

1ª Edição

# Programação Computacional Básica

## Linguagem C

David Calhau Jorge  
David Gabriel Beilfuss Jorge

ISBN: 978-65-00-82989-1



9 786500 829891

CL

# Programação Computacional Básica – Linguagem C

*David Calhau Jorge*  
*David Gabriel Beilfuss Jorge*

2023



## Regras de utilização do texto.

- 1) Todos os textos e programas contidos neste documento são de direito autoral do autor.
- 2) A distribuição deste texto e seus programas poderá ser realizada de forma integral ou parcial sem nenhum ônus para o interessado, sendo que este item pode ser revogado pelo autor a qualquer momento sem prévio aviso.
- 3) É vetada a utilização de qualquer um dos programas incluídos neste guia para fins militares ou que possam causar danos a propriedades e pessoas.
- 4) Os programas e rotinas deste texto são meramente instrutivos e em hipótese alguma serão de responsabilidade do autor quaisquer danos de quaisquer espécies causados por eles, se você discorda deste item não utilize este texto.
- 5) É vetada a reprodução deste texto ou qualquer parte dele em qualquer outro texto, livro e afins, assim como meios eletrônicos ou qualquer outra forma que venha a existir com fins lucrativos, sem a devida autorização do autor por manuscrito original.

# Sumário

<b>1-LINGUAGEM DE PROGRAMAÇÃO – FUNDAMENTOS.....</b>	<b>4</b>
<b>2-OPERAÇÃO NUMÉRICA DO COMPUTADOR.....</b>	<b>6</b>
2.1-FUNDAMENTOS DA ÁLGEBRA DE BOOLE.....	6
2.2-BASES NUMÉRICAS.....	8
<b>3-LINGUAGENS DE PROGRAMAÇÃO.....</b>	<b>11</b>
<b>4-INTRODUÇÃO HISTÓRICA DA LINGUAGEM C.....</b>	<b>13</b>
<b>5-ESTRUTURA DO PROGRAMA.....</b>	<b>14</b>
<b>6-MANIPULAÇÃO DE VARIÁVEIS E CONSTANTES.....</b>	<b>15</b>
6.1-VARIÁVEIS.....	15
6.2-CONSTANTES.....	16
6.3-ARITMÉTICA DE PONTOS FLUTUANTES.....	16
6.4-ARMAZENAMENTO DE VARIÁVEIS.....	16
<b>7-OPERAÇÕES MATEMÁTICAS.....</b>	<b>17</b>
<b>8-COMANDO DE E/S DE DADOS VIA TECLADO E MONITOR.....</b>	<b>19</b>
8.1-COMANDO “PRINTF” (C).....	19
8.2-COMANDO “STD::COUT” (C++).....	20
8.3- COMANDO PUTS.....	21
8.4-COMANDO SCANF (C).....	21
8.5-COMANDO “CIN” (C++).....	22
8.6- COMANDO GETS.....	22
<b>9-OPERADORES.....</b>	<b>24</b>
9.1-OPERADORES BIT A BIT.....	24
9.2-O OPERADOR “VÍRGULA”.....	25
9.3-O OPERADOR “?”.....	25
<b>10-DIRETIVA “#DEFINE”.....</b>	<b>27</b>
<b>11-TOMADAS DE DECISÃO E LOOP’S.....</b>	<b>28</b>
11.1-COMANDO “IF-ELSE” (SE-SENÃO).....	28
11.2-O LAÇO DE “FOR” (PARA).....	30
11.3-O LAÇO DE “WHILE” (ENQUANTO).....	31
11.4-O LAÇO “DO { } WHILE” (FAÇA { } ENQUANTO).....	32
11.4.1-Validação de dados.....	33
11.4.2-Pausa na tela.....	34
11.5-INSTRUÇÕES DE CONTROLE DE LOOPS.....	35
11.5.1-Instrução break.....	35
11.5.2-Instrução continue.....	35
11.6-COMANDO “SWITCH”.....	36
<b>12-MATRIZES E VETORES.....</b>	<b>38</b>
12.1-EXEMPLO DE MANIPULAÇÃO DE MATRIZES: MULTIPLICAÇÃO DE MATRIZES.....	40
12.2-EXEMPLO DE MANIPULAÇÃO DE MATRIZES: SOLUÇÃO DE SISTEMAS LINEARES.....	41
<b>13-PONTEIROS.....</b>	<b>44</b>
<b>14-FUNÇÕES.....</b>	<b>48</b>
14.1-EXEMPLO DE FUNÇÃO: ORDENAÇÃO DE ELEMENTOS EM UM VETOR.....	50
14.2-EXEMPLO DE FUNÇÃO: DETERMINAÇÃO DE UMA INTEGRAL DEFINIDA.....	51
<b>15-VARIÁVEIS LOCAIS E GLOBAIS.....</b>	<b>54</b>
<b>16-ARQUIVOS.....</b>	<b>56</b>
16.1-MANIPULAÇÃO DE ARQUIVOS COM A BIBLIOTECA STDIO.H”.....	57
16.2- MANIPULAÇÃO DE ARQUIVOS EM C++ - BIBLIOTECA “FSTREAM.H”.....	60
<b>17-ESTRUTURAS E UNIÕES.....</b>	<b>61</b>
<b>BIBLIOGRAFIA.....</b>	<b>63</b>

## 1-Linguagem de Programação – Fundamentos

Os computadores existem atualmente na vasta maioria dos lares de diversas pessoas. Sua origem poderia remontar de várias fontes, mas cabe citar a máquina de Jacquard utilizada na tecelagem, criada por Joseph Marie Jacquard (1752-1834), vide fig.1(a) que inventou a máquina que elaborava complexas formas no tecido, vide fig.1(b).



(a)



(b)

*Figura 1 – Joseph Marie Jacquard e um exemplo de sua programação por cartões. Fontes: (a) Produzido em 1839 utilizando a programação de Jacquard, (b) foto de George H. Williams (jun. 2004) Museu da Ciência e da Indústria de Manchester, Inglaterra.*

Uma das razões para recorrer a essa origem como referência no texto é que essa forma de programação de computadores pode ser encontrada em exemplos relativamente modernos, com os computadores programados com cartão.

Seria correto afirmar que a programação seria uma sequência de instruções que devem ser rigorosamente seguidas para se obter um resultado final. Um exemplo simples poderia ser a rotina para se sair por uma porta que poderia ser detalhada como a seguir:

1. Dirija-se a porta.
2. Gire a maçaneta.
3. Empurre a porta.
4. Saia pela abertura da porta.

A sequência das instruções é tão relevante quanto a forma como ela é informada a pessoa que deseja se retirar pela porta. Uma mudança em sua ordem, ou uma ordem ou uma instrução obscura poderiam não resultar no resultado desejado, que é sair de um local.

As linguagens de programação possuem essa característica, sendo ainda mais rigorosas com as instruções a serem seguidas, que devem ser exatas e sempre com uma forma pré-definida.

Sua aplicação na área de ciências exatas, médicas e humanas é extremamente vasta e poderia realizar diversas tarefas que manualmente seriam exaustivas ou temporalmente impossíveis. Podemos tomar um exemplo simples de um problema, como o proposto a seguir:

Sendo a equação  $f(x)=2^x+3^x$ , sendo  $x \in \mathbb{N}$  qual seria o valor de  $x$  que satisfaz a solução:  $f(x)=97$  ?

Resposta: Uma maneira de descobrir este valor seria a utilização de cálculos sequenciais a partir de um determinado valor, seguindo o diagrama abaixo:

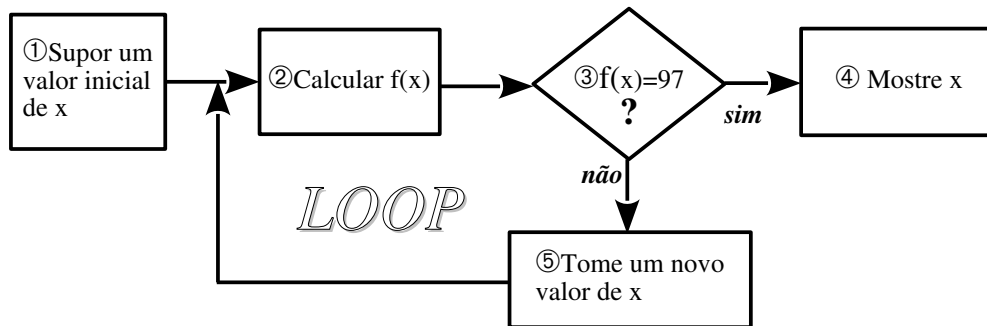


Figura 2 – Exemplo de um diagrama para resolver o problema proposto.

Notemos que há uma sequência lógica a ser seguida e um laço que se repete (“loop”) para que a resposta seja obtida, como temos uma indicação que o número é Natural seria mais fácil obter a resposta para esse caso. Realizando a sequência indicada:

- 1 - supor  $x=0$
- 2 -  $f(x)=2$
- 3 -  $f(x)$  não é igual a 97 – ir para 5
- 5 - supor  $x=1$
- 2 -  $f(x)=5$
- 3 -  $f(x)$  não é igual a 97 – ir para 5
- 5 - supor  $x=2$
- 2 -  $f(x)=13$
- 3 -  $f(x)$  não é igual a 97 – ir para 5
- 5 - supor  $x=3$
- 2 -  $f(x)=35$
- 3 -  $f(x)$  não é igual a 97 – ir para 5
- 5 - supor  $x=4$
- 3 -  $f(x)$  é igual a 97 – ir para 4
- 4 – Mostrar que  $x=4$  é solução.**

Porém este problema poderia se complicar rapidamente se o valor não pertencesse aos números Naturais, mas aos números Reais se tivéssemos por exemplo que  $f(x)=8559,640141987$  e desejássemos uma resposta com até duas casas decimais de precisão esse diagrama seria trabalhoso de se realizar manualmente, neste caso deveríamos utilizar um computador, com esse mesmo diagrama programado para realizar esse cálculo.

Pode-se observar que computadores podem ser utilizados para realizar operações repetitivas (“loops”) com uma certa facilidade. Em uma maioria dos casos os procedimentos realizados pelos computadores serão muito mais rápidos que os procedimentos manuais. Esta ideia pode ser ampliada para programações de ações repetitivas em plantas industriais. O que pode ser comprovado com a origem aqui analisada da Máquina de Jacquard.

## 2-Operação numérica do computador

### 2.1-Fundamentos da Álgebra de Boole

As operações realizadas na vasta maioria dos computadores são provenientes de operações realizadas com números binários, representados por "0" e "1". Os números binários poderiam ser comparados ao conceito de "ligado" e "desligado" ou "verdadeiro" e "falso". Em um circuito elétrico a detecção do estado "ligado/desligado" é uma das operações mais simples de ser constatada.

A álgebra de Boole, desenvolvida inicialmente por George Boole (1815-1864) em seu livro "The Mathematical Analysis of Logic" (1847) é utilizada em variáveis binárias, visando possibilitar a criação de todas as operações e atuações dos computadores digitais.

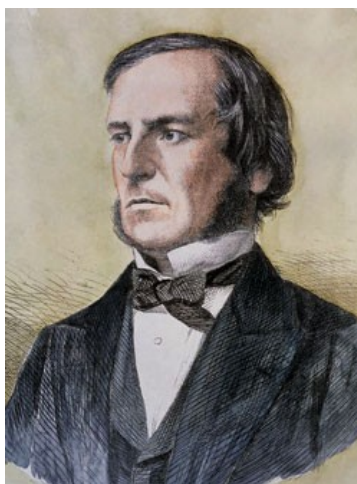


Figura 3 – George Boole (autor desconhecido).

Esta lógica se baseia em três operadores lógicos:

- Operador **E** - concebe uma ideia de limitação, somente é satisfeito quando as variáveis são verdadeiras. Símbolo  $\wedge$
- Operador **OU** - indica uma flexibilidade, uma vez que ao menos uma das variáveis deve ser verdadeira para que a expressão resultante seja verdadeira. Símbolo:  $\vee$
- Operador **NÃO** - inverte a indicação da variável (de falso para verdadeiro ou de verdadeiro para falso). Símbolo  $\neg$

É possível elaborar uma tabela com a análise desses operadores para duas variáveis em todas as situações possíveis, sendo essa conhecida como tabela verdade, apresenta o funcionamento destes operadores.

Supondo que a situação: 0-falso e 1-verdadeiro, seja proposta para duas variáveis A e B, poderíamos construir todas as combinações possíveis das duas variáveis e analisar como os operadores indicados seriam observados. Vide a Tabela 1

Tabela 1 – Tabela verdade com duas variáveis.

A	B	A ou B ( $A \vee B$ )	A e B ( $A \wedge B$ )	Não A ( $\neg A$ )	Não B ( $\neg B$ )
0	0	0	0	1	1
0	1	1	0	1	0
1	0	1	0	0	1
1	1	1	1	0	0

Poderiam surgir situações mais complexas com a álgebra de Boole, com a presença de mais variáveis. Existem ainda propriedades relevantes a essa análise, dentre as quais poderiam ser enumeradas as a seguir (O'Regan, 2008):

*Comutativa:*  $A \vee B = B \vee A$

$$A \wedge B = B \wedge A$$

*Associativa:*  $A \wedge (B \wedge C) = (A \wedge B) \wedge C$

$$A \vee (B \vee C) = (A \vee B) \vee C$$

*Identidade:*  $A \wedge \text{Falso} = \text{Falso} (\forall A)$

$$A \vee \text{Verdadeiro} = \text{Verdadeiro} (\forall A)$$

*Distributiva:*  $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

*De-Morgan:*  $\neg(A \wedge B) = \neg A \vee \neg B$

$$\neg(A \vee B) = \neg A \wedge \neg B$$

*Idempotente:*  $A \wedge A = A$

$$A \vee A = A$$

Uma análise da Álgebra de Boole pode ser representada também por diagramas de Venn, como observado na fig. 4.

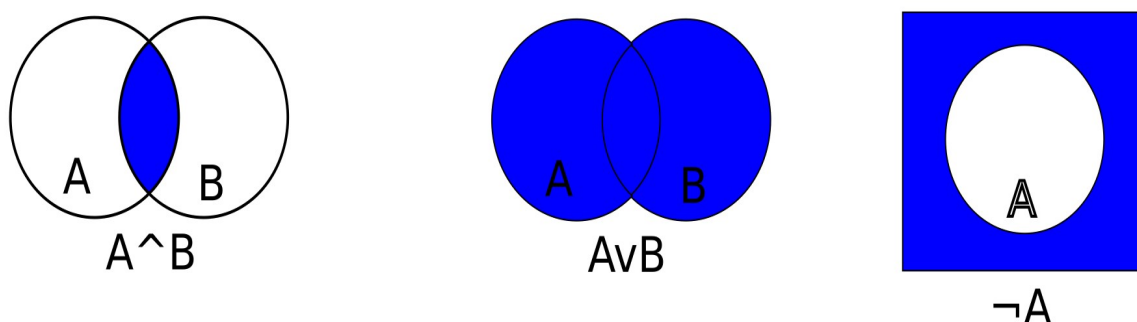


Figura 4 – Diagrama de Venn da conjunção, disjunção e complemento.

É possível analisar problemas lógicos para interpretar a proposta da Álgebra de Boole. Por exemplo a afirmação: “Contrata-se engenheiro que tenha fluência em Inglês e Alemão.” O conector “e” limita a situação e apenas quando a fluência em ambas as línguas ocorrer tem-se um candidato elegível. Substituindo o conector “e” por “ou” muda não apenas a frase, mas os tipos de candidatos elegíveis, observa-se que: “Contrata-se engenheiro que tenha fluência em Inglês ou Alemão.” possibilita candidatos que sejam fluentes em uma das línguas ou em ambas. Essa situação é observável na terceira e quarta colunas da Tabela 1.



## 2.2-Bases Numéricas

Os computadores em sua maioria se baseiam em uma lógica binária, onde apenas os estados 0 e 1 existem. A Álgebra de Boole é de extrema importância para análises dessa base, porém é necessário citar que o conhecimento dessa base é importante. Existe a possibilidade de se trabalhar matematicamente em diversas bases numéricas, assim como converter valores de uma base para outra.

A base binária é fundamentada na existência de apenas dois símbolos (como citado), diferente da base indo arábico decimal que utiliza dez símbolos (e é largamente difundida na atualidade).

Decimal: 0 1 2 3 4 5 6 7 8 9 - utiliza-se em alguns casos a presença de um "d" minúsculo ao final para identificação

Binária: 1 0 (ligado/desligado) - utiliza-se em alguns casos a presença de um "b" minúsculo ao final para identificação

A presença de um número tão reduzido de símbolos torna imprescindível a utilização de agrupamento de números, semelhante a utilizada na lógica decimal:

Decimal:

379d temos:      3-casa das centenas ( $3 \cdot 10^2$ ) = 300  
                         7-casa das dezenas ( $7 \cdot 10^1$ ) = 70  
                         9-casa das unidades ( $9 \cdot 10^0$ ) = 9

logo  $300d + 70d + 9d = 379d$

Binário, note que os múltiplos se encontram na base binária ( $2^{10} = 1.024$ ):

0/1 - bit e é a menor unidade da informação dos computadores.

4 bits – 1 nibble

8 bits – 1 byte


1kilobyte – 1.024 bytes

1Megabyte – 1.024 kB

1Gigabyte – 1.024 MB

A conversão de números binários para números decimais pode ser feita utilizando-se de um recurso simples, a divisão em sequência do número na forma decimal pelo número de elementos da base requerida, posteriormente os restos encontrados são transcritos inversamente, como pode ser observado na Tabela 2.

