



# Programação Estruturada

Cristiane Koehler  
Eduardo Cardoso Melo

Curitiba  
2016

K77p Koehler, Cristiane  
Programação estruturada / Cristiane Koehler, Eduardo Cardoso  
Melo. - Curitiba: Fael, 2016.  
188 p.: il.  
ISBN 978-85-60531-52-3

1. Programação (Informática) 2. Linguagem estruturada I. Melo,  
Eduardo Cardoso II. Título

CDD 005.113

---

Direitos desta edição reservados à Fael.

É proibida a reprodução total ou parcial desta obra sem autorização expressa da Fael.

#### FAEL

<b>Direção de Produção</b>	Fernando Santos de Moraes Sarmento
<b>Coordenação Editorial</b>	Raquel Andrade Lorenz
<b>Revisão</b>	FabriCO
<b>Projeto Gráfico</b>	Sandro Niemicz
<b>Capa</b>	Vitor Bernardo Backes Lopes
<b>Imagem Capa</b>	Shutterstock.com/dabobabo/Svetoslav Radkov
<b>Diagramação</b>	FabriCO
<b>Arte-Final</b>	Evelyn Caroline dos Santos Betim

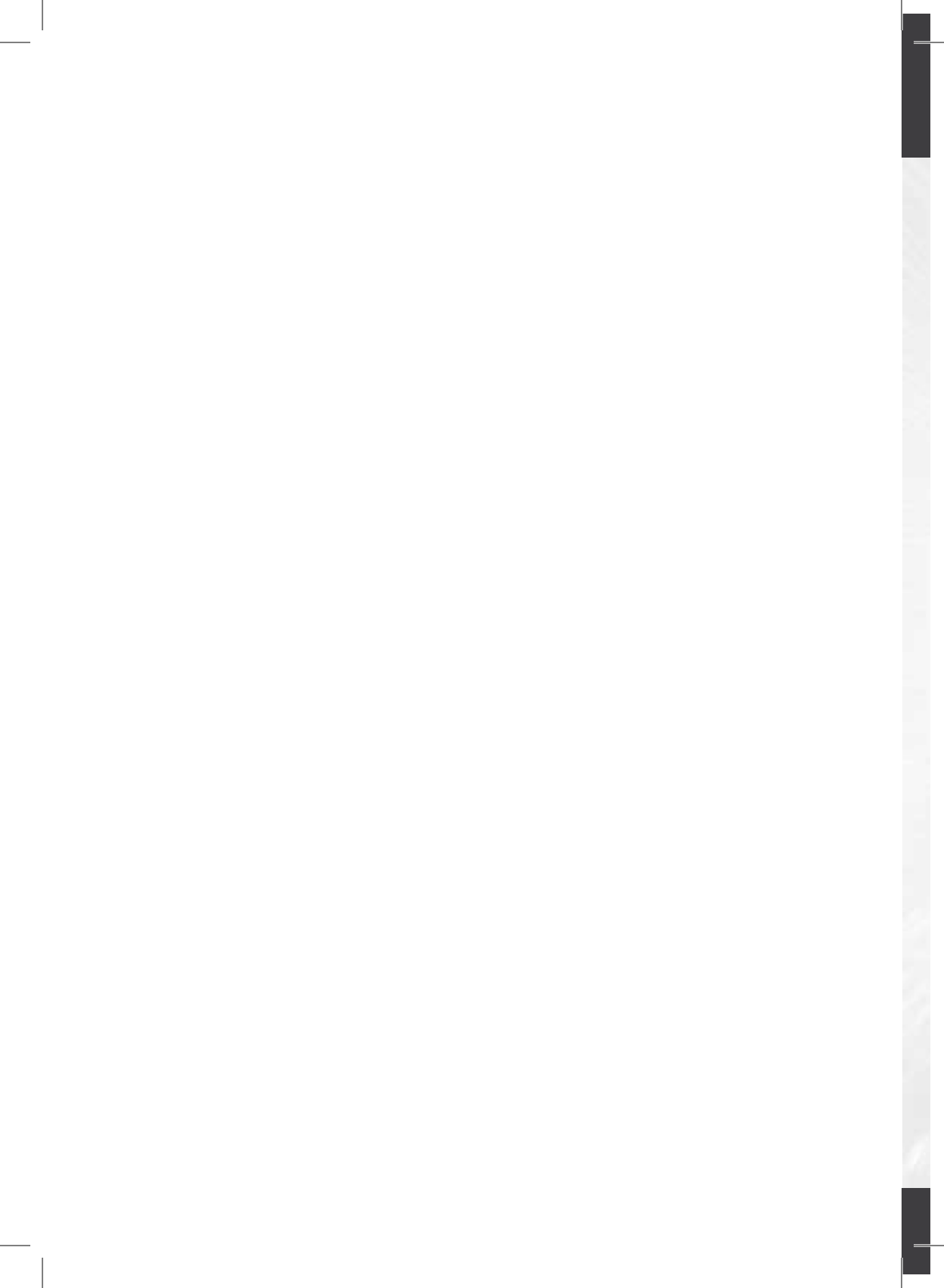
# Sumário

CARTA AO Aluno | 5

1. TIPOS DE Linguagens de Máquina e suas Características | 7
2. INTRODUÇÃO A Linguagem de Programação em C e suas Características | 25
3. CONHECENDO A Ferramenta Dev-C | 47
4. ESTRUTURAS DE Seleção na Linguagem de Programação Estruturada C | 59
5. ESTRUTURAS DE Repetição na Linguagem de Programação Estruturada C | 69
6. UTILIZAÇÃO DE Vetores Unidimensionais e Multidimensionais | 79
7. APLICAÇÕES DE Funções e Procedimentos | 99
8. UTILIZANDO PESQUISA e ordenação | 115
9. ESTRUTURAS DE Dados Heterogêneos | 133
10. INTRODUÇÃO À Programação Básica em Interface Gráfica | 151
11. PONTEIROS | 169

CONCLUSÃO | 183

REFERÊNCIAS | 185



# Carta ao Aluno

Prezado aluno.

A disciplina de Programação Estruturada tem fundamental importância para o seu curso. Nela, você desenvolverá o raciocínio lógico, baseado em técnicas de programação e utilização de uma linguagem de programação estruturada, denominada de C, que o levará a mergulhar no universo do mundo da programação computacional. Você também irá aprender sobre os diversos aspectos técnicos, as diferenciações existentes, os tipos de linguagens utilizadas para programação de computadores, emprego de expressões matemáticas, resolução de problemas computacionais simples, utilização das estruturas de seleção e suas particularidades, uso de estruturas de repetições para resoluções de problemas computacionais complexos e outros. Além disso, você poderá aprender a armazenar

informações em vetores de dados, tendo a base de como os sistemas gerenciadores de banco de dados trabalham, técnicas e procedimentos de otimização de programação estruturados e outras técnicas que irão prepará-lo para um mercado promissor.

A linguagem de programação estruturada utilizando o C é amplamente utilizada nos meios acadêmicos e no mercado de trabalho, pois é introdutória para a utilização de outras linguagens de programação como Python, Java, #Net.

Orientamos você, caro aluno, a ler com o máximo de atenção e cuidado todos os capítulos, uma vez que o conhecimento da linguagem é progressivo. Não deixe de fazer e refazer todos os estudos de casos propostos. Procure utilizar todos os comandos e sintaxe propostos nas atividades

“A arte de programar consiste em organizar e dominar a complexidade.”

Ao final da disciplina, você deverá ser capaz de desenvolver programas computacionais complexos utilizando a linguagem de programação C++.

Bons estudos!

# 1

## Tipos de Linguagens de Máquina e suas Características

PARA INICIARMOS OS estudos sobre programação estruturada, é importante você compreender o funcionamento do computador, pois desenvolveremos sistemas computacionais com esta ferramenta. Basicamente o computador é constituído de quatro unidades básicas: unidade de entrada, unidade central de processamento, unidade de memória, unidade de saída.

A *unidade de entrada* permite utilizarmos dispositivos para a introdução de dados a serem processados pelo computador, como por exemplo: mouse, teclado, microfone, touch screen etc. A unidade central de processamento conhecida como CPU- Unidade Central de Processamento- é responsável por processar os dados enviados através dos dispositivos de entrada de dados.

A unidade de memória é responsável por guardar as informações que serão processadas pela unidade de processamento central e a unidade de saída fica responsável por dar o resultado da operação solicitada pelo usuário. Sobre estes dispositivos podemos destacar: as impressoras, os autôfalantes, os monitores, entre outros.

Para que todas estas unidades realizem seus trabalhos de forma dinâmica, as comunicações utilizadas para o entendimento das tarefas solicitadas devem ser apresentadas de forma clara e objetiva. Para isso utiliza-se linguagens que possam ser compreendidas por todas as unidades envolvidas no processamento das informações. A forma como é realizada a comunicação entre as unidades envolvidas nesse processo é o principal tema que você irá estudar neste primeiro capítulo.

### Objetivo de Aprendizagem:

- × compreender, descrever e contextualizar os tipos de linguagens utilizadas para Programação de Computadores.

## 1.1 Estrutura de um Sistema Operacional

Para compreender o funcionamento de uma linguagem de programação, você terá que entender o conceito de linguagens de máquina e suas características. Esta concepção passa pela compreensão dos sistemas operacionais que são responsáveis por gerenciar todas as operações e solicitações pedidas por um usuário ou programa.

Um Sistema Operacional pode ser definido como um dispositivo lógico-físico que realiza trocas entre o usuário e o computador. Nele são inseridos alguns softwares que administram todas as partes do sistema e também as apresentam de forma amigável ao usuário. Segundo Tanenbaum (2002), um sistema operacional tem duas funções não relacionadas: estender a máquina e gerenciar recursos.

Para entender os recursos, os sistemas operacionais devem conversar com as memórias, mouse, teclado, vídeos, impressora, discos rígidos. Estes sistemas não farão essa administração sem entender a máquina, ou sem falar a linguagem na qual a máquina foi projetada, isto é, a linguagem do hardware. Este



gerenciamento de hardware e usuário deve ser realizado de forma eficiente e transparente ao usuário.

Tanenbaum (1997) afirma que os sistemas operacionais surgiram com dois objetivos principais: criar uma camada de abstração entre o hardware e as aplicações, e gerenciar os recursos de forma eficiente (TANENBAUM AND WOODHULL, 1997). A palavra chave desse contexto é “eficiência”, por isso que a cada ano surgem versões diferentes de sistemas operacionais, pois os recursos de hardware aumentam, os softwares evoluem e os sistemas operacionais precisam ser atualizados.

Para a realização da administração do hardware e recursos de software existe uma divisão dentro do sistema operacional denominada de Kernel ou núcleo. Segundo Tanenbaum (2003), Kernel é o núcleo do sistema operacional, gerencia todo o hardware assim como provê as chamadas de sistemas (syscalls). A grande maioria dos programadores não fazem uso das System Call, embora a maioria dos programas utilizem estes recursos.

---

---

**System Call:** É uma implementação de mecanismos de proteção ao núcleo do sistema e de acesso aos seus serviços. Essa chamada de sistema tem por importância garantir a integridade do sistema.

Para cada serviço existe uma System Call associada e cada sistema operacional tem seu próprio conjunto de chamadas.

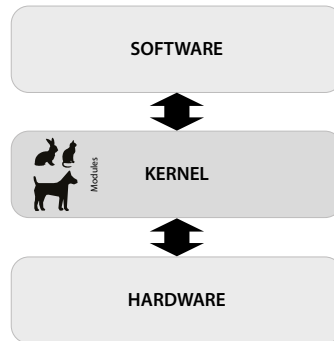
SILBERSCHATZ, A., GAGNE, G., GALVIN, P. B. (2004).

---

---

O Sistema Operacional instalado em uma máquina possui as seguintes divisões: o Kernel que é parte essencial do sistema operacional que realiza a comunicação entre o hardware (disco rígido, placa de som, modem, slots etc), e o software da máquina (aplicativos, editores de textos jogos). A forma como é realizado o gerenciamento destes recursos podem variar entre os sistemas operacionais existentes. Hoje existem no mercado alguns tipos de sistemas operacionais mais conhecidos. São eles: Windows, IOS, Linux, e a forma como gerenciam seus recursos computacionais são completamente diferentes (TANENBAUM, 2003).

Figura 1: Divisão do sistema operacional.



Fonte: Viva Linux (2015).

Para que você compreenda como funciona um sistema operacional é necessário saber como são administrados os recursos de hardware e os recursos de softwares. Para os recursos de hardware existem gerência de processos, gerência de memória, gerência de controle de sistema de arquivos, operações de entrada e saída. Para os recursos de softwares utilitários temos manipulação de arquivos, informações sobre o sistema, suporte para linguagens de programação, carregamento e execução de programas.

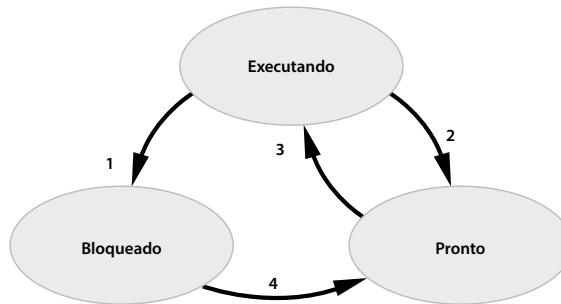
- × **Gerência dos Processos:** segundo Tanenbaum (2003) um processo é um programa em execução. Na programação estruturada, o programador poderá declarar variáveis de diversos tipos como: inteiro, real, caracter, lógico. Quando o programa é executado na memória da máquina, essas variáveis passarão por alguns processos antes de sua execução.

Para o programador é importante conhecer os estados destes processos. Quando a máquina é ligada, várias tarefas são inicializadas e essas tarefas devem ser processadas pela CPU e alguns critérios e procedimento são estabelecidos para essa execução.

Para cada processo criado é atribuído um número único, chamado de Identificador de Processo. Por essa identificação será gerenciada a prioridade de sua execução deixando as variáveis em três estados: pronto, executando, bloqueado. Este gerenciamento possibilita o computador executar várias tarefas ao mesmo tempo, enquanto se

lê um arquivo no disco rígido, um processo pode estar ocupando a memória naquele instante. Na figura abaixo, TANENBAUM (2000) apresenta estes processos e seus estados.

Figura 2: Estados dos Processos.



Fonte: Tanenbaum (2000).

Na figura 2 podemos ver como um processo pode estar executando e ser bloqueado (1). Quando isso acontece, o processador seleciona um outro processo na fila de pronto (2). Após essa seleção ele é executado (3) e o processo que estava bloqueado ou esperando alguma definição para ser executado entra novamente na fila de pronto para ser executado (4).

- × **Gerência de Memória:** Quando os processos são carregados para a memória principal da máquina (RAM), para entrar na fila de processamento e há pouco espaço, a memória pode ficar sobrecarregada. Como é resolvida essa situação? O sistema operacional realiza dois procedimentos: gerência de memória virtual com paginação e swapping.

Estas técnicas permitem o uso dos discos rígidos para armazenar as informações que não couberam na memória e realizar a troca de informações entre a memória e os discos através dessas técnicas (paginação e swapping), dando espaço para novos processos. Na prática, a paginação é realizada quando falta espaço na memória e usa o disco rígido criando uma memória virtual, e Swapping quando é realizado a troca das informações entre disco rígido e memória.



## Saiba mais

**Swapping:** Impõe aos programas um grande custo em termos de tempo de execução. Copiar todo o processo da memória para o disco e mais tarde de volta para a memória é uma operação demorada para que a máquina corresponda de forma eficiente.

Sistemas Operacionais (2008).



Atualmente cada vez mais os programadores necessitam de memórias e mais programas rodando simultaneamente para que a máquina corresponda de forma eficiente. Pensar na utilização da memória de forma eficiente é um dos deveres do programador. Requisitos como segurança, isolamento, performance, tratamento de erros entre outros são preocupações que devem prevalecer.

- × **Controle do Sistema de Arquivos:** Para a organização dos arquivos criados pelos usuários e sistemas, o controle do sistema de arquivo - estabelece em forma de estrutura de “árvores” um sistema chamado de diretórios. Os diretórios são arquivos que contém informações de como encontrar outros arquivos. Esta separação permite que haja uma -disposição dos arquivos no disco rígido sem comprometer o funcionamento do sistema operacional. Em um sistema de diretório organizado, na forma de árvore, como demonstrado na figura a seguir, qualquer arquivo ou subdiretório pode ser identificado de forma não ambígua, ou seja, não duplicada, através de um caminho especificado partindo da raiz, no caso do Windows o “C” da máquina e Linux a “/”.
- × **Operações de Entrada e Saída:** Segundo Tanenbaum (2000), uma das principais funções do Sistema Operacional é controlar todos os dispositivos de E/S (entrada e saída) e oferecer uma interface entre os dispositivos e o restante do sistema que seja simples e fácil de usar. Quando um programa é executado, haverá a possibilidade de desempenho de operações de acesso ao hardware da máquina como disco rígido, placa de vídeo.

Esta função recebe o nome de operações de entrada/saída (E/S) e pode envolver qualquer dispositivo físico instalado na máquina. O sistema operacional é o responsável por fornecer meios adequados para essa execução, criando a interação com cada dispositivo por meio de drivers e modelos abstratos que permitam agrupar vários dispositivos distintos sob a mesma interface de acesso. A preocupação do programador é sobre como utilizar as bibliotecas dos sistemas operacionais que acessam os recursos de hardware. A Linguagem C possui bibliotecas, que são coleções de ferramentas, que facilitam o acesso a esses recursos.

Existem os softwares utilitários que são programas utilizados para ‘rodar’ dentro do Kernel. Eles são utilizados para gerenciar e realizar manutenções nos computadores, como por exemplo a manipulação de arquivos que ajudam na manutenção e gerência de arquivos de dados. Pode realizar operações como incluir, apagar, copiar, ver, renomear e imprimir um arquivo.

Para o suporte de linguagens de programação, podemos citar os compiladores, montadores e interpretadores, além de carregamento e execução de programas. Quando o programa é carregado para a memória da máquina, ele deve ser compilado e montado pelo sistema operacional, que oferece condições para que essas operações sejam efetuadas eficientemente.

Figura 3: Estrutura de Diretório.



Fonte: Batisti (2000).

## Você sabia

Compilação: é o processo utilizado para a tradução de um código fonte em uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível (por exemplo, Assembly ou código de máquina). Completo e Total. Helbert Shildt (1996).

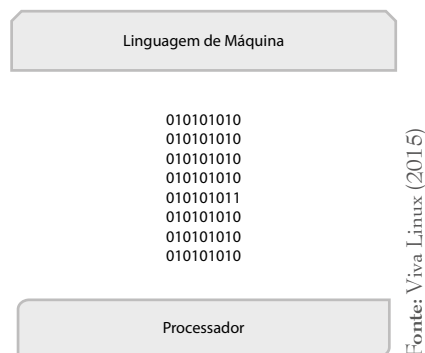
Existem no mercado diversos tipos de sistemas operacionais, para o estudante da linguagem C, é importante saber diferenciar as formas e interação com os aplicativos que serão desenvolvidos pois estes estão ligados diretamente ao desempenho.

### 1.1.1 Linguagem de Máquina

Você já viu anteriormente como o sistema operacional trabalha para organizar e gerenciar um computador. Como ocorre a comunicação entre os diversos dispositivos como placas mãe, processador, memórias. Os cientistas que projetaram os computadores estabeleceram **dois** símbolos básicos para a linguagem. Essa quantidade de símbolos foi escolhida pelo fato de que através de fenômenos físicos é muito fácil obter dois estados distintos e não confundíveis, como passar corrente elétrica/não passar corrente elétrica, estar magnetizado/não estar magnetizado, podendo cada um desses estados ser um dos símbolos.

Assim, a linguagem utilizada para comunicação interna num computador, chamada linguagem de máquina, possui apenas dois símbolos, zero e um (0,1) Evaristo (2001).

Figura 4: Fluxo de dados – Linguagem de máquina.



No mundo da computação todas as instruções realizadas através do processador são feitas por uma linguagem, que pode ser definida como um conjunto de instruções. São chamadas de códigos de máquina e representadas por sequências de bits, normalmente limitadas pelo número de bits do registrador principal da CPU. Esse código é chamado de código binário. Eles são formados por 0 e 1. Na figura a seguir podemos verificar o fluxo das informações através das linguagens de máquina com o processador.

Quando se fala em níveis, as linguagens de programação podem ser classificadas em dois grupos: linguagens de baixo nível e linguagem de alto nível. A linguagem de máquina é definida como uma linguagem de baixo

nível, pois sua compreensão para o ser humano é mais complexa por utilizar o sistema binário (0,1).

Para o desenvolvedor de programas realizar uma série de operações utilizando o sistema binário é muito complicado, pois demandaria muito tempo e pouca produtividade. Por este motivo foi criado uma linguagem denominada linguagem de montagem ou *Assembly Language*, composta de códigos mnemônicos, que são técnicas utilizadas para auxiliar o processo de memorização. Este recurso foi implementado mas não resolveu o problema da produtividade, pois a implementação nesse tipo de linguagem ainda gerava muito custos de programação.

## Importante

A grande vantagem da linguagem de montagem é a comunicação fácil com o hardware da máquina, pois é uma linguagem que trabalha em baixo nível. Podemos encontrar especificações de linguagem de montagem diferenciadas, ligadas ao hardware e isso depende de cada máquina específica. Cada arquitetura de computador tem sua própria linguagem de máquina e, portanto, a sua própria linguagem de montagem.

As linguagens de montagens diferem no número e no tipo de operações que suportam. Pode-se diferenciá-las pela quantidade e tamanho de registradores, operações que são realizadas e tipos de dados suportados. Para realizar programação com acesso diretamente ao hardware, é importante conhecer essa linguagem. Ela permite entender como programas escritos em linguagens de alto nível, como C ou Java, são traduzidos para a linguagem de máquina. Na figura 5, podemos observar como é realizada a programação na linguagem *assembly*, com seus comandos manipulando os registradores e memórias.

Figura 5: Exemplo de linguagem Assembly.

00000000	push	ebp
00000001	mov	ebp, esp
00000003	movzx	ecx, [ebp+arg_0]
00000007	pop	ebp
00000008	movzx	dx, cl
0000000C	lea	eax, [edx+edx]
0000000F	add	eax, edx
00000011	shl	eax, 2
00000014	add	eax, edx
00000016	shr	eax, 8
00000019	sub	cl, al
0000001B	shr	cl, 1
0000001D	add	al, cl
0000001F	shr	al, 5
00000022	movzx	eax, al
00000025	ret	

Fonte: Laifi (2015).

## 1.2 Linguagem de Programação de Alto Nível

Existem centenas de linguagens de programação desenvolvidas desde o início da computação. Essas linguagens foram agrupadas de acordo com suas características e a época em que foram desenvolvidas. São quatro gerações: linguagem de primeira geração, linguagem de segunda geração, linguagem de terceira geração, linguagem de quarta geração.

- × A linguagem de primeira geração teve início na década de 50 e tem como características a linguagem de máquina e assembly.
- × A linguagem de segunda geração é marcada pela programação multiusuário, destacando-se as linguagens Fortran, Cobol, Algol.
- × A terceira geração se deu nos anos de 1974 até 1986 e são caracterizadas pela grande capacidade procedural e estrutural de seus dados. C, Pascal, PL/1 e Modula-2 são as principais linguagens desta categoria, sendo que as duas primeiras continuam bastante usadas atualmente.
- × A quarta geração das linguagens de programação foram desenvolvidas a partir de 1986 e tiveram como características principais a geração de sistemas especialistas, o desenvolvimento de inteligência artificial e a possibilidade de execução dos programas em paralelo.

A linguagem de programação de alto nível possui esse nome, pois está mais próxima da linguagem humana. Quando utilizamos linguagem de programação classificada como linguagem de alto nível, estamos aproximando a linguagem humana ou linguagem natural como forma de elaboração de sistemas computacionais.

Para desenvolvimento na linguagem de alto nível, os recursos devem permitir ao desenvolvedor a utilização do raciocínio natural, usando palavras em inglês e notações algébricas de forma intuitiva. Dessa forma, sua produtividade em termos de desenvolvimento e praticidade aumenta consideravelmente em relação à linguagem de baixo nível e, também, favorece a portabilidade, pois os programas escritos em linguagens de alto nível são portáveis. Ou seja, são independentes de plataformas, o que significa que podem ser migrados de uma máquina para outra sem precisar de implementações.



Um grande avanço ocorreu na computação quando se conseguiu desenvolver programas que traduzissem instruções escritas originariamente numa linguagem dos seres humanos para a linguagem de máquina, EVARISTO (2001). Para utilizar todos os recursos, as linguagens de alto nível utilizam EDD's – Estruturas Dinâmicas de Dados, em que os dados são armazenados em um vetor ou pilha de dados e podem sofrer variações de tamanho no decorrer da execução do programa. Este recurso é de fundamental importância, pois permite aos programadores criar estruturas de dados variáveis que se adaptem às necessidades reais dos programa.

A desvantagem na utilização de linguagens de alto nível está associada a fatores de desempenho, mas essa desvantagem depende do tipo de aplicação que será desenvolvida. Por ser uma linguagem mais próxima ao hardware, as linguagens de baixo nível exigem menos conversões para a “linguagem de máquina” do dispositivo. Quando o código é escrito em uma linguagem de alto nível, muitas conversões são necessárias para conversar com o hardware, já que esta deve ser transformada em “linguagem de máquina”. Como essa geração é feita de forma automática, o código resultante acaba sendo maior do que se fosse escrito diretamente por um programador usando a linguagem de baixo nível.

As linguagens estruturadas possuem como características esconder as informações necessárias para realização de tarefas específicas, controle do fluxo do programa, desvios condicionais, laços de repetição, procedimentos e funções e sub-rotinas são elementos que podem ser utilizados para esconder informações e tornar a Linguagem Estruturada mais eficiente. A Linguagem C possui todas as características de uma linguagem estruturada, principalmente a utilização de sub-rotinas que podem ser empregadas em todos os blocos de programas.

Um das formas de executar o programa em uma máquina é através de compiladores e interpretadores. Segundo SCHILDT (1996), os termos compiladores e interpretadores referem-se a maneira de como um programa é executado. A forma como a linguagem de programação é compilada ou interpretada não é definida pela linguagem na qual é escrita, pois os compiladores e interpretadores são programas que operam sobre o código-fonte de seu programa.

Quando o interpretador realiza a operação de interpretar o código para a linguagem de máquina, ele fará a operação lendo linha por linha do código desenvolvido, executando as instruções contidas nessas linhas. Podemos visualizar esse processo na figura 6.

Figura 6: Processo simplificado Interpretador.



Fonte: Viva Linux. Disponível em: [www.vivaolinux.com.br](http://www.vivaolinux.com.br).

Acesso em: 29/01/2015.

Na figura 7 temos o processo de compilação que é realizado de forma diferente do processo de interpretação. Ele faz toda a operação de leitura do programa (código fonte), ou seja, realiza a leitura do programa desenvolvido pelo programador e transforma esse código em um programa denominado de código-objeto. Este código-objeto é interpretado pelo computador como tradução do código fonte e executado na máquina. A linguagem C, que iremos estudar, é considerada uma linguagem compilada.

Figura 7: Processo simplificado do Compilador.



Fonte: Viva Linux. Disponível em: [www.vivaolinux.com.br](http://www.vivaolinux.com.br).

Acesso em: 29/01/2015.

### 1.3 Sintaxe e Semântica de uma Instrução Em C

Quando estudamos uma linguagem de programação, estamos falando de regras, pois uma linguagem é um conjunto de regras sintáticas e semânticas usadas para definir uma forma de comunicação. Ao realizarmos uma comparação com a língua portuguesa, temos as regras sintáticas, segmento que estuda a função que as palavras desempenham dentro da oração, e as regras semânticas que trata do significado e das possíveis relações interpretativas das

palavras e sentenças.. Na linguagem estruturada C, não é diferente, essas duas regras são realizadas quando o programa é executado.

Para programar na linguagem C, deve-se utilizar essas regras compostas de sintaxe e semântica que ajudam a padronizar a linguagem para ser utilizada por todos os desenvolvedores. A sintaxe são as regras de como os comandos e termos escritos de uma linguagem devem ser construídos corretamente. Caso as regras sejam obedecidas pelos programadores, as operações serão executadas, caso contrário será apontado o erro de sintaxe.

Algumas linguagens apontam o tipo de erro ocorrido e a forma correta de utilizá-los. A sintaxe é um conjunto de regras formais que especificam a composição dos algoritmos a partir de letras, dígitos e outros símbolos. Na figura 8 a seguir podemos perceber que o símbolo ( { ) é utilizado na linguagem C para especificar o início do programa, e o símbolo ( } ) fecha a estrutura.

Ao observarmos a Figura 8, verificamos outro exemplo que poderia ocorrer, isto é, o uso do comando **else**. Caso ele estivesse escrito de forma errada, o programa não seria executado e apontaria o erro de sintaxe.

Figura 8: Estrutura Semântica da Linguagem C.

```
#include<iostream>
#include<math.h>
using namespace std;
int main () {
    int num;int x;
    cout<<"Digite o numero"<<endl;
    cin>>num;
    if {num<=1} {
        cout<<"Impressao:"<<1<<endl;
    }else if (num<=2&&num>1) {
        cout<<"impressao:"<<2<<endl;
    }else if (num>2 && num<=5) {
        x=pow (num,2);
        cout<<"impressao:"<<x<<endl;
    }else{
        x=pow (num,3);
        cout<<"impressao:"<<xx<<endl;
    }return 0;}
```

Fonte: Geeksbr (2015).

---

**Semântica:** Na linguística, a semântica se caracteriza como um dos componentes do conhecimento que tem a função de representar o significado. Na computação a semântica constitui um dos estágios de processamento, tanto na compreensão quanto na geração da linguagem natural.

<http://revistaseletronicas.pucrs.br/ojs/index.php/fale/article/viewFile/606/437>

---

Quando vamos executar alguma ação, usamos a lógica para sua execução. Quando estamos em casa e queremos uma pizza, temos que separar alguns itens e sequências lógicas para que essa ação seja realizada com sucesso. As principais ferramentas para executar essa ação serão:

- × chave do carro;
- × chave da porta da casa;
- × cartão de crédito ou dinheiro;
- × endereço da pizzaria;
- × escolha do sabor;

Podemos afirmar que na construção de um programa de computadores, temos que declarar as ferramentas para resolução do problema proposto como citamos no exemplo da pizza. Para que a ação comprar uma pizza seja realizada com sucesso, é fundamental que o uso e a sequência das instruções sejam realizadas logicamente.

Na programação de computadores não é diferente. A utilização da sequência lógica para a solução computacional resultará a correta resolução do problema. As correções da semântica de um algoritmo são mais difíceis que as correções da sintaxe, por que ela depende do conhecimento do problema computacional proposto. O usuário poderá compilar o programa e este funcionar perfeitamente de forma lógica, mas se não resolver o problema proposto, o programa realizado não atingirá seu objetivo.

## 1.4 Por que Aprender a Programar em C?

A linguagem de programação em C foi criada em meados do ano de 1970, por Dennis Ritchie, no AT&T Bell Labs. O objetivo foi desenvolver a implementação do sistema operacional UNIX, que na época era o sistema operacional mais utilizado para grandes servidores. A linguagem C é derivada de uma outra linguagem ainda utilizada na Europa denominada de BCPL, que teve como criador Martin Richards.

No final da década de 70, a linguagem C era uma das linguagens mais utilizadas para o desenvolvimento de aplicações, fazendo frente com seu principal concorrente a linguagem BASIC. No início dos anos 80, houve a neces-

sidade da criação de padrões para que a linguagem fosse utilizada de forma mais profissional, pois haviam muitas discrepâncias em sua utilização para o mercado comercial. Para resolver essa questão a ANSI (American National Standards Institute) estabeleceu em 1983 o Padrão ANSI C.

Segundo SCHILDT (1996), a linguagem C pode ser denominada uma linguagem de nível médio. Isso não significa que a linguagem não seja poderosa em recursos, pois ela é o ponto de partida para linguagens mais avançadas com Java, Python e # Net. A linguagem C também não pode ser comparada com a linguagem *Assembly*, pois está em um nível acima dessa, já que possui portabilidade. Conforme SHILDT(1996), portabilidade significa que é possível adaptar um software escrito para um tipo de computador a outro. Ampliando esse conceito, a ISO/IEC 9126-1 define portabilidade como um conjunto de atributos que evidenciam a capacidade do software ser transferido de um ambiente para outro sem muitas adaptações.


A linguagem C é denominada uma linguagem estruturada. Já a definição de linguagem estruturada é dada por Schidt (1996), em seu livro: *Completo e Total*, onde ele declara que a estruturação está ligada à compartimentalização do código e dos dados e habilidades de seccionar e esconder o resto do programa e as informações necessárias à realização de uma tarefa específica. Em outras palavras, utiliza comandos com uma entrada e uma saída. Outra definição dada a programação estruturada é a ideia do programa dividido em pequenos procedimentos estruturados, chamados subrotinas ou funções.

O uso de variáveis locais, globais e temporais, bem como a utilização de sub-rotinas, compartilhamento de seções de códigos, laços de repetições como **while** (enquanto), **do while** (enquanto faça) e **for** (para), são características de linguagens estruturadas.



### Saiba mais

Programação estruturada é uma forma de programação de computadores que preconiza que todos os programas possíveis podem ser reduzidos a apenas três estruturas: sequência, decisão e iteração (esta última também é chamada de repetição), a definição foi desenvolvida por Michael A. Jackson no livro "Principles of Program Design" de 1975.

























A capacidade de criar rotinas e sub-rotinas dentro dos programas é a característica principal da programação estruturada. Apesar da programação orientada a objetos ser utilizada hoje como substituta da programação estruturada, sua utilização é muito influente, pois grande parte das pessoas ainda aprendem programação através da programação estruturada. As universidades e escolas especializadas em treinamento de linguagem de computação, na grande maioria, adotam a Linguagem C como forma de desenvolvimento do raciocínio lógico, já que a resolução de problemas é relativamente mais simples e direta, diferentemente da programação orientada a objetos em que a forma de pensar para resolução de problemas é mais complexa.

