vetores do tipo caractere ou variáveis que armazenam texto). A função *scanf()* também pode ser utilizada para a leitura de strings, mas ela só lê até o primeiro caractere de espaço que é informado pelo usuário como parte da string. Essa limitação não ocorre com *gets()*.

Exemplo de programa que usa a função *gets()*:

```
#include <stdio.h>
```

```
int main(void)
{
    char Texto[11];
    printf("Escreva um texto com até 10 caracteres:");
    gets(Texto);
    // scanf("%s", Texto); /*scanf() equivalente a gets(). Lembrando da limitação do
    scanf( ) para leitura de texto que finaliza no primeiro caractere de espaço
    pressionado*/
    printf("Texto informado: %s", Texto);
}
```

3.6 Estruturas de controle

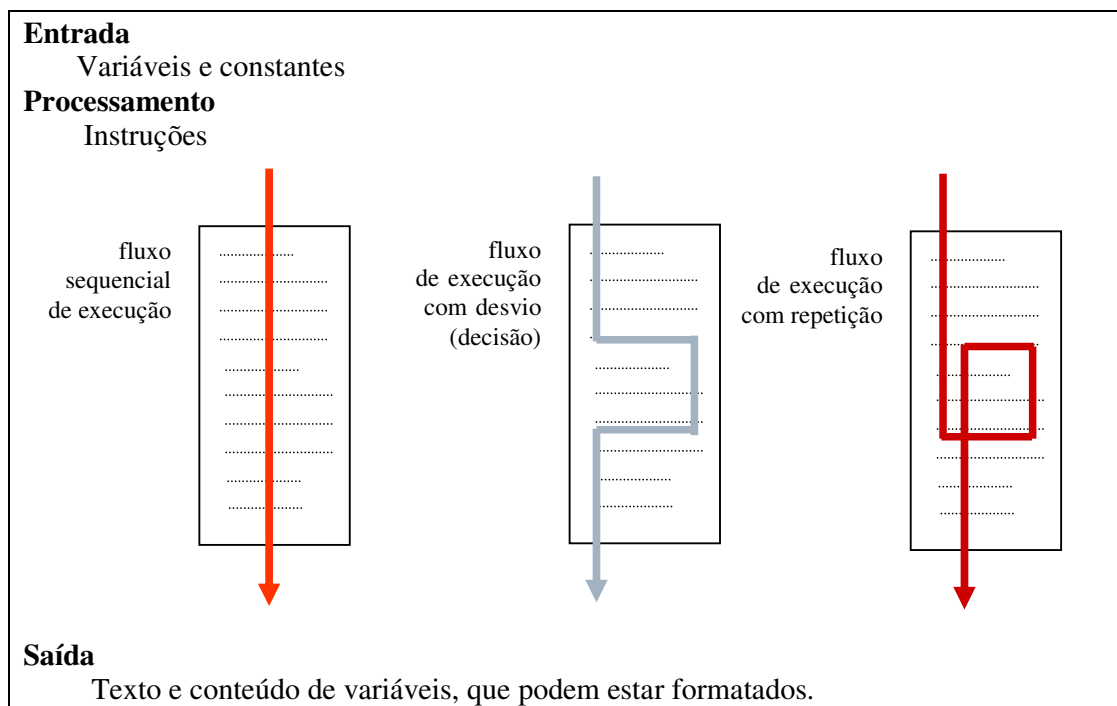
As instruções que compõem um programa em linguagem C podem ser executadas de três formas básicas:

- a) Todas as instruções são executadas e na sequência em que elas estão no código do programa, a cada vez que o programa é utilizado;
- b) Parte delas será executada, dependendo se determinadas condições ocorrerem;
- c) Parte delas será repetida um determinado número de vezes ou até que uma condição seja atendida ou enquanto uma condição está sendo atendida.

A forma de execução das instruções de um programa define o que é denominado estrutura de controle.

A Figura 19 representa esquematicamente as três estruturas de controle da linguagem C. Essas estruturas indicam como o processamento de um programa é realizado. A Figura 19 simplesmente mostra que o processamento pode ocorrer de uma dessas três formas. Ressalta-se que declaração de variáveis, entradas e saídas de dados podem fazer parte dessas estruturas.

Figura 19 - Representação esquemática das estruturas de controle



De acordo com a representação da Figura 19, o processamento ocorre de três formas básicas: sequencial, decisão e repetição. As estruturas de decisão e de repetição são denominadas de controle porque controlam se determinadas instruções serão ou não realizadas. Esses três tipos de estruturas podem ser utilizados nas mais diversas combinações, desde que devidamente aninhadas, ou seja, uma estrutura deve estar completamente contida dentro de outra.

3.6.1 Estrutura sequencial

Em uma estrutura sequencial todas as instruções são executadas exatamente na ordem

em que elas estão colocadas no código do programa, de cima para baixo e da esquerda para a direita, da primeira até a última instrução.

A forma geral da representação de uma estrutura sequencial é:

Instrução1

Instrução2

...

InstruçãoN-1

InstruçãoN

Onde:

Instrução são as instruções que compõem o programa.

Exemplo de programa utilizando estrutura sequencial para calcular a área de um círculo:

Descrição do problema: calcular a área de um círculo.

Análise do problema:

Dados de entrada:

Raio do círculo.

Valor de Pi.

Processamento:

Calcular a área do círculo pela fórmula: $\text{Área} = \text{Pi} * \text{raio}^2$.

Saída:

O valor obtido do cálculo da área do círculo.

Na Figura 20 está a representação em fluxograma e na linguagem C do algoritmo para calcular a área de um círculo com o raio informado pelo usuário.

Figura 20 – Fluxograma e programa C para calcular a área de um círculo

Em fluxograma	Em linguagem C
<pre> graph TD Inicio([início]) --> LerRaio[/Ler Raio Pi <-- 3.14159/] LerRaio --> CalculaArea[Area <-- Pi * Raio * Raio] CalculaArea --> EscreverArea[/Escrever Area/] EscreverArea --> Fim([fim]) </pre>	<pre> #include <stdio.h> int main(void) { float Raio; float Area; float Pi; Pi = 3.14159; printf("Informe o valor do raio: "); scanf("%f",&Raio); Area = Pi * Raio * Raio; printf("A área do círculo é de %.2f",Area); } </pre>

O valor de Pi é uma constante e dessa forma não precisa ser informado pelo usuário. O que a define como constante é o fato de a mesma permanecer com um valor fixo independentemente das instruções ou no número de execuções do programa. O valor de Pi poderia ser incluído diretamente na expressão que calcula a área do círculo ($\text{Area} = 3.14159 *$

Raio * Raio). A variável Pi também poderia ter sido definida com constante com a declaração `const float Pi`.

As variáveis Area, Raio e Pi foram declaradas do tipo float para que fosse possível armazenar valores reais (com casas decimais).

A linguagem C não possui um operador para a potenciação (apenas funções). Assim, raio ao quadrado é indicado por `Raio * Raio`;

A formatação `%.2f` define que sejam mostradas apenas duas casas decimais do conteúdo da variável Area.

As instruções do programa representado na Figura 19 são realizadas sequencialmente, da primeira para a última. Isso caracteriza uma estrutura sequencial.

Outro exemplo de estrutura sequencial com o objetivo de calcular a média entre duas notas.

Descrição do problema: calcular a média de duas notas.

Análise do problema:

Dados de entrada:

Duas notas.

Processamento:

Somar as duas notas.

Dividir o resultado da soma por 2.

Saída:

A média obtida.

A Figura 21 contém a representação em fluxograma e na linguagem C do algoritmo para calcular a média de duas notas informadas pelo usuário.

Figura 21 – Fluxograma e programa C para calcular a média de duas notas

Em fluxograma	Em linguagem C
<pre> graph TD Inicio([início]) --> Ler[Ler Nota1 Ler Nota2] Ler --> Processa[Media <-- (Nota1 + Nota2)/2] Processa --> Escreve[/Escrever Media/] Escreve --> Fim([fim]) </pre>	<pre> #include <stdio.h> int main(void) { float Nota1; float Nota2; float Media; printf("Informe a primeira nota: "); scanf("%f",&Nota1); printf("Informe a segunda nota: "); scanf("%f",&Nota2); Media = (Nota1 + Nota2) / 2; printf("\nA média entre %.2f e %.2f é %.2f",Nota1,Nota2, Media); } </pre>

Para calcular a média, primeiro é necessário somar as duas notas e depois dividir o resultado dessa soma pela quantidade de notas somadas. Assim, é indispensável indicar por meio de parênteses que primeiro seja feita a soma. Isso é necessário em decorrência da prioridade das operações: primeiro a divisão e depois a soma. Se não fosse feito dessa forma,

primeiro seria dividido o valor da segunda nota por 2 (a quantidade de notas) e o resultado dessa divisão seria somada com a primeira nota. Não haveria erro de sintaxe, as instruções estariam corretas, mas haveria erro de contexto, porque o resultado obtido não é a média dos valores, e sim a metade da segunda nota somada com a primeira.

Nesses programas (Figuras 20 e 21) todas as instruções são executadas na sequência em que foram definidas no programa, independentemente dos valores informados. Além disso, para calcular o raio de um novo círculo ou a média entre outras duas notas, o programa deve ser executado novamente.

Contudo, nem sempre todas as instruções devem ser executadas em todas as execuções de um programa. E, também, pode ser que seja necessário repetir um determinado conjunto de instruções, como calcular a média de trinta pares de números, por exemplo.

Para atender a essas necessidades são utilizadas as estruturas de controle de fluxo. Elas são fundamentais para qualquer linguagem de programação. As estruturas de controle de fluxo da linguagem C podem ser divididas em decisão (if, if ... else if, switch case) e repetição (for, while, do ... while).

3.6.2 Estruturas de controle decisão

As estruturas de decisão permitem alterar a sequência de execução do programa dependendo de resultados de testes lógicos que incluem operações realizadas pelo programa e valores de entrada.

Uma estrutura de decisão é também conhecida como estrutura condicional. Nesse tipo de estrutura a execução de um conjunto de instruções está subordinada ao resultado de um teste lógico. Uma estrutura de decisão deve ser usada quando é necessário que determinadas instruções que compõem o programa sejam realizadas em decorrência do resultado de um teste lógico.

Com as instruções de decisão é possível fazer com que determinadas instruções sejam ou não realizadas. As estruturas de decisão da linguagem C são: if (se então), if else (se então ... senão) e switch case (caso ... selecione).

3.6.2.1 if

A estrutura de decisão if (se ... então) é utilizada para executar um conjunto de instruções se o resultado de um teste lógico for verdadeiro.

A forma geral da estrutura de decisão if na linguagem C é:

Em português estruturado	Em linguagem C
<i>se <condição> então</i> <i><conjunto de instruções></i> <i>fim se</i>	<i>if (condição)</i> <i>{</i> <i>//conjunto de instruções;</i> <i>}</i>

Onde:

condição se refere ao resultado um teste lógico.

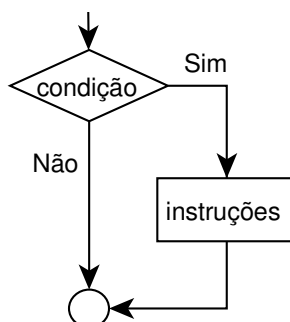
conjunto de instruções são as instruções que serão executadas se o resultado da condição é verdadeiro.

A linguagem C não possui um tipo de dado booleano. É feita uma associação com valores numéricos. Zero é falso e qualquer valor diferente de zero é verdadeiro.

Se o conjunto de instruções possuir mais de uma instrução é indispensável delimitar o bloco de código por chaves ({ e }) e cada instrução desse conjunto é finalizada com ponto e vírgula (;). O resultado do teste lógico (realizar ou não as instruções) é aplicado somente às instruções que estão dentro das chaves que delimitam a estrutura if.

A Figura 22 apresenta o fluxograma da estrutura de decisão if.

Figura 22 - Fluxograma da estrutura de decisão if



Esse fluxograma indica que um teste lógico é realizado, definindo uma condição (verdadeira ou falsa). Se o resultado do teste lógico é verdadeiro as instruções serão realizadas e é executada a próxima instrução do programa após o bloco de instruções pertencentes à estrutura. Se o resultado é falso o programa segue diretamente na próxima instrução após a finalização da estrutura.

Exemplo de programa utilizando estrutura de decisão:

Descrição do problema: determinar se um aluno está aprovado caso sua nota seja maior ou igual a 7.

Análise do problema:

Dados de entrada:

Nota do aluno

Processamento:

Verificar se a nota é igual ou maior que 7 e em caso afirmativo informar que o mesmo está aprovado.

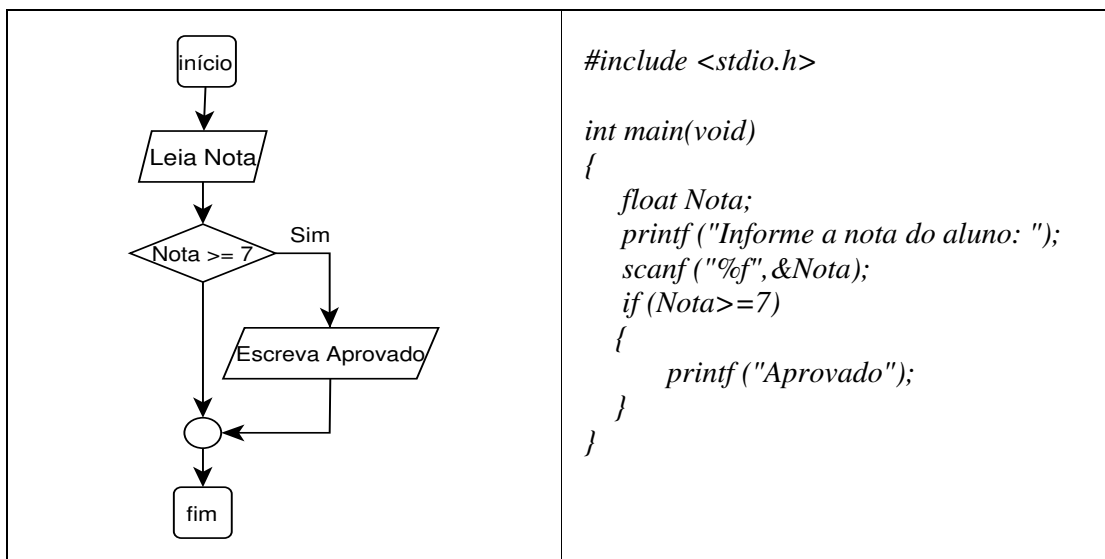
Saída:

Informar que o aluno está aprovado se a sua nota é igual maior que 7.