**Tabela 2 – Exemplo da conversão do número 76d para binário e dos números 1101b para decimal**

76d - ?b			1101b - ?d	
76/2=38	resto 0		$2^0 \cdot 1 = 1$	
38/2=19	resto 0		$2^1 \cdot 0 = 0$	
19/2=9	resto 1		$2^2 \cdot 1 = 4$	
9/2=4	resto 1		$2^3 \cdot 1 = 8$	
4/2=2	resto 0		$1 + 0 + 4 + 8 = 13d$	
2/2=1	resto 0		então teremos 13d	
1/2	resta 1			
então teremos o número 1001100b				

Para efetuar a conversão de uma base qualquer para a base decimal basta recorrer a operação da eq. (1).

$$O = \sum_{i=0}^n A_i \cdot s^i \quad (1)$$

Onde  $s$  é o número de símbolos,  $A$  o valor do símbolo convertido na base decimal e  $i$  a posição onde se encontra o símbolo, a partir da posição zero. Exemplos de conversão decimal↔binário, podem ser observados na Tabela 2.

Um detalhe importante a ser observado é que há um limite numérico de valores que um computador pode operar. Dessa forma é necessário converter valores que são analógicos, ou seja, valores sem lacunas e contínuos com infinitas posições possíveis, para valores de representação limitada. A representação limitada mais comum em computadores para programação é a digital. Por exemplo se tivermos um termômetro analógico de álcool com infinitas posições possíveis devemos reduzir nossas medidas a valores digitalmente representáveis, como a fig. 5.

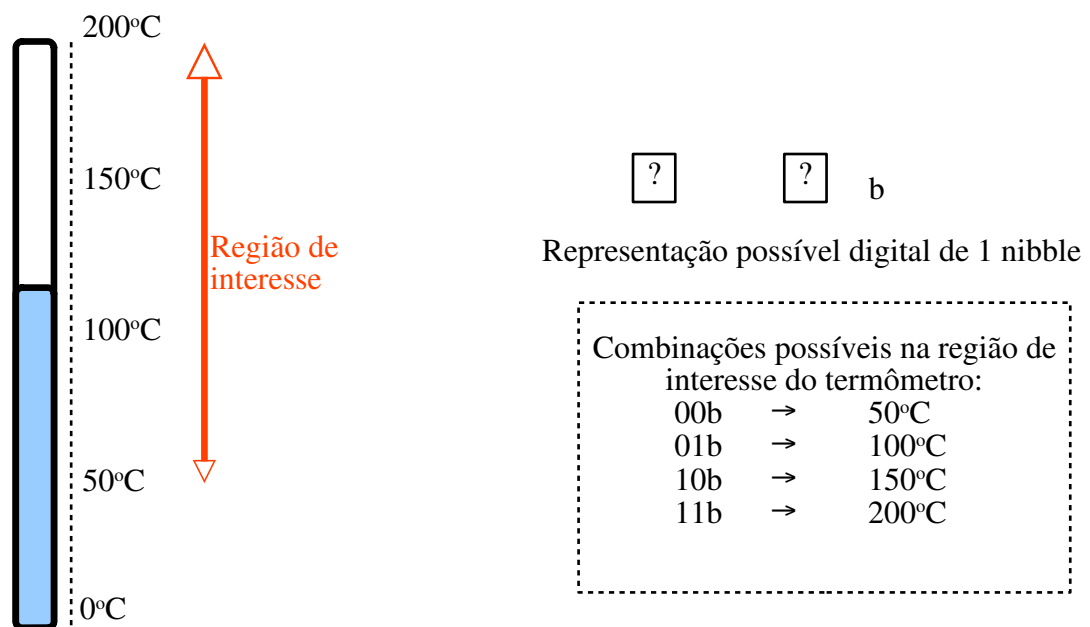


Figura 5 – Exemplo de uma conversão analógico – digital.

Nota-se que na fig. 5, os valores da região de interesse foram limitados a 4 representações possíveis. No caso do exemplo em particular seria possível obter uma leitura mais exata utilizando a gradação do termômetro, mas deverá ocorrer um arredondamento para cima ou para baixo quando esse valor for utilizado no modo digital proposto.

Essa análise pode se estender a diversas outras aplicações, seria correto afirmar que no modo digital os valores são limitados, mas essa é a forma mais simples de lidar com programação de computadores. Mais informações podem ser obtidas em (Kester, 2006).

Observe ainda que é possível realizar a conversão analógico para digital ou seu inverso utilizando componentes eletrônicos apropriados, dispositivos ou circuitos destinados a essa finalidade. Atualmente uma vasta gama de aplicações substituíram dispositivos analógicos por digitais, por exemplo: câmeras fotográficas, aparelhos de som, televisores, dentre outros.

Como é possível converter para bases binárias e decimais o mesmo pode ocorrer para outras bases numéricas existentes, a forma de conversão é semelhante a apresentada anteriormente, porém os termos irão depender da base escolhida, vide a

### Tabela 3.

Dentre as bases existentes se destacam:

- Base Ternária (t) – com três símbolos: 0, 1, 2
- Base Octal (o) – com sete símbolos: 0, 1, 2, 3, 4, 5, 6, 7
- Base Hexadecimal(h)–com 16 símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.  
Note que Ah=10d, Bh=11d, Ch=12d, Dh=13d, Eh=14d e Fh=15d.

Essas bases podem ser utilizadas em locais restritos, mas tem seu uso observado em diversos locais: como computação (binário, octal e hexadecimal), *transponders* aéreos (octal), circuitos eletrônicos e dispositivos (ternária), dentre outros.

**Tabela 3 – Exemplo da conversão do número 76d para ternário e dos números 1101b para decimal**

76d - ?t 76/3=25      resto 1 25/3=8        resto 1 8/3=2         resto 2 resta 2 então teremos o número 2211t	76d - ?o 76/8=9        resto 4 9/8=1         resto 1 resta 1 então teremos o número 114o	76d - ?h 76/16=4       resto 12(C) resta 4 então teremos o número C4h
---	--	--

A conversão da parte fracionária, ou após a vírgula, pode também ser realizada. Neste caso os elementos após a vírgula devem ser negativos, como observado na eq. 2.

$$\dots \text{base}^4 \text{base}^3 \text{base}^2 \text{base}^1 \text{base}^0, \text{base}^{-1} \text{base}^{-2} \text{base}^{-3} \text{base}^{-4} \dots \quad (2)$$

A conversão para a base após a vírgula é realizada de modo diferente: deve-se multiplicar o número pela base escolhida e então guardar o resultado antes da vírgula e repetir o processo eliminando números antes da vírgula até que a quantidade de dígitos desejado seja atingida, um valor sem dígitos após a vírgula seja obtido ou uma dízima periódica, o número após a vírgula é obtido na ordem que aparece na regra os números antes da vírgula. Por exemplo convertendo o número 10,75d para binário seria como apresentado abaixo:

Parte inteira, antes da vírgula = **10** logo: 10/2=5(resto **0**)/2=2(resto**1**)/2=**1** então teríamos que **110b=10d**

Parte fracionária, após a vírgula = **0,75** logo: 0,75x2=**1**,5x2→0,5x2=**1**,0 então temos que **0,75d=0,11b**

**Então temos que 10,75d=10,11b**

Verificação: convertendo de binário para decimal:  $1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 2,75$

### 3-Linguagens de Programação

Existem diversos modos de classificar e utilizar as linguagens de programação destinadas a computadores. A maioria das linguagens de programação segue uma lógica clara das instruções desejadas mediante uma sintaxe rigorosa, que não pode ser modificada e deve conter uma sequência exata das operações desejadas.

Pode-se considerar a existência de gerações das linguagens de computador, sendo elas, apresentadas de modo resumido:

- Primeira geração - Linguagem de máquina, diretamente apresentada à máquina, sem utilização de programas auxiliares para sua interpretação. Ininteligível para o ser humano, geralmente na forma binária ou hexadecimal.
- Segunda geração - Linguagem de montagem (Assembly), onde um código próximo as instruções das linguagens de primeira geração são utilizadas. Pode ser mais compreensível e analisável para o ser humano.
- Terceira geração - Linguagens procedurais, são concebidas para serem facilmente compreensíveis aos seres humanos.
- Quarta geração - Linguagens aplicativas, permitindo ao usuário especificar o que se deseja ser feito. Visam principalmente a criação de aplicações comerciais.
- Quinta geração - Linguagens voltadas a Inteligência artificial como as linguagens lógicas e as funcionais, tais como Prolog e LISP.

O código fonte é a forma como as instruções são apresentadas na linguagem escolhida e devem seguir rigorosamente as instruções das linguagens. Cabe salientar, no entanto, que muitas linguagens são concebidas em diferentes Sistemas Operacionais (SOs) e nesse caso pode não ocorrer uma compatibilidade completa do código fonte. Em alguns casos recursos disponibilizados em um SO pode não estar disponível em outro, ou se apresentar com uma forma diferente. A versatilidade de algumas linguagens deve ser considerada na hora da escolha de qual será utilizada. A forma para se obter o programa executável neste caso é conhecido como Compilação, apresentado na fig. 6.

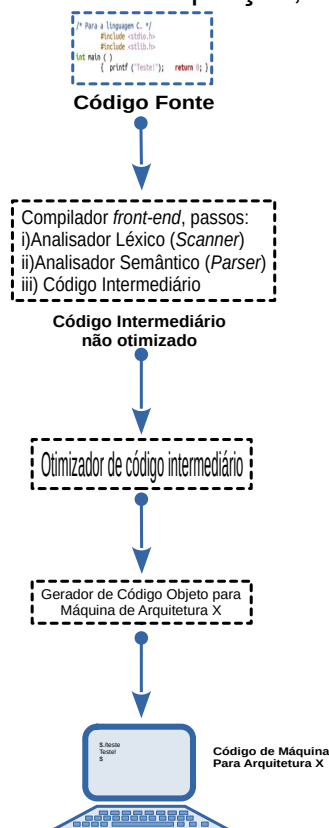


Figura 6 – Processo de Compilação.

É possível ainda desenvolver programas que envolvam mais de uma linguagem. Dessa forma não é incomum observar um programa que possui diversos códigos fontes diferentes.

Um programa de computador, após a concepção de seu código fonte pode ser executado de duas maneiras principais diferentes:

1. O código pode ser compilado, neste caso há uma sequência como apresentada anteriormente, até a criação de um código executável. É o caso de diversas linguagens tais como: Basic, Pascal, Fortran, C, C++, Visual C, etc.
2. A linguagem pode ser interpretada, neste caso é necessária a presença de um aplicativo que irá executar a linguagem de modo interativo. Essa forma possui uma maior compatibilidade entre códigos fonte de diferentes SOs, porém na maioria dos casos sua velocidade de execução é reduzida. É o caso de linguagens como: Java, Python, C#, entre outras.

A escolha da forma de execução é inerente a necessidade da aplicação que está sendo desenvolvida pelo programador.

Atualmente existem centenas de escolhas possíveis para o programador, sendo em alguns casos a opção pessoal a responsável pela determinação da linguagem a ser utilizada. A popularidade das linguagens muda constantemente, mas é possível notar que não ocorrem mudanças significativas, a seguir estão enumeradas as 20 linguagens mais populares em 2020:

- |                      |                  |
|----------------------|------------------|
| 1. C                 | 11. Swift        |
| 2. Java              | 12. Go           |
| 3. Python            | 13. Ruby         |
| 4. C++               | 14. Assembly     |
| 5. C#                | 15. MATLAB       |
| 6. Visual Basic .NET | 16. Perl         |
| 7. JavaScript        | 17. PL/SQL       |
| 8. PHP               | 18. Scratch      |
| 9. R                 | 19. Visual Basic |
| 10. SQL              | 20. Rust         |

Após a escolha da linguagem e a concepção do código fonte são necessários diversos testes para verificar a correta execução do programa elaborado. Alguns detalhes sempre devem ser considerados, dentre eles:

- i. Pode haver mais de um modo de se criar um código fonte, obtendo-se os mesmos resultados. Neste caso o que poderia ser otimizado é o desempenho do código fonte, tema de estudos como na Complexidade de Algoritmos.
- ii. A correta Compilação de um código fonte não indica que o programa funciona corretamente, apenas que ele está com ausência de erros de sintaxe ou fundamentos lógicos.
- iii. Os erros podem ser de: sintaxe ou fundamentos lógicos, facilmente observáveis nos Compiladores disponíveis atualmente; ou de concepção do código que são muito mais difíceis de se detectar, neste caso o programa está Compilado, porém não executam as tarefas desejadas.
- iv. Existem atualmente diversas linguagens que possuem inúmeros Compiladores distintos, podendo ser necessário alguns ajustes nos códigos fonte para que eles sejam Compilados corretamente.

Para os exemplos e análises de código de programação apresentados a seguir a Linguagem C foi escolhida como exemplo, porém outras linguagens dispõem das mesmas características apresentadas. O usuário poderia optar por outra linguagem, observando a forma como a mesma trata os assuntos ora abordados.

## 4-Introdução Histórica da Linguagem C

A linguagem C foi inicialmente implementada por Dennis Ritchie (1941-2011), vide fig. 7, em um computador DEC PDP-11, que utilizava o sistema operacional UNIX. Todo o histórico da linguagem C poderia se resumir aos itens apresentados a seguir:

1963- C.P.L. - Combined Programming Language

1967- B.C.P.L. - Basic Combined Programming Language

1970- B (Ken Thompson - Bell Labs)

1972- C (Dennis Ritchie)

1979 – Criação do C++ (Bjarne Stroustrup)

1983- Padronização do C com a criação do ANSI-C

1998- Padronização do C++ (que ocorre em média a cada 3 anos a próxima deverá ser em 2020).



Figura 7 – Dennis Ritchie fotografado por Denise Panyik em 2011.

A linguagem C possui diversas características que a tornam a mais importante linguagem a ser estudada no momento, entre elas:

- É uma linguagem estruturada, permitindo a presença de sub-rotinas com variáveis internas distintas das variáveis do programa principal, e uma forma de programação mais bem elaborada e sucinta.
- É uma linguagem compacta, possuindo apenas 32 palavras-chaves fundamentais:

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

- Possui uma vasta quantidade de recursos e “bibliotecas” com funções implementadas.
- Possui uma enorme portabilidade, uma vez que a maioria dos equipamentos de informática possui interpretadores ou compiladores C.
- É extremamente flexível, podendo ser utilizada em conjunto com outras linguagens de programação com relativa facilidade, uma vez que sua aceitação entre os desenvolvedores de “softwares” é muito grande.
- Desempenha de forma veloz o processamento em comparação com outras linguagens do mesmo tipo.
- Diversos dispositivos e equipamentos eletrônicos utilizam a linguagem C como primeira linguagem para sua programação.

A linguagem C é classificada como uma linguagem de médio nível, se encontrando entre uma linguagem de baixo nível (como o “Assembly”), possibilitando manipular bits e bytes diretamente e uma linguagem de alto nível (como Pascal, Fortran, etc.) as quais são utilizadas por programadores e possuem diversos recursos e uma melhor manipulação de diferentes tipos variáveis.

## 5-Estrutura do programa

A linguagem C é “*case sensitive*” ou seja possui discriminação entre letras maiúsculas e minúsculas, este fato não ocorre em todas as linguagens de programação. Os comandos em C devem ser escritos com letras minúsculas e as variáveis declaradas devem seguir sempre a norma utilizada em sua criação, por exemplo se uma variável criada for chamada de "A" ela deverá ser utilizada como "A" maiúsculo durante todo o programa.

Os programas desenvolvidos em C possuem como característica relevante a presença marcante de solicitação de acesso a bibliotecas no início do programa. Isto ocorre principalmente devido ao limitado número de palavras-chaves na linguagem, o que torna essencial a necessidade de bibliotecas com mais comandos disponíveis. Há uma grande quantidade de bibliotecas desenvolvidas para a linguagem C. É comum entre os programadores o desenvolvimento de bibliotecas particulares visando aperfeiçoar os recursos disponíveis e criar uma personalização dos programas desenvolvidos.

Inicialmente pode-se observar as características apresentadas a seguir em um programa desenvolvido em C:

- Os comentários ao longo do programa devem se encontrar entre /\* e \*/ os quais marcam início e fim de comentário, respectivamente, como o exemplo:

/\* Aqui vai o comentário. \*/

Para se comentar apenas uma linha basta utiliza “//” no início do comentário e tudo a partir deste ponto é ignorado para o código do programa.

- Não há necessidade de tabulação inicial, podendo o código, respeitando a grafia correta, se iniciar e terminar em qualquer coluna da linha de código, o que não ocorre em todas as linguagens, como por exemplo o Fortran.
- As bibliotecas a serem utilizadas são descritas no início do programa, precedidas de *#include*, e devem ter seu nome colocado entre <>, como no exemplo a seguir:

*#include <stdio.h>*

- A indicação “std::” faz referência a biblioteca padrão para execução da função apresentada a seguir dos dois pontos.
- Uma sequência de instruções distinta deve sempre se encontrar entre chaves, existe ao menos uma sequência de instruções em todos os programas, porém é comum a presença de mais que um par de chaves pois diversos trechos do programa podem estar relacionados a sub-rotinas ou a laços de repetição.
- Todas as sub-rotinas do programa em C devem possuir um nome e pertencer a uma classe de variável e devem retornar um valor, tal situação pode ser contornada declarando a classe da rotina como *void* (nula). A primeira sub-rotina a ser executada é a principal (*main*) esta sub-rotina deve obrigatoriamente existir no programa principal e seu conteúdo deve indicar todos os principais procedimentos do programa. As sub-rotinas podem receber valores externos para retornar valores dependendo dos valores recebidos, de forma semelhante ao que ocorre em funções matemáticas, como nos exemplos a seguir:

f(x)=5x            implica que    f(2)=10  
f(a,b)=a+b       implica que    f(3,2)=5

### Exemplo:

```
/* Para a linguagem C. */  
#include <stdio.h>  
#include <stdlib.h>  
int main ( )  
{ printf ("Teste!"); return 0; }
```

```
// Para a linguagem C++.  
#include <iostream>  
int main ( )  
{ std::cout << "Teste!" << std::endl;  
return 0; }
```

## 6-Manipulação de variáveis e constantes

A utilização de variáveis e constantes em linguagens de programação é inevitável. Na linguagem C as variáveis ou constantes possuem distinção quanto a grafia maiúscula ou minúscula. Algumas versões da linguagem C aceitam diferentes variações nos nomes das variáveis utilizadas (tais como começar com números por exemplo), porém estas variações devem ser evitadas visando a maior portabilidade do programa.