### Você sabia

O conceituado site TIOBE, especializado na avaliação e qualidade de software, classificou a linguagem C como uma das linguagens mais utilizadas entre Janeiro de 2014 e Janeiro de 2015. Dentro do universo do mundo da programação, a linguagem C é utilizada por 16,7% dos desenvolvedores. TIOBE. Disponível em: <http://www.tiobe.com/index.php/content/company/Home.html>. Acesso em: 22/01/2015.

Figura 9: Classificação da Linguagem C ano 2014.

Language Rank	Types
1. Java	   
2. C	  
3. C++	  
4. Python	 
5. C#	  
6. PHP	
7. Javascript	 
8. Ruby	 
9. R	
10. MATLAB	

Fonte: IEEE, (2014).

A revista IEEE - Instituto de Engenheiros Eletricistas e Eletrônicos, principal revista especializada no assunto, realizou em 2014 uma pesquisa sobre as principais linguagens utilizadas no mundo da computação. O estudo dividiu as linguagens em quatro categorias: web, desktops, dispositivos móveis e software embarcado. Nesta pesquisa, a linguagem C aparece em segundo lugar, contrariando a revista TIOPE que classifica a linguagem C como a mais utilizada linguagem de programação. Na figura 9, podemos ver

ficar as classificações das outras linguagens como: java, C++ e Python, todas essas linguagens possuem similaridade com a linguagem C.

Podemos perceber o quanto é importante o domínio dessa linguagem para o desenvolvedor de programas. Ela é a base para qualquer linguagem de programação. Apesar de ser uma linguagem desenvolvida na década de 70, a linguagem C ainda continua sendo bastante utilizada no mercado de software.

## Resumindo

Neste capítulo analisamos as principais estruturas de sistemas operacionais e como é possível trabalhar de diferentes formas com o hardware para resolução de gerenciamento das atividades computacionais. A partir dessas noções é possível ter uma visão básica de como o processador resolve as questões de processamento. Este conhecimento é importante para entender os tipos de linguagens de baixo nível e de alto nível, e a forma como elas se comunicam com o sistema operacional e o hardware, bem como suas diferenciações em relação à produtividade, quando se trata de desenvolvimento de sistema.

A definição de sintaxe e semântica, que você também viu nesse capítulo, será muito útil na hora de colocar a mão na massa para entender como será compilado o programa. Não menos importante foi o aprendizado sobre a utilização da linguagem C, trazendo um histórico sobre a linguagem e sua posição no mercado de trabalho.





# 2

## Introdução a Linguagem de Programação em C e suas Características

TODA A LINGUAGEM de programação apresenta uma estrutura. Para que se dê início à atividade de programação de computadores, é indispensável conhecer a estrutura e as características gerais da linguagem de programação escolhida. A Linguagem Estruturada em C, possui particularidades importantes que devem ser aprendidas e assimiladas para que se possa dominar a técnica de programação. Neste capítulo, aprenderemos sobre como aplicar os principais tipos de expressões lógicas e aritméticas da linguagem C, variáveis simples, constantes, expressões aritméticas e lógicas, comandos de entrada e saída de dados, que possibilitam a um programa retornar aos usuários os resultados dos processamentos realizados. Também estudaremos os comandos de atribuições - que permitem alterar o conteúdo da variável na memória do computador - estudaremos como realizar operações matemáticas e conversões de tipos de dados

– isto é, alterar o tipo de dado de uma variável em tempo de execução do programa. A intenção nesse capítulo é proporcionar uma base sólida sobre as características da linguagem C, e isso somente será possível se a motivação e compromisso de resolver os estudos de caso forem realizados com entusiasmo. Então, vamos lá!

### Objetivo de Aprendizagem:

- × Compreender, saber e aplicar os principais tipos de expressões lógicas e aritméticas da linguagem C.

## 2.1 Estrutura de um Programa em C

Um programa é composto de uma ou mais funções. Sendo que, na linguagem de programação C, a única função obrigatória é a **main()**. Esta é a primeira função a ser chamada toda vez em que o programa é executado.

Segundo Manzano (2002), as funções são as entidades operacionais básicas dos programas em C, que por sua vez são uma união de uma ou várias funções executando cada qual o seu trabalho. No início de um programa em C, toda função deve ter o seu nome precedido de parênteses “()”, indicando que se trata de uma função. Os símbolos “{” e “}” representam o início e o término da função respectivamente.

Na Figura 1, o programa mostra a estrutura básica de um programa escrito na linguagem C. A palavra reservada **void** na frente da função **main()** indica que ela não retorna valor.

**Atenção!** Para comentar algum item, devemos usar: // e tudo o que for escrito após // será ignorado pelo compilador.

Figura 1 - Função main

```
void main ( )  
{  
  
}
```

Fonte: Elaborada pelo autor (2015).

Note que a palavra **main** é muito importante e aparece sempre e uma única vez em qualquer programa em C. Na sequência da palavra **main** existe um par de parênteses que indicam para o compilador que se trata de uma função (*function*). Em atividades posteriores mostraremos em detalhes o que é uma função. Todas as vezes que iniciar um programa em C considere incluir **SEMPRE** este par de parênteses em todos os seus programas.

## 2.2 O que é uma variável

Já comentamos que a memória é uma das unidades básicas do computador e cuja finalidade é armazenar dados e informações que serão manipulados pelo processador. Todos os programas que serão executados em um computador devem estar armazenados na memória da máquina.

Para que seja possível armazenar dados e informações, a memória é dividida em partes, chamadas posições de memória. O sistema operacional que gerencia o sistema de computação pode acessar cada uma dessas posições para o armazenamento dos dados. Para que o acesso às posições de memória seja possível, a cada uma delas está associada uma sequência de bits, chamada endereço da posição de memória. Como uma sequência de bits corresponde a um número inteiro escrito no sistema binário, cada endereço pode ser visto como um inteiro escrito no sistema decimal.

Uma variável simples é uma posição de memória cujo conteúdo pode ser modificado durante a execução de um programa. A referência a uma variável no programa é feita através do seu identificador. Os valores que podem ser nela armazenados dependem do seu tipo de dado.

O **sucesso de um programa** depende da correta declaração de variáveis e constantes. A utilização de operadores é fundamental e, se utilizados de forma errônea, o programa pode apresentar o famoso: erro de lógica.

**Erro de lógica significa que o:** programa não tem erro nos comandos, porém apresenta um resultado inesperado ou errado. Este tipo de erro refere-se ao que é denominado de **erro de lógica**. Em outras palavras, um erro de lógica é quando os operadores lógicos ou relacionais são usados de maneira

inadequada. Segundo Ascencio (1999, p.10), quando escrevemos um programa, este recebe os dados que devem ser armazenados no computador para que possam ser utilizados no processamento e na memória do computador.

Segundo Forbellone (2000), “um dado é classificado como variável quando tem a possibilidade de ser alterado em algum determinado momento de execução do programa”.

**“Toda variável deverá ser declarada antes de ser utilizada”.** Na sua declaração, informaremos um **tipo** para ela. Um tipo significa **“informar ao computador o que/qual tipo de dado essa variável poderá receber e armazenar na memória do computador.**

A referência a uma variável no programa é feita através do seu identificador que é o próprio nome da variável. O nome de uma variável é o identificador único daquele conteúdo que foi armazenado naquela posição de memória. O conteúdo de uma variável deve ser obrigatoriamente do mesmo tipo de dado que a mesma foi declarada.

O **identificador** de uma variável é o nome, propriamente dito, da variável. Esse identificador pode ser definido com uma sequência de letras e/ou números, os quais representarão o **nome** da variável. Geralmente, o nome de uma variável é uma palavra que signifique o conteúdo que será armazenado, por exemplo, se o conteúdo a ser armazenado é o nome de um aluno, então o nome da variável pode ser algo como **nome\_aluno**. Dessa forma, o nome da variável já informa ao programador o conteúdo desta variável. Os compiladores da linguagem C fazem distinção entre letras maiúsculas e minúsculas e, portanto, **Número** e **número** são dois nomes de variáveis diferentes. Isto é, a linguagem de programação estruturada C cria duas variáveis diferentes.

### 2.2.1 Declaração de Variável na Linguagem C

A declaração de uma variável em linguagem de programação C requer duas definições: de um **nome (identificador)** e do **tipo de dado** da variável. No exemplo, temos a declaração de uma variável com o nome de **nota**, do tipo número float (Figura 2). Neste caso, o compilador da linguagem de pro-

gramação estruturada C entende que deverá reservar um espaço na memória da máquina, uma variável do tipo **int** com o nome (identificador) **nota**.

Figura 2 - Declaração de variável em linguagem de programação estruturada C

```
float nota;
```

Fonte: Elaborado pelo autor, 2015.

O tipo de dado associado a uma variável é o conjunto dos valores que podem ser armazenados nela. Muitos são os tipos de variáveis e dependem da linguagem de programação adotada. A linguagem de programação C prevê o uso dos principais tipos de variáveis:

- × **Inteiros** - para valores inteiros (números positivos ou negativos - sem vírgulas, isto é, sem o ponto decimal que identifica um valor fracionário);
- × **Caracter** - para valores que receberão somente um caracter (este caracter pode ser uma letra, um número de 0 a 9, ou um caracter especial como, por exemplo, um símbolo matemático);
- × **Real** - para valores decimais, isto é, números com casas após a vírgula, positivos (+) ou negativos (-);
- × **Lógico** - para retornar os valores Falso e Verdadeiro, os quais identificam os números binários.

Uma das etapas mais importantes na estruturação de um programa é a definição dos tipos de dados das variáveis (identificadores) que serão utilizadas em todo o programa.

Quadro 1 - Tipos de dados que podem ser declarados em C.

Tipo de Dado	Quantidade de Bytes	Conjunto de Valores
Char	1	caracteres codificados no código ASCII
Int	2	números inteiros de -32768 a 32767
long ou longint	4	números inteiros de -65536 a 65535

Tipo de Dado	Quantidade de Bytes	Conjunto de Valores
Float	4	números reais de $-3,4 \times 10^{38}$ a $-3,4 \times 10^{-38}$ e 3,4
Double	8	números reais de $-1,7 \times 10^{308}$ a $-1,7 \times 10^{-308}$ e 1
Void	0	conjunto vazio

Fonte: Elaborado pelo autor, 2015.

O Quadro 1 apresenta o tipo de dado, a quantidade de bytes que cada tipo de dado aloca na memória para armazenamento de dados, e o conjunto de valores que cada tipo de dado suporta.

Observe com atenção outros exemplos de declarações de variáveis:

- × **double** nome da variável;
- × **longint** nome da variável;
- × **float** nome da variável.

Os principais cuidados que deveremos ter na declaração de variáveis são:

- × A linguagem C interpreta o nome das variáveis em maiúscula de minúscula como variáveis diferentes. Por exemplo, a variável com nome “idade” é diferente da variável com nome “Idade”. Para a linguagem C são duas variáveis diferentes. Não se trata da mesma variável.
- × A linguagem C não aceita que o nome de uma variável seja definido iniciando com um número. Por exemplo, o nome de variável “2endereco”. O número 2 como primeiro caractere do nome da variável não é aceito pela linguagem C.
- × A linguagem C não aceita espaços em branco em um nome de variável. Por exemplo, o nome de variável “nome do aluno”. Se for necessário um nome de variável com mais de uma palavra, então sugere-se usar o *underline* como segue: “nome\_do\_aluno”.

É importante salientar que quando criamos uma variável precisamos prestar atenção para não utilizarmos as palavras reservadas da linguagem de programação estruturada C. Por exemplo, o quadro 2 abaixo mostra uma relação das palavras reservadas da linguagem de programação estruturada C. As palavras reservadas são palavras que são utilizadas pelo compilador da linguagem de programação para realizar tarefas específicas durante a execução do programa. Por isso, não podemos criar variáveis com os nomes das palavras reservadas, porque estas palavras possuem um significado específico na linguagem de programação.

Quadro 2 - Listas de Palavras reservadas.

auto	Double	Int	struct
break	Else	Long	switch
case	enum	Register	typedef
char	extern	Eturnr	union
const	float	short	unsigned
continue	for	Signed	void
default	goto	sizeof	volatile
do	if	static	while

Fonte: Elaborado pelo autor, 2015.

## 2.3 O que é uma constante

A utilização de constantes na linguagem C é muito útil. Imagine uma situação em que você precisaria informar a todo o momento um valor específico, que é sempre igual e esse valor fosse um grande numeral. Podemos citar os valores de expressões geométricas, as siglas dos estados brasileiros, etc. Para isso servem as constantes, que são declaradas no início do programa. Uma vez definida uma constante, o seu valor não poderá ser modificado. Caso o valor de um imposto tenha sido alterado, o programador altera o valor da constante no início do programa, e não precisará realizar essa alteração no programa inteiro.

### 2.3.1 Declarações de Constantes

Em linguagem C, utilizamos o prefixo **const** associado a um tipo, um nome de identificador e um valor para definir uma constante. Assim:

```
const <tipo> <nome> = <valor>;
```

Por exemplo:

```
const int eterna = 256;
```

## 2.4 Expressões Aritméticas

Os compiladores da linguagem C são capazes de avaliar expressões aritméticas que envolvam as operações binárias de multiplicação, divisão, soma e subtração, e a operação unária de troca de sinal. Essas operações são realizadas através de operações aritméticas.

Chamamos de operadores aritméticos o conjunto de símbolos que representa as operações básicas da matemática. Estas operações são convertidas em binárias pelo compilador do programa. Na linguagem estruturada C, utilizamos as quatro operações básicas da matemática. Confira no quadro 3 abaixo, os símbolos utilizados como operadores aritméticos na linguagem C.

Quadro 3 - Expressões Básicas de Matemática.

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
-	Unário
--	Decremento
++	Incremento
% (MOD)	Módulo (resto da divisão inteira).

Fonte: Elaborado pelo autor, 2015.



Uma expressão que envolva diversas operações é avaliada de acordo com as regras de prioridade da matemática. Em primeiro lugar são realizadas as operações que estão dentro dos parênteses  $()$ , depois as que estão dentro dos colchetes  $[]$ , e por último, as que estão dentro das chaves  $\{ \}$ .

### Saiba mais

A operação Módulo retorna o resto da divisão entre dois números inteiros. Por exemplo, na divisão do número 5 pelo número 2, o resto da divisão desses inteiros é 1. Logo, o Módulo da expressão  $(5/2)$  é representado como  $MOD(5,2)$  e o resultado dessa expressão é 1.

$$MOD(5,2) = 1$$

A seguir apresentamos exemplos de expressões aritméticas:

`celsius = (fahrenheit - 32) * 5.0 / 9.0;`

`força = massa * aceleração;`

`i = i + 1;`

É muito importante observar que quando as prioridades da matemática não são respeitadas, o resultado das operações dentro de uma expressão aritmética pode ser totalmente diferente do esperado.

## 2.5 Operadores Relacionais

Um operador relacional é utilizado em expressões lógicas muito utilizadas na linguagem de programação C. Os operadores relacionais são utilizados quando precisamos comparar dados para saber se esses dados são iguais ou não, e se são maiores, menores que outro determinado dado, que pode ser um número ou um caractere. O resultado da comparação feita por um operador relacional será o ou 1. Em outras palavras, a comparação pode resultar um valor verdade ou um valor falso. O quadro 4 mostra os operadores relacionais que a linguagem de programação C disponibiliza aos programadores:

Quadro 4 - Operadores relacionais

Operador Relacional	O que significa
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
!= ou <>	Diferente de
= ou ==	Igual

Fonte: Elaborado pelo autor, 2015.

Os operadores relacionais são utilizados na maioria dos programas.

Por exemplo, vamos considerar que  $X = 2$ ,  $Y = 10$ , e as expressões abaixo. É importante lembrar que, em linguagem de programação estruturada C, o símbolo = significa **atribuição de valores a uma variável** e que == significa **comparação de igualdade entre valores**.

Agora observe as comparações abaixo:

$X > Y$ , resposta: falso

Leia-se: X é maior que Y ?

$X >= Y$ , resposta: falso

Leia-se: X é maior ou igual a Y ?

$X < Y$ , resposta: verdadeiro

Leia-se: X é menor que Y ?

$X <= Y$ , resposta: verdadeiro

Leia-se: X é menor ou igual a Y ?

$X == Y$ , resposta: falso

Leia-se: X é igual a Y ?

$X != Y$ , resposta: verdadeiro

Leia-se: X é diferente de Y ?



### Saiba mais

Os operadores relacionais IGUAL e DIFERENTE podem ter sintaxes diferentes de acordo com a linguagem de programação, como por exemplo:

- o IGUAL pode ser `"=="` ou `"=`
- o DIFERENTE pode ser `"!="` ou `"<>"`



## 2.6 Operadores Lógicos

Os operadores lógicos definem o modo como os operadores relacionais e os próprios valores lógicos, podem ser conectados. Para simplificar a apresentação desses operadores serão usadas variáveis para substituir as relações. Essas variáveis podem assumir dois valores: **falso ou verdadeiro**. Em C qualquer valor diferente de zero é considerado verdadeiro. Segundo Mizrahi (1994, p.53), “operadores lógicos fazem comparações”. A diferença entre comparações lógicas e relacionais está na forma como os operadores avaliam seus operandos. Essa avaliação resulta em verdadeiro ou falso, conforme pode ser visto no quadro 5.

Quadro 5 - Operadores lógicos

Algoritmo	C	Função
E	<code>&amp;&amp;</code>	Será verdadeiro o resultado somente se todos os valores forem verdadeiros
OU	<code>  </code>	Será verdadeiro o resultado se um dos valores for verdadeiro
Não	<code>!</code>	Será verdadeiro somente se a expressão for falsa

Fonte: Elaborado pelo autor, 2015.

Para trabalharmos com os operadores matemáticos, relacionais e lógicos, precisamos conhecer as regras de precedência dos operadores, ou regras de prioridade. O Quadro 6 mostra, em ordem decrescente de prioridade, as regras de precedência dos operadores em C.

Quadro 6 - Regras com as prioridades dos operadores.

Operador	Descrição	Associatividade
++ --	Incremento e decremento pós-fixos	Esquerda para direita
( )	Parênteses (chamada de função)	Esquerda para direita
++ --	Incremento e decremento préfixos	Direita para esquerda
+ -	Adição e subtração unária	Direita para esquerda
! ~	Não lógico	Direita para esquerda
* / %	Multiplicação, divisão e módulo (resto)	Esquerda para direita
+ -	Adição e subtração	Esquerda para direita
< <=	“Menor que” e “menor ou igual que”	Esquerda para direita
> >=	“Maior que” e “maior ou igual que”	Esquerda para direita
== !=	“Igual a” e “Diferente de”	Esquerda para direita
&&	“E” lógico	Esquerda para direita
	“OU” lógico	Esquerda para direita
=	Atribuição	Direita para esquerda
+=	Atribuição por adição	Direita para esquerda
-=	Atribuição por subtração	Direita para esquerda
*=	Atribuição por multiplicação	Direita para esquerda
/=	Atribuição por divisão	Direita para esquerda
%/	Atribuição por módulo	Direita para esquerda

Fonte: Elaborado pelo autor, 2015.

### Saiba mais

Incremento e Decremento Pré-fixados ( $++i$ ) ou ( $--i$ ) significa que primeiro é realizado o incremento ou decremento da variável  $i$  e depois é retornado seu valor.

Incremento e Decremento Pós-fixados ( $i++$ ) ou ( $i--$ ) significa que primeiro é retornado o valor de  $i$  e depois é realizado seu incremento ou decremento.

### Importante

Quando aprendemos sobre uma linguagem de programação específica, neste caso a linguagem de programação estruturada em C, conhecemos a sintaxe para a escrita do código de programação desta linguagem. Uma linguagem de programação possui - assim como um idioma Português, Inglês, Francês, Alemão, Italiano - uma gramática, uma sintaxe desta gramática, regras e exceções. Com uma linguagem de programação não é diferente. A sintaxe da linguagem de programação estruturada em C está sendo apresentada durante todo este curso, no entanto, algumas funções, operadores, podem ter o mesmo significado em outras linguagens de programação, mas apresentarem sintaxes diferentes. Por exemplo, o operador de atribuição de dados a uma variável ou constante, em linguagem de programação C, que é utilizado é o caracter igual " $=$ ". No entanto, na linguagem de programação Java, o operador de atribuição é representado pelos caracteres dois pontos e igual juntos " $:=$ ". Logo, os dois operadores possuem a mesma função que é a atribuição de dados a variáveis ou constantes, mas cada linguagem de programação tem uma sintaxe diferente.

## 2.7 Funções de Entradas e Saídas de Dados

As funções de entrada e saída de dados permitem ao usuário fazer a inserção de dados no sistema (entrada) e a visualização dos resultados obtidos após o processamento desses dados (saída). A inserção de dados e a visualização dos resultados podem ser feitas a partir dos dispositivos de entrada e saída

de dados, tais como: teclado, mouse, telas *touchscreen*, microfone, câmeras de vídeo, pendrives, monitor, entre outras.

A linguagem C disponibiliza uma vasta biblioteca relacionada ao controle de entrada e saídas de dados.

Para utilizar as funções de entrada e saída de dados na linguagem C, devemos utilizar a biblioteca `<stdio.h>`. Todas as vezes que forem utilizados comandos de entrada e saída de dados essa biblioteca deve ser declarada. Como em todos os programas, temos dados de entrada e o uso dessa biblioteca se torna essencial.



### Saiba mais

Biblioteca é uma coleção de subprogramas utilizados no desenvolvimento de software. A biblioteca padrão ANSI C consiste de 24 cabeçalhos, cada um contendo uma ou mais declarações de funções, tipos de dados e macros.

Para utilizarmos as funções definidas em uma biblioteca padrão ANSI C é necessário inserirmos essa biblioteca no início do código fonte do nosso programa. Para isso, utilizamos a diretiva `#include` e o nome da biblioteca. Por exemplo, se pretendemos utilizar as funções de entrada e saída de dados em tela, precisamos fazer o `#include` da biblioteca `<stdio.h>`. Esta biblioteca contém a definição de todas as funções de entrada e saída de dados da linguagem de programação C.



Após a declaração da biblioteca de entrada e saída de dados, podemos utilizar as funções destas bibliotecas. Em C, existem muitas funções pré-definidas que tratam de troca de informações. As funções de entrada e saída mais comuns são **`scanf()`** e **`printf()`** que são apresentadas na sequência.

### 2.7.1 Entrada de Dados e Saída de Dados

O fornecimento dos dados (entrada de dados) que o programa utilizará é feita a partir das funções de entrada de dados que são definidas na Biblioteca `<stdio.h>`. A função de entrada de dados permite ao programa ler dados formatados da entrada padrão que é o teclado.

A Figura 3 mostra a função **`scanf()`** é utilizada para fazer a leitura de dados via teclado.

Sintaxe:

**scanf (“expressão de controle”, lista de argumentos);**

Figura 3 - Função para entrada de dados scanf()

```
scanf ("%d", &idade);
```

Fonte: Elaborado pelo autor, 2015.

Onde, a “**expressão de controle**” quer dizer o tipo de dado que está sendo esperado e, obrigatoriamente, essa “expressão de controle” precisa estar entre aspas. A **lista de argumentos** é o nome da variável que receberá o dado digitado pelo usuário via teclado. Neste exemplo, a “expressão de controle” chamada de **%d** significa que a função está preparada para receber um número do tipo inteiro e **&idade** é o nome da variável que receberá a idade a ser digitada. O valor lido será armazenado no endereço de memória **&** da variável idade (&idade).

A seguir, a tabela 1 mostra os tipos de dados básicos e suas representações para serem utilizadas na função scanf():

Tabela 1 - Tipos de dados e as representações básicas de uma expressão de controle

Linguagem C	Formato	Tipo de dados
char	%c	caracter
int	%d	inteiro
float	%f	real
char [ ]	%s	cadeira de caracteres (string)

Fonte: Elaborado pelo autor, 2015.

O retorno dos dados processados pelo programa é feito a partir das funções de saída de dados que, também, são definidas na Biblioteca <stdio.h>. A função de saída de dados permite ao programa mostrar os resultados do processamento na tela do computador.

A Figura 4 mostra a função **printf()** que é utilizada para fazer a saída de dados via tela do computador.

Sintaxe:

**printf** (“expressão de controle”, lista de argumentos);

Figura 4 - Função para saída de dados printf()

```
printf("\\nA soma dos números é: %d\\n", &soma);
```

Fonte: Elaborado pelo autor, 2015.

Onde, a “**expressão de controle**” mostra uma frase ou uma palavra que será mostrada na tela do computador com o tipo de dado que será apresentado, lembrando sempre que essa “expressão de controle” deve estar entre aspas, conforme mostrado na Figura 4. E a **lista de argumentos** é o nome da variável que terá o seu conteúdo mostrado na tela. Neste exemplo, a “expressão de controle” mostra a frase. **A soma dos números é:** e o **%d** significa que a função está preparada para mostrar na tela um número do tipo inteiro. **Soma** é o nome da variável que terá o seu conteúdo mostrado na tela.

## Importante

Para que as funções **scanf()** e **printf()** sejam utilizadas em um programa na linguagem de programação C, é obrigatório a inclusão da Biblioteca **<stdio.h>**.

Figura 5 - Exemplo de programa com funções de entrada e saída de dados em tela

```
#include <stdio.h>
int main ( )
{
    int a, b, c, soma;
    printf("\\nDigite o primeiro número: ");
    scanf("%d", &a);

    printf("\\nDigite o segundo número: ");
    scanf("%d", &b);

    printf("\\nDigite o terceiro número: ");
    scanf("%d", &c);

    soma= a + b + c;
    printf("\\nA soma dos números é: %d\\n", soma);
}
```

Fonte: Elaborado pelo autor, 2015.



No exemplo apresentado na Figura 5, o programa inicia com a diretiva **#include** para inserir a Biblioteca **<stdio.h>**, na linha 1 do programa. Esta biblioteca define todas as funções de entrada e saída de dados em linguagem de programação C. Em seguida, na linha 4 são definidas as variáveis **a**, **b**, **c**, soma do tipo inteiro. Nas linhas 5, 8 e 11, a função **printf()** mostra na tela a mensagem de que o usuário precisa digitar um valor, e nas linhas 6, 9 e 12, a função **scanf()** espera a entrada de dados via teclado para armazenar o valor digitado nas variáveis **a**, **b**, **c**. Na linha 14 é feito a soma dos valores e essa soma é armazenada na variável **soma**. E, por fim, na linha 15 a função **printf()** mostra na tela do computador uma mensagem com a soma e o valor armazenado na variável **soma**, como pode ser observado na Figura 6.

Figura 6 - Exemplo de saída de dados em tela.

```
Digite o primeiro número:4
Digite o segundo número: 5
Digite o terceiro número: 6
A soma dos números é: 15
-----
Process exited after 5.17 seconds with return value 0
Pressione qualquer tecla para continuar...
```

Fonte: Elaborado pelo autor, 2015.

## Importante

Observar os tipos de argumentos da função **printf()** e o que cada argumento faz quando o programa está em execução.

A tabela 2 apresenta os argumentos que podem ser utilizados na função **printf()**. Estes argumentos significam que o programa executará algumas ações já pré-definidas na linguagem de programação C. Dentre as diversas ações, um exemplo, são as ações referentes aos espaçamentos entre linhas e/ou entre caracteres, como pode ser observado a seguir.

Tabela 2 - Tipos de argumentos da função printf ()

Argumento	O que será executado pelo programa
\n	Insere uma nova linha abaixo do comando em execução semelhante à tecla ENTER do teclado.
\t	Insere um espaço semelhante à tecla “TAB” do teclado.
\b	Exclui um espaço semelhante à tecla “RETROCESSO” do teclado.
\"	Insere aspas durante a execução do programa.
\\	Insere duas barras durante a execução do programa.
\f	salta formulário
\0	Nulo

Fonte: Elaborado pelo autor, 2015.

## 2.8 Comando de Atribuição

Com a linguagem de programação C é possível realizarmos atribuições através de comandos que são criados no início do programa ou atribuições dadas quando o programa está na memória da máquina. A atribuição é indicada pelo sinal “=” sempre da direita para esquerda e é uma das formas essenciais para alterar o conteúdo de uma variável.

Figura 7 - Exemplos de Comandos de Atribuição

Fonte: Elaborado pelo autor, 2015.

```
#include <stdio.h>
int main ( )
{
    int a, b, c, soma;

    a = 10;

    b = 20;

    c = 5;

    soma = a + b + c;
}
```

A variável que estiver do lado esquerdo do igual sempre recebe o valor do resultado de uma variável ou expressão da direita. Um exemplo prático ao se digitar **x=b**, está-se dizendo que **x** receberá o valor contido em **b**. Se for digitado **x=b+y**, significa que **x** receberá o valor de **b+y**. É importante lembrar sempre que o conteúdo da variável **x** é alterado a cada nova atribuição.

A Figura 7 mostra que na linha 4 é realizada a atribuição de valores às variáveis do tipo inteiro **a**, **b**, **c** e **soma**. Neste exemplo,

nas linhas 5 a 10, a variável **a** recebe o número inteiro 10; a variável **b** recebe o número inteiro 20; a variável **c** recebe o número inteiro 5; e na linha 12, a variável **soma** recebe o resultado da soma de **a**, **b** e **c**.

## 2.9 Conversão de Tipos de Dados

A linguagem de programação C permite que o programador altere o tipo de dado de uma variável - que foi declarada no início do programa - por outro tipo de dado, durante a execução desse mesmo programa. Esse tipo de operação é comum no desenvolvimento de um programa de computador, pois permite que tipos de variáveis declaradas no início da programação sofram alterações por meio de comandos estabelecidos pelo programador. Esse tipo de procedimento é chamado de *Typecasting* que consiste em uma técnica utilizada em C para realização de conversões simples, às vezes temos a necessidade de não converter um tipo de dado, mas fazer com que pareça com outro tipo de dado.

Um exemplo prático de como podemos fazer conversão de dados é quando precisamos converter uma variável do tipo inteiro para outra variável do tipo char ou vice-versa. Sabemos que pela Tabela ASCII (American Standard Code for Information Intechange, disponível em: <<http://www.asciitable.com/>>), cada caracter do teclado tem um número decimal correspondente. Desta forma, podemos fazer as conversões de dados, tendo como base esse padrão internacional de troca de informações.

### Saiba mais

A Tabela ASCII é um código padrão americano para a troca de informações. Em outras palavras, é um código binário que codifica um conjunto de 255 caracteres. A codificação ASCII é usada para representar letras, números, caracteres gráficos, sinais de acentuação, sinais da matemática. Em linguagem de programação é muito utilizada para fazer a conversão de códigos binários para letras em maiúsculas e minúsculas.

A Tabela ASCII mostra todos os caracteres que estão no teclado de um computador e o seu respectivo número decimal. Dessa forma, podemos converter caracteres em inteiros e inteiros em caracteres.

A Figura 8 mostra a conversão de dados de uma variável inteira para um caracter, e vice-versa.

Figura 8 - Exemplo de conversão de dados de inteiro para char e vice-versa

```
#include<stdio.h>
int main()
{
    int x=65;
    char letra;

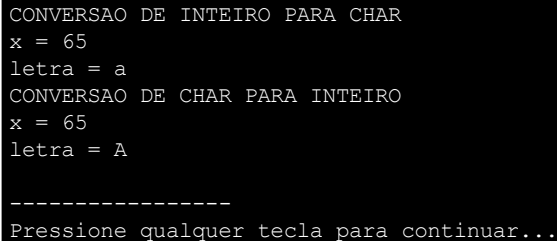
    letra = char(x);

    printf ("CONVERSAO DE INTEIRO PARA CHAR\n");
    printf ("x = %d\n", x);
    printf ("letra = %c\n", letra);

    x = int (letra);

    printf ("CONVERSAO DE CHAR PARA INTEIRO\n");
    printf ("x = %d\n", x);
    printf ("letra = %c\n", letra);

}
```



```
CONVERSAO DE INTEIRO PARA CHAR
x = 65
letra = a
CONVERSAO DE CHAR PARA INTEIRO
x = 65
letra = A
-----
Pressione qualquer tecla para continuar...
```

Fonte: Elaborado pelo autor, 2015.

Nesse exemplo, na linha 4 a variável inteira x recebe o número 65, e na linha 7 a variável char letra recebe a atribuição char (x). Nesse momento da execução do programa é realizada a conversão de dados da variável x, que inicialmente, tem o inteiro 65, para um caracter. A conversão é feita com base na Tabela ASCII. O decimal 65 na Tabela ASCII corresponde à letra A em maiúscula. Da mesma forma, na linha 13, é feita a conversão da letra A para o inteiro 65.

## Saiba mais

Quando convertemos ponto flutuante para inteiros e caracteres, a parte fracionária é desprezada. Ao realizar a conversão de dados poderá ocorrer um estouro de faixa alterando o valor. Considera-se estouro de faixa quando a conversão de tipo de dados é muito grande para ser representado dentro do intervalo de valores permitido para esse tipo de variável (MANZANO, 2002).

Quadro 7 - Conversão de tipos de dados

Tipo de Destino	Tipo de Expressão	Possível informação Perdida
signed char	char	Se valor > 127, o destino é negativo
Char	short int	Os primeiros 8 bits mais significativos
Char	int	Os 8 bits mais significativos
Char	longint	Os 24 bits mais significativos
Int	longint	Os 16 bits mais significativos
Int	float	A parte fracionária e/ou mais
float	double	Precisão, o resultado arredondado
double	longdouble	Precisão, o resultado é arredondado

Fonte: Elaborado pelo autor, 2015.