Na Figura 23 estão o fluxograma e o programa em linguagem C para indicar se um aluno está aprovado em decorrência da sua nota.

Figura 23 – Fluxograma e programa C para indicar se um aluno está aprovado em decorrência da sua nota

Em fluxograma	Em linguagem C
---------------	----------------



De acordo com a representação da Figura 23, um teste lógico é realizado para verificar se o conteúdo da variável *Nota* é maior ou igual a 7. Caso seja igual ou maior que 7, a mensagem aprovado é apresentada. Se o valor contido na variável *Nota* é menor que 7, as instruções contidas no *if* não serão realizadas.

Ressalta-se que para testar igualdades é utilizado o operador `==` e não `=`. Isto porque o operador `=` representa uma atribuição na linguagem C.

Por exemplo, no código:

```

int Numero = 30;
if (Numero=10) /*essa instrução está sintaticamente correta, mas está
semanticamente incorreta para o contexto do programa, que é verificar se o conteúdo da
variável Numero é igual a 10.*/
{
    printf("%d é igual a 10",Numero);
}
  
```

Nesse exemplo, o compilador atribui o valor 10 à variável *Numero* e a expressão *Numero=10* retorna 10 porque foi atribuído 10 à variável *Numero*, fazendo com que o valor de *Numero* seja modificado e assim a instrução `printf("%d",Numero)` será sempre executada, independentemente do valor que estava armazenado em *Numero* antes da comparação no *if*. Em decorrência dessa atribuição, o resultado da comparação será sempre verdadeiro. O programa está sintaticamente correto, o mesmo é compilado e executado. Porém, ele não realiza o objetivo pretendido que é verificar se o conteúdo da variável *Numero*, atribuídos antes da comparação *if*, é igual a 10.

3.6.2.1 else if

A estrutura *if else* (se ... então ... senão) é utilizada para executar um conjunto de instruções se uma determinada condição é verdadeira e para executar outro conjunto de instruções se essa condição é falsa.

Forma geral de uma estrutura de decisão **if else** na linguagem C:

Em português estruturado	Em linguagem C
<i>se <condição> então</i> <conjunto de instruções 1> <i>senão</i> <conjunto de instruções 2> <i>fim se</i>	<i>if (condição)</i> { conjunto de instruções 1; } else { conjunto de instruções 2; } }

Onde:

condição determina um teste lógico.

conjunto de instruções 1 define as instruções que serão realizadas se o resultado do teste lógico for verdadeiro.

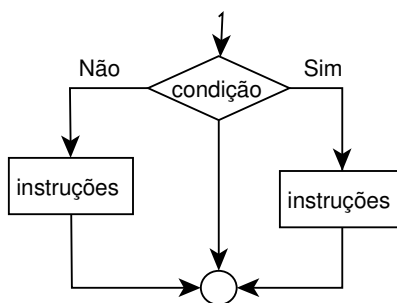
conjunto de instruções 2 define as instruções que serão realizadas se o resultado do teste lógico for falso.

O else pode ser pensado como um complemento do if. Porque ele define o que será realizado se o resultado do teste lógico contido em if é falso.

Lembrando que a linguagem C não possui um tipo booleano. Então, a expressão da condição será avaliada e se o seu resultado for diferente de zero o conjunto de instruções 1 será executado; se for zero, o conjunto de instruções 2 será executado. Em uma estrutura if else uma das duas declarações será executada, nunca serão executadas ambas e nunca nenhuma delas.

Figura 25 apresenta o fluxograma da estrutura de decisão if else.

Figura 25 - Fluxograma da estrutura de decisão if else



Exemplo de programa utilizando a estrutura de decisão if else.

Descrição do problema: determinar se um aluno está aprovado se a sua nota é maior ou igual a 7 ou reprovado se a nota é menor que 7.

Análise do problema:

Dados de entrada:

Nota do aluno

Processamento:

Verificar se a nota.

Se a nota é maior ou igual a 7 informar que o aluno está aprovado.

Se a nota não é maior ou igual a 7 informar que o aluno está reprovado.

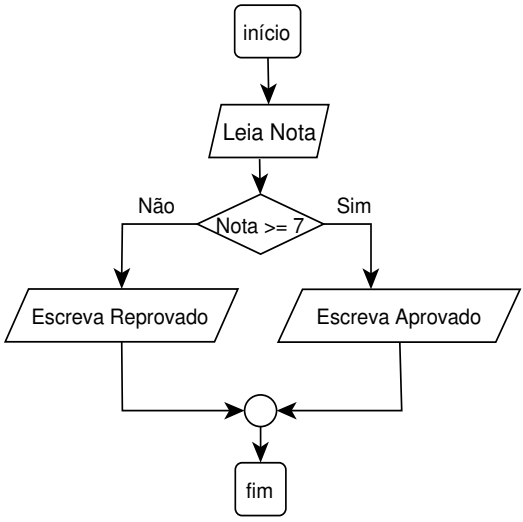
Saída:

Informar que o aluno está aprovado ou reprovado de acordo com a nota informada e o resultado do teste realizado.

Na Figura 26 está o fluxograma e o programa em linguagem C para indicar se um

aluno está aprovado ou reprovado em decorrência da sua nota.

Figura 26 – Fluxograma e programa C para indicar se um aluno está aprovado ou reprovado em decorrência da sua nota

Em fluxograma	Em linguagem C
 <pre> graph TD inicio([início]) --> leiaNota[/Leia Nota/] leiaNota --> notaGe7{Nota >= 7} notaGe7 -- Não --> escrevaReprovado[/Escreva Reprovado/] notaGe7 -- Sim --> escrevaAprovado[/Escreva Aprovado/] escrevaReprovado --> fim([fim]) escrevaAprovado --> fim </pre>	<pre> #include <stdio.h> int main(void) { float Nota; printf("Informe a nota do aluno: "); scanf("%f", &Nota); if (Nota >= 7) { printf("Aprovado"); } else { printf("Reprovado"); } } </pre>

De acordo com a representação da Figura 26, um teste lógico é realizado para verificar se o conteúdo da variável Nota é maior ou igual a 7. Caso seja igual ou maior que 7, a mensagem aprovado é apresentada. Caso não atenda a essa condição, a mensagem reprovado é apresentada.

3.6.1.3 if else if

A estrutura if else if é uma extensão da estrutura if else, ou uma combinação de if e else.

Forma geral de uma estrutura de decisão if else if na linguagem C:

Em português estruturado	Em linguagem C
<pre> se <condição_1> então <conjunto de instruções 1> senão se <condição_2> então <conjunto de instruções 2> senão <conjunto de instruções 3> fim se </pre>	<pre> if (condição_1) { conjunto de instruções 1; } else if (condição_2) { conjunto de instruções 2; } else { conjunto de instruções 3; } </pre>

Em uma estrutura if else if: inicialmente é testada a condição 1, se o resultado desse teste é falso é testada a próxima condição e assim sucessivamente enquanto o teste atual é falso e houverem condições para serem testadas. Se todas as condições testadas possuem resultado falso é executado o conjunto de instruções que está em else. else contém as instruções que serão realizadas se nenhuma das anteriores foi realizada.

Somente um conjunto de instruções será executado: o primeiro que obtiver um resultado verdadeiro do teste lógico ou o else se nenhum obteve resultado verdadeiro. Ressalta-se que a existência de else (o conjunto de testes ser finalizado por um else) não é obrigatória. E pode ser que seja necessário testar todas as condições consideradas.

Exemplo da estrutura if else if para verificar se um número é igual, maior ou menor que 10.

```
#include <stdio.h>

int main(void)
{
    int Num;
    printf("Digite um numero: ");
    scanf("%d",&Num);
    if (Num>10)
    {
        printf("\O número é maior que 10");
    }
    else if (Num==10)
    {
        printf("O número é igual a 10.");
    }
    else
    {
        printf("O número é menor que 10");
    }
}
```

No else é colocada a condição não testada. No caso do exemplo existem somente três possibilidades (maior, menor ou igual) e o else realiza a condição não testada. Se houvesse mais condições não testadas, para todas seriam realizadas as instruções contidas no else.

3.6.2.3 Estruturas condicionais encadeadas

As estruturas condicionais encadeadas permitem verificar condições sucessivas, em que um determinado conjunto de instrução será executado se a respectiva condição for satisfeita. Juntamente com o conjunto de instruções executadas podem ser feitas novas verificações de condições. Este tipo de estrutura poderá possuir diversos níveis de condições, sendo chamadas de aninhamentos ou encadeamentos.

O if aninhado é simplesmente um if contido da declaração de um outro if.

Exemplo de programa utilizando if aninhados.

Descrição do problema: determinar se um aluno está aprovado caso sua média seja maior ou igual a 7 e em recuperação se a nota for menor que 7. Se o aluno está em recuperação,

ler a nota de recuperação e informar se que ele está aprovado se a nota de recuperação é igual ou maior que 8 e reprovado em caso contrário.

Análise do problema:

Dados de entrada:

Média do aluno.

Processamento:

Verificar se a média é maior que 7.

Se a média é maior ou igual a 7 informar que o aluno está aprovado.

Se a média não é maior que 7 solicitar nota de recuperação.

Verificar se a nota de recuperação é maior que 8.

Se a nota de recuperação é igual ou maior que 8 informar que o aluno está aprovado após recuperação.

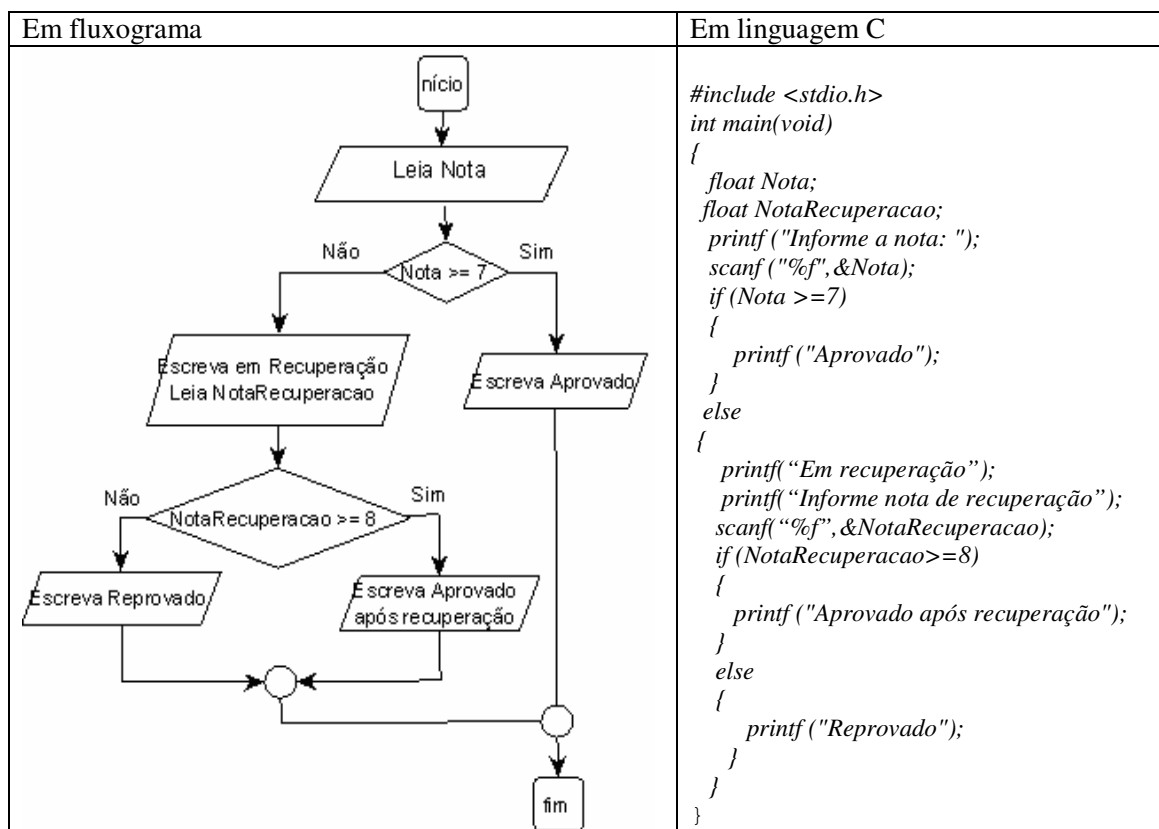
Se a nota de recuperação é menor que 8 informar que o aluno está reprovado após recuperação.

Saída:

Informar que o aluno está aprovado ou reprovado de acordo com a sua média ou nota de recuperação.

A Figura 27 contém o fluxograma e o código em linguagem C do algoritmo para indicar se um aluno está aprovado ou reprovado em decorrência da sua média ou nota de recuperação.

