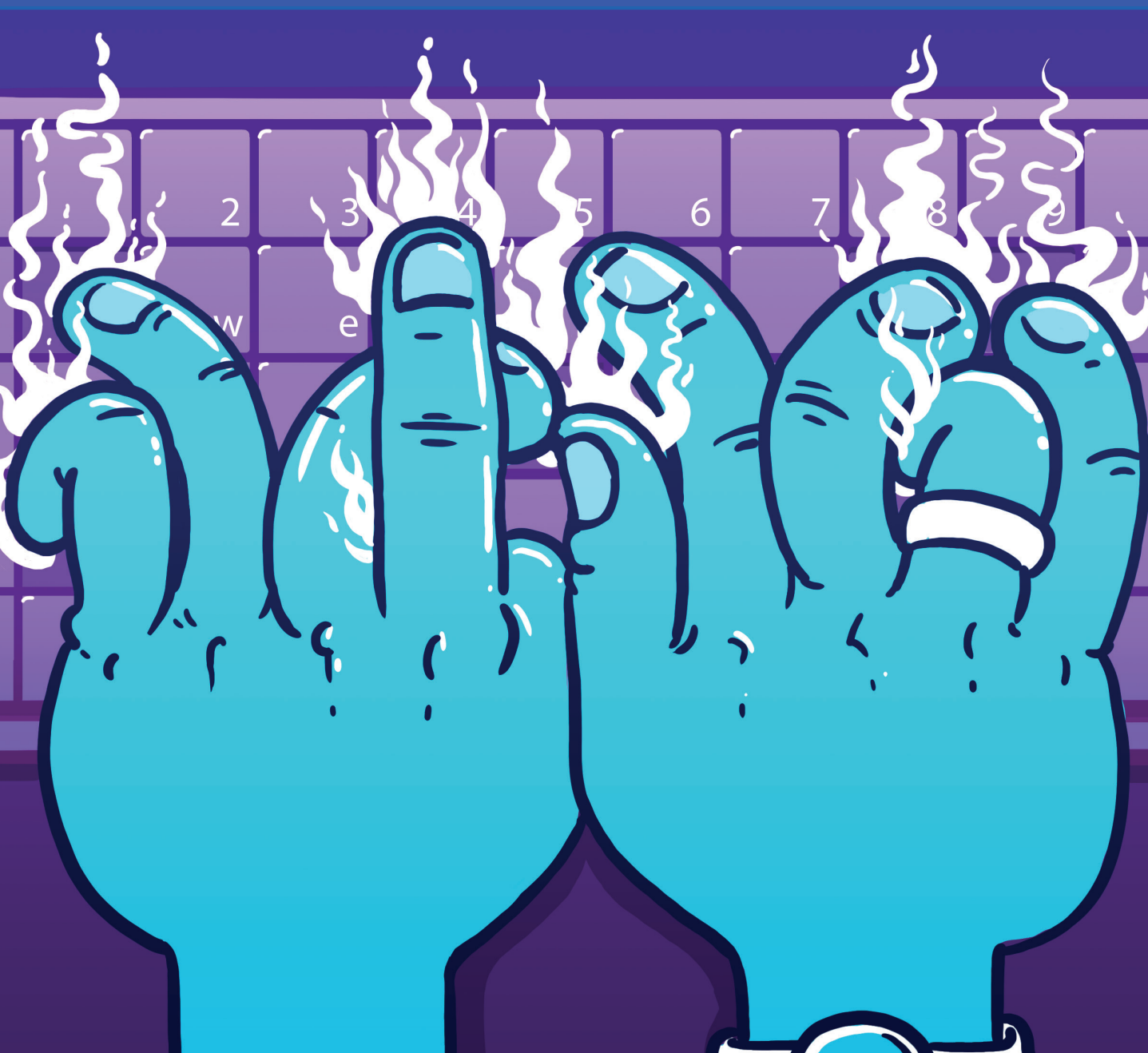
LINGUAGEM DE PROGRAMAÇÃO I

AUTORES
Cristiano Bertolini
Fábio José Parreira
Guilherme Bernardino da Cunha
Ricardo Tombesi Macedo



LICENCIATURA EM COMPUTAÇÃO

LINGUAGEM DE PROGRAMAÇÃO I

AUTORES
Cristiano Bertolini
Fábio José Parreira
Guilherme Bernardino da Cunha
Ricardo Tombesi Macedo

1ª Edição
UAB/NTE/UFSM

UNIVERSIDADE FEDERAL DE SANTA MARIA

Santa Maria | RS
2019

Este caderno foi elaborado pelo Núcleo de Tecnologia Educacional da Universidade Federal de Santa Maria para os cursos da UAB.

NÚCLEO DE TECNOLOGIA EDUCACIONAL

DIRETOR DO NTE
Paulo Roberto Colusso

ELABORAÇÃO DO CONTEÚDO
Cristiano Bertolini, Fábio José Parreira
Guilherme Bernardino da Cunha, Ricardo Tombesi Macedo

REVISÃO LINGUÍSTICA
Camila Marchesan Cargnelutti
Maurício Sena

APOIO PEDAGÓGICO
Eloísa Berlote Brenner
Caroline da Silva dos Santos
Keila de Oliveira Urrutia

EQUIPE DE DESIGN
Carlo Pozzobon de Moraes
Juliana Facco Segalla
Matheus Tanuri Pascotini
Raquel Pivetta

PROJETO GRÁFICO
Ana Letícia Oliveira do Amaral

L755 Linguagem de programação I [recurso eletrônico] / Cristiano Bertolini ... [et al.]. – 1. ed. – Santa Maria, RS : UFSM, NTE, 2019.
1 e-book

Este caderno foi elaborado pelo Núcleo de Tecnologia Educacional da Universidade Federal de Santa Maria para os cursos da UAB
Acima do título: Licenciatura em computação
ISBN 978-85-8341-246-5

1. Linguagem de programação 2. Linguagem C I. Bertolini, Cristiano II. Universidade Aberta do Brasil III. Universidade Federal de Santa Maria. Núcleo de Tecnologia Educacional

CDU 004.438

Ficha catalográfica elaborada por Alenir Goularte - CRB-10/990
Biblioteca Central da UFSM

APRESENTAÇÃO

A disciplina de Linguagem de Programação I abrangerá os conceitos básicos de algoritmos e programação utilizando-se da Linguagem de Programação C. O livro de Linguagem de Programação I pode ser utilizado em disciplinas entre 60 a 90 horas e como pré-requisito espera-se que todos tenham visto e compreendido as disciplinas de matemática discreta e algoritmos.

A disciplina está estruturada em 6 capítulos como segue. O Capítulo 1 apresenta uma introdução a linguagens de programação, bem como um breve histórico e conceitos importantes como compiladores e interpretadores. Também é visto o ambiente de programação que será utilizado por toda a disciplina chamado NetBeans. O Capítulo 2 apresenta os conceitos básicos da linguagem C como declarações de bibliotecas, função principal, comandos de repetição e comandos condicionais. O Capítulo 3 apresenta duas estruturas básicas encontradas em todas as linguagens de programação chamadas de vetores e matrizes. O Capítulo 4 apresenta funções, onde são abordadas as funções pré-definidas pela linguagem bem como a definição e uso de funções definidas pelo programador. O Capítulo 5 apresenta com registros em C são definidos e como eles podem ser utilizados para a resolução de problemas. O Capítulo 6 apresenta a manipulação de arquivos em C, desde a sua definição, leitura e escrita bem como outras funções importantes quanto à utilização de arquivos.

O estudo de linguagens de programação é focado na utilização de exemplos e resolução de problemas simples. Cabe ressaltar que a disciplina é prática e necessita de dedicação no desenvolvimento das atividades. Cada atividade prevista neste livro pode ser escrita de várias formas diferentes, se utilizando de diferentes comandos da linguagem C e espera-se que ao final da disciplina os alunos sejam capazes de apresentar, para cada atividade contida neste livro, pelo menos duas soluções.

ENTENDA OS ÍCONES

ATENÇÃO: faz uma chamada ao leitor sobre um assunto, abordado no texto, que merece destaque pela relevância.

INTERATIVIDADE: aponta recursos disponíveis na internet (sites, vídeos, jogos, artigos, objetos de aprendizagem) que auxiliam na compreensão do conteúdo da disciplina.

SAIBA MAIS: traz sugestões de conhecimentos relacionados ao tema abordado, facilitando a aprendizagem do aluno.

TERMO DO GLOSSÁRIO: indica definição mais detalhada de um termo, palavra ou expressão utilizada no texto.

SUMÁRIO

APRESENTAÇÃO ·5

UNIDADE 1 – INTRODUÇÃO A LINGUAGENS DE PROGRAMAÇÃO ·10

Introdução ·12

1.1 Introdução as linguagens de programação ·13

1.2 Compilador ·16

1.3 Interpretador ·18

1.4 Linguagem C ·20

1.5 Ambiente de programação ·22

UNIDADE 2 –LINGUAGEM C ·32

Introdução ·34

2.1 Introdução as linguagens de programação ·35

2.2 Estruturas condicionais ·44

2.3 Comandos de repetição ·48

2.4 Exemplos ·53

UNIDADE 3 – VETORES E MATRIZES ·56

Introdução ·58

3.1 Declaração de vetores e matrizes ·59

3.2 Acesso aos elementos ·65

3.3 Acesso aos vetores e matrizes ·67

UNIDADE 4 – FUNÇÕES ·70

Introdução ·72

4.1 Declaração de funções ·73

4.2 Recursividade ·82

UNIDADE 5 – REGISTRO ·85

Introdução ·87

5.1 Declaração ·88

5.2 Acesso aos registros ·90

5.3 Exemplos de *Struct* ·94

▷ **UNIDADE 6 – ARQUIVOS ·98**
Introdução ·100
6.1 Abrindo e fechando arquivos ·101
6.2 Lendo arquivos ·104
6.3 Escrevendo em arquivos ·106
6.4 Outras funções ·108

▷ **CONSIDERAÇÕES FINAIS ·112**

▷ **REFERÊNCIAS ·113**

▷ **APRESENTAÇÃO DOS PROFESSORES ·114**

1

INTRODUÇÃO A LINGUAGENS DE PROGRAMAÇÃO

INTRODUÇÃO

Linguagens de modo geral são utilizadas para comunicação, por exemplo, uma linguagem natural como o Português é utilizada para comunicação entre pessoas. Todos aprendemos pelo menos uma linguagem natural desde que nascemos. Mas então porque que não conseguimos utilizar uma linguagem como o português para a programação de computadores? O problema encontrasse na ambiguidade das linguagens naturais, ou seja, uma sentença escrita em português, por exemplo, pode representar diferentes significados, dependendo do contexto ou até mesmo possuir diferentes interpretações. Veremos a seguir que linguagens de programação também possuem sintaxe e semântica como as linguagens naturais, no entanto, elas não são ambíguas e podem ser traduzidas também em sentenças matemáticas, semelhantes às que vimos em Lógica Matemática.

Neste capítulo será introduzido os principais conceitos que envolvem as linguagens de programação, como compiladores e interpretadores. Também abordaremos o ambiente necessário para a execução de um programa simples escrito na linguagem de programação chamada C. Existem inúmeras linguagens de programação disponíveis. Neste livro abordaremos a linguagem C que é uma das linguagens mais conhecidas e utilizadas para o desenvolvimento de programas de computador. Ao final desse capítulo, você deverá ser capaz de entender o que são as linguagens de computador, citar alguns exemplos de linguagens, configurar um ambiente de programação para a linguagem C e editar e compilar um programa simples na linguagem C.

1.1
INTRODUÇÃO ÀS LINGUAGENS DE PROGRAMAÇÃO

Existem relatos de linguagens de programação muito antes de 1940, que foi quando as primeiras linguagens de programação modernas e os computadores começaram a surgir. As linguagens de programação no começo eram códigos matemáticos. A ideia de uma linguagem de programação era um código especializado para uma aplicação. As linguagens de programação surgiram da evolução da lógica matemática, no qual abstrai conceitos complexos da matemática e podia ser utilizada para resolver problemas específicos.

Dois conceitos importantes nas linguagens de programação são: Sintaxe e Semântica. Podemos fazer uma analogia com uma linguagem natural como o Português. Por exemplo, considerando a seguinte sentença:

"O mar é azul"

Podemos dizer que a sintaxe da sentença acima corresponde em como a sentença está escrita e a semântica corresponde ao significado da sentença. Neste caso, ambas, sintaxe e semântica, estão corretas. Considere agora a seguinte sentença:

"O mar é quadrado"

Novamente temos uma sentença com a sintaxe correta. No entanto, a semântica está incorreta, pois a sentença não faz sentido.

No caso de linguagens de programação, também temos, para cada linguagem, uma sintaxe e uma semântica. A sintaxe corresponde a como está escrito e a semântica ao significado. Quando encontramos algum erro em algum software que usamos no nosso dia a dia, encontramos um erro semântico. Assim, caso um programa de computador consiga ser executado ele é sintaticamente correto, ou seja, o programa pertence à linguagem de programação em questão, mas eventualmente pode apresentar erros de ordem semântica.

Os primeiros computadores foram criados a partir de 1940 onde eram programados na linguagem *Assembly*, também chamada de linguagem de montagem ou linguagem de baixo nível. Essa linguagem é usualmente considerada difícil pois o programador precisa conhecer a estrutura da máquina para usá-la, ou seja, ela é associada a estrutura de uma CPU (Unidade Central de Processamento). Apesar da linguagem *Assembly* ser associada como baixo nível, ela ainda precisa ser transformada em linguagem que a máquina entenda, por exemplo, código binário. Atualmente *Assembly* é usada para manipulação direta de hardware e para sistemas que necessitem de performance crítica como sistemas de tempo real e sistemas embarcados. No entanto, seu uso está cada vez mais restrito pois, linguagens modernas possuem inúmeras vantagens sobre elas.

Algumas das principais linguagens da década de 40 foram:

- 1943: *ENIAC Coding System*
- 1949: C-10

A partir de 1950 começaram a surgir as primeiras linguagens de programação modernas como FORTRAN (*FORmula TRANslator)*, LISP, (*LISt Processor*) COBOL (*COmmon Business Oriented Language*), entre outras. Essas linguagens também são conhecidas como linguagens de alto nível, pois possuem uma abstração maior que as linguagens de baixo nível, ou seja, um comando escrito em uma linguagem de alto nível geralmente é representado por vários comandos em uma linguagem de baixo nível. A linguagem FORTRAN, foi uma das primeiras linguagens de programação de alto nível e foi proposta e implementada para auxiliar os programadores na codificação de problemas técnicos e científicos cuja solução requer a utilização de computadores eletrônicos. O LISP é uma linguagem de alto nível, criada por John McCarty em 1959, tendo em vista facilitar a programação de sistemas de raciocínio baseados em lógica. COBOL é uma das linguagens de programação mais antigas, pertencendo à segunda geração das linguagens de programação. É muito utilizada em aplicações voltadas ao mundo financeiro, devido à sua precisão e rapidez na aritmética de ponto flutuante, ou seja, possui grande precisão numérica. Ainda hoje temos muitos sistemas, principalmente financeiros que ainda usam programas em COBOL.

Algumas das principais linguagens da década de 50 foram:

- 1952: Autocode
- 1954: FORTRAN
- 1958: LISP
- 1958: ARGOL 58
- 1959: COBOL

Nas décadas de 60 e 70 começaram a surgir inúmeras linguagens de programação que são utilizadas até hoje. Foi a partir desse período que os paradigmas de linguagens de programação se estabeleceram e deram origem a linguagens poderosas como a, que veremos nesse livro, linguagem de programação C. O paradigma de programação que começou a ser amplamente utilizado foi o chamado Paradigma Estruturado que consiste no uso de rotinas e sub-rotinas de forma estruturada.

Algumas das principais linguagens das décadas de 60 e 70 foram:

- 1962: APL
- 1962: Simula
- 1964: BASIC
- 1970: Pascal
- 1972: C
- 1972: Smalltalk
- 1972: Prolog
- 1973: ML
- 1978: SQL

Já a partir da década de 80 as linguagens se consolidaram e também começou a se popularizar outros paradigmas de programação, como a programação orientada a objetos que consiste na composição e interação entre unidades de código chamadas objeto. Também a partir desse período os computadores começaram a ficar mais populares e baratos, e obviamente, as linguagens de programação começaram a se popularizar.