## Resumindo

Nesta unidade foi descrito os conceitos básicos da estrutura de programação estruturada da linguagem C. Vimos as expressões matemáticas utilizadas, os operadores lógicos e relacionais, os comandos que devem ser executados para dar início ao programa, os tipos de variáveis utilizadas para leitura e saída de dados, as bibliotecas que devem ser usadas, os comandos de atribuições de dados, conversões de dados para manipulações de variáveis. A assimilação de todo esse conteúdo virá com a prática e com o emprego dos comandos descritos.



# 3

## Conhecendo a Ferramenta Dev-C

PARA DAR INÍCIO ao nosso aprendizado na linguagem de programação estruturada C, vamos conhecer o software Dev-C, criada pela Empresa Bloodshed Software (<http://www.bloodshed.net/index.html>). Este software é um ambiente integrado de programação que disponibiliza uma interface gráfica Windows com os principais recursos para criação, edição e compilação de arquivos em código fonte C.

O uso de ambientes integrados de programação, também chamados de IDE – *Integrated Development Environment* – são muito bem vistos pelos programadores porque temos todos os recursos necessários para a programação, em apenas um lugar e, temos a possibilidade de salvar todos os programas em um único arquivo que chamamos de projeto. O Dev-C é um ambiente de programação que integra:

- × editor de texto simples.
- × compilador da linguagem C.
- × depurador de programas (*debugger*) da linguagem C.

Então siga em frente e bons estudos!

## Objetivo de Aprendizagem:

- × Utilizar a ferramenta Dev-C como plataforma de apoio às atividades de aprendizado da linguagem de programação C.

### 3.1 A Instalação do Ambiente de Programação Dev-C

Para iniciarmos a programar com o Dev-C precisamos instalá-lo em nosso computador. Primeiramente, precisamos fazer o download do programa de instalação do Dev-C e, depois fazer a instalação passo-a-passo do software.

Para fazer o download do programa precisamos acessar o site: <http://www.bloodshed.net/download.html> como mostrado na Figura 1:

Figura 1 - Site da empresa BloodshedSoftware.



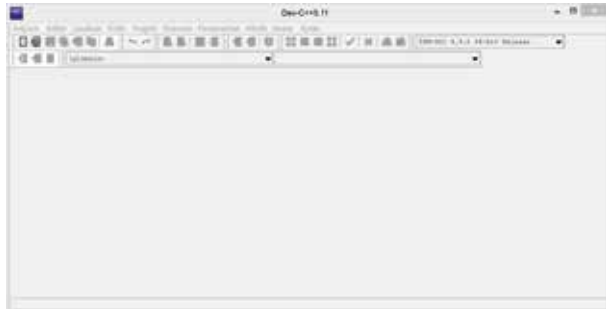
Fonte: <http://www.bloodshed.net/download.html>

Após acessar o site é necessário escolher o link do Dev-C para fazer o download do arquivo de instalação do ambiente de programação e instalá-lo no seu computador. Para a elaboração de todos os exemplos apresentados nesta disciplina utilizamos a versão 5.11. A instalação do Dev-C é bem simples e sugerimos a instalação completa do software.

A interface gráfica do Dev-C, versão 5.11, é totalmente em português e apresentada no padrão Windows, como mostrado na Figura 2.



Figura 2 - Interface gráfica do Dev-C.

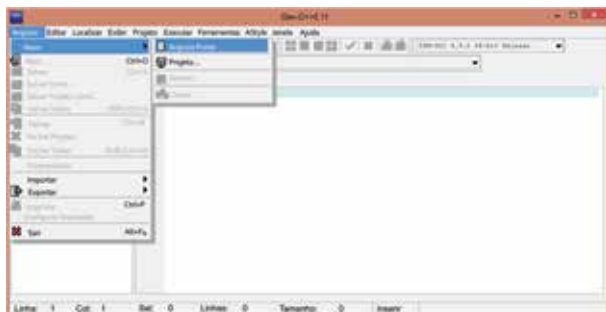


Fonte: Elaborado pelo autor, 2015.

### 3.2 A Interface do Dev-C

Para iniciarmos a programar com o Dev-C vamos ao Menu Arquivo. Neste menu temos o submenu Novo. Este submenu possui duas opções disponíveis: Arquivo Fonte e Projeto, como mostrado na Figura 3:

Figura 3 - Abrindo um arquivo fonte em C.

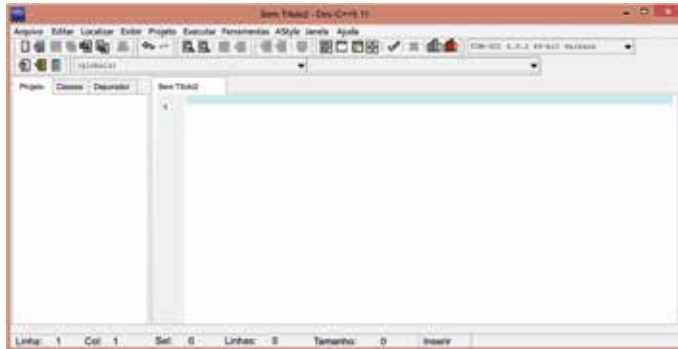


Fonte: Elaborado pelo autor, 2015.

A opção Arquivo Fonte abre o editor de texto do Dev-C para iniciarmos a escrita de um novo arquivo de código fonte em linguagem C. A opção Projeto permite a criação de um conjunto de programas que ficarão salvos dentro de um único arquivo. Neste momento, iniciamos com a criação do nosso

primeiro programa em C, com um novo Arquivo Fonte, que abrirá o editor de texto, como mostrado na Figura 4:

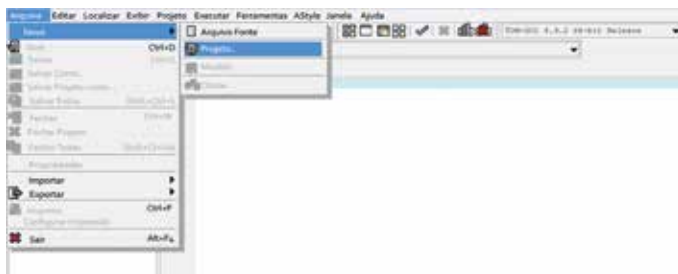
Figura 4 - Criando o primeiro programa em C.



Fonte: Elaborado pelo autor, 2015.

Para iniciarmos um projeto com o Dev-C vamos ao Menu Arquivo, sub-menu Novo e escolhemos a opção Projeto, como mostrado na Figura 5:

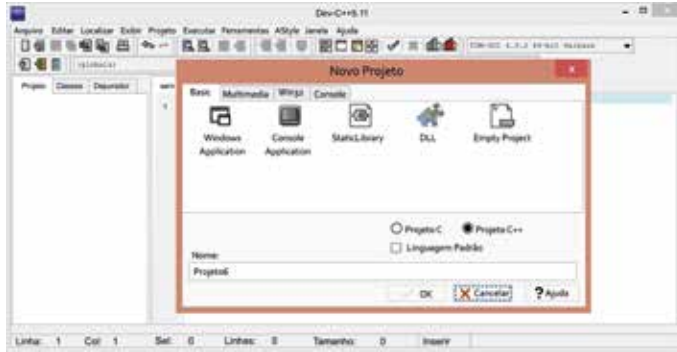
Figura 5 - Abrindo um projeto em C.



Fonte: Elaborado pelo autor, 2015.

Um projeto em linguagem de programação C pode ser uma Janela Windows (Window Application), um Aplicativo de Console (Console Application), uma Biblioteca Estática (Static Library), uma Biblioteca Dinâmica (DLL), e um Projeto em Branco (Empty Project), como mostrado na Figura 6:

Figura 6 - Opções para iniciar um projeto em C.



Fonte: Elaborado pelo autor, 2015.

### 3.3 Os Menus

O Dev-C disponibiliza menus que seguem o padrão Windows de organização: Arquivo, Editar, Localizar, Exibir, Projeto, Executar, Ferramentas, AStyle, Janela, Ajuda, como mostrados nas Figuras a seguir.

O menu Arquivo (Figura 7) disponibiliza as funções básicas de criação, edição e manipulação de arquivos: novo, abrir, fechar, salvar, propriedades, importar e exportar arquivos, imprimir.

O menu Editar (Figura 8) disponibiliza as funções de edição copiar, recortar, colar, desfazer, selecionar, comentar e excluir comentário, indentar e excluir indentação, entre outras mais específicas de programação. É importante salientar que a maioria das

Figura 7 - Menu Arquivo.



Fonte: Elaborado pelo autor, 2015.

Figura 8 - Menu Editar.



Fonte: Elaborado pelo autor, 2015.

opções dos menus possuem teclas de atalho ao lado da opção no menu (Copiar).

O menu Localizar (Figura 9) apresenta opções de localizar arquivos e palavras no código fonte e, também de ir para uma determinada função ou linha.

O menu Exibir (Figura 10) apresenta as funcionalidades de exibição das barras de status e ferramentas, do gerenciador de projeto flutuante e da janela de relatórios flutuante. Neste menu podemos personalizar a área de trabalho deixando visível apenas as barras e relatórios que necessitamos naquele momento. Caso contrário, podemos deixar a tela mais limpa visualmente para trabalhar.

Figura 9 - Menu Localizar.

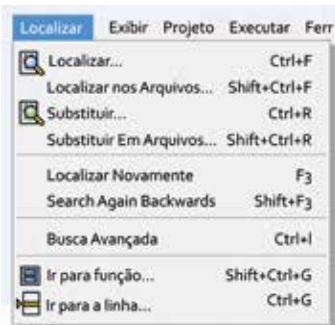
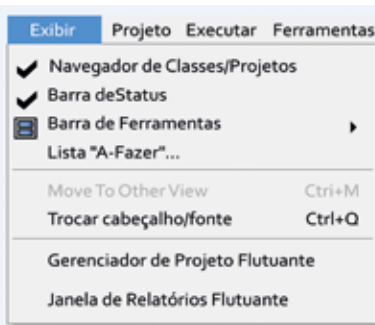


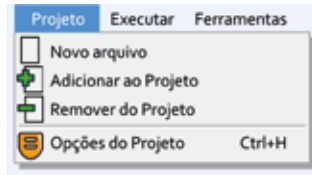
Figura 10 - Menu Exibir.



Fonte: Elaborado pelo autor, 2015.

O menu Projeto (Figura 11) disponibiliza as opções para manipulação dos projetos de software em linguagem de programação C. Nesta opção é possível iniciar um novo arquivo de projeto, adicionar e remover arquivos a um projeto e por último, temos a opção de opções de configurações de um projeto.

Figura 11 - Menu Projeto.



Fonte: Elaborado pelo autor, 2015.

O menu Executar (Figura 12) é um dos menus mais importantes do Dev-C porque apresenta as opções de compilação (F9), execução (F10), compilação e execução (F11) e recompilar tudo (F12). Estas opções e as suas respectivas teclas de atalho são as mais utilizadas pelos programadores. Além destas opções, ainda temos as opções de checar a sintaxe do código fonte, criar breakpoint e depurar o código fonte. Algumas destas opções são para programadores em fase avançada de programação. Nesta disciplina iremos nos deter nas quatro primeiras opções deste menu.

Figura 12 - Menu Executar.



Fonte: Elaborado pelo autor, 2015.

## 3.4 O Primeiro Programa em C

O nosso primeiro programa em C será a tradicional mensagem “Olá Mundo” escrita pela maioria dos programadores quando inicia os estudos de uma nova linguagem de programação. A Figura 13 apresenta o editor de texto com cinco (05) linhas de código de programação em C para mostrar a mensagem na tela:

Figura 13 - Mensagem “Olá Mundo” em um programa em C.



Fonte: Elaborado pelo autor, 2015.

### 3.4.1 Compilando o Primeiro Programa em C

Após a edição do nosso primeiro programa em C precisamos compilá-lo e executá-lo para sabermos se está funcionando. O processo de compilação é quando o compilador da linguagem de programação C faz uma varredura em todo o arquivo código fonte e o codifica em linguagem executável. Neste momento, precisamos salvar o arquivo código fonte com um nome com extensão .CPP e o compilador após a compilação gerará um arquivo com o mesmo nome mas com extensão .EXE. Em seguida, o próprio Dev-C executará o arquivo .EXE.

#### Saiba mais

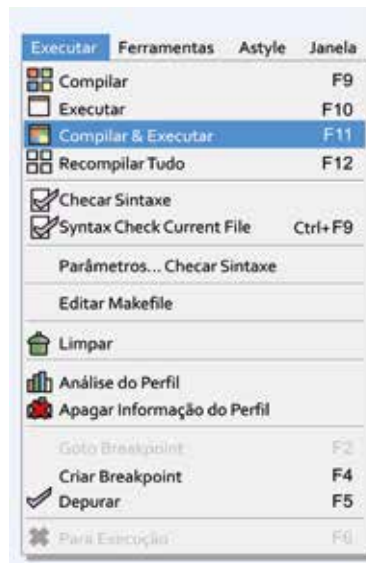
A compilação é um processo no qual o Compilador da linguagem de programação faz uma leitura do arquivo que contém o código fonte (\*.C) e o “traduz” para linguagem de máquina (\*.EXE). A compilação é a passagem de um arquivo fonte para um arquivo que pode ser executado pelo usuário. Isto é, o usuário não consegue executar (rodar) um arquivo fonte. É necessário que o arquivo fonte passe pelo processo de compilação para que seja gerado o arquivo Executável, e esse arquivo Executável pode ser executado (rodado) pelo usuário.

A Execução de um arquivo Executável é a chamada para que aquele arquivo faça o que foi programado no arquivo fonte. Em programação

chamamos “rodar um programa”, que nada mais é do que clicar duas vezes sobre o arquivo Executável e esperar que ele faça os comandos que foram programados no arquivo fonte.

Para realizarmos este processo de compilação é necessário acessarmos o menu Executar e a opção Compilar & Executar, como mostrada na Figura 14:

Figura 14 - Menu Executar opção Compilar e Executar.



Fonte: Elaborado pelo autor, 2015.

Em seguida aparecerá na tela a saída de tela mostrada na Figura 15:

Figura 15 - Saída de tela do primeiro programa em C.

```
OLA MUNDO!  
-----  
Pressione qualquer tecla para continuar...
```

Fonte: Elaborado pelo autor, 2015.

### 3.5 Comentários em C

Um recurso muito utilizado para a organização dos códigos-fonte em programação é a inclusão de comentários. Um programa comentado, com a identificação do autor, ano e versão, é uma forma de organização do programador que auxiliará, posteriormente, quando outros programadores forem efetuar alguma manutenção no programa. Por isso, o hábito de comentar os códigos-fonte é uma forma de documentar o trabalho do programador e é muito bem visto por programadores e analistas que estão no mercado de trabalho.

Para comentar um programa em linguagem de programação C utilizamos as barras duplas “//”, como mostrado na Figura 16:

Figura 16 - Saída de tela do primeiro programa em C.

Ola mundo.cpp

```
// Autor: Cristiane KoehLer
// Ano: 2015
// Versão: 1.0
#include <stdio.h>
int main()
{
    printf ("OLA MUNDO!");

    // este é o nosso primeiro programa em Linguagem C
}
```

Fonte: Elaborado pelo autor, 2015.

Na Figura 16 as linhas 1, 2, 3 e 9 apresentam comentários sobre a identificação deste código fonte, como autor, ano e versão, nas três primeiras linhas e na linha 9, temos uma informação que este é o nosso primeiro programa em linguagem C. Estas linhas que estão com as barras duplas “//” não são interpretadas pelo compilador como sendo parte do código fonte a ser executado. São interpretadas e compiladas como simplesmente comentários que não são linhas de código executáveis. Para distinguir estas linhas das outras que serão compiladas e gerado um código executável, o editor as identifica com uma cor diferente das linhas que serão executadas.



## Resumindo

Neste capítulo, vimos como elaborar os nossos primeiros programas em linguagem de programação estruturada em C, com o ambiente de programação Dev-C. Para isso, aprendemos a instalar o programa de programação Dev-C, conhecemos a sua interface gráfica, seus menus e principais funcionalidades para iniciar a programação. Vimos como editar e compilar o nosso primeiro programa em C e como inserir comentários no código fonte como forma de documentação de programas.

---

---

**Aplicativo do Console:** um executável que aceita a entrada e envia a saída para o console, também conhecido como Prompt de Comando. Por meio dele, é possível criar consoles para fazer o trabalho básico ou executar tarefas muito sofisticadas. Também é possível usá-lo como uma demonstração (teste) de conceito de funcionalidade para, posteriormente, inserir um aplicativo no Windows.

---

---



# 4

## Estruturas de Seleção na Linguagem de Programação Estruturada C

VOCÊ ESTÁ PRONTO para estudar uma das estruturas mais importantes em linguagem de programação estrutura C? Isso mesmo! É a **estrutura de seleção**. Até o momento, vimos a programação estruturada de forma sequencial, mas sabemos que os problemas a resolver exigem que façamos escolhas, testes e é aí que a estrutura de seleção entra em cena. Ela tem o objetivo de testar o conteúdo das variáveis para seguir com a execução do programa. E como será a nossa organização para entender o assunto? Para compreender o assunto, vamos estudar:

- × o que é uma estrutura de seleção;
- × para que serve;
- × quais são os tipos de estruturas de decisão: simples *if* e composta *if.. else*;
- × a estrutura do comando de decisão múltipla: *switch*.

Vamos começar? Então, acompanhe-nos!

## Objetivo da Aprendizagem:

- × Analisar, compreender e aplicar estruturas de seleção e suas características em problemas computacionais.

### 4.1 Estruturas de Seleção

Quando falamos em programação, a estrutura de seleção é um comando utilizado quando precisamos decidir sobre algo ou alguma coisa. O que isso quer dizer? Em outras palavras, é quando precisamos tomar uma **decisão** diante dos dados que temos e, para isso, fazemos um **teste condicional**, que possui apenas duas respostas esperadas: verdadeiro ou falso (valor lógico). Quando o teste condicional resulta em um valor **verdadeiro**, a execução do programa segue por uma sequência de comandos que estão de acordo com o teste condicional ter resultado verdadeiro. Mas e se o teste condicional resulta em um valor **falso**? Então, a execução do programa segue por outra sequência de comandos, considerando este outro resultado do teste condicional.

É importante lembrarmos que o teste condicional fará um **desvio** na execução do programa. Para que serve? Esse desvio condicional é usado para **decidir** se um conjunto de comandos deve ou não ser executado.

Ou seja, para que exista um teste condicional, primeiramente, é preciso que exista uma **condição**. Ela é formulada com o uso dos **operadores relacionais** e **operadores lógicos** que tem como resultado os valores lógicos: verdadeiro ou falso. Não se lembra quais são eles? Então, acompanhe a Figura 1 a seguir:

Figura 1 – Operadores relacionais e operadores lógicos.

---

---

RELEMBRANDO OS OPERADORES RELACIONAIS:

- × Igual (==)
- × Diferente (<>) ou (!=)
- × Maior (>)
- × Menor (<)
- × Maior ou igual (>=)
- × Menor ou igual (<=)

RELEMBRANDO OS OPERADORES LÓGICOS:

- × E (&&)
  - × Ou (||)
  - × Negação (!)
- 
- 

Fonte: Elaborada pelo autor (2015).

## 4.2 Tipos De Estruturas De Seleção

As estruturas de seleção são divididas em dois tipos:

- × estrutura de seleção simples;
- × estrutura de seleção composta.

E para que servem? Esses tipos de estruturas de seleção são utilizados de acordo com o que precisamos decidir, isto é, temos um problema e precisamos resolvê-lo. Para elaborarmos uma estrutura de seleção, podemos utilizar os operadores relacionais e lógicos disponíveis na linguagem de programação estruturada C. Vejamos mais detalhes sobre esses tipos de estruturas. Acompanhe-nos!

### 4.2.1 Estrutura de Seleção Simples

Uma estrutura de seleção simples é formada pelo comando *if* e por um **teste condicional**. Confira a sintaxe na Figura 2:

Figura 2 – Sintaxe da estrutura de seleção simples com um comando.

```
if (testecondicional)
    comando;
```

Fonte: Elaborada pelo autor (2015).

A estrutura de seleção simples é implementada a partir de um teste condicional e pelo comando que será executado, se o teste condicional for verdadeiro. O que isso quer dizer na prática? Significa que o comando só

será executado se o teste condicional for verdadeiro. Se o teste condicional for **verdadeiro**, o programa **executa** o comando que está subordinado à estrutura de seleção *if*. Se o teste condicional for **falso**, o programa **não executa** o comando e segue a execução adiante.

Figura 3 – Programa com estrutura de seleção simples com um comando.

```
#include <stdio.h>
main()
{
    float notai, nota2, nota3, mediafinal, mediafinal_semestre;

    printf("Digitar a nota 1:\n");
    scanf("%f", &nota1);

    printf("Digitar a nota 2:\n");
    scanf("%f", &nota2);

    printf("Digitar a nota 3:\n");
    scanf("%f", &nota3);

    mediafinal= (nota1+nota2+nota3)/3;

    if (mediafinal >=7)
        printf ("Parabens! Media final maior que 7: %f\n", mediafinal);

    if (mediafinal<7)
        printf ("Atencao, media menor que 7: %f\n", mediafinal);
}
```

Fonte: Elaborada pelo autor (2015).

Vamos usar a Figura 3 para compreender como isso funciona. O programa faz a leitura de três notas. Nas linhas 6 até 13, armazena-as nas variáveis **nota1**, **nota2** e **nota3**; na linha 15, calcula a média aritmética das notas; e, por fim, nas linhas 17 a 20, faz o teste condicional para saber se a média final está igual ou acima de 7,0 ou abaixo de 7,0. Neste exemplo, utilizamos uma estrutura de seleção simples, com apenas um teste condicional e um comando a ser executado.

Uma estrutura de seleção simples pode necessitar que o programa execute mais de um comando, caso o teste condicional seja verdadeiro. Nesse caso, a sintaxe do comando *if* é representada, conforme mostrado na Figura 4:

Figura 4 – Sintaxe da estrutura de seleção simples com mais de um comando.

```
if (testecondicional)
{
    comando1;
    comando2;
    comando3;
}
```

Fonte: Elaborada pelo autor (2015).

Note que a sintaxe da estrutura de seleção simples com mais de um comando, na linguagem de programação estruturada C, espera a utilização de **chaves {}** - que identificam um bloco de comandos - quando existe mais de um comando a ser executado. Os comandos entre chaves {} somente serão executados se o teste condicional for verdadeiro.

Já o programa da Figura 5 executa os mesmos comandos que o programa anterior, no entanto, utilizamos uma estrutura de seleção simples que executa mais de um comando como resultado do teste condicional. Ou seja, quando é necessária a execução de mais de um comando em uma estrutura de seleção simples, é **obrigatório** o uso das **chaves {}**, conforme pode ser visto nas linhas 18 até 21 e da linha 24 até 27. Interessante, não é mesmo? Agora, acompanhe-nos para compreender como funciona a estrutura de seleção composta.

Figura 5 – Programa com estrutura de seleção simples com mais de um comando.

```
#include <stdio.h>
main()
{
    float nota1, nota2, nota3, mediafinal, mediafinal_semestre;

    printf("Digitar a nota 1:\n");
    scanf("%f", &nota1);

    printf("Digitar a nota 2:\n");
    scanf("%f", &nota2);

    printf("Digitar a nota 3:\n");
    scanf("%f", &nota3);

    mediafinal= (nota1+nota2+nota3)/3;

    if (mediafinal >=7)
    {
        printf ("Parabens! Media final maior que 7: %f\n", mediafinal);
        mediafinal_semestre = mediafinal;
    }

    if (mediafinal<7)
    {
        printf ("Atencao, media menor que 7: %f\n", mediafinal);
        mediafinal_semestre = mediafinal;
    }

    printf ("Por favor, passar na secretaria para fazer a rematricula\n");
}
```

Fonte: Elaborada pelo autor (2015).

#### 4.2.2 Estrutura de Seleção Composta

A estrutura de seleção composta também é implementada pelo comando *if*, por um **teste condicional**, mas inclui o comando *else*, conforme a sintaxe mostrada na Figura 6:

Figura 6 – Sintaxe da estrutura de seleção composta (*if...else*) com um comando.

```
if (testecondicional)
    comando1;
else
    comando2;
```

Fonte: Elaborada pelo autor (2015).



A estrutura de seleção composta também é implementada a partir de um **teste condicional**. Nele, o **comando1** será executado se o teste condicional for verdadeiro. Por outro lado, caso o teste condicional seja falso, o **comando2** é executado. O que acontece com o comando1 se o teste for falso? Não será executado o comando, passando diretamente para a opção **else**, e executará o **comando2**.

Uma estrutura de seleção composta também pode necessitar que o programa execute mais de um comando. E como fica a sintaxe do comando **if...else**? Note que, na linguagem de programação estruturada C, a sintaxe da estrutura de seleção composta com mais de um comando necessita das **chaves {}** quando existir mais de um comando a ser executado. Confira na Figura 7 a seguir.

Figura 7 – Sintaxe da estrutura de seleção composta (*if...else*) com mais de um comando.

```
if (testecondicional)
{
    comando1;
    comando2;
    comando3;
}
else
{
    comando4;
    comando5;
    comando6;
}
```

Fonte: Elaborada pelo autor (2015).

O programa da Figura 8 também executa os mesmos comandos que o programa anterior, no entanto, utilizamos uma estrutura de seleção composta **if...else** e as chaves (linhas 18 até 21 e linhas 23 até 26), uma vez que mais de um comando é executado em cada opção de teste. Onde entra o *Switch* neste contexto? Acompanhe-nos para compreender melhor!

Figura 8 – Programa com estrutura de seleção composta com mais de um comando.

```
#include <stdio.h>
main()
{
    float notai, nota2, nota3, mediafinal, mediafinal_semestre;

    printf("Digitar a nota 1:\n");
    scanf("%f", &notai);

    printf("Digitar a nota 2:\n");
    scanf("%f", &nota2);

    printf("Digitar a nota 3:\n");
    scanf("%f", &nota3);

    mediafinal= (notai+nota2+nota3)/3;

    if (mediafinal >=7)
    {
        printf ("Parabens! Media final maior que 7: %f\n", mediafinal);
        mediafinal_semestre = mediafinal;
    }
    else
    {
        printf ("Atencao, media menor que 7: %f\n", mediafinal);
        mediafinal_semestre = mediafinal;
    }
}
```

Fonte: Elaborada pelo autor (2015).

### 4.3 Estrutura do comando switch

A estrutura de seleção composta ou múltipla *case*, também chamada de *switch*, é utilizada quando existem situações mutuamente exclusivas, isto é, se uma situação for executada, as demais não serão. Quando for este o caso, um comando seletivo é o mais indicado e, em linguagem de programação estruturada C, esse comando chama-se *switch*, conforme mostrado na Figura 9:

Figura 9 – Sintaxe da estrutura de seleção composta *switch*.

```

switch (variavel)
{
    case valor1: lista de comandos; break;
    case valor2: lista de comandos; break;
    case valor3: lista de comandos; break;
    case valor4: lista de comandos; break;

    default: lista de comandos;
}

```

Fonte: Elaborada pelo autor (2015).

A estrutura de seleção composta ***switch*** testa o conteúdo de uma **variável**. Caso o conteúdo dessa variável seja verdadeiro, para algum dos casos dentro da estrutura *switch*, uma lista de comandos é executada e a execução da estrutura de seleção é encerrada com o comando ***break***. Se nenhuma das opções de teste for verdadeira, a lista de comandos da opção ***default*** será executada e, por fim, a estrutura de seleção ***switch*** é encerrada.

Figura 10 – Programa com estrutura de seleção composta *switch*.

```

#include <stdio.h>
main()
{
    int nota;

    printf("Digitar a nota do aluno:\n");
    scanf("%d", &nota);

    switch (nota)
    {
        case 7:
            printf("Nota = 7\n"); break;

        case 8:
            printf("Nota = 8\n"); break;

        case 9:
            printf("Nota = 9\n"); break;

        case 10:
            printf("Nota = 10\n"); break;

        default:
            printf("NOTA DIGITADA < 7\n"); break;
    }
}

```

Fonte: Elaborada pelo autor (2015).

O programa da Figura 10 executa, na linha 7, a leitura de um valor inteiro para uma nota, e na linha 9, testa esse valor na estrutura de seleção composta **switch**. Observe que, se um dos valores for verdadeiro, será mostrada uma mensagem na tela informando o valor da nota e o programa é encerrado, como pode ser observado nas linhas 10 até 22, com o comando **break**. Se o valor digitado não for suficiente para qualquer uma das condições do *case*, então, na linha 23, o comando que está no **default** será executado, e o programa encerra a execução.

Em outras palavras, o comando *switch* avalia o valor de uma variável para decidir qual *case* será executado. Cada *case* está associado a um possível valor da variável, que deve ser, obrigatoriamente, do tipo **caractere** ou **inteiro**. Outros tipos de dados não são aceitos pela linguagem de programação C para fazer o teste no comando *case*. O comando *break* encerra a execução da estrutura da seleção *switch* e o comando *default* executa uma lista de comandos, caso nenhuma das opções anteriores tenham sido executadas.

## Resumindo

Neste capítulo, apresentamos a estrutura de seleção e vimos que ela é uma das estruturas que trabalha as decisões quando o assunto é programação.

Aprendemos que a estrutura de seleção muda o raciocínio de programas simplesmente sequenciais para um raciocínio com base em tomadas de decisões, ou seja, com base em escolhas.

Por fim, vimos o que é uma estrutura de seleção, suas sintaxes, funcionalidades, os tipos simples e composta e múltipla, bem como exemplos implementados na linguagem de programação estruturada em C.

# 5

## Estruturas de Repetição na Linguagem de Programação Estruturada C

NESTE CAPÍTULO VAMOS apresentar as **estruturas de repetição** em linguagem de programação estruturada em C. A partir do que estudamos nas estruturas de seleção, estamos percebendo que os problemas computacionais não são resolvidos somente de forma sequencial. Os problemas computacionais exigem, muitas vezes, que alguns comandos sejam repetidos um determinado número de vezes, e para isso, usaremos as estruturas de repetição em programação estruturada. Para compreendermos como utilizar as estruturas de repetição, vamos estudar: o que é uma **estrutura de repetição**; para que serve; os comandos *for while*, e *do while*.

## Objetivos de Aprendizagem:

- × Analisar, compreender e aplicar estruturas de repetição e suas características em problemas computacionais.

## 5.1 Estruturas de Repetição

Uma estrutura de repetição em programação é um comando utilizado quando precisamos literalmente **repetir** algo ou alguma coisa. Em outras palavras, é quando precisamos repetir um ou mais comandos. Essa repetição é realizada enquanto que um **teste condicional** seja falso. Isto é, a execução do(s) comando(s) será realizada até que uma condição seja satisfeita.

Uma estrutura de repetição é utilizada quando uma parte do programa ou até mesmo o programa inteiro precisa ser repetido. O número de repetições pode ser fixo ou estar relacionado a uma condição. Dessa forma existem estruturas de repetição para cada situação (ASCENCIO e CAMPOS, 2007, p. 93).

### 5.1.1 Tipos de Estruturas de Repetição

Os tipos de estruturas de repetição são divididos em 3 (ASCENCIO e CAMPOS, 2007, p. 93):

- × **for**: estrutura de repetição para número definido de repetições (Estrutura **PARA**);
- × **while**: estrutura de repetição para número indefinido de repetições e teste no *início* (Estrutura **ENQUANTO**); e
- × **do .. while**: estrutura de repetição para número indefinido de repetições e teste no *final* (Estrutura **FAÇA .. ENQUANTO**).

A seguir, detalhamos cada um dos tipos de estruturas de repetição em programação estruturada em C.

### 5.1.2 Estrutura de Repetição para Número Definido de Repetições

Uma estrutura de repetição é utilizada quando se sabe **o número de vezes** que uma parte do programa deve ser repetido. O formato geral dessa estrutura é apresentado com o comando **FOR**, conforme mostrado na Figura 1:

Figura 1 – Sintaxe da estrutura de repetição para número definido de repetições.

```
=0; i<10; i++)
comando;
```

Fonte: Elaborada pelo autor (2015).

O comando será executado utilizando-se a variável *i* como variável de controle. O seu conteúdo vai variar do valor inicial (*i=0*) até o valor final (*i<10*). O comando *i++* incrementa a variável *i* em mais 1 a cada repetição. Esse incremento continua até que o **teste condicional seja falso**, isto é, quando a variável *i* for (*>= a 10*), como pode ser observado na Figura 2 a seguir:

Figura 2 – Programa com estrutura de repetição para número definido de repetições com apenas um comando.

```
#include <stdio.h>
main()
{
    int i;

    for (i=0; i<10; i++)
        printf ("\n O valor de i: %d", i);
}
```

```
O valor de i: 0
O valor de i: 1
O valor de i: 2
O valor de i: 3
O valor de i: 4
O valor de i: 5
O valor de i: 6
O valor de i: 7
O valor de i: 8
O valor de i: 9
```

Fonte: Elaborada pelo autor (2015).

O programa acima, na linha 6, define a variável de controle *i* como um inteiro. O comando **for** inicializa a variável com 0, faz o teste condicional se a variável *i* é menor que o número 10, incrementa a variável *i* depois de executar os comandos internos.

A linha 7 executa a exibição na tela dos valores da variável *i*, que varia de 0 a 9, como pôde ser observado na Figura 2.

Figura 3 – Sintaxe da estrutura de repetição para número definido de repetições com mais de um comando.

```
for (i=0; i<10; i++)
{
    comando1;
    comando2;
    comando3;
}
```

Fonte: Elaborada pelo autor (2015).

Na Figura 3, observe que da mesma forma que a estrutura de seleção, a sintaxe da estrutura de repetição, com mais de um comando, espera-se a utilização de **chaves { }** – que identifica o bloco de comandos que será repetido - para quando existir mais de um comando a ser executado. Os comandos entre chaves { } somente serão executados **enquanto** o teste condicional **for** verdadeiro. No caso da Figura 3, quando *i* < 10.