Figura 27 – Fluxograma e programa C para indicar se aluno está aprovado ou reprovado



Quando existem vários if encadeados, ocorre um aninhamento, que se refere ao fato de existir uma seleção dentro de outra. Para que o aninhamento esteja correto é necessário que

uma construção interna esteja completamente inserida na construção imediatamente mais externa. A Figura 28 apresenta um modelo de indentação quando estruturas *if* estão aninhadas. Esse padrão de indentação pode ser utilizado para qualquer estrutura de um programa em linguagem C. Em comandos de repetição, por exemplo, os indicadores de início e de fim da repetição (as chaves que delimitam o bloco de código correspondente à instrução) determinam a indentação de um bloco de instruções.

Figura 28 – Modelo de indentação de estruturas *if else*

```

if (condição_1)
{
    if (condição_2)
    {
        if (condição_3)
        {
        }
        else if (condição_4)
        {
        }
    }
    else if (condição_5)
    {
        if (condição_6)
        {
        }
    }
    else if (condição_7)
    {
    }
    else
    {
    }
}
else
{
}

```

3.6.2.4 switch case

A estrutura de decisão *switch case*, também definida como caso seletivo, é utilizada para testar o valor de uma variável. Se esse valor é verdadeiro todo o código subsequente é executado (todos os casos subsequentes), a menos que exista uma instrução *break*. Nesse caso, as instruções do respectivo *case* são executadas e o fluxo do programa segue após a chave de fechamento da estrutura *switch case*.

O *switch case* é indicado para testar uma variável em relação a diversos valores pré-estabelecidos que ela pode assumir. Sua forma geral é:

Em português estruturado	Em linguagem C
--------------------------	----------------

escolha (variável) caso resposta 1: <conjunto de instruções 1>; caso resposta 2: <conjunto de instruções 2>; caso resposta n: <conjunto de instruções n>; senão <conjunto de instruções n>; fim escolha	switch (variável) { case constante_1: conjunto de instruções 1; break; case constante_2: conjunto de instruções 2; break; case constante_n: conjunto de instruções 3; break; default: conjunto de instruções default; }
--	---

Onde:

variável é a variável cujo conteúdo será avaliado. De acordo com o valor contido nessa variável será escolhido um dos case. Essa variável deve ser do tipo inteiro ou caractere.

case contém os possíveis conteúdos para a variável sendo testada. Esse conteúdo determina qual conjunto de instruções será realizado.

constante contém os valores esperados do resultado da condição (case), ou seja, contidos na variável sendo testada.

default (padrão) é executado se o conteúdo da variável sendo testada não corresponder a nenhum dos valores (casos) estabelecidos. Os case são avaliados na ordem sequencial. Se nenhuma das respostas corresponde ao resultado da expressão, o conjunto de instruções contido em default será realizado. A existência de um default é opcional. Se não existir e o resultado do teste não corresponder a nenhuma constante em case, nenhuma instrução será realizada.

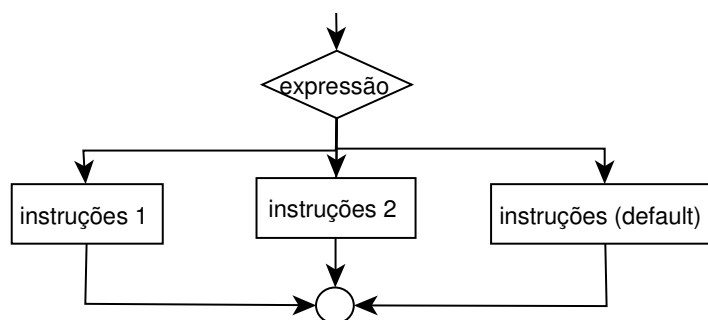
conjunto de instruções indica as instruções que serão realizadas se é verdadeiro o resultado do teste corresponder à constante do respectivo case. Se o valor da variável não corresponder a nenhuma constante definida será executado o conjunto de instruções em default.

break faz com que o switch seja interrompido assim que o respectivo conjunto de instruções é executado. Mas ele não é essencial ao comando switch. Se após a execução da declaração não houver um break, o programa continuará executando as instruções dos cases posteriores, até encontrar um break ou o final da estrutura switch.

A diferença fundamental entre switch e if else if é que a estrutura switch não aceita expressões, apenas constantes. O switch testa a variável e executa a declaração cujo case corresponda ao valor atual da variável. A declaração default é opcional e será executada apenas se o conteúdo da variável que está sendo testada não for igual a nenhuma das constantes.

A Figura 29 apresenta o fluxograma da estrutura de decisão switch case

Figura 29 - Fluxograma da estrutura de decisão switch case



Exemplo de algoritmo utilizando a estrutura de decisão switch case

Descrição do problema:

Escrever um número por extenso.

Análise do problema:

Para reduzir a extensão, o algoritmo considerará apenas os números 0 e 1. Para os outros números informará que se trata de outro número.

Dados de entrada:

Um número.

Processamento:

Se o número é 0, escrever “zero”.

Se o número é 1, escrever “um”.

Se é outro número, escrever “outro número”.

Saída:

Escrever o número por extenso.

A Figura 30 contém o algoritmo para escrever um número por extenso utilizando switch case representado em fluxograma e em linguagem C.

Figura 30 – Fluxograma e programa em linguagem C para números por extenso.

Em fluxograma	Em linguagem C
<pre> graph TD Inicio([início]) --> LeiaNum[/Leia Num/] LeiaNum --> Num0{Num = 0} Num0 -- Sim --> EscrevaZero[Escreva zero] Num0 -- Não --> Num1{Num = 1} Num1 -- Sim --> EscrevaUm[Escreva um] Num1 -- Não --> EscrevaOutro[Escreva outro número] EscrevaZero --> Conexao(()) EscrevaUm --> Conexao EscrevaOutro --> Conexao Conexao --> Fim([fim]) </pre>	<pre> #include <stdio.h> int main(void) { int Num; printf("Informe um número: "); scanf("%d", &Num); switch (Num) { case 0: printf("zero."); break; case 1: printf("um."); break; default: printf("outro número."); } } </pre>

No exemplo da Figura 30, se o conteúdo da variável Num é 0 ou 1, esse conteúdo será escrito por extenso. Caso contrário será escrito “outro número”.

3.6.3 Estruturas de controle repetição

Uma estrutura ou comando de repetição é utilizada para que um determinado conjunto de instruções seja executado um número definido de vezes, enquanto uma condição lógica permanece verdadeira ou até que uma condição lógica seja alcançada.

A linguagem C possui três estruturas de repetição: **for**, **while** e **do while**. De maneira geral, excetuadas particularidades, as três estruturas podem ser aplicadas indistintamente. Contudo, como forma de elucidar o uso de cada uma dessas estruturas, define-se que:

a) **for** é utilizado quando se sabe previamente a quantidade de vezes que um conjunto de instruções será repetido. Por exemplo, ler notas de 100 alunos ou mostrar a tabuada (de 0 a 10) de um número.

b) **while** é utilizado quando a condição de parada de repetição das instruções é resultante da execução dessas instruções. Enquanto a condição permanece verdadeira o conjunto de instruções é realizado. Por exemplo, ler notas de alunos até que seja informado um valor negativo para nota ou ler números até que seja informado o valor zero.

c) **do ... while** é indicado quando é necessário que pelo menos uma vez o conjunto de instruções seja realizado. Enquanto a condição de saída não é alcançada o programa permanece em execução. Por exemplo, ler notas de alunos enquanto não for informado um valor negativo para nota.

Com o **for** e o **while** é possível que a condição não seja verdadeira na primeira execução e assim as instruções não serão realizadas nenhuma vez. Com o **do ... while** isso não ocorre porque o teste é realizado após todas as instruções pertencentes a essa estrutura serem realizadas.

3.6.3.1 for

A estrutura de repetição **for** (para ... até ... faça) é usada para repetir um conjunto de instruções um determinado número de vezes e normalmente se sabe previamente quantas vezes esse conjunto será repetido. Essa é a aplicação mais típica da estrutura de repetição, mas não é a única. As estruturas de repetição **do while** e **while**, também podem ser utilizadas quando se conhece previamente o número de repetições a serem realizadas.

A forma geral da estrutura de repetição **for** é:

Em português estruturado	Em linguagem C
<i>para <inicialização da variável de controle></i> <i>até <verificação da condição de repetição> faça</i> <conjunto de instruções a ser repetido> <i>fim faça</i>	<i>for (inicialização;condição;incremento/ decremento)</i> { //conjunto de instruções; }

Onde:

inicialização da variável de controle define o valor inicial da variável que irá determinar a quantidade de vezes que o conjunto de instruções pertencentes ao **for** será repetido.

condição é a verificação da condição de repetição por meio de um teste lógico para determinar se o conjunto de instruções será novamente repetido ou se a estrutura de repetição será finalizada. A verificação da condição não necessariamente precisa ser um teste lógico

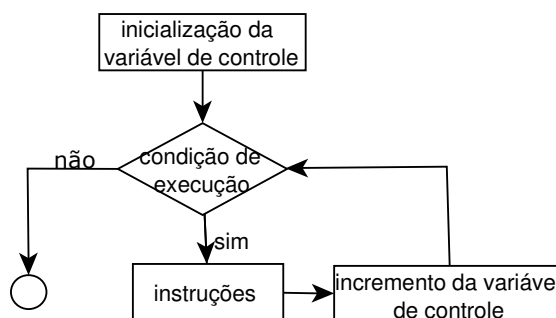
relacionado ao número de repetições, ela pode estar associada a uma outra condição, com, por exemplo, a variável de controle ser positiva. Nesse caso, essa variável precisa ser alterada por instruções que pertencem ao for.

incremento/decremento define o incremento ou decremento da variável de controle. O objetivo é permitir que o número de iterações pretendido seja alcançado pelo incremento ou decremento da variável de controle. O incremento define um acréscimo positivo na variável de controle e o decremento um decréscimo (acréscimo negativo) nessa variável. O incremento pode ser de qualquer valor, incluindo operações de multiplicação e divisão, além de adição e subtração e a combinação de operações. E mais de uma variável pode ser incrementada, como, por exemplo: $X = X + X * 2$, $Y = Y - Y / 3$. Nesse caso, elas são separadas por vírgula. O mesmo ocorre com o incremento e com a condição. Contudo, é indispensável garantir que a condição de finalização seja alcançada para que o programa não entre em um processo infinito de execução.

conjunto de instruções define as instruções que serão realizadas enquanto a condição for estabelecida, ou seja, é verdadeira.

O for executa a inicialização apenas na primeira iteração das suas instruções e testa a condição em todas as iterações posteriores. Quando a condição se tornar falsa é executada a próxima instrução após o conjunto de instruções do for. Se a condição é verdadeira, o conjunto de instruções e o incremento ou decremento são realizados e a condição é testada novamente. Essas operações são repetidas até que a condição se torne falsa. A Figura 35 apresenta o fluxograma da estrutura de repetição for.

Figura 35 - Fluxograma da estrutura de repetição for



Uma estrutura de repetição for possui uma variável de controle. Essa variável é inicializada na primeira iteração dessa estrutura e é definida uma condição que determina quando a repetição para. A cada iteração essa variável é incrementada ou decrementada e é verificado se a condição de parada é alcançada.

A variável de controle funciona como um contador da quantidade de repetições que a estrutura executará. O incremento ou decremento na variável de controle é realizado a cada repetição da estrutura até que seja alcançada a condição de finalização.

Exemplo de uso da estrutura for para apresentar a tabuada de um número.

Descrição do problema:

Apresentar a tabuada de um número informado pelo usuário.

Análise do problema:

Dados de entrada:

Um número.

Processamento:

Repetir o cálculo e mostrar a saída do número informado entre por 0 e 10.

Saída:

A tabuada do número a cada iteração da estrutura de repetição.

Na Figura 36, o algoritmo para imprimir a tabuada de determinado número está representado em fluxograma e em linguagem C.

Figura 36 - Algoritmo para imprimir a tabuada de determinado número

Em fluxograma	Em linguagem C
<pre> graph TD inicio([início]) --> LeiaNum[/Leia Num/] LeiaNum --> ParaCont[Para Cont <- 0 até 10 faça] ParaCont -- Sim --> Escreva[Escreva Num * Cont] Escreva --> ParaCont ParaCont -- Não --> fim([fim]) </pre>	<pre> #include <stdio.h> int main(void) { int Num; int Cont; printf("Informe um número: ") scanf("%d", &Num); for(Cont=0; Cont<=10; Cont++) { printf("%d * %d = %d\n", Num, Cont, Num * Cont); } } </pre>

Nesse exemplo, já está previamente definido que a instrução Escreva Num * Cont será realizada onze vezes (de zero até dez). A variável Cont que inicia com 0 é incrementada em cada iteração. A própria estrutura for controla a finalização das repetições.

Cont++ é equivalente a Cont = Cont + 1 que determina o incremento de um a cada vez que o conjunto de instruções da estrutura for é executado.

Explicação figurativa da estrutura for:

```

for(Cont=0;Cont<=10;Cont++)
{
    printf ("%d * %d = %d\n",Num, Cont, Num * Cont);
}

```

a) A variável Cont é inicializada com zero (poderia ser qualquer outro valor determinado pelo contexto do programa). A inicialização ocorre somente na primeira iteração da estrutura for (na primeira execução dessa estrutura).

b) Em seguida e em todas as iterações posteriores o teste Cont <=10 é realizado.

b.1) Se o resultado desse teste é verdadeiro, o conjunto de instruções pertencente ao for é realizado, após isso a variável Cont é incrementada. O ciclo testar a condição (se resultado verdadeiro), realizar as instruções e incrementar a variável de controle prossegue enquanto o resultado da condição de teste é verdadeiro.

b.2) Se o resultado do teste é falso, a execução do programa prossegue após o final da estrutura for. O final da estrutura é determinado pelas chaves que delimitam as instruções que pertencem a essa estrutura.