### 6.1-Variáveis

As variáveis utilizadas na linguagem C são apresentadas em diversas formas, entretanto, poderiam ser classificadas de forma sucinta entre os grupos apresentados a seguir:

- *char* - São compostas de um único caractere. Ocupa 1 byte de memória. Ex.: 0,1,Z,?...
- *string* - São compostas por uma cadeia de caracteres. Utilizam a variável *char*, como um vetor com n elementos declarados. Ex.: "Rua Setembro, 234"
- Números inteiros - São subdivididas em classes, conforme a Tabela 3.

**Tabela 4 – Números inteiros presentes na declaração de variáveis das Linguagens C e C++**

Especificador	BITS	Faixa (mínima)
int	16	-32.768 a +32.767
short [int]	16	"
unsigned [int]	16	0 a 65.535
unsigned short [int]	16	"
long [int]	32	-2.147.483.648 a 2.147.483.647
unsigned long [int]	32	0 a 4.294.967.295

- Valores com ponto flutuante - São os números reais, podem possuir simples ou dupla precisão e possuem no mínimo 32 bits para sua representação. Também apresentam diferentes valores, como os da Tabela 4.

**Tabela 5 – Números reais presentes na declaração de variáveis das Linguagens C e C++**

Especificador	BITS	Faixa (mínima)
float	32	-3,4.10 <sup>-38</sup> a +3,4.10 <sup>+38</sup>
double	64	-1,7.10 <sup>-308</sup> a +1,7.10 <sup>+308</sup>
long double	80	3,4.10 <sup>-4932</sup> a 1,1.10 <sup>+4932</sup>

- *enum* - Enumera os valores válidos de uma variável.
- *void* - Ocupam 0 bits, também conhecida como variável nula.
- Ponteiro - Não contém a informação, mas o endereço na memória onde a informação pode ser encontrada.

As variáveis podem ser de acesso global, onde todas as sub-rotinas do programa acessam seus valores, ou de acesso local a uma sub-rotina. Todas as variáveis que serão utilizadas devem ser declaradas com antecedência, variáveis que pertencem a mesma classe podem ser declaradas na mesma linha separadas apenas por vírgula. Após a declaração da variável é possível fornecer um valor inicial a mesma. Tem-se os exemplos de variáveis abaixo:

```
char Resposta='S';      float x,y,z;      long area=10, peso=30;
double =3.1234;         char x[12];      char a[3]="ABC";
```

enum tipo {dolar, real, libra}; enum tipo moeda; /\* No exemplo apresentado "tipo" não é uma variável e sim os tipos disponíveis e moeda é a variável que receberá estes.\*/



## 6.2-Constantes

É comum a utilização de constantes no programa, para tal basta incluir a palavra *const* no início da declaração, o tipo da variável e seu valor. Ex.:

```
const int min=0, max=10;  
const float pi=3.1415926;
```

É possível também definir um valor para um determinado nome, utiliza-se o comando “*#define*”. Ex.:

```
#define area 10; /* Toda ocorrência da palavra área será substituída por 10. */
```

## 6.3-Aritmética de pontos flutuantes

A maioria dos computadores operam baseados no conceito de *ponto flutuante*, estes são números que podem ser representados na forma:

$$\text{número} = \text{mantissa} . \text{base}^{\text{expoente}}$$

A mantissa pode ser representada por um número com *n* casas decimais que devem possuir um valor menor que o valor da base, o formato genérico destes números seria  $\pm 0.d_1d_2d_3d_4\dots$

O expoente pode se encontrar dentro de uma quantidade de valores que irá depender do computador que se encontra em uso. A representação mais comum do sistema de ponto flutuante é:

$$\text{número} = f(\text{base}, \text{número de dígitos da mantissa}, \text{expoente inferior}, \text{expoente superior})$$

Com estas restrições é notável que se um ponto flutuante for escolhido não será possível representar todos os números reais em um intervalo, pois a sequência dos números deverá obedecer a representação numérica escolhida para o sistema. Este fato é percebido quando operações que deveriam resultar em valores "redondos" não atingem seu objetivo, uma vez que a aritmética empregada se baseia no ponto flutuante da máquina em questão.

Tome-se um exemplo:

suponha um dispositivo com o sistema de ponto flutuante do tipo F(10,3,-5,5)

supondo uma operação do tipo  $x=1 / 3$

o novo valor de *x* será 0.333

se ocorrer uma multiplicação do valor de *x* por 3 teremos:  $x=0.999$  e não  $x=1$  (o correto)

## 6.4-Armazenamento de variáveis

É importante observar que as variáveis podem conter apenas um valor em cada instante do programa. A qualquer instante elas poderiam trocar seus valores, por exemplo:

1.  $x=1$ ;
2.  $y=2$ ;
3.  $x=4$ ;

No instante 1 *x* vale 1, porém no instante 3 seu valor é modificado para 4.

Esta situação deve ser considerada quando desejarmos “trocar” entre as variáveis seus valores, pois se não utilizarmos uma variável para armazenar temporariamente um dos valores ele irá se perder. Então se fosse necessário trocar o valor de *x* e *y* deveria ser feito com a forma:

1.  $\text{temp}=x$ ;
2.  $x=y$ ;
3.  $y=\text{temp}$ ;

O que impediria a perda do valor de *x* durante o processo.

## 7-Operações matemáticas

A linguagem C e C++ possui todas as operações aritméticas convencionais, e operações aritméticas extras em outras bibliotecas. A forma de apresentação das operações aritméticas na linguagem C e C++ difere um pouco daquele presente nas demais linguagens, tem-se os exemplos a seguir de algumas operações aritméticas:

```
x=y=0;      /* faz os valores de x e y iguais a zero */
x=1;        /* x recebe o valor numérico 1 */
x=x+1;      /* x recebe o valor anterior de x mais 1, 2 caso se tome a sequência acima */

x++;        /* faz x=x+1 */
++x;        /* faz x=x+1 */
x--;        /* faz x=x-1 */
--x;        /* faz x=x-1 */

total+=x;    /* faz total=total+x */
total-=x;    /* faz total=total-x */
total*=x;    /* faz total=total*x */
total/=x;    /* faz total=total/x */
total%=x;    /* faz total=resto da divisão de total/x */
total=x++;  /* faz total=x e posteriormente x=x+1 */
total=x--;  /* faz total=x e posteriormente x=x-1 */
total=++x;  /* faz x=x+1 e posteriormente total=x */
total=--x;  /* faz x=x-1 e posteriormente total=x */
```

Todas as expressões aritméticas são separadas apenas por parênteses. É permitido expressões tais como `x=total+(y=3)`; mas estas expressões devem ser evitadas visando manter um bom nível de portabilidade do programa.

É possível acompanhar a maioria das variáveis através das janelas de observação ou depuradores presentes nos diversos compiladores.

A utilização da biblioteca *math.h* ou equivalente possibilita a utilização de outras funções matemáticas, apresentadas a seguir:

*abs(x)* - apresenta o valor absoluto de um número inteiro x.

*cabs(z)* - valor absoluto do número complexo z.

*fabs(x)* - valor absoluto do número real x.

*labs(x)* - calcula o valor absoluto do long int x.

*acos(x)* - apresenta o arco-coseno do valor x.

*asin(x)* - apresenta o arco-seno do valor x.

*atan(x)* - apresenta o arco-tangente do valor x.

*atan2(y,x)* - apresenta o arco-tangente de y/x.

*cos(x)* - coseno de x.

*cosh(x)* - coseno hiperbólico de x.

*sin(x)* - seno de x.

*sinh(x)* - seno hiperbólico de x

*tan(x)* - tangente de x

*tanh(x)* - tangente hiperbólico de x

*hypot(x,y)* - calcula o valor da hipotenusa do triângulo retângulo com x e y como catetos.

*atof(palavra)* - converte a string palavra em um número real.

*floor(x)* - arredondamento para baixo do valor de x.

*ceil(x)* - arredondamento para cima do valor de x

$\exp(x)$  -  $e^x$ .  
 $fmod(x,y)$  - resto da divisão de  $x/y$ .  
 $frexp(x,expoente)$  - retorna o valor de  $m$  tal que  $x=m.2^{expoente}$   
 $ldexp(x,exp)$  - calcula  $x.2^{exp}$   
 $\log(x)$  -  $\ln(x)$   
 $\log10(x)$  -  $\log_{10}(x)$   
 $pow(x,y)$  -  $x^y$   
 $pow10(x)$  -  $10^x$   
 $modf(x,y)$  - separa o número  $x$  em inteiro e fração, armazena o inteiro em  $y$  e retorna a fração.  
 $poly(x,y,vetor)$  - cria um polinômio de grau  $y$  com o valor de  $x$  e os índices do vetor.  
 $\sqrt{x}$  -  $\sqrt{x}$

Existem ainda bibliotecas com referência a operações com números complexos (*complex.h*). A inclusão desta biblioteca permite a utilização de variáveis complexas no programa, definidas pelo nome de *complex*. As variáveis pertencentes a esta classe possuem dois números:

*complex z=complex (x,y); /\* sendo os valores de x e y números reais, correspondentes a parte real e complexa respectivamente. \*/*

As novas funções adicionadas serão:

*conj(z)* - complexo conjugado de  $z$

*imag(z)* - retorna a parte imaginária do número complexo  $z$

*real(z)* - retorna a parte real do número complexo  $z$

*arg(z)* - fornece o ângulo do número complexo  $z$

*norm (z)* - fornece o módulo do número complexo  $z$  ( $z = \sqrt{x^2 + y^2}$ )

*polar(mag,angulo)* - fornece o número complexo com a magnitude e o ângulo fornecidos

## 8-Comando de E/S de dados via teclado e monitor

A entrada e saída de dados é imprescindível em todos os programas. O primeiro programa proposto já apresentava a saída de dados, a frase “Teste!” era escrita no monitor. Os dois comandos de saída de dados utilizados, para o C e para o C++, são descritos a seguir.

### 8.1-Comando “printf” (C)

O comando de saída de dados utilizado pelo ANSI-C, este comando apresenta entre parênteses todo o texto que será impresso, incluindo variáveis. O formato deste comando é:

***printf (“\comando Texto %tipo de variável”,variável);***

Nota-se que o texto é fielmente representado e há presença de comandos e tipos de variáveis a serem mostradas, sendo alguns dos comandos:

<code>\n</code>	nova linha	<code>\f</code>	avanço do papel da impressora
<code>\o</code>	fim de string	<code>\b</code>	retrocesso do carro da impressora
<code>\a</code>	beep	<code>\t</code>	tabulação
<code>\'</code>	impressão do caractere a seguir		
<code>\"</code>	apreça aspas duplas		
<code>\\</code>	insere barra invertida		

Os tipos de variáveis a serem apresentadas são:

<code>%x</code>	hexadecimal	<code>%d</code>	inteiro decimal
<code>%c</code>	caractere	<code>%f</code>	real
<code>%s</code>	string	<code>%o</code>	octal

sendo possível ainda utilizar parâmetros da apresentação dos números reais:

***%+AB.Cf*** onde:

+ - o sinal do número é apresentado.

A - poderá ser o número 0 que indica o preenchimento com zero do espaço restante.

B - espaço reservado para escrever o número real.

C - número de casas decimais a ser apresentada.

Exemplos:

```
printf ("Este é\num teste!");      printf ("\nX=%d \ne Y=%+3.2f",X,Y);
printf("Pode-se encadear %s em uma saída","diversas frases");
```

## 8.2-Comando “std::cout” (C++)

O comando de saída utilizado geralmente no C++ é o comando **cout**, a principal diferença deste comando está na facilidade com que os comandos e as variáveis são apresentados. Todos os comandos e variáveis são separados por << fornecendo a idéia de saída de dados. Observa-se que é necessário referenciar o uso da biblioteca padrão utilizando “std::” antes do comando **cout**. Não é necessário expressar a forma da variável, ela é automaticamente identificada e apresentada. A estrutura fundamental seria:

```
std::cout<<"texto1"<<std::XXX<<...variável1<<"texto2"<<variável2...<<std::endl;
```

Nota-se, porém, ao final a expressão *std::endl* insere uma nova linha e esvazia o buffer da cadeia de caracteres impressas. Ela não é necessária na maioria dos casos, mas torna mais adequada a orientação da saída dos dados.

Tem-se então para alguns dos exemplos apresentados no comando printf:

```
std::cout <<“Este é\`num teste!”<<endl;
std::cout <<“\`nX=”<<X<<“\`ne Y=”<<Y<<std::endl;
```

Nota-se porém que é comum utilizar os recursos apresentados pelo comando **cout** os quais facilitam a saída de dados. Existem ainda modos avançados de proceder com o formato da saída do comando **cout**. Para tais recursos a biblioteca utilizada deverá ser a *iostream*, *iomanip*, dentre outras. Pode-se converter a forma dos valores:

- **std::dec** O número inteiro é apresentado em decimal.
- **std::oct** O número inteiro é apresentado em octal.
- **std::hex** O número inteiro é apresentado em hexadecimal.

Por exemplo:

```
#include <iostream>
int main()
{
    // Exemplo da conversao de numeros.

    long int x = 123456789;
    std::cout << "Temos o valor 123456789 como:\`n" << std::endl;

    std::cout << "decimal:    " << std::dec << x << std::endl;
    std::cout << "octal:      " << std::oct << x << std::endl;
    std::cout << "hexadecimal: " << std::hex << x << std::endl ;

    return 0;}
```

A formatação numérica também é possível, eis alguns exemplos:

- **std::left** O número é apresentado a esquerda (left).
- **std::showpos** O sinal (+) é apresentado.
- **std::scientific** Formato científico.
- **std::unsetf** Utilizado para desativar os itens ativados por setf.
- **std::setw(20)** Espaço de 20 caracteres para representar o número, biblioteca “*iomanip*”.
- **std::setprecision(3)** Apenas 3 dígitos após o ponto decimal são apresentados, biblioteca “*iomanip*”.

Dentre outros, que dependem de uso de bibliotecas específicas.

Estes são alguns dos exemplos possíveis para a formatação dos números.

### 8.3- Comando *puts*

Quando se deseja a saída apenas de uma cadeia de caracteres o comando *puts* é mais otimizado que seu equivalente *printf*, porém este comando aceita apenas uma *string* na saída não podendo ser utilizado para mostrar variáveis junto ao comando. O comando requer a biblioteca *string.h* e sua estrutura fundamental é:

***puts ("cadeia de caracteres");***

Vale observar que esta instrução reconhece os mesmos comandos especiais da função *printf*. O exemplo a seguir imprime na tela a string definida entre as aspas.

```
#include <string.h>
void main()
{ puts("Este é um teste.\n"); }
```

Existe ainda o comando *putchar* que opera de forma semelhante ao comando *puts* porém insere apenas um caractere na tela.

### 8.4-Comando *scanf* (C)

Para a entrada de dados via teclado o comando *scanf* é utilizado pela linguagem C, este comando se encontra na biblioteca *stdio.h*, a função irá devolver o número de itens de dados que forem lidos as respectivas variáveis.

***scanf(""%<opcional tamanho máximo>tipo de dado",&variável a receber o dado lido);***

os tipos de dados são:

%c	caractere	%o	octal
%d	inteiro decimal com sinal	%s	string
%i	inteiro decimal sem sinal	%x	hexadecimal
%e	número real	%p	ponteiro
%f	ponto flutuante	%u	inteiro sem sinal
%g	usa o mais curto entre %e ou %f		

Pode-se opcionalmente entre o % e o tipo de valor lido definir um tamanho máximo limitante para leitura.

Um problema que existe com a utilização deste comando é que ele não consegue ler cadeia de caracteres que possuem um espaço entre as palavras, neste caso é necessário utilizar outro comando, o comando *gets* desempenha esta função e será abordado logo a seguir. O esvaziamento do *buffer* (memória) do teclado também pode ser uma opção.

#### Exemplo:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{ int i;
  printf("\ni=");          /* É sempre aconselhável indicar qual variável se deseja. */
```

```

scanf("%d", &i);    /* Leitura do valor de i pelo teclado. */
i++;    /* Cálculo do sucessor de i (i+1). */
printf("O sucessor de i é %d",i); } /* Saída do sucessor de i. */

```

Pode-se ler mais de um número, basta indicar com \* o local onde será desprezado o caractere em questão, como no exemplo:

```

...    scanf ("%f*c%f",%a,&b);    ...

```

## 8.5-Comando “cin” (C++)

Para a versão C++ da *linguagem C* o comando *cin* é utilizado, este comando faz parte da biblioteca *iostream* e é facilmente utilizado, basta acrescentar ao comando a variável que irá receber o valor digitado. Não há necessidade de indicar qual o tipo de variável que será lida.

***std::cin >>variável;***

Pode-se entrar com mais de uma variável, bastando para tal que os delimitadores “>>” sejam novamente repetidos seguidos da nova variável.

### Exemplo:

```

#include <iostream>
void main()
{ int i;
std::cout<<"ni=";    // É sempre aconselhável indicar qual valor se deseja.
std::cin>>i;    // Leitura do valor de i pelo teclado.
i++;    // Cálculo do sucessor de i (i+1).
std::cout<<"O sucessor de i é:"<<i; }    // Saída do sucessor de i.

```

## 8.6- Comando gets

O comando *gets* e suas variantes serve para entrada de caracteres, existem ainda diversas variantes deste comando, sendo que estes comandos são utilizados na forma como são apresentados a seguir:

```

getchar()    lê um caractere após a tecla <enter>.
getche()    lê um caractere sem aguardar a tecla <enter>, o caractere é mostrado na tela.
getch()    lê um caractere sem aguardar a tecla <enter>, o caractere não é mostrado na tela.

```

A estrutura fundamental é apresentada a seguir:

***gets(<variável>);***

A função é mais eficiente que a função *scanf* pois pode assimilar uma string que possui espaço em sua formação. Um exemplo da utilização da função *gets* é apresentada a seguir, onde uma sequência de caracteres de nomes é lida e posteriormente apresentada em uma frase.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{char nome[20];
puts("Entre com o nome da pessoa:");
gets(nome);
printf("O nome entrado foi %s.\n",nome);}
```

A utilização das variantes do comando *gets* possuem diversas utilidades, entre elas aguardar até que alguma tecla seja pressionada.