Figura 4 – Programa com estrutura de repetição para número definido de repetições com mais de um comando.

```
#include <stdio.h>
main()
{
    int i, j;

    j=0;
    for (i=0; i<10; i++)
    {
        printf ("\n O valor de i: %d", i);
        j=i+1;
        printf ("\n O valor de j: %d", j);
    }
}
```

Fonte: Elaborada pelo autor (2015).

A Figura 4 mostra um programa que executa os mesmos comandos que o programa anterior, no entanto, utilizamos uma estrutura de repetição que



**executa mais de um comando** como resultado do teste condicional. Observe que o **i** aumenta em 1, após executar os comandos internos, e antes de testar a condição de parada da repetição. Quando é necessária a execução de mais de um comando em uma estrutura de repetição, assim como em uma estrutura de seleção, é **obrigatório** o uso das **chaves { }**, conforme mostrado nas linhas 8 a 12. A Figura 5 mostra a saída de tela do programa com estrutura de repetição para um número definido de repetições.

Figura 5 – Saída na tela do resultado do programa, com estrutura de repetição para número definido de repetições com mais de um comando.

```
O valor de i: 0
O valor de j: 1

O valor de i: 1
O valor de j: 2

O valor de i: 3
O valor de j: 4

O valor de i: 5
O valor de j: 6

O valor de i: 7
O valor de j: 8

O valor de i: 9
O valor de j: 10

Pressione qualquer tecla para continuar...
```

Fonte: Elaborada pelo autor (2015).

### 5.1.3 Estrutura de Repetição para Número Indefinido de Repetições e Teste no Início

Essa estrutura de repetição é utilizada quando **não** se sabe o **número de vezes** que uma parte do programa deve ser repetido, embora também possa ser utilizada quando se conhece esse número. Essa estrutura baseia-se na análise de uma condição, que conforme já vimos, é formada por operadores lógicos ou relacionais, com respostas verdadeiras ou falsas. A repetição será feita enquanto a condição mostrar-se verdadeira. Nesses casos, os comandos de dentro da estrutura de repetição não serão executados (ASCENCIO e CAMPOS, 2007, p. 94-95). O formato geral dessa estrutura é apresentado com o comando **WHILE**, conforme mostrado na Figura 6 a seguir:

Figura 6 – Sintaxe da estrutura de repetição do while para número indefinido de repetições e teste no Início.

```
while (testecondicional)
    comando;
```

Fonte: Elaborada pelo autor (2015).

## Saiba mais

Existem situações em que o teste condicional da estrutura de repetição, que fica no início, resulta em um valor falso logo na primeira comparação.

Essa estrutura de repetição é implementada a partir de um **teste condicional**, isto é, enquanto o teste condicional for verdadeiro, o **comando** será executado. No momento que o teste condicional for falso, a execução da estrutura de repetição é encerrada e o programa não executará mais o comando.

Figura 7 – Programa e saída na tela do resultado do programa com estrutura de repetição para número indefinido de repetições com mais de um comando.

```
#include <stdio.h>
main()
{
    int i;

    i=0;

    while (i<10)
    {
        printf("\n O valor de i: %d",
            i=i+2;
    }
}
```

```
O valor de i: 0
O valor de i: 2
O valor de i: 4
O valor de i: 6
O valor de i: 8
```

Fonte: Elaborada pelo autor (2015).

Observe, na Figura 7, que primeiramente, na linha 6, a variável de controle *i* é inicializada com zero (0). Em seguida, na linha 8, o teste condicional é realizado; se o mesmo for verdadeiro, nas linhas 10 e 11, o programa executa os comandos, caso contrário, na linha 12, encerra a execução do comando de repetição. É importante salientar que, neste tipo de estrutura, é necessário incrementar a variável de controle *i*, na linha 11, para que ela chegue ao valor de parada da repetição.

### Saiba mais

Um *looping* ou laço infinito é aquele que apresenta sempre uma condição de teste verdadeira e por isso não consegue encerrar a execução. Para resolver este problema o programador deve se certificar de que em algum momento a condição seja falsa. Criando expressões corretas na condição de teste e alterando de forma correta a variável de controle.

#### 5.1.4 Estrutura de Repetição para Número Indefinido de Repetições e Teste no Final

Essa estrutura de repetição é utilizada quando o comando a ser executado deve ser feito ao menos uma única vez, por isso do teste condicional ser no final da estrutura de repetição. Essa estrutura baseia-se na execução dos comandos e o teste condicional após a primeira execução. Se o teste condicional for falso, então encerra-se a execução do programa tendo executado ao menos uma vez a sequência dos comandos. A repetição será feita enquanto a condição mostrar-se verdadeira (ASCENCIO e CAMPOS, 2007, p. 95-96).

Nessa situação, o teste condicional da estrutura de repetição fica no final da estrutura, isto é, os comandos serão executados, ao menos uma vez, mesmo que o teste condicional seja falso. O formato geral dessa estrutura é apresentado com o comando ***DO.. WHILE***, conforme Figura 8 a seguir:

## Programação Estruturada

Figura 8 – Sintaxe da estrutura de repetição do `do .. while` para número indefinido de repetições e teste no Final.

```
do
{
    comando1;
    comando2;
    comando3;
}while (testecondicional);
```


Fonte: Elaborada pelo autor (2015).

Essa estrutura de repetição é implementada a partir da execução da sequência de comandos, no mínimo, uma vez. No final, verifica-se o **teste condicional**. Se ele for falso, a estrutura de repetição continua a execução. No entanto, se o teste for verdadeiro, então a estrutura de controle de repetição é finalizada.

Figura 9 – Programa e saída na tela do programa com estrutura de repetição para número indefinido de repetições com teste no final.

```
#include <stdio.h> main()
main()
{
    int i=11;

    do
    {   printf("\n O valor de i: %d", i);
        i=i+2;
    }while (i<10);
}
```



O valor de i: 11

Fonte: Elaborada pelo autor (2015).

Observe que, na linha 4, a variável `i` é inicializada com o número 11. Em seguida, na linha 7, inicia a estrutura de controle de repetição, onde a sequência de comandos foi executada, até a linha 10. Somente, no final da execução dos comandos, na linha 11, é que o teste condicional foi realizado.

Em outras palavras, podemos observar que o programa executa o `printf()` mostrando na tela o valor da variável `i` que é o número 11; executa o

comando que incrementa a variável de controle  $i$  em 2 ( $i=i+2$ ), resultando 13. Daí o teste condicional resulta em falso e encerra o programa.

## Resumindo

No capítulo 5 conhecemos a estrutura de controle de repetição, uma das mais importantes estruturas de controle em programação de computadores. Neste capítulo apresentamos os três tipos de estruturas de controle: **for**, **while** e **do..while**.

Vimos, inclusive, que a estrutura de controle **for** é utilizada quando temos um número definido de repetições, isto é, quando sabemos exatamente quantas vezes é necessário que haja a repetição de determinados comandos. Já a estrutura de controle de repetição do tipo **while** é utilizada quando não sabemos o número definido de repetições, sendo que estas repetições dependem de um teste condicional. E a estrutura de controle de repetição do tipo **do..while** é utilizada quando não sabemos o número exato de repetições, também depende de um teste condicional.

Além disso, entendemos qual é a diferença entre as estruturas de controle de repetição **while** e **do..while**: a primeira tem o teste condicional no início e a segunda, no final.



# 6

## Utilização de Vetores Unidimensionais e Multidimensionais

VOCÊ SABIA QUE um tipo de dado muito utilizado em programação de computadores é o **Vetor**? Este tipo de dado é largamente utilizado porque é uma forma de organizar os dados na memória do computador, sem precisar criar muitas variáveis do mesmo tipo, o que facilita o acesso aos dados e essa facilidade, agiliza o processamento e pesquisa dos dados. E que ele pode ser de várias formas? Pois é! Por isso ele é visto como um tipo de dado complexo.

E essa complexidade vai além de sua forma, passa também pela maneira como é manipulado dentro do código fonte.

Para compreender como é possível manipular os dados armazenados em um vetor, vamos ao longo deste conteúdo estudar:

- × o que é um vetor;
- × o que são vetores unidimensionais;
- × o que são vetores multidimensionais.

Preparado para começar? Então siga em frente e bons estudos!

## Objetivo de Aprendizagem:

- × Analisar, compreender e aplicar o uso de vetores para armazenar;
- × Classificar e pesquisar listas e tabelas de valores.

## 6.1 Vetores

Segundo a professora Ana Fernanda Gomes Ascencio (2012):

os vetores, também conhecidos como variáveis compostas homogêneas unidimensionais, são um conjunto de variáveis do mesmo tipo, que possuem o mesmo identificador (nome) e são alocadas sequencialmente na memória. Como as variáveis têm o mesmo nome, o que as distingue é um índice que referencia sua localização dentro da estrutura. (ASCENCIO, 2012, p.145).

Em outras palavras, podemos definir um vetor como uma variável dividida em várias “caixas”, no qual cada “caixa” é identificada por um número que se refere à posição dessa “caixa” no vetor. Esse número é chamado de índice do vetor.

Em um vetor, cada uma das “caixas” pode armazenar um dado diferente, mas, obrigatoriamente, todos esses dados precisam ser do mesmo tipo de dado. Isso quer dizer que, se um vetor for do tipo de dado inteiro, somente dados inteiros poderão ser armazenados nas “caixas”. Da mesma forma que, se o vetor for do tipo de dado **caractere**, somente dados do tipo texto poderão ser inseridos nessas “caixas”.

Após essa visão geral, vamos a seguir aprender a criar um vetor. Acompanhe-nos!



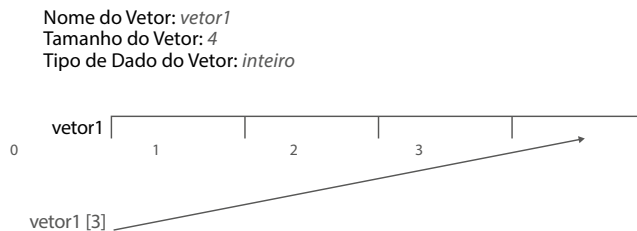
### 6.1.1 Criando um Vetor

Para criar um vetor, é necessário definir um **nome**. E como qualquer outra variável simples, também deve ser definido o **tipo de dado** e o **tamanho** do vetor.

O tipo de dado nada mais é que o tipo básico de dados que serão armazenados no vetor. Já o tamanho do vetor deve ser entendido como a quantidade de variáveis que vão compor esse vetor, ou seja, a quantidade de “caixas” que ele terá para armazenar os dados.

Exemplo:

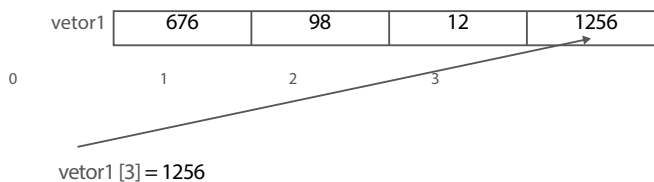
Figura 1 - Representação de um vetor.



Fonte: Elaborada pelo autor (2015).

Quando trabalhamos com um vetor, precisamos pensar em uma variável com várias “caixas”, e que cada “caixa” pode receber um determinado valor. No caso de nosso exemplo, estamos utilizando um **dado do tipo inteiro**. Isso quer dizer que o vetor1 pode receber até quatro (04) números inteiros. Veja a seguir.

Figura 2 - Dados inseridos em um vetor.



Fonte: Elaborada pelo autor (2015).

Perceba que, para criar um vetor, precisamos saber o **nome**, o **tipo de dado** e o **tamanho do vetor**. Mas é importante saber que os vetores podem ser de duas formas: unidimensionais ou multidimensionais.

A partir de agora, aprenda o que são vetores unidimensionais e como utilizá-los em seus projetos.

## 6.2 Vetores Unidimensionais

A sintaxe de um vetor unidimensional é:

```
tipo_de_dado nome_do_vetor [tamanho],
```

Ou seja, a declaração de um vetor é feita conforme pode ser visto abaixo e significa que foi declarado um vetor de quatro (04) posições. Confira:

```
int vetor1[4]
```

Na linguagem de programação C, os vetores têm 0 como índice do primeiro elemento, portanto quando criamos um vetor de inteiros com quatro (04) elementos, isso quer dizer que o **índice** do vetor varia de **0 a 3**. Assim, quando o compilador C encontra uma declaração de vetor, automaticamente, é reservado um espaço suficiente na memória do computador para armazenar o número de “caixas” especificadas como índice.

Por exemplo:

```
float vetor1[20]
```

Ao declarar isso, o compilador C reservará  $4 \times 20 = 80$  bytes. Vale lembrar que esses bytes são reservados sequencialmente, um após o outro.

### Importante

É necessário tomar cuidado para não referenciar uma posição de memória maior que o número definido para o tamanho do vetor. Ou seja, se o vetor foi definido com o tamanho de vinte (20) posições,

não se pode referenciar a posição vinte e dois (22). Isso porque podemos sobrepor outro dado que já esteja armazenado nessa mesma posição de memória do computador. Se isso acontecer, o conteúdo desse endereço de memória será substituído pelo dado atribuído na posição vinte e dois (22).



## 6.3 Atribuindo Valores a um Vetor

Quando desejamos preencher um vetor com dados, é necessário que o endereço da “caixa” seja informado, ou seja, o *índice* do vetor. Como mostrado abaixo, cada “caixa” do vetor possui um *índice*, e para preenchê-lo, precisamos informar a posição do vetor a que queremos atribuir um determinado valor. A seguir, mostramos como fazer a atribuição de valores às posições do *vetor1*:

```
vetor1 [0] = 676
vetor1 [1] = 98
vetor1 [2] = 12
vetor1 [3] = 1256
```

As atribuições em vetor são realizadas a partir do comando chamado de *atribuição de valor*, que tem como objetivo inserir um determinado conteúdo na variável vetor, na sua respectiva posição de memória (*índice*), como mostrado acima. Todos os vetores têm o primeiro elemento no índice zero (0). Assim, se tomarmos “k” como sendo o tamanho do vetor, a última posição do vetor é a de *índice* “k-1”. A sintaxe abaixo, mostra que foi colocado o inteiro 676 na **primeira** posição do vetor:

```
vetor1[0] = 676
```

E, a sintaxe a seguir, mostra que foi colocado o inteiro 1256 na **última** posição do vetor:

```
vetor1[3] = 1256
```

### 6.3.1 Projeto da Cadeia de Suprimento (ou Estratégia)

Preencher um vetor significa atribuir valores a todas as suas posições ao mesmo tempo. Dessa forma, devemos implementar um mecanismo que con-

trole o valor do **índice** do vetor. Para controlar esse valor, será criada uma variável do tipo inteiro, chamada **[i]**. Essa variável será usada no comando de repetição **for**, conforme exemplo a seguir.

Figura 3 - Preenchimento de um vetor com valor determinado.

```
main ()
{
    int i, vetor1[4];

    for(i=0; i<4; i++)
        vetor1 [i] = 30;
}
```

Fonte: Elaborada pelo autor (2015).

O preenchimento de um vetor pode ser feito de diversas formas, desde a mais simples, onde é atribuído um mesmo valor em todas as posições do vetor, até outras formas de preenchimento, as quais exigem uma atenção especial quanto à manipulação da posição que desejamos preencher no vetor.

Nesse momento, precisamos retomar os estudos sobre as estruturas de repetição.

Perceba que a sintaxe utilizada no último exemplo foi a da estrutura de repetição **for**. Veja também que o **vetor1** será preenchido com o inteiro trinta (30), da posição inicial zero (0) até a posição final três (3).

Lembre-se que a variável **i** indica a posição no vetor. Essa variável é inicializada com zero (0) para que o compilador **C** comece a atribuição de valores na posição **vetor1[0]**. Inclusive, é incrementada **i++** em uma posição (1), enquanto o **i** for menor que quatro (4): **i<4**.

Há outras formas de preencher um vetor? Sim, é o que vamos ver a seguir.

Figura 4 - Preenchimento de um vetor com valor determinado.

```
main ()
{
    int i, vetor1[4];

    for(i=0; i<4; i++)
        vetor1 [i] = i+1;
}
```

Fonte: Elaborada pelo autor (2015).

### 6.3.2 Preenchendo um Vetor com Intervalo de Números Determinados

Preencher um vetor com uma sequência de números de 1 a 4, por exemplo, significa que estamos atribuindo valores incrementados em um (1) **i+1**, a cada uma das suas posições. Assim, a sintaxe do comando para preenchimento do **vetor1** será como na Figura 4.

Ou, para preencher um vetor com uma sequência de números de 1 a 4, mas de forma decrescente, isto é, de 4 a 1. Para isto, a sintaxe do comando para preenchimento do **vetor1** será a seguinte, mostrada na Figura 5.

Isso quer dizer que podemos manipular o incremento do índice do vetor para preenchermos as posições de acordo com as necessidades.

Figura 5 - Preenchimento de um vetor com valor decrementado.

```
main ()
{
    int i, j, vetor1[4];
    j=4;

    for(i=0; i<4; i++)
    {
        vetor1 [i] = j;
        j--;
    }
}
```

Fonte: Elaborada pelo autor (2015).

### 6.3.3 Preenchendo um Vetor com Constantes para Definir o Tamanho do Vetor

Para preencher um vetor, podemos utilizar também constantes, definindo o tamanho do vetor, como pode ser observado a seguir.

Conforme a Figura 6, o comando cria uma constante chamada **TAM\_MAX** de tamanho dez (10) e utiliza essa constante no comando de repetição **for** para gerenciar o tamanho do **vetor1**. Com isso, o comando de repetição atribui os valores 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 às posições de zero (0) a nove (9) do **vetor1**.

Figura 6 - Preenchimento de um vetor com constante para o tamanho do vetor.

```
main ()
{
    int i, j, vetor1[4];
    j=4;

    for(i=0; i<4; i++)
    {
        vetor1 [i] = j;
        j--;
    }
}
```

Fonte: Elaborada pelo autor (2015).

### 6.3.4 Preenchendo um Vetor com Constantes para Definir o Tamanho do Vetor

Para preencher um vetor com dados de outro vetor, ainda podemos utilizar a constante tamanho do vetor **TAM\_MAX**. No entanto, é necessário definirmos outro vetor, que será chamado de **vetorcopia** para receber os dados do **vetor1**. Observe a seguir.

```
1 #define TAM_MAX 10
```

```
2 int vetor1 [TAM_MAX], vetor copia[TAM_MAX]
3 for(i=0; i<TAM_MAX; i++)
4     vetor copia[i] = vetor1[i]
```

Note que os comandos atribuem os valores das posições do **vetor1** para o **vetor copia**.

### 6.3.5 Encontrando o Maior e o Menor Valor Dentro de um Vetor

Dentre as operações mais utilizadas quando trabalhamos com vetores, vamos encontrar o **maior** e o **menor** valor dentro de um vetor. Essas duas operações são as mais básicas e essenciais para qualquer programador. Por exemplo, quando precisamos descobrir qual foi o país que mais exportou no ano de 2014? Ou qual foi o vendedor que teve mais vendas no mês de setembro? Ou qual foi o mês de maior alta do dólar? Para qualquer uma destas situações, precisamos saber quem é o maior elemento. .

Para encontrar o **maior valor** em um vetor, observamos os comandos a seguir:

```
1 #define TAM_MAX 10
2 int vetor1[TAM_MAX]
3 int maior
4 for(i=0; i<TAM_MAX; i++)
5     scanf("%f",&vetor1[i])
6 maior = vetor1[0]
7 for(i=0; i<TAM_MAX; i++)
8     if vetor1[i] > maior
9         maior = vetor1[i]
10 printf("O maior elemento é %f\n",maior)
```

Nesse exemplo, na linha 1, observe que a constante **TAM\_MAX** é definida com tamanho dez (10). Já na linha 2, a variável **vetor1** é definida como

um inteiro de tamanho **TAM-MAX**; e na linha 3, a **variável maior** é definida como um inteiro.

Seguindo a análise do código-fonte, nas linhas 4 e 5, o **vetor1** é lido a partir do comando **scanf**, em que o usuário pode digitar os valores para serem inseridos no vetor.

Na linha 6, a variável **maior** recebe o conteúdo da posição zero (0) do **vetor1**. Reveja essa linha na sequência.

```
6    maior = vetor1[0]
```

Mas o que isso quer dizer? Inicialmente, quer dizer que o conteúdo da primeira posição do **vetor1** é considerado como sendo o maior valor do vetor. Vale ressaltar que esse comando é importante, pois a partir dessa atribuição serão feitas as comparações para encontrar o maior valor do **vetor1**.

Nas linhas 7, 8 e 9, são realizadas as comparações para encontrar o maior valor do **vetor1**, conforme podemos conferir:

```
7    for(i=0; i<TAM_MAX; i++)  
8        if vetor1[i] > maior  
9            maior = vetor1[i]
```

Observe que comando **for**, na linha 7, faz a varredura no **vetor1** para encontrar o **maior** valor, a partir do comando de seleção **if**. Tal comando de seleção compara o conteúdo do **vetor1** na posição **i** com o conteúdo da variável **maior**. Assim, se esse conteúdo for maior, então o mesmo é atribuído à variável **maior** e passa a ser o maior valor encontrado no vetor. Lembre-se que esses comandos são executados até que o **vetor1** seja percorrido totalmente em todas as suas posições. A linha 7 mostra a execução do comando **for** que tem como objetivo percorrer todos os índices do **vetor1** para encontrar o maior valor armazenado no vetor. Se o programa encontrar um valor maior que o conteúdo da variável **maior** então este valor é atribuído à variável **maior**, que por sua vez, tem o seu conteúdo atualizado.

Agora, para encontrar o menor valor de um vetor, devemos proceder da mesma forma, apenas trocando o operador para < (menor) no momento de

comparar os valores das variáveis, na linha 8, e atribuindo o conteúdo do **vetor1** a uma variável chamada de **menor**, na linha 6, conforme mostrado a seguir:

```
1 #define TAM_MAX 10
2 int vetor1[TAM_MAX]
3 int menor
4 for(i=0; i<TAM_MAX; i++)
5     scanf("%f",&vetor1[i])
6 menor = vetor1[0]
7 for(i=0; i<TAM_MAX; i++)
8     if vetor1[i] < menor
9         menor = vetor1[i]
10 printf("O menor elemento é %f\n",menor)
```

## 6.4 Vetores Multidimensionais

Agora vamos dar início ao estudo sobre **vetor multidimensional**, também chamado de **matriz**. A partir desse ponto, adotamos o nome **matriz** para referenciar um **vetor multidimensional** para que possamos, mais facilmente, diferenciar um tipo de vetor do outro.

Segundo a professora Ana Fernanda Gomes Ascencio (2012):

Uma matriz é uma variável composta homogênea multidimensional. Ela é formada por uma sequência de variáveis, todas do mesmo tipo, com o mesmo identificador (mesmo nome), e alocadas sequencialmente na memória. Uma vez que as variáveis têm o mesmo nome, o que as distingue são índices que referenciam sua localização dentro da estrutura. Uma variável do tipo matriz precisa de um índice para cada uma das suas dimensões. (ASCENCIO, 2012, p. 187).

Em outras palavras, podemos definir uma matriz como uma grande tabela, com linhas e colunas, onde o encontro de cada linha com cada coluna forma uma “caixa”. Nesse caso, cada “caixa” é identificada por dois números: um número indica a linha e o outro número indica a coluna, e que ambos



significam a posição do elemento dentro da matriz. Da mesma forma que nos vetores, esses dois números, de linha e coluna, indicam o **índice da matriz**.

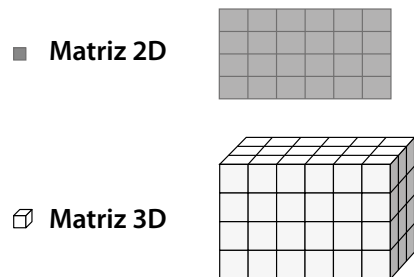
Logo, em uma matriz, assim como em um vetor, cada uma das “caixas” pode armazenar um dado diferente, mas obrigatoriamente, todos esses dados precisam ser do mesmo **tipo**. Isso quer dizer que, se uma matriz for do tipo de dado **float**, somente dados do tipo **float** poderão ser armazenados em suas “caixas”. Uma matriz pode ser de 2 ou mais dimensões. Uma matriz com 2 (2D) dimensões possui “n” linhas e “n” colunas.

No entanto, também é possível implementarmos, em linguagem de programação estruturada, matrizes de 3 (3D), considerando linhas, colunas e uma terceira dimensão, o plano, conforme mostrado na Figura 7.

A estrutura de uma matriz é composta por uma quantidade x de linhas e y de colunas, diferentemente, da estrutura de um vetor, que é composto apenas por uma linha com n colunas. Por exemplo, se precisarmos armazenar as 3 notas de avaliações de uma turma de 20 alunos. Nesse caso, utilizaremos uma estrutura do tipo matriz, onde poderemos definir 20 linhas para armazenar as notas dos 20 estudantes e 3 colunas para armazenar as 3 notas de avaliações, de cada estudante. Vamos imaginar que uma matriz é semelhante a um tabuleiro de xadrez ou a uma planilha eletrônica no Microsoft Excel, onde temos linhas e colunas para armazenar os dados. Os dados por sua vez, serão acessados informando qual o número da linha e o número da coluna.

Assim como no Microsoft Excel (Figura 8), as linhas e as colunas são acessadas por um número de linha e um número de coluna. No exemplo do Microsoft Excel, as linhas são numeradas a partir do número 1 e as colunas são identificadas a partir da letra A. Para acessar um elemento é necessário fazer referência à letra da coluna e ao número da linha, como A1 que representa a coluna A e a linha 1, e C19 que representa a coluna C e a linha 19. A vantagem de usar uma estrutura desse tipo é que temos todos os dados armazenados

Figura 7 - Dimensões de vetores e matrizes.



Fonte: Elaborada pelo autor (2015).

Figura 8 - Planilha no Microsoft Excel como exemplo de uma matriz.

	A	B	C
1	A1		
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			C19
20			
21			

Fonte: Elaborada pelo autor (2015).

somente em um lugar, em uma sequência de memória.

Veja que a sintaxe de uma **matriz de 2**.

(2D) **dimensões** é:

```
tipo_de_dado      nome_da_
matriz [tamanho_linha, tama-
        nho_coluna]
```

Ou seja, a declaração de uma **matriz** é feita sempre dessa forma. Por exemplo, uma matriz com quatro (04) linhas e três (03) colunas, terá doze (12) posições, do conceito de matriz em Matemática:  $4 \times 3 = 12$ . Veja:

```
int
matriz1[4,3]
```

É importante ressaltar que as matrizes têm **[0,0]** como índice do primeiro elemento. Portanto, quando criamos uma matriz de inteiros com quatro (04) linhas e três (03) colunas, significa que o **índice** da matriz varia doze (12) vezes. Assim, quando o compilador C encontra uma declaração de matriz, automaticamente é reservado um espaço grande suficiente na memória do computador, para armazenar o número de “caixas” especificadas como **índice**.

Por exemplo:

```
float matriz1[4,3]
```

Se declararmos a matriz acima o compilador C reservará  $4 \times 12 = 48$  bytes, e esses bytes serão reservados sequencialmente, um após o outro.

Um exemplo prático para utilização de uma matriz de 2 (2D) dimensões é quando precisamos armazenar as notas bimestrais de “n” estudantes. Então teremos para cada linha, os índices representando os nomes dos estudantes, e para cada coluna, as suas respectivas notas bimestrais.

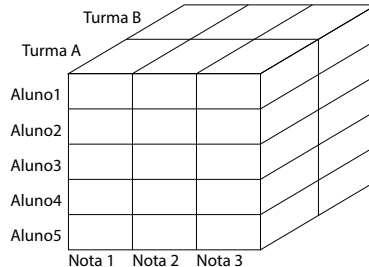
Para criar uma matriz de 3 (3D) dimensões, a sintaxe na linguagem de programação estruturada em C é a mesma que usamos para criar uma matriz de 2 (2D) dimensões, conforme mostrado a seguir:

Sintaxe para criar uma **matriz de 3 (3D) dimensões** é:

```
tipo_de_dado  nome_da_matriz  [tamanho_linha,  tamanho_
coluna, tamanho 3ª dimensão]
```

Retomando o exemplo da matriz de 2 (2D) dimensões, das notas dos estudantes, uma matriz de 3 (3D) dimensões pode ser usada para armazenar as “n” notas bimestrais, dos “n” estudantes, de “n” turmas da escola. Então teremos para cada linha, os nomes dos estudantes (Aluno1, Aluno2, Aluno3, AlunoN), para cada coluna, as respectivas notas bimestrais (N1, N2, N3), e na profundidade, as turmas (Turma A, Turma B), como mostrado na Figura 9.

Figura 9 - Exemplo de uma matriz de 3 (3D) dimensões.



Fonte: Elaborada pelo autor (2015).

### 3.5 Criando uma Matriz

Para criar uma matriz, é necessário definir um **nome**, como qualquer outra variável simples, bem como o **tipo de dado** e o **tamanho** da matriz. Sendo que o **tipo de dado** é o tipo básico que será armazenado na matriz e o seu **tamanho** corresponde à quantidade de linhas e colunas que vão compor a matriz, ou seja, a quantidade de “caixas” que a matriz terá para armazenar os dados.

### Exemplo:

Figura 10 - Representação gráfica de uma matriz.

Nome da Matriz: *matriz1*  
Tamanho da Matriz: *[4,3]*  
Tipo de Dado da Matriz: *inteiro*

		coluna	
	0	1	2
0			
1			
linha 2			
3			

Matriz1[2,1]

Fonte: Elaborada pelo autor (2015).

Quando trabalhamos com uma matriz, precisamos pensar em uma tabela com várias “caixas” (Figura 10), e que cada “caixa” pode receber um determinado valor, nesse caso, um dado do tipo **inteiro**. Isso quer dizer que a **matriz1** pode receber até doze (12) números inteiros, como mostrado na Figura 11, a seguir:

Figura 11 - Dados inseridos em uma matriz.

matriz1:

	0	1	2
0	20	43	654
1	9823	765	1345
2	591	6	123
3	715	251	875

Matriz1 [2,1] = 6

Fonte: Elaborada pelo autor (2015).

Perceba que, para criar essa matriz, precisamos saber o **nome**, o **tipo de dado** e o **tamanho** da matriz.

### 3.6 Atribuindo Valores a uma Matriz

Quando desejamos preencher uma matriz com dados, é necessário que o endereço da “caixa” seja informado, isto é, o **índice** da matriz. Assim, cada “caixa” da matriz possui um **índice**, que é composto pelo número da linha e número da coluna, para preenchê-lo, precisamos informar a posição da matriz que queremos atribuir um determinado valor. A seguir, veja como fazer a atribuição de valores das posições da **matriz1**:

```
matriz1 [0,0] = 20
matriz1 [1,2] = 1345
matriz1 [3,1] = 251
matriz1 [2,0] = 591
```

As atribuições de valor em uma matriz, ou em qualquer posição de memória atribuída a uma variável em um programa, são realizadas a partir do comando chamado de **atribuição de valor**, que tem como objetivo inserir um determinado conteúdo na posição determinada da matriz (índice), como mostra o último exemplo.

Todas as matrizes têm o primeiro elemento na linha e coluna zero [0,0]. A sintaxe abaixo, mostra que foi colocado o inteiro 20 na **primeira** posição da matriz:

```
matriz1[0,0] = 20
```

E, a sintaxe a seguir, mostra que foi colocado o inteiro 875 na **última** posição da matriz:

```
matriz1[3,2] = 875
```

#### 6.6.1 Preenchendo uma Matriz

Preencher uma matriz significa atribuir valores a todas as suas posições ao mesmo tempo. Dessa forma, devemos implementar mecanismos que controlem o valor da **linha** e da **coluna**, chamados de **índices da matriz**. Para controlar o valor dos índices, serão criadas duas (02) variáveis do tipo inteiro,

uma denominada **i** e a outra, **j**, representando as linhas e as colunas, respectivamente. Essas variáveis serão usadas no comando de repetição **for**, conforme o seguinte exemplo, mostrado na Figura 12.

É importante salientar que o primeiro comando **for**, que tem como variável de controle **i** (que representa as linhas da matriz), inicia o preenchimento da matriz na linha 0, enquanto o segundo **for**, que tem como variável de controle **j** (que representa as colunas da matriz), executa o preenchimento da matriz, coluna a coluna, até concluí-las, e passar para a linha seguinte.

Figura 12 - Preenchendo uma matriz de inteiros com o número 30.

```
#define TAM_MAX 10
#include <stdio.h>
main ()
{
    int i, vetor1[TAM_MAX];

    for(i=0; i<TAM_MAX; i++)
        vetor1[i] = TAM_MAX - i;
    for(i=0; i<TAM_MAX; i++)
        printf("%d\n", vetor1[i]);
}
```

Fonte: Elaborada pelo autor (2015).

o inteiro trinta (30), desde a linha zero (0) e coluna zero (0) até a posição final, na linha quatro (4) e coluna três (3), como mostrado na Figura 13.

Vale lembrar que o preenchimento de uma matriz pode ser feito de diversas formas, desde a mais simples até outras formas de preenchimento, as quais exigem uma atenção especial quanto à manipulação da posição na matriz que desejamos preencher.

Por isso, vamos analisar o exemplo da Figura 12. De acordo com ele, a sintaxe utilizada foi a da estrutura de repetição **for**, sendo que a **matriz1** foi preenchida com

Figura 13 - Matriz preenchida com o número 30.

	0	1	2
0	30	30	30
1	30	30	30
2	30	30	30
3	30	30	30
4	30	30	30
5	30	30	30

	0	1	2
6	30	30	30
7	30	30	30
8	30	30	30
9	30	30	30

Fonte: Elaborada pelo autor (2015).

Na linha 6, o comando For com a variável **i** indica a linha da matriz. Na linha 8, a variável **j** mostra a coluna da matriz que deverá ser armazenado o valor. Tais variáveis são inicializadas com zero (0), linha 6 e linha 8, para que o compilador C comece a atribuição de valores na posição **matriz1[0,0]**. Portanto as variáveis são assim incrementadas, na linha 6 e linha 8, da seguinte forma: (**i++**; **j++**) em uma (01) posição, enquanto **i < 10** que é o número de linhas, e **j < 3**, que é o número de colunas.