O for é bastante flexível. Tanto em termos da forma de definição da inicialização, controle, incremento/decremento, como da própria existência dessas partes. Qualquer uma

dessas partes pode ser qualquer expressão válida da linguagem C. As seguintes formas de uso da estrutura for são válidas:

```
for ( Cont=1; Cont < 100; Cont++)
{
...
}

for (Cont=1; Cont < Quantidade; Cont++)
{
...
}

for (Cont=1; Cont < Quantidade + Numero / 10; Cont++)
{
...
}

for (Cont=100; Cont > Quantidade + 10; Cont--)
{
...
}

for (Cont=1; Cont < ContarQuantidade( ); Cont=Cont + 2)
{
...
}
```

No penúltimo exemplo ocorre decremento da variável Cont, com Cont--. Essa variável inicia com 100 e é decrementada de um até alcançar o valor determinado pela variável Quantidade + 10.

No último exemplo, o incremento está sendo feito de dois em dois, Cont = Cont + 2. Além disso, no teste está sendo utilizada uma função ContarQuantidade() que retorna um valor que é comparado com Cont. A instrução Cont+=2 é equivalente a Cont = Cont + 2;

Em relação à existência das partes, a inicialização pode ser realizada fora da estrutura de repetição. E a seguinte forma de uso do for é válida:

```
Cont = 1;
for ( ; Cont < ContarQuantidade( ); Cont+=2)
{
...
}
```

O controle pode ser realizado nas instruções pertencente a estrutura for. Por exemplo:

```
for (Cont=1; ; Cont+=2)
{
....
if (Cont < ContarQuantidade( ))
{
break; //a instrução break determina a saída do loop do qual ela pertence
}
....
}
```

O incremento/decremento também pode ser realizado pelas instruções que compõem a estrutura for. Por exemplo:


```

for (Cont=1; Cont < ContarQuantidade( );)
{
    ....
    Cont=Cont + 2;
    ....
}

```

E essas formas de inicialização, controle, incremento/decremento podem ser combinadas.

Com a estrutura for é possível definir um loop infinito ou execução contínua. O loop infinito pode ter a seguinte forma:

```

for (Cont=0; ;Cont++)
{
    ;
}

for ( ; ;)
{
    ;
}

```

Um loop é denominado infinito porque será executado para sempre (não existindo a condição de finalização, ela será sempre considerada verdadeira), a não ser que ele seja interrompido por meio de instruções no seu corpo. Para que a execução dessa estrutura for possa parar em algum momento é necessário que as suas instruções contenham a condição de saída. É possível que uma instrução break possa ser necessária, isto porque a condição está vazia em ambos os casos.

No segundo exemplo, for(; ;), a declaração do for está vazia e no primeiro somente a condição está vazia, ou seja, em ambos não é realizado o teste para verificar se as instruções do corpo do for serão novamente executadas. A condição determina se as repetições ocorrerão novamente. Contudo, é indispensável garantir que essa condição seja alcançada, com o incremento ou decremento, por exemplo.

Exemplo de programa sem conteúdo para gerar tempo de espera:

```
#include <stdio.h>
```

```

int main(void)
{
    long int Cont;
    printf("início");
    for (Cont =0; Cont <10000000; Cont ++ )
    {
        ; /* Espera 10 milhões de iterações */
    }
    printf("fim das 10 000 000 de iterações do for");
}

```

Para exemplificar de uso desse tipo de loop está um programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuário pressione uma tecla sinalizadora de final. Nesse exemplo é a letra 'X'.

```

#include <stdio.h>

int main(void)
{
    int Cont;
    char Letra;
    printf(" Digite uma letra - <X para sair> ");
    for (Cont=1;;Cont++)
    {
        scanf("%c", &Letra);
        if (Letra == 'X')
        {
            break;
        }
        printf("\nLetra: %c\n",Letra);
        fflush(stdin); //para Windows
        //__fpurge(stdin); //para ambientes Unix
        // scanf("%c", &Letra); //para apanhar o enter
    }
}

```

Nesse exemplo, a variável Letra é do tipo char e sendo assim ela armazena apenas um único caractere, que pode ser uma letra, um número ou um símbolo, como o que representa a tecla enter. Ao informar um caractere o usuário pressiona a tecla do caractere e a tecla enter. Desta forma serão dois caracteres informados e ficam no buffer do teclado nessa ordem. O primeiro caractere é armazenado na variável Letra e o segundo (o enter) é armazenado na variável Letra na próxima iteração do for. Há algumas maneiras de resolver isso. Uma delas é colocando um segundo scanf() para apanhar esse enter ou utilizando funções prontas como o fflush(stdin) no ambiente Windows ou __fpurge(stdin) no ambiente Linux. Essas funções têm o objetivo de limpar o buffer do teclado.

3.6.3.2 while

A estrutura de repetição while (enquanto ... faça) é utilizada quando um conjunto de instruções deve ser executado repetidamente, enquanto uma determinada condição (expressão lógica) permanecer verdadeira. O número de repetições não é previamente conhecido. A finalização de execução ocorrerá em decorrência de condições obtidas durante a própria execução do programa.

A estrutura de repetição while tem a seguinte forma geral na linguagem C:

Em português estruturado	Em linguagem C
enquanto <condição> faça <conjunto de instruções> fim enquanto	while (condição) { //conjunto de instruções; }

Onde:

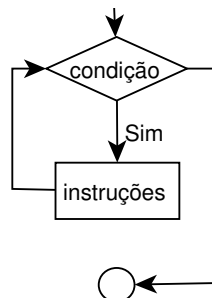
condição determina um teste lógico que define se o conjunto de instruções pertencente à

estrutura while será ou não executado.

conjunto de instruções define as instruções que serão realizadas enquanto a condição for estabelecida. Dependendo do resultado do teste da condição, o conjunto de instruções poderá não ser executado nenhuma vez (se for falsa no primeiro teste), ser executado várias vezes (enquanto for verdadeira) ou ser executado infinitamente se a condição nunca for alterada para falsa.

A Figura 31 apresenta o fluxograma da estrutura de repetição while.

Figura 31 - Fluxograma da estrutura de repetição while



A Figura 31 mostra que a estrutura while testa a condição. Se esta for verdadeira o conjunto de instruções é executado e o teste é feito novamente, e assim sucessivamente. É possível fazer um loop infinito colocando uma expressão que é sempre verdadeira na condição.

O programa a seguir mostra a tabuada de um número informado pelo usuário. É um exemplo de uso da estrutura while.

Descrição do problema:

Apresentar a tabuada de um número informado pelo usuário.

Análise do problema:

Dados de entrada:

Um número

Processamento:

Repetir o cálculo e mostrar a saída do número informado multiplicado por 0 até 10.

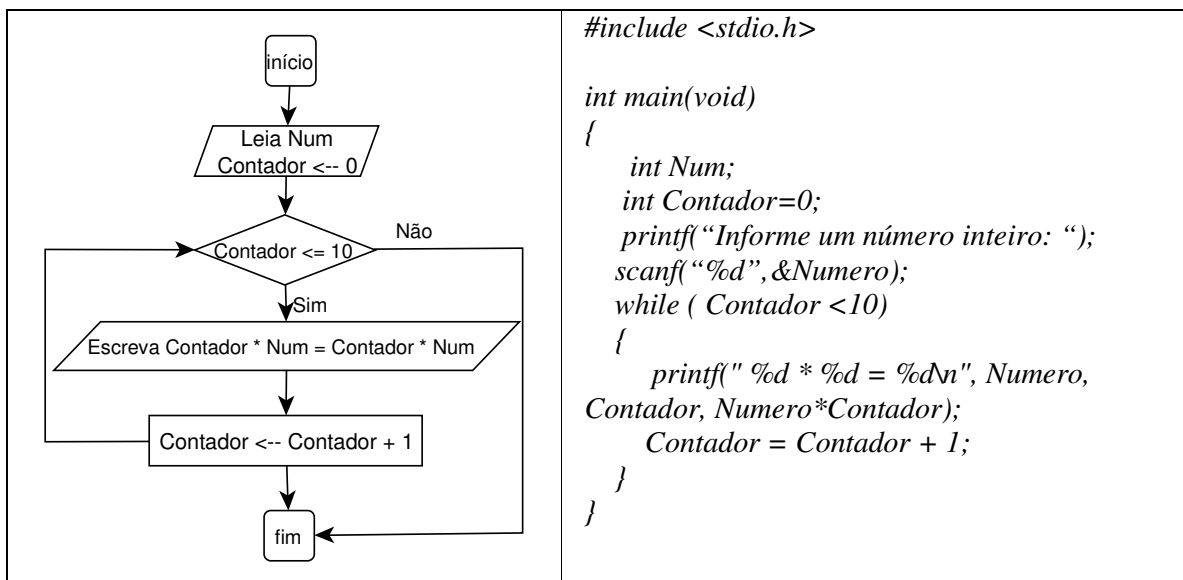
Saída:

A tabuada do número.

Na Figura 32, o algoritmo para imprimir a tabuada de um número está representado em fluxograma e em linguagem C.

Figura 32 - Programa para imprimir a tabuada de um número.

Em fluxograma	Em linguagem C
---------------	----------------



Nesse programa, $\text{Contador} < 10$ testa se o valor da variável *Contador* é menor que 10. Se for as instruções subordinadas ao *while*, delimitadas pelas chaves, serão realizadas. A instrução `printf(" %d * %d = %d\n",Num,Contador,Num*Contador);` imprime uma linha da tabuada na tela e coloca o cursor no início da próxima linha. A instrução `Contador = Contador + 1;` incrementa 1 ao valor armazenado na variável *Contador* a cada vez que o conjunto de instruções do *while* é executado. Enquanto essa condição permanecer verdadeira, a tabuada do número é mostrada. Quando o teste condicional não for mais verdadeiro, quando *Contador* for igual ou maior do que 10, o programa é finalizado. Nesse programa, a variável *Contador* precisa ser inicializada porque não se sabe que valor está armazenado no endereço de memória reservado para ela e são realizados incrementos sucessivos ao conteúdo dessa variável a partir do seu valor inicial.

Na própria instrução `printf()` é feito o cálculo da tabuada ($\text{Numero} * \text{Contador}$). Em uma instrução `printf()`, o caractere de formatação `%d` pode ser substituído por uma constante, uma variável, uma expressão lógica ou matemática ou o retorno de uma função.

Outro exemplo de uso do *while*. Esse programa é executado enquanto *Contador* for menor que 100.

```
#include <stdio.h>
```

```

int main(void)
{
    int Contador = 0;
    while (Contador < 100)
    {
        printf(" %d ", Contador);
        Contador ++;
    }
}
  
```

O programa a seguir espera o usuário digitar a letra n minúscula ou maiúscula para finalizar:

```
#include <stdio.h>
```

```

int main(void)
  
```

```

{
    char Opcao;
    Opcao = 'a';
    while ((Opcao != 'N') || (Opcao != 'n'))
    {
        scanf("%c", &Opcao);
    }
}

```

Com a estrutura while o teste é realizado no início, desta forma é necessário iniciar a variável Opcao de maneira que a estrutura seja acessada, ou seja, que a condição do while seja verdadeira. Nesse programa, o programa permanecerá em execução enquanto um caractere diferente de 'N' ou 'n' for informado.

3.6.3.3 do while

Na estrutura de repetição do while (faça ... enquanto), primeiro são executadas as instruções e somente depois é realizado o teste da condição. Se o resultado do teste da condição é verdadeiro, as instruções são executadas novamente, caso seja falso é encerrada a repetição das instruções. Essa estrutura possui indicação de uso quando é necessário que o conjunto de instruções seja executado pelo menos uma vez, independentemente da condição. Isto significa que mesmo que a condição seja inicialmente falsa o conjunto de instruções será realizado pelo menos uma vez.

A estrutura de repetição do while possui a seguinte forma geral:

Em fluxograma	Em linguagem C
faça <conjunto de instruções> enquanto <condição>	do { conjunto de instruções; } while (condição);

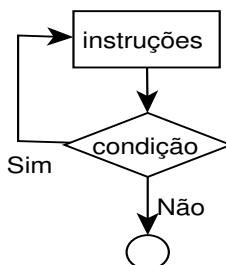
Onde:

condição determina um teste lógico. O ponto e vírgula final é obrigatório.

conjunto de instruções define as instruções que serão realizadas enquanto a condição for estabelecida. Independentemente do resultado do primeiro teste da condição, a conjunto de instruções será realizado pelo menos uma vez. Contudo, as execuções subsequentes somente serão realizadas enquanto a condição for verdadeira.

A Figura 33 apresenta o fluxograma da estrutura de repetição do while.

Figura 33 - Fluxograma da estrutura de repetição do while



Como mostra a representação da Figura 33, inicialmente são realizadas as instruções que fazem parte do corpo da estrutura do while, depois é realizado o teste. Se o resultado do teste for verdadeiro o processo se repete. Se for falso, é executada a próxima instrução após o final do do while.

Exemplo de algoritmo utilizando estrutura de repetição do while para obter (ler) um número positivo.

Descrição do problema:

Ler números até que seja informado um número positivo.

Análise do problema:

Dados de entrada:

Um número

Processamento:

Ler um número.

Verificar se o número é negativo.

Se o número é negativo repetir a leitura.

Se o número é positivo finalizar a repetição de leitura e informar que o número é positivo.

Saída:

Informar que o número é positivo.