## 9-Operadores

Na *linguagem C* qualquer valor diferente de zero é considerado verdadeiro, para tornar comum a maioria dos programas supõem o valor “0” como falso e o “1” como verdadeiro. A álgebra de Boole será utilizada amplamente nestas análises. As comparações entre elementos podem envolver as desigualdades ou a igualdade, entre todos os diferentes tipos de variáveis. Is comparadores são:

>      Maior que  
<      Menor que  
>=    Maior ou igual  
<=    Menor ou igual  
!=     Diferente de  
==    Igual a

Pode-se comparar variáveis ou constantes entre si.

Na *linguagem C* as operações fundamentais da álgebra de Boole são substituídas pelos termos apresentados a seguir:

conector E     :      &&  
conector OU    :      ||  
conector NÃO:     !

Supondo F=Falso e V=Verdadeiro podemos compor a Tabela 6.

**Tabela 6 – Exemplo de tabela verdade na Linguagem C/C++**

a	b	a&&b	a  b	!a
F	F	F	F	V
V	F	F	V	F
F	V	F	V	V
V	V	V	V	F

A ordem de precedência é: **NÃO** seguido de **E** finalmente **OU**. Pode-se, no entanto, mudar esta ordem com a utilização de parênteses para a composição da operação.

### 9.1-Operadores bit a bit

É possível realizar operações em dados do tipo *inteiro* e *chart* bit a bit sendo estas operações responsáveis por deslocamentos de bits e operações lógicas. Estas operações se aplicam apenas a estes tipos de dados, sendo impossível seu emprego em diferentes dados como os números reais. Tais operações se encontram presentes na *linguagem C* devido a utilização desta linguagem como substituta ao *Assembly*. Os operadores utilizados se encontram especificados a seguir:

<u>Operador</u>	<u>Significado</u>
&	E
	OU
^	OU EXCLUSIVO
~	COMPLEMENTO DE 1
<<	DESLOCAMENTO DE BITS P/ ESQUERDA (multiplica por 2)
>>	DESLOCAMENTO DE BITS P/ DIREITA (divide por 2)

### Exemplos:

01111111 ^ 01111000 = 00000111

```
x=7   0000 0111
x<<1  0000 1110  (x=14)
x<<3  0111 0000  (x=112)
x>>1  0011 1000  (x=56)
```

### Exemplo:

```
/* Deslocamento de bits, em C. */
#include <stdio.h>
#include <stdlib.h>
int main ()
{ int i;
  i=1;
  printf("\nO valor de i=%d",i);
  i=i<<1; /* Desloca 1 bit para esquerda, o que é o mesmo que multiplicar por 2. */
  printf("\nO novo valor de i=%d",i);
  return 0;}
```

## **9.2-O operador “vírgula”**

Utilizado para encadear diversas instruções o operador vírgula é muito utilizado por alguns programadores em C. É possível realizar diversas expressões em uma única linha, tornando o código mais sucinto. Todas as operações realizadas a esquerda da vírgula são executadas independentemente e apenas o último valor dentre as operações realizadas será atribuído a variável em questão.

### Exemplos:

x=(y=2,y+4); // Faz-se y=2 primeiramente, posteriormente x=y+4, com o novo valor de y.

equivale a:  
y=2; x=y+4;

a=(b=4,5+d); // Faz-se b=4 e posteriormente a=5+b.  
equivale a:  
b=4; a=5+d;

x=(y=2,z=2\*y,z+4); // Três sequências de operações realizadas sucessivamente.  
equivale a:  
y=2; z=2\*y; x=z+4;

## **9.3-O operador “?”**

A linguagem C possui o operador ? que atua como tomada de decisão quando é necessário atribuir um valor a uma determinada variável. Têm-se a estrutura fundamental apresentada a seguir:

**Variável=Condição?Expressão1:Expressão2;**

A *Variável* irá assumir o valor da *Expressão1* se a *Condição* for satisfeita, caso contrário a variável assumirá o valor da *Expressão2*. Tome-se o exemplo a seguir:

```
x=2;           /* x recebe o valor 2. */
y=x>9?x:0;     /* Como x>9 é Falso y receberá o valor 0. */
```

#### Exemplo:

```
/* Utilizando a Linguagem C. */
#include <stdio.h>
#include <math.h>
void main ()
{
    float x,y;           /* Cria as variáveis x e y reais. */
    printf("\nx=");
    scanf("%f",&x);       /* Lê o valor da variável x. */
    y=x>0?sqrt(x):0;     /* Caso x seja maior que zero y recebe o valor de raiz
quadrada
                                de x, senão y recebe 0. */
    printf("\nO valor de y é: %.4f",y); /* Saída: valor de y. */
}
```

-----X-----

```
// Utilizando a Linguagem C++.
#include <iostream>
#include <math.h>
void main ()
{
    float x,y;           // Cria as variáveis x e y reais.
    std::cout<<"\nx=";
    std::cin>>x;         // Lê o valor da variável x.
    y=x>0?sqrt(x):0;
    // Caso x seja maior que zero y recebe o valor de raiz quadrada de x, senão y recebe 0.

    std::cout<<"\nO valor de y é:"<<y; // Saída: valor de y.
}
```

## 10-Diretiva “#define”

A linguagem C e C++ possuem um modo de definir constantes ou expressões que possui poderosos recursos para o programador, a instrução *#define*. Esta diretiva possui uma estrutura que substitui tudo que está em uma cadeia pelo conteúdo de uma segunda cadeia, na forma apresentada a seguir:

**#define cadeia1 cadeia2**

É possível substituir o valor de uma constante ou mesmo de uma expressão, pois assim que a constante ou expressão de *cadeia1* surgir no programa principal será substituído pelo conteúdo da *cadeia2*. As aplicações para este recurso são das mais vastas. A diretiva *#define* deve se encontrar no início do programa, junto a citação das bibliotecas utilizadas para o programa. É possível encadear os comandos *#define*, isto é uma nova diretiva pode fazer uso de uma diretiva anteriormente declarada.

### Exemplos:

```
...#define x 10      /* A letra x será substituída pelo número 10. */...
...#define cubo(x) x*x*x /* A ocorrência de qualquer expressão no formato cubo(x) será
                        substituída por x*x*x */...
```

```
...#define e 1.6e-19 /* É definida a carga do elétron (ou próton) [Coulomb]. */
#define q n*e      /* Utiliza-se a definição anterior para determinar a carga elétrica. */...
```

### Programa Exemplo em C:

*/\* O programa a seguir determina a força que uma carga elétrica aplica sobre a outra quando estas se encontram a uma distância d entre si. ANSI-C.\*/*

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

```
#include <stdlib.h>
```

```
#define k 9e9      /* Definição do valor da constante eletrostática [N.m2/C2]. */
```

```
#define f(q1,q2,d) k*q1*q2/(d*d) /* Substitui a fórmula da força quando necessário.*/
```

```
int main ()
```

```
{
```

*/\* As variáveis q1 e q2 se referem aos valores das cargas a variável d ao valor da distância. \*/*

```
float q1, q2, d;
```

```
    /* Leitura dos valores a serem entrados. */
```

```
    printf("\nEntre com o valor da carga 1:");
```

```
    scanf("%f",&q1);
```

```
    printf("\nEntre com o valor da carga 2:");
```

```
    scanf("%f",&q2);
```

```
    printf("\nEntre com o valor da distância:"); scanf("%f",&d);
```

*/\* Saída dos resultados utilizando os parâmetros da diretiva #define.\*/*

```
    printf("\nA força entre as cargas é de %8.3f N\n",f(q1,q2,d));
```

```
    return 0;}
```

## 11-Tomadas de decisão e LOOP's

Durante a execução de um programa uma ou mais condições podem fazer com que o programa tenha mais que um caminho possível a ser percorrido. A tomada de decisão é representada sob a forma de diversos comandos. Existem ainda casos onde os comandos devem ser repetidos até que uma ou mais condições sejam satisfeitas, este ato é conhecido como *loop* na programação. A linguagem C e C++, assim como as demais linguagens presentes nos computadores permite que tais situações sejam realizadas com uma certa facilidade, para tal utiliza diversos comandos visando suportar tais problemas. Essas situações já foram observadas na fig. 2.

### 11.1-Comando “if-else” (se-senão)

Este é um comando de tomada de decisão simples. Caso uma determinada expressão seja satisfeita um grupo de comandos será executado, senão um outro grupo de comandos poderá ser executado alternativamente. A estrutura fundamental deste comando é:

***if (expressão) {Comandos1} else {Comandos2};***

Tomando a estrutura apresentada: caso a expressão seja diferente de zero (verdadeira) a sequência de instruções contidas em *Comandos1* será executada caso a expressão seja zero (falsa) a sequência a ser executada será a contida em *Comandos2*.

A presença do *else* é opcional, pode-se ter uma situação onde é desejado que apenas uma situação seja satisfeita, caso contrário nada será realizado, para tal basta eliminar o comando *else* e os comandos referentes a sua citação. A expressão pode envolver números ou caracteres em comparações utilizando as expressões lógicas (vide item 9).

#### Exemplo:

```
if (x>0 || y<10) { x++; y--; } else { x=0; y++; };
```

/\* Na situação apresentada se o valor de x for maior que 0 ou o de y menor que 10 então x receberá o valor anterior de x mais um e y será subtraído de um, senão x receberá zero e y será acrescido de um. \*/

É possível ainda entre os diversos comandos que existem dentro das chaves do *if* existir outros *if* e assim sucessivamente, criando um encadeamento da decisão. Tal situação pode ser observada na fig. 8.

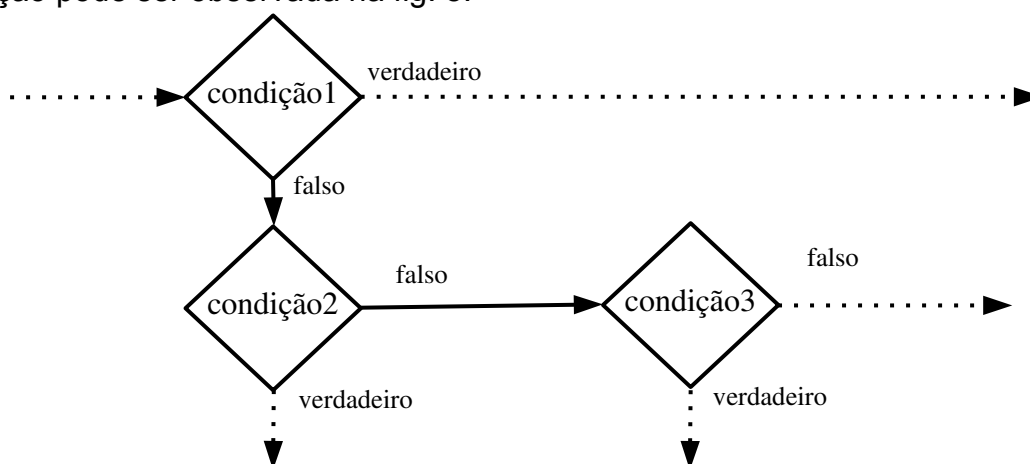


Figura 8 – Exemplo de encadeamento de expressões utilizando if.

Tal instrução é muito importante em diversas aplicações envolvendo cálculos matemáticos e manipulação de dados. A seguir é apresentado um programa que calcula uma equação do 2º grau utilizando a fórmula de Baskara, tal programa utiliza os recursos apresentados até o momento e pode distinguir entre os três tipos de soluções possíveis (duas raízes distintas  $\in \mathbb{R}$ , duas raízes  $\in \mathbb{R}$  de igual valor ou nenhuma raiz  $\in \mathbb{R}$ . }

```
/* Este programa calcula a eq. de grau 2 utilizando a formula de Baskara, para ANSI-C. */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define quad(x) x*x
#define delta(a,b,c) quad(b)-4*a*c
int main ( )
{ float a,b,c,delta,x1,x2;

/* Entrada pelo teclado dos termos da eq. de grau 2. */
printf("Entre com o valor de a:"); scanf("%f",&a);
printf("Entre com o valor de b:"); scanf("%f",&b);
printf("Entre com o valor de c:"); scanf("%f",&c);
delta=delta(a,b,c);
/* Determinação de qual as raízes são solução para o problema, sendo neste */
/* caso particular estudados apenas os casos de raízes reais. */
    if (delta<0)
        printf("\nNao existe raiz real!");
    else
        { if (delta>0) {x1=(-b+sqrt(delta))/(2*a);
                        x2=(-b-sqrt(delta))/(2*a);
                        printf("X1=%+10.4f e X2=%+10.4f",x1,x2);}
          else
            {x1=(-b+sqrt(delta))/(2*a);
              printf("\nX1=X2=%+10.4f",x1);}}
return 0;}
```

-----X-----

```
// Este programa calcula a eq. de grau 2 utilizando a formula de Baskara, para C++.
#include <iostream>
#include <iomanip>
#include <math.h>
#define quad(x) x*x
#define delta(a,b,c) quad(b)-4*a*c
int main ( )
{ float a,b,c,delta,x1,x2;

// Entrada pelo teclado dos termos da eq. de grau 2.
std::cout<<"Entre com o valor de a:"; std::cin>>a;
std::cout<<"Entre com o valor de b:"; std::cin>>b;
std::cout<<"Entre com o valor de c:"; std::cin>>c;
delta=delta(a,b,c);
```

```
// Determinação de qual as raízes são solução para o problema, sendo neste
```

```
// caso particular estudados apenas os casos de raízes reais.
if (delta<0)
std::cout<<"\nNao existe raiz real!\n";
else
{ if (delta>0) {x1=(-b+sqrt(delta))/(2*a);
x2=(-b-sqrt(delta))/(2*a);
// É estabelecido 4 dígitos de precisão a serem mostrados na saída.
std::cout<<std::setprecision(4)<<"X1="<<x1<<" e X2="<<x2<<"\n";}
else
{x1=-b/(2*a);
std::cout<<"\nX1=X2="<<x1<<"\n";}}

return 0;}
```

## 11.2-O laço de “for” (para)

Em alguns casos é necessária a repetição sistemática de uma sequência de instruções, para tal existem diversos recursos nas linguagens de programação. As repetições podem depender de uma determinada situação para serem interrompidas, ou seu número de repetições ser conhecido. Na *linguagem C* um dos laços de repetição mais utilizados é o laço de **for**, este laço possui uma inicialização, uma condição e uma equação responsável por sua modificação em direção ao valor desejado.

**for (inicialização do contador; condição de parada; atualização do contador)  
{ Instruções a serem repetidas }**

Exemplo:

```
for (x=1;x<=50;x++) /* x se inicia com o valor 1, as instruções presentes no
delimitador que se segue serão repetidas até que  $x \leq 50$ , a cada
repetição  $x=x+1$ . */
```

Este *loop* é de grande utilidade para a programação, uma vez que comandos podem ser repetidos até que uma determinada condição seja alcançada.

Estas operações são utilizadas em diversas situações como cálculos iterativos, operações acumulativas, etc. Um exemplo é apresentado a seguir com o cálculo de uma P.A. (progressão aritmética).

*/\* Calculo de uma  $PA=A1+...+A2$  , sendo fornecido o intervalo e a razão, para ANSI C. \*/*

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{int a1,a2;
float pa=0,r,s; /* O valor de pa é inicializado como zero. */

/* Entrada dos valores da PA. */
printf ("Entre com o intervalo da PA (a1 a2):");
scanf("%d %d",&a1,&a2); /* São lidos dois valores simultaneamente, separados
por espaço. */
printf ("Entre com a razao (r):"); scanf("%f",&r);
```

```

/* Calculo do valor da PA */
for (s=a1;s<=a2;s+=r)      /* Comando de repetição, loop de for. */
pa+=s;      /* Esta é a instrução repetida, até que  $s \leq a2$ . */
printf("PA=%12.4f",pa);
return 0;}

```

-----X-----

// Calculo de uma  $PA=A1+...+A2$  , sendo fornecido o intervalo e a razão, para C++.

```

#include <iostream>
int main ()
{int a1,a2;
float pa=0,r,s;          // O valor de pa é inicializado como zero.

// Entrada dos valores da PA.
std::cout << "Entre com o intervalo da PA (a1 a2):";
std::cin >> a1 >> a2; //São lidos dois valores simultaneamente, separados por
// espaço.
std::cout << "Entre com a razao (r):";      std::cin >> r;

// Calculo do valor da PA
for (s=a1;s<=a2;s+=r)      // Comando de repetição, loop de for.
pa+=s;      // Esta é a instrução repetida, até que  $s \leq a2$ .
std::cout << "PA=" << pa << std::endl;
return 0;}

```

### 11.3-O laço de “while” (enquanto)

Em algumas situações a sequência de instruções que se deseja repetir terá seu número de repetições dependente de uma ou mais condições. neste caso durante a repetição das instruções esta condição deverá ser satisfeita em um momento, caso contrário ocorrerá uma repetição infinita destas instruções.

**while (condição) { Instruções a serem repetidas. }**

Exemplo:

```
while (x<8 || y>6)    { x+=2;      y+=4; }
```

Este *loop* apresenta a particularidade de verificar no início da repetição se a condição já foi satisfeita, caso esta condição não tenha sido satisfeita ele então irá repetir as instruções contidas dentro dos delimitadores apresentados a seguir.