Logo, quando essas condições forem satisfeitas, o compilador C compreende que a matriz já foi preenchida totalmente, e encerra o programa. Dessa forma, a **matriz1** é preenchida com o inteiro trinta (30) em todas as suas posições, desde a posição **matriz1[0,0]** até a posição **matriz**, que totaliza as trinta (30) posições da matriz.

## Importante

É necessário ressaltar que todas as manipulações de matrizes são semelhantes às manipulações dos vetores. A única diferença, no caso das matrizes, é que para realizar qualquer comando, é preciso estar atento aos índices que indicam a posição das linhas e colunas (Figura 14 e Figura 15). Isso porque precisamos ter em mente que uma matriz é um **vetor multidimensional**, isto é, que ela possui duas ou mais dimensões: **linha** e **coluna**.

O preenchimento de uma matriz pode ser feito de outras formas, como por exemplo, preencher apenas uma linha específica da matriz, como mos-

trado na linha 9, da Figura 14. Observe que somente a linha 1 foi preenchida. Note que o índice que representa a linha está fixo com o número 1, o que significa que esse comando preencherá somente a linha 1, e encerrará a execução do programa. Da mesma forma, pode ser feito com a coluna, podemos preencher somente uma coluna, fixando o número do índice da coluna.

Figura 14 - Matriz preenchida com o número da linha fixo na linha 1.

```
notas [1][j] = j;

#include <stdio.h>
int main ()
{
    int i, j, notas[10][3];

    for (j=0; j<3; j++)
    {
        notas[1][j] = j;

        for (j=0; j<3; j++)
        {
            printf("Matriz linha 1: %d\n\n", notas[1][j]);
        }
    }
}
```

Fonte: Elaborada pelo autor (2015).

Na figura 15, apresentamos um exemplo de como somar o conteúdo da linha 1 da matriz.

Figura 15 - Soma dos conteúdos da linha 1 da matriz.

```
include <stdio.h>
int main ()
{
    int i, j, notas[10][3], soma;

    for (i=0; i<10; i++)
    {
        for (j=0; j<3; j++)
        {
            notas[i][j] = 3;
        }
    }

    soma=0;
}
```



```

for (j=0; j<3; j++)
{
    soma = soma + notas[1][j];
    i++;
}

printf("Soma linha 1: %d\n\n", soma);
}

```

Fonte: Elaborada pelo autor (2015).

No exemplo, o primeiro comando `for` preenche a matriz `notas[10][3]` com o número 3 em todas as posições, e o segundo `for` soma o conteúdo da linha 1 da matriz e o armazena na variável `soma`.

## Resumindo

Neste capítulo, vimos o que são vetores. Inclusive, conhecemos dois tipos: os unidimensionais e os multidimensionais, chamados também de matrizes, por trabalharem com duas ou mais dimensões.

Percebemos ainda que os dados armazenados em vetores são como se estivessem dentro de “caixas”, organizadas sequencialmente na memória do computador. Além disso, compreendemos que não se pode mexer em uma “caixa” não alocada na memória, porque senão podemos perder dados previamente armazenados provenientes de outras operações.

Também aprendemos neste capítulo a definir vetores unidimensionais e vetores multidimensionais, também chamados de matrizes. E, por fim, vimos operações básicas para manipulação de vetores e matrizes em linguagem de programação estruturada C.



# 7

## Aplicações de Funções e Procedimentos

NESTE CAPÍTULO TRABALHAREMOS o conceito de **Funções** e **Procedimentos** em programação estruturada com C. Compreenderemos como inserir e quando usar uma função na estrutura de um programa em C.

E, também, aprenderemos como inserir e quando usar um procedimento na estrutura de um programa em C. Discutiremos a utilização de funções e procedimentos na programação estruturada.

Então siga em frente e bons estudos!

### Objetivo de aprendizagem:

- × Saber como utilizar as funções e os procedimentos, a fim de otimizar os programas feitos em C.

## 7.1 Funções

A modularização em programação de computadores tem como objetivo principal dividir o código fonte de um software em diversas partes. Uma das vantagens da modularização é a legibilidade do código, isto é, a facilidade de leitura do código fonte por qualquer outra pessoa. A modularização organiza o código fonte em partes diferentes, com início, meio, fim, e objetivos bem definidos. A modularização facilita a fase de testes do software, porque pode-se testar partes do sistema. Desta forma, facilita a manutenção do software, e agiliza a implementação de melhorias no sistema. Uma das principais vantagens do uso da modularização é a possibilidade de reutilização do código fonte em outros sistemas.

Segundo a professora Ana Fernanda Gomes Ascencio (2012):

Sub-rotinas, também, chamadas de subprogramas, são blocos de instruções que realizam tarefas específicas. O código de uma sub-rotina é carregado uma vez e pode ser executado quantas vezes forem necessárias. Como o problema pode ser subdividido em pequenas tarefas, os programas tendem a ficar menores e mais organizados. (ASCENCIO, 2012, p. 230).

Em outras palavras, a modularização é implementada a partir de funções ou procedimentos. Uma função, que também pode ser chamada de sub-rotina ou subprograma, é uma unidade autônoma do código-fonte que foi projetada para executar determinados comandos em particular. Entende-se por código-fonte, em linguagem de programação C, um composto de pequenas partes que, quando executadas em conjunto, desempenham as tarefas programadas. Assim, as funções representam a divisão de grandes tarefas, em pequenas partes que são executadas uma por vez. Uma função recebe dados, processa-os e retorna uma informação como resposta.

Perceba que, ao criar uma função, evitamos a repetição de códigos que têm o mesmo objetivo dentro de um código-fonte. De modo que, se uma tarefa específica é repetida, essa sequência pode ser transformada em uma função. Logo, essa função poderá ser chamada diversas vezes pelo programa principal.

Vale lembrar que a linguagem de programação C foi toda pensada usando funções. Um exemplo disso são as funções que utilizamos na biblioteca padrão de funções do C.

Até o momento todos os exemplos que trabalhamos possuem apenas uma função, a função principal chamada **main()**. Isso faz com que tenhamos programas maiores em termos de quantidade de linhas de código e repetição de trechos, já que o programa é executado sequencialmente dentro da função principal.

O tamanho do código-fonte é um indicador importante ao falarmos de otimização e uso do processador, porque a quantidade de linhas de código-fonte tem influência direta no desempenho do software. Isto é, quanto mais linhas de código-fonte, mais “trabalho” o processador terá para interpretá-las e executá-las. Consequentemente, o processador ocupará mais memória e tempo de processamento para concluir a execução. Por isso, quanto mais otimizado for o código-fonte, melhor será o desempenho final do software.

Uma alternativa é dividirmos esse código em partes reusáveis, isto é, em partes que executam determinadas tarefas, para quando precisarmos, poderemos utilizá-las quantas vezes forem necessárias, sem termos que reescrever toda a função principal. Um exemplo prático desta situação é quando precisamos implementar o cadastro de um cliente em um banco de dados. Esse mesmo código-fonte que implementa o cadastro de um cliente, pode ser utilizado para outros cadastros, como fornecedor, estudante, colaborador, entre outros. As particularidades de cada caso devem ser tratadas conforme necessário, mas a essência do cadastro de uma pessoa pode ser reutilizada quantas vezes forem necessárias.

Para isso, precisamos “dividir para conquistar”<sup>1</sup>, ou seja, dividir os problemas em subproblemas menores e mais tratáveis. Essa divisão do código em partes menores, chamamos de **modularização**. Além de evitar repetições de código, a modularização deixa o programa cada vez mais otimizado, isto quer dizer, que a modularização deixa o programa cada vez mais reutilizável e de manutenção bem mais simples.

Na linguagem de programação C, é possível criar módulos de código que desempenham alguma tarefa específica. Esses módulos são chamados de **funções**. E essas funções podem ou não retornar um valor.

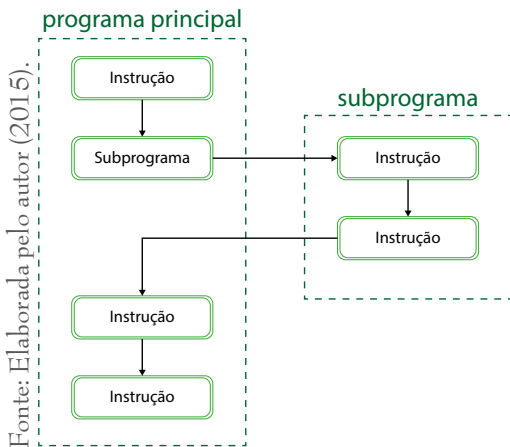
---

<sup>1</sup> Expressão utilizada pelo governante romano César (*divide et impera*) e pelo imperador francês Napoleão Bonaparte. Disponível em: <<http://www.infoescola.com/historia/revolucao-francesa/>>. Acesso em: 24 out. 2015.

A seguir veja as principais vantagens da modularização:

- × Evitar a repetição do mesmo trecho de código-fonte.
- × Reaproveitar códigos-fonte existentes.
- × Facilitar as modificações no código-fonte.
- × Organizar os programas em partes que podem ser compreendidas isoladamente.
- × Melhorar a estruturação dos programas.
- × Facilitar o entendimento dos programas por parte de quem não escreveu o código inicialmente.
- × Facilitar os testes dos programas.
- × Facilitar a solução de problemas complexos.

Figura 1 – Desvio do fluxo de execução em um programa.



É de suma importância saber que uma função não pode ser executada individualmente, isto é, uma função precisa ser chamada ou invocada pela função principal **main()**. Essa “chamada” desvia o fluxo de controle para a função, além de executar os comandos da função e retorná-los para a função principal, isto é, em programação estruturada, o código tem somente uma entrada de dados e uma saída de dados, mesmo havendo desvios no programa – como estruturas de seleção, repetição e funções – o programa sempre retornará ao

mesmo ponto inicial após o desvio de execução, como mostrado na figura 1.

Na linguagem de programação C, existem funções predefinidas como, por exemplo: **clrscr()**, **gets()**, **strcmp()**, **strcpy()** etc. Essas funções são adicionadas aos programas pela diretiva **#include**, no momento da compilação.

## Glossário

A **diretiva #include** informa ao compilador que um arquivo especificado (bibliotecas) precisa ser inserido no momento da compilação.

A linguagem de programação estruturada C possui diversas bibliotecas, que por sua vez, possuem uma infinidade de funções já definidas. Uma biblioteca é definida como um arquivo com extensão **.h** e possui a definição de funções, macros, variáveis e/ou constantes que podem ser utilizados nos nossos códigos-fonte, bastando usar a diretiva **#include** para chamá-las. Quando inserimos uma diretiva **#include** nos nossos programas e especificamos o nome da biblioteca **.h**, estamos informando ao compilador, que naquele ponto do código-fonte devem ser inseridas as funções já definidas na biblioteca especificada. Isto porque, estas funções serão utilizadas ao longo do código-fonte do nosso programa. Dessa forma, quando estas funções forem “chamadas”, já estarão inseridas no código-fonte, e, portanto, poderão ser executadas.

A biblioteca padrão C ou “**C standard library**” é a biblioteca padrão de todo compilador C. Esta biblioteca padrão fornece macros, definição de tipos e funções para lidar com sequência de caracteres (strings), computação matemática, processamento de entrada e saída de dados em tela, alocação dinâmica de memória, e muitas outras funções. A seguir, apresentamos algumas bibliotecas que utilizamos até o momento:

Quadro 1 – Bibliotecas da linguagem de programação estruturada C.

Biblioteca	Descrição
stdio.h	Define funções para realizar operações de entrada e saída de dados.
stdlib.h	Define funções para geração de números aleatórios, comunicação com o ambiente, aritmética de inteiros, busca, ordenação e conversão de dados.
string.h	Define funções para manipulação de strings (cadeia de caracteres).
time.h	Define funções para ler e converter datas e horas

Fonte: SCHILDT (1999).

Além das funções predefinidas, o programador pode criar quantas funções forem necessárias, dependendo da complexidade do problema que está por resolver. Muitas vezes, as funções precisam receber valores externos para realizar a sequência de comandos, chamados de **parâmetros**.

## Glossário

**Parâmetros** são representados por uma lista de variáveis que são organizadas sequencialmente, dentro de parênteses, logo após o nome da função. Todo parâmetro passado para uma função tem um nome e um tipo de dado, assim como uma variável, que possui um nome e um tipo de dado. Parâmetros são dados que a função necessita receber do programa que a chamou para poder executar.

### 7.1.1 Definição de uma Função

Toda função em linguagem de programação C deve ser declarada ou criada antes da função principal. Essa declaração deve conter o tipo do valor a ser retornado pela função, além do nome da função e dos tipos de parâmetros que precisam ser fornecidos em uma chamada da função. No corpo da função contém o cabeçalho completo desta, bem como a implementação dos comandos (as suas tarefas). Quando a função termina a execução dos comandos internos, o programa retorna ao ponto inicial de onde a função foi chamada (invocada) e continua a execução daquele ponto em diante, ou finaliza a execução.

Quando todos os comandos do corpo da função são executados, a execução do programa retorna para linha, imediatamente, seguinte à chamada da função, como mostrado na Figura 1.

Veja a forma geral para definir uma função:

```
tipo_retornado nome_da_função (lista de parâmetros)
{
    corpo da função;
}
```

Onde, **tipo\_retornado** é o dado que a função mostrará como resultado da execução dos comandos no corpo da função. Já o **nome\_da\_fun-**



**ção** é o nome dado à função para que ela tenha uma identificação única dentro do programa, assim como as variáveis possuem um nome único. Por fim, a **lista de parâmetros** é relação dos dados que precisam ser passados para a função, a fim de que os comandos do corpo da função sejam executados corretamente.

A primeira função que conhecemos na linguagem de programação C++, é a função principal chamada de `main ()`. A função `main ()` devolve um inteiro para o processo chamador, que é, geralmente, o sistema operacional. (SCHILDT, 1990, p. 139).

Você sabia que existem diversos tipos de funções e, para cada tipo, há uma funcionalidade e uma aplicação diferente? Então, conheça a seguir os **tipos de funções**, suas funcionalidades, bem como alguns exemplos de sua utilização.

## 7.2 Funções Sem Passagem de Parâmetros e Sem Retorno

Uma função pode ser chamada, e concluir a execução sem ter recebido valores como parâmetros, nem tampouco, ter de retornar algum resultado. Essas funções são chamadas de **funções sem passagem de parâmetros e sem retorno**. Isto quer dizer que, não necessariamente, se envia parâmetros à uma função, nem se espera algum valor de retorno. A função simplesmente executa o que há de ser executado, sem precisar de dados de entrada, nem retornar algum dado de saída. Nesses casos, quando uma função não retorna um valor para a função que a chamou ela é declarada como **void**.

O tipo mais simples de função é aquele que não recebe nenhuma informação no momento de sua chamada e que também não repassa nenhum valor para quem a chamou. (ASCENCIO, 2012, p. 235).

Por exemplo, a função para limpar a tela denominada **`clrscr()`** e que está definida na biblioteca `<conio.h>`. Essa função quando executada não recebe parâmetros e não envia nenhum retorno após a sua conclusão. Essa função apenas executa o comando de limpar a tela. É um exemplo típico de função sem passagem de parâmetros e sem retorno.

## Programação Estruturada

Figura 2 – Exemplo função sem passagem de parâmetros e sem retorno.

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

int main(){

    printf ("Estamos testando a funcao clrscr () da biblioteca <conio.h> para limpar a tela\n\n");
    printf ("Tecle ENTER para limpar a tela").;
    getchar();
    clrscr();
    printf ("Parabéns!");
    getchar();
}
```

Fonte: Elaborada pelo autor (2015).

Figura 3 – Exemplo de função sem passagem de parâmetros e sem retorno.

```
#include <stdio.h>
void soma ( )
{
    int V1, V2, soma;
    printf("\nDigite o primeiro número: ");
    scanf("%d", &V1);
    printf("\nDigite o segundo número: ");
    scanf("%d", &V2);
    soma = V1 + V2;
    printf("\nA soma dos números é: %d\n", soma);
}

int main ()
{
    soma();
}
```

Fonte: Elaborada pelo autor (2015).

Observe na Figura 2, que, nas linhas 6 e 7, o programa mostra duas mensagens na tela explicando que está sendo feito um teste com a função para limpar a tela. Na linha 9, a função `clrscr()` é chamada sem nenhuma passagem de parâmetro, executa a limpeza da tela, e encerra a execução da mesma. Em seguida, é exibido o

Parabéns na tela e encerrado o programa. As linhas 8 e 11 tem como objetivo esperar que o usuário digite a tecla ENTER para prosseguir. Após a execução da linha 11, o programa encerra a execução.

Outro exemplo, o cálculo da soma entre dois números inteiros “a” e “b” armazena um resultado em uma variável chamada **soma**. Veja:

### Importante

A execução de todo programa escrito em linguagem de programação C sempre inicia pela função principal chamada `int main()`, especificada nas linhas 13, 14, e 15.

No exemplo da Figura 4, perceba que a execução propriamente dita do programa, inicia na **linha 13**. Já na linha 15, temos a chamada da função **soma()**. Nesse momento, o fluxo da execução é desviado para a **linha 2**. Em seguida, as **linhas 3 até 11** são executadas uma após a outra. Quando a execução chega à **linha 11**, encontra a marca final da função **}**, após encerra-se sua execução. Na sequência, a execução do programa retorna à **linha 15** e encerra o programa.

Deve-se destacar que, no momento em que a função **soma()** foi chamada, na **linha 15**, nenhum valor ou variável foi colocado entre parênteses, indicando que não houve passagem de parâmetros. Ela não retornou valor para quem a chamou. Por essas razões, seu tipo é **void**, já que a **soma()** é uma função que não tem passagem de parâmetros, nem retorno.

Figura 4 – Corpo do programa.

```
int main ()
{
    soma ();
}
```

Fonte: Elaborada pelo autor (2015).

### 7.2.1 O Comando Return ()

Vale ressaltar que o comando **return ()** tem dois objetivos:

Primeiro, ele tem a função de provocar uma saída imediata da função que o contém. Isto é, faz com que a execução do programa retorne ao código chamador; segundo, ele pode ser usado para devolver um valor para a função chamadora. (SCHILDT, 1990, p. 150).

Existem duas maneiras pelas quais uma função termina a execução e retorna ao código principal que fez a chamada.

- × A **primeira** ocorre quando o último comando da função for executado, ou seja, a chave “**}**” é encontrada. Quando isso acontece, a função termina e a execução do programa retorna para o ponto de onde foi desviado.
- × A **segunda** é quando a execução encontra o comando **return ()**. Esse comando paralisa a execução do programa e o retorna ao ponto inicial. Este comando pode levar ou não um valor como retorno.

Lembre-se: o uso do comando **return ()** auxilia a manutenibilidade e entendimento do código, deixando-o mais organizado e legível para outros progra-

madores que forem trabalhar no mesmo código. Isso porque além de encerrar a execução da função, também é uma forma de documentar o código.

É importante salientar que todas as funções devem retornar um valor, exceto as funções do tipo void. Esse valor é especificado pelo comando `return ()`. Se nenhum comando `return ()` estiver presente, então o valor de retorno da função será indefinido. Geralmente os compiladores C devolvem o número 0 quando nenhum valor de retorno for especificado. (SCHILDT, 1990, p. 152).

Figura 5 – Exemplo de função com passagem de parâmetros e com retorno.

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

int multiplica(int N1, int N2)
{
    int resultado;
    resultado = N1 * N2;
    return(resultado); //retornando o valor
para main
}

int main(void)
{
    int V1, V2, resultado;
    printf("Digite o primeiro valor:");
    scanf("%d", &V1);
    printf("Digite o segundo valor:");
    scanf("%d", &V2);

    resultado = multiplica(V1,V2);
    printf("Resultado = %d\n", resultado);
    getch();
    return 0;
}
```

Fonte: Elaborada pelo autor (2015).

No exemplo da Figura 5, as variáveis V1 e V2 são lidas, nas linhas 15 e 17, e são passadas como parâmetro para a função **multiplica**, na linha 19. A função **multiplica**, nas linhas 4 a 9, executa a multiplicação de V1 por V2, na linha 7 e retorna o resultado desse cálculo, na linha 8, para a função **main**, na linha 19, que o mostra na tela, na linha 20.

## 7.3 Funções Sem Passagem de Parâmetros e Com Retorno

O terceiro tipo de função é representado por aquelas que não recebem parâmetros, mas que no final, devolvem um valor para quem as chamou (retorno). (ASCENCIO, 2012, p. 236-237).

Para entender melhor essa função, acompanhe a seguir o exemplo que calcula a **multiplicação de dois números**.

Figura 6 – Exemplo de função sem passagem de parâmetros e com retorno.

```
include <stdio.h>
int multiplicacao ()
{
    int multiplicando, multiplicador, produto;
    printf("\nDigite o valor do multiplicando : ");
    scanf("%d", &multiplicando);
    printf("\nDigite o valor do multiplicador : ");
    scanf("%d", &multiplicador);
    produto = multiplicando * multiplicador;
    return (produto);
}

int main ()
{
    int resposta;
    resposta = multiplicacao();
    printf("\n0 produto é: %d\n", resposta);
}
```

Fonte: Elaborada pelo autor (2015).

Como você já viu, todo programa em linguagem de programação C inicia pela função principal **int main ()**. Dessa forma, o código executado no exemplo começou na **linha 13**, na **linha 15** a variável **resposta** foi definida como um número **int** e, em seguida, na **linha 16** a função **multiplicação ()** foi chamada. Assim, a execução do programa é desviada para a **linha 2**, que executa os comandos sequencialmente, da **linha 3** até a **linha 11**.

Note que na **linha 10** temos algo novo, o comando **return (produto)**, que retorna o conteúdo da variável **produto** para o programa principal. No programa principal, o conteúdo da variável **produto** é armazenado na variável **resposta**. E na **linha 17**, o conteúdo de **resposta** é mostrado na tela.

## 7.4 Funções Com Passagem de Parâmetros e Sem Retorno

O segundo tipo de função é representado por aquelas que recebem valores no momento em que são chamadas (parâmetros), mas que, no final, não devolvem valor para quem as chamou (retorno). (ASCENCIO, 2012, p. 235-236).

Para entender melhor como funciona, confira a seguir o exemplo que calcula a **média entre dois números**.

Como visto anteriormente, todo programa em linguagem de programação C inicia pela função **void main ()**. Na Figura 7, a execução inicia na **linha 8** e, em seguida, são executadas, sequencialmente, da **linha 9** até

a **linha 15**. Observe que nas **linhas 13** e **15** dois números são digitados pelo usuário e armazenados nas variáveis **num1** e **num2**.

Já na **linha 16** a execução do programa é desviada para a função **media()**, e esses números são passados como parâmetros **media(num1, num2)**. Assim, a execução segue para a

**linha 2**, onde está definida a função **void media (int a, int b)**, que recebe como parâmetros os números **num1** e **num2**, os quais são atribuídos como parâmetros às variáveis inteiras **a** e **b**.

Dessa forma, a média é calculada na **linha 5** e mostrada na tela conforme mostrado na **linha 6**. Como na função **void media (int a, int b)** não foi usado o comando **return()**, isso significa que ela não retornou o valor para quem a chamou. Por essa razão, seu tipo definido foi o **void**.

Figura 7 – Exemplo de função com passagem de parâmetros e sem retorno.

```
#include <stdio.h>
float media (int a, int b)
{
    float media;
    media = (a + b)/2;
    printf("\nA média dos números é: %f\n", media);
}

int main ()
{
    int V1, V2;
    printf("\nDigite o primeiro número: ");
    scanf("%d", &V1);
    printf("\nDigite o segundo número: ");
    scanf("%d", &V2);
    media(V1, V2);
}
```

Fonte: Elaborada pelo autor (2015).

## 7.5 Funções Com Passagem de Parâmetros e Com Retorno

O quarto tipo de função é representado por aquelas que recebem valores no momento em que são chamadas (parâmetros), e que no final, devolvem um valor para quem as chamou (retorno). (ASCENCIO, 2012, p. 237).

Na prática, uma função com passagem de parâmetros e com retorno, como o cálculo da **divisão de dois números**, comporta-se assim:

Figura 8 – Exemplo de função com passagem de parâmetros e com retorno.

```
#include <stdio.h>
int divisao (int dividendo, int divisor)
{
    int q;
    q = dividendo / divisor;
    return (q);
}
int main ()
{
    int n1, n2;
    int resposta;
    printf("\nDigite o valor do dividendo : ");
    scanf("%d", &n1);
    printf("\nDigite o valor do divisor : ");
    scanf("%d", &n2);
    resposta = divisao (n1,n2);
    printf("\nO resultado da divisão é: %d\n", resposta);
}
```

Fonte: Elaborada pelo autor (2015).

Iniciando a execução do programa **na linha 8**, observe que as variáveis inteiras **n1** e **n2**, bem como a variável **resposta**, são declaradas. Em seguida, o usuário digita dois valores, sendo um para cada uma das variáveis inteiras. Já **na linha 16** a função **divisão (n1, n2)** é chamada e leva como parâmetros os conteúdos das variáveis **n1** e **n2**.

Quando a função **divisão ()** for executada, **na linha 2**, o conteúdo das variáveis **n1** e **n2** será passado para as variáveis **dividendo** e **divisor**, respectivamente. **Na linha 5**, o cálculo é realizado, e **na linha 6** a função **return (q)** volta para a função principal, com o conteúdo da divisão armazenado na variável **q**.

Dessa forma, na **linha 16**, esse conteúdo é atribuído à variável resposta e na **linha 17** esse valor é mostrado na tela. Por fim, a **linha 18** encerra a execução do programa.

## 7.6 Passagem de Parâmetros por Valor e Passagem de Parâmetros por Referência

Há duas formas de usarmos a passagem de parâmetros. Uma delas é a passagem de parâmetros por **valor**, e outra é a passagem de parâmetros por **referência**. Até o momento trabalhamos com passagem de parâmetros por valor.

Mas, a seguir, estudaremos o que é exatamente uma chamada **por valor** e uma chamada **por referência**. Prossiga!

### 7.6.1 Passagem de Parâmetros por Valor

A passagem de parâmetros por valor significa que a função trabalhará com cópias de valores passados no momento da sua chamada. (ASCENCIO, 2012, p. 238). Isso significa que, para a execução da função, serão geradas cópias dos valores de cada um dos parâmetros, e esses valores serão armazenados nas variáveis definidas como parâmetros.

Assim, é feito uma passagem do conteúdo da variável. Mas, para isso, o parâmetro deve, necessariamente, ser do mesmo tipo de dado da variável. Cada variável passada no momento da chamada será copiada para o parâmetro correspondente da função, na ordem da declaração.

Em outras palavras, uma chamada **por valor** copia o conteúdo de uma variável e o envia para a função chamada. Em uma chamada de parâmetro **por valor**, os tipos de dados que são enviados para a função chamada são todos os tipos de dados permitidos pela linguagem de programação C, como: **inteiro, float, double, char, string** (com exceção do tipo de dado ponteiro).

A passagem de parâmetros por valor é a forma mais utilizada para passagem de parâmetros usando funções em linguagem de programação estruturada C. Um exemplo clássico da matemática são as funções da Trigonometria. A função seno(), por exemplo, recebe o valor de um ângulo, que é



um número real, e devolve o seno desse ângulo; e a função cosseno, também recebe o valor de um ângulo.

Os valores que são passados como parâmetros para as funções seno e cosseno, que são números reais, representam a passagem de parâmetros por valor em uma função. Para resolver problemas da matemática ou da computação gráfica, por exemplo, as funções com passagem de parâmetros por valor, são muito utilizadas.

### 7.6.2 Passagem de Parâmetros por Referência

A passagem de parâmetros por referência significa que os parâmetros passados para uma função correspondem a endereços de memória ocupados por variáveis. Dessa maneira, toda vez que for necessário acessar um determinado valor, isso será feito por meio de referência ao seu endereço. (ASCÊNCIO, 2012, p. 239).

Isto é, para a execução da função, não será passado o conteúdo da variável, mas será passado por parâmetro o **endereço de memória** da variável. Nesse caso, o parâmetro é do **tipo ponteiro** (conceito que será tratado no Capítulo 11).

Em outras palavras, uma chamada **por referência** copia o endereço de memória de uma variável e o envia para a função chamada. Vale ressaltar que, em uma chamada de parâmetro **por referência**, o único tipo de dado enviado para a função chamada é o tipo de dado chamado de **ponteiro**. Exemplos de funções com passagem de parâmetros por referências estão detalhados no Capítulo 11.

## Resumindo

Neste capítulo, vimos o que é uma Função, e que esta também pode ser chamada de sub-rotina. Além disso, aprendemos o conceito de funções e procedimentos em programação estruturada com C.

Inclusive, ao longo do conteúdo, compreendemos como é possível inserir uma função em um programa estruturado em linguagem de programação C, bem como a utilizar essas funções a partir de exemplos.

Por fim, conferimos o que são funções com passagens de parâmetros por valor e funções com passagem de parâmetros por referência. Além disso, vimos como funciona o comando return em uma função.



# 8

## Utilizando pesquisa e ordenação

VOCÊ JÁ PAROU para pensar o quão importante é uma pesquisa e uma ordenação de dados para estruturar um programa em linguagem de programação C? Por isso, neste capítulo, vamos juntos compreender os conceitos básicos de **Pesquisa** e **Ordenação**, bem como conhecer os algoritmos de pesquisa e ordenação desses dados. Dentre eles:

- × uma ordenação simples, pelo método bolha (*bubble sort*);
- × uma ordenação básica, pelo método de seleção (*selection sort*);
- × uma pesquisa sequencial; e
- × uma pesquisa binária.

Lembrando que todos os procedimentos a serem realizados irão utilizar os comandos da programação estruturada em C.

Preparado para começar? Então, siga em frente e bons estudos!

## Objetivo de Aprendizagem

- × Compreender como se realiza buscas e ordenação em arquivos binários;
- × Saber aplicar pesquisa e ordenação em arquivos binários.

## 8.1 Pesquisa e Ordenação

A pesquisa e a ordenação de dados são tarefas essenciais para um programador de computadores. É a partir desses algoritmos que conseguimos encontrar um dado importante para tomada de decisão, ou para realizar um determinado cálculo. Por exemplo, os algoritmos, que fazem busca/pesquisa em sites na Web e que são chamados de *crawlers* ou *robots*, são programados para serem guiados através de textos e links na internet, bem como fazem um rastreamento para procurar (principalmente textos), para indexar, para processar, para classificar/ordenar e para estimar o conteúdo de um site específico. Um exemplo de algoritmo de busca/pesquisa é o Googlebot, que tem como objetivo principal o rastreio de novidades sobre toda e qualquer informação relevante, seguindo links de um site a outro e fazendo o mapeamento de todos os links encontrados.

---

O termo *Web Crawler* diz respeito a um programa de computador ou *software*, cujo objetivo é navegar de forma sistemática pela *web* e fazer uma varredura em busca de informações entendidas como relevantes para seu propósito.

O termo *Robot* indica programas utilizados pelos motores de busca com o objetivo de explorar de forma automática a internet e baixar conteúdos de determinados sites.

---

Mas qual a diferença entre pesquisa e ordenação?

Veja bem, a pesquisa por um dado pode ser feita de forma sequencial ou não, dependendo da estrutura que os dados estão armazenados. Essa estrutura dependerá da sua ordenação. Ou seja, a ordenação é responsável por auxiliar a pesquisa e, na maioria das vezes, encontrar os dados mais rapidamente.

Na Ciência da Computação, “essas rotinas são utilizadas em praticamente todos os programas de bancos de dados, compiladores, interpretadores e sistemas operacionais” (SCHILDT, 1990, p. 501).

Por isso, compreender a lógica dos algoritmos de pesquisa e ordenação é de suma importância para um programador de computadores. Afinal, o objetivo de ordenar os dados, geralmente, é facilitar e acelerar o processo de pesquisa.

Como a ordenação exerce uma importância muito grande para realizar pesquisas, vamos a seguir começar nosso aprendizado por essa rotina. Acompanhe-nos!

### 8.1.1 Ordenação

Segundo Schildt (1990, p. 501), a “ordenação é o processo de arranjar um conjunto de informações semelhantes numa ordem crescente ou decrescente”.

Em muitos casos, os dados devem ser armazenados obedecendo à determinada ordem, por exemplo, crescente de notas, decrescente de salários etc. Essa ordenação contribui com a execução dos algoritmos do ponto de vista do desempenho computacional, isto é, quanto mais ordenados os dados estiverem armazenados, mais ágil será a execução do código.

Mas o que precisamos fazer para obter os dados organizados em alguma ordem? Para isso, temos basicamente duas alternativas:

- 1) no momento da inclusão dos dados, o usuário já deve inseri-los respeitando a ordenação, e isso se chama ordenação garantida por construção; ou
- 2) a partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos.

Quando pensamos em um algoritmo de ordenação, precisamos ter em mente que eles precisam ser eficientes, tanto em termos de tempo (devem ser rápidos) quanto em termos de espaço (devem ocupar pouca memória durante a execução).

Vale lembrar que os algoritmos de ordenação podem ser aplicados a qualquer tipo de dado, desde que existam condições de determinar uma ordem definida entre os elementos. Portanto, podemos ordenar um vetor de números inteiros, adotando uma ordem crescente ou decrescente, ou podemos também aplicar algoritmos de ordenação em vetores que armazenam dados mais complexos, por exemplo, um vetor que guarda os dados relativos a alunos de uma turma, com nome, endereço, CEP, cidade, estado, número de matrícula, curso matriculado, entre outros.

Para facilitar o entendimento, a partir de agora, vamos usar como exemplo um algoritmo de ordenação, considerando os seguintes aspectos:

- × que a entrada de dados é um vetor, cujos elementos precisam ser ordenados;
- × que a saída de dados é o mesmo vetor, com seus elementos na ordem especificada; e
- × que o espaço que pode ser utilizado é o do próprio vetor.