Na Figura 34, o algoritmo para ler números até que seja informado um número positivo está representado em fluxograma e em linguagem C.

Figura 34 – Programa para ler números até que seja informado um número positivo.

Em fluxograma	Em linguagem C
<pre> graph TD inicio([início]) --> LeiaNum[/Leia Num/] LeiaNum --> Num0{Num < 0} Num0 -- Sim --> LeiaNum Num0 -- Não --> Escreva[/Escreva Num é positivo/] Escreva --> fim([fim]) </pre>	<pre> #include <stdio.h> int main(void) { int Num; do { printf("Informe um número: "); scanf("%d",&Num); } while (Num < 0); printf("O número %d é positivo",Num); } </pre>

O programa da Figura 34 permanece em execução até que um número positivo seja informado. Nesse programa independentemente do valor armazenado em Num, pelo menos uma vez as instruções pertencentes ao do while serão executadas. Esse programa é útil, por exemplo, para validar uma entrada, no sentido de obter um número positivo. A execução permanece até que essa condição seja satisfeita.

A seguir um exemplo de validação de entrada.

```
#include <stdio.h>

int main(void)
{
    int Opcao;
    do
    {
        printf("\nEscolha a fruta pelo numero:\n\n");
        printf("(1)...Uva\n");
        printf("(2)...Pêssego\n");
        printf("(3)...Ameixa\n\n");
        scanf("%d", &Opcao);
    } while ((Opcao<1) || (Opcao>3));
}
```

3.6.3.4 O comando break

O comando break pode ser utilizado para finalizar a execução de um comando (como o switch) ou interromper a execução de qualquer estrutura de repetição (como for, while ou do while). O break faz com que a execução do programa continue na primeira linha seguinte ao bloco que está sendo interrompido. Isso significa que em estruturas de repetição aninhadas o break causará saída para a estrutura imediatamente mais externa em que o break está.

Por exemplo:

```
#include <stdio.h>

int main(void)
{
    int Cont1;
    char Cont2;
    for(Cont1=0; Cont1<10; Cont1++)
    {
        Cont2=97;
        for(;;)
        {
            printf("%c ", Cont2);
            Cont2++;
            if(Cont2==123)
            {
                break;
            }
        }
        printf("\n");
    }
}
```

Esse código imprimirá o alfabeto dez vezes na tela. Toda vez que o `break` é encontrado, o controle é devolvido para o laço `for` externo. É possível utilizar `Cont=97` para atribuir o caractere 'a' para `Cont2`. E realizar comparação `Cont2 == 123` para '{' (que é o próximo caractere após z) porque cada caractere possui um código numérico correspondente.

3.6.3.5 O comando *continue*

O comando *continue* tem o objetivo de interromper a iteração atual de uma estrutura de repetição. Quando o comando *continue* é encontrado, a execução reinicia na próxima iteração da estrutura de repetição.

O programa a seguir exemplifica o uso do *continue*:

```
#include <stdio.h>
```

```
int main(void)
{
    int Opcao = 1;
    while (Opcao != 5)
    {
        printf("\n\n Escolha uma opção entre 1 e 5: ");
        scanf("%d", &Opcao);
        if ((Opcao > 5) || (Opcao < 1))
        {
            continue; /* volta ao início do loop, na próxima iteração*/
        }
    }
}
```

Nesse programa é recebida uma opção do usuário. Se esta opção for inválida, o *continue* faz com que o fluxo seja desviado de volta ao início do loop. Caso a opção escolhida seja válida o programa segue normalmente. A variável `Opcao` foi inicializada por 1 para garantir que o teste em `while (Opcao != 5)` seja verdadeiro e assim as instruções contidas no `while` sejam executadas pelo menos uma vez.

3.7 Estruturas de dados homogêneas

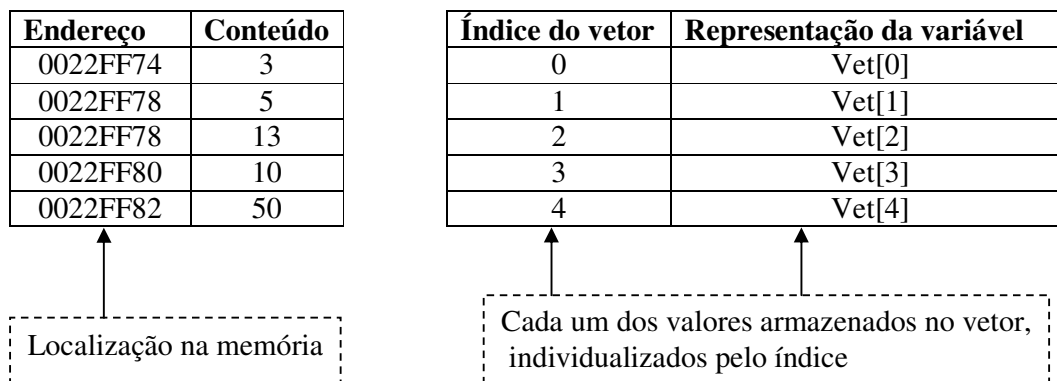
As estruturas de dados homogêneas definem um conjunto de dados do mesmo tipo. Esse conjunto é tratado como uma variável, mas é possível acessar individualmente cada um dos dados desse conjunto. O nome da variável que define esse conjunto de dados é acompanhado de um índice que individualiza ou identifica cada um dos elementos desse conjunto. O índice indica a posição de armazenamento de cada valor e é definida em relação ao primeiro endereço reservado para a variável.

Variáveis indexadas de um só índice são conhecidas como vetores. Se elas tiverem mais de um índice são denominadas matrizes. Contudo, ressalta-se que essa é apenas uma maneira de denominação. Também é encontrada a denominação vetores ou matrizes uni e multidimensionais. Neste texto é utilizada a denominação vetor para variável indexada de uma dimensão e matriz para variável indexada de mais de uma dimensão, independentemente do tipo de dado que armazenam.

A Figura 37 traz uma representação esquemática de um vetor na memória. A declaração para esse vetor é `int Vet[5]`. É um vetor para armazenar cinco números inteiros. É

um vetor de inteiros com tamanho cinco. Pela representação dessa figura, a variável Vet no índice 0 possui armazenado o valor 3 e Vet no índice 1 possui armazenado o valor 5.

Figura 37 – Representação esquemática de um vetor na memória



A Figura 37 apresenta uma definição ilustrativa de endereços. Contudo, essa representação significa que um inteiro ocupa dois bytes em memória. Isso justifica o primeiro endereço ser 0022FF74 e o segundo 0022FF76, dois bytes adiante.

Por definição, o índice de um vetor inicia em zero, assim, os índices abrangem de zero até o tamanho do vetor – 1.
O conteúdo, o valor armazenado no endereço de memória, é identificado pelo nome da variável. O índice é a posição do referido valor em relação ao endereço inicial da variável, o índice zero.

3.7.1 Vetores

Os vetores podem armazenar dados numéricos (int, float, double) ou caracteres (char) que são os dados simples da linguagem C.

A forma geral para declaração de um vetor é:

tipo_da_variável nome_da_variável [tamanho];

Quando uma declaração de um vetor é feita, a linguagem C reserva espaço na memória suficiente para armazenar a quantidade de dados definida como tipo da variável. Por exemplo, na declaração:

float VetorA [20];

a linguagem C reservará espaço para armazenar 20 valores do tipo float. São reservados endereços contíguos de memória para armazenar esses valores. Isso permite que os valores armazenados em um vetor possam ser acessados em sequência, como, por exemplo, em iterações de uma estrutura de repetição.

Na linguagem C, a numeração do índice de um vetor inicia em zero. Isso significa que na instrução *float VetorA [20]*, os dados serão indexados de 0 a 19. Para acessá-los é necessário indicar o nome do vetor e o índice:

VetorA [0]

VetorA [1]

.

.
.
VetorA [19]
Mas é possível fazer:

VetorA [30]

VetorA [103]

Isso é possível porque a linguagem C não verifica se o índice usado está dentro dos limites válidos.

Quando são utilizados índices além do espaço reservado para o vetor, pode estar sendo utilizado espaço reservado para outras variáveis cujos valores são sobrescritos ou espaço utilizado por outros programas, podendo ocasionar problemas como a parada do sistema operacional, por exemplo.

Exemplo de vetores:

#include <stdio.h>

```
int main(void)
{
    int Num[100]; /* Declara um vetor para armazenar 100 valores inteiros ou um
vetor com 100 posições para armazenar valores inteiros, com os índices de 0 a 99*/
    int Cont;
    int TotalNums=0
    do
    {
        printf ("Informe um número (0 para finalizar a entrada): ");
        scanf ("%d",&Num[Cont]);
        TotalNum++;
    } while (Num[Cont-1]!=0 && TotalNums<=100);
    TotalNums=TotalNums-1;
    printf ("\n Os números digitados foram:\n");
    for (Cont=0;Cont<TotalNums;Cont++)
    {
        printf (" %d\t",Num[Cont]);
    }
}
```

Nesse exemplo, a variável TotalNums é inicializada com 0. O programa repete a solicitação de entrada de números até que seja informado o valor 0, que é condição de saída, ou que sejam lidos 100 números, que é o tamanho do vetor. À medida que os números são lidos eles são armazenados no vetor e o contador do vetor é incrementado. Quando o usuário digitar o valor 0 ou a variável TotalNums chegar ao valor 100 a estrutura do while é finalizada. Em seguida, a estrutura de repetição for é realizada, sendo percorridos os índices do vetor (Cont=0; Cont< TotalNums; Cont++) e mostrado o seu conteúdo.

Na linguagem C uma string é um vetor de caracteres finalizado com o caractere nulo. O conjunto de espaços ou endereços de memória reservado é do tamanho definido para o vetor na sua declaração que deve considerar o espaço necessário de um caractere para o terminador nulo. O terminador nulo pode ser escrito pela string '\0'.

A declaração geral para uma string é:

```
char nome_da_string [tamanho];
```

Essa instrução declara um vetor de caracteres com o número de posições igual a tamanho. Como é necessário reservar um caractere para ser o terminador nulo que indica o final da string, é preciso declarar o comprimento da string com um caractere a mais do que a maior string que se pretende armazenar nessa variável.

Para ler uma string fornecida pelo usuário pode ser utilizada a função `gets()`. Essa função é mais adequada que `scanf()` para ler strings. A função `scanf()` finaliza a leitura no primeiro caractere de espaço informado pelo usuário.

Exemplo de uso da função `gets()`. A função `gets()` coloca o terminador `\0` na string.

```
#include <stdio.h>
int main(void)
{
    char Texto[101];
    printf("Digite uma string de até 100 caracteres: ");
    gets (Texto);
    printf ("Texto: %s",Texto);/*%s indica que uma string será colocada na tela*/
}
```

Nesse exemplo, o tamanho máximo da string é 100 caracteres porque um dos caracteres, o último no conteúdo da string, será utilizado para armazenar `\0`. Contudo, é possível informar mais de 100 caracteres. A linguagem C não faz verificação automática de limites de variáveis do tipo vetor. E isso pode ocasionar problemas porque estará sendo utilizado espaço de memória que não foi reservado para a variável.

Como as strings são vetores de caracteres, para acessar um determinado caractere de uma string basta indicar o nome da string e o seu índice. Da mesma forma que ocorre com vetores numéricos. Na string *Texto* é possível acessar o seu terceiro caractere, por exemplo, da seguinte forma:

```
Texto[2] = 'a';
printf("Segundo caractere de %s é %c",Texto,Texto[1]);
```

%s imprime, mostra na tela, uma string.

%c imprime, mostra na tela, um caractere.

Strings são vetores e assim o índice também inicia em zero. O primeiro caractere da string sempre estará na posição 0, o segundo na posição 1 e assim sucessivamente.

O programa a seguir imprimirá a segunda letra da string "Linguagem C". Em seguida, essa letra será alterada para 'A' e a string é mostrada novamente.

```
#include <stdio.h>

int main(void)
{
    char Texto[15] = "Linguagem C";
    printf("\nString: %s", Texto);
    printf("\nSegunda letra da string: %c", Texto[1]);
    str[1] = 'A';
    printf("\nString resultante: %s", Texto);
}
```

Na string Texto, o terminador nulo está na posição 12. Das posições 0 a 11 há caracteres válidos para o contexto do programa (“Linguagem C” possui 11 caracteres, considerando o espaço) e, portanto, podem ser escritos. Na declaração de uma string pode ser atribuído conteúdo a mesma, como ocorreu com “Linguagem C” que foi atribuída à string Texto. Porém, depois de declarada, uma string não pode mais receber texto por atribuição direta.

Exemplo de programa para ler uma string e contar quantos caracteres são iguais a um determinado caractere informado pelo usuário.

```
#include <stdio.h>

int main(void)
{
    char Texto[100]; /* texto com até 99 caracteres, o último conterá \0 que indica
final da string */
    int Cont=0;
    int Quantidade=0;
    char Caractere;
    printf("Digite uma frase: ");
    gets(Texto); /* Lê a string */
    fflush(stdin);
    printf("Digite um caractere: ");
    scanf("%c",&Caractere); /*Lê o caractere*/
    while (Texto[Cont] != '\0')
    {
        if(Texto[Cont] == Caractere)
        {
            Quantidade++;
        }
        Cont++;
    }
    printf("\nQuantidade de caracteres %c na string %s",Quantidade, Texto);
}
```

Nesse programa uma frase foi lida e armazenada em Texto. Em seguida um caractere foi lido. A função fflush() foi utilizada para limpar o buffer do teclado porque antes de ler o caractere, uma outra informação havia sido lida.

A estrutura de repetição `while (Texto[Cont] != '\0')` tem o objetivo de percorrer a string até o seu final. ‘\0’ indica final de string.