Algumas das principais linguagens da década de 80 foram:

- 1983: Aida
- 1983: C++
- 1985: Eifell
- 1987: Perl

Com a popularização dos computadores pessoais e o surgimento e popularização da Internet, a partir dos anos 90, as linguagens começaram a se adaptar e o desenvolvimento de software começou a tornar-se cada vez mais importante. As linguagens de programação evoluíram e começaram a ser utilizadas para o desenvolvimento de sistemas complexos, para a substituição de sistemas manuais pelos informatizados.

Algumas das principais linguagens da década de 90 foram:

- 1990: Haskell
- 1991: Python
- 1991: Java
- 1993: Ruby
- 1993: Lua
- 1995: JavaScript
- 1995: PHP

Como pode-se observar existe uma enorme variedade de linguagens de programação. Algumas são mais fáceis outras mais complexas, algumas são adaptadas para resolver determinados tipos de problema outras focam em características específicas como desempenho. A escolha da linguagem está, na maioria das vezes, relacionada ao tipo de problema que se pretende resolver, por exemplo, caso precisamos desenvolver uma loja virtual online poderíamos optar pela linguagem PHP ou Java. Para um programador o importante não é aprender apenas uma linguagem ou focar em uma linguagem, o importante é compreender os fundamentos e técnicas de programação. O domínio de uma linguagem específica se dá com treino e persistência. Um bom programador só é bom porque ele ou ela programa muito.

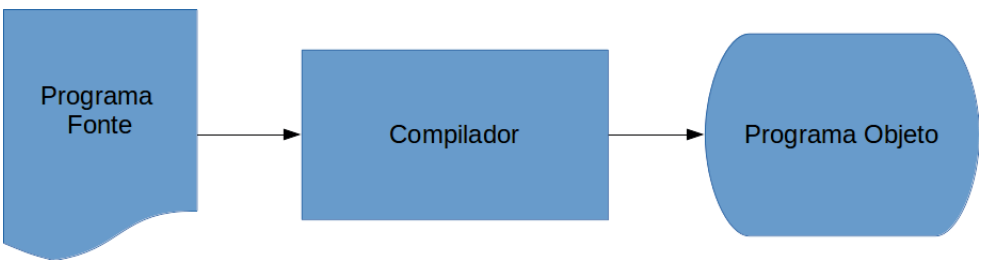
1.2 COMPILADOR

Os programas de computador escritos em alguma linguagem de programação precisam ser compilados ou interpretados (veremos os interpretadores na próxima seção) para serem executados pelo processador. Os compiladores são programas especializados em converter as instruções escritas na linguagem de programação em linguagem de máquina. Desta forma, um programa escrito em uma linguagem de programação precisa ser compreendido pelo computador para que possa executar. Não precisamos entender como os compiladores fazem a tradução, basta sabermos como uma determinada linguagem de programação funciona (sintaxe e semântica) que o compilador traduzirá para uma linguagem que possa ser executada.

Definição: Compiladores são tradutores de linguagens de programação para uma linguagem que seja possível de ser executada por um computador, também chamada de linguagem objeto.

A Imagem 1 apresenta a ideia de um compilador. Temos como entrada para o compilador o programa fonte escrito em alguma linguagem de programação de alto nível, por exemplo, a linguagem de programação C. O compilador analisa o programa fonte e traduz para um programa objeto (linguagem de máquina) que será executado pelo computador.

Imagem 1 – Compilador



Fonte: Autor.

Uma tarefa importante do compilador é informar os erros ao usuário. Atualmente, os compiladores são muito amigáveis e possuem um bom tratamento de erros informando, por exemplo, qual o tipo de erro e onde (qual linha) do programa fonte o erro aparece. Quando dizemos que um programa de computador possui portabilidade, ou seja, pode ser executado em diferentes plataformas (Windows, Linux, Mobile, etc) podemos dizer que o compilador consegue traduzir uma linguagem fonte para diferentes linguagens objeto.

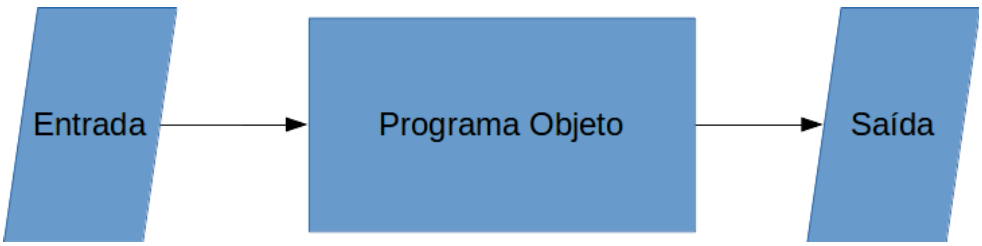
As principais propriedades de um compilador são:

• Geração do código objeto correto.
• Geração de programa objeto independente do tamanho do programa fonte, desde que a quantidade de memória permita.
• Deve mostrar as mensagens de erro de forma clara e onde está errado.

Algumas características como velocidade de compilação e tamanho do compilador já não são mais um problema para os compiladores modernos.

A Imagem 2 apresenta o processamento de um programa objeto. Basicamente, o programa objeto consiste no programa em que o usuário utilizará. O programa aceita entradas, processa e gera as saídas de acordo com o programa.

Imagem 2 – Programa Objeto



Fonte: Autor.

Observe que, mesmo um programa tendo sido compilado corretamente isso não garante que a saída está correta. Erros podem acontecer e envolvem a semântica do programa. Compiladores também são largamente utilizados em outras áreas, por exemplo, em editores de texto. Quando digitamos um texto em um editor, e o mesmo sublinha uma palavra ou uma sentença, significa que o processador utiliza um compilador que, a medida em que o usuário está digitando, o compilador está verificando erros ortográficos e gramaticais.

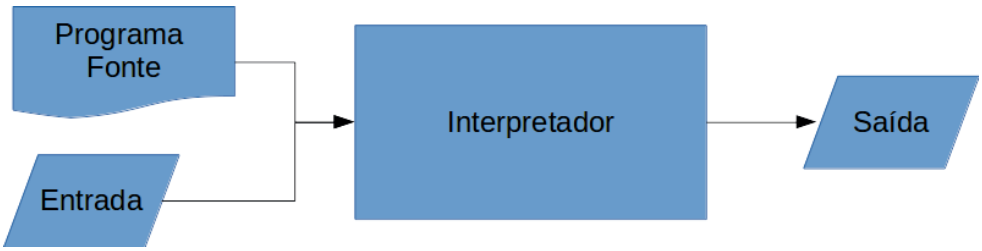
1.3 INTERPRETADOR

Existem duas formas de traduzir um código de alto nível para um código que o computador compreenda e consiga executar: (i) utilizando um compilador, ou (ii) utilizando um interpretador. A tradução do programa escrito em uma linguagem de alto nível gera um novo código chamado de código objeto.

O compilador traduz todo o código fonte de alto nível para o código objeto. Já o interpretador vai traduzindo o código linha por linha à medida em que ele vai sendo executado. Desta forma, dizemos que o programa é interpretado.

A Imagem 3 apresenta como funciona o interpretador. As entradas são o programa fonte e a entrada do usuário, sendo que o interpretador produz a saída correspondente.

Imagem 3: Interpretador

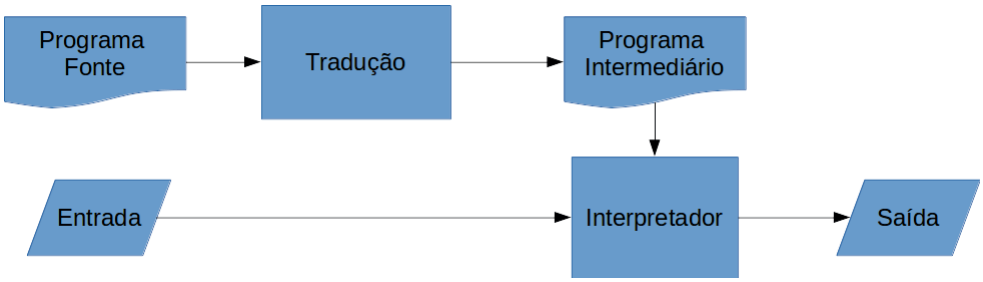


Fonte: Autor.

Com um interpretador, não há tradução mas sim interpretação do código fonte e de sua respectiva entrada. Desta forma a interpretação é um processo mais lento e frequentemente requer mais espaço. Uma linguagem interpretada que ficou muito popular foi Java e a sua principal crítica era com relação a lentidão dos programas gerados. Com relação ao Java, a linguagem inicialmente não era apenas interpretada, ela era chamada de híbrida pois ela gerava um código intermediário chamado de *byte code* que facilitava o processo de interpretação.

Na Imagem 4 vemos o Interpretador com mais detalhe. Em vez de traduzir de uma vez o programa fonte para então executá-lo, são traduzidas apenas pequenas unidades básicas do programa para executá-las imediatamente (não mantém o código-objeto para sempre). Este é um enfoque de interpretação, por isso dizemos que não há tradução, ou seja, o interpretador refere-se não só ao programa que simula a execução do código Intermediário, mas a todo o processo.

Imagem 4: Interpretador



Fonte: Autor

1.4 LINGUAGEM C

Estudaremos neste livro a linguagem de programação C, que atualmente continua sendo uma das linguagens mais populares do mundo. A linguagem de programação C foi criada por Dennis Ritchie, em 1972, no centro de Pesquisas da Bell Laboratories. Existem diversos índices de popularidade das linguagens de programação, um muito conhecido é o TIOBE .

INTERATIVIDADE: Acesse TIOBE em: https://www.tiobe.com/tiobe-index/

A linguagem de programação C começou a ser utilizada para a reescrita do Sistema Operacional UNIX, que até então era escrito na linguagem Assembly. Nos anos 70 o sistema operacional UNIX passou a ser utilizado nas universidades e isso impulsionou o desenvolvimento de compiladores C e a popularização, não apenas do sistema operacional UNIX, mas também da linguagem C que era utilizada para a programação de sistemas para o UNIX.

A linguagem de programação C é uma linguagem de propósito geral, sendo adequada à programação estruturada. Ela é muito utilizada para escrever compiladores, analisadores léxicos, bancos de dados, editores de texto, entre outros programas. No entanto, temos a linguagem C++ que é baseada na linguagem C, tendo uma sintaxe bem parecida, mas que possibilita a programação usando o paradigma orientado a objetos. É uma linguagem que possui portabilidade, podendo ser compilada para diferentes sistemas operacionais.

A linguagem de programação C é considerada uma linguagem de médio nível, pois permite a manipulação de bits, bytes e endereços de memória. No entanto, também possui muitos elementos de alto nível, sendo uma linguagem bem versátil.

C é uma linguagem estruturada, significa que como tal linguagem podemos ter procedimentos ou funções. Além das funções, podemos estruturar o código em blocos de código, que são um grupo de comandos de programa conectado logicamente que é tratado como uma unidade, por exemplo:

```
1  if( x < 10 ) {
2     printf("muito pouco, tente novamente\n");
3     scanf("%d", &x);
4  }
```

Neste exemplo temos que os dois comandos (*printf* e *scanf*) serão executados apenas se x for menor que 10. Então estes comandos, juntos com as chaves representam um bloco de código. Assim, o que está entre chaves representa um bloco. Veremos durante os próximos capítulos os detalhes da sintaxe e semântica da linguagem.

Para criar, compilar e executar os programas na linguagem de programação C usaremos um IDE (*Integrated Development Environment)* chamado NetBeans. Ele fornece um ambiente integrado para o desenvolvimento C e será também utilizado nas disciplinas futuras de programação com outras linguagens.

INTERATIVIDADE: Acesse NetBeans em:
https://netbeans.org/

1.4.1 Meu Primeiro Programa em C

Para demonstrar o ambiente de programação que utilizaremos durante a disciplina vamos seguir como exemplo o programa simples e popularmente utilizado como primeiro exemplo de qualquer linguagem de programação. O programa chama-se "Olá Mundo".

O código abaixo mostra um exemplo de programa escrito na linguagem de programação C. A linha 1 apresenta a biblioteca stdio.h que representa um conjunto de comandos já definidos. Essa biblioteca é utilizada sempre que temos leitura de dados (entrada) ou escrita (saída) de dados. A linha 2 apresenta a função principal chamada *main*(). Todo programa em C terá uma função *main*() que representa o início do programa, ou seja, onde o programa começará a ser executado. A linha 3 utiliza o comando printf para exibir a mensagem "Olá Mundo" na tela. O que está depois das barras (//) são os comentários do programa. Apesar de não serem necessários para a execução eles são úteis para identificar o que cada linha do programa faz. Durante a disciplina, usaremos como padrão, para cada linha com algum comando também colocarmos um comentário explicando a linha.

```
1  #include <stdio.h> //Biblioteca padrão de entrada e saída
2  main() { //Função principal chamada main
3    printf("Olá Mundo"); //comando que exibe uma mensagem na tela
4  }
```

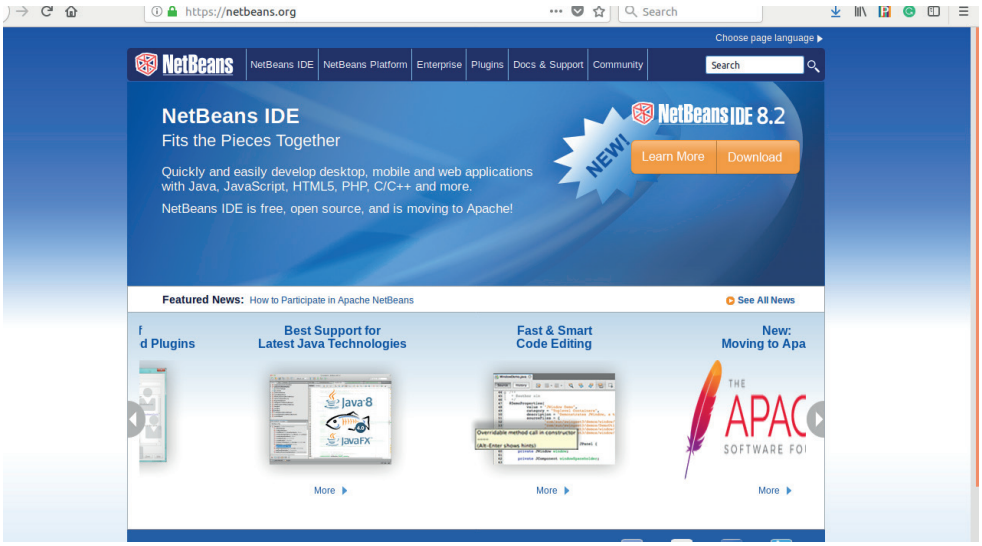
Para a próxima seção usaremos esse programa para exemplificar o uso do ambiente de programação NetBeans, no qual usaremos durante toda a disciplina.

1.5
AMBIENTE DE PROGRAMAÇÃO

Para escrever programas em C, a princípio, precisamos de um editor (que pode ser qualquer editor de texto) e um compilador instalado. No entanto, atualmente é comum o uso de IDEs para a programação. As IDEs são ambientes integrados que possuem editor de texto, ajudam na correção da sintaxe, na utilização de bibliotecas e na compilação dos programas.

A Imagem 5 apresenta a tela inicial do site do Netbean acessada em https://netbeans.org. Nela você poderá escolher a linguagem (canto superior direito). Cabe ressaltar que o NetBeans é de graça, código aberto e multiplataforma.

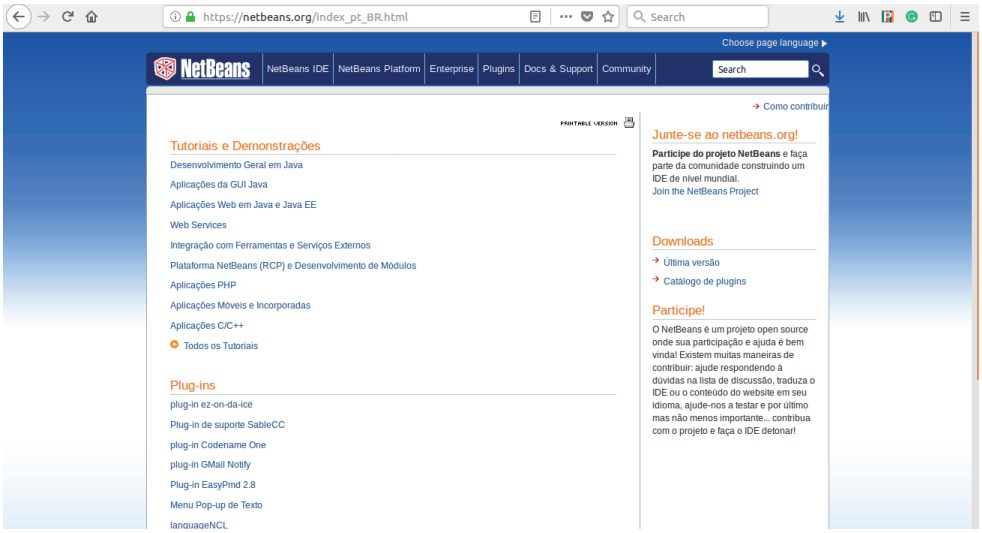
Imagem 5 – Tela inicial do site do NetBeans



Fonte: NetBeans. Disponível em: <https://netbeans.org>. Acesso em: 08/11/2018.