```

/* O programa mostra o arcosseno de ângulos (graus) de um determinado intervalo ANSI-
C */
#include <stdio.h>
#include <math.h>
int main ()
{const float pi=3.1415926;
float x=0, b;

// Leitura do final do intervalo, o início será o valor zero. */
printf ("\nEntre com o valor final do seno(x)=");      scanf("%f",&b);

```



```

/* loop de repetição, onde o ângulo é calculado e convertido de radianos-graus. */
while (x<b)
{ printf ("\narcoseno(%.3f)=%.5f", x, asin(x)*180/pi);
/* Incremento de 0.1 para o novo valor, até atingir o patamar desejado. */
  x+=0.1; }
return 0;}

```

-----X-----

```

// O programa mostra o arcoseno de ângulos (graus) de um determinado intervalo C++
#include <iostream>
#include <math.h>
int main ( )
{const float pi=3.1415926;
float x=0, b;

```

```

// Leitura do final do intervalo, o início será o valor zero.
std::cout<<"\nEntre com o valor final do seno(x)=";    std::cin>>b;

```

```

// loop de repetição, onde o ângulo é calculado e convertido de radianos-graus.
while (x<=b)
{ std::cout<<"\narcoseno("<x<<")="<< asin(x)*180/pi;
// Incremento de 0.1 para o novo valor, até atingir o patamar desejado.
  x+=0.1; }
return 0;}

```

#### 11.4-O laço “do { } while” (faça { } enquanto)

De forma semelhante ao loop “while” este loop diferencia dos demais pois ele realiza o teste após a primeira execução das instruções, portanto ao menos uma vez todas as instruções a serem repetidas serão executadas.

**do { Instruções a serem repetidas. } while (condição);**

Sua utilização é justificável quando há necessidade de executar ao menos uma vez as instruções.

```

/* Exemplo do cálculo médio de uma sequência de números ANSI-C */
#include <stdio.h>

int main ( )
{ float x, soma=0, n=0;
/* Loop com a repetição da leitura, soma dos valores lidos e número de valores lidos(n). */
do { printf("\nEntre com o valor do número (0 para terminar):");    scanf("%f",&x);
    if (x){soma+=x;    n++;}}
while (x);

/* Saída de resultados. */
printf("\nA média é:%.4f",soma/n);
return 0;}

```

-----X-----

```
// Exemplo do cálculo médio de uma sequência de números C++
#include <iostream>

int main ( )
{ float x, soma=0, n=0;

// Loop com a repetição da leitura, soma dos valores lidos e número de valores lidos(n).
do { std::cout<<"\nEntre com o valor do número (0 para terminar):";      std::cin>>x;
    if (x){soma+=x;      n++;}}
while (x);

/* Saída de resultados. */
std::cout<<"\nA média é:"<<soma/n;
return 0;}
```

#### 11.4.1-Validação de dados

É comum que a leitura de determinados dados fornecidos pelo usuário possam ser validados (verificados) para então serem utilizados no programa. O comando *do-while* se presta a este serviço de forma eficiente, basta notar que as propriedades apresentadas por este comando são adequadas para esta tarefa. O comando é executado ao menos uma vez, e a leitura deve ocorrer ao menos uma vez para possibilitar a validação da informação. Uma vez lidos os valores eles podem ser verificados pelo comando *while*, confirmando que se encontram dentro da região possível de seus valores, e então utilizados no programa. Um exemplo de validação é apresentado a seguir em um programa que irá calcular a soma da raiz quadrada de *n* valores reais e imprimir o resultado, supondo que a quantidade de valores será fornecida pelo usuário.

```
/* Exemplo do cálculo da soma da raiz quadrada dos números fornecidos. ANSI-C*/
#include <stdio.h>
#include <math.h>

int main ()
{ float x,soma=0;
  int n,i;

printf ("\nEntre com o número de elementos a serem lidos:");
scanf ("%d",&n);

/* Loop para leitura dos elementos. */
for (i=1;i<=n;i++)
{
/* Loop de validação do i-ésimo número fornecido (x deve ser maior ou igual a zero). */
do {printf("Entre com o elemento %d:",i); scanf("%f",&x);} while (x<0);
/* Cálculo da soma das raízes quadradas. */
soma+=sqrt(x);}

/* Saída de dados. */
printf("A soma das raízes é:%.4f",soma);
return 0;}
```

-----X-----

```
// Exemplo do cálculo da soma da raiz quadrada dos números fornecidos. C++
#include <iostream>
#include <math.h>

int main ()
{ float x,soma=0;
  int n,i;

  std::cout<<"\nEntre com o número de elementos a serem lidos:";
  std::cin>>n;

  /* Loop para leitura dos elementos. */
  for (i=1;i<=n;i++)
  {
    /* Loop de validação do i-ésimo número fornecido (x deve ser maior ou igual a zero). */
    do {std::cout<<"Entre com o elemento "<<i<<":"; std::cin>>x;} while (x<0);
    /* Cálculo da soma das raízes quadradas. */
    soma+=sqrt(x);}

  /* Saída de dados. */
  std::cout<<"A soma das raízes é:"<<soma<<std::endl;
  return 0; }
```

#### 11.4.2-Pausa na tela.

Em alguns momentos da programação pode ser necessário causar uma pausa na tela, alguns compiladores se encontram em um ambiente diferente daquele onde é executado o programa sendo necessário algum artifício para possibilitar a pausa da tela, até que alguma tecla seja pressionada.

O loop de *do-while* novamente se presta de forma eficiente a este trabalho. Para que seja possível uma pausa na tela deve-se recorrer a algum artifício que irá capturar uma tecla, tal artifício existe na maioria dos compiladores C, sendo necessária a utilização de um comando que capture apenas uma tecla do teclado, tal comando existe na biblioteca *conio.h* de alguns compiladores antigos é a função conhecida como *getch()* / *getche()*, primeira apenas captura o caractere e não o escreve, enquanto que a segunda captura o caractere e o "ecoa" na tela. Um exemplo de sua utilização é fornecido a seguir.

```
/* Exemplo de uma pausa na tela. ANSI-C */
# include <conio.h>
# include <stdio.h>
{ ... char pausa='\0';
  printf ("\n\nAguardando que uma tecla seja pressionada...");
  do { pausa=getch();} while (pausa=='\0');} ... }
```

-----X-----

```

/* Exemplo de uma pausa na tela. C++ */
#include <conio.h>
#include <iostream.h>
{ ... char pausa='\0';
std::cout<<"\n\nAguardando que uma tecla seja pressionada..."<<std::endl;
do { pausa=getch();} while (pausa=='\0'); ... }

```

## 11.5-Instruções de controle de loops

### 11.5.1-Instrução break

Se durante a execução do *loop* for necessária a interrupção dele por qualquer motivo não programado na sequência natural das instruções a instrução *break* irá realizar a saída forçada do *loop*. Após a saída o programa continuará na instrução seguinte ao *loop*. O exemplo a seguir apresenta uma parada forçada. É calculada a média da potência aparente de um grupo de cargas, como a potência não pode possuir valores negativos (ou a carga estará gerando energia) caso isto ocorra o processo será interrompido pelo *break*, fornecendo a média apenas dos valores válidos fornecidos.

```

/* Programa que calcula a média da potência aparente de um grupo de cargas ANSI-C */
#include <stdio.h>
int main ( )
{ float v, i, p=0, media;
  int contador, n, m=0;
  /* Leitura do número de cargas */
  printf ("\nEntre com o número de cargas:"); scanf ("%d",&n);
  /* Leitura dos valores de tensão e corrente das cargas. */
  for (contador=1; contador<=n; contador++)
  { printf ("\nValor da corrente da carga (A):"); scanf ("%f",&i);
    printf ("Valor da tensão da carga (V):"); scanf ("%f",&v);
    /* Se a carga fornecer potência (P<0) o loop quebrado, desconsiderando inclusive o
    último valor válido fornecido */
    if (i*v<0) break; p+=i*v; m++;}
  /* Cálculo da média das potências aparentes. */
  media=p/m;
  /* Saída do resultado. */
  printf ("P=%.4f V.A",media);
  return 0;}

```

Observemos que no código acima a instrução *break* ocorre antes do cálculo acumulativo do valor *p*, dessa forma o valor com a potência negativa é desconsiderado, assim como o do número de valores fornecidos (*m*).

### 11.5.2-Instrução continue

Semelhante a instrução *break* a instrução *continue* interrompe a sequência de instruções a serem repetidas, porém não ocorre a saída do *loop* e sim a desconsideração das instruções seguintes ao *continue* dentro do *loop*. Pode-se dizer que caso seja necessária uma invalidação temporária das instruções basta incluir a função *continue* e

caso se deseje interromper definitivamente o *loop* utiliza-se a função *break*. Tomando o programa acima e substituindo a instrução *break* por *continue* o programa irá continuar a realizar a leitura das demais cargas citadas (e não parar como acontece com a instrução *break*), porém a carga com valor lido inválido será desprezada das operações. Vide o código a seguir:

```
/* Programa que calcula a média da potência aparente de um grupo de cargas ANSI-C */
#include <stdio.h>
int main ( )
{ float v, i, p=0, media;
  int contador, n, m=0;
/* Leitura do número de cargas */
  printf ("\nEntre com o número de cargas:"); scanf ("%d",&n);
/* Leitura dos valores de tensão e corrente das cargas. */
  for (contador=1; contador<=n; contador++)
  { printf ("\n\nValor da corrente da carga (A):"); scanf ("%f",&i);
    printf ("Valor da tensão da carga (V):"); scanf ("%f",&v);
/* Se a carga fornecer potência (P<0) o loop quebrado, desconsiderando inclusive o
último valor válido fornecido */
    if (i*v<0) continue; p+=i*v; m++;}
/* Cálculo da média das potências aparentes. */
media=p/m;
/* Saída do resultado. */
printf ("P=%.4f V.A",media);
return 0;}
```

### 11.6-Comando “switch”

Na linguagem C/C++ é possível executar um comando de múltiplas seleções, onde o caminho a ser tomado pelo processo irá depender de um valor obtido em uma expressão. A estrutura deste comando é apresentada a seguir:

```
switch (expressão)
{case valor1:
    instruções a serem executadas caso o valor1 seja o valor da expressão;
    break;          /* Parada das instruções, opcional.
case valor2:
    instruções a serem executadas caso o valor2 seja o valor da expressão;
    break;          /* Parada das instruções, opcional.
...default:
    instruções a serem executadas caso o valor da expressão não esteja acima;
...}
```

Nota-se que as comparações utilizadas pelo comando *switch* são de igualdade e não é possível existir dois valores idênticos nas funções *case*. Caso os valores apresentados em *case* não correspondam a nenhuma das opções então os comandos de *default* serão executados. A expressão pode ter como resultado números inteiros ou caracteres. Este comando é utilizado quando é necessário executar uma série de comandos a partir de uma determinada entrada. A presença do comando *break* não é obrigatória, porém se esta for omitida o comando irá continuar a processar as instruções

que se encontram a seguir. A sequência pode ser substituída por uma série de comandos *if*, obtendo-se o mesmo resultado.

Exemplo:

```
...switch(a)
{
    case 1:
        b=5*a;
    case 2:
    case 3:
        c=3*a;
        break;
    default:
        d=3*a;}...
```

Nota-se que no exemplo apresentado se o valor de *a* for 1 irá ser executado o comando *b=5\*a*; e o comando *c=3\*a*; se o valor de *a* for 2 ou 3 o comando executado será o mesmo (*c=3\*a*) e se o valor de *a* não for nenhum destes valores a expressão *d=3\*a*; será executada.

No exemplo a seguir uma variável (*x*) é lida e são apresentadas 4 opções de operações com esse número: sua raiz quadrada, seu quadrado, seu dobro ou manter seu valor original para uma variável *y*.

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

int main()
{
    float x,y;
    int resp;
    printf("Entre com o valor de x:");
    scanf("%f",&x); //Leitura do valor de x.
    printf("Qual operacao voce deseja?\n");
    printf("1-raiz quadrada\n2-quadrado\n3-dobro\nQualquer outro valor mantem o\nnumero\n");
    printf("Resposta:"); scanf("%d",&resp);
    switch(resp)
    {
        case 1: //Primeira situacao (raiz quadrada).
            y=sqrt(x);
            break;
        case 2: //Segunda situacao (quadrado).
            y=x*x;
            break;
        case 3: //Terceira situacao (dobro).
            y=2*x;
            break;
        default: //Situacao padrao onde y=x.
            y=x;}

    printf("\ny=%f\n",y);
    return 0;}
```

## 12-Matrizes e vetores

As operações com vetores e matrizes são uma constante na programação de computadores. Todas as vezes que os cálculos implicam uma grande quantidade de variáveis e permite que operações semelhantes sejam empregadas a utilização de vetores e matrizes é adequada.

Como na maioria das linguagens os elementos pertencentes a uma matriz devem pertencer a uma mesma classe de variável. O nome da matriz indica o endereço do primeiro elemento da mesma. As matrizes podem ser criadas explicitamente, durante a execução do padrão ou definidas. As matrizes criadas por condição padrão tomam a forma:

**<tipo de variável> <nome da variável> [dimensão 1] [dimensão 2]...;**

Pode-se ainda inicializar os valores das matrizes durante sua declaração, alguns compiladores C permitem que seja desprezada a declaração das dimensões neste caso.

Exemplos:

```
int A[4];
char resposta[ ]=[1 2 3]; /* Neste caso o número de elementos é implícito. */
float x[2] [3];
...
```

A maioria dos programas que envolvem a utilização de matrizes recorrem aos *loops* para percorrer todos os elementos da matriz. Para que seja possível "percorrer" todos os elementos de um vetor ou de uma matriz o índice numérico deverá se iniciar em um valor e terminar em um outro, sendo que os índices são números inteiros. O *loop* de *FOR* possui a variável contadora a qual apresenta exatamente estas características. A linguagem C inicia o índice de um determinado valor automaticamente. O número máximo de elementos do vetor/matriz também deve ser declarado, no entanto este número máximo não significa que todos os elementos devam ser utilizados, eles estarão disponibilizados apenas. O trecho de leitura de uma matriz X com M por N elementos, seria o trecho de algoritmo a seguir:

```
/* Este programa lê uma matriz de dimensões m x n e mostra esta matriz. ANSI-C */
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{ int m,n,i,j; float temp,x[9][9];
```

```
/* Leitura das dimensoes da matriz X. */
printf("Entre com as dimensoes da matriz (linha,coluna) máximo 9x9:");
scanf("%d,%d",&m,&n);
```

```
/* Lendo os elementos da matriz Xmn. */
```

```
for (i=1;i<=m;i++)
{ for (j=1;j<=n;j++)
{ printf("X[%d,%d]=",i,j); scanf("%f",&temp); x[i][j]=temp; }}
```

```
/* Mostrando os valores lidos. */
```

```
printf("A matriz X que o usuario entrou foi:\n\n");
for (i=1;i<=m;i++)
{ for (j=1;j<=n;j++) { printf("%+8.2f",x[i][j]); printf("\n"); } return 0;}
```

-----X-----

```

// Este programa lê uma matriz de dimensões  $m \times n$  e mostra esta matriz. C++
#include <iostream>
#include <iomanip>
int main()

{ int m,n,i,j; float temp,x[9][9];

// Leitura das dimensoes da matriz X.
std::cout<<"Entre com as dimensoes da matriz (linha,coluna) máximo 9x9:";
std::cin>>m>>n;

// Lendo os elementos da matriz Xmn.
for (i=1;i<=m;i++)
{ for (j=1;j<=n;j++)
{ std::cout<<"X["<<i<<" "<<j<<"]=";    std::cin>>temp;    x[i][j]=temp; }}
// Mostrando os valores lidos.
std::cout<<"A matriz X que o usuario entrou foi:\n\n";
for (i=1;i<=m;i++)
{ for (j=1;j<=n;j++)
{ std::cout<<std::setw(9)<<x[i][j];}
std::cout<<"\n"; } return 0;}

```

Deve-se notar que no programa apresentado a matriz é lida e escrita através da utilização de dois *loops* de *for*, este procedimento permite percorrer todos os elementos da matriz. Para compreender seu funcionamento basta notar que o loop mais externo fixa um valor de linha (ou coluna) e dentro de seus comandos a serem repetidos existe um outro *loop* que irá "percorrer" as colunas (ou linhas) desta posição fixada e esta operação irá se repetir por todas as linhas (ou colunas). Observando:

```

... for (i=0; i<=m; i++)
    for (j=0; j<=n; j++)
        { ...          //Operações com todos os elementos da matriz mxn.}
...

```

Nota-se que tal rotina irá percorrer todos os elementos de uma matriz, porém eventualmente pode-se desejar que apenas uma parte da matriz seja percorrida, isto pode ser facilmente ajustado com modificações no intervalo interno ou externo do loop. Os exemplos a seguir ilustram alguns formatos distintos da matriz convencional que serão percorridos com sua execução:

```

... for (i=0; i<=m; i++)
    for (j=i; j<=n; j++)
        { ...          //Operações com os elementos do triângulo superior da matriz
mxn.}
...

```

```

... for (i=0; i<=m; i++)
    { ...          //Operações com os elementos da diagonal principal da matriz
mxn.}
...