`if(Texto[Cont] == Caractere)` compara se o caractere atual da string sendo percorrida é igual ao caractere informado. Se sim, é incrementado 1 ao conteúdo da variável Quantidade.

`Cont++` incrementa o índice do vetor Texto que é a string sendo percorrida.

Existem outras maneiras de limpar o buffer do teclado. A função fflush() apresenta restrições e é considerada mais adequada para arquivos. Exemplos:

```
char A;

//a sequência %[^\\n] indica que o caractere \\n que representa uma quebra
de linha deve ser desconsiderado.
scanf(" %[^\\n]", &A); //se tiver espaço após o caractere anterior é
necessário haver espaço antes de %

//ler o caractere que representa enter.
```

```

    getchar(); //getchar() lerá o caractere e não o armazenará porque o
    retorno dessa função não está sendo atribuído para alguma variável.
    scanf("%c",&A);

    //desconsiderar o primeiro caractere
    scanf(" %c", &A); //o espaço antes de %c indica que o primeiro
    caractere deve ser desconsiderado.

```

A seguir são apresentadas as funções `strcpy`, `strcat`, `strlen`, `strcmp`. Que são algumas das funções existentes na biblioteca padrão `string.h` para manipular strings.

a) **strcpy** – a função `strcpy()` copia o conteúdo da string de origem para a string de destino. Sua forma geral é:

```
strcpy (string_destino, string_origem);
```

A string de origem pode ser uma variável do tipo vetor de caracteres ou uma string constante. Nesse caso ela é informada entre aspas duplas.

```
strcpy (Endereco,OutroEndereco);
strcpy (Endereco,"Avenida Tupy, 12345");
```

É importante ressaltar que a string de destino (que recebe a cópia) deve ter tamanho suficiente para armazenar a string de origem (que será copiada), incluindo o caractere que indica final de string. Se a string de destino possui informação armazenada a mesma será sobrescrita no processo de cópia.

Exemplo de programa que faz a funcionalidade de `strcpy()`

```

#include <stdio.h>

int main (void)
{
    char Origem[101];
    char Destino[101];
    int Cont=0; /*percorrerá os índices d a string, iniciar em zero para o primeiro
    índice*/

    printf("Informe um texto com até 100 caracteres: ");
    gets(Origem);

    while (Origem[Cont] != '\0') //enquanto não for final de string
    {
        Destino[Cont] = Origem[Cont];
        Cont++;
    }
    Destino[Cont] = '\0';//para finalizar a string de destino
    printf("String copiada: %s",Destino);
}

```

Nesse exemplo, a string de origem é percorrida até o seu final e paralelamente é copiada para a string de destino. Considera-se que a string de origem possui menos de 100 caracteres. Após finalizada a cópia, o caractere `'\0'` é colocado no final da string de destino. Essa instrução é indispensável porque a cópia ocorreu enquanto `Origem[Cont] != '\0'`.

b) **strcat** - a função strcat() concatena a string de origem ao final da string de destino. Sua forma geral é:

```
strcat (string_destino, string_origem);
```

No final da string de origem é acrescentada a string de destino. É importante ressaltar que é necessário declarar a string de destino com tamanho suficiente para conter a string de origem, além dos caracteres que a mesma já possui.

Exemplo de programa que faz a funcionalidade de strcat()

```
# include <stdio.h>
```

```
int main(void)
```

```
{
```

```
    char Origem[101];
```

```
    char Destino[201];
```

```
    int Cont1=0;//para contar a quantidade de caracteres da string de destino
```

```
    int Cont2=0;
```

```
    printf("Informe um texto: ");
```

```
    gets(Origem);
```

```
    printf("Informe outro texto: ");
```

```
    gets(Destino);
```

```
    /*contar quantos caracteres a string de destino possui para iniciar a escrita dos  
    caracteres após o último caractere da string de origem */
```

```
    while (Destino[Cont1] != '\0') //enquanto não for final de string
```

```
    {
```

```
        Cont1++;
```

```
    }
```

```
    while (Origem[Cont2] != '\0') //enquanto não for final de string
```

```
    {
```

```
        Destino[Cont1] = Origem[Cont2];/*a string de origem é armazenada na string  
de destino a partir de Cont1 que contém a quantidade de caracteres da string de  
origem*/
```

```
        Cont1++;//incrementar contador de Destino
```

```
        Cont2++;//incrementar contador de Origem
```

```
    }
```

```
    Destino[Cont1] = '\0';//finalizar a string de destino
```

```
    printf("String concatenada: %s",Destino);
```

```
}
```

c) **strlen** - a função strlen() retorna o tamanho, a quantidade de caracteres, da string fornecida que é passada como parâmetro. Sua forma geral é:

```
int strlen (string);
```

Exemplos:

```
Tamanho = strlen(Destino);
```

```
Tamanho = strlen("Linguagem C");
```

O terminador nulo de uma string não é contado por `strlen()`.

Exemplo de programa que faz a funcionalidade de `strlen()`.

```
# include <stdio.h>
```

```
int main(void)
{
    Char Texto [101];
    int Tamanho=0;
    printf("Informe um texto com até 100 caracteres: ");
    gets(Texto);

    while (Texto[Tamanho] != '\0') //enquanto não for final de string
    {
        Tamanho++;
    }
    printf("A string %s possui %d caracteres",Texto, Tamanho);
}
```

d) **strcmp** - A função `strcmp()` compara duas strings. Sua forma geral é:

```
strcmp (string1,string2);
```

Se as duas strings forem idênticas a função retorna zero. Se a primeira for maior que a segunda é retornado um valor positivo e se a segunda for maior que a primeira é retornado um valor negativo.

Exemplo de programa que verifica se duas strings são idênticas ou diferentes entre si.

```
# include <stdio.h>
```

```
int main(void)
{
    char Texto1[101];
    char Texto2[101];
    int Cont=0; //percorrer a string
    char Condicao = 's';

    printf("Informe um texto com até 100 caracteres: ");
    gets(Texto1);
    printf("Informe outro texto com até 100 caracteres: ");
    gets(Texto2);

    while (Texto1[Cont] != '\0' || Texto2[Cont] != '\0') //até o primeiro que finalizar
    {
        if(Texto1[Cont] != Texto2[Cont])
        {
            Condicao='n';
            break; /*não é mais necessário percorrer a string, elas são diferentes */
        }
    }
```

```

        Cont++;
    }

    if (Condicao == 's')
    {
        printf("textos são iguais.");
    }
    else
    {
        printf("textos são diferentes");
    }
}

```

Para não utilizar break a estrutura while desse programa poderia ser feita da seguinte forma:

```

while ((Texto1[Cont] != '\0' || Texto2[Cont] != '\0') && (Condicao != 'n'))
{
    if (Texto1[Cont] != Texto2[Cont])
    {
        Condicao='n';
    }
    Cont++;
}

```

A comparação de caracteres é realizada pela tabela ASCII, portanto letras maiúsculas são consideradas “menores” que letras minúsculas. Assim, “Linguagem” é menor que “linguagem” porque nessa tabela primeiro estão os caracteres minúsculos. Pela função strcmp() resultaria valor negativo ou positivo dependendo de qual string é informada primeiro. E pelo programa implementado retornaria que os textos são diferentes.

3.7.2 Matrizes

Matrizes são vetores de mais de uma dimensão. Por exemplo, armazenar dados de produtos, por categorias e por setor; armazenar notas por aluno, disciplina, professor, turma e curso; ou, ainda, armazenar nomes de várias pessoas, requer uma estrutura de várias dimensões. Uma matriz pode ter várias dimensões, mas os dados armazenados devem ser do mesmo tipo.

As matrizes que possuem duas dimensões são denominadas bidimensionais. Sua forma geral de declaração é:

tipo_de_dado nome_da_variável [tamanho_dimensao1][tamanho_dimensao2];

Onde:

Tipo_de_dado – indica o tipo de dado (int, float, double, char e seus modificadores) que serão armazenados na matriz.

Nome_da_variável – o identificador, nome, da matriz.

Tamanho_dimensão – indica o tamanho de cada uma das dimensões.

Essa forma de declaração é aplicada às matrizes de string e numéricas. Uma matriz

bidimensional pode ser vista como uma matriz de linhas e colunas, em que a primeira dimensão indica a quantidade de linhas e a segunda a de colunas.

Uma matriz que armazena strings (no sentido de elementos distintos como, por exemplo, o nome de dez alunos) é uma matriz com duas dimensões (bidimensional). Na declaração de matrizes desse tipo a primeira dimensão indica a quantidade de strings (elementos) e a segunda o tamanho máximo dessas strings (o nome de dez alunos, sendo que cada nome pode ter até 80 caracteres). Ressalta-se que nesse tamanho deve ser considerado o caractere de final de string. Para acessar uma string é indicado o nome da matriz e o primeiro índice.

Quando uma matriz é lida ou preenchida na linguagem C, o índice mais à direita varia mais rapidamente que o índice mais à esquerda. Todos os índices iniciam em 0.

Exemplo de uma matriz numérica de duas dimensões. Nesse exemplo são lidas duas notas para dez alunos.

```
#include <stdio.h>
```

```
int main(void)
{
    int Notas [10][2]; /* 10 indica a quantidade de alunos e 2 a quantidade de
    notas */
    int Aluno, Nota;
    for (Aluno=0;Aluno<10;Aluno++)
    {
        for (Nota=0;Nota<2;Nota++)
        {
            printf("Informe a %da. nota do aluno %d: ", Nota+1,Aluno+1);
            scanf("%d",&Notas[Aluno][Nota]);
        }
    }
}
```

Nesse exemplo, a estrutura for mais externa percorre o primeiro índice da matriz (de 0 a 9) e a estrutura for mais interna o segundo índice (de 0 a 1). A instrução `scanf("%d",&Notas[Aluno][Nota]);` lê a nota e a armazena nos índices indicados pelos contadores das estruturas de repetição (Aluno e Nota).

Exemplo de programa que lê dez nomes de alunos, strings, e os exibe na tela:

```
#include <stdio.h>
```

```
int main(void)
{
    char Nomes [10][100];/*para armazenar 10 nomes com até 99 caracteres cada*/
    int Cont;
    for(Cont=0;Cont<10;Cont++)
    {
        printf("Informe o %d. nome: ",Cont+1);
        gets (Nomes [Cont]);
    }
    printf("Os nomes informados são: \n");
    for (Cont=0;Cont<10;Cont++)
    {
```

```

        printf ("%do. nome: %s\n",Cont+1,Nomes[Cont]);
    }
}

```

Nesse exemplo, a instrução `gets(Nomes[Cont])` lê um nome e o armazena na posição `Cont` da matriz `Nomes`. É informado apenas o primeiro índice da matriz (`Nomes[Cont]`), o segundo indica o tamanho da string. Na leitura de uma string não é colocado o `&` antes do nome da matriz porque em uma matriz ou vetor de caracteres o seu nome (identificador) é um ponteiro para um endereço de memória.

Para declarar uma matriz multidimensional é necessário informar a quantidade de dimensões e o tamanho de cada uma dessas dimensões.

Sua forma geral de declaração é:

```

tipo_de_dado nome_da_variável [tam1][tam2] ... [tamN];

```

A forma geral para declarar e inicializar simultaneamente uma matriz é:

```

tipo_de_dado nome_da_variável [tam1][tam2] ... [tamN] = {lista_de_valores};

```

Onde:

Tipo_de_dado – indica o tipo de dado (int, float, double, char e seus modificadores) que serão armazenados na matriz.

Nome_da_variável – o identificador, nome, da matriz.

tam – indica o tamanho de cada uma das dimensões.

lista_de_valores – os valores que inicializam os endereços de memória reservados para a variável matriz.

A lista de valores é composta por dados (do mesmo tipo da variável) separados por vírgula. Os valores serão armazenados na matriz na ordem indicada.

Exemplos de inicialização de matrizes:

```

float Alturas[3] = { 1.23, 4.56, 678.90};
int Trimestres [4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char Nome1[6] = { 'N', 'o', 'm', 'e', ' ', '\0' };
char Nome2[6] = "Nome1";
char Nomes[3][11] = { "Nome1", "Nome2", "Nome3" };

```

Nesses exemplos a vírgula separa cada um dos elementos da matriz.

O primeiro exemplo mostra a inicialização de vetores.

A matriz `Trimestres [4][3]` armazena quatro linhas com três colunas, ou seja, três valores por linha.

Os três últimos exemplos são de matrizes de string. A forma de inicialização de `Nome1[6]` é equivalente a `Nome2[6]`. Isso porque é armazenado um conjunto de caracteres. Quando a inicialização é feita por caracteres separados, é necessário colocar o terminador de string (`\0`) no final para indicar fim da string.

No último exemplo é declarada uma matriz para armazenar até três nomes com até dez caracteres de tamanho cada.

3.8 Estruturas de dados heterogêneas

Uma estrutura (struct) agrupa um conjunto de variáveis que podem ser de tipos diferentes.

A forma geral de uma struct é:

```
struct identificador_da_struct
{
    //membros da estrutura;
};
```

Onde:

struct - palavra reservada que identifica uma estrutura. Ela indica ao compilador que um tipo de dado está sendo declarado e que será identificado pelo identificador_do_tipo.

Identificador_da_struct – o identificador é o nome da estrutura e indica que está sendo definido um tipo de dado com esse nome. Esse tipo será utilizado para declarar variáveis do tipo dessa estrutura.

{ } - delimita a declaração dos membros da estrutura. O ponto e vírgula (;) final é indispensável e indica o final da declaração da estrutura.

membros da estrutura – são as variáveis que fazem parte da estrutura e os seus respectivos tipos de dados. Esses dados podem ser do tipo simples (int, float, double, char), homogêneos (vetores, matrizes) ou heterogêneos (estruturas).