Quando selecionamos a opção para o português, a página apresenta, no lado esquerdo, vários tutoriais e demonstrações e no lado direito a opção de download, conforme visto na Imagem 6. Observa-se que o Netbeans possui suporte a outras linguagens além do C como Java, C++, JavaScript e PHP.

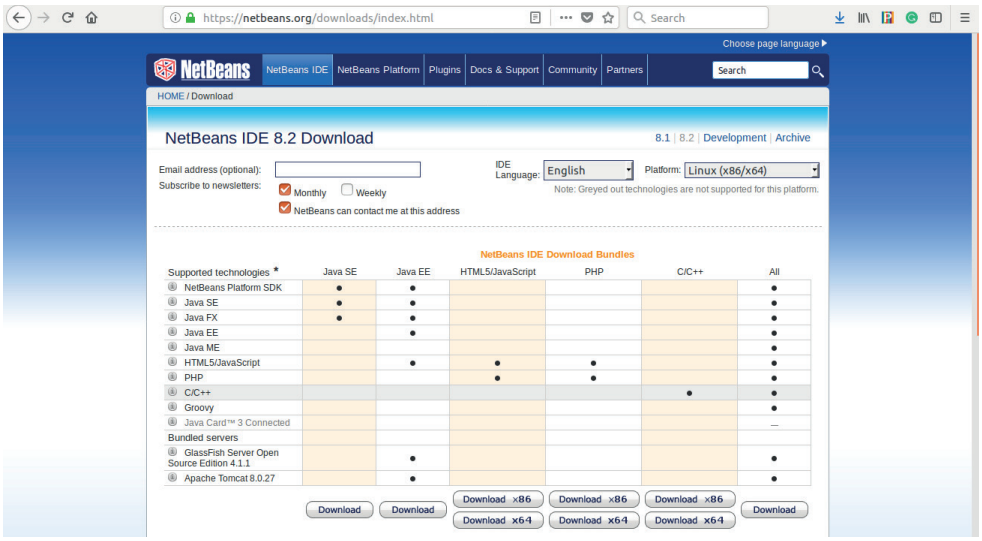
Imagem 6 – Tela inicial do site NetBeans em Português



Fonte: NetBeans. Disponível em: <https://netbeans.org>. Acesso em: 08/11/2018.

A Imagem 7 apresenta a tela de download do NetBeans. O NetBeans está disponível em várias linguagens, inclusive o português e também nos 3 principais sistemas operacionais: Windows, Linux e Mac OS. Durante a disciplina e nos exemplos usarei o NetBeans em Inglês, pois a grande maioria de IDEs e ferramentas de desenvolvimento são utilizadas na indústria de software em inglês.

Imagem 7 - Tela de download do NetBeans



Fonte: NetBeans. Disponível em: <https://netbeans.org>. Acesso em: 08/11/2018.

1.5.1 NetBeans

Entre as principais características do NetBeans podemos listar: é de código aberto; de graça; possui um editor de código fonte integrado e rico em recursos para o desenvolvimento não apenas de aplicações em C ou Java mas também para aplicações Web; está sendo utilizado por diversas empresas no mundo todo; possui suporte a banco de dados; e é independente de plataforma, podendo rodar em Windows, Linux e Mac OS.

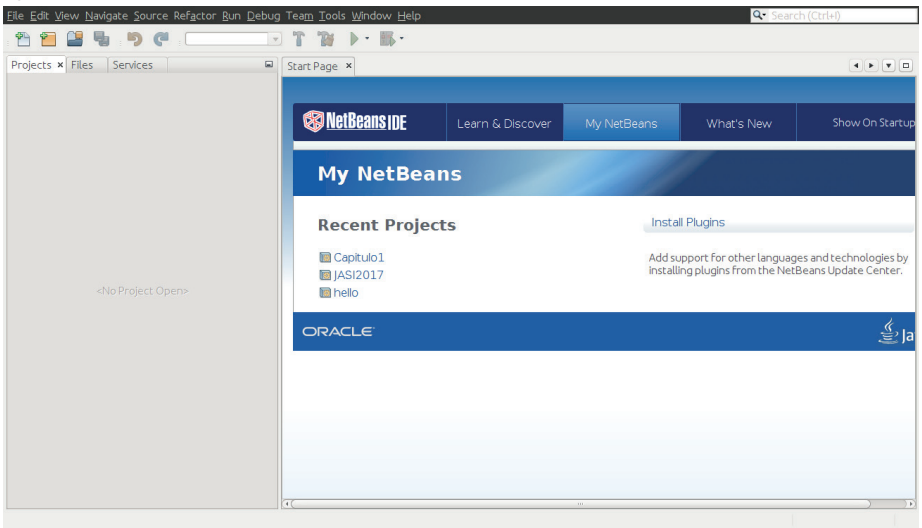
A Instalação do NetBeans depende do sistema operacional que será utilizado. Nos sistemas Linux e Mac OS a instalação pode ser feita também via linha de comando e de forma muito simples. Por exemplo, no Linux (distribuição Ubuntu) usa-se o comando:

sudo apt-get install netbeans

Onde será instalado e configurado o NetBeans e todas as suas dependências. No site do NetBeans encontramos instruções para a instalação em cada um dos sistemas operacionais. No entanto, um dos sistemas operacionais mais simples de ser utilizado para desenvolvimento de software e totalmente gratuito é o Linux Ubuntu.

A Imagem 8 apresenta a tela inicial do NetBeans.

Imagem 8 – Tela inicial do NetBeans versão 8.2



Fonte: Autor.

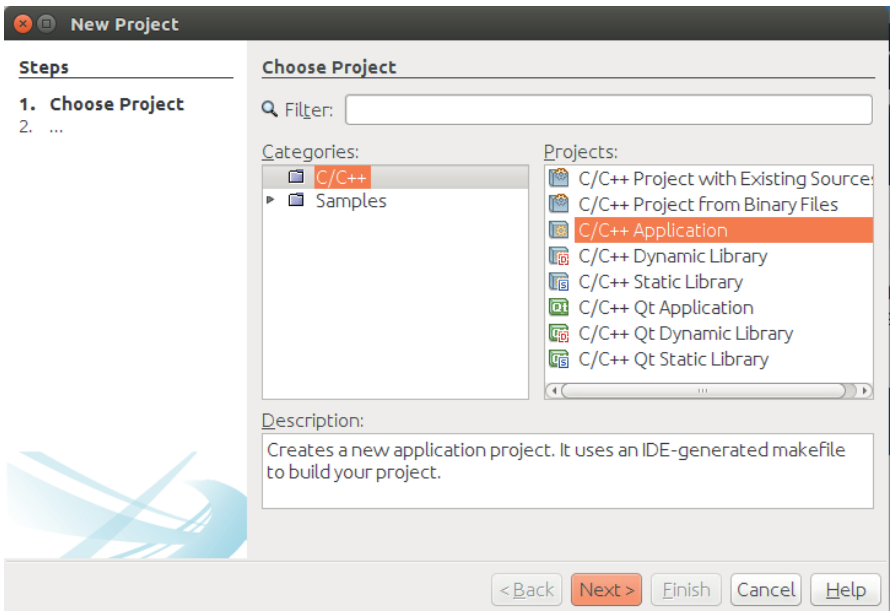
Temos no lado esquerdo a parte dos projetos, onde organizaremos os programas em projetos. No lado direito será visualizado o editor. O menu superior apresenta as opções como novo projeto, salvar, executar, etc.

1.5.2 Criando meu Primeiro Programa

Para o primeiro programa utilizaremos o exemplo do Olá Mundo visto na Seção 1.4.1. Primeiro devemos abrir o NetBeans. Quando o NetBeans carregar e a tela inicial for apresentada (conforme Imagem 8) estaremos prontos para começar a criar o nosso primeiro programa.

Com o NetBeans aberto selecione *File* >> *New project* (Arquivo >> Novo Projeto). A Imagem 9 apresenta a tela para a criação de um novo projeto. Selecione a opção C/C++ *Application* e clique em *Next* (Próximo).

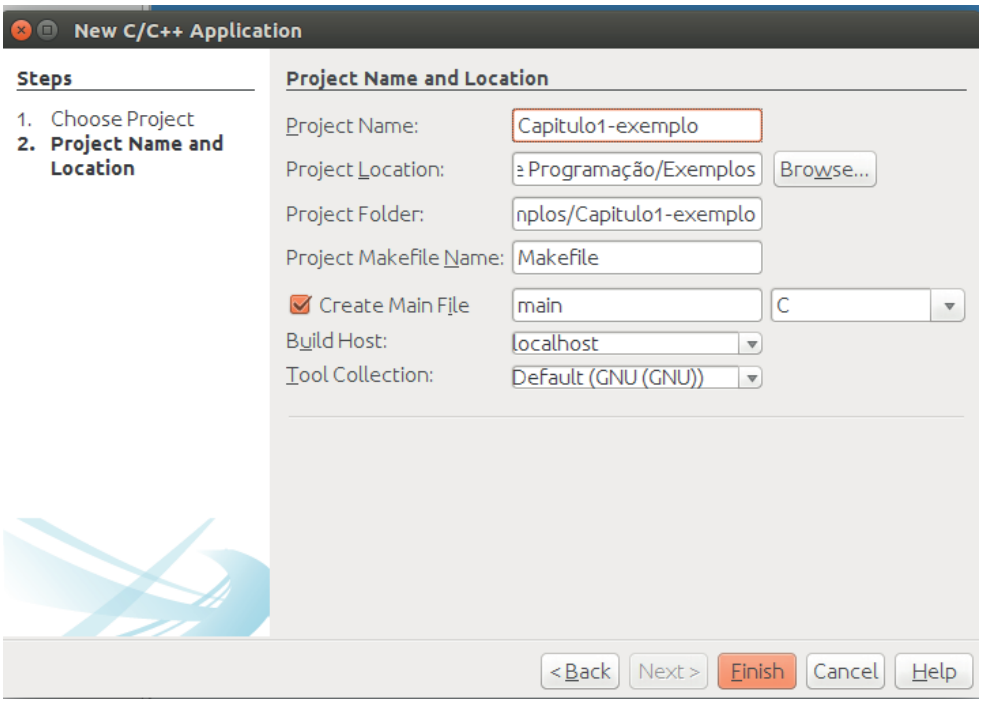
Imagem 9 – Tela de criação de um novo projeto



Fonte: Autor.

Próximo passo é atribuir um nome ao projeto. Neste exemplo usaremos o nome Capitulo1-exemplo. Também podemos selecionar a localização (diretório) onde o projeto ficará armazenado no computador. A Imagem 10 apresenta a tela para selecionar o nome e localização do programa. Após selecionar o nome e localização é preciso clicar em Finish (Concluído).

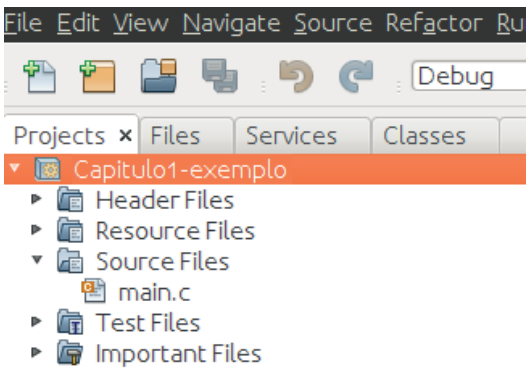
Imagem 10 – Tela para seleção do nome do projeto



Fonte: Autor

O novo projeto irá ser criado. O principal diretório que trabalharemos será o Source Files (Arquivos Fontes), que contém os arquivos fontes (programas C). A Imagem 10 apresenta a árvore de diretórios do projeto criada com o arquivo main.c, que representa o arquivo principal do programa.

Imagem 11 – Tela de criação de um novo projeto

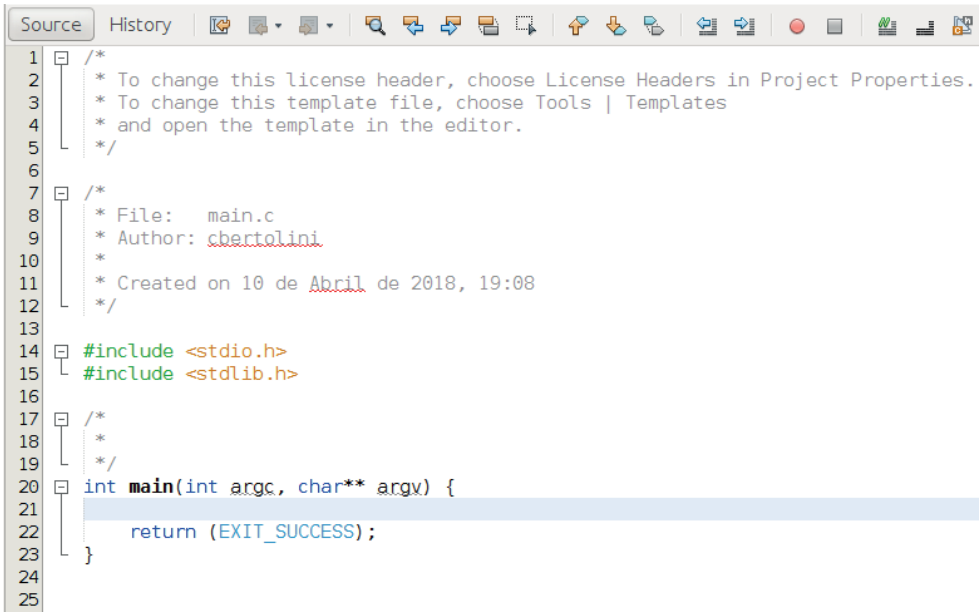


Fonte: Autor.

Selecionando o arquivo *main.c*, o Netbeans, por padrão apresenta um *template* básico para o início de um programa em C. A Imagem 11 apresenta o arquivo *main.c* criado pelo NetBeans. O que está entre /* */ representa os comentários do programa. Por padrão o primeiro bloco de comentário é para colocar informações sobre a licença de distribuição do programa. Para a disciplina podemos excluir

o primeiro bloco pois criaremos apenas programas C para uso na disciplina. O segundo bloco de comentários é para colocarmos o nome do arquivo principal, autor e data que foi criado. Este cabeçalho usaremos em todos os nossos exemplos e deverá ser utilizado em todos os exercícios da disciplina.

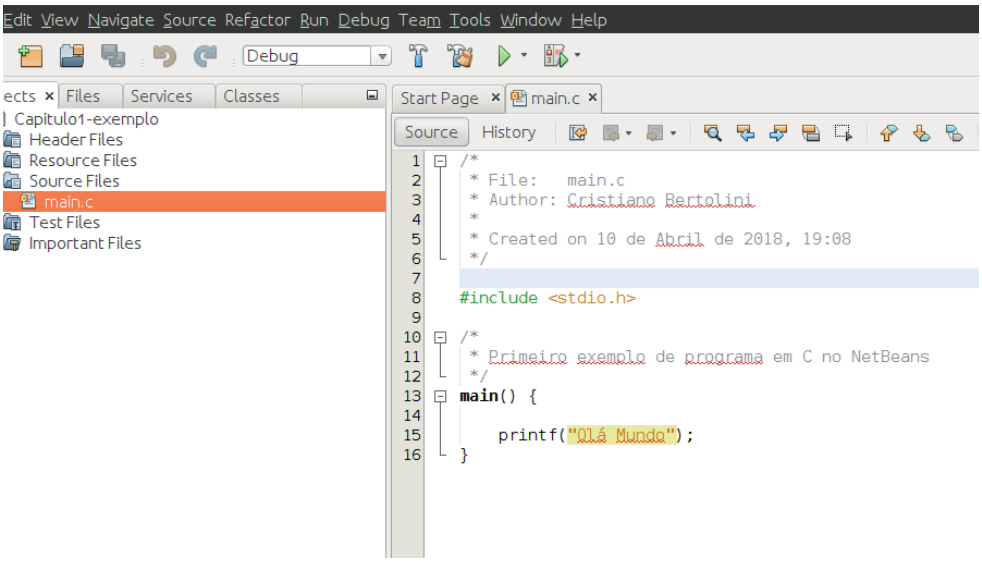
Imagem 12 - Programa *main.c* criado pelo NetBeans



Fonte: Autor.