```



### 12.1-Exemplo de manipulação de matrizes: Multiplicação de matrizes

A multiplicação de duas matrizes é uma operação muito encontrada quando ocorre a manipulação de dados com matrizes. Supondo a multiplicação de duas matrizes:

$$X_{mn} \cdot Y_{nj} = Z_{mj}$$

Os termos da matriz resultante Z podem ser determinados individualmente, basta observar que cada elemento é formado pela soma da multiplicação dos elementos da linha da matriz X, sendo a linha igual ao número da linha do elemento de Z, pelos elementos da coluna da matriz Y, sendo a coluna igual ao número da coluna do elemento Z, como observado na eq. 3.

$$Z_{a,b} = \sum_{k=1}^n X_{a,k} \cdot Y_{k,b} \quad (3)$$

A seguir é apresentado um programa que calcula a multiplicação de duas matrizes.

```
/* Este programa calcula a multiplicação de duas matrizes X.Y=Z. ANSI-C*/
#include <stdio.h>
#include <stdlib.h>
int main()
{ int m,n,j,a,b,k; float temp,x[9][9],y[9][9],z[9][9];

/* Leitura das dimensoes das matrizes X e Y. */
printf("Entre com as dimensoes da matriz X (linha,coluna com um máximo 9x9):");
scanf("%d %d",&m,&n);
printf("Entre com o número de colunas da matriz Y (máximo 9):");
scanf("%d",&j);

/* Lendo os elementos da matriz Xmn. */
for (a=0;a<=m;a++)
{ for (b=0;b<=n;b++)
  { printf("X[%d,%d]= ",a,b);   scanf("%f",&temp);   x[a][b]=temp; }}

/* Lendo os elementos da matriz Ynj. */
for (a=0;a<=n;a++)
{ for (b=0;b<=j;b++)
  { printf("Y[%d,%d]= ",a,b);   scanf("%f",&temp);   y[a][b]=temp; }}

/* Cálculo da multiplicação das matrizes (Z), onde Z=X.Y. */
for (a=0;a<=m;a++)
  for (b=0;b<=j;b++)
    {z[a][b]=0; // Inicializacao de cada elemento da nova matriz resultante.
     for (k=0;k<=n;k++)
       z[a][b]+=x[a][k]*y[k][b];} // Expressao da eq. 3 apresentada.
/* Saída do valor da multiplicação das duas matrizes. */
printf("A multiplicação de X.Y que o usuario entrou foi:\n\n");
for (a=0;a<=m;a++)
{ for (b=0;b<=j;b++)
  { printf("%+8.2f",z[a][b]);}
  printf("\n"); }      return 0;}
```

-----X-----

```

// Este programa calcula a multiplicação de duas matrizes X.Y=Z. C++
#include <iostream>
#include <iomanip>
int main()
{ int m,n,j,a,b,k; float x[9][9],y[9][9],z[9][9];

// Leitura das dimensoes das matrizes X e Y.
std::cout<<"Entre com as dimensoes da matriz X (linha,coluna com um máximo 9x9):";
std::cin>>m>>n;
std::cout<<"Entre com o número de colunas da matriz Y (máximo 9):";
std::cin>>j;

// Lendo os elementos da matriz Xmn.
for (a=0;a<=m;a++)
{ for (b=0;b<=n;b++)
{ std::cout<<"X("<<a<<","<<b<<")=";    std::cin>>x[a][b]; }}

// Lendo os elementos da matriz Ynj.
for (a=0;a<=n;a++)
{ for (b=0;b<=j;b++)
{ std::cout<<"Y("<<a<<","<<b<<")=";    std::cin>>y[a][b]; }}

// Cálculo da multiplicação das matrizes (z).
for (a=0;a<=m;a++)
for (b=0;b<=j;b++)
{ z[a][b]=0;
for (k=0;k<=n;k++)
z[a][b]+=x[a][k]*y[k][b];}

// Saída do valor da multiplicação das duas matrizes.
std::cout<<"A multiplicação de X.Y que o usuario entrou foi:\n\n";
for (a=0;a<=m;a++)
{ for (b=0;b<=j;b++)
{std::cout<<std::setw(10)<<z[a][b];}
std::cout<<"\n"; }
return 0;}

```

## 12.2-Exemplo de manipulação de matrizes: Solução de sistemas lineares

A resolução de sistemas de equações lineares é um exemplo da utilização de matrizes. Existem diversos métodos que possibilitam a solução destes sistemas, cujo objetivo é determinar as variáveis que satisfazem a um determinado sistema de equações. Um, entre os diversos métodos existentes, utilizado para a solução de sistemas lineares é o Método da Eliminação de Gauss, apresentado a seguir:

a) Supondo o sistema:  $\vec{A} \cdot \vec{B} = \vec{C}$

b) Transforma-se a matriz A em uma matriz triangular superior, através de permutações e combinações lineares.

c) Utilizando a “retrossubstituição” é possível determinar o último valor de variável desconhecida e proceder com a substituição dos termos imediatamente superiores ao termo determinado.

Supondo o exemplo apresentado a seguir:

$$2a+b+c=7$$

$$a+b=3$$

$$3a+c=6$$

tem-se que:

$$\vec{A} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} ; \quad \vec{x} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} ; \quad \vec{y} = \begin{pmatrix} 7 \\ 3 \\ 6 \end{pmatrix}$$

“triangularizando” a matriz:

$$\left[ \begin{array}{ccc|c} 2 & 1 & 1 & 7 \\ 1 & 1 & 0 & 3 \\ 3 & 0 & 1 & 6 \end{array} \right] \begin{matrix} L_1 \\ L_2 \\ L_3 \end{matrix} \rightarrow \begin{matrix} L_1 \\ -1/2 L_1 + L_2 \\ -3/2 L_1 + L_3 \end{matrix} = \left[ \begin{array}{ccc|c} 2 & 1 & 1 & 7 \\ 0 & 1/2 & -1/2 & -1/2 \\ 0 & -3/2 & -1/2 & -9/2 \end{array} \right] \rightarrow \begin{matrix} L_1 \\ L_2 \\ 3 L_2 + L_3 \end{matrix} = \left[ \begin{array}{ccc|c} 2 & 1 & 1 & 7 \\ 0 & 1/2 & -1/2 & -1/2 \\ 0 & 0 & -2 & -6 \end{array} \right]$$

É possível determinar o valor de  $c=3$ , posteriormente com a equação de  $L_2$  determina-se o valor de  $b=2$  ( $L_2$ ) e finalmente o valor de  $a=1$  ( $L_1$ ). O programa apresentado a seguir irá solucionar sistemas de equações lineares:

*/\* Método de eliminação de Gauss para solução de equações. ANSI-C \*/*

*#include <stdio.h>*

*#include <stdlib.h>*

*int main()*

*{ int i,j,k,n; float a[9][9],x[9],temp;*

*/\* Leitura do numero de equações. \*/*

*printf("Entre com o numero de equações:");*

*scanf("%d",&n); n--;*

*/\* Leitura dos termos das equações. \*/*

*for (i=0;i<=n;i++)*

*{ printf("Linha %d:\n",i);*

*for (j=0;j<=n;j++)*

*{ printf("Forneça o termo de X%d:",j); scanf("%f",&temp); a[i][j]=temp; }*

*printf("O valor da solução e:"); scanf("%f",&temp); a[i][j]=temp;}*

*/\* Mostra a matriz original. \*/*

*printf("\n\n\nMatriz original:\n");*

*for (i=0;i<=n;i++)*

*{ for (j=0;j<=n+1;j++) printf("%+8.2f",a[i][j]); printf("\n");}*

*/\* Processo de triangularização da matriz. \*/*

*for (k=0;k<n;k++)*

*for (i=k+1;i<=n;i++)*

*{ temp=(-1.0)\*a[i][k]/a[k][k];*

*for (j=k;j<=n+1;j++)*

*a[i][j]=(temp\*a[k][j])+a[i][j]; }*

*/\* Mostra a matriz modificada. \*/*

*printf("\n\n\nMatriz modificada:\n");*

*for (i=0;i<=n;i++)*

*{ for (j=0;j<=n+1;j++) printf("%+8.2f",a[i][j]);*

*printf("\n");}*

```

/* Determinação da solução das equações. */
for (i=n;i>=0;i--)
{ x[i]=a[i][n+1];
  for (j=n;j>i;j=j-1)
    x[i]-=x[j]*a[i][j];
  x[i]=x[i]/a[i][i];}

/* Saída de Resultados. */
for (i=0;i<=n;i++)
printf("X[%d]=%+8.2f\n",i,x[i]);

return 0;}

```

-----X-----

## 13-Ponteiros

As variáveis utilizadas na maioria dos programas são do tipo estática, possuem um nome para identificá-las e mantêm um valor que pode ser modificado ao longo do programa. A quantidade de variáveis estáticas que podem ser manipuladas é limitada (e menor que a memória disponível no computador). Uma vez declarada as variáveis são impossíveis de variar este valor.

Uma maneira mais eficiente de manipular os valores em um programa seria a utilização de *ponteiros*, estes não possuem um valor armazenado mas o endereço na memória onde se encontra o valor. Existem inúmeras vantagens na utilização de *ponteiros*, entre elas:

- A quantidade de valores a serem utilizados é maior, ficando limitado a uma porção disponível da memória do computador. Esta quantidade é muito maior que a obtida com a utilização de variáveis estáticas.
- A manipulação dos valores ocorre de forma mais eficiente e rápida.
- Os dados serão armazenados em uma sequência da memória e podem ser acessados, modificados ou levados a outro local a qualquer momento do programa.
- entre outras.

Há, porém, uma certa dificuldade de compreender como acontece a manipulação dos valores através de seu endereço, a representação seria:

**123,45** → **12Ah**      Esta representação informa que o número 123,45 está guardado na posição 12Ah (h de hexadecimal) da memória.

Não se deve confundir o endereço com o valor mantido na memória. Estes valores são independentes. É possível que os endereços mudem ao longo da execução do programa, porém tal fato não afeta diretamente a execução do programa.

Com o conhecimento do tamanho de cada valor armazenado no endereço é possível estabelecer onde se localiza o próximo valor armazenado, o tamanho pode ser obtido com a função *sizeof*. Cabe salientar também que a variável do tipo *ponteiro* mantém uma relação direta com o tipo de variável armazenada, isto é, ela deve ser declarada com o mesmo grupo ao qual pertence o valor indicado pela mesma. Para identificar a variável ponteiro é inserido um \* antes do nome da variável, portanto a declaração de uma variável ponteiro seria:

```
int *valor;  
char *teste;
```

Quando se deseja indicar o conteúdo de um valor contido no ponteiro basta utilizar o \* antes da variável ponteiro; porém se é desejado inserir um endereço no ponteiro a variável deverá possuir um & antes de seu nome, este & indica que o endereço da variável estará sendo armazenado e não seu valor. Um exemplo é apresentado a seguir:

```
...int *endereco;    //ponteiro do tipo inteiro.  
int valor;          //variável do tipo inteiro.  
...valor=12;  
...endereco=&valor;    //o endereço da variável é armazenado no ponteiro.  
...cout<<"O valor armazenado no endereço do ponteiro é:"<<*endereco;  
...
```

A utilização de *ponteiros* pode se estender ainda a diversos níveis, uma vez que o *ponteiro* utilizado pode "apontar" também para outro *ponteiro*, basta observar que a cada novo "apontamento" surge um novo \*. A razão para estes diferentes "apontamentos" é

que algumas vezes ocorre a manipulação de partes da memória pelo sistema, o sistema poderia modificar o endereço de um objeto da memória o que causaria um desastre, porém se a atualização do objeto for realizada e um local apenas “apontar” para sua localização não ocorrerá instabilidade no comportamento das variáveis do programa. Um exemplo está apresentado a seguir:

```
...int valor=2;
int *ponteiro1;
int **ponteiro2;
int ***ponteiro3;
...ponteiro1=&valor;           //ponteiro1 “aponta” para o endereço da variável valor.
ponteiro2=&ponteiro1;         //ponteiro2 “aponta” para o endereço do ponteiro1.
ponteiro3=&ponteiro2;         //ponteiro3 “aponta” para o endereço do ponteiro2. ...
```

A variável ponteiro pode também indicar a posição de um vetor ou matriz, basta para tal que ela esteja associada diretamente a(o) matriz(vetor) em questão, ou a seu primeiro elemento:

```
...float vetor[5];    float *pont_vetor;
...pont_vetor=vetor;  //Poderia ser substituído por: pont_vetor=&vetor[0];
```

A indicação do próximo elemento de um endereço armazenado em uma variável *ponteiro* poderá ser obtida mediante a simples inclusão da sequência ++ à frente da variável *ponteiro*, tal indicação tomará o próximo valor acrescentando o número de bytes necessários automaticamente. Tome-se o exemplo:

```
...float *ponteiro;
...ponteiro++;... //Este acréscimo indica o próximo valor.
```

A seguir é apresentado um programa que irá realizar a leitura de um vetor e colocar seu valor inicial em uma variável *ponteiro*, posteriormente os valores serão apresentados a partir do ponteiro.

```
/* Exemplo da utilização de ponteiros em ANSI-C */
#include <stdio.h>
#include <stdlib.h>
int main ( )
{ float vetor[10],temp;    int i,n;
  float *ponteiro;
  /* Leitura dos elementos do vetor */
  printf("\nEntre com o número de elementos:"); scanf("%d",&n);
  for (i=0;i<=n;i++)
  { printf("vetor[%d]=",i);    scanf("%f",&temp); vetor[i]=temp; }
  ponteiro=vetor; /* A variável ponteiro aponta para o primeiro elemento do vetor */
  /* A repetição dos valores lidos, utilizando a variável ponteiro */
  for (i=0;i<=n;i++)
  { printf("vetor[%d]=%f\n",i,*ponteiro);    ponteiro++; }
  return 0;}
```

-----X-----

```

#include <iostream> // Exemplo da utilização de ponteiros em C++
#include <iomanip>
int main ( )
{ float vetor[10];    int i,n;
  float *ponteiro;

  // Leitura dos elementos do vetor
  std::cout<<"\nEntre com o número de elementos:"; std::cin>>n;
  for (i=0;i<=n;i++)
  { std::cout<<"vetor["<<i<<"]=";    std::cin>>vetor[i]; }
  ponteiro=&vetor[0]; // A variável ponteiro aponta para o primeiro elemento do vetor

  // A repetição dos valores lidos, utilizando a variável ponteiro
  std::cout<<"\n\nVetor obtido a partir do ponteiro:"<<std::endl;
  for (i=0;i<=n;i++)
  {std::cout<<"vetor["<<i<<"]="<<*ponteiro<<std::endl; ponteiro++;}
  return 0;}

```

É possível associar a uma variável ponteiro uma cadeia de caracteres, neste caso apenas o endereço da primeira letra da cadeia de caracteres será armazenado. Então pode-se tomar o exemplo:

```
char *expressao = "Valor inválido";
```

Nem todas as operações aritméticas são permitidas entre ponteiros, não é permitido operar entre ponteiros com somas, subtrações ou divisões. A soma ++ que fornece o endereço do próximo valor a ser lido também deve ser utilizada com cautela, caso a forma seja:

```

++ponteiro...
ponteiro++...

```

os resultados irão diferir, pois a associação dos valores se dará de forma diferente.

A utilização da função `sizeof( )` é de grande valia, sendo a mesma empregada para determinar o “tamanho” utilizado em cada variável indicada pelo ponteiro. Tome-se o exemplo:

```

/* Exemplo da utilização de sizeof( ) em ANSI-C */
#include <stdio.h>
int main ( )
{ /* São declarados as variáveis e seus ponteiros e associados seus valores */
  int i=2, *pont_i;
  float f=3, *pont_f;
  char c='a', *pont_c;
  /* Seus endereços são indicados aos ponteiros */
  pont_i=&i;
  pont_f=&f;
  pont_c=&c;
  /* Impressão dos valores e o espaço ocupado pelas mesmas */
  printf("\nA variável inteira i=%d, ocupa %d bytes", *pont_i, sizeof(i));
  printf("\nA variável real f=%f, ocupa %d bytes", *pont_f, sizeof(f));
  printf("\nA variável inteira c=%c, ocupa %d bytes", *pont_c, sizeof(c));
  return 0;}

```

-----X-----

Pode-se notar que a presença do `&` antes de uma variável indicará o endereço na memória desta variável e não seu valor. Ao se utilizar o comando *scanf* da linguagem ANSI-C o que ocorria é o endereçamento do valor lido para a posição na memória ocupada pela variável em questão, na linguagem C++ isto também ocorre, porém de forma mais discreta.

Ao ser executado um programa em C parte a memória disponível em 4 partes: a parte da codificação do programa, os dados globais, a pilha e o *“heap”* (área livre da memória). O *“heap”* pode ser controlado com os comandos *malloc* e *free* (ANSI-C) ou *new* e *delete* (C++), estes comandos são utilizados para reservar e liberar memória contínua durante a execução do programa.



## 14-Funções

As funções são a chave mestra da linguagem C. Todos os programas escritos em C possuem uma grande quantidade de funções definidas, cada uma responsável por uma determinada execução. A utilização de funções é justificável quando o programa toma dimensões muito grandes, ou quando há necessidade de executar um determinado grupo de instruções diversas vezes. As linguagens ANSI-C e C++ possuem algumas diferenças entre suas funções, porém todas as funções desenvolvidas pela linguagem ANSI-C podem ser interpretadas sem problemas por um compilador C++. Existem ainda diversas propriedades que a utilização de funções, entre elas:

- Possibilidade de testar trechos do programa e verificar a eficiência de rotinas.
- Manter uma “biblioteca” de funções desenvolvidas para utilizações posteriores em outros programas.
- Simplificar a análise dos programas e modificar trechos do mesmo.
- Diminuir o tamanho do arquivo do programa.
- etc.

A estrutura básica de uma função é:

**<tipo da função> <nome da função> (<valores recebidos>);**

A função pode ser do tipo *void*, *int*, *char*, *float*, etc. este tipo irá determinar qual será a classe da variável retornada para o local que chamou a função. Os valores recebidos serão declarados com o tipo da variável recebida (a qual será utilizada ao longo da subrotina).