Ou seja, vamos utilizar um dos métodos de ordenação mais comuns que é a **ordenação de vetores**, por meio de dois algoritmos de ordenação, conhecidos como **ordenação bolha** (*bubble sort*) e **ordenação básica** (*selection sort*).

### 8.1.2 Ordenação Bolha (*bubble sort*)

O algoritmo de “ordenação bolha” (*bubble sort*) recebeu esse nome pelo fato de que os elementos menores serão armazenados no início do vetor e os maiores vão sendo armazenados no fim do vetor. E na medida em que os valores vão tomando seu lugar no vetor, parece uma bolha subindo ou descendo dentro do próprio vetor. O seu objetivo é fazer uma série de comparações entre todos os elementos do vetor, verificando qual é o elemento maior entre dois elementos, no caso da ordem crescente. Assim, o elemento maior é trocado de posição em direção ao final do vetor, enquanto o elemento menor é trocado de posição em direção ao início do vetor. Isto é, quando dois elementos estão fora de ordem, há uma inversão, ou seja, esses dois elementos são trocados de posição, ficando na ordem desejada (crescente ou decrescente).

Dessa forma, o primeiro elemento é comparado com o segundo. Se o primeiro elemento for maior que o segundo, a troca de posição é realizada. Em seguida, independente se houve ou não troca após a primeira comparação, o segundo elemento é comparado com o terceiro, e, caso o segundo elemento seja maior que o terceiro, a troca de posição é feita.

O processo continua até que o penúltimo elemento seja comparado com o último. Com esse processo, garante-se que o elemento de maior valor do vetor será levado para a última posição e que os menores elementos permaneçam nas primeiras posições do vetor.

É importante frisar que este processo de troca de posições continua posicionando o segundo maior elemento, o terceiro e assim sucessivamente, até que todo o vetor esteja ordenado.

Para exemplificar, vamos considerar o vetor com valores inteiros e ordená-lo em ordem crescente (Figura 1).

Antes de continuarmos, é importante lembrar que as representações apresentadas nas figuras a seguir são vetores; um vetor possui um tamanho; e os dados são armazenados em posições do vetor, chamadas de índices.

Figura 1 – Vetor sem ordenação.

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).

Perceba que, na Figura 2, foi feita a comparação entre os elementos 25 e 48, para verificar se 25 é maior que 48. Como 25 não é maior que 48, então **não** houve troca de posição.

Figura 2 – Vetor sem troca de posição entre os elementos 25 e 48.

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).

A seguir, na Figura 3, a comparação foi realizada entre os elementos 48 e 37 para verificar se 48 é maior que 37. Como 48 é maior que 37, houve troca de posição.

Figura 3 – Vetor com troca de posição do elemento 48 pelo elemento 37.

25	48	37	12	57	86	33	92
25	37	48	12	57	86	33	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).

Na Figura 4, a comparação foi entre os elementos 48 e 12, para verificar se 48 é maior que 12. Como é possível analisar, 48 é maior que 12, então houve a troca de posição.

Figura 4 – Vetor com troca de posição do elemento 48 pelo elemento 12.

25	37	48	12	57	86	33	92
25	37	12	48	57	86	33	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).

Agora, na Figura 5, os elementos 48 e 57 foram comparados, para verificar se 48 é maior que 57. Como 48 não é maior que 57, então **não** houve troca de posição.

Figura 5 – Vetor sem a troca de posição entre os elementos 48 e 57.

25	37	12	48	57	86	33	92
25	37	12	48	57	86	33	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).



Na Figura 6, a comparação foi realizada entre os elementos 57 e 86, para verificar se 57 é maior que 86. Como 57 é menor que 86, logo **não** houve a troca de posição.

Figura 6 – Vetor sem a troca de posição entre os elementos 57 e 86.

25	37	12	48	57	86	33	92
25	37	12	48	57	86	33	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).

Já na Figura 7, a comparação entre os elementos 86 e 33 foi verificar se 86 é maior que 33. Como 86 é maior que 33, então foi realizada a troca de posição.

Figura 7 – Vetor com a troca de posição entre os elementos 86 e 33.

25	37	12	48	57	86	33	92
25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

índices do vetor

Fonte: Elaborada pelo autor (2015).

Por fim, na Figura 8, foi feita a comparação entre os elementos 86 e 92, para verificar se 86 é maior que 92. Como é possível ver, 86 é menor que 92, portanto, **não** teve troca de posição.

Figura 8 – Vetor sem a troca de posição entre os elementos 86 e 92.

25	37	12	48	57	33	86	92
25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	6

índices do vetor

Fonte: Elaborada pelo autor (2015).

Nesse momento, ao final da primeira passada por todo o vetor, o maior elemento, que é o 92, já está na sua posição final.

A partir de então, o algoritmo da ordenação bolha segue sucessivamente até que o 86 esteja na sua posição final, depois o 57, após o 48, em seguida os números 37, 33 e 25, por fim, o 12.

Ao final da ordenação, o vetor estará organizado em ordem crescente, como é possível visualizar a seguir:

Figura 9 – Vetor de números inteiros em ordem crescente.

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	6
índices do vetor							

Fonte: Elaborada pelo autor (2015).

Considerando que o vetor está preenchido com os valores da Figura 1, o algoritmo abaixo executa o método de ordenação bolha e ordena os valores em ordem crescente, como mostrados na Figura 9.

Figura 10 – Algoritmo que implementa a ordenação bolha.

```
#include <stdio.h>
int main ()
{
    inti, j, temp, vetor[7];
    vetor[0]=25;
    vetor[1]=48;
    vetor[2]=37;
    vetor[3]=12;
    vetor[4]=57;
    vetor[5]=86;
    vetor[6]=33;
    vetor[7]=92;

    for (i=7-1; i>=1; i--)
    {
        for (j=0; j<i; j++)

            if (vetor[j] > vetor[j+1])
            {
                temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
            }
    }
    for (i=0; i<8; i++)
        printf("vetor ordenado - Metodo Bolha: %d\n\n", vetor[i]);
}
```

Fonte: Elaborada pelo autor (2015).

Na Figura 10, o algoritmo que implementa a ordenação bolha apresenta na linha 4, a declaração das  $i$ ,  $j$ ,  $temp$  e  $vetor$  [7] e nas linhas 5 a 12. Assim, o vetor é preenchido, posição por posição, com um número constante.

Na linha 14, o comando de repetição *for* inicializa a variável  $i$  em  $(7-1)$ , testa a repetição enquanto o  $i$  for menor ou igual a 1 e decrementa o  $i$  em  $-1$ , a cada iteração. Esse comando *for* significa que o algoritmo vai repetir a execução do comando *for*, que está na linha 16, durante 6 vezes.

Na linha 16, o segundo comando de repetição *for* é inicializado com as variáveis de controle  $j=0$ , é repetido enquanto essa variável de controle for menor ou igual à variável  $i$  e incrementa a variável de controle  $j$  em  $+1$ .

Nas linhas 18 a 22, o algoritmo testa se o conteúdo que está na posição  $j$  é maior que o conteúdo que está na posição  $j+1$ , significando que o algoritmo testa se o conteúdo da posição seguinte ( $j+1$ ) à posição atual é maior ou não à posição atual ( $j$ ). Se o conteúdo da posição seguinte ( $j+1$ ) à posição atual é maior ( $j$ ) é **maior**, então o algoritmo faz a troca do conteúdo de posição dentro do vetor, como pode ser visto nas linhas 20, 21 e 22. O conteúdo da posição atual ( $j$ ) é atribuído à variável  $temp$ . O conteúdo da posição seguinte ( $j+1$ ) é atribuído à posição atual ( $j$ ), e da posição seguinte ( $j+1$ ) recebe o conteúdo da variável  $temp$ .

Dessa forma, o algoritmo de ordenação Bolha faz todas as trocas mostradas da Figura 1 até a Figura 9. Já a saída em tela do algoritmo pode ser vista na Figura 11, a seguir:

Figura 11 – Saída de tela do algoritmo que implementa a ordenação bolha.

```
Vetor ordenado - Método Bolha: 12
Vetor ordenado - Método Bolha: 25
Vetor ordenado - Método Bolha: 33
Vetor ordenado - Método Bolha: 37
Vetor ordenado - Método Bolha: 48
Vetor ordenado - Método Bolha: 57
Vetor ordenado - Método Bolha: 86
Vetor ordenado - Método Bolha: 92
```

Fonte: Elaborada pelo autor (2015).

### Saiba mais

O método de ordenação bolha (*bubble sort*) é o método mais simples e conhecido. Ele faz a ordenação por trocas, comparando o elemento da ordenação com o elemento “bolha” e assim sucessivamente, até o fim do vetor. O *bubble sort* faz a troca de valores entre posições consecutivas e depois ordena o vetor, sequencialmente. Ou seja, percorre o vetor e compara elemento a elemento. No entanto, esse método é considerado lento para um número grande de dados. Sendo assim, ele é mais recomendado para vetores com poucos elementos.

### 8.1.3 Ordenação básica (*selection sort*)

O objetivo da ordenação pela seleção é reorganizar um conjunto de dados a partir da mudança do menor valor para o início do vetor (na primeira posição), em seguida o segundo menor valor é transferido para a segunda posição do vetor, e assim é feito sucessivamente com todos os valores restante até que eles estejam devidamente ordenados. Por possuir baixa complexidade, pode facilmente ser utilizado em vetores pequenos e apresenta boa performance de execução/ordenação.

Na prática, este algoritmo procura o menor elemento do vetor e troca-o com o elemento localizado na primeira posição do vetor. Como o menor valor já está alocado na primeira posição do vetor, o algoritmo então procura o próximo menor valor entre os elementos restantes e faz o mesmo processo, armazenando-o neste caso na segunda posição do vetor. Essa rotina vai se repetindo até que o vetor esteja completamente ordenado.

Em termos de performance – pelo fato de as buscas por menores valores serem feitas apenas na parte “restante” do vetor, ou seja, apenas nos elementos que ainda não foram ordenados e mudados para as primeiras posições do vetor –, existe um ganho de velocidade na execução de algoritmos baseados na ordenação por seleção.

Para entendermos melhor o funcionamento da ordenação por seleção, analise a Figura 12, a qual apresenta como seria todo o processo de ordenação dos elementos de um vetor com cinco posições.

Figura 12 – Etapas da ordenação de um vetor pelo método de seleção.



Fonte: Elaborada pelo autor (2015).

Analisando a figura, é possível notar que foram necessárias três trocas para que o vetor fosse totalmente ordenado. Na primeira verificação, o elemento da terceira posição (número 1) foi trocado com o elemento da primeira posição (número 3). Na segunda verificação, houve troca entre os elementos da segunda posição (número 5) e da quarta posição (número 2). Na terceira verificação, nenhuma troca foi necessária, pois o número 3 já estava na terceira posição do vetor. Na quarta verificação foi realizada a última troca, a qual envolveu os elementos da quarta posição (número 5) e da quinta posição (número 4).

Agora iremos implementar o código em Linguagem C que realiza a ordenação pelo método de seleção (Figura 13).

Figura 13 – Algoritmo que implementa a ordenação por seleção.

```
include <stdio.h>

int main ()
{
    int i, j, min, temp, tamanho, vetor[5];

    //Preenche os valores iniciais do vetor
    vetor[0] = 3;
    vetor[1] = 5;
    vetor[2] = 1;
    vetor[3] = 2;
    vetor[4] = 4;

    tamanho = 5;

    //Realiza as verificações e trocas de elementos
    for (i = 0; i < (tamanho-1); i++)
    {
        min= i;
        for (j = (i+1); j < tamanho; j++) {
            if(vetor[j] < vetor[min])
                min = j;
        }

        if (i != min) {
            temp = vetor[i];
            vetor[i] = vetor[min];
            vetor[min] = temp;
        }
    }
    //Imprime o vetor ordenado
    for (i = 0; i <= (tamanho-1); i++)
        printf("Vetor ordenado - metodo de selecao: %d\n\n", vetor[i]);
}
```

Fonte: Elaborada pelo autor (2015).

No código podemos notar que foi declarado um vetor de inteiros com cinco posições, todas elas devidamente inicializadas com os mesmos valores apresentados anteriormente na Figura 12. Na linha 17 há um laço “for” para percorrer toda a extensão do vetor; ação similar ocorre na linha 20, porém o vetor é percorrido apenas a partir da posição seguinte à indicada pela variável auxiliar “i”. Na linha 21, acontece a verificação se um elemento é menor que o outro e, na linha seguinte, o armazenamento da posição deste menor elemento em uma variável auxiliar chamada “min”. Por último, o código apresentado nas linhas 26 a 28 realiza a troca de valores do vetor quando existe um elemento menor que o outro.

A execução deste programa resultaria na saída apresentada pela Figura 14:

Figura 14 – Saída gerada pelo programa de ordenação por seleção.

```
Vetor ordenado - método de selecao: 1
Vetor ordenado - método de selecao: 2
Vetor ordenado - método de selecao: 3
Vetor ordenado - método de selecao: 4
Vetor ordenado - método de selecao: 5
```

Fonte: Elaborada pelo autor (2015).

Este exemplo apresentado realizou a ordenação de um vetor pequeno, com apenas cinco posições. É interessante que você realize outros testes com vetores maiores, analisando principalmente a performance da ordenação dos elementos do vetor.



### Saiba mais

Outro método relativamente simples para ordenação dos dados é conhecido como ordenação por inserção (*insertion sort*). A ideia básica desse método é percorrer o vetor da esquerda para a direita e, conforme vai ocorrendo este avanço, os menores valores são alinhados à sua esquerda. Para conhecer mais a respeito desse método de ordenação, acesse os links a seguir:

[http://www.ime.usp.br/~pf/analise\\_de\\_algoritmos/aulas/insert.html](http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/insert.html)

<http://tiagomadeira.com/2006/01/ordenacao-por-insercao/>

<https://tatulab.com/index.php/programacao/c-c/32-c-ordenacao-por-insercao-insertion-sort>



## 8.2 Pesquisa

Segundo Herbert Schildt (1990, p. 522), os “bancos de dados existem para que de tempos em tempos, um usuário possa localizar o dado de um registro, simplesmente digitando sua chave de acesso”.

Assim, encontrar dados em um vetor desordenado requer uma pesquisa sequencial, começando no primeiro elemento e parando quando o elemento procurado, ou o final do vetor, for encontrado. Esse método deve

ser usado em dados desordenados, e também pode ser aplicado a dados ordenados (SCHILDT, 1990).

Quando o vetor está ordenado, é possível usar a **pesquisa binária**. Mas caso não estejam ordenados, é necessário usar a **pesquisa sequencial**. Assim, como os algoritmos de ordenação, os algoritmos de pesquisa são muitos e variados. Porém, é possível destacar os dois principais algoritmos: o de **Busca/Pesquisa Sequencial** e o de **Busca/Pesquisa Binária**, os quais vamos conhecer na sequência. Prossiga!

### 8.2.1 Pesquisa Sequencial

Você sabia que a pesquisa sequencial é fácil de ser codificada? Isso porque o algoritmo que implementa a busca sequencial executa a pesquisa de um elemento em um vetor, comparando-o aos elementos do vetor, um após o outro, até obter o resultado pesquisado, ou até chegar ao fim do vetor.

Porém, vale ressaltar que esse tipo de busca só é viável se o vetor for **pequeno e/ou não estiver ordenado**. Afinal, devido ao seu modo de operação, a pesquisa sequencial consome muito tempo de execução.

O exemplo a seguir, extraído da obra de Schildt (1990), mostra o algoritmo de pesquisa sequencial em um vetor de caracteres de tamanho conhecido, a partir de uma chave específica, para encontrar o elemento procurado. Confira:

Figura 15 – Algoritmo de pesquisa sequencial.

```
pesquisa_sequencial (char *item, int, count, char key
{
    register int t;
    for (t = 0; t < count; t++)
        if (key == item[t]) return t;
    return -1;
}
```

Fonte: (Schildt, 1990).

O exemplo acima, na linha 5, retorna o índice da entrada encontrada, se existir; caso contrário, na linha 6, devolve -1.

É importante salientar que a pesquisa sequencial é normalmente lenta; se os dados estiverem armazenados em disco, o tempo de procurar pode ser muito longo. Por outro lado, se os dados estiverem desordenados, a pesquisa sequencial é o único método possível.



### Importante

A pesquisa sequencial utiliza o método mais básico de pesquisa de dados, isto é, pesquisa todos os dados, um a um, sequencialmente, até encontrar o dado pesquisado. Esse tipo de pesquisa é útil caso o vetor esteja desordenado. Mas se o vetor estiver ordenado, o desempenho desse tipo de pesquisa é muito reduzido.

## 8.2.2 Pesquisa Binária

Primeiramente, é importante lembrar que a pesquisa binária só deve ser executada em vetores previamente ordenados, seja de forma crescente ou decrescente. Isso porque ela divide por dois o vetor analisado e compara o valor.

Se o valor central for maior que o objeto da pesquisa, o algoritmo divide novamente a lista em dois, dessa vez considerando apenas a parte central e o início da lista. Caso o valor central seja menor, a nova divisão será feita entre a parte central e o final da lista. Lembrando que o vetor estará ordenado em ordem crescente.

Em seguida, o algoritmo compara novamente o objeto da pesquisa com o valor apresentado e continua dividindo o vetor, até encontrar o valor desejado positivo, ou até que não seja mais possível realizar a divisão do vetor. Se isto ocorrer, é porque o valor não foi encontrado e o algoritmo devolve esse resultado.

Note que essa pesquisa é muito rápida, aliás, é a mais adequada para uso com vetores ou matrizes muito grandes.

Por exemplo, para encontrar o **número 14** no vetor a seguir, a pesquisa binária primeiro testa o elemento médio, nesse caso o **15**.

Figura 16 – Vetor ordenado.

11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	

índices do vetor

Fonte: (SCHILDT, 1990, p. 524).

Visto que ele é maior que **14**, a pesquisa continua com a primeira metade, ou seja:

Figura 17 – Vetor ordenado, 1ª parte.

11	12	13	14	15
0	1	2	3	4

índices do vetor

Fonte: (SCHILDT, 1990, p. 524).

Perceba que o elemento central agora é o **13**, que é menor que **14**. Então, a primeira metade é descartada, e a pesquisa continua com:

Figura 18 – Vetor ordenado, 2ª parte.

14	15
0	1

índices do vetor

Fonte: (SCHILDT, 1990, p. 524).

Veja que é nesse momento (Figura 18) que o elemento é encontrado.

A seguir, a Figura 19, mostra um algoritmo que implementa a Pesquisa Binária em um vetor de inteiros:

Figura 19 – Pesquisa binária em um vetor de inteiros

```
int pesquisaBinaria (int x, int n, int vetor[])
{
    int esquerda, meio, direita;

    esquerda = 0; direita = n-1;

    while (esquerda <= direita)
    {
        meio = (esquerda + direita)/2;
        if (vetor[meio] == x) return meio;
        if (vetor[meio] < x) esquerda = meio + 1;
        else direita = meio - 1;
    }
    return -1;
}
```

Fonte: Elaborado pela autora (2015).

A função `int pesquisaBinaria (int x, int n, int vetor[])` recebe um inteiro  $x$  e um vetor, também de inteiros.

Na linha 3, as variáveis do tipo inteiro *esquerda*, *meio* e *direita* são declaradas. Na linha 5, a variável *esquerda* é inicializada com o número 0 (zero) e a variável *direita* é inicializada com  $(n-1)$ .

Na linha 7, o comando de repetição *while* executa os comandos das linhas 9 até 12, enquanto a variável *esquerda* for menor ou igual à variável *direita*. Se o teste da linha 7 for verdadeiro, é atribuído à variável *meio*, ou seja, a média aritmética entre as variáveis *esquerda* e *direita*. Se o conteúdo do vetor[meio] for igual ao inteiro  $x$  então o *if* retorna o conteúdo da variável *meio*. Se o vetor[meio] for menor que o inteiro  $x$  então a variável *esquerda* recebe o conteúdo da variável *meio* acrescido de 1 ( $\text{meio} + 1$ ). Se o teste da linha 11 resultar falso, é atribuído o conteúdo da variável *meio* decrescido de 1 ( $\text{meio}-1$ ) à variável *direita*.

Os nomes das variáveis *esquerda*, *direita* e *meio* representam, exatamente, as áreas do vetor (por exemplo, o meio do vetor, é realmente, a metade do vetor). Como as variáveis *esquerda* e *direita* são variáveis do tipo inteiro, o resultado da divisão por 2 na expressão  $(\text{esquerda} + \text{direita})/2$  é considerado apenas a parte inteira do resultado da divisão. Isto quer dizer que se o resultado da divisão for um número com casas decimais, será considerado apenas a parte inteira do número.

Relembrando, a Pesquisa Binária é muito mais rápida que a Pesquisa Sequencial. No entanto, é muito importante que o vetor esteja, previamente, ordenado. Afinal, a Pesquisa Binária é muito parecida com o método de pesquisa que usamos para encontrar um nome em uma lista telefônica, por exemplo.



## Importante

A pesquisa binária é o método mais avançado de busca, e somente ocorre de forma eficiente se os dados estiverem devidamente ordenados. A ideia do algoritmo é testar o elemento que buscamos com o valor do elemento armazenado no **meio do vetor**. Se o elemento buscado for menor que o elemento do meio, sabemos que, caso o elemento esteja presente no vetor, ele estará na primeira parte do

vetor; agora se for maior, estará na segunda parte do vetor; e se for igual, achamos o elemento no vetor. Assim, ao concluirmos que o elemento está numa das partes do vetor, repetimos o procedimento, considerando apenas a parte que restou, e comparamos o elemento que buscamos com o elemento armazenado no meio dessa parte. Esse procedimento é continuamente repetido, subdividindo a parte de interesse até encontrarmos o elemento, ou chegarmos a uma parte do vetor com tamanho zero.



## Resumindo

Neste capítulo, vimos os conceitos de **Pesquisa** e **Ordenação** em programação estruturada com C. Entendemos, inclusive, o quanto é importante a ordenação dos dados para a otimização das pesquisas em programação estruturada.

Assim, conhecemos dois tipos de algoritmos: o algoritmo de ordenação simples pelo método bolha (*bubble sort*) e o algoritmo de ordenação básica (*selection sort*).

Por fim, aprendemos como realizar dois tipos de pesquisa: a pesquisa sequencial e a pesquisa binária, ambas utilizando os comandos da programação estruturada em C.

# 9

## Estruturas de Dados Heterogêneos

VOCÊ SABIA QUE podemos armazenar dados de vários tipos de dado em uma única estrutura para melhor acessá-los e fazer isso com recursos da linguagem de programação estruturada C? Esse tipo de estrutura em linguagem C chama-se **Estrutura de Dados Heterogêneos**, que foi nomeada de **struct** ou **registro**.

Para entendermos como armazenar dados na forma de um banco de dados, vamos estudar:

- × o que é um registro;
- × o que é um campo em um registro;
- × o que é um arquivo;
- × como gravar, ler e acessar dados em um arquivo.

Preparado para começar? Então siga em frente e bons estudos!

## Objetivo de Aprendizagem

- × Utilizar, aplicar e construir arquivos para leitura e gravação de dados.

### 9.1 O Que São Estruturas?

A linguagem de programação C permite criar tipos de dados definíveis pelo usuário de cinco formas diferentes. A primeira delas será estudada neste capítulo, a qual é chamada de **estrutura**. “Uma estrutura é um agrupamento de variáveis em uma variável apenas, sob um mesmo nome.” (SCHILDT, 1990, p. 167).

Ela funciona como uma espécie de “ficha”, a qual armazena diversos dados relacionados, porém de tipos diferentes. Ou seja, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dado.

#### Importante

As variáveis internas que compõem uma estrutura são chamadas de **membros da estrutura**. Por isso, uma estrutura pode ser chamada de **registro**, e os membros da estrutura podem ser chamados de **campos do registro**.

Cada informação contida em um **registro** é chamada de **campo**. E cada campo pode conter diferentes tipos primitivos de dados, como um inteiro ou real, ainda pode representar outras estruturas. É por isso que os registros (estruturas) são conhecidos como **dados heterogêneos** (que são tipos de dados diferentes).

Afinal, os programadores podem gerar novos tipos de dados, não se limitando apenas à utilização dos tipos de dados primitivos, fornecidos pelas linguagens de programação.

Para entender melhor como isso funciona, confira a seguir a sintaxe de um **registro** (estrutura), que na linguagem de programação C é chamado de **struct**.

Figura 1 – Sintaxe de uma struct.

```

struct <identificador>
    <listagem dos tipos e membros>;
}

struct <identificador> <variavel>;

```

Fonte: Elaborada pelo autor (2015).

A Figura 1 mostra a sintaxe para criar uma variável do tipo *struct* em linguagem de programação C, onde *struct <identificador>* significa que será necessário nomear a *struct* assim como toda e qualquer variável em C. A *<listagem dos tipos e membros>* significa a lista dos nomes das variáveis que vão compor a *struct* e os seus respectivos tipos de dados, e *struct <identificador> <variável>* significa que estamos definindo que uma variável será declarada como do tipo *struct*, conforme mostrado na Figura 2, abaixo.

## 9.2 Comandos de manipulação de dados

Os comandos de manipulação de dados são muito importantes em linguagem de programação porque é a partir deles que conseguimos salvar, ler, modificar e consultar o conteúdo das variáveis. A seguir, conheceremos as funções para manipulação de dados em linguagem de programação C.

### 9.2.1 Declaração da Variável Estrutura

A declaração de variáveis do tipo registros em C é feita a partir do comando **struct**. É importante ressaltar que, para um programa utilizar uma *struct*, é necessária a declaração de variáveis desse tipo, da seguinte forma:

```
Nome_da_estrutura    nome_da_variável;
```

Considerando que as estruturas representam novos tipos de dados, todas as operações e declarações realizadas com os tipos predefinidos da linguagem também poderão ser realizadas com as estruturas. Dessa maneira,

além de variáveis simples, vetores e matrizes também podem ser declaradas como **struct**.

No exemplo a seguir, a variável **aluno** é do tipo **ficha de aluno**. Ou seja, poderá armazenar valores diferentes, tais como: nome do aluno, endereço, CEP, cidade, estado e curso matriculado.

Lembrando que a variável do tipo **estrutura (struct)** pode conter variáveis simples e variáveis mais complexas, como vetores e matrizes. Confira a seguir um exemplo de declaração da **struct** em linguagem de programação C.

Figura 2 – Exemplo de declaração de uma variável do tipo struct.

---

```
struct ficha_de_aluno
{
    char nome[50];
    char disciplina[30];
    float nota_prova1;
    float nota_prova2;
};

    struct ficha_de_aluno aluno;
```

---

Fonte: Elaborada pelo autor (2015).

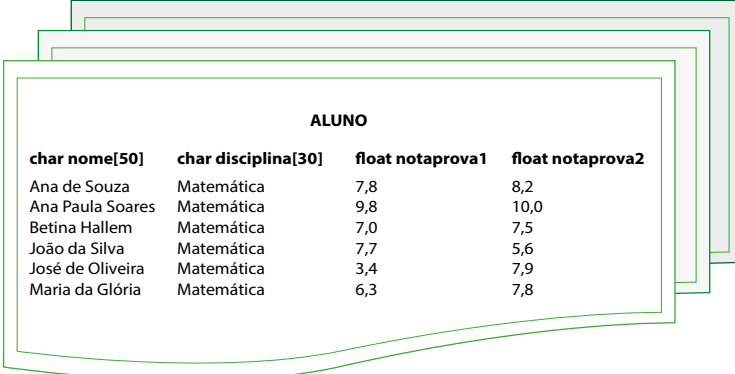
No último exemplo, percebe-se que foi criada uma estrutura **ficha\_de\_aluno**, com as variáveis **char nome[50]**, **char disciplina[30]**, **float nota\_prova1** e **float nota\_prova2**. Após a criação da estrutura, veja que foi preciso criar também a variável que vai utilizá-la. Por isso, foi colocada a variável **aluno**, que será do tipo **ficha\_de\_aluno**: `struct ficha_de_aluno aluno`.

Em outras palavras, significa que o programa fará referência à variável **aluno** quando precisar fazer acesso aos campos que compõem esta *struct*. A partir da nova estrutura definida, o programa considera que existe um novo tipo de dado a ser utilizado chamado **aluno**. Esse novo tipo de dado é capaz de armazenar várias informações, cujos tipos de dados podem ser diferentes.

Vale lembrar que as variáveis **char nome[50]**, **char disciplina[30]**, **float nota\_prova1** e **float nota\_prova2** são chamadas de **campos do registro**.



Figura 3 – Ilustração de uma variável do tipo struct.



ALUNO			
char nome[50]	char disciplina[30]	float notaprova1	float notaprova2
Ana de Souza	Matemática	7,8	8,2
Ana Paula Soares	Matemática	9,8	10,0
Betina Hallem	Matemática	7,0	7,5
João da Silva	Matemática	7,7	5,6
José de Oliveira	Matemática	3,4	7,9
Maria da Glória	Matemática	6,3	7,8

Fonte: Elaborada pelo autor (2015).

## Importante

○ registro somente poderá ser utilizado dentro do bloco onde foi definido.

A Figura 4, a seguir, mostra um exemplo completo da utilização da estrutura (registro) **ficha de aluno**.

Figura 4 – Exemplo da utilização da estrutura (registro).

```
#include <stdio.h>
#include <conio.h>
int main (void)
{
    struct ficha_de_aluno
    {
        char nome[50];
        char disciplina [30];
        float nota_prova1;
        float nota_prova2;
    } ficha_de_aluno;
    struct ficha_de_aluno aluno;
```

```
printf("\n ----- Cadastro de Aluno ----- \n\n\n");
printf("Nome do Aluno:");
scanf ("%s\n", &aluno.nome);

printf("Disciplina: ");
scanf ("%s\n", &aluno.disciplina);

printf("Nota da Prova 1: ");
scanf ("%2f\n", &aluno.nota_prova1);

printf("Nota da Prova 2: ");
scanf ("%2f\n", &aluno.nota_prova2);

printf ("\n\n ----- Lendo dados da struct----- \n\n");

printf ("Nome do Aluno: %s \n\n", aluno.nome);
printf ("Disciplina: %s\n\n", aluno.disciplina);
printf ("Nota da Prova 1: %f\n\n", aluno.nota_prova1);
printf ("Nota da Prova 2: %f\n\n", aluno.nota_prova2);
getch();
}
```

Fonte: Elaborada pelo autor (2015).

O programa da Figura 4 faz o registro de um aluno com nome, disciplina e nota das provas 1 e 2. Da linha 5 até a linha 11, a *struct* é definida com as variáveis nome, disciplina e notas da prova 1 e notas da prova 2. Na linha 12, a variável **aluno** é definida como sendo uma variável do tipo *struct ficha\_de\_aluno*. Da linha 14 até a linha 27, é feita a leitura dos dados, e da linha 31 até a linha 34, os dados lidos são mostrados na tela.

Observe que o programa inicia, nas linhas 1 e 2 com a inserção dos `#includes <stdio.h>` e `<conio.h>`.

### 9.2.2 Acesso e Manipulação da Variável Estrutura

Você sabia que, para acessar uma variável do tipo **estrutura (struct)** e manipular o seu conteúdo, é necessário informar o nome da variável e o do campo desejado? Mas ao informá-las, não se esqueça de separá-las por um ponto. Por exemplo:

```
aluno.nome[50]
```

```
aluno.disciplina[30];
aluno.nota_prova1;
aluno.nota_prova2;
```

Para fazer atribuições de valores a uma variável do tipo **estrutura (struct)**, também é necessário informar o nome da variável e o do campo desejado, separado por um ponto, com a atribuição do dado, conforme mostrado a seguir:

```
aluno.nome[50] = "João da Silva"
aluno.disciplina[30] = "Programação de
Computadores"
aluno.nota_prova1 = 8,7
aluno.nota_prova2 = 5,6
```

Nesse exemplo, veja que a estrutura **aluno** contém os dados sobre o acadêmico **João da Silva**, que cursa a disciplina de **Programação de Computadores** e que tem duas notas: a nota da prova 1, com o valor de **8,7**; e a nota da prova 2, com o valor de **5,6**.

Após compreender como acessar e manipular a variável estrutura, siga em frente e conheça as principais funções da manipulação de arquivos na linguagem de programação C.

## 9.3 Arquivos de Textos e Arquivos Binários

Os dados que utilizamos nos programas de computador, muitas vezes, precisam ser armazenados em arquivos para serem lidos posteriormente. É através do processo de digitação que os usuários informam os dados para o programa e esse processo é chamado de entrada de dados. Para isso, faz-se necessário conhecer o que a linguagem de programação C disponibiliza de funções para criação e manipulação desses dados em arquivos.

Antes de conhecer as funções essenciais para manipular **arquivos de textos e binários**, é preciso saber que o processo de trabalhar com eles, ao utilizar a linguagem de programação C, consiste em três etapas:

- × Abrir um arquivo;
- × Ler e/ou gravar dados em um arquivo;
- × Fechar o arquivo.

### Importante

○ Existem diversas formas de salvar (e recuperar) informações em arquivos com o uso de funções que salvam (e recuperam) as informações armazenadas nas **estruturas** de dados (**struct**).

Vale frisar que os arquivos são identificados por seu **nome, pasta e unidade de disco**. O nome dos arquivos é, em geral, composto pelo nome em si, seguido de uma extensão. Essa extensão identifica a natureza da informação armazenada no arquivo. Por exemplo, a extensão **“.c”** é usada para identificar arquivos que têm códigos fontes escritos na linguagem de programação C, enquanto que a extensão **“.doc”** é usada para identificar arquivos salvos no editor de textos Word da Microsoft.

Como você já viu, na maioria dos sistemas operacionais, um arquivo pode ser visto de duas maneiras:

- × em **modo texto**, como um texto composto de uma sequência de caracteres; ou
- × em **modo binário**, como uma sequência de bits (0 e 1), ou números binários.