Agora podemos editar o arquivo *main.c*. A Imagem 13 apresenta o exemplo Olá Mundo já visto neste capítulo.

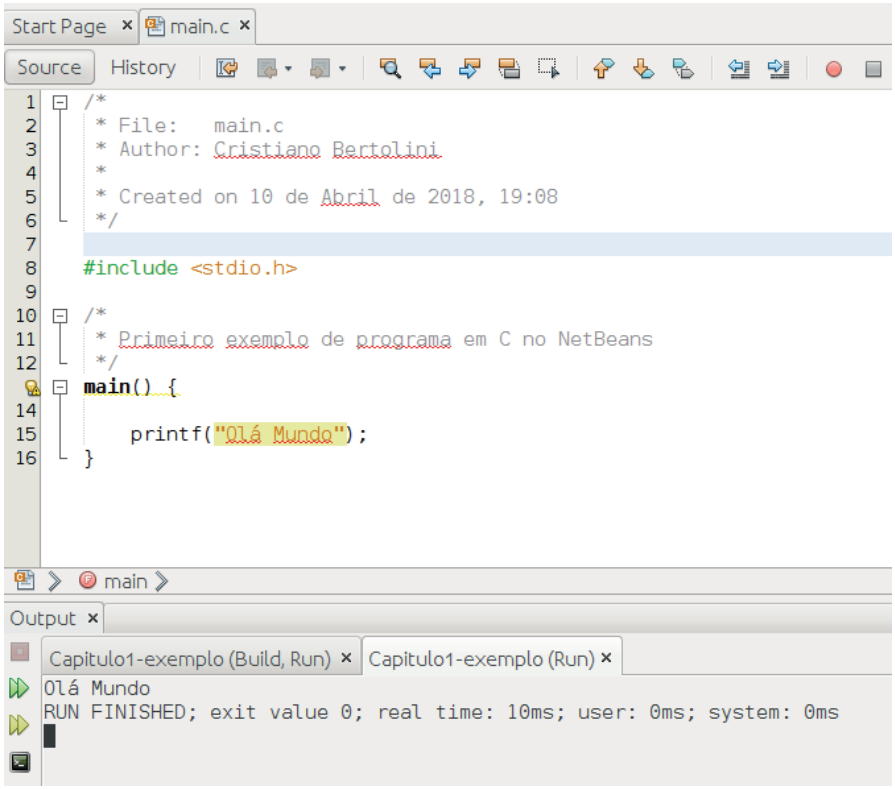
Imagem 13 - Programa Olá Mundo



Fonte: Autor.

Com o programa pronto, agora podemos executá-lo. Para executar o programa clicamos na seta verde localizada na primeira barra horizontal de ícones do NetBeans. Também podemos acessar a opção via menu *Run* >> *Run Project* (Executar >> Executar Projeto).
A Imagem 14 apresenta o programa Olá Mundo com a sua respectiva execução. Observamos que o resultado foi a exibição da mensagem "Olá Mundo" com a mensagem que a execução terminou e algumas métricas de tempo de execução do programa.

Imagem 14 - Resultado da execução do programa Olá Mundo



Fonte: Autor.

1.5.3 Criando Exemplos em um Único Projeto

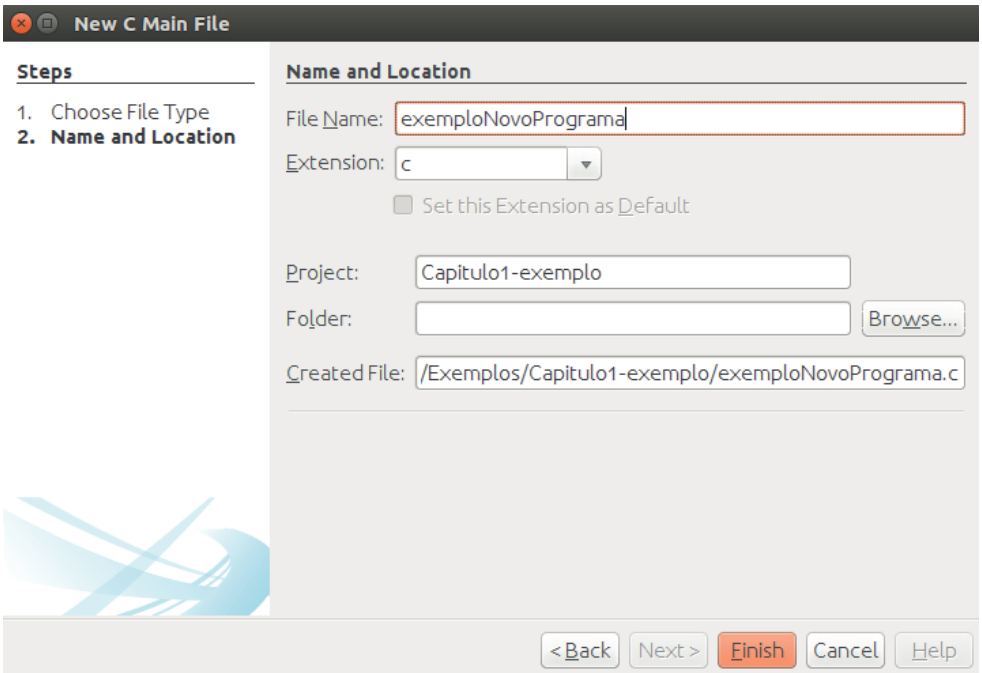
Uma vez que criamos o projeto para uma aplicação e precisamos criar outro programa, temos que realizar todos os passos de criação de projeto vistos na seção anterior. No intuito de facilitar a criação de programas simples em C, veremos aqui uma estratégia para que possamos criar apenas um projeto para um conjunto de programas em C.

Uma característica de cada projeto é que teremos que ter apenas uma função *main*() (função principal), que será onde a execução do nosso programa irá começar. Assim, quando adicionarmos um novo programa ao projeto teremos que renomear a função main() do programa anterior e criarmos uma nova função main() no novo programa, como veremos a seguir.

Para criarmos um novo programa, temos que clicar com o botão direito no projeto e selecionar *New* >> *C Main File* (Novo >> Arquivo C Main). A Imagem 15

apresenta a tela de criação de um novo programa. Selecionamos o nome do novo programa, neste exemplo o programa se chamará exemploNovoPrograma.

Imagem 15 - Tela de criação de um novo programa C



Fonte: Autor.

Se executarmos o novo programa teremos um erro, pois o projeto apresenta dois arquivos com a função *main()*. Assim, temos que renomear a função main() para *main01()* (ou qualquer outro nome que quisermos) e executarmos novamente o programa.

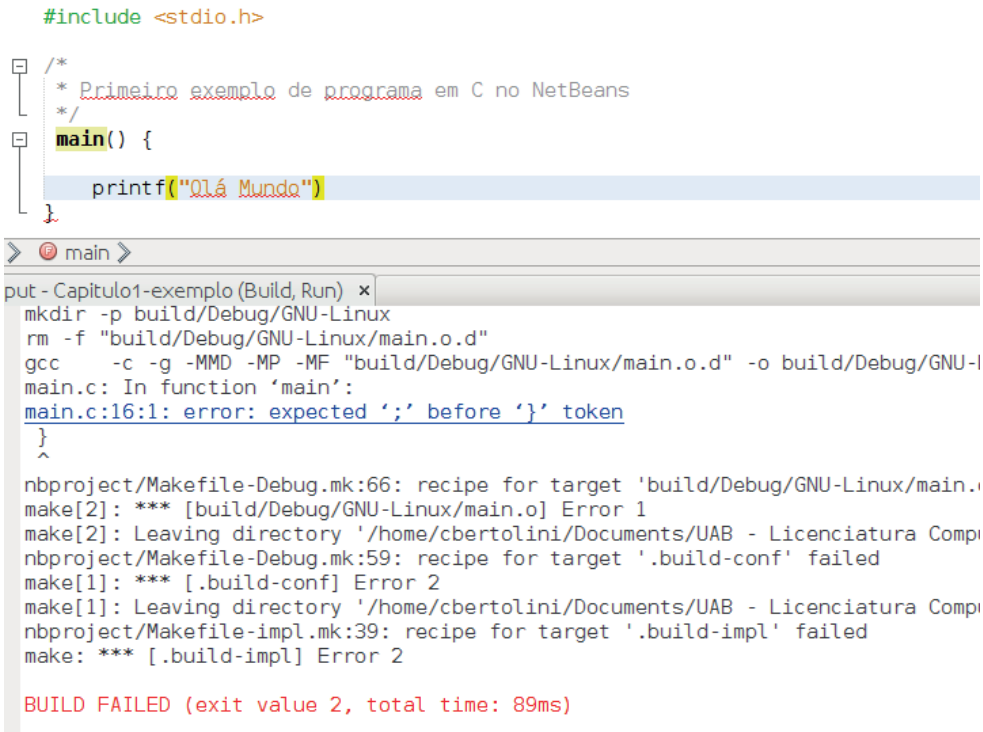
Cabe ressaltar que essa estratégia é apenas para facilitar a criação de vários programas em um único projeto. No entanto, é possível criar um novo projeto para cada programa.

1.5.3 Criando Exemplos em um Único Projeto

Uma das vantagens de se utilizar um IDE para o desenvolvimento de programas é o tratamento de erro. Por exemplo, suponhamos que, no exemplo anterior, tivéssemos esquecido de terminar o comando *printf* com o ponto e vírgula. Quando tentamos executar o programa ele não executará e apresentará um erro.

A Imagem 16 apresenta o exemplo Olá Mundo e logo abaixo o erro apresentado pelo IDE NetBeans. O erro em azul diz exatamente o problema do programa, ele espera um ; (ponto e vírgula) antes do fechamento do bloco. Além disso, várias outras informações são apresentadas para o usuário, que podem ser extremamente úteis quando temos programas grandes e complexos.

Imagem 16 - Identificação de erro em um programa C



Fonte: Autor.

ATIVIDADES - UNIDADE 1

Pesquise e responda as seguintes perguntas no ambiente virtual Moodle:

1. O que são linguagens de programação?

2. Faça uma tabela comparativa entre Compiladores e Interpretadores.

3. Instale o NetBeans em seu computador e execute o programa Olá Mundo. Após altere a mensagem "Olá Mundo" para "Olá <meu nome>", onde você deve colocar o seu nome. Salve um *print screen* da tela com o seu programa e o resultado da execução.

2

LINGUAGEM C

INTRODUÇÃO

Veremos neste capítulo o básico da linguagem de programação C. Já vimos no capítulo anterior um pequeno histórico sobre a linguagem e um exemplo básico, chamado de Olá Mundo. A linguagem C tornou-se uma das linguagens mais populares do mundo, e mesmo nos dias atuais, com várias linguagens modernas ela continua a ser muito utilizada pois é uma linguagem flexível, estruturada, portátil e eficiente. C é dita como uma linguagem de propósito geral, pois é possível desenvolver uma gama muito grande de programas para os mais diversos fins.

A Linguagem C possui uma gama muito grande de comandos e funções. Os conceitos, comandos, sintaxe e semântica que aprenderemos neste capítulo serão utilizados no decorrer de toda a disciplina e certamente serão utilizados em qualquer programa C desenvolvido. Cabe ressaltar que a sintaxe (como os programas são escritos) e a semântica (o significado dos comandos utilizados nos programas) devem ser seguidos. A utilização de um ambiente de programação como o NetBeans ajudará, e muito, a escrita dos programas (sintaxe), no entanto, a lógica e semântica sempre fica a cargo do programador.

Neste capítulo veremos o básico da linguagem C, como estrutura básica, tipos de dados, estruturas condicionais e de repetição. Cabe relembrar que na disciplina de Algoritmos estudamos os mesmos conceitos, no entanto aqui veremos estes conceitos com a sintaxe da linguagem C. Entendendo o conceito de algoritmos o aprendizado de uma linguagem de programação como C se torna mais intuitivo e simples. No final deste capítulo você deverá ser capaz de resolver problemas básicos em C que envolvam comandos de decisão e repetição.

2.1
INTRODUÇÃO ÀS LINGUAGENS DE PROGRAMAÇÃO

Os comandos básicos da linguagem C são importantes pois serão utilizados com muita frequência em vários programas escritos em C. É muito importante observar que o material não é exaustivo, ou seja, não possui todos os comandos da linguagem C. Cabe a cada programador estar sempre com algum guia ou livro de referência sobre a linguagem.

INTERATIVIDADE: ARAUJO, JAIRO. **Dominando a Linguagem C**. Rio de Janeiro: Ciência Moderna, 2004.

Para aprendermos os comandos básicos, vamos utilizar novamente o exemplo "Olá Mundo", visto no capítulo anterior:

```
1  #include <stdio.h> //Biblioteca padrão de entrada e saída
2  main() { //Função principal chamada main
3     printf("Olá Mundo"); //comando que exibe uma mensagem na tela
4  }
```

Na primeira linha (Linha 1) temos a definição de uma biblioteca chamada stdio.h. As bibliotecas em C constituem um conjunto de funções prontas que podem ser utilizadas para os mais devidos fins. Por exemplo, a biblioteca stdio.h é utilizada para funções de entrada e saída. No nosso exemplo, utilizamos o comando printf que imprime uma mensagem na tela. Esse comando está definido na biblioteca stdio.h.

A Tabela 1 apresenta uma lista das principais bibliotecas utilizadas na linguagem C. Além dessas existem muitas outras bibliotecas definidas.

Tabela 1 - Principais bibliotecas da linguagem C

Nome da Biblioteca	Descrição
alloc.h	Funções para gerenciamento de memória.
float.h	Funções para tratamento de números de ponto flutuante.
stddef.h	Funções para tratamento de tipos de dados.
math.h	Funções matemáticas.
stdlib.h	Funções utilizadas para o tratamento de conversões.
string.h	Funções para tratamento de strings.
time.h	Funções para tratamento de hora e data.

Fonte: (SCHILDT, H., 1997, p. 276.)

Na segunda linha (Linha 2) do programa Olá Mundo temos a nossa função principal *main*, que deverá estar contida em todos os programas que escreveremos durante a disciplina. Assim, cada programa em C tem uma função principal que deve ser nomeada *main*. A função main serve como ponto de início para a execução de um programa. Em geral, ela controla a execução direcionando as chamadas para outras funções no programa. A sintaxe básica da função *main* é:

```
1  main() { //função principal chamada main
2  }
```

No corpo da função *main*, ou seja, o que está entre chaves é onde escreveremos os comandos do nosso programa. No caso do exemplo Olá Mundo, temos o comando *printf* que imprime na tela uma mensagem. Ao final de cada comando em C temos que terminar ele com um ponto e vírgula.

2.1.1 Tipos de Dados

Programas de computador processam entradas e apresentam saídas como resultado. As entradas e saídas são dados, que podem ser: números, letras, texto, imagens, etc. Em um programa em C dados podem ser representados em duas formas:

- **Constantes**: representam dados que não são alterados durante toda a execução do programa. Por exemplo, se definirmos uma constante chamada ano com valor de 2018, temos que em toda a execução do programa a constante ano terá o mesmo valor.

- **Variáveis**: representam dados que são manipulados durante a execução do programa e alteram o seu valor. Por exemplo, se definirmos uma variável chamada media que representa a média aritmética entre dois números ela poderá assumir diferentes valores dependendo dos valores de entrada do programa. Na linguagem C não é permitido variáveis que comecem com dígito (números) e espaços.

Em geral, como boa prática de programação, definem-se variáveis com nomes em letra minúscula e constantes com letras maiúsculas.

A linguagem C define vários tipos de dados, os principais são representados na Tabela 2. Na primeira coluna temos o tipo e na segunda a escala de valores possíveis que uma variável pode assumir.