Uma função pode chamar a si mesma, esta possibilidade é conhecida como recorrência.

Supondo uma função que retorne a média aritmética de dois números, retornando um valor do tipo real (*float*):

```
/* Exemplo de função em ANSI-C. Cálculo da média aritmética de 2 números */
#include <stdio.h>
#include <stdlib.h>
float media(int x, int y); /* Modelo da função */
int main ( )
{
    int a,b;
    printf("A="); scanf ("%d",&a);
    printf("B="); scanf ("%d",&b);
    printf("A média entre A e B é=%f",media(a,b)); return 0;}
float media(int x, int y)    /* Declaração da subrotina. */
{
    float z;
    z=(float)(x+y)/2;        /* Cálculo da média com operação forçada do tipo real. */
    return (z); }            /* Retorno do valor calculado */
```

-----X-----

```
// Exemplo de função em C++. Cálculo da média aritmética de 2 números
#include <iostream>

float media(int x, int y); // Modelo da função
int main ( )
{
    int a,b;
    std::cout<<"A="; std::cin>>a;
    std::cout<<"B="; std::cin>>b;    std::cout<<"A média entre A e B é="<<media(a,b);
    return 0;}
float media(int x, int y)    // Declaração da subrotina.
{float z;    z=(float)(x+y)/2;    // Cálculo da média com operação forçada do tipo real.
    return (z); }    // Retorno do valor calculado.
```

Nota-se a presença de uma chamada da função *media*, esta função deve ser previamente apresentada, através de seu modelo e posteriormente declarada. A função em questão é do tipo real e recebe dois valores inteiros. Cabe salientar que os nomes das variáveis recebidas não devem obrigatoriamente ser iguais aos nomes utilizados pelas mesmas durante a execução da rotina que chamou a função. No exemplo citado a variável *a* recebe o nome de *x* durante a função e a variável *b* recebe o nome de *y*. A função pode também receber valores por referências de ponteiros, no programa apresentado as modificações seriam nos trechos que contenham referências a sub-rotina, então:

onde se encontra:

```
...float media (int x, int y) ...
...media (a,b) ...
```

seriam trocados por:

```
...float media (int *x, int *y) ...
...media (&a, &b) ...
```

Nota-se que a partir deste momento apenas os endereços das variáveis são utilizados, na forma apresentada no capítulo sobre ponteiros.

Quando não existem argumentos para a função utilizada a função é declarada como *void* (variável nula) o mesmo ocorrendo no local com os *valores recebidos*. Supondo uma função que não possua nenhum valor a ser retornado e apenas deva ser chamada, ela poderá ser declarada da forma:

**void <nome da função>(void)**

Um exemplo poderia ser a função apresentada a seguir, a qual irá limpar a tela e inscrever uma apresentação a cada vez que for chamada:

```
/* Cabeçalho em ANSI-C */
...void cabecalho (void)
{ ...    printf ("Calculo da média aritmética entre dois números:\n\n"); } ...

-----X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-X-----

//Cabeçalho em C++
...void cabecalho (void)
{...    cout<<"Calculo da média aritmética entre dois números:\n\n"); } ...
```

### 14.1-Exemplo de função: Ordenação de elementos em um vetor

Para ordenar um vetor, na ordem crescente por exemplo, uma das maneiras mais simples seria: comparar todos os elementos e colocar o menor elemento na primeira posição, posteriormente comparar do segundo elemento em diante e colocar o menor elemento na segunda posição, repetindo este processo até o último elemento. Pode-se ilustrar tal situação a seguir:

$$\begin{pmatrix} 3 \otimes \\ ]b[ \\ 2 \\ -1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \otimes \\ 3 \\ ]b[ \\ -1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \otimes \\ 2 \\ 3 \\ ] 1[ \end{pmatrix} \Rightarrow \begin{pmatrix} -1 \otimes \\ 2 \\ 3 \\ 0 \end{pmatrix}$$

Durante a programação a maneira mais simples de se obter tal sequência seria trocando de posição o elemento fixo com o seguinte sempre que esta situação ocorrer. Nota-se no exemplo a determinação do menor de todos os elementos, o primeiro elemento (marcado com  $\otimes$ ) é comparado aos demais (note o elemento comparado marcado entre  $] [$ ) e troca de posições sempre que encontra um elemento de menor valor. Este modo não apresenta tanta eficiência, uma vez que para vetores de grandes dimensões ele irá demorar um tempo muito elevado de processamento pois vai realizar  $n!$  repetições de comparação para obter o resultado final.

*/\* Programa em ANSI-C que lê um vetor e posteriormente utiliza uma função para ordenar os elementos deste vetor. \*/*

*#include <stdio.h>*

*#include <stdlib.h>*

*void ordena(float \*p, int n);*      */\* Declaração da função de ordenação. \*/*

*/\* Função principal. \*/*

*int main ( )*

*{ float x[20],temp;    int i, n;*

*/\* Leitura dos dados do vetor. \*/*

*printf ("Número de elementos:"); scanf("%d",&n);*

*for(i=0;i<=n;i++)*

*{printf("x[%d]=",i);    scanf("%f", &temp); x[i]=temp;}*

*ordena(&x[0],n);    /\* Chama a função de ordenação. \*/*

*/\* Saída do vetor ordenado. \*/*

*for(i=0;i<=n;i++)printf("\n\nx[%d]=%f\n",i,x[i]);*

*return 0;}*

*/\* Função que executa a ordenação dos elementos do vetor, utilizando ponteiros. \*/*

*void ordena(float \*p, int n)*

*{int i,j;*

*float temp,\*q;*

*/\* A ordenação é feita pelos ponteiros, comparando um elemento aos demais e sempre mantendo o maior entre eles. \*/*

*for (i=0;i<=n;i++)*

```

{q=p; for (j=i;j<=n;j++)
  {if (*p>*q) /* CRESCENTE UTILIZA *p>*q, DECRESCENTE *p<*q. */
    {temp=*p; *p=*q; *q=temp;} q++;} p++;}}

```

-----X-----

## 14.2-Exemplo de função: Determinação de uma integral definida

Para se encontrar a solução de diversos problemas é necessário algumas vezes o conhecimento prévio das funções envolvidas. Pode-se exemplificar tal situação com um problema para determinar a integral definida de uma função. Existem diversas maneiras de se determinar o valor da integral definida de uma função, o cálculo aproximado do valor da área sob a curva da função é um dos métodos mais utilizados. Entre os métodos utilizados para tal cálculo o conhecido como “Regra dos Trapézios” se destaca como um dos mais simples. Supondo a função  $f(x)=x^2$  apresentada na fig.8.

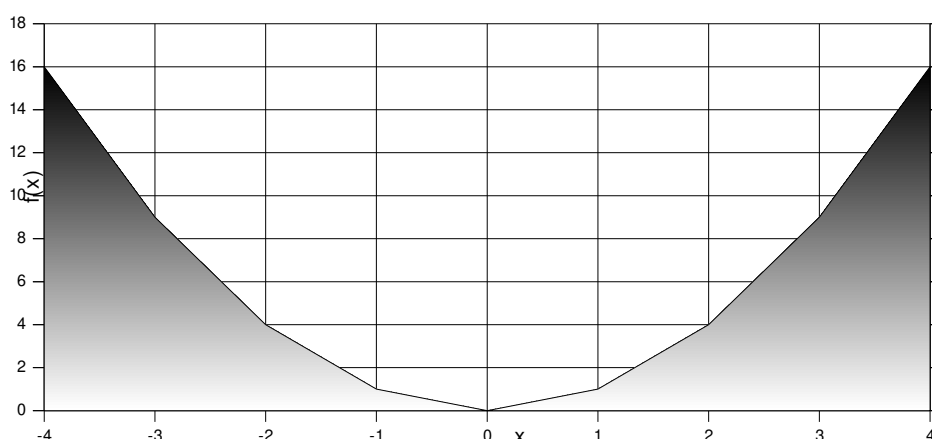


Figura 8 – Gráfico da função  $f(x) = x^2$

A integral definida da função  $f(x)=x^2$  de  $[-4,4]$  ( $\int_{-4}^4 x^2 .dx$ ) corresponde a área hachurada na fig.8. Para encontrar o valor desta área seria possível somar uma quantidade de figuras geométricas cuja dimensão seja de fácil obtenção, como na fig.9.

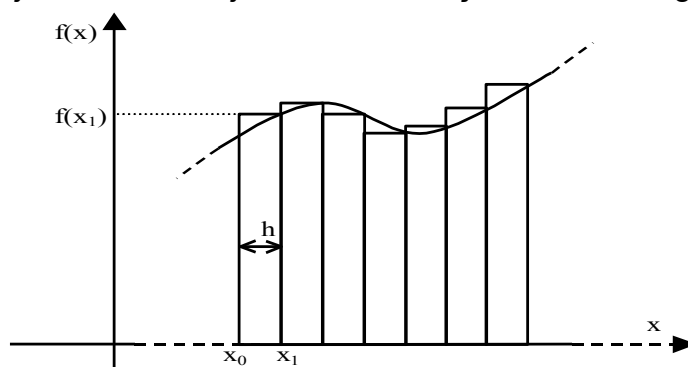


Figura 9 – Determinação de uma integral definida.

Nota-se que os retângulos possuem todos seus valores conhecidos pois a função  $f(x)$  é conhecida, portanto, os valores de  $x$  que tocam na função formando o retângulo são

conhecidos. A área de cada retângulo seria:

$$A = f(x) \cdot h$$

A soma total dos retângulos irá resultar na área total sob a figura e determinará o valor aproximado da integral definida. Como o valor de  $f(x)$  é conhecido em todos os pontos e o valor de  $h$  é definido pelo usuário é possível determinar a área sob a curva em um determinado intervalo. Quanto menor o valor de  $h$  determinado pelo usuário maior será a precisão da área calculada, uma vez que a área determinada possui brechas, observa-se no desenho áreas não ocupadas ou ocupadas excessivamente no trecho próximo a curva. O valor de  $h$  irá depender da função utilizada e do intervalo da integração, porém não será necessário a utilização de valores muito pequenos uma vez que a precisão requerida irá depender da aplicação e a utilização de valores muito pequenos poderá aumentar o trabalho computacional desnecessariamente.

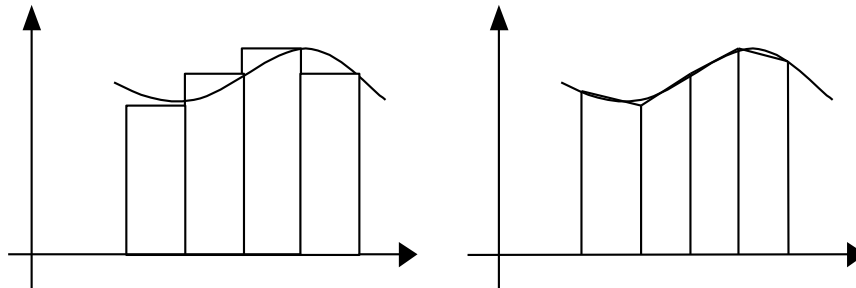


Figura 10 – Comparação entre retângulo e trapézio no cálculo da integral definida.

Uma variação deste método seria a utilização de trapézios, conforme a fig. 10, os trapézios trariam uma maior precisão a determinação da área como pode ser visualizado na figura citada.

Nota-se que a área preenchida pelo trapézio é a que mais se aproxima da área real ocupada pela figura analisada. De forma semelhante a área de cada trapézio também pode ser estabelecida, como anteriormente, então a área de cada trapézio seria:

$$A = \frac{[f(x_0) + f(x_1)]h}{2}, \text{ onde } x_1 = x_0 + h$$

A soma de todos os trapézios irá definir a área total sob a curva, que terá um valor próximo ao valor da integral definida desta função, esta é a “Regra dos Trapézios”.

Fica claro que será necessário repetir uma grande quantidade de vezes o cálculo da área de cada retângulo, a utilização de uma linguagem de programação para determinar o valor total da área será de grande valia e se enquadra na característica de cálculos sequenciais repetitivos, uma das principais vantagens das linguagens de programação. Os programas em C que utilizariam tal método estão apresentados a seguir:

```
/* Programa em ANSI-C que calcula uma integral definida. */
#include <stdio.h>
#include <stdlib.h>
float f(float x); /* Função que se deseja determinar a integral. */
float integral(float a, float b); /* Função que irá calcular a integral. */
int main ( ) /* Função principal. */
{ float inicial, final;
  printf ("Entre com o valor inicial do intervalo:"); scanf ("%f",&inicial);
  printf ("Entre com o valor final do intervalo:"); scanf ("%f",&final);
```

```

printf("O valor da integral deste intervalo é:%f",integral(inicial,final));
return 0;}
/* Função que calcula a integral. */
float integral(float a,float b)
{ float x,h,s;
  h=0.001;    /* Intervalo de integração. */
  x=a;  s=0;
  do
  { s+=h*(f(x)+f(x+h))/2;
    x+=h;} while (x<=b);
  return (s);}

/* Função a ser integrada. */
float f(float x)
{float s;
  s=x*x;
  return (s);}

```

-----X-----

```

/* Programa em C++ que calcula uma integral definida. */
#include <iostream>
#include <math.h>
float f(float x);    /* Função que se deseja determinar a integral. */
float integral(float a,float b);    /* Função que irá calcular a integral. */
int main ( )    /* Função principal. */
{ float inicial, final;
  std::cout<<"Entre com o valor inicial do intervalo:"; std::cin>>inicial;
  std::cout<<"Entre com o valor final do intervalo:"; std::cin>>final;
  std::cout<<"O valor da integral deste intervalo é:"<<integral(inicial,final);
  return 0;}
/* Função que calcula a integral. */
float integral(float a,float b)
{ float x,h,s;
  h=0.001;    /* Intervalo de integração. */
  x=a;  s=0;
  do
  { s+=h*(f(x)+f(x+h))/2;
    x+=h;} while (x<=b);
  return (s);}

/* Função a ser integrada. */
float f(float x)
{float s;
  s=x*x;
  return (s);}

```

Deve-se notar que quanto menor o intervalo de integração, o valor de h neste programa exemplo, mais precisa a resposta; porém quanto maior a quantidade de cálculos a serem executados, maior o erro acumulado nos arredondamentos realizados pelo computador e mais tempo para o cálculo da integral.

## 15-Variáveis Locais e Globais

Ao utilizar uma função é possível criar variáveis locais, as quais possuem a maior precedência durante a execução da função, porém é possível utilizar variáveis globais, bastando inserir “::” antes do nome da variável em questão. Tome-se os exemplos:

```
// Programa em C++ que calcula a área de uma circunferência.
#include <iostream>
#include <math.h>

float pi=3.1415926; // Variável global, com efeito para todas as funções do programa.
double area(double raio);
// Função principal.
int main( )
{ double r;
  std::cout<<"Entre com o valor do raio:"; std::cin>>r;
  std::cout<<"A area é:"<<area (r);
  return 0;}

// Função que calcula a área da circunferência.
double area(double raio)
{ double a, pi=3.14; // Variável local da função area.
  a=pi*pow(raio,2);
  return(a);}
```

Para o programa acima a resposta seria menos precisa, apesar do valor de  $\pi$  estar definido até a sétima casa decimal como variável global, a função utiliza o valor da variável local de  $\pi$  (definido apenas até a segunda casa decimal). Para contornar este problema utiliza-se o “::”, que “forçara” a utilização da variável global nas operações, então o programa será:

```
// Programa em C++ que calcula a área de uma circunferência.
#include <iostream>
#include <math.h>

float pi=3.1415926; // Variável global, com efeito para todas as funções do programa.
double area(double raio);
// Função principal.
int main( )
{ double r;
  std::cout<<"Entre com o valor do raio:"; std::cin>>r;
  std::cout<<"A area é:"<<area (r);
  return 0;}

// Função que calcula a área da circunferência.
double area(double raio)
{ double a, pi=3.14; // Variável local da função area.
  a=::pi*pow(raio,2);
  return(a);}
```

Para esta situação a resposta terá uma precisão maior, pois o valor de  $\pi$  que será utilizado pelos cálculos será o valor declarado como variável global.

Opcionalmente a inclusão das palavras "static" e "extern" podem propiciar resultados semelhantes aos encontrados. Supondo que uma variável antes de sua declaração possua a palavra "extern" a mesma estará visível e disponível para todo o programa. Ao utilizar-se a palavra "static" a variável estará visível apenas na rotina que a declarou (apenas localmente). As demais variáveis podem ser consideradas como "auto", ou "register" e apenas se distinguem pelo fato das variáveis do tipo "register" poderem utilizar registradores se possível, tornando a execução do programa mais rápida.

Suponha o programa apresentado a seguir, onde um determinado valor é recalculado em unidades científicas de potência.

*// Programa em C++ para determinação de um valor em diferentes níveis de potência.*

```
#include <iostream>
```

```
float func_pot(float z);    // Pré-declaração da função.
```

*// São declaradas as variáveis que atingem o programa globalmente.*

```
extern float kilo=1000, mega=1000000;
```

```
int main()
```

```
{static float x;          // Esta é a variável x declarada somente para a função main  
(localmente).
```

```
std::cout<<"Valor (em unidades):";    std::cin>>x;
```

```
func_pot(x); return (0);}
```

```
float func_pot(float z)
```

```
{static float x,y;        // A variável x é novamente declarada para a função func_pot.
```

```
x=z/kilo;    y=z/mega;
```

```
std::cout<<"Valor="<<x<<"kilo"<<" ou Valor="<<y<<"mega";}
```



## 16-Arquivos

Quando a quantidade de dados a ser utilizada em alguns programas é muito elevada poderá ser necessário recorrer a utilização de arquivos. A utilização de arquivos facilita a programação e possibilita realizar modificações constantes nos dados fornecidos, assim como na saída dos dados para o usuário.