Ou seja, podemos optar por salvar (e recuperar) informações em disco, usando um dos dois modos: **texto** ou **binário**.

Uma vantagem do arquivo texto é que pode ser lido por um ser humano e editado com editores de textos convencionais. Em contrapartida, com o uso de um arquivo binário, é possível salvar (e recuperar) grandes quantidades de informação de forma bastante eficiente.

Aliás, o sistema operacional pode tratar arquivos **texto** de maneira diferente da utilizada para tratar arquivos **binários**. Assim, dependendo da situação, é mais adequado utilizar arquivos do tipo **texto**, porém, em outros casos, é melhor utilizar arquivos do tipo **binário**. Nessa hora, é de suma importância que o programador tenha em mente o que pretende fazer com os dados e como pretende acessá-los.

Para minimizar a dificuldade com que arquivos são manipulados, os sistemas operacionais oferecem um conjunto de serviços para ler e escrever informações no disco. A linguagem de programação C é uma delas, já que possui diversas funções para manipulação de arquivos.

A seguir, uma relação das principais funções que tratam da manipulação de arquivos na linguagem de programação C:

- × **Abrir um arquivo:** o arquivo é encontrado no disco e aberto na memória para manipulação dos dados.
- × **Ler um arquivo:** é feito uma varredura no arquivo aberto para manipulação dos dados.
- × **Gravar em um arquivo:** dados são acrescentados no arquivo que foi, previamente, aberto.
- × **Fechar um arquivo:** a área da memória utilizada com esses dados é totalmente liberada.

Uma informação importante, mantida pelo sistema operacional, é a do **ponteiro de arquivo** (*file pointer*). Essa informação indica a **posição de trabalho no arquivo**. Isto é, para ler um arquivo, o apontador percorre todo o arquivo, do início até o fim. E, para gravar dados em um arquivo, estes são acrescentados apenas quando o apontador estiver posicionado no fim do arquivo.

Para entender melhor sobre leitura e gravação de dados em um arquivo de linguagem C, acompanhe-nos no próximo tópico!

## 9.4 Leitura e gravação de dados em arquivos

Primeiramente, você precisa saber que as funções utilizadas para acessar os dados em arquivos estão definidas na biblioteca padrão da linguagem de

programação C, chamada de **biblioteca de entrada e saída: stdio.h**. Algumas das principais funções para manipulação de arquivos dessa biblioteca são:

Figura 5 – Funções definidas na biblioteca stdio.h.

```
fopen() - Abre um arquivo  
fclose() - Fecha um arquivo  
fputc() ou putc () - Escreve um caracter em um  
arquivo  
fgetc() ou getc ()- Lê um caracter de um arquivo  
fputs() ou puts () - Escreve uma string em um  
arquivo  
fgets() ou gets () - Lê uma linha de um arquivo  
fprintf() - Equivalente à printf()  
fscanf() - Equivalente à scanf()  
rewind() - Posiciona o arquivo no início  
feof()- Retorna verdadeiro se chegou ao fim do  
arquivo  
remove () - Exclui um arquivo  
ferror () - Retorna verdadeiro se ocorreu um erro
```

Fonte: Elaborada pelo autor (2015).

Antes de qualquer operação ser executada com o arquivo, ele deve ser **aberto**. Isso porque essa operação associa um fluxo de dados a um arquivo.

E lembre-se: um arquivo pode ser aberto para diversas atividades. É importante salientar que a declaração correta da forma de criação e de abertura de um arquivo, garante a segurança dos dados gravados no respectivo arquivo.

#### 9.4.1 Abrir e fechar um arquivo no modo texto e no modo binário

A função básica para abrir um arquivo é **fopen**:

```
FILE* fopen (char* nome_arquivo, char* modo)
```

**FILE** é um tipo definido pela biblioteca padrão que representa uma abstração do arquivo. Quando abrimos um arquivo, essa função tem como

valor de retorno um ponteiro para o tipo **FILE**, bem como as operações relacionadas à ação de abrir arquivos. Se o arquivo não puder ser aberto, a função tem como retorno o valor **NULL**.

Como você já viu, existem diferentes modos de abertura de um arquivo. Podemos abri-lo para leitura ou para escrita, e devemos especificar se o arquivo será aberto em **modo texto** ou em **modo binário**.

O parâmetro **modo** da função **fopen** é uma cadeia de caracteres, a qual se espera a ocorrência de caracteres que identificam o modo de abertura. Os caracteres interpretados no modo são:

Quadro 1 – Parâmetros da função `fopen()`.

Modo de Abertura e Leitura	Ação	Significado
<i>r</i>	<i>ready-only</i>	Abre um arquivo somente para leitura
<i>a</i>	<i>append</i>	Grava dados no final de um arquivo
<i>t</i>	<i>text</i>	Grava dados em um arquivo do tipo texto
<i>b</i>	<i>binary</i>	Grava dados em um arquivo do tipo binário
<i>w</i>	<i>write</i>	Grava dados em um arquivo

Fonte: Elaborado pelo autor (2015).

Quando solicitamos a abertura de um arquivo no modo **r**, isso quer dizer que apenas a função de leitura será realizada perante os dados armazenados. Agora quando solicitamos a abertura de um arquivo no modo **a**, este é preservado e todos os novos dados são gravados no fim do arquivo, sem sobrepor os dados que já estavam ali armazenados.

Mas quando solicitamos a abertura de um arquivo no modo **b** ou **t**, estamos indicando que os dados encontrados nesses arquivos são, respectivamente, do **tipo binário ou texto**. Já quando solicitamos a abertura de um arquivo no modo **w**, o arquivo é totalmente apagado, e um novo arquivo vazio é criado no lugar.

A seguir, veja um exemplo de como abrir um arquivo na linguagem de programação C, no modo **gravar dados (w)**:

Figura 6 – Exemplo de como criar e abrir um arquivo.

```
#include <stdio.h>
int main (void)
{
    FILE *arquivo;
    arquivo = fopen ("teste1.txt", "w");

    if (arquivo==0)
        printf ("Erro na leitura do arquivo TESTE 1 \n");
    else
        printf ("Arquivo TESTE 1 aberto com sucesso \n");

    fclose (arquivo);

    FILE *f;
    if ((f=fopen("teste2.txt", "w")) == NULL)
    {
        printf ("Arquivo TESTE 2 não pode ser aberto \n\n\n");
    }
    else
        printf ("Arquivo TESTE 2 aberto com sucesso \n\n\n");

    remove (f);
}
```

Fonte: Elaborada pelo autor (2015).

Para começar a trabalhar com arquivos, primeiramente, você precisa inserir a biblioteca (linha 1), que define as funções de manipulação de arquivos:

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

Para ler ou escrever arquivos, no entanto, é preciso usar ponteiros de arquivos. Por isso, na linha 4 um ponteiro de arquivo deve ser inicializado, como uma variável do tipo **FILE**:

```
FILE *arquivo
```

Quando for abrir um arquivo, é imprescindível testar se esse comando ocorreu com sucesso. Para fazer essa verificação, é necessário realizar uma operação de teste da abertura de arquivos. Na linha 5, a função `fopen()` abre o arquivo **teste1.txt**, no modo “w”, e o resultado dessa abertura de arquivo



é armazenado na variável **arquivo**. Na linha 6, é feito o teste de abertura do arquivo; se a variável **arquivo** é igual a zero, então o arquivo não foi aberto com sucesso. Se a variável **arquivo** é diferente de zero, então o arquivo foi aberto com sucesso, e as respectivas mensagens são mostradas na tela. Na linha 11, o arquivo é fechado com a função `fclose()`.

Vale lembrar que, após **abrir, ler e escrever** os dados de um arquivo, você deve fechá-lo. Para fechar um arquivo, é necessário usar a função **fclose**, que espera como parâmetro o **ponteiro do arquivo** o qual se deseja fechar. Afinal, a função **fclose** fecha o arquivo que foi aberto por uma chamada à função **fopen**, após isso grava os dados que ainda permanecem na memória do computador e, por fim, fecha normalmente o arquivo. Um exemplo dessa função é:

```
int fclose (FILE* fp)
```

O **fp** é o ponteiro de arquivo devolvido pela função **fopen**. Se o valor de retorno da função for **zero**, significa que o arquivo foi fechado com sucesso. Já se o retorno for **EOF** (*end of file*, definido pela biblioteca), significa que ocorreu um erro ao fechar o arquivo, e o mesmo **não foi fechado** com sucesso. Por exemplo:

```
Return 0;  
Return EOF;
```

### Importante

A função **fopen** gerencia a abertura de um arquivo, abrindo um arquivo que já existe, pode abri-lo de diversos modos, como: abrir um arquivo **somente para leitura**; abrir um arquivo para **leitura e gravação**; abrir um arquivo para **gravar dados do tipo texto**; abrir um arquivo para **gravar dados do tipo binário**, e abrir um arquivo **posicionando a execução do programa no final do arquivo**.

#### 9.4.2 Gravar dados em arquivo no modo texto

Para gravar dados do tipo texto em um arquivo com extensão **.TXT**, utilizaremos a função **fputc()**. Esta função grava dados do tipo texto em um arquivo que foi previamente aberto para gravação (**w**) por meio da função **fopen()**.

A sintaxe da função **fputc()** é:

```
int fputc (char nomePessoa, FILE* arquivo)
```

Onde o primeiro parâmetro da função **char nomePessoa** contém os dados caracteres a serem gravados no arquivo .TXT e o segundo parâmetro da função **FILE \*arquivo** é o nome da variável definida como **arquivo**. Atenção: o valor de retorno dessa função é o próprio caractere escrito ou **EOF** (*end of file*), se ocorrer um erro na escrita.

A Figura 7 mostra a abertura e criação de um arquivo chamado “**teste1.txt**” que, foi criado na linha 6, com o comando **FILE \*arquivo**, e depois na linha 10 foi aberto com a função **fopen()**, no modo **w**, para gravação.

Figura 7 – Exemplo de escrita em arquivo no modo texto.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *arquivo;
    char nomePessoa[100];
    int i=0;

    arquivo = fopen("teste1.txt","w");

    if(arquivo==0)
        printf("Erro na criação do arquivo\n");
    else
    {
        printf("Arquivo criado e aberto com sucesso\n");

        printf ("Este programa demonstra a criacao e gravacao de dados do tipo
        TEXTO em arquivo\n");
        printf("Digitar uma frase com ate 100 caracteres. Essa frase sera gravada no
        arquivo textel.TXT: \n");
        gets (nomePessoa);
        for (i=0; nomePessoa[i]!='\0';i++)
            fputc(nomePessoa[i], arquivo);
    }

    printf ("\nLocalize o arquivo na pasta em que o programa esta rodando e
    abra-o\n\n");
    fclose(arquivo);
}
```

Fonte: Elaborada pelo autor (2015).

Neste exemplo, na linha 20, a função **gets()** lê uma sequência de até 100 caracteres digitados no teclado, e na linha 22, a função **fputc()** grava os caracteres lidos, um a um, no arquivo **teste1.txt**.

Encerrado a gravação, na linha 26, a função **fclose (arquivo)**, fecha o arquivo **teste1.txt** que pode ser encontrado no disco rígido do computador, e o seu conteúdo pode ser visualizado com o Bloco de Notas do Windows.

### Importante

A função **gets()** lê dados digitados do teclado e armazena em uma variável do tipo **caractere**. A função **fputc()** grava estes dados em um arquivo texto chamado **teste1.txt**. Este arquivo precisa ter sido aberto, previamente, com a função **fopen()**, no modo **w**, para gravação de dados.

### 9.4.3 Ler dados em arquivo no modo texto

Para acessar dados em arquivos no modo texto, faz-se necessário a leitura do arquivo com a função **getc()** que lê caracteres de um arquivo que foi previamente aberto com a função **fopen()**, no modo leitura **r**.

### Importante

É obrigatório abrir, previamente, o arquivo que será lido com as funções de leitura de arquivos, como a **getc()**, por exemplo. Caso contrário, se o arquivo não estiver aberto, a leitura não será efetivada e retornará um erro de leitura do arquivo por ele estar fechado.

A função **getc()** captura os dados de um arquivo aberto para leitura e os armazena na memória do computador e tem a seguinte sintaxe:

```
int getc (FILE* arquivo)
```

Perceba que essa função retorna o caractere lido. Quando o fim do arquivo for alcançado, a constante **EOF (end of file)** é retornada. Na Figura 8, o comando **while** mostra na tela o conteúdo da variável caractere **nome-Pessoa**, enquanto na linha 16, a função **getc()** for diferente de EOF (fim do

arquivo) . Quando a condição for verdadeira, isto é, quando a função **getc ()** retornar o valor EOF, então significa que foi encontrado o fim do arquivo e o comando **while** é encerrado, conforme mostrado na linha 16.

Figura 8 – **Exemplo de leitura de caracteres em arquivo no modo texto.**

```
#include <stdio.h>
#include <stdlib.h>
int main()

FILE *arquivo;
char nomePessoa;
int i=0;

arquivo = fopen("testel.txt", "r");

if(!arquivo)
    printf("Erro na abertura do arquivo\n");
else
{
    while ((nomePessoa=getc(arquivo)) != EOF)
        printf ("%c", nomePessoa);
    fclose(arquivo);
    system("pause");
}
```

Fonte: Elaborada pelo autor (2015).

#### 9.4.4 Ler, gravar e remover dados em arquivo no modo binário

Os arquivos no **modo binário** são utilizados para gravar e ler dados diretamente da memória do computador. A memória é escrita copiando-se o conteúdo de cada byte da memória para o arquivo. Uma das grandes vantagens de se usar arquivos binários é que podemos gravar e ler uma grande quantidade de dados de forma eficiente. Para gravar nesse modo, usamos a função **fwrite**, da seguinte maneira:

```
int fwrite (void* p, int tam, int nelem, FILE* fp)
```

Perceba que o primeiro parâmetro dessa função representa o **endereço de memória**, que possui o conteúdo que se deseja salvar no arquivo binário.

Já o parâmetro **tam** indica o tamanho, em bytes, de cada elemento, enquanto que o parâmetro **nelem** indica o número de elementos. Por fim, passa-se o ponteiro do arquivo binário para o qual o conteúdo da memória será copiado.

A função para a leitura dos dados de arquivos binários é análoga. Assim, o conteúdo do disco é copiado para o endereço de memória passado como parâmetro. Para ler esses dados, usa-se a função **fread**. Por exemplo:

```
int fread (void* p, int tam, int nelem, FILE* fp)
```

Agora para excluir dados em arquivos no modo binário, você deve usar a função **remove**, como mostrado na Figura 9 a seguir:

```
int remove (char *nome_do_arquivo)
```

Figura 9 – Exemplo com a função para remover um arquivo.

```
#include <stdio.h>
int main (void)
{
    FILE *arquivo;
    arquivo = fopen ("testel.txt", "w");
    if (arquivo==0)
        printf ("Erro na leitura do arquivo TESTE 1 \n");
    else
        printf ("Arquivo TESTE 1 aberto com sucesso \n");

    fclose (arquivo);
    remove ("testel.txt");

    FILE *f;
    if ((f=fopen("teste2.txt", "w")) == NULL)
    {
        printf ("Arquivo TESTE 2 não pode ser aberto \n\n\n");
    }
    else
        printf ("Arquivo TESTE 2 aberto com sucesso \n\n\n");
    remove ("teste2.txt");
}
```

Fonte: Elaborada pelo autor (2015).

Nas linhas 12 e 24, a função **remove()** é chamada para excluir os arquivos que foram criados e salvos no disco rígido do computador.

## Resumindo

Neste capítulo, vimos o conceito de **Estrutura de Dados Heterogêneos** em programação estruturada com C.

Inclusive, compreendemos como utilizar, aplicar e construir arquivos de texto, bem como a abrir e fechar arquivos binários.

E, por fim, aprendemos como realizar a leitura e gravação de dados no modo texto e binário, conforme as regras de programação estruturada da linguagem C.

# 10

## Introdução à Programação Básica em Interface Gráfica

VOCÊ CONHECE quais são os principais comandos para a programação de uma interface gráfica? Se você conhece, sabe então utilizá-las em linguagem de programação estruturada com C?

Caso não saiba, não se preocupe! Este será o tema central deste capítulo. Afinal, é a interface gráfica que permite a interação entre o usuário e os dispositivos digitais, por meio do uso de elementos gráficos, tais como: ícones, janelas e outros elementos que facilitam o uso dos softwares ou aplicativos. E nas linguagens de programação, especialmente na linguagem de programação estruturada C, há

diversas bibliotecas com funções definidas para ajudar na implementação de diferentes tipos de interfaces.

Por isso, a partir de agora, iremos trabalhar com a programação básica usada em uma interface gráfica, levando em consideração a estrutura que estamos aprendendo: a **C**. Inclusive, veremos os principais conceitos e recursos necessários para, no final do capítulo, estarmos apto a implementar exemplos básicos de interfaces gráficas em linguagem de programação **C**.

Preparado para se aventurar nas funções gráficas em linguagem de programação **C**? Então, siga em frente e bom estudo!

## Objetivo de Aprendizagem

- × Conhecer os principais comandos para programação de interface gráfica;
- × Utilizar tais funções gráficas em linguagem de programação **C**.

## 10.1 A Interface Gráfica Gui (*Graphical User Interface*)

Na Ciência da Computação, **interface gráfica** é um conceito que explica a forma de interação entre o usuário e um programa de computador. Esse tipo de interação constitui-se por meio de uma tela ou representação gráfica e visual, com desenhos, imagens, ícones, objetos gráficos etc.

Vale lembrar que tal interface é o primeiro contato do usuário com o programa de um computador. Por exemplo, a interface gráfica do sistema operacional Windows é constituída por um conjunto de barras de ferramentas, atalhos, menus e botões de acesso rápido.

Até o momento, trabalhamos com programas na linguagem de programação **C**, nos quais as saídas do resultado das operações eram feitas em uma tela muito semelhante à do sistema operacional, que chamamos de modo texto. Só que, para trabalharmos no modo gráfico, usaremos as funções definidas na Biblioteca *Graphics.h*. Vamos aprender a utilizá-la? Então, continue em frente!



### 10.1.1 A Biblioteca *Graphics.h*

A biblioteca *Graphics.h*, como o próprio nome já diz, oferece diversas funções gráficas, já definidas na própria linguagem de programação C. Seu objetivo é facilitar o desenvolvimento de programas com recursos gráficos como telas, menus, gráficos, manipulações de imagens, entre outros.

Mas qual a vantagem de usar funções gráficas, você sabe? Uma delas é tornar o *software* com uma interface mais amigável ao usuário. Outra vantagem é permitir uma visualização dos dados de diferentes formas, por exemplo: gráficos e tabelas. Afinal, se não fossem essas funções, tal visualização não seria possível, certo?

Por isso, quando falamos em interface gráfica, precisamos ter em mente o conceito de “janela” ou “*window*”, que é o espaço na tela onde os resultados serão exibidos. Esse espaço, por sua vez, é definido pelo tamanho da “janela” que precisamos utilizar. Ou seja, podemos definir “janelas” de vários tamanhos conforme nossa necessidade.

Na linguagem de programação C, os tamanhos de tela são diferentes para o modo texto e para o modo gráfico. No modo texto, o menor elemento a ser exibido é um *caracter*, enquanto no modo gráfico é um *pixel*. Conclusão: as funções gráficas vão sempre se referir à visualização e exibição de *pixels*.

Porém, a resolução de tela das janelas gráficas depende da resolução do monitor que está sendo utilizado. Dessa forma, é importante detectarmos qual o tipo de monitor no qual o programa será executado. Em outras palavras, essa verificação será uma das primeiras funções a serem executadas em um programa no modo gráfico.

Mas lembre-se de que, no ambiente gráfico, a configuração do tamanho da tela é alterada de acordo com o modo de vídeo que foi detectado. Isso porque, em aplicações gráficas, a detecção do tipo de vídeo é muito importante. Portanto, uma das primeiras funções em programas gráficos é, também, a de detectar o tipo de vídeo utilizado, para depois, trabalhar com as outras funções gráficas.

#### **Importante**

Geralmente, o tipo de vídeo utilizado é o VGA que, no modo gráfico, tem as dimensões 640x480. Isto é, possui uma linha com 640 pixels e uma coluna de 480 pixels, respectivamente.

A Biblioteca *Graphics.h* oferece um conjunto de funções gráficas que contêm funções para desenhar *pixels*, linhas, retângulos, arcos, círculos e outras formas geométricas. Aliás, tais funções são chamadas de primitivas, porque desenhavam apenas formas básicas, ou seja, são estruturas e formas que resultam em outras formas gráficas.

Antes de iniciarmos a programação propriamente dita, vamos conversar sobre os modos de vídeo e sobre as funções gráficas? Então, acompanhe-nos a seguir!

### 10.1.2 Modos de Vídeo

Como sabemos, existem diversos tipos de vídeos disponíveis no mercado. Os mais comuns são o monocromático, o CGA (Adaptador Gráfico Colorido), o EGA (Adaptador Gráfico Estendido) e o VGA (Matriz Gráfica de Vídeo). Na Figura 1, Schildt (1999 p. 443) apresenta os diversos modos de tipo de vídeo, tanto para texto como para gráficos. Porém, é importante salientar que, em um modo gráfico, tanto texto quanto gráficos podem ser apresentados. Já em um modo texto, a apresentação será somente em texto.

Figura 1 – Modos de tela para os diversos adaptadores de vídeo.

Modo	Tipo	Dimensões gráficas	Dimensões de texto
0	Texto, b/p	n/d	40x25
1	Texto, 16 cores	n/d	40x25
2	Texto, b/p	n/d	80x25
3	Texto, 16 cores	n/d	80x25
4	Gráficos, 4 cores	320x200	40x25
5	Gráficos, 4 tons de cinza	320x200	40x25
6	Gráficos, 2 cores	640x200	80x25
7	Texto, b/p	n/d	80x25
8	Gráficos PCjr 16 cores (obsoleto)	160x00	20x25
8	Hercules Graphics, 2 cores	720x348	80x25
9	Gráficos PCjr 16 cores (obsoleto)	320x200	40x25
10	Reservado		
11	Reservado		
12	Reservado		
13	Gráficos, 16 cores	320x200	40x25
14	Gráficos, 16 cores	640x200	80x25
15	Gráficos, 2 cores	640x350	80x25
16	Gráficos, 16 cores	640x350	80x25
17	Gráficos, 2 cores	640x480	80x30
18	Gráficos, 16 cores	640x480	80x30
19	Gráficos, 256 cores	320x200	40x25

Fonte: SCHILDT (1999, p. 443).

## Importante

No modo gráfico, os pixels individuais são referenciados por suas coordenadas X e Y, sendo X o eixo horizontal. Em qualquer um dos modos gráficos, o canto superior esquerdo da tela é a posição 0,0 (SCHILDT, 1999, p. 444, grifo nosso).

### 10.1.3 Funções Gráficas

Para conhecer bem as funções gráficas definidas na Biblioteca *Graphics.h*, nada melhor que vê-las dispostas em uma tabela, não é verdade? Pensando nisso, observe em seguida a Tabela 1, na qual são apresentadas todas as funções da biblioteca para que possamos trabalhar no modo gráfico com a linguagem de programação C.

Tabela 1 – Funções gráficas da biblioteca *Graphics.h*.

Função	Descrição	Sintaxe
arc	Desenha um arco circular.	arc (int x, int y, int anguloini, int angfim, int raio);
bar	Desenha uma barra bidimensional.	bar (int esquerda, int topo, int direita, int base);
bar3d	Desenha uma barra tridimensional.	bar 3d (int esquerda, int topo, int direita, int base, int profund, int topflag);
circle	Desenha um círculo.	circle (int x, int y, int raio);
cleardevice	Limpa a tela gráfica.	cleardevice (void);
clearviewport	Limpa o viewport atual.	clearviewport (void);
closegraph	Fecha o sistema de gráficos.	closegraph (void);
detectgraphc	Determina o controlador e o modo do hardware.	detectgraph (int far *contro_graf, int far *modogrf);
drawpoly	Desenha o contorno de um polígono.	drawpoly (int numPontos, int far *Pontos_poli);

Função	Descrição	Sintaxe
ellipse	Desenha um arco elíptico.	ellipse (int x, int y, int anguloini, int angfim, int xraio, int yraio);
fillellipse	Desenha e preenche uma oval.	fillellipse (int x, int y, int xraio, int yraio);
fillpoly	Desenha e preenche um polígono.	fillpoly (int numPontos, int far *Pontos_poli);
floodfill	Preenche toda uma região delimitada.	floodfill (int x, int y, int limites);
getarcoords	Obtém as coordenadas da última chamada para o <i>arc</i> .	getarcoords (struct arcoordstype far *coords_arco);
getaspectratio	Obtém a relação altura/largura para o modo atual.	getaspectratio (int far *xrelacao, int far *yrelacao);
getbkcolor	Retorna a cor de fundo atual.	getbkcolor (void);
getdefaultpallet	Seleciona a definição da estrutura da paleta.	getdefaultpallet (void);
getdrivename	Aponta para uma string com o controlador gráfico.	getdrivename (void);
getfillpattern	Copia o padrão de preenchimento do usuário para a memória.	getfillpattern (char far *padrao);
getfillsettings	Obtém informações sobre o padrão de preenchimento e cor atual.	getfillsettings (struct fillsettingstype far *info_preenche);
getgraphmode	Retorna o modo gráfico corrente.	getgraphmode (void);
getimage	Salva a imagem especificada na memória.	getimage (int esquerda, int topo, int direita, int base, void far *mapabits);
getlinesettings	Obtém o estilo de linha, padrão e espessura.	getlinesettings (struct linesettingstype far *info_linha);
getmaxcolor	Retorna o valor máximo que pode ser enviado para o <i>setcolor</i> .	getmaxcolor (void);
getmaxmode	Retorna o valor máximo para o modo para o controlador.	getmaxmode (void);

Função	Descrição	Sintaxe
getmaxx	Retorna a máxima coordenada de tela x (em pixel).	getmaxx (void);
getmaxy	Retorna a máxima coordenada de tela y (em pixel).	getmaxy (void);
getpalette	Retorna informações sobre a paleta atual.	getpalette (struct pallettype far *paleta);
getpalletsize	Obtém o tamanho da tabela de consulta da paleta de cores.	getpalletsize (void);
getpixel	Obtém a cor do pixel especificado.	getpixel (int x, int y);
gettextsettings	Retorna as informações da fonte do gráfico de texto atual.	gettextsettings (struct textsettingstype far *infotipotexto);
getviewsettings	Retorna o estado do <i>viewport</i> do gráfico corrente.	getviewsettings (struct viewporttype far *viewport);
getx	Informa a posição x atual na tela.	getx (void);
gety	Informa a posição y atual na tela.	gety (void);
graphdefaults	Reinicializa todos os gráficos com valores padrões.	graphdefaults (void);
grapherrormsg	Retorna o ponteiro para string da mensagem de erro.	grapherrormsg (int cdgerro);
_graphfreemem	Oferece uma entrada para deslocação da memória gráfica.	_graphfreemem (void far *ptr, unsigned tamanho);
_graphgetmem	Oferece uma entrada para alocação da memória gráfica.	_graphgetmem (unsigned tamanho);
graphresult	Retorna o código de erro da última operação que falhou.	graphresult (void);
imagesize	Retorna o número de bytes necessário para armazenar uma imagem.	imagesize (int esquerda, int topo, int direita, int base);
initgraph	Inicializa o sistema gráfico.	initgraph (int far contro_graf, int far *modograf, char far *caminho_controlador);
installuserdriver	Instala um controlador fornecido pelo fabricante.	installuserdriver (char far nome, int grandao(*detect) (void) );

## Programação Estruturada

Função	Descrição	Sintaxe
installuserfonte	Instala uma fonte fornecida pelo fabricante.	installuserfonte (char far *nome);
line	Desenha uma linha.	line (int x1, int y1, int x2, int y2);
linerel	Desenha uma linha a uma determinada distância de um ponto.	linerel (int dx, int dy);
lineto	Desenha uma linha do ponto atual até o ponto especificado.	lineto (int x, int y);
moveto	Move a posição atual até um determinado ponto.	moveto (int x, int y);
outtex	Exibe uma string no <i>viewport</i> .	outtext (char far *stringtext);
outtextxy	Exibe uma string em uma determinada posição.	outtextxy (int x, int y, char far *stringtext);
pieslice	Desenha e preenche uma fatia do gráfico setorial.	pieslice (int x, int y, int anguloini, int angfim, int raio);
putimage	Saída de bit de imagem para a tela.	putimage (int esquerda, int topo, void far *mapabits, int op);
putpixel	Coloca um pixel numa determinada posição.	putpixel (int x, int y, int cor);
rectangle	Desenha um retângulo.	rectangle (int esquerda, int topo, int direita, int base);
sector	Desenha e preenche uma fatia do gráfico setorial elíptico.	sector (int x, int y, int anguloini, int angulofim, int xraio, int yraio);
setactivepage	Inicializa a página ativa para a saída gráfica.	setactivepage (int pagina);
setallpalette	Modifica todas as paletas de cores como especificado.	setallpalette (struct pallettype far *palette);
setspectratio	Modifica a relação altura/largura padrão.	setspectratio (int xrelacao, int yrelacao);
setbkcolor	modifica a cor de fundo.	setbkcolor (int cor);
setfillpattern	Modifica o padrão de preenchimento.	setfillpattern (char far *upadrao, int cor);

Função	Descrição	Sintaxe
setfillstyle	Modifica o estilo de preenchimento.	setfillstyle (int padrao, int cor);
setgraphbufsize	Modifica o tamanho do buffer gráfico.	setgraphbufsize (undigned tambuf);
setgraphmode	Define o modo gráfico e limpa a tela.	setgraphmode (int modo);
setlinestyle	Define o tamanho e o estilo da linha.	setlinestyle (int estilolinha,unsigned upadrao, int espessura);
setpalette	modifica uma paleta de cor.	setpalette (int numcor, int cor);
setrgbpalette	Define as cores para a tabela da paleta.	setrgbpalette (int numcor, int vermelho, int verde, int azul);
settextjustif	Define a justificação de texto para gráficos.	settextjustif (int horiz, int vert);
settxtstyle	Define as características de texto para gráficos.	settxtstyle (int fonte, int direcao, int tamchar);
setusercharsize	Define a altura e o tamanho das fontes vetoriais.	setusercharsize (int multx, int divx, int multy, int divy);
setviewport	Define o tamanho do <i>viewport</i> .	setviewport (int esquerda, int topo, int direita, int base, int corte);
setvisualpage	Define o número da página visual.	setvisualpage (int pagina);
setwritemode	Define o modo escrita para o desenho da linha.	setwritemode (int modo);
textheight	Retorna a altura de uma string em pixel.	textheight (char far *stringtexto);
textwidth	Retorna a largura de uma string em pixel.	textwidth (char far *stringtexto);

Fonte: SCHILDT (1999, p. 444-471).

Após esse panorama das funções gráficas, vamos juntos aprender a configurar o ambiente Dev-C? Então, vamos lá!

## 10.2 Configurando o Ambiente Dev-C

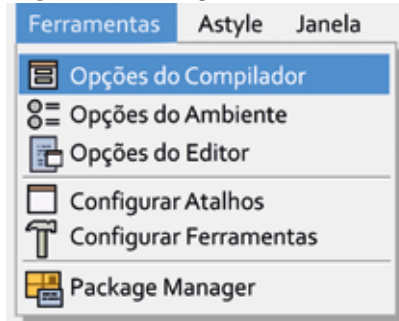
Para começarmos a programar no modo gráfico, precisamos primeiro configurar o ambiente. Isso facilitará o compilador a encontrar os arquivos necessários para a execução dos programas, não é mesmo?

Inicialmente, vamos verificar se os arquivos *graphics.h* e *libbgg.a* estão nas suas pastas correspondentes. Ou seja:

- × *graphics.h* precisa estar na pasta ***include***; e
- × *libbgg.a* precisa estar na pasta ***lib***.

Em seguida, é necessário configurarmos as chamadas dos arquivos no **Menu Ferramentas, Opções do Compilador**, conforme mostra a Figura 2.

Figura 2 – Configurando o Ambiente.



Fonte: Elaborada pelo autor (2015).

Após, precisamos inserir a linha de comandos, de acordo com a Figura 3.

Observe que os comandos (**-lbg; -lgdi32; -lcomdlg32; -luuid; -loleaut32; -lole32**) devem ser digitados como uma das opções do Compilador, mais especificamente para serem adicionados à linha de comando do **Linker** no momento da compilação. E lembre-se: esses comandos são muito importantes para a compilação dos programas no modo gráfico.

Figura 3 – Linha de Comandos.



Fonte: Elaborada pelo autor (2015).

Agora que preparamos o ambiente, que tal começarmos a programar? Então, prossiga para o próximo tópico!



## 10.3 Começando a Programação com a Biblioteca Graphics.h

Para iniciarmos nossa programação com a Biblioteca **Graphics.h**, o que precisamos fazer em um primeiro momento? Temos que, inicialmente, detectar o controlador e o modo de vídeo atual, e só após inicializar o sistema gráfico.

A fim de detectar o tipo de vídeo, utilizaremos as funções **detect()** e **initgraph()**. A função **detect()** atribui o número zero (0) como um valor padrão a uma variável correspondente ao driver da Biblioteca BGI. Já a função **initgraph()** inicializa o modo gráfico a partir dos seguintes parâmetros:

- × o driver de vídeo a partir do resultado da função **detect()**; e
- × a localização do diretório BGI corrente.

Figura 4 – Inicializando o modo gráfico em C.

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
    int gdriver, gmode, errorcode;

    gdriver=DETECT;
    initgraph(&gdriver, &gmode, " ");
    errorcode=graphresult();

    if (errorcode !=0)
    {
        printf ("Erro ao inicializar o modo grafico: %s\n", grapherrormsg(errorcode));
        printf ("Pressione qualquer tecla para continuar...");
        getch();
        exit(1);
    }

    printf ("Modo grafico inicializado com sucesso\n\n");
    getch();
    closegraph();
    return(0);
}
```

Fonte: Elaborada pelo autor (2015).