Tabela 2 - Tipos de dados em C

Tipo	Escala de valores
void	Nenhum
int	– 32768 a + 32767
char	– 128 a + 127
float	3.4 e– 38 a 3.4 e+ 38
double	1.7 e– 308 a 1.7 e+ 308

Fonte: (SCHILDT, H., 1997, p. 16.)

Os tipos de dados principais podem ser definidos como:

• **void**: é um tipo de dado genérico que pode representar qualquer coisa e geralmente é dito como nenhum.

• **int**: armazenam valores numéricos do tipo inteiro.

• **char**: armazenam valores do tipo carácter, por exemplo, letras do alfabeto. Observa-se que a escala é numérica, isto se dá pelo fato que internamente o computador representa cada carácter como um código (número).

• **float**: armazenam valores fracionários.

• **double**: armazenam valores fracionários maiores que o *float*.

Exemplos de declaração de variáveis em C:

```
1  main() { //função principal chamada main
2     int idade, x, y;  //declara 3 variáveis do tipo int
3     char tecla; //declara uma variável do tipo char
4     float salario, media; //declara 2 variáveis do tipo float
5  }
```

Exemplos de declaração de constantes:

```
1  main() { //função principal chamada main
2     #define ANO 2018; //constante chamada ANO com o valor de 2018
3     #define ICMS 0.17; //constante chamada ICMS com o valor de 0.17
4  }
```

Observa-se que podemos definir inúmeras variáveis de um mesmo tipo em um mesmo comando (mesma linha). Para a definição das constantes usamos o comando #define seguido do nome da constante e seu valor.

2.1.2 Entrada e Saída

Já vimos a função *printf* que foi utilizada para exibir a mensagem "Olá mundo" na tela. Assim, a função *printf* permite exibir informações formatadas na saída (tela). A função printf possui a seguinte sintaxe:

```
printf ("mensagem", arg1, arg2, argn);
```

Onde o que está entre aspas é a mensagem a ser exibida podendo ou não ser seguida por argumentos. Os argumentos podem ser variáveis, expressões ou valores e a formatação é dada pelos caracteres de controle. Os principais caracteres de controle são:

Tabela 3 – Caracteres de controle

Caractere de controle	Resultado
\b	O cursor retrocede uma coluna
\n	O cursor avança para uma nova linha
\"	Exibe uma única aspa
\'	Exibe um único apóstrofo
\\	Exibe uma única barra invertida

Fonte: (SCHILDT, H., 1997, p. 51)

Exemplos do uso da função *printf* com caracteres de controle:

```
1  #include <stdio.h>//define funções de entrada e saída
2  main() { //função principal chamada main
3     printf("\n Atenção!!! "); //imprime a mensagem: Atenção!!!
4     printf("\n Qual é a sua idade?"); //imprime a mensagem: Qual é a sua idade?
5  }
```

Neste exemplo, se não colocarmos o \n as duas mensagens aparecerão na mesma linha.

A função *printf* também permite que os campos de exibição sejam formatados. A seguir, as opções de formatação mais utilizadas em C:

Formatação	Resultado
%d	Exibe um número inteiro
%f	Exibe um número *float*
%c	Exibe um carácter
%s	Exibe uma *string*
%s%%	Exibe o carácter %

Fonte: (SCHILDT, H., 1997, p. 204).

Exemplos do uso da função *printf* com formatação dos campos:

```
1  #include <stdio.h>//define funções de entrada e saída
2  main() { //função principal chamada main
3     printf("\n 10 %% "); //saída:  "10  %"
4     printf("\n A média final é %f",3.48); //saída:  "A média final é 3.480000"
5     printf("\n Um carácter %c e um inteiro %d",'D',120); //saída:  "Um carácter D e um inteiro
   120"
6     printf("\n %s e um exemplo","Este"); //saída:  "Este e um exemplo"
7     printf("\n %s%d%%","Juros de ",4); //saída: "Juros de 4%"
8  }
```

A função printf geralmente é usada para representar as saídas do programa para o usuário. Outra função muito utilizada é a de leitura de dados. Na linguagem C podemos utilizar a função *scanf* para efetuarmos a leitura.

A função scanf() permite que um valor seja lido do teclado e armazenado em uma variável. Sua sintaxe consiste na definição da formatação da entrada de dados e seguida por uma lista de variáveis. A função *scanf* possui a seguinte sintaxe:

```
scanf ("formatação", var1, var2, varn);
```

A formatação é composta por especificadores de formato que indicam a quantidade e os tipos dos dados que serão lidos pela função. Sempre se utiliza o carácter & antes do nome da variável, pois este indica o endereço de memória da variável.

No exemplo abaixo, temos um programa em C que lê a idade e imprime o que foi lido na tela:

```
1  #include <stdio.h>//define funções de entrada e saída
2  main() { //função principal chamada main
3     int idade; //define uma variável inteira chamada idade
4     printf("\n Qual é a sua idade?\n"); //imprime na tela uma pergunta ao usuário
5     scanf("%d", &idade); //lê uma entrada do teclado
6     printf("\n Vc possui %d anos",idade); // imprime a mensagem com a idade lida
7  }
```

Observe que as mensagens, tanto antes da leitura quanto na apresentação do resultado do programa, são importantes para o usuário saber o que fazer. Como boa prática de programação, considerando esse exemplo, temos que as mensagens devem ser claras e a definição das variáveis também devem ter algum significado, pois facilita a leitura e entendimento do programa.

A função *scanf* possui os seguintes especificadores de formato:

Tabela 5 – Especificadores

Especificador	Resultado
%c	Representa um único carácter.
%o	Representa um número inteiro em octal.
%d	Representa um número inteiro em decimal.
%x	Representa um número inteiro em hexadecimal.
%u	Representa um número inteiro em base decimal sem sinal.
%ld	Representa um número inteiro longo em base decimal.
%f	Representa um número real de precisão simples.
%lf	Representa um número real de precisão dupla.
%s	Representa uma cadeia de caracteres (*string*).
%%	Representa um único sinal de porcentagem.

Fonte: (SCHILDT, H. 1997, p. 204).

2.1.3 Operadores Aritméticos

A linguagem C oferece operadores para as quatro operações aritméticas e também um operador especial usado para calcular o resto da divisão entre dois números inteiros.

A Tabela 6 apresenta os operados da linguagem C.

Tabela 6 - Operadores Aritméticos

Operador	Resultado
+	Soma de dois números quaisquer
-	Diferença entre dois números quaisquer
*	Produto de dois números quaisquer
/	Quociente da divisão de dois números
%	Resto da divisão de dois números inteiros

Fonte: (SCHILDT, H., 1997, p. 41).

Os operadores de soma, diferença e produto funcionam da mesma forma estabelecida na matemática básica. No entanto, o operador de divisão tem algumas diferenças, por exemplo, ele fornece o resultado inteiro somente quando ambos operadores forem inteiros. Já o operador do resto da divisão somente pode ser utilizado com números inteiros.

2.1.4 Operadores Relacionais

Na linguagem C não existe um tipo específico para a representação de valores lógicos (verdadeiro e falso). Entretanto, qualquer valor pode ser interpretado como um valor lógico. Podemos definir que:

- 0 (zero) representa falso e
- Qualquer outro valor representa verdade. Também é usual representar verdade com o valor 1 (um).

Para produzir um valor lógico (verdadeiro ou falso), usamos os operadores relacionais. Desta forma, podemos comparar dois valores e temos como resultado da avaliação de um operador relacional: 0 se a comparação é falsa e 1 se verdadeira.

A Tabela 7 apresenta os operadores relacionais.

Tabela 7 - Operadores Relacionais

Operador	Resultado
x == y	Verdade se x for igual a y
x != y	Verdade se x for diferente de y
x < y	Verdade se x for menor que y
x > y	Verdade se x for maior que y
x <= y	Verdade se x for menor ou igual a y
x >= y	Verdade se x for maior ou igual a y

Fonte: (SCHILDT, H., 1997, p. 44).

Considerando o exemplo abaixo temos que a saída (resultado) será o 1. Ou seja, falso e verdadeiro pois a primeira operação (10<1) é falso e a segunda é falsa (10>1).

```
printf ("%d %d", 10<1, 10>1);
```

2.1.5 Operadores Lógicos

A linguagem C oferece operadores lógicos que funcionam da mesma forma que os operadores da lógica matemática. A Tabela 8 apresenta os operadores lógicos.

Tabela 8 - Operadores Lógicos

Operador	Resultado
! x	Verdade se e só se x for falso
x && y	Verdade se e só se x e y forem verdade
x || y	Verdade se e só se x ou y for verdade

Fonte: (SCHILDT, H., 1997, p. 44).

No exemplo abaixo temos que a saída (resultado) será 1 1. Ou seja, ambas expressões são verdadeiras. A primeira é verdadeira pois qualquer valor representa verdadeiro. A segunda também é verdadeiro pois temos o operador ou, assim, basta um dos valores ser verdadeiro para a expressão ser verdadeira.

```
printf ("%d %d", 10&&1, 0||1);
```

2.1.6 Palavras Reservadas

As palavras reservadas da linguagem C são comandos de uso muito específicos dentro da linguagem. Já vimos a palavra *int* que representa o tipo de dado inteiro em um programa em C. Sendo uma palavra reservada ela não pode ser utilizada, por exemplo, como nome de variável ou constante. A Tabela 9 apresenta a lista com 32 palavras reservadas na linguagem C que só podem ser utilizadas para o seu propósito. Além dessas, foram introduzidas novas palavras reservadas, no entanto essas continuam sendo as mais utilizadas.

Tabela 9 - Palavras reservadas na linguagem C

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

Fonte: (SCHILDT, H., 1997, p. 10).

2.1.7 Comentários

Comentários em programas são explicações de como o programa funciona, documentação do programa, notas informativas e/ou explicativas. Na linguagem C podemos usar comentários de duas formas:

- **Bloco de comentários**: são representados por /* <bloco de comentário> */. Geralmente utilizado para uma explicação mais detalhada ou nota que pode incluir várias linhas.
- **Comentário de uma linha:** são representados por // <comentário>. Geralmente utilizado para comentários curtos de até uma linha.

Observa-se, que uma boa prática em programação é sempre comentar o seu programa.

2.2
ESTRUTURAS CONDICIONAIS

Estruturas condicionais, como o próprio nome diz, avaliam expressões e estas podem ser verdadeiras ou falsas. Para tanto, a linguagem C oferece alguns comandos de decisão. Nesta seção veremos quatro comandos de decisão: *if, if-else, switch, operador condicional.*

2.2.1 Comando: *if*

O comando *if* (significa em português "se") é utilizado para a tomada de uma decisão simples. Na linguagem C a sintaxe do if é definida como:

```
if (condição) instrução;
```

Onde a condição pode ser: verdadeira ou falsa. Caso a condição seja verdadeira a instrução será executada, caso a condição seja falsa a instrução não será executada.

O programa a seguir é um exemplo do uso do comando *if*. Neste exemplo a entrada é dada pelo usuário. O usuário entra com uma letra qualquer do alfabeto. O comando condicional *if* (ou comando de decisão) faz o seguinte teste: caso a letra digitada pelo usuário for a letra "s" então o programa vai imprimir para o usuário a mensagem "Vc pressionou a tecla s.", caso contrário o programa terminará sem apresentar nenhuma mensagem.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     char ch; //define uma variável inteira chamada idade
4     printf("\n Digite uma letra do alfabeto: "); //imprime na tela uma pergunta ao usuário
5     scanf("%c", &ch); //lê uma entrada do teclado
6        if( ch == 's') //testa se a entrada é igual ao carácter s
7           printf("\n Voce pressionou a tecla s."); // imprime uma mensagem para o usuário
8  }
```

Podemos também ter um bloco de instruções a serem executadas no comando if. Por exemplo, no programa abaixo temos dois comandos a serem executados quando o if for verdadeiro. Desta forma temos que colocar o bloco de instruções do *if* entre chaves.

```c
#include <stdio.h>//Define funções de entrada e saída
main() { //Função principal chamada main
   char ch; //define uma variável inteira chamada idade
   printf("\n Digite uma letra do alfabeto: "); //imprime na tela uma pergunta ao usuário
   scanf("%c", &ch); //lê uma entrada do teclado
   if(ch == 's'){ //testa se a entrada é igual ao carácter s
      printf("\n Voce pressionou a tecla s."); // imprime uma mensagem para o usuário
      printf("\n s significa saída."); // imprime uma mensagem para o usuário
   }
}
```

2.2.2 Comando: if-else

O comando *if-else* é semelhante ao if com a diferença do *else* (significa em português "senão") O comando *if-else* possui a seguinte sintaxe:

```
if (condição) instrução; else instrução
```

Onde a condição pode ser: verdadeira ou falsa. Caso a condição seja verdadeira a instrução a seguir será executada, caso a condição seja falsa a instrução do comando else será executada. Ou seja, uma das duas instruções será sempre executada pois a condição sempre será verdadeira ou falsa.

No programa a seguir temos um exemplo do uso do *if-else:*

```c
#include <stdio.h>//Define funções de entrada e saída
main() { //Função principal chamada main
   char ch; //define uma variável inteira chamada idade
   printf("\n Selecione M para ver a mensagem: "); //imprime na tela uma pergunta ao usuário
   scanf("%c", &ch); //lê uma entrada do teclado
   if(ch == 'M') //testa se a entrada é igual ao carácter s
      printf("\n Olá, Tenha um bom dia!!"); // imprime uma mensagem para o usuário
    else
       printf("\n Mensagem ignorada"); // imprime uma mensagem para o usuário
}
```

Cabe lembrar que caso queiramos escrever várias instruções dentre de um *if* ou *else*, usamos as chaves para determinar o começo e final do bloco de instruções.

2.2.3 Comando: *switch*

O comando *switch* (em português "escolha"), também conhecido como estrutura de decisão múltipla, é adequada quando precisamos escolher uma entre várias alternativas previamente definidas, por exemplo, em um menu. A linguagem C apresenta a seguinte sintaxe para o comando *switch*:

```
switch( expressão ) {
case constante1 : comando1 ; break;
case constante2 : comando2 ; break;
...
case constanten : comandon ; break;
default
: comando;
}
```

O comando switch tem a seguinte semântica: a expressão, que deve ser *char* ou *int*, é avaliada. Então para cada *case* (em português "caso"), encontrasse a constante cujo o valor é igual à expressão e executa todos os comandos a seguir até o *break* (parar). Se não existir tal caso, então é executado o comando associado ao caso default. Observa-se que o caso *default* é opcional e pode aparecer em qualquer posição entre os casos especificados.

No exemplo abaixo temos a definição de um menu com três opções: abrir, fechar e salvar. O usuário então deverá digitar uma das opções, identificadas pela letra inicial e maiúscula de cada opção. Assim, o programa apresentará a opção selecionada. Se o usuário selecionar outra opção será exibida a mensagem de "Opção inválida".

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     char opçao; //define uma variável inteira chamada idade
4     printf("\n Escolha uma opção: "); //imprime na tela uma mensagem ao usuário
5     printf("\n (A)brir "); //imprime na tela uma mensagem ao usuário
6     printf("\n (F)echar"); //imprime na tela uma mensagem ao usuário
7     printf("\n (S)alvar \n"); //imprime na tela uma mensagem ao usuário
8     scanf("%c", &opcao); //lê uma entrada do teclado
9     switch(opcao){
10       case 'A': printf("\n Vc selecionou a opção Abrir"); break; //opção == A
11       case 'F': printf("\n Vc selecionou a opção Fechar"); break; //opção == F
12       case 'S': printf("\n Vc selecionou a opção Salvar"); break; //opção == S
13       default: printf("\n Opção Inválida"); break; //opção == qualquer outro carácter
14    }
15 }
```

2.2.4 Comando: operador condicional

O operador condicional é uma forma de representar uma decisão de maneira mais simples e compacta. A vantagem de utilizar o operador condicional é na simplificação do programa. Sua sintaxe é:

```
condição? expressão1: expressão2
```

Onde a condição é avaliada podendo ser verdadeira ou falsa. Caso a condição seja verdadeira então a *expressão1* será executada senão a *expressão2* será executada.