Existem diversos tipos de arquivos que podem ser utilizados, porém os arquivos no formato ASCII (American National Standard Code for Information Interchange – Código padrão americano para intercâmbio de informações entre diferentes equipamentos de processamento e comunicação de dados), este formato garante uma grande portabilidade para os arquivos. É reconhecido na maioria dos computadores existentes pois não requer nenhum interpretador especial, os dados se encontram sob uma forma totalmente legível. Existem arquivos que possuem formatos especiais, os bancos de dados são as aplicações mais frequentemente encontradas com formato especial. A linguagem C/C++ utiliza arquivos ASCII e arquivos no formato binário.

Os bancos de dados são utilizados com frequência em diversas atividades e ao abrir um arquivo com um formato especial é necessário possuir um interpretador adequado, que pode ser encontrado em diversas linguagens (Clipper, Pascal, etc.). A utilização de arquivos com formatos especiais possibilita incluir nos dados informações úteis que podem facilitar a manipulação da informação. A linguagem C/C++ não possuem um formato especial em seus arquivos, porém é possível criar interpretadores que possibilitem a leitura de arquivos com formatos especiais.

Existem 3 tipos básicos de arquivos: I/O com stream, portas de comunicação e de "baixo nível". Os dados sequenciais podem atingir qualquer tamanho e formato e são os mais empregados quando se deseja total portabilidade entre arquivos. Ao manipular arquivos sequenciais é possível ler, escrever e anexar informações ao arquivo. Deve-se notar que existe ainda a possibilidade de acessar os dados através de um *buffer* (memória temporária RAM que armazenará os dados até que se encontre cheia, só então os dados serão descarregados no disco), o qual irá permitir uma maior velocidade no processamento de informações do arquivo. A utilização de *buffers* é arriscada caso não seja corretamente "esgotado", pois a informação se encontra somente na memória RAM e caso não seja passada para o disco poderá ser perdida, em casos críticos a utilização de "baixo nível" talvez se mostre mais adequada.

A biblioteca *stdio.h* possui algumas propriedades de grande importância, como a indicação de fim de arquivo (EOF) e deve ser anexada se possível.

Alguns dos comandos mais utilizados para a manipulação de arquivos em ANSI-C estão descritos a seguir:

Abrir	Fechar	Escrita	Leitura	Manipulação
fopen fdopen freopen	fclose fcloseall	não formatadas: printf fwrite caracteres: putc fputc putchar fputchar putch string: puts fputs inteiro: putw	não formatadas: scanf caracteres: getc fgetc getchar fgetchar getch string: gets fgets inteiro: getw	fim de uma stream: feof remover arquivo: remove renomeia arquivo: rename controle do buffer: setbuf setvbuf ungetch flushall

### 16.1-Manipulação de arquivos com a biblioteca *stdio.h*

A variável do tipo *FILE* é encontrada na biblioteca *stdio.h*, assim como os comandos apresentados na tabela acima. A variável será declarada como um ponteiro, o qual irá “apontar” para o arquivo onde se deseja gravar ou ler os dados. Os dados de arquivos podem ser abertos para leitura, escrita ou leitura/escrita. Para determinar a forma com a qual o arquivo será manipulado utiliza-se uma função de abertura, a mais comum é a função *fopen*:

**<variável do tipo file>=fopen(“<nome do arquivo e caminho>”, “<parâmetros>”);**

Onde os parâmetros são:

#### Arquivos texto e binário

- “r” - apenas leitura (arquivo existente).
- “w” - somente escrita (cria um novo arquivo).
- “a” - anexar dados ao fim do arquivo.
- “r+” - atualização (lê/escreve) arquivo existente.
- “w+” - atualização (lê/escreve) arquivo existente.
- “a+” - atualização de um arquivo/anexação.

#### Arquivos texto

- “rt” - somente leitura de um arquivo existente.
- “wt” - cria um novo arquivo, somente saída.
- “at” - modo de junção, escreve no fim do arquivo.
- “rt+” - lê/escreve em arquivo já existente.
- “wt+” - lê/escreve, cria um novo arquivo.
- “at+” - atualização ao final de um arquivo já existente, ou criação de um novo.

#### Arquivos binário

- “rb” - somente leitura de um arquivo existente.
- “wb” - cria um novo arquivo, somente saída.
- “ab” - modo de junção, escreve no fim do arquivo.
- “rt+b” - lê/escreve em arquivo já existente.
- “wt+b” - lê/escreve, cria um novo arquivo.
- “at+b” - atualização ao final de um arquivo já existente, ou criação de um novo.

É de suma importância especificar o local onde se deseja ler/gravar o arquivo, ou ele será lido/criado no mesmo local onde está o programa executável.

O final do arquivo deverá ser marcado com um carácter nulo “\0”, existe a definição deste caractere como NULL em ANSI-C, a presença deste define o final do arquivo. Existe a possibilidade de observar o comportamento do arquivo durante sua execução, são o comando *stderr* define se ocorreu algum erro durante a tentativa de aplicar o comando *fopen*.

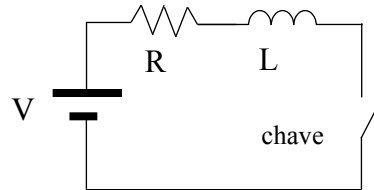


Figura 11 – Circuito R-L, com alimentação C.C.

Supondo o circuito apresentado na fig. 11, é possível determinar o comportamento da corrente elétrica em função do tempo, após o fechamento da chave:

$$R \cdot i + \frac{L \cdot di}{dt} = V$$

$$\text{Cujas solução é: } i = \frac{V}{R} (1 - e^{-(R/L) \cdot t})$$

O objetivo deste trabalho não é discutir a equação apresentada, mas criar um programa que utilizando esta equação possa determinar dados  $i \times t$ , para o problema apresentado. Os dados calculados serão armazenados em um arquivo. Os valores de R, L e V devem ser conhecidos e o tempo inicial considerado será  $t=0$ . O programa, está apresentado a seguir:

```
//Programa que calcula a curva i(t) de um circuito RL.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ()
{ float V,R,L,I,t,tfinal,dt;
FILE *saida;
//Abertura do arquivo de saida (i(t).dat).
if ((saida=fopen("i(t).dat","wt"))==NULL)
{printf("Nao foi possivel criar o arquivo!."); return;}
printf("Entre com o valor de R:"); scanf("%f",&R);
printf("Entre com o valor de L:"); scanf("%f",&L);
printf("Entre com o valor de V:"); scanf("%f",&V);
printf("Entre com o valor de tfinal:"); scanf("%f",&tfinal);
printf("Entre com o valor de dt:"); scanf("%f",&dt);
t=0;
do
{ I=V/R*(1-exp(R/L*t*-1));
fprintf(saida,"%f %f\n",t,I);
t+=dt;} while (t<=tfinal);
fprintf(saida,NULL);
fclose(saida);
return 0;}
```

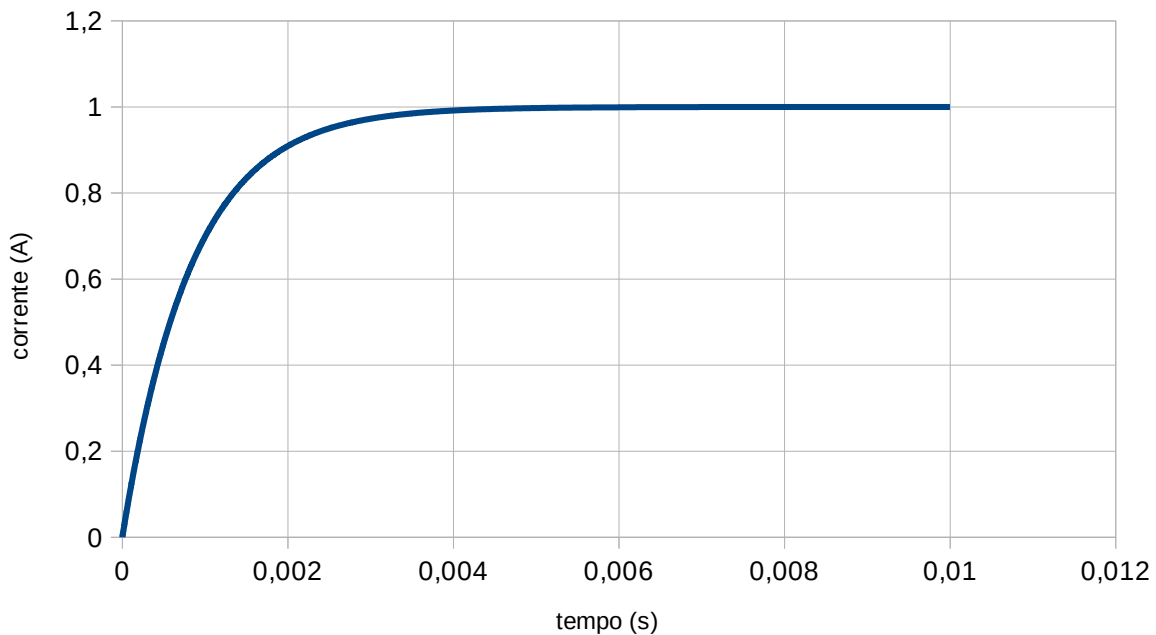


Figura 12 – Gráfico obtido do programa para cálculo de  $i(t)$  em circuitos R-L.

Supondo que os valores de  $R=12\Omega$ ,  $L=0,01H$ ,  $V=12V$ ,  $T_{\text{final}}=0,01$  e  $\Delta t=0,00001$ , este programa gerou um arquivo com os dados de  $i(t)$  apresentado na fig. 12.

A seguir é apresentado um exemplo da duplicação de um arquivo em ANSI-C:

```
/* Programa que duplica um arquivo. */
#include <stdio.h>
int main()
/* São duas as variáveis FILE: in (arquivo a ser duplicado) e out (arquivo que receberá os
dados duplicados). */
{ FILE *in, *out;
/* Abertura dos arquivos no formato texto para leitura/escrita. */
  if ((in = fopen("i(t).dat", "rt"))== NULL)
  {fprintf(stderr, "Não foi possível abrir o arquivo \n");
   return; }
  if ((out = fopen("i(t).bak", "wt"))== NULL)
  {fprintf(stderr, "Não foi possível criar o arquivo.\n");
   return;}

  while (!feof(in)) /* Procede enquanto não ocorrer fim de arquivo. */
  {fputc(fgetc(in), out);
   printf ("Escrevendo dados!\n");}

  fclose(in);
  fclose(out);
  return 0;}
```

Nota-se que a leitura/escrita ocorre no formato *texto*. É possível criar arquivos no formato binário, bastando utilizar o formato adequado.

## 16.2- Manipulação de arquivos em C++ - biblioteca “*fstream.h*”

Para a manipulação de arquivos em C++ é possível manipular as *streams* de arquivos utilizando a biblioteca *fstream*. As classes básicas presentes na biblioteca são:

- *fstream* - liga um arquivo a entrada e saída de dados
- *ifstream* - liga um arquivo a entrada de dados
- *ofstream* - liga um arquivo a saída de dados

Estes comandos básicos são associados a uma variável, que irá indicar o arquivo referido pelo mesmo, como na estrutura apresentada a seguir para abertura:

```
std::ofstream <var_arq> (“<nome e caminho do arquivo>”,std::ofstream::comando);  
...    <var_arq>.close();
```

Após associar os arquivos a um nome a informação poderá ser manipulada através dos comandos:

- **in** (input) - Entrada.
- **out** (output) – Saída.
- **binary** - Operação no modo binário.
- **ate** (at end) – Saída inicia no fim do arquivo.
- **app** (append) – Todas operações ocorrem no final do arquivo, anexando se já existir conteúdo.
- **trunc** (truncate) – É descartado todo conteúdo do arquivo que existia antes de sua abertura.

É possível ainda executar separadamente as instruções:

```
std::ofstream <var_arq>;  
    <var_arq>.open(“<nome e caminho do arquivo>”);  
...    <var_arq>.close();
```

A seguir é apresentado um exemplo da utilização da biblioteca *ifstream*, onde um arquivo é aberto para inserir o texto “Isto é um teste”:

```
// Escrita em um arquivo utilizando C++  
#include <fstream>  
  
int main () {  
    std::ofstream arquivo ("teste.txt", std::ofstream::out);  
  
    arquivo << "\n\nIsto é um teste! \n\n";  
  
    arquivo.close();  
  
    return 0;}
```

## 17-Estruturas e uniões

É possível criar uma estrutura semelhante a utilizada em banco de dados na linguagem C/C++, através da função *struct*. A estrutura deve ser declarada como uma função e conter um nome e as variáveis assim como a classe a que elas pertencem. A forma básica apresentada é:

```
struct <nome da estrutura> { <tipo da variável 1> <nome da variável 1>;  
    <tipo da variável 2> <nome da variável 2>;  
    <tipo da variável 3> <nome da variável 3>;  
    ...  
    <tipo da variável n> <nome da variável n>; }<estrutura 1>[número de dados];
```

A partir da declaração as referências a cada variável contida na estrutura é realizada através da citação:

```
cin>> <nome da estrutura>.<nome da variável>;
```

Pode-se anexar mais estruturas semelhantes a declarada, bastando incluí-las ao final, separadas por vírgulas. As estruturas podem se encontrar dentro de funções neste caso serão apenas internas a estas funções. É possível também aninhar uma estrutura dentro de outra, isto é uma estrutura pode conter entre suas variáveis outra estrutura.

Um exemplo da utilização de *struct* em C++ com a criação de um banco de dados com nome, endereço, idade e salário é apresentada a seguir:

```
#include <iostream>  
  
// Declaração da estrutura (global).  
struct tipo {char nome[15];          char endereco[20]; int idade;    float salario;};  
// Programa principal  
int main ()  
  
{ int n,i;  
  
// É criada a variável dados com até 40 elementos do tipo da estrutura  
struct tipo dados[40];  
std::cout<<"Número de dados:";  
std::cin>>n;  
  
//São lidos os elementos da estrutura.  
for (i=0;i<=n;i++)  
{std::cout<<"Nome:";      std::cin>>dados[i].nome;  
std::cout<<"Endereço:";  std::cin>>dados[i].endereco;  
std::cout<<"Idade:";      std::cin>>dados[i].idade;  
std::cout<<"Salario:";    std::cin>>dados[i].salario;}  
//É mostrado na tela todos os elementos que constam da estrutura lida.  
for (i=0;i<=n;i++)  
{std::cout<<dados[i].nome<<"\n";      std::cout<<dados[i].endereco<<"\n";  
std::cout<<dados[i].idade<<"\n";  std::cout<<dados[i].salario<<"\n";}  
return 0;}
```

Nota-se no exemplo apresentado que a variável que assume o tipo da estrutura é um vetor. É permitido que vetores assumam estruturas, neste caso é necessária a presença

de índices junto a variável indicando qual a posição do vetor está sendo analisada.

As uniões são declaradas de forma idêntica as estruturas, o nome *union* é utilizado, porém não operam de forma semelhante a estrutura pois a união contém vários tipos de dados, porém só pode utilizar um deles de cada vez. A inclusão de outras uniões não faz sentido neste caso.

Pode-se dizer que a *union* cria uma variável do tipo mista, que pode tomar apenas uma das formas utilizadas. Tomando-se o exemplo onde diversos tipos de variáveis são utilizados em uma *union*, previamente declarada, apenas a última forma utilizada é armazenada. Para demonstrar que apenas um valor é armazenado, posteriormente é solicitado que se mostre os diferentes tipos de variáveis da *union*.

```
#include <iostream>
// Declaração da union (global).
union tipo { float a; int b; char c;};
int main ( )
{ union tipo teste;
  std::cout<<"Entre com um número real:";      std::cin>>teste.a;
  std::cout<<"Entre com um número inteiro:";    std::cin>>teste.b;
  std::cout<<"Entre com um caractere:";         std::cin>>teste.c;

  // São apresentados os valores de cada tipo da union após a leitura. Nota-se o erro na
  saída
  // destes valores.
  std::cout<<"\n\n    Valores após a última leitura:\n"<<std::endl;
  std::cout<<"union float: "<<teste.a<<"\n";
  std::cout<<"union int: "<<teste.b<<"\n";
  std::cout<<"union char: "<<teste.c<<"\n";
  return 0;}
```

Para o exemplo acima em uma execução foram obtidos os resultados apresentados a seguir:

**Entre com um número real:12.45**

**Entre com um número inteiro:23**

**Entre com um caractere:x**

**Valores após a última leitura:**

**union float: número aleatório**

**union int: 120**

**union char: x**

Nota-se claramente o erro dos valores apresentados, devido a particularidade da função *union*.

As variáveis do tipo *union* podem também ser apresentadas sob a forma vetorial, porém continuam armazenando apenas um valor.

## Bibliografia

Jorge, D.C., *Introdução a Programação de Computadores na Engenharia*, 1998.

Kelly-Bootle, S., *Dominando o Turbo C*. Rio de Janeiro : Ciência Moderna Ltda, 1989.

Kester, W. *DATA CONVERTER HISTORY* , Penn State College of Engineering, 2006, <<http://www.cse.psu.edu/~chip/course/analog/lecture/ADChistory.pdf>>, Acesso em: 16 de julho de 2020.

Pappas, C. H. e M., William H., *Turbo C++ completo e total*. São Paulo : Makron, McGraw-Hill, 1991.

Press, W. H.; Teukolsky S. A.; Vetterling, W. T.; Flannery B. P., *Numerical Recipes in C*. Estados Unidos da América: Cambridge Press, 1994.

Ross, P. W., *The Handbook of Software for Engineers and Scientists*. Estados Unidos da América: CRC Press, Inc., 1996.

O'Regan, G., *A brief history of computing*, Springer, 2008.

Schildt, H., *C completo e total*. São Paulo : Makron, McGraw-Hill, 1990.