Perceba que, da linha 1 à linha 4, foram inseridas as bibliotecas com a diretiva **#include**. Enquanto as variáveis **gdriver**, **gmode** e **errorcode** foram definidas na linha 7. Na linha 9, a variável **gdriver** recebeu o resultado de DETECT, que identifica o tipo de vídeo utilizado.

Na linha 10, veja que a função ***initgraph()*** inicializou o modo gráfico e, na linha 11, a função ***graphresult()*** retornou o resultado da verificação do modo gráfico. Tal função, inclusive, trará como resultado um valor 0 ou 1.

Na linha 13, a variável ***errorcode*** foi testada quanto ao seu conteúdo. Se o conteúdo da variável ***errorcode*** for diferente de zero (por exemplo, igual a 1), então a função retornará um erro. Assim, a estrutura de seleção ***if*** mostrará uma mensagem de erro, como pode ser vista nas linhas 15 e 16. Caso contrário, na linha 21, será mostrada uma mensagem que o programa inicializou o modo gráfico com sucesso. Finalmente, na linha 24, a função ***closegraph()*** encerrará o modo gráfico e retornará ao modo texto.

Depois da inicialização, o que fazer? Como desenvolver o primeiro programa no modo gráfico? A seguir iremos analisar os parâmetros de cada função para que elas funcionem na hora da programação. Siga avanti!

### 10.3.1 Primeiro Programa no Modo Gráfico

Todo programa no modo gráfico necessita da inicialização, conforme acompanhamos na Figura 4. E depois? O que é preciso para começar a programar? Você concorda que precisamos usar as funções gráficas definidas na Biblioteca *Graphics.h*? Mas antes disso, vamos juntos observar os parâmetros de cada função, isto é, o que cada função exige para efetivamente funcionar?

Na Figura 5, por exemplo, o modo gráfico é inicializado da linha 7 até a linha 22. E, em seguida, da linha 24 até a linha 34, temos a chamada de algumas funções gráficas, as quais foram apresentadas na Tabela 1.

Figura 5 – Primeiro programa no modo gráfico.

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
    int gdriver, gmode, errorcode;

    gdriver=DETECT;
    initgraph(&gdriver, &gmode, " ");
    errorcode=graphresult();

    if (errorcode !=0)
    {
```

```

int gdriver, gmode, errorcode;

gdriver=DETECT;
initgraph(&gdriver, &gmode, " ");
errorcode=graphresult();

if (errorcode !=0)
{
    printf ("Erro ao inicializar o modo grafico: %s\n", grapherrormsg(errorcode));
    printf ("Pressione qualquer tecla para continuar...");
    getch();
    exit(1);
}

printf ("Modo grafico inicializado com sucesso\n\n");
getch()

cleardevice();
setcolor(14);
arc(20,50,9,90,40);
bar(70,20,110,40);
bar3d(130, 20, 170,40, 10, 10);
circle(220,30,25);
ellipse(418,30,260,100,50,20);
moveto(x,y);
lineto(320,200);
getchar();
closegraph();
}

```

Fonte: Elaborada pelo autor (2015).

Em relação aos parâmetros passados para as diversas funções gráficas utilizadas, é importante ressaltar que cada função possui seu próprio conjunto de parâmetros esperados. Ou seja, os valores informados como parâmetros variam de função para função. Veja, a seguir, uma explicação sobre os parâmetros passados para os métodos da Figura 5.

- × No caso da função “setcolor”, devemos passar como parâmetro qual cor será utilizada para escrever ou desenhar na tela.
- × Na função “arc”, há cinco parâmetros no nosso exemplo: os dois primeiros indicam o  $x$  e o  $y$ , os quais definem o ponto central da circunferência; o terceiro e quarto parâmetros indicam, respectivamente, o ângulo inicial e final da circunferência; o último parâmetro determina o tamanho do raio.
- × Na função “bar”, existem quatro parâmetros, indicando a coordenada do canto esquerdo, a coordenada do topo, a coordenada do canto direito e a coordenada da base, respectivamente.

- × Na função “bar3d”, têm os mesmos parâmetros da função “bar”, acrescidos de um parâmetro para indicar a profundidade da barra e de outro para especificar se a linha superior da barra será desenhada.
- × Na função “circle”, há parâmetros das coordenadas  $x$  e  $y$ , além do raio do círculo.
- × Na função “ellipse”, existem os mesmos parâmetros da função “arc”, acrescido de outro valor de raio (utiliza  $x$  e  $y$  para especificação do raio).
- × Na função “lineto”, por fim, têm parâmetros dois números, que desenharam uma linha da primeira posição até a segunda.

### Importante

A tela no modo gráfico está dividida em pequenos pontos chamados de pixels (Picture elements), que são os pontos que formam a imagem. A posição inicial (0,0) indica o canto superior esquerdo da tela.

Após vermos uma das possibilidades de programa no modo gráfico, vamos conhecer mais alguns exemplos? Siga-nos!

## 10.4 Exemplos de Programas no Modo Gráfico

O programa da Figura 6 apresenta uma sequência de quatro círculos de diferentes cores.

Figura 6 – Segundo programa no modo gráfico.

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
void opengraph(void) {
    int driver, mode;
    detectgraph(&driver, &mode);
    initgraph(&driver, &mode, "");
}
void main(void) {
    int opcao, driver, mode;
    char *GDrivers[] = {"DETECT",
                       "CGA",
                       "MCGA",
                       "EGA",
                       "EGA64",
```

```

        "EGAMONO",
        "1BM8514",
        "HERCMONO",
        "ATT400",
        "VGA",
        "PC3270");

opengraph();
detectgraph(&driver, &mode);
printf("%s", GDrivers[driver]);
setcolor(1)
circle(100, 100, 50)
setcolor(2);
circle(200, 100, 50);
setcolor(3);
circle(300, 100, 50);
setcolor(4);
circle(400, 100, 50);
getche();
closegraph();
}

```

Fonte: Elaborada pelo autor (2015).

Perceba que, nesse exemplo, temos uma chamada para a função *opengraph()*, a qual detecta o vídeo e inicializa o modo gráfico.

O programa apresentado na Figura 7 inicializa, na linha 6, uma janela gráfica de 640 x 480 *pixels*. Nas linhas 7, 11, 15 e 17, seleciona as cores RED, BLUE, GREEN e WHITE, respectivamente. Já, nas linhas, 8 a 10, linhas 12 a 14 e linhas 16 e 18, desenha uma linha com a função *line()*, com as cores anteriormente escolhidas.

Figura 7 – Terceiro programa no modo gráfico.

```

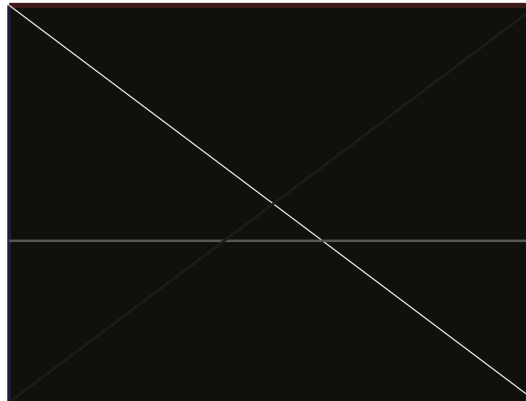
#include<graphics.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
    initwindow(640,480);
    setcolor(RED);
    line(0,0,640,0);
    line(0,1,640,1);
    line(8,2,640,2);
    setcolor(BLUE);
    line(0,0,0,480);
}

```

```
line(1,0,1,480);  
line(2,0,2,480);  
setcolor(GREEN);  
line(0,480,640,0);  
setcolor(WHITE);  
line(0,0,640,480);  
system("PAUSE");  
closegraph();  
return 0;  
}
```

Fonte: Elaborada pelo autor (2015).

Figura 8 – Saída de tela do terceiro programa no modo gráfico.



Fonte: Elaborada pelo autor (2015).

### Importante

A saída de tela de um programa no modo gráfico será realizada dentro dos parâmetros de inicialização da janela gráfica, a qual será definida a partir das coordenadas, ou seja, das posições da janela e dos pixels.

## Resumo

Neste capítulo, entendemos como inicializar o modo gráfico na linguagem de programação C. Além disso, compreendemos como configurar o modo gráfico no ambiente Dev-C.

O modo gráfico pode ser útil para diversos propósitos, porém como ele é composto de diversas funções, cada uma com suas características particulares, é importante que você tente criar programas que executem essas funções e veja também como é o funcionamento de cada uma delas. Por isso, para cada função executada, altere os valores dos parâmetros e teste o funcionamento do programa, a fim de verificar o efeito das mudanças realizadas no código-fonte.

E lembre-se: as funções gráficas podem ser utilizadas para a construção de simples jogos de computador, que não demandam riqueza de detalhes das interfaces. Faça uma pesquisa na internet a respeito deste tema e tente implementar um pequeno jogo para testar suas habilidades com a Linguagem C e sua biblioteca de gráficos.

Por fim, vimos as principais funções para criar formas geométricas em linguagem de programação estruturada com C, bem como conhecemos alguns exemplos de programa no modo gráfico.





# 11

## Ponteiros

NESTE CAPÍTULO CONHECEREMOS um outro tipo de variável em linguagem de programação estruturada C, que são os Ponteiros. A compreensão do que é um ponteiro e de quando e como utilizá-lo é muito importante para programadores C. O uso de ponteiros é uma vantagem que a linguagem de programação C oferece para agilizar a execução de um programa, porém, alguns cuidados importantes são necessários, pois o seu uso indevido pode acarretar muitos problemas ao programador.

### Objetivo da Aprendizagem:

- × Conhecer e compreender o conceito de ponteiros em linguagem de programação estruturada C.

## 11.1 Ponteiros

Segundo o programador e autor do livro mais famoso sobre a linguagem C, Herbert Schildt:

Um ponteiro é uma variável que contém o endereço de memória de outra variável. São utilizados para alocação dinâmica, podendo substituir matrizes com mais eficiência. Também fornecem a maneira pelas quais as funções podem modificar os argumentos que são chamados. (SCHILDT, 1990, p113).

Em outras palavras, um ponteiro é uma variável cujo conteúdo é o endereço de memória de outra variável. O conteúdo de uma variável do tipo inteiro é um número inteiro, o conteúdo de uma variável do tipo float, é um número com casas decimais, o conteúdo de uma variável do tipo char, é uma sequência de caracteres, e por fim, o conteúdo de uma variável ponteiro, é o **endereço de memória**, de outra variável. Se uma variável armazenará o endereço de memória, ela deve ser declarada como um **ponteiro**.

É importante lembrar que quando declaramos uma variável, em linguagem de programação estruturada C, precisamos definir qual o tipo de dado que essa variável vai armazenar, bem como, qual o seu nome, que será o que a identificará unicamente dentro do programa. Por exemplo, quando declaramos uma variável como sendo um **int idade**, estamos solicitando ao compilador C, que seja reservado um espaço na memória do computador, que seja do tipo inteiro e que seja identificado com o nome de **idade**. Nesse momento, além de criar a variável com o tipo de dado e nome escolhidos, o compilador precisa **alocar um espaço na memória** do computador para armazenar essa variável em tempo de execução do programa. Dessa forma, o compilador determina um **endereço de memória**, onde armazenará o conteúdo dessa variável durante a execução do programa.

Quando não precisamos acessar o **conteúdo** de uma variável, mas o que precisamos é saber qual o **endereço de memória** no computador em que esta variável se encontra, então precisamos usar o recurso chamado **ponteiros**.

A Figura 1 ilustra o nome, o tipo de dado e o conteúdo de uma variável, e o nome tipo de dado e conteúdo de uma variável do tipo ponteiro.

Figura 1 – Variável e Variável Ponteiro

Nome da Variável	Tipo de Dado da Variável	Conteúdo da Variável	Nome da Variável Ponteiro	Tipo de Dado da Variável Ponteiro	Conteúdo da Variável Ponteiro
idade	inteiro	54	<b>p</b> idade	inteiro	1003
peso	float	67,6	<b>pp</b> eso	float	1004
nome	char	João da Silva	<b>pn</b> ome	char	1005

Fonte: Elaborado pelo autor (2015)

Uma variável do tipo ponteiro também é igualmente armazenada em um endereço de memória no computador. Todas as variáveis, independentemente, do seu tipo de dado, são armazenadas em um endereço na memória. A Figura 2 mostra que o endereço de memória 1000 tem como conteúdo o endereço de memória 1003.

Figura 2 – Variável Ponteiro

Endereço de Memória	Conteúdo da Variável na Memória
1000	<b>1003</b>
1001	
1002	
<b>1003</b>	
1004	
1005	
1006	

Fonte: (SCHILDT, 1990, p. 114)

É importante distinguirmos quando referenciamos o *endereço de memória* de uma variável, de quando referenciamos o *conteúdo dessa variável*, e este conteúdo pode ser exatamente, o *endereço de memória de outra variável*.

### 11.1.1 Alocação de Memória

Quando falamos que ao criar uma variável o compilador reserva um espaço na memória do computador, isto quer dizer que o compilador está fazendo uma **alocação de memória**.

**O que é isso ? Alocação dinâmica de memória** é quando o compilador reserva um espaço na memória para todos os dados declarados inicialmente, em um programa em linguagem de programação C. O compilador faz automaticamente essa reserva de memória ao executar o programa. No entanto, se pretendemos usar os endereços de memória das variáveis, precisamos, obrigatoriamente, reservar um espaço de memória para armazenar os endereços de memória das variáveis.

Quando precisamos reservar um espaço de memória em tempo de execução do programa, essa tarefa é chamada de **alocação dinâmica de memória** e significa que temos a possibilidade de inserir dados em uma variável em tempo de execução, mas que não sabemos exatamente a quantidade de dados que essa variável vai precisar. Logo, precisamos organizar a memória do computador para que essa nos permita executar o programa mesmo não sabendo, inicialmente, quantos dados serão inseridos.

Um exemplo bem prático dessa situação é quando não sabemos, no momento da programação, a quantidade de alunos que deverão ser inseridos em uma turma. A linguagem de programação estruturada C permite essa **reserva de espaço na memória** em tempo de execução do programa. Outro exemplo clássico são os editores de texto, nos quais não se sabe a quantidade de letras que o usuário irá digitar.

Da mesma maneira que declaramos variáveis do tipo **char**, **int** e **float**, devemos declarar as variáveis que armazenarão os endereços de memória. Para isso, precisamos declarar as variáveis com o **tipo de dado ponteiro**.

Uma variável do tipo de dado ponteiro armazena o endereço de memória de uma outra variável. Ao declarar uma variável do tipo de dado ponteiro, deve-se informar também que tipo de informação estará contida no endereço que o ponteiro irá armazenar. Por exemplo, um ponteiro **int** aponta para um **inteiro**, isto é, ele é capaz de guardar o endereço de um inteiro.

## 11.2 Declarando Ponteiros

Para declarar uma variável do tipo de dado ponteiro, é necessário conhecermos dois símbolos que são usados em linguagem de programação C para a caracterização desse tipo de dado.

O primeiro símbolo é o **\*** (**asterisco**). O asterisco significa que aquela variável é uma variável que armazenará um endereço de memória. O asterisco informa ao compilador que aquela variável é uma variável do tipo de dado ponteiro.

Logo, se uma variável armazenará o **endereço de memória** de outra variável, então essa variável precisa ser declarada como segue:

```
int idade, *pidade;
```

Na declaração acima, o compilador entende que foram declaradas duas variáveis, sendo a variável **idade** uma variável do tipo inteiro e a variável **\*pidade**, sendo uma variável do tipo **ponteiro**.

```
pidade = &idade;
```

O segundo símbolo é o **&** (**& comercial**). O **&** comercial significa que o que está sendo atribuído à variável **pidade** é o endereço de memória da variável **idade**. O **&** comercial informa ao compilador que deverá ser acessado o **endereço de memória daquela variável** e não o seu conteúdo propriamente dito.

Isto quer dizer que a variável ponteiro **pidade** aponta para um inteiro, e que terá como **conteúdo o endereço de memória** da variável **idade**, e **não o seu conteúdo**.

Dessa forma, vamos observar abaixo, como devemos prestar atenção na sintaxe para acessar o conteúdo das variáveis. Por exemplo, na Figura 1 o conteúdo da variável **idade** é 54 e o conteúdo da variável **pidade** é 1003, conforme segue:

```
idade = 54
*pidade = 1003
```

É importante observar que, se for necessário armazenar o conteúdo da variável **idade** em uma outra variável também do tipo inteira de nome **x**, precisamos fazer a atribuição da seguinte forma:

```
int x;  
x=idade;
```

A Figura 3 ilustra a atribuição de dados feita na variável **x**.

Figura 3 – Conteúdo da variável **x** após atribuição do conteúdo da variável **idade**

Nome da Variável	Tipo de Dado da Variável	Conteúdo da Variável
X	Inteiro	54

Fonte: Elaborado pelo autor (2015)

Após a atribuição de dados, a variável **x = 54**.

E, se for necessário armazenar o endereço da variável **idade** em uma outra variável também do tipo inteira de nome **y**, essa variável precisa ser do tipo ponteiro e a atribuição é feita da seguinte forma:

```
int *y;  
y=&idade;
```

Figura 4 – Conteúdo da variável **y** após atribuição do endereço de memória da variável **idade**

Nome da Variável Ponteiro	Tipo de Dado da Variável Ponteiro	Conteúdo da Variável Ponteiro
Y	inteiro	1003

Fonte: Elaborado pelo autor (2015)

Após a atribuição de dados, a variável **y = 1003**.



## Saiba mais

### Problema com Ponteiros

Um dos problemas mais comuns ao utilizarmos ponteiros em linguagem de programação C, é chamado de **ponteiro perdido**. Esse é um dos problemas de programação em C mais difícil de ser encontrado, por

que cada vez que a operação com um ponteiro é utilizada, o compilador poderá estar lendo ou gravando em posições desconhecidas da memória. Isto é, pode acontecer de usarmos um endereço de memória que já está sendo usado para armazenar outros dados. Se isso acontece, há o problema de sobreposições das áreas de dados, o que causa a perda de dados que já estavam armazenados na memória.

```
intidade, *pidade;

pidade=10;
```

Observando a atribuição acima, a variável ponteiro **pidade** está recebendo o número 10, isso atribui o número 10 a uma posição de memória completamente desconhecida pelo compilador. Esse tipo de atribuição direta a uma variável ponteiro gera muitos problemas ao programador porque não se sabe em qual endereço de memória está sendo feita a atribuição.



## 11.3 Como Funcionam os Ponteiros

Um exemplo prático de como funciona a lógica a partir do conceito de ponteiros, é quando precisamos anotar o endereço de um colega. Nesse momento, o que fazemos? Sem perceber, nesse momento estamos criando **ponteiros**. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço.

Qual é o sentido disso? **Simples**. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O compilador C faz a mesma coisa, grava o endereço de algo em uma variável chamada **ponteiro** para depois encontrar esse “algo” mais rapidamente. Da mesma maneira, como usamos uma Agenda!! Em uma Agenda guardamos os endereços dos nossos amigos e essa Agenda pode ser vista como uma **matriz de ponteiros** em C.

Um ponteiro também tem um tipo de dado. Veja: um ponteiro pode ser do tipo **inteiro**, **char**, **float**, etc. Em linguagem de programação C; devemos declarar os ponteiros de acordo com o tipo de dado da variável que vamos apontá-lo. Um ponteiro do tipo inteiro **int** apontará para uma variável que armazena um dado do tipo **inteiro**.

## 11.4 Expressões com Ponteiros

Em geral, expressões envolvendo ponteiros concordam com as mesmas regras de qualquer outra expressão em linguagem de programação estruturada em C. Vejamos alguns aspectos especiais das expressões com ponteiros:

- × Atribuições de ponteiros.
- × Aritmética de ponteiros.
- × Comparação de ponteiros.

### 11.4.1 Atribuição de Ponteiros

Como é o caso com qualquer variável, um ponteiro pode ser usado no lado direito de um comando de atribuição para passar seu valor para outro ponteiro, como pode ser observado a seguir:

Figura 5 – Atribuição com ponteiros

```
#include <stdio.h>
int main ()
{
    int x=10;
    int *p1, *p2;
    p1 = &x;
    p2 = p1;
    printf("O endereço de p2 eh: %p\n", p2);
    printf("O conteúdo do endereço apontado pelo ponteiro p1 eh: %d\n", *p1);
    printf("O conteúdo do endereço apontado pelo ponteiro p2 eh: %d\n", *p2);
    system("pause");
}
```

Tanto p1 quanto p2 apontam para o endereço de memória da variável x

Fonte: Elaborado pelo autor (2015)

A saída no console do programa será:

Figura 6 – Saída no console sobre atribuição com ponteiros

```
O endereço de p eh: 0022FF74
O conteúdo do endereço apontado pelo ponteiro p1 eh: 10
O conteúdo do endereço apontado pelo ponteiro p2 eh: 10
Pressione qualquer tecla para continuar..._
```

Fonte: Elaborado pelo autor (2015)



### 11.4.2 Aritméticas de Ponteiros

Existem apenas duas operações aritméticas que podem ser usadas com ponteiros:

- × **adição (incremento ++)** e
- × **subtração (decremento --).**

Os incrementos e decrementos dos endereços podem ser realizados com os operadores ++ e --, que possuem procedência sobre o \* em operações matemáticas, além de serem avaliados da direita para a esquerda. Supondo que pidade é um ponteiro, as operações são escritas como:

```
*p++;
```

```
*p--;
```

#### Importante

Estamos falando de operações com ponteiros e não de operações com o conteúdo das variáveis para as quais eles apontam.

Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Se tivermos um ponteiro para um inteiro e o incrementamos, ele passa a apontar para o próximo ponteiro do tipo inteiro. Isso justifica a necessidade do compilador conhecer o tipo de um ponteiro. Se incrementarmos um ponteiro char \* ele anda 1 byte na memória. Se incrementarmos um ponteiro float \* ele anda 4 bytes na memória. Se incrementarmos um ponteiro double \* ele anda 8 bytes na memória.

Figura 7– Saída no console sobre operações aritméticas com ponteiros

```
#include <stdio.h>
int main ()
{
    float *p1;
    printf("\0 endereço de p1 eh: %p", p1);
    p1++;
    printf("\n\0 novo endereço de p1 eh: %p\n\n", p1);
    system("pause");
}
```

p1 apontará para o próximo elemento do tipo float (4 bytes adiante)

```
O endereco de p1 eh: 000000002
O novo endereco de p1 eh: 000000006
Pressione qualquer tecla para continuar..._
```

Fonte: Elaborado pelo autor (2015)

A aritmética de ponteiros não se limita apenas ao incremento ++ e decremento - - ++. Podemos, também, somar ou subtrair inteiros de ponteiros, como pode ser observado nos exemplos a seguir.

Figura 8– Saída no console sobre a operação soma com ponteiros

```
#include <studio.h>
int main ()
{
    float *p1;
    printf("O endereco de p1 eh: %p", p1);
    p1 = p1 + 20; //equivale à p1+=20;
    printf("\n\nO novo endereco de p1 eh: %p\n\n", p1);
    system("pause");
}
```

p1 apontará para o  
20° elemento do tipo  
float adiante (20 x 4  
bytes adiante)

```
O endereco de p1 eh: 0000000000000002
O novo endereco de p1 eh: 0000000000000052
Pressione qualquer tecla para continuar..._
```

Fonte: Elaborado pelo autor (2015)

Agora, vamos retornar ao nosso exemplo, e considerar a variável ponteiro **idade** que aponta para o conteúdo da variável **idade**. Ok !Então o ponteiro **\*idade** pode ser utilizado em qualquer lugar que a variável **idade** aparece. Lembrando que o operador **\*** tem maior precedência que as operações aritméticas, assim a expressão a seguir pega o conteúdo do endereço que **idade** aponta e soma **1** ao seu conteúdo, conforme exemplo abaixo:

Figura 9– Saída no console com operações de incremento ++ e decremento -- com ponteiros

```
#include <stdio.h>
main() {
    int y, idade, *pidade;
    idade=54;
    y=0;
    pidade=&idade;

    printf ("idade: %d\n\n", idade);
    printf ("y: %d\n\n", y);
    printf ("pidade: %p\n\n", *pidade);
    printf ("%d\n\n", &idade);

    pidade++;
    printf ("pidade++: %p\n\n", *pidade);

    pidade--;
    printf ("pidade--: %p\n\n", *pidade);

    y=*pidade;
    printf ("y: %d\n\n", y);

    y=*pidade+3;
    printf ("*pidade +1: %d\n\n", y);
    *pidade++;
    printf ("*pidade++: %p\n\n", *pidade);
    *pidade--;
    printf ("*pidade--: %p\n\n", *pidade);

    pidade++;
    printf ("pidade++: %p\n\n", pidade);

    pidade--;
    printf ("pidade--: %p\n\n", pidade);
}
```

```
idade: 54
y: 0
pidade: 0000000000000036
&idade: 2293308
pidade++: 000000000024FE40
pidade--: 000000000000036
y: 54
*pidade +1: 57
*pidade++: 000000000024FE40
*pidade--: 000000000000036
pidade++: 000000000024FE40
pidade--: 000000000024FE3C
Pressione qualquer tecla para continuar...
```

Fonte: Elaborado pelo autor (2015)

Por exemplo, para incrementar o **conteúdo** da variável apontada pelo ponteiro **pidade**, é preciso fazer o seguinte:

```
*pidade+3;
```

Nesse exemplo, o conteúdo apontado pela variável ponteiro **pidade** será incrementado em 3 números, isto é, será somado 3 ao número 54 que é o conteúdo armazenado na variável **idade**.

## Importante

Entretanto, há operações que você não pode efetuar com ponteiros. Por exemplo, você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair **floats** ou **doubles** de ponteiros. Em outras palavras, além de adição e subtração entre um ponteiro e um inteiro, nenhuma outra operação aritmética pode ser efetuada com ponteiros. Não podemos multiplicar ou dividir ponteiros.

### 11.4.3 Comparação de Ponteiros

É possível comparar dois ponteiros em uma expressão relacional (<, <=, > e >=) ou se eles são iguais (=) ou diferentes (!=). A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer, como podemos observar a seguir:

Figura 10– Saída no console da comparação de ponteiros

```
#include <stdio.h>
int main()
{
    int x=10, y=10;
    int *p1, *p2;
    p1 = &x;
    p2 = &y;
    if (p1>p2)
        printf ("A variavel x esta armazenada em um endereco de memoria acima da variavel
y");
    else
        printf ("\n\nA variavel y esta armazenada em um endereco de memoria acima da variavel
x");
    printf ("\n\nCertificando ... \n\t\tEndereco de x: %p \n\t\tEndereco de y: %p\n\n");
}
```

```
A variavel x esta armazenada em um endereco de memoria acima da variavel y

Certificando...
Endereco de x: 003D3858
Endereco de y: 003D29B8
Pressione qualquer tecla para continuar...
```

Fonte: Elaborado pelo autor (2015)

Na linha 8, da Figura 10 Como vimos na Figura 10, podemos observar que como fazemos comparações entre ponteiros, em uma expressões relacionais de uma forma muito simples. Neste exemplo, o compilador analisa se as

posições de memórias estão acima ou abaixo de  $p1$  e  $p2$ . Importante lembrar, que somente é possível comparar ponteiros de mesmo tipo de dado.

## **Resumindo**

Neste capítulo conhecemos o que são Ponteiros. Compreendemos o que são, como os definimos, para que servem. Quando e como são utilizados. Compreendemos as suas particularidades de implementação, e por fim, vimos expressões com ponteiros: atribuição, aritmética e comparação de ponteiros.



# Conclusão

Caro aluno, ao analisarmos as principais estruturas de sistemas operacionais, você aprendeu como é possível trabalhar de diferentes formas com o *hardware* na resolução de gerenciamento das atividades computacionais.

Nesta disciplina, estudamos as expressões matemáticas, os operadores lógicos e relacionais, os comandos que devem ser executados para dar início ao programa, os tipos de variáveis utilizadas para leitura e saída de dados, as bibliotecas que devem ser usadas, além dos comandos de atribuições e conversões de dados para manipulações de variáveis. Descobrimos também como elaborar os nossos primeiros programas em linguagem de programação estruturada em C, com o ambiente de programação Dev-C.

Aprendemos que a estrutura de seleção muda o raciocínio de programas simplesmente sequenciais para um raciocínio com base em tomadas de decisões, ou seja, com base em escolhas. E conhecemos os três tipos de estruturas de controle: *for*, *while* e *do..while*; além de percebermos ainda que os dados armazenados em vetores são como se estivessem dentro de “caixas”, organizadas, sequencialmente, na memória do computador. Além disso, compreendemos que não se pode mexer em uma “caixa” não alocada na memória, pois, do contrário, podemos perder dados armazenados de outras operações.

Esta disciplina também possibilitou a sua compreensão sobre a inserção de uma função em um programa estruturado, bem como a utilizar essas funções a partir de exemplos.

Por fim, você pode aprender como realizar a leitura e a gravação de dados no modo texto e binário e estudou os conceitos de Pesquisa e Ordenação. Entendemos, inclusive, o quanto é importante a ordenação dos dados para a otimização das pesquisas em programação estruturada.

Assim, analisamos dois tipos de algoritmos: o algoritmo de ordenação simples, pelo método bolha (*bubble sort*), e o algoritmo de ordenação básica (*selection sort*). A partir dessas noções, foi possível ter uma visão básica de como o processador resolve as questões de processamento. Este conhecimento é importante para entendermos os tipos de linguagens de baixo e alto nível, e a forma como elas se comunicam com o sistema operacional e o *hardware*, bem como suas diferenciações em relação à produtividade, quando se trata de desenvolvimento de sistema.



## Referências

ASCENCIO, Ana Fernanda Gomes. **Lógica de Programação com Pascal**. São Paulo: Makronbooks, 1999.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. **Fundamentos da programação de computadores: algoritmos, Pascal, C/C ++**. 2 ed. São Paulo: Pearson Education do Brasil, 2007.

ASCENCIO, Ana Fernanda Gomes. **Fundamentos da programação de computadores: algoritmos, Pascal, C/C ++**. 3 ed. São Paulo: Pearson Education do Brasil, 2012.

DEITEL, H. M.; DEITEL, P. J. **C++ Como Programar**: 3 ed. São Paulo: Bookman, 2001. 1098 p.

DEITEL, H.M. **Sistemas Operacionais**. São Paulo: Pearson Prentice Hal, 2005.

DIJKSTRA, Edsger; Thomas J. Misa, “**Uma entrevista com Edsger W. Dijkstra**”, Editor. (2010-08).

EVARISTO, J., **Aprendendo a Programar Programando em Linguagem C**. Book Express, Rio de Janeiro, 2001.

FLYNN, Ida M. **Introdução aos sistemas operacionais**. São Paulo: Pioneira Thomson Learning, 2002.

FORBELLONE, André Luiz Villar, EBERSPASCHER, Henri Frederico. **Lógica de Programação: a construção de algoritmos e estrutura de dados**. 2.ed. São Paulo: Makron Books, 2000.

ISO/IEC (2003), NBR ISO/IEC 9126- NBR ISO/IEC 9126-1, Engenharia de Software 1, Qualidade de produto, Parte 1: Modelo de qualidade, **Associação Brasileira de Normas Técnicas (ABNT)**

MANZANO, J. A. N. G. **Estudo Dirigido de Linguagem C**. 11. Ed. São Paulo: Érica, 2002.

MIZRAHI. Victorine Viviane. **Treinamento em Linguagem C++: curso completo módulo 1**. São Paulo: McGraw-Hill, 1990.

Oliveira, Rômulo Silva de, Carissimi, Alexandre da Silva e Tocani, Simão Sirineo. **Sistemas Operacionais**. Editora SangralLuzzato, ISBN 85-241-0643, 2008.

Revista Abril <<http://info.abril.com.br/noticias/it-solutions/2014/07/quais-as-linguagens-mais-populares-este-grafico-responde.shtml>>. Acesso em 19 de janeiro de 2015.

Ranking das **linguagens mais utilizadas no Mercado de Softwares**. <<http://www.geeksbr.com/2010/11/programacao-em-c-funcao-isalpha.html>>. Acesso em 19 de janeiro de 2015.

Ranking das **linguagens mais utilizadas no Mercado de Softwares**. <<http://www.infoq.com/br/news/2014/10/ranking-linguagens-ieee>>. Acesso em 20 de janeiro de 2015.

SCHILDT, Herbert. **C Completo e Total**. 3ª ed. Revista e atualizada. Tradução e revisão técnica Roberto Carlos Mayer. São Paulo: Makron Books, 1996.

SCHILDT, H.. **C Completo e Total**. São Paulo: Makron books, 1999.

Site especializado em **qualidade e produtividade de software**. <<http://www.tiobe.com/index.php/content/company/GeneralInfo.html>>. Acesso em 15 de janeiro de 2015.

Site especializado em programação Linux. <<http://www.vivaolinux.com.br/artigo/Da-programacao-ao-IDE-NetBeans?pagina=3>>. Acesso em 25 de janeiro de 2015.

Site especializado em dicas de programação. <<http://www.geeksbr.com/>>. Acesso em 25 de janeiro de 2015

Site especializado em Linguagens de Programação. <[http://www.laifi.com/laifi.php?id\\_laifi=608&idC=5791#>](http://www.laifi.com/laifi.php?id_laifi=608&idC=5791#>)

Silberschatz, **Abraham Sistemas Operacionais**: conceitos. São Paulo: Prentice Hall, 2000.

Silberschatz, A., Gagne, G., Galvin, P. B. **Sistemas Operacionais com Java**: conceitos e aplicações. Rio de Janeiro : Campus, 2004.

Tanenbaum, Andrew S. (2003) **Sistemas operacionais modernos**. 2. ed. São Paulo : Prentice Hall, 2003.