No programa a seguir vemos o uso do operador condicional. O programa lê dois números inteiros do usuário e calcula e apresenta para o usuário qual dos números é o maior. Observe que na Linha 6 temos o operador condicional dentro de um comando *printf*. Outra forma de escrevermos um programa para verificar o maior número digitado seria utilizando o comando *if-else*.

```
1  #include <stdio.h>//define funções de entrada e saída
2  main() { //função principal chamada main
3     int valor1, valor2; //define duas variáveis inteiras
4     printf("\n Informe dois valores: "); //imprime na tela uma mensagem ao usuário
5     scanf("%d %d", &valor1, &valor2); //lê dois números do teclado
6     printf("\n O maior valor é %d", valor1>valor2 ? valor1 : valor2);//usa o operador condicional para verificar qual dos valores digitados é o maior
7  }
```

2.3
COMANDOS DE REPETIÇÃO

Para o estudo dos comandos de repetição, primeiro vamos ver algumas formas de escrita de expressões. Um conceito muito importante e utilizado na Linguagem de programação C é a atribuição. Por exemplo, se quisermos atribuir o valor de uma expressão a uma variável, utilizamos a seguinte sintaxe:

```
variável = expressão;
```

Onde variável é o nome de uma determinada variável e a expressão pode ser qualquer expressão onde o resultado dela seja do mesmo tipo da variável que receberá o valor. Por exemplo, se quisermos guardar o valor da soma de dois números em uma variável chamada soma temos que:

```
soma = valor1 + valor2;
```

Assim, a variável soma armazenará o valor da expressão (valor1+valor2).

Na linguagem C temos os operadores aritméticos de atribuição que combinam em um único operador: uma operação aritmética e uma atribuição. Por exemplo, se tivermos:

```
soma = soma + novoValor;
```

Neste exemplo, temos um acumulador chamado soma onde o valor armazenado na variável soma será sempre o da própria variável soma mais o novo valor. Com um operador aritmético de atribuição podemos reescrever o exemplo como:

```
soma += novoValor;
```

A vantagem em se utilizar uma abreviação é a geração de código mais rápido pelo compilador. O código acima poderia ser escrito da forma soma = soma + novoValor.

A Tabela 10 apresenta os operadores aritméticos de atribuição, onde na primeira coluna temos a expressão e na segunda a mesma expressão usando os operadores aritméticos de atribuição.

Tabela 10 - Operadores aritméticos de atribuição

Expressão	Expressão com operadores aritméticos de atribuição
x = x + y	x += y
x = x - y	x -= y
x = x * y	x *= y
x = x / y	x /=y
x = x % y	x %= y

Fonte: (SCHILDT, H., 1997, p. 44).

Outro conceito importante, e muito utilizado nos comandos de repetição, são o Incremento e o Decremento. Muitas vezes precisamos incrementar ou decrementar o valor de uma variável, por exemplo, se temos uma variável que representa a ordem de entrada começando por 1, e sabendo que o incremento é sempre de 1 em 1, então podemos usar o operador de incremento para isso.

Assim, se uma expressão incrementa ou decrementa o valor da variável, podemos escrevê-la numa forma mais compacta. Para incrementar usamos o operador ++ e para decrementar usamos o operador ––. Esses operadores podem ser usados tanto na forma pré-fixada quanto pós-fixada. A sintaxe desses operadores é:

```
variável++, variável-- //forma pós-fixada
++variável, --variável //forma pré-fixada
```

No programa abaixo vemos o uso dos operadores de incremento e decremento. São definidas duas variáveis inteiras recebendo o *valor inicial* 10. Então a variável *valor1* é incrementada e a variável *valor2* decrementada. Como resultado será apresentado ao usuário 11 e 9.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     int valor1=10, valor2=10; //define duas variáveis inteiras com valor inicial de 10
4     valor1++;  //Incrementa em 1 a variável valor1
5     --valor2;  //Decrementa em 1 a variável valor2
6     printf("\n %d %d", valor1, valor2);//Imprime as variáveis valor1 e valor2
7  }
```

A seguir veremos os comandos de repetição *for*, *while* e *do-while*, que são os comandos de repetição mais utilizados em linguagens de programação em geral.

2.3.1 Comando: *for*

O comando *for* define uma estrutura (instruções) que será repetida por um número determinado de vezes. Para que seja possível determinar quantas vezes precisamos executar as instruções, precisamos sempre definir um contador para o comando for. Na linguagem C a sintaxe do comando *for* é:

```
For (inicialização; condição; alteração) comando;
```

Onde, a inicialização é uma expressão que atribui um valor inicial ao contador, a condição possui um resultado verdadeiro ou falso e é utilizada para verificar se a contagem chegou ao fim e a alteração modifica o valor do contador. Enquanto a contagem não termina, o comando associado ao for é repetidamente executado.

Por exemplo, o programa define um contador que é inicializado com 1 e incrementa ele 10 vezes, exibindo o resultado de cada incremento.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     int contador; //define uma variável chamada contador
4  for(contador=1; contador<=10; contador++)  //inicializa o contador com 1; define a condição e incrementa o contador
5        printf("\n %d", contador);//Imprime os valores do contador
6  }
```

Caso precisamos executar uma lista de comandos dentro do for teremos que colocar essa lista em um bloco, ou seja, entre chaves. No exemplo abaixo o usuário digita um número de 1 a 10 e o programa exibe a sua respectiva tabuada.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     int contador, numero, resultado; //define as variáveis utilizadas
4     printf("\n Digite um número entre 1 e 10: ")//exibe mensagem para o usuário
5     scanf("%d", &numero);//lê um número
6     printf("\n A tabuada do número %d é:", numero); //exibe mensagem para o usuário
7      for(contador=1;  contador<=10;  contador++){   //inicializa o contador com 1; define a condição e incrementa o contador
8           resultado = numero*contador;//multiplica o número com o contador e armazena o resultado
9        printf("\n %d x %2d = %3d", numero, contador, resultado);//exibe a tabuada
10    }
11 }
```

2.3.2 Comando: *while*

O comando *while* define uma estrutura (instruções), que será repetida por um número indeterminado de vezes. Este comando também define uma pré-condição, e caso seja verdade, a estrutura irá ser executada enquanto a condição for verdadeira. O critério de parada do comando while é caso a condição seja falsa. A sintaxe do comando *while* é definida por:

```
while (condição){ comando};
```

Por exemplo, podemos reescrever o programa anterior, que utilizava o comando for, utilizando o comando *while*:

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3    int contador=1 numero, resultado; //define as variáveis utilizadas e inicia o contador com 1
4    printf("\n Digite um número entre 1 e 10: ")//exibe mensagem para o usuário
5    scanf("%d", &numero);//lê um número
6    printf("\n A tabuada do número %d é:", numero); //exibe mensagem para o usuário
7    while(contador<=10){   //inicializa o contador com 1; define a condição e incrementa o contador
8          resultado = numero*contador;//multiplica o número com o contador e armazena o resultado
9      printf("\n %d x %2d = %3d", numero, contador, resultado);//exibe a tabuada
10     contador++; //incrementa o contador
11   }
12 }
```

Observa-se que as diferenças são no uso do contador. Neste caso inicializamos o contador com 1 e dentro do bloco do while incrementamos o contador. Mesmo sendo possível de representar o mesmo problema da tabuada tanto com *for* quanto com *while*, o ideal neste caso é representar com for pois sabemos quantas vezes temos que repetir as instruções (a tabuada é de 1 a 10). Desta forma, em termos gerais se não soubéssemos quantas vezes precisamos repetir um conjunto de instruções o ideal seria usar o *while*. No entanto, ambas soluções estão corretas.

2.3.3 Comando: *do-while*

O comando *do-while* possui como diferença ao while a verificação da condição. Enquanto no comando while as instruções só serão executadas caso a condição seja verdadeira, no comando *do-while* primeiro as instruções são executadas (uma vez) e então a condição é verificada. Caso a condição seja verdadeira as instruções serão novamente executadas e caso seja falsa a execução termina. Assim, podemos dizer que com o comando *do-while* as instruções serão sempre executadas pelo menos uma vez.

A sintaxe do comando do-while é definida por:

```
do{comando} while(condição);
```

No exemplo abaixo, o programa solicita qual é a sua idade e depois compara com a idade dos seus colegas. Depois que você entra com a sua idade o programa executa o comando *do-while*. O programa solicita a entrada da idade de seu colega e então compara com a sua, dizendo se você é mais novo ou velho e quantos anos de diferença existem entre você e seu colega. O critério de parada é o usuário digitar o número zero. Desta forma você poderá comparar a sua idade com n colegas.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3    int minhaidade, idade; //define as variáveis utilizadas
4    printf("\n Entre com a sua idade: ")//exibe mensagem para o usuário
5    scanf("%d", &minhaidade); //lê a sua idade
6    do{
7       printf("\n Entre com a idade do seu colega ou zero para sair: ");//exibe mensagem
8       scanf("%d", &idade); //lê a idade de seu colega
9       if(idade!=0){
10         if(minhaidade>idade)//testa se minha idade é maior que a idade do colega
11                printf("\n Vc é %d anos mais velho que seu colega", minhaidade-idade);//exibe o
resultado
12         else
13                printf("\n Vc é %d anos mais novo que seu colega",idade-minhaidade); //exibe
o resultado
14   }while(idade!=0); //verifica se a idade é igual a zero
15 }
```

Neste exemplo, você pode observar o uso das chaves. Observe que o comando *if* é utilizado sem chaves, pois temos um comando apenas para ser executado em cada *if*. Também observamos que podemos encadear diversos comandos, ou seja, definir um comando dentro do bloco de outro comando. Isso vale para todos os comandos vistos até agora.

2.4 EXEMPLOS

Nesta seção veremos a resolução de alguns problemas. Para cada problema você deve pensar primeiro na lógica de programação (algoritmos) e depois pensar em como representar a solução do problema na linguagem de programação C.

2.4.1 Problema: Conversão de Temperatura

Neste problema, o usuário deverá entrar com a temperatura em graus *fahrenheit* e o programa deverá calcular o correspondente em graus *celsius*. Para conseguirmos resolver, primeiro precisamos saber como se dá a conversão de graus *fahrenheit* para *celsius*. Para isso usamos a fórmula (temperatura-32)*5/9. Sabendo isso temos o seguinte programa em C:

```
1  #include <stdio.h>//define funções de entrada e saída
2  main() { //função principal chamada main
3     int temperatura; //define uma variável chamada temperatura
4     printf("\n Entre com a temperatura em Fahrenheit:  "); //solicita a temperatura
5     scanf("%d", &temperatura);
6     printf("\n A temperatura em Celcius é:  %d", (temperatura-32)*5/9);
7  }
```

Observe que a conversão da temperatura foi feita dentro na própria função *printf*. Outra opção seria criar uma outra variável, armazenar o cálculo na nova variável e então utilizar esta outra variável na função *printf*.

2.4.2 Problema: Multiplicação com o Operador de Soma

Neste problema, o usuário deverá entrar com dois números inteiros e o programa deverá calcular a sua multiplicação. No entanto, o usuário deverá utilizar apenas o operador de soma. Para resolvermos este problema, temos que solicitar os dados do usuário e usarmos um comando de repetição para conseguirmos multiplicar apenas com o operador de soma.

```
1  #include <stdio.h>//define funções de entrada e saída
2  main() {//Função principal chamada main
3    int numero1, numero2, contador, resultado; //definição das variáveis
4    printf("\n Entre com dois números inteiros: "); //mensagem solicitando os números
5    scanf("%d %d", &numero1, &numero2); // leitura e armazenamento dos valores lidos
6    resultado=0; //atribuindo zero a variável resultado
7     for(contador=0;contador<numero1;contador++){ //comando de repetição que define um
     contador inciando por zero; será executado até que o contador for menor que o primeiro
     número informado
8        resultado += numero2; //resultado é igual ao próprio resultado mais o segundo número
     informado
9    }
10   printf("\n Resultado é %d ",resultado); //exibe o resultado da multiplicação
11 }
```

O programa inicia com a definição das variáveis (Linha 3) e solicita ao usuário os dois números a serem multiplicados. Na Linha 5 são lidos os dois números e atribuídos as variáveis *numero1* e *numero2*. A variável *resultado* (Linha 6) é inicializada com zero, no entanto isto poderia ter sido feito já na definição da variável (Linha 3). Na linha 7 temos o comando de repetição for, desta forma a lógica utilizada é somarmos o *numero2 n* vezes onde n é o *numero1*.

ATIVIDADES – UNIDADE 2

1. Escrever um Programa em C que leia a idade de um eleitor e verifique se ele pode ou não votar. Se ele puder votar, informe se o seu voto é facultativo. Sabe-se que a partir dos 16 até os 18 anos o voto é facultativo, assim como para os maiores de 70 anos.

2. Escrever um Programa em C que lê três valores e encontre o maior dos valores lidos. Escrever o maior valor ao final. Caso um valor lido já tenho sido digitado o programa deverá solicitar um novo valor ao usuário.

3 (Adaptado de ASCENCIO; CAMPOS, 2002) Uma empresa decidiu dar uma gratificação de Natal aos seus funcionários, baseada no número de horas extras e no número de horas que o funcionário faltou ao trabalho. O valor da gratificação é obtido a partir do coeficiente encontrado por meio da fórmula: número de horas extras – (2/3 * número de horas de faltas). Este coeficiente é aplicado segundo a tabela abaixo:

Coeficiente	Gratificação
>= 24	R$500,00
[18...24)	R$400,00
[12...18)	R$300,00
[6...12)	R$200,00
< 6	R$100,00

Escrever um Programa em C que leia o número de horas extras e o número de horas de faltas de um funcionário e calcule e escreva o valor da sua gratificação.

4. Escrever um Programa em C que lê um número não determinado de valores para M, todos inteiros e positivos, um de cada vez. Verificar se o valor M é par ou ímpar, escrevendo a mensagem correspondente. Contar o número de valores ímpares lidos e calcular a soma dos valores pares. No final, escrever o contador e a soma calculados.

3

VETORES E MATRIZES

INTRODUÇÃO

Neste capítulo veremos duas estruturas de dados muito comuns e utilizadas em linguagens de programação: vetores e matrizes. Nas linguagens de programação essas estruturas são utilizadas para armazenar (ou representar) muitos elementos de um mesmo tipo de dado. Um vetor é uma sequência de vários elementos (do mesmo tipo de dado) armazenados sequencialmente na memória e usando o mesmo nome de variável para acessar esses elementos. Um vetor também pode ser considerado uma matriz de apenas uma dimensão (uma linha).

Para a utilização de vetores e matrizes sempre será necessário a utilização de comandos de repetição, pois essas estruturas armazenam muitos elementos e uma das formas mais eficientes de se escrever ou ler valores de vetores e matrizes é por meio dos comandos de repetição vistos no capítulo anterior. Observamos também que os conceitos de vetores e matrizes vem da matemática e, por tanto, esses conceitos já devem ser compreendidos previamente. O que veremos neste capítulo será como a linguagem de programação C implementa as estruturas de vetores e matrizes.

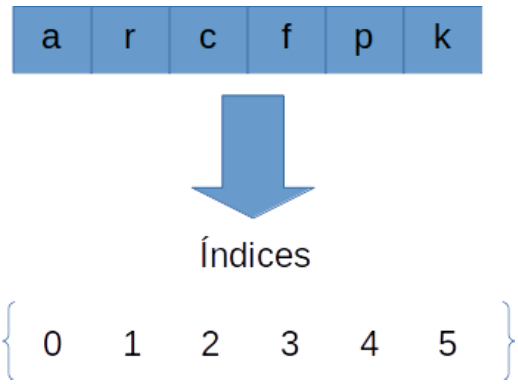
Veremos a definição de vetores e matrizes e a sua manipulação, ou seja, como criar, ler e armazenar elementos em vetores e matrizes na linguagem C. Por meio de exemplos, veremos alguns problemas que podem ser resolvidos usando vetores e matrizes. Por fim, as atividades possibilitarão a fixação do conteúdo visto neste capítulo, lembrando que toda a estrutura básica dos programas em C neste capítulo já foi vista no capítulo anterior.

3.1
DECLARAÇÃO DE VETORES E MATRIZES

Vetor é uma coleção de variáveis do mesmo tipo, acessível com um único nome, armazenados contiguamente na memória e acessados por meio de um índice.

A Imagem 17 apresenta a representação gráfica de um vetor. Este vetor armazena seis letras do alfabeto, onde a primeira letra possui o índice 0 a segunda o índice 1 e assim por diante. Ou seja, o índice de um vetor na linguagem C sempre começa pelo 0 (zero).

Imagem 17 - Representação gráfica de um vetor e seus índices



Fonte: Autor

Na linguagem C, um vetor é definido por:

```
tipo nome[quantidade elementos];
```

Onde o tipo representa o dado (*int, char, etc*), o nome a identificação do vetor e entre colchetes a quantidade de elementos que compõem o vetor. Por exemplo, o vetor representado pela Imagem 17 pode ser definido, na linguagem C, como:

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     char letras[6]; //define um vetor chamado letras com 6 elementos
4  }
```

Observa-se que apenas definimos o vetor. Agora precisamos inicializar ele com as letras. Para inicializarmos o vetor com as letras, conforme exemplo da Figura 15, temos que atribuir cada letra a um índice do vetor. Assim, temos:

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3    char letras[6]; //define um vetor chamado letras com 6 elementos
4    letras[0] = 'a'; //coloca a letra a no índice 0 do vetor
5    letras[1] = 'r; //coloca a letra r no índice 1 do vetor
6    letras[2] = 'c'; //coloca a letra c no índice 2 do vetor
7    letras[3] = 'f'; //coloca a letra f no índice 3 do vetor
8    letras[4] = 'p'; //coloca a letra p no índice 4 do vetor
9    letras[5] = 'k'; //coloca a letra k no índice 5 do vetor
10 }
```

Vejamos alguns exemplos de vetores. No exemplo abaixo definimos um vetor de números inteiros com 10 elementos (ou seja, 10 números).

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3    int vetor[10]; //define um vetor chamado vetor com 10 elementos
4  }
```

Podemos definir que a quantidade máxima de elementos de um vetor é sempre 10 usando uma constante, como o exemplo abaixo. Isto é útil quanto temos vários vetores onde sabemos qual é o tamanho máximo deles.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define MAXIMO 10 //Define a constante MAXIMO com valor 10
3  main() { //Função principal chamada main
4    int vetor[MAXIMO]; //define um vetor chamado vetor com 10 elementos
5  }
```

Se quisermos inicializar o vetor com os números de 1 a 10 devemos usar um comando de repetição, como segue. Observamos que a Linha 6 armazenamos o índice mais 1 pois a variável i começa com 0, desta forma, armazenamos os números de 1 a 10 no vetor.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define MAXIMO 10 //Define a constante MAXIMO com valor 10
3  main() { //Função principal chamada main
4    int vetor[MAXIMO]; //define um vetor chamado vetor com 10 elementos
5    for(int i=0; i<10;i++){//laço de 0 A 9
6        vetor[i] = i+1;//armazena o índice mais 1 no vetor
7    }
8  }
```

Caso queiramos colocar números múltiplos de 10 começando com 10. Neste caso, precisamos apenas alterar a Linha 6 onde teremos que colocar uma fórmula para armazenarmos os múltiplos de 10, conforme exemplo.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define MAXIMO 10 //Define a constante MAXIMO com valor 10
3  main() { //Função principal chamada main
4     int vetor[MAXIMO]; //define um vetor chamado vetor com 10 elementos
5     for(int i=0; i<10;i++){//laço de 0 A 9
6           vetor[i] = 10*(i+1);//armazena os múltiplos de 10 no vetor
7     }
8  }
```

Outro procedimento que podemos precisar fazer é manipular mais de um vetor ao mesmo tempo. Por exemplo, podemos querer copiar os valores de um vetor para outro vetor. No exemplo abaixo criamos três vetores *vetor1* e *vetor2*, ambos com tamanho 5, e um vetor chamado *vetorResultado* com tamanho de 10 que armazena os valores do *vetor1* e *vetor2*. Inicializamos os valores do *vetor1* com os números pares iniciando por 0 e o vetor2 com os números ímpares. Então copiamos os valores do vetor1 e vetor2 para o vetorResultado.

Neste exemplo, definimos primeiro os vetores (Linhas 4, 5 e 6) após, definimos duas variáveis que usamos para controlar os índices do *vetor1* e *vetor2*. O primeiro for inicia na Linha 8 e será responsável por armazenar os 5 primeiros números pares (começando por 0) no *vetor1* e os 5 primeiros números ímpares (começando por 1) no *vetor2*. Para sabermos se um número é par ou ímpar usamos o operador % que determina o resto da divisão, ou seja, se o resto da divisão for zero significa que o número é par, caso contrário o número é ímpar. A linha 17 inicia o *for* que copia os vetores *vetor1* e *vetor2* para o vetorResultado. Observe que copiamos os primeiros 5 valores pares do *vetor1* para as primeiras 5 posições do *vetorResultado* e os valores ímpares para as últimas 5 posições do *vetorResultado*.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define MAXIMO 5 //Define a constante MAXIMO com valor 5
3  main() { //Função principal chamada main
4     int vetor1[MAXIMO]; //define um vetor chamado vetor com 5 elementos
5     int vetor2[MAXIMO]; //define um vetor chamado vetor com 6 elementos
6     int vetorResultado[10]; //define um vetor chamado vetor com 10 elementos
7     int indice1=0, indice2=0; //define variáveis de controle de índices
8     for(int i=0; i<10;i++){//laço de 0 A 9
9        if(i%2 == 0){//verifica se o número é par
10             vetor1[indice1] = i;//armazena o número no vetor
11             indice1++;//incrementa o indice1
12       } else {//se o número não é par então ele é ímpar
13             vetor2[indice2] = i;//armazena o número no vetor
14             indice2++;//incrementa o indice2
15       }
16    }
17    for(int i=0; i<5; i++){//laço de 0 a 5
18                vetorResultado[i] = vetor1[i];//copia o vetor1 nas primeiras 5 posições do
vetorResultado
19                vetorResultado[i+5] = vetor2[i];//copia o vetor2 nas últimas 5 posições do
vetorResultado
20    }
21    for(int i=0;i<10;i++){//laço de 0 a 10
22       printf("\n vetorResultante[%d]: %d",i, vetorResultado[i]);//mostra o vetorResultado
23    }
24 }
```

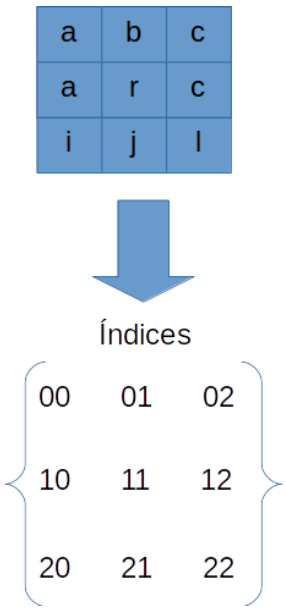
Outro procedimento útil com vetores é a leitura de dados pelo usuário. Suponhamos que precisamos saber a média aritmética das notas de 5 alunos. Para isso, podemos escrever o seguinte programa em C:

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define MAXIMO 5 //Define a constante MAXIMO com valor 5
3  main() {//Função principal chamada main
4    float nota[MAXIMO]; //define um vetor chamado nota com 5 elementos
5    float soma=0; //define uma variável soma inicializada por zero
6    double media; //define uma variável media
7    for(int i=0; i<MAXIMO;i++){//laço de 0 A 5
8        scanf("%f",&nota[i]);//lê a nota e armazena no vetor
9        soma += nota[i];//calcula a soma de todas as notas lidas
10   }
11   media=soma/MAXIMO;//calcula a média aritmética
12   printf("A média é: %.2lf", media);//mostra a média aritmética
13 }
```

Vimos até então como podemos representar vetores em C. Vetores são matrizes com apenas uma única linha (dimensão). Assim, veremos agora como representar matrizes na linguagem C.

A Imagem 18 apresenta a representação gráfica de uma matriz. Esta matriz armazena 9 letras do alfabeto, sendo composta por três linhas de três colunas cada. Os índices são dados por linha e coluna, ou seja, linha 0 e coluna 0, linha 0 e coluna 1, linha 0 e coluna 2, linha 1 e coluna 0, e assim por diante.

Imagem 18 - Representação gráfica de uma matriz



Fonte: Autor.

Na linguagem C, uma matriz é definida por:

```
tipo nome[quantidade linhas][quantidade colunas];
```

Onde, o tipo representa o dado (*int*, *char*, etc), o nome a identificação da matriz e entre os dois colchetes a quantidade de linhas e colunas que compõe a matriz respectivamente. Por exemplo, a matriz representada pela Imagem 18 pode ser definida, na linguagem C, como:

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() {//Função principal chamada main
3      char matrizLetras[3][3]; //define uma matriz chamada matrizLetras com 3 linhas e 3 colunas
4  }
```

A manipulação de matrizes se dá da mesma forma que dos vetores. No entanto, numa matriz temos que manipular as linhas e colunas. Por exemplo, vemos no programa abaixo como preencher uma matriz com a soma dos índices (linha e coluna). Observa-se que precisamos de dois comandos de repetição encadeados, um para percorrer todas as linhas e outro para percorrer todas as colunas.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define NLINHA 3
3  #define NCOLUNA 4
4  main() {//Função principal chamada main
5    int matriz[3][4]; //define uma matriz chamada matriz com 3 linhas e 4 colunas
6    for(int i=0; i < NLINHA; i++) //percorre as linhas
7        for(int j=0; j < NCOLUNA; j++){ //percorre as colunas
8            matriz[i][j] = i+j; //armazena a soma dos índices (linha e coluna)
9            printf("\n matriz[%d][%d] = %d",i,j,matriz[i][j]); //mostra os resultados
10       }
11 }
```

Suponhamos que quiséssemos ler valores do usuário em uma matriz e então mostrar os valores lidos. O programa C abaixo demonstra como poderíamos fazer isso.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define NLINHA 3
3  #define NCOLUNA 4
4  main() { //Função principal chamada main
5     int matriz[3][4]; //define uma matriz chamada matriz com 3 linhas e 4 colunas
6     int valorLido; //define uma variável para armazenar o valor lido
7     for(int i=0; i < NLINHA; i++) //percorre as linhas
8        for(int j=0; j < NCOLUNA; j++){ //percorre as colunas
9           printf("informe o valor para a matriz[%d][%d]: ",i,j); //mostra mensagem para o
   usuário
10          scanf("%d",&valorLido); //lê um valor do usuário
11          matriz[i][j] = valorLido; //armazena o valor lido na matriz
12       }
13    for(int i=0; i < NLINHA; i++) //percorre as linhas
14       for(int j=0; j < NCOLUNA; j++) //percorre as colunas
15          printf("\n%d",matriz[i][j]);
16 }
```

3.2
ACESSO AOS ELEMENTOS

Vimos como declarar um vetor e uma matriz e como inicializar eles. Cabe ressaltar que, quando definimos um vetor ou uma matriz ,temos que ter cuidado na utilização dos índices pois se atribuirmos valores a índices inexistentes teremos resultados indesejados. Vimos também que um vetor nada mais é do que uma matriz com uma dimensão (linha).

O acesso aos elementos de um vetor se dá por: gravação de valores e leitura de valores. Lembrando que o acesso é pelo índice. Desta forma podemos definir alguns procedimentos básicos que podemos utilizar sempre que temos algum problema onde precisamos armazenar vários dados de um mesmo tipo em um vetor ou matriz, como por exemplo: ler um vetor, ler uma matriz, exibir os valores de um vetor, exibir os valores de uma matriz, copiar um vetor em outro, copiar uma matriz em outra, etc.

O acesso aos índices é importante pois caso desenvolvemos um programa que acesse um índice inválido teremos algum erro. Desta forma a regra é que índices sempre começam por 0 (zero) até o seu tamanho menos 1, ou seja:

```
Índice = 0...(TAMANHO -1)
```

A leitura e gravação de valores em um vetor ou matriz sempre se dá por um comando de repetição, onde acessamos cada índice e, assim, podemos gravar ou ler, ou seja, sempre teremos:

```
for (int i=0; i<TAMANHO;i++)
```

Onde a inicialização, condição e incremento da variável i pode mudar de acordo com o problema, mas, em geral, sempre teremos um comando de repetição para a manipulação de valores de vetores e matrizes.

A leitura de valores e exibição de valores lidos do usuário se dá pelas funções já vistas no capítulo anterior. Portanto, podemos ler valores do usuário usando *scanf* e atribuir algum valor ou apenas mostrar os valores que existem na estrutura com o comando *printf*.

3.2
ACESSO AOS ELEMENTOS

Vimos como declarar um vetor e uma matriz e como inicializar eles. Cabe ressaltar que, quando definimos um vetor ou uma matriz ,temos que ter cuidado na utilização dos índices pois se atribuirmos valores a índices inexistentes teremos resultados indesejados. Vimos também que um vetor nada mais é do que uma matriz com uma dimensão (linha).

O acesso aos elementos de um vetor se dá por: gravação de valores e leitura de valores. Lembrando que o acesso é pelo índice. Desta forma podemos definir alguns procedimentos básicos que podemos utilizar sempre que temos algum problema onde precisamos armazenar vários dados de um mesmo tipo em um vetor ou matriz, como por exemplo: ler um vetor, ler uma matriz, exibir os valores de um vetor, exibir os valores de uma matriz, copiar um vetor em outro, copiar uma matriz em outra, etc.

O acesso aos índices é importante pois caso desenvolvemos um programa que acesse um índice inválido teremos algum erro. Desta forma a regra é que índices sempre começam por 0 (zero) até o seu tamanho menos 1, ou seja:

```
Índice = 0...(TAMANHO -1)
```

A leitura e gravação de valores em um vetor ou matriz sempre se dá por um comando de repetição, onde acessamos cada índice e, assim, podemos gravar ou ler, ou seja, sempre teremos:

```
for (int i=0; i<TAMANHO;i++)
```

Onde a inicialização, condição e incremento da variável i pode mudar de acordo com o problema, mas, em geral, sempre teremos um comando de repetição para a manipulação de valores de vetores e matrizes.

A leitura de valores e exibição de valores lidos do usuário se dá pelas funções já vistas no capítulo anterior. Portanto, podemos ler valores do usuário usando *scanf* e atribuir algum valor ou apenas mostrar os valores que existem na estrutura com o comando *printf*.

3.3
EXEMPLOS DE VETORES E MATRIZES

Vejamos alguns exemplos de resolução de problemas com vetores e matrizes.

3.3.1 Problema: Ordem de Exibição

Precisamos criar um programa em C que leia 10 números inteiros, exiba todos os elementos na ordem inversa e também todos os elementos pares. Primeiro definimos as variáveis que precisamos, neste caso o vetor com 10 posições e uma variável para a leitura dos números. Depois precisamos de um comando de repetição for para ler e armazenar os números no vetor. Então podemos exibir ele na tela na ordem invertida. Para isso basta percorrermos os índices do vetor do final (índice 9) até o índice inicial que é o 0 (zero). Por fim, exibimos os números pares testando cada valor se o resto da divisão é zero ou não. Se for zero é porque o número é par.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3    int vetor[10]; //define um vetor de 10 elementos
4    int numero; //define uma variável inteira numero
5    //lê 10 números e armazena em um vetor
6     for (int i=0;i<10;i++){
7        printf("\nEntre com um número inteiro: ");
8         scanf("%d",&numero);
9         vetor[i]=numero;
10    }
11    //mostra os números em ordem invertida
12    printf("\n-----------------------");
13    printf("\nO Vetor invertido é:");
14    for (int i=9;i>=0;i--){
15       printf("\n%d", vetor[i]);
16    }
17     //mostra os números pares
18    printf("\n-----------------------");
19    printf("\n Os números pares são:");
20    for (int i=0;i<10;i++){
21       if(vetor[i]%2==0)
22            printf("\n%d", vetor[i]);
23    }
24 }
```

3.3.2 Problema: Maior e Menor Elemento

Precisamos criar um programa em C que leia 5 números inteiros e exiba o maior e menor elemento lido. Para isso, primeiro precisamos ler os valores e armazenar no vetor. Também precisamos definir duas variáveis, uma para armazenar o maior e outra o menor valor. Assim, uma vez que os valores estejam armazenados no vetor precisamos percorrer e comparar elemento por elemento para encontrar o maior e o menor.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  main() { //Função principal chamada main
3     int matriz1[3][2];
4     int matriz2[3][2];
5     int matrizResultante[3][2];
6     int numero;
7     //lê a matriz1
8     printf("----------matriz1---------");
9     for(int i=0; i < 3; i++) //percorre as linhas
10       for(int j=0; j < 2; j++){//percorre as colunas
11          printf("\n Entre com um valor para matriz1[%d][%d]: ",i,j);
12          scanf("%d",&numero);
13          matriz1[i][j]= numero;
14       }
15    //lê a matriz2
16    printf("----------matriz2---------");
17    for(int i=0; i < 3; i++) //percorre as linhas
18       for(int j=0; j < 2; j++){//percorre as colunas
19          printf("\n Entre com um valor para matriz2[%d][%d]: ",i,j);
20          scanf("%d",&numero);
21          matriz2[i][j]= numero;
22       }
23    //soma a matriz1 com a matriz2
24    for(int i=0; i < 3; i++) //percorre as linhas
25       for(int j=0; j < 2; j++){//percorre as colunas
26          matrizResultante[i][j] = matriz1[i][j]+matriz2[i][j];
27       }
28    //imprime a matriz resultante
29    for(int i=0; i < 3; i++) //percorre as linhas
30       for(int j=0; j < 2; j++){//percorre as colunas
31          printf("\n Valor para matrizResultante[%d][%d] é: %d",i,j,matrizResultante[i][j]);
32       }
33 }
```

ATIVIDADES – UNIDADE 3

1) Escreva um Programa em C que leia um vetor de números inteiros e ordene-o em ordem crescente. Exiba o vetor lido e o vetor resultante.