Exemplo de estrutura funcionário:

```
struct Funcionario
{
    int Codigo;
    char Nome[40];
    long int Cpf;
    float Salario;
};
```

Para declarar variáveis do tipo *Funcionario* que é uma estrutura, é utilizada a instrução:
Struct Funcionario Chefe, Atendente ;

Declara Chefe e Atendente como uma estrutura do tipo *Funcionario*. São variáveis do tipo estrutura *Funcionario*.

Para referenciar os membros de uma estrutura basta indicar o nome da estrutura seguido por ponto e do nome da variável membro que se quer referenciar. Exemplos:

```
Chefe.Codigo = 123;
Chefe.Salario = 1234.56;
Atendente.Salario = Chefe.Salario;
if (Chefe.Salario > 1200.23)
{
}
```

O tratamento das variáveis membros da estrutura é feito normalmente, com o acréscimo do nome da estrutura.

A declaração:

```
struct funcionario Funcionarios [10];
```

declara um vetor de estruturas do tipo *Funcionario*. Cada um dos 10 elementos dessa estrutura é do tipo *Funcionario*.

As seguintes instruções exemplificam o acesso aos elementos de *Funcionarios*:

```
Funcionarios [0].Codigo= 1223;
Funcionarios [1].Salario = 1234.56;
```

Exemplo de estrutura para data:

```
struct Data
{
    int Dia;
    int Mes;
    int Ano;
};
```

A instrução:

```
struct Data Data1, Data2;
```

Declara *Data1* e *Data2* como uma estrutura do tipo *data*.

Exemplos de manipulação de membros da estrutura *Data1* que é do tipo *data*:

```
Data1.Dia = 7;
Data1.Mes = 11;
Data1.Ano = 2010;
if (Data1.Ano > Data2.Ano)
{
}
}
```

Na declaração de estruturas, em vez de colocar um identificador de tipo da estrutura podem ser declaradas as suas variáveis diretamente.

Assim, a declaração das variáveis *Chefe* e *Atendente* do tipo estrutura para conter o registro de funcionários da empresa, ficaria:

```
struct Funcionario
{
    int Codigo;
    char Nome[40];
    long int Cpf;
    float Salario;
} Chefe, Atendente;
```

Para isso, após a chave que indica a finalização das variáveis membro da estrutura são colocadas as variáveis declaradas do tipo da estrutura.

Outra forma de declaração é:

```
struct
{
    int Codigo;
    char Nome[40];
    long int Cpf;
    float Salario;
} Chefe, Atendente;
```

A estrutura é declarada sem identificador de tipo e após a chave que indica a finalização das suas variáveis membro são colocadas as variáveis declaradas do tipo da estrutura. Com essa forma de declaração novas variáveis a serem declaradas desse tipo de estrutura devem ser acrescentadas à sua declaração.

3.9 Funções

Uma função é um conjunto de instruções de programa com uma finalidade específica.

Um programa definido na linguagem C é composto por um conjunto de funções. Quando é um programa executável, não é um arquivo de cabeçalho, por exemplo, uma dessas funções deve ser a função `main()`. A execução de um programa é realizada a partir da função `main()`. Dessa função são chamadas as outras funções que podem estar no mesmo arquivo que está a função `main()` ou em arquivos separados. Se estiverem em arquivos separados é necessário explicitar com `include` o arquivo que as contém.

As funções estão associadas à subprogramação, no sentido de dividir um problema e sua solução (o código de programa) em partes e implementá-las separadamente. Essa forma de procedimento facilita a implementação e a verificação de algoritmos, especialmente para problemas complexos e extensos, e possibilita reusar mais facilmente partes de código.

A forma geral de uma função:

```
tipo_de_retorno_da_função nome_da_função (parâmetros)
{
    //instruções;
    //return(Valor-a-ser-retornado); //pode ou não existir
}
```

Onde:

tipo_da_função nome_da_função (parâmetros) - é o cabeçalho, a assinatura, a interface da função. Descreve o tipo do valor retornado pela função, o nome ou identificador pelo qual é realizada a chamada da função e a lista de parâmetros, ou dados de entrada.

instruções - que podem incluir *return*, compõem o corpo da função. Define o conjunto de instruções que serão executadas quando a função é utilizada (chamada). O corpo de uma função que é delimitado pelos caracteres delimitadores de bloco que são { e }.

tipo_de_retorno_da_função - indica o tipo de valor que será retornado pela função. Uma função não necessariamente precisa ter retorno. Contudo, se ela tiver retorno é necessário que ela contenha uma instrução *return* (parâmetro). O tipo de dado retornado deve ser do mesmo tipo definido para a função. É possível que o tipo de retorno da função precise ser tratado na função chamadora. Por exemplo, uma função que verifica se um número é primo e retorna 's' se primo e 'n' se não primo, pode ter o seguinte tratamento na função chamadora:

```
Retorno = Eprimo(Numero);
if (Retorno == 's')
{
    printf("%d é primo",Numero); /*Numero é o valor passado para a função
para ser verificado e Retorno é a variável que contém o retorno da função. Nesse caso foi
convencionado que 's' significa que o número verificado é primo*/
}
else
{
    printf("%d não é primo",Numero); /*foi convencionado que 'n' significa não
```

```
primo.*/  
}  
/*Nesse exemplo, 's' e 'n' são os únicos valores possíveis de retorno da função */
```

O retorno de uma função não necessariamente precisa ser atribuído para uma variável, ele pode ser utilizado diretamente como argumento de uma função printf() ou em uma expressão matemática, por exemplo.

Exemplos:

```
printf("a raiz quadrada de 10 é %d", sqrt(10));
```

sqrt() calcula a raiz quadrada de um número. Essa função retorna um valor correspondente à raiz quadrada do número passado como parâmetro. No exemplo, a constante 10 é passada como parâmetro. Esse retorno é mostrado na tela do computador por meio da função printf(), isto é, sem necessidade de esse retorno ser armazenado em uma variável.

```
Var1 = trunc(10.234) + pow(2,3);
```

A função trunc() recebe um número e trunca a parte fracionária do mesmo. Essa função, no exemplo, retorna o número 10 que é somado com o retorno da função pow(). A função pow() recebe dois argumentos, o primeiro é a base e o segundo o expoente e calcula a potência desse número. Os retornos dessas funções são somados e armazenados na variável Var1.

```
Var2 = 12.34;
```

```
Var3 = pow (trunc(Var2), Var1);
```

A função pow() recebe como primeiro argumento o retorno da função trunc() e o conteúdo de Var1. A função trunc() recebe como parâmetro o conteúdo de Var2, trunc() esse valor e o passa como parâmetro para pow(). O retorno da função pow() é armazenado em Var3.

nome_da_função - é o nome pelo qual a função é identificada e chamada quando do seu uso. As regras para o nome de uma função são as mesmas para os nomes de variáveis.

*parâmetros*⁸ - parâmetros são os valores passados para a função. Parâmetros são também denominados argumentos. Quando a função é definida é necessário explicitar o tipo de dado que corresponde a cada um dos seus parâmetros. Se houver mais de um parâmetro eles serão separados por vírgula. Uma função não necessariamente precisa ter parâmetros. Na implementação da função esses parâmetros são utilizados nos cálculos e em comparações lógicas. Na execução da função são utilizados os valores contidos nesses argumentos, sejam variáveis ou constantes para realizar as ações explicitadas no código da função.

Uma variável que é parâmetro de uma função é uma variável local. E, portanto, tem validade apenas dentro da função na qual é declarada. As variáveis que são parâmetros da função e na chamada da função podem ter nomes distintos, mas devem ser do mesmo tipo. Por serem variáveis locais, o conteúdo de variáveis passado por parâmetro não é alterado pela função chamada. O tipo de passagem de argumento é por valor. Nesse caso, o valor da variável passado como parâmetro para a função é copiado para variável correspondente na função chamada.

{ ... } - delimitam o corpo da função, ou seja o início e o final do conjunto de instruções que pertencem à função.

⁸ No uso de uma função é necessário saber quantos parâmetros ela necessita, quais são, o tipo deles e a ordem deles. Exemplificando: na função potência é necessário saber que o primeiro parâmetro se refere à base e o segundo é o expoente e que essa função se aplica somente a números inteiros, por exemplo. Embora, não impõe o nome desses parâmetros.

3.9.1 Tipos de funções

Uma função é usada por meio do seu nome, o identificador. Quando uma função é usada deve ser considerado se a mesma possui ou não retorno e se ela requer ou não parâmetros. As combinações entre ter ou não retorno e/ou parâmetros são:

- a) sem retorno e sem parâmetros;
- b) sem retorno e com parâmetros, seja um ou mais;
- c) com retorno e sem parâmetros;
- d) com retorno e com parâmetros, seja um ou mais;

Essas combinações definem os tipos que as funções podem assumir considerando parâmetros e retorno.

Parâmetros ou argumentos são as entradas para a função, são os valores que a função recebe. Esses valores podem ser constantes ou variáveis. As funções `printf()` e `scanf()` são exemplos de funções que recebem argumentos. Parâmetros não são a única forma de uma função receber valores. Os dados podem ser lidos a partir de instruções definidas no corpo da função. Da mesma forma, instruções de saída podem fazer parte das instruções que compõem o corpo de uma função.

Exemplo:

```
int Quadrado(int X); //declaração da função
```

```
int main(void)
{
    int Num=10;
    int Retorno;
    Retorno = Quadrado(Num); //recebe um parâmetro do tipo inteiro
    printf("O quadrado de %d é %d",Num,Retorno);
}
```

```
int Quadrado(int X)//espera receber um parâmetro do tipo inteiro
{
    int Quad;
    Quad = X * X;
    return(Quad); //retorna um valor do tipo inteiro que é o tipo da função
}
```

É importante observar que:

a) Quando uma função é utilizada ou chamada é indispensável atender aos seus requisitos quanto ao tipo e à quantidade de parâmetros.

b) Os nomes das variáveis que são passadas para a função quando a mesma é chamada não tem qualquer vínculo com os nomes definidos para os parâmetros da função. No exemplo, o conteúdo da variável `Num`, ao ser passada como argumento para `Quadrado(int X)`, é copiado para a variável `X`. Na função `Quadrado(int X)` é utilizada a variável `X` para fazer os cálculos. O valor de `X` é mudado na função `Quadrado(int X)`, mas o valor de `Num` na função `int main(void)` permanece inalterado.

c) A instrução `Quad = X * X` é válida porque primeiro é realizada a operação que está à direita do sinal de igual, ou seja, `X * X` e em seguida o resultado dessa operação é armazenado na variável `Quad`.

d) A função precisa ser conhecida antes de ser utilizada pela função `main()`. Isso é

feito pela declaração do seu cabeçalho antes da função `main()` ou pela inserção de toda a função (o seu código) antes da função `main`.

O tipo de retorno de uma função é especificado antes do nome da função. O retorno para o nome da função é feito por meio do `return()`.

O comando `return` pode ter as seguintes formas:

```
return valor_de_retorno;
```

```
return;
```

```
return (valor_de_retorno);
```

O valor de retorno pode ser uma constante ou uma variável. Quando uma função está sendo executada e é encontrada uma chamada para `return()` a função é encerrada e, se o valor de retorno é informado, a função retorna esse valor. Uma função pode ter mais de uma instrução `return()` que são executadas condicionalmente. O programa é finalizado quando é processada a primeira instrução `return()`.

Exemplo de uma função sem retorno e sem parâmetros.

Declaração:

```
void Imprime_Texto( void)
{
    //instruções
}
```

Uso dessa função:

```
Imprime_Texto( );
```

Essa função não recebe parâmetros e não possui retorno. Pelo nome presume-se que ela imprimirá algum texto por meio de uma instrução de saída que está definida nas suas instruções. `void` entre parênteses significa que não é passado nenhum valor (denominado parâmetro) para a função e `void` no início da função, especificando o seu tipo, significa que ela não fornece nenhum retorno para a função na qual ela é usada, chamada.

Exemplo de função sem retorno e com parâmetros.

```
void Par_Impar(int Numero)
{
    if (Numero % 2 == 0)
    {
        printf(“%d é par”, Numero);
    }
    else
    {
        printf(“%d é ímpar”, Numero);
    }
}
```

Usos dessa função:

```
Par_Impar(Num);
```

```
Par_Impar(5);
```

Essa função recebe como parâmetro um número e não possui retorno. A função verifica se o número recebido como parâmetro é par ou ímpar e imprime uma mensagem correspondente.

Para passar um parâmetro para uma função: se for uma variável, ela deve ter sido declarada com tipo compatível à respectiva variável no parâmetro da função; se for uma função, o seu retorno deve ser de tipo compatível à variável do parâmetro que ela substitui; se for uma constante deve ter tipo compatível com o tipo estabelecido na declaração da função. O nome da variável passada como parâmetro não precisa ser o mesmo declarado na função, mas o tipo de dado deve ser compatível.

Exemplo de uma função com retorno e sem parâmetros.

```
char ValorMaior100 (void )
{
    int Num;
    char Retorno;
    printf("Informe um número inteiro: ");
    scanf("%d",&Num);
    if(Num >= 100)
    {
        Retorno='s';
    }
    else
    {
        Retorno='n';
    }
    return(retorno);
}
```

Uso dessa função:

```
ValorRetornado = ValorMaior100 ( );
```

Essa função não recebe parâmetros e possui retorno. No corpo da função está a instrução para ler um número, verificar se o mesmo é maior ou igual a 100 ou menor que 100 e fornecer um retorno 's' significando sim e 'n' não, conforme o caso.

A chamada de uma função que tem retorno é, normalmente, atribuída a uma variável. Em `ValorRetornado = ValorMaior100();` a variável `ValorRetornado` conterá o valor contido na variável que é retornado pela instrução `return()`⁹. Essa variável deve ser do mesmo tipo (tipo compatível) que o retorno definido para a função. Contudo, essa chamada não necessariamente precisa ser atribuída a uma variável ela pode ser colocada diretamente em uma instrução para imprimir, fazer parte de uma expressão numérica, ser parâmetro para outra função, dentre outros. Por exemplo: `printf("%c",ValorMaior100(Num));`

Exemplo de uma função com retorno e com parâmetros.

```
char Divisivel3 (int Num)
{
    char Retorno;
    if (Num % 3 == 0)
```

⁹ Uma função que não é do tipo void possui um return para indicar o valor retornado da função. Para a função `int main(void)`, de certa forma, é opcional ter um comando `return` porque o valor retornado é passado para o sistema operacional. Neste texto não será utilizado o comando `return` na função `main`, mesmo que ela seja declarada do tipo `int`. Esse retorno é importante se há interesse em tratá-lo, por exemplo, saber se o programa executou da maneira esperada.

```

    {
        Retorno = 's';
    }
    else
    {
        Retorno = 'n';
    }
    return(Retorno);
}

```

Uso dessa função:

```
ValorRetornado = Divisivel3(Valor );
```

Essa função verifica se o número passado como parâmetro é divisível por 3. Ela recebe um parâmetro do tipo inteiro que é o número a ser verificado e retorna um caractere 's' para indicar que o número é divisível por 3 e 'n' para indicar que ele não é divisível por 3. O retorno é recebido quando a função é chamada, isto é, quando utilizada, essa função retorna uma constante 's' ou 'n'.

Exemplo de uma função com retorno e com parâmetros.

Quando uma função possui mais de um parâmetro, eles são separados por vírgula e o tipo de cada um dos argumentos deve ser explicitado. Ressalta-se que os parâmetros passados para a função não necessitam ser variáveis, mesmo sendo constantes serão copiados para a respectiva variável de entrada da função. Essa função retorna o valor resultante do cálculo efetuado para a função que a chamou.

```
#include <stdio.h>
```

```
//função para multiplicar dois números passados como parâmetros
float Multiplicacao (float A, float B);
```

```
int main(void)
{
    float Num1;
    float Num2;
    float Resposta;
    printf("Informe um número: ");
    scanf("%f",&Num1);
    printf("Informe outro número: ");
    scanf("%f",&Num2);
    Resposta = Multiplicacao (Num1,Num2);
    printf("%f * %f = %f",Num1,Num2,Resposta);
}

```

```
float Multiplicacao (float A, float B);
{
    float Retorno;
    Retorno = A *B;
}

```

```

    return(Retorno);
}

```

A variável Resposta recebe o retorno da função que multiplica dois valores. Os valores multiplicados são do tipo float. Como a função mult retorna o resultado dessa multiplicação, ela também precisa ser float.

3.9.2 Protótipos de funções

Um programa executável e implementado na linguagem C deve ter uma função main(). As outras funções que compõem esse programa podem ser colocadas (o código que as define) antes da função main(). Por exemplo, uma função para calcular a soma de dois números, que recebe como parâmetro dois valores e retorna o valor calculado, poderia ter o seguinte código.

```

#include <stdio.h>

int Soma(int Num1, int Num2)
{
    int Resposta;
    Resposta = Num1 + Num2;
    return(Resposta);
}

int main(void)
{
    int Numero1;
    int Numero2;
    int Retorno;
    printf("Informe o primeiro número para somar: ");
    scanf("%d",&Numero1);
    printf("Informe o segundo número para somar: ");
    scanf("%d",&Numero2);
    //chamada da função
    Retorno = Soma (Numero1, Numero2);
    printf("Soma de %d + %d = %d",Numero1, Numero2, Retorno);
}

```

A função main() precisa conhecer as funções que ela chama para gerar o código executável corretamente. Assim, o compilador já as compilou e sabe o seu formato no momento de usá-las. Nesse exemplo, a função criada é colocada antes da função int main(void). Essa função está fisicamente antes da função int main(void).

Quando as funções não estão colocadas fisicamente antes da função main() é necessário declará-las antes da main(). Essa declaração pode ser de duas formas: com a inclusão de bibliotecas que possuem as funções por meio da diretiva # include ou pela declaração do protótipo, o cabeçalho, da função antes da main().

A declaração da função tem a seguinte forma:

```

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

```

onde:

tipo-de-retorno, *o nome-da-função* e *a declaração-de-parâmetros* são os mesmos usados na definição e implementação da função.

Exemplo de função utilizando declaração:

```
# include <stdio.h>
```

```
int Soma(int Num1, int Num2); //declaração do cabeçalho, protótipo da função
```

```
int main(void)
```

```
{
    int Numero1;
    int Numero2;
    int Retorno;
    printf("Informe o primeiro número para a soma: ");
    scanf("%d",&Numero1);
    printf("Informe o segundo número para a soma: ");
    scanf("%d",&Numero2);
    //chamada da função
    Retorno = Soma (Numero1, Numero2);
    printf("Soma de %d + %d = %d",Numero1, Numero2, Retorno);
}
```

```
int Soma(int Num1, int Num2)
```

```
{
    int Resposta;
    Resposta = Num1 + Num2;
    return(Resposta);
}
```

Nesse exemplo, a função Soma() está colocada depois de int main(void), mas a sua declaração (protótipo, cabeçalho) está antes. Assim, ela já está conhecida antes de ser utilizada.

3.9.3 Escopo de variáveis

O escopo determina a validade de uso de variáveis declaradas em um programa. Em relação ao escopo ou abrangência de uso as variáveis são classificadas em:

a) variáveis globais. São declaradas no início do programa, antes de todas as funções. Elas são conhecidas e podem ser usadas e alteradas por todas as funções do programa. Quando uma função tem uma variável local com o mesmo nome de uma variável global, a variável local terá preferência.

b) variáveis locais. Essas variáveis só têm validade dentro do bloco no qual elas são declaradas. Um bloco começa com uma chave de abertura e termina com o fechamento dessa chave. Esse escopo pode ser uma função (se a variável é declarada no início da função) ou uma estrutura de repetição (se a variável é declarada dentro dessa estrutura). Uma estrutura for, por exemplo, pode ter variáveis locais e que não serão conhecidas fora das chaves que limitam o seu bloco de código.

Por exemplo, em:

```
for (int Cont=0; Cont<10; Cont++)
{
    //escopo desta estrutura for
}
```

A variável *Cont* somente é conhecida no escopo dessa estrutura *for*.

c) variáveis locais como parâmetro de função. Essas variáveis são declaradas como sendo as entradas de uma função. Apesar de essas variáveis receberem valores externos, elas são conhecidas apenas pela função na qual são declaradas.

Exemplo:

```
#include <stdio.h>
```

```
int CalculaDobro(int A);
```

```
int Cont; //escopo global, visível para todas as funções declaradas neste programa
```

```
int main(void)
```

```
{
    char Sim = 's'; // escopo para a função main
    for (int Cont=0; Cont<10; Cont++) // Cont escopo para este for
    {
        int Resposta; //escopo para este for
        Resposta = CalculaDobro(Cont);
        printf("%d\n", Resposta);
    }
    printf("\n %c\n", Sim);
    for (Cont=0; Cont<3; Cont) //Cont de escopo global
    {
        printf("%d\n", Cont);
    }
}
```

```
int CalculaDobro(int A) //A escopo para esta função
{
    int Dobro; //Dobro escopo para esta função
    Dobro = A * A;
    return(Dobro);
}
```

A variável *Cont* é uma variável global e é acessível de qualquer função que compõe o programa. A variável *Sim* só existe dentro de *int main(void)*. É variável local à função *main*. A variável *Dobro* só é conhecida dentro da função *CalculaDobro* é uma variável local a essa função. A variável *Resposta* é um exemplo de uma variável de bloco. Ela somente é conhecida dentro do bloco do *for*, pertencente à função *main*. A variável *A* é um exemplo de declaração na lista de parâmetros de uma função (a função *CalculaDobro*).

Uma variável precisa ser declarada para que possa ser utilizada. Assim, uma variável não necessariamente precisa ser declarada no início do bloco que define o seu escopo. Contudo,

ela somente pode ser utilizada após a sua declaração.

3.9.4 Passagem de parâmetros por valor e por referência

Na linguagem C quando uma função é usada, os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros.

Exemplo:

```
#include <stdio.h>
```

```
float Soma (float Num1, float Num2);
```

```
int main(void)
```

```
{
    float Numero1;
    float Numero2;
    float Retorno;
    printf("Informe o primeiro número para a soma: ");
    scanf("%f",&Numero1);
    printf("Informe o segundo número para a soma: ");
    scanf("%f",&Numero2);
    //chamada da função
    Retorno = Soma (Numero1, Numero2);
    printf("Soma de %.2f + %.2f = %.2f",Numero1, Numero2, Retorno);
}
```

```
float Soma (float Num1, float Num2)
{
    float Resultado;
    Resultado = Num1 + Num2;
    return (Resultado);
}
```

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros dentro da função alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função é denominado de chamada por referência. A função utiliza a referência aos endereços das variáveis para alterar os valores das variáveis passadas como parâmetros.

Para que uma chamada por referência possa ser realizada é necessário que as variáveis que são parâmetros de uma função sejam declaradas como *ponteiros*. Ponteiro é a referência para poder alterar o conteúdo de uma variável fora da função. Quando a função for usada é necessário colocar o símbolo **&** antes do nome das variáveis passadas para a função. Esse símbolo indica que está sendo passado o endereço da variável.

Exemplo:

```
#include <stdio.h>
```

```
void Troca (int *A, int *B);
```

```

int main(void)
{
    int Num1,Num2;
    Num1=100;
    Num2=200;
    Troca (&Num1,&Num2);
    printf("\nNúmeros após a troca: %d %d\n",Num1,Num2);
}

void Troca (int *A, int *B)
{
    int Aux;
    Aux=*A;
    *A=*B;
    *B=Aux;
}

```

Nesse exemplo, é passado para a função *Troca* o endereço das variáveis *Num1* e *Num2*. Estes endereços são copiados nos ponteiros *A* e *B*. Por meio do operador *** é acessado o conteúdo apontado pelos ponteiros e modificado.

A função *scanf()* usa chamada por referência porque ela precisa alterar o conteúdo das variáveis que são passadas para ela. Por isso que é passado para a função o endereço da variável a ser modificada.

3.9.5 Vetores como argumentos de funções

Um vetor pode ser passado como argumento de uma função. Para isso a função deve ter um vetor como parâmetro. Por exemplo, uma função para somar os valores armazenados em um vetor:

```
int Funcao1(int Valores[], int Quantidade)
```

Essa função tem como parâmetros um vetor de inteiros e um valor inteiro que significa a quantidade de elementos armazenados no vetor.

Para usar a função:

```
int Funcao1(int Valores[], int Quantidade); //declarar o cabeçalho da função
```

```

int main(void)
{
    int Vet[10];
    int Cont; //variável com escopo para a função main
    int Soma; //variável com escopo para a função main
    for(Cont=0;Cont<10;Cont++)
    {
        printf("Informe o valor para Vet[%d]: ",Cont);
        scanf("%d",&Vet[Cont]);
    }
    //chamar a função
}

```

```

        Soma = Funcao1(Vet, 10);
        printf("A soma dos valores armazenados no vetor é %d",Soma);
    }

    int Funcao1(int Valores[], int Quantidade)
    {
        int Cont; //variável local para esta função
        int Soma=0; //variável local para esta função
        for(Cont=0;Cont<10;Cont++)
        {
            Soma = Soma + Vet[Cont];
        }
        return(Soma);
    }

```

3.9.6 Arquivos de cabeçalho

Arquivos de cabeçalho são compostos por funções tem o objetivo de facilitar o reuso de código. Por isso eles também são conhecidos como bibliotecas. Arquivos de cabeçalho são arquivos incluídos com a diretiva `# include`. É comum que os arquivos de cabeçalho tenham a extensão `.h`, mas isso não é obrigatório pode ser incluído um arquivo com extensão `.c` por exemplo ou outra que seja reconhecida pelo compilador em uso. A extensão **.h** vem de **header** (cabeçalho em inglês).

Ressalta-se que um arquivo incluído, não necessariamente precisa ter a extensão `.h`. É possível incluir arquivos com outras extensões, como `.c`, por exemplo.

Exemplo:

Uma função para determinar se um número é par poderia ser declarada em um arquivo de cabeçalho. O arquivo de cabeçalho chamado de 'funcao.h' tem a seguinte declaração:

```
int EPar(int Numero);
```

O código da função que está nesse arquivo de cabeçalho é:

```

int EPar (int Numero)
{
    char Resposta;
    if (Num%2 == 0)/* Verifica se Num é divisível por dois */
    {
        Resposta = 's';
    }
    else
    {
        Resposta = 'n';
    }
    return (Resposta);
}

```

No arquivo do programa que utilizará a função `EPar()`, o arquivo denominado `função.h` é incluído como um arquivo de cabeçalho.

`#include <stdio.h>` //os símbolos de `<` e `>` indicam que o arquivo está em pastas

definidas como “include” pelo compilador.

#include "funcao.h" //aspas indicam que o arquivo está na mesma pasta ou em subpastas que está a função main()

```
int main(void)
{
    int Num;
    printf("Informe um número: ");
    scanf("%d",&Num);
    if (EPar(Num)=='s') // o retorno da função é utilizado pelo if
    {
        printf("\n %d é par.",Num);
    }
    else
    {
        printf("\n%d é ímpar",Num);
    }
}
```