2) Escreva um Programa em C que leia um vetor de 10 posições e imprime todos os valores menores que 10, e suas respectivas posições no vetor.

3) (Adaptado de LOPES; GARCIA, 2002) Escreva um Programa em C que leia dois vetores de 25 posições cada um. A seguir, criar um terceiro vetor, intercalando os dados dos dois vetores. Este terceiro vetor deve ser impresso no final.

4) Escreva um Programa em C que leia duas matrizes de dimensão 3 x 3, calcule e imprima a subtração das matrizes.

5) Escreva um Programa em C que leia uma matriz quadrada de dimensão 4 e encontre e imprima:

a) A matriz lida
b) O menor valor
c) O maior valor

4

FUNÇÕES

INTRODUÇÃO

O estudo de funções (ou procedimentos) torna-se necessário quando queremos organizar nossos programas de uma melhor forma. Funções na linguagem de programação C são mais gerais que na matemática. Podemos dizer que cada função é uma pequena parte de um determinado programa. Cada programa em C pode ser composto por inúmeras funções, uma delas a função main já vimos desde o início do estudo da linguagem C. A função main sempre será a função principal, ou seja, por onde a execução de um determinado programa em C inicia. Desta forma, dizemos que uma função em C é um grupo de sentenças (comandos) que executam uma determinada tarefa, e em todo programa C, teremos sempre pelo menos uma função que é a função main. Também já utilizamos nos capítulos anteriores várias funções já definidas pela linguagem C, como por exemplo, as funções de leitura e gravação: scanf() e printf(). Estas funções estão definidas na biblioteca stdio.h. Além dessas, podemos usar inúmeras outras funções da linguagem C que já foram definidas, para facilitar o desenvolvimento de programas. No entanto, quando temos algum problema específico, por exemplo, as atividades já desenvolvidas, podemos criar funções específicas para cada problema.

Neste capítulo veremos o que são funções, como declarar e definir uma função. Veremos também como organizar um programa em funções. Por fim, veremos como funcionam as funções recursivas, algumas vantagens e desvantagens. Entre os benefícios no uso de funções, o principal que será visto neste capítulo será a organização do código.

4.1
DECLARAÇÃO DE FUNÇÕES

Já vimos a utilização de algumas funções como *scanf*() e *printf*(). Assim, a linguagem de programação C oferece uma enorme quantidade de funções já definidas. Por exemplo, se quisermos escrever um programa que precise calcular o módulo de um número, podemos utilizar a função abs() que está contida na biblioteca stdlib.h. Assim teríamos:

```
1  #include <stdio.h>//define funções de entrada e saída
2  #include <stdlib.h>//define funções de entrada e saída
3  main() { //função principal chamada main
4     int a,b; //define duas variáveis inteiras
5     a = abs(5.6);//atribui a variável a o módulo do número 5,6
6     printf("Módulo de a = %d\n", a);
7     b = abs(-10);//atribui a variável b o módulo do número -10
8     printf("Módulo de b = %d\n", b);
9  }
```

Neste caso, temos que o módulo de um número real é a parte inteira e de um número negativo é o mesmo número só que positivo. O uso de funções já definidas facilita muito o desenvolvimento de programas.

Funções podem ser descritas como pequenos algoritmos (ou procedimentos) dentro de um programa. Uma função é definida como:

```
tipo-retorno nome-função(parâmetros){
 declarações
 comandos
 valor-retorno;
}
```

Onde:

- **tipo-retorno**: define o tipo de retorno da função, por exemplo, se a função retorna um número inteiro o tipo será int. Caso não tenha retorno será do tipo void.

- **nome-função**: define o nome da função. É um identificador único dentro do programa.

• **parâmetros**: lista de tipos de dado e parâmetros que a função recebe. É opcional pois podemos ter uma função que não passe nenhum parâmetro.

• **declarações**: podemos definir diversas declarações como variáveis, vetores, etc.

• **comandos**: podemos definir diversos comandos de decisão, repetição, outras funções, etc.

• **valor-retorno**: valor a ser retornado pela função. É opcional pois se a função não retornar nenhum valor não precisa ser colocado.

Veremos a seguir vários pequenos exemplos de definição e uso de funções. No exemplo abaixo, definimos uma função soma que recebe dois números inteiros e retorna como resultado a soma desses dois números (Linhas 7, 8 e 9). Observe que a função principal (*main*) foi definida como tipo inteiro e como valor de retorno o (zero). O *return*(0) é definido para dizer ao compilador que a função main terminou com sucesso.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  int main () {//define a função principal main
3     printf("A Soma de 1 e 2 é: %d",soma(1,2));//mostra a soma de dois números
4     return(0);
5  }
6   int soma(int a, int b){//definição da função soma, responsável por somar dois números
7      return a+b;
8  }
```

Agora, vamos dizer que queiramos ler valores do usuário para passar para a função soma. Assim podemos definir uma função leNumero() retornando um número lido, como segue:

```
1  #include <stdio.h>//define funções de entrada e saída
2  int main () {
3     int a,b; //define duas variáveis inteiras
4     a = leNumero(); //atribui a variável a um número
5     b = leNumero();//atribui a variável b um número
6     printf("A Soma de %d e %d é: %d",a,b,soma(a,b));//imprime a soma de dois números
7     return(0);
8  }
9   int soma(int a, int b){//definição da função soma, responsável por somar dois números
10     return a+b;
11 }
12 int leNumero(){//definição da função leNumero, responsável por ler um número inteiro
13    int numero;
14    printf("Entre com um valor: ");
15    scanf("\n%d",&numero);
16    return numero;
17 }
```

Observe que nesse caso generalizamos a solução do exemplo anterior, pois agora, podemos ler qualquer número do usuário. Outra observação importante é com relação à modularização do programa, ou seja, temos agora duas funções: uma que soma dois números e uma que lê um número inteiro. Assim, quando precisarmos ler um número inteiro podemos sempre utilizar a função *leNumero*.

Funções também podem ser usadas para executar alguma sequência de comandos que não retornam nada. Por exemplo, muitas vezes precisamos mostrar valores como um vetor. No exemplo abaixo veremos alguns conceitos importantes.

O programa abaixo simplesmente declara um vetor, associando valores a ele e então imprime por meio da função imprimeVetor. O primeiro conceito importante no uso de funções é a sua declaração explicita (Linha 3). Apesar de ser possível ter declarações implícitas de funções, na linguagem C recomenda-se sempre declarar a função antes da função principal *main*. Desta forma, temos apenas a declaração da função imprimeVetor antes da função *main* e definimos a sua implementação a partir da Linha 11.

Observamos também que no exemplo a baixo a função *imprimeVetor* é do tipo *void*, ou seja, ela não retorna nenhum valor apenas executa os comandos necessários para a impressão dos valores do vetor.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #define MAXIMO 10
3  void imprimeVetor();//declaração explicita da função imprimeVetor
4  int main () {//define a função principal main
5     int vetor[MAXIMO];
6     for(int i=0; i<MAXIMO;i++)//lê valores para o vetor
7        vetor[i] = i+1;
8     imprimeVetor(vetor);//chama a função imprimeVetor passando o vetor como parâmetro
9     return(0);
10 }
11  void imprimeVetor(int vetor[MAXIMO]){//define a função imprimeVetor
12    for(int i=0; i<MAXIMO;i++)
13       printf("\n vetor[%d]= %d",i,vetor[i]);
14 }
```

Como vimos podemos passar valores por parâmetros. Podemos passar por exemplo, uma lista de valores e de tipos diferentes. No exemplo a seguir, veremos como podemos passar mais de um parâmetro para uma função, inclusive outra função.

O próximo exemplo consiste em um programa que calcula o Índice de Massa Corporal (IMC) e analisa os resultados. Definimos duas funções, uma para o cálculo do IMC (*calculaIMC*) e outra para mostrar os resultados (*mostraResultadosIMC*). Na função principal main solicitamos que o usuário entre com os dados de peso (em kilograma) e altura (em metros), lembrando que números fracionários em C são separados por ponto. Na Linha 10 usamos o printf para imprimir o valor do IMC e chamamos a função *calculaIMC* (de dentro da função printf) passando como parâmetro o peso e a altura. O retorno da função *calculaIMC* será o resultado e será mostrado ao usuário. A função *mostraresultadoIMC* recebe como parâmetro a função *calculaIMC*.

A primeira vantagem em desenvolver um programa com diferentes funções é a sua organização. Por consequência programas mais organizados são mais fáceis de serem alterados. Suponhamos que queremos rever os resultados do IMC pois surgiram novas pesquisas que demonstraram que o IMC poderia ser classificado apenas em Abaixo do Peso, Normal e Acima do Peso. Desta forma, precisamos alterar apenas a função *mostraResultadoIMC*.

```
1  #include <stdio.h> //Define funções de entrada e saída
2  void mostraResultadoIMC(float IMC);
3  float calculaIMC(float peso, float altura);
4  int main(){
5      float peso, altura;
6      printf("Entre com o seu peso (Kg): ");
7      scanf("%f",&peso);
8      printf("Entre com a sua Altura (Metros): ");
9      scanf("%f",&altura);
10     printf("\nSeu IMC é: %0.2f\n",calculaIMC(peso, altura));
11     mostraResultadoIMC(calculaIMC(peso, altura));
12 }
13 float calculaIMC(float peso, float altura){
14     return peso/(altura*altura);
15 }
16 void mostraResultadoIMC(float IMC){
17     if(IMC<17){
18         printf("Muito abaixo do peso.\n");
19     }else{
20         if(IMC>17 && IMC<18.49){
21         printf("Abaixo do peso.\n");
22         }else{
23             if(IMC>18.5 && IMC<24.99){
24                 printf("Peso normal.\n");
25             }else{
26                 if(IMC>25 && IMC<29.99){
27                     printf("Acima do peso.\n");
28                 }else{
29                     if(IMC>30 && IMC<34.99){
30                         printf("Obeso.\n");
31                     }else{
32                         if(IMC>35 && IMC<39.99){
33                             printf("Obesidade Severa.\n");
34                         }else
35                             if(IMC>40)
36                                 printf("Obesidade MORBIDA.\n");
37                         }
38                     }
39                 }
40             }
41     }
42 }
```

4.1.1 Declaração de Variáveis

Quando trabalhamos com funções temos um importante conceito com relação à declaração das variáveis chamado **escopo de variáveis**. Podemos declarar variáveis dentro das funções bem como fora delas. Quando declaramos uma variável dentro de uma função chamamos de **variável local**. Quando declaramos uma variável fora de uma função chamamos de **variável global**.

As variáveis locais, como já vimos desde o Capítulo 2, são aquelas declaradas dentro de uma função e não podem ser modificadas ou utilizadas por outras funções. Estas variáveis existem enquanto a função onde ela está declarada está sendo utilizada. Desta forma, podemos ter duas funções diferentes que declaram, cada uma delas, uma variável do mesmo tipo e nome, e mesmo assim, são consideradas variáveis diferentes.

As variáveis globais são aquelas declaradas fora de uma função inclusive da função *main*. Estas variáveis existem no decorrer de todo o programa e podem ser usadas e modificadas em qualquer parte do programa. Observe que o uso de variáveis globais tende a tornar os programas mais complexos pois exige do programador, a cada modificação, que ele verifique se a variável global mantém os valores corretos durante toda a execução de um programa.

Vejamos o exemplo abaixo que calcula o quadrado de um número. Definimos uma variável global chamada numero e inicializamos ela com zero na Linha 3. Na função principal main primeiro mostramos o valor dela. Após, lemos um valor do usuário e armazenamos na mesma variável. Por fim chamamos a função *calculaQuadrado* onde a variável global numero é modificada para armazenar o seu quadrado. Assim, após a definição da variável global alteramos ela tanto na função principal como na função *calculaQuadrado*.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  void calculaQuadrado();
3  int numero = 0;
4  int main () {//define a função principal main
5     printf("O número vale: %d\n", numero);
6     printf("Entre com um valor: ");
7     scanf("%d",&numero);
8     calculaQuadrado();//chama a função calculaQuadrado
9     printf("O quadrado do número lido é: %d", numero);
10 }
11 void calculaQuadrado(){//definição da função que calcula o quadrado do número
12    numero *= numero;
14 }
```

No próximo exemplo vemos como o uso de variáveis locais funciona. Temos uma variável local chamada numero que é definida dentro da função principal *main*, mas também é passada por parâmetro e usada dentro da *função calculaQuadrado*. Na definição da variável numero atribuímos o valor 5 (Linha 4), a seguir imprimimos o valor da variável e como resultado devemos ter 5. Após chamamos a função *calculaQuadrado* passando a variável como parâmetro e após calcular o

seu quadrado imprimimos o seu valor. Por fim, na função *main* imprimimos novamente a variável numero e verificamos que ela continua com o valor 5. Isto se deve ao escopo da variável local que, como vimos, é válido apenas dentro da sua função.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  void calculaQuadrado(int numero);
3  int main () {//define a função principal main
4      int numero = 5;
5      printf("O número vale: %d\n", numero);
8      calculaQuadrado(numero);//chama a função calculaQuadrado passando o numero como parâmetro
9      printf("O número vale: %d\n", numero);
10 }
11 void calculaQuadrado(){//definição da função que calcula o quadrado do número
12     numero *= numero;
13     printf("O quadrado do número lido é: %d", numero);
14 }
```

4.1.2 Exemplos

Nesta seção reescreveremos alguns dos exemplos dos capítulos 2 e 3. O objetivo é mostrar que podemos modularizar os exemplos já vistos com o uso de funções.

O primeiro exemplo que vemos é a multiplicação usando o operador de soma (Capítulo 2 Seção 2.4.2). Abaixo temos o mesmo programa agora reescrito e usando funções. Definimos duas funções, uma para ler os números que serão somados e a outra para calcular a soma. A solução com funções é considerada elegante e com uma melhor manutenabilidade. Ou seja, ela se torna mais simples caso precise fazer alguma alteração. Outro aspecto a ser observado é a utilização das variáveis locais que se mantêm no escopo das funções e torna o entendimento do programa mais simples.

```c
#include <stdio.h>//define funções de entrada e saída
int leNumero();
int somaDoisNumeros(int numero1, int numero2);
main() { //Função principal chamada main
   int numero1, numero2, resultado;
   numero1 = leNumero();
   numero2 = leNumero();
   resultado = somaDoisNumeros(numero1,numero2);
  printf("\n Resultado é %d ",resultado);
}
int somaDoisNumeros(int numero1, int numero2){
   int resultado = 0;
   for(int contador=0;contador<numero1;contador++){
       resultado += numero2;
   }
   return resultado;
}
int leNumero(){
   int numero;
   printf("Entre com um valor: ");
   scanf("\n%d",&numero);
   return numero;
}
```

Uma possível simplificação do exemplo acima seria passar como parâmetro para a função *somaDoisNumeros* a função *leNumero*, assim, não precisaríamos definir as variáveis locais na função *main*. Vejamos como ficaria o mesmo código agora simplificado e passando as funções como parâmetros. Desta forma, a função *main* apenas chama as funções e imprime o resultado.

Observe que o exemplo abaixo é a terceira solução diferente para o mesmo problema. Quando escrevemos um programa (algoritmo) usamos de uma lógica, também chamado de pensamento lógico, que é única de cada programador. Portanto, podemos ter várias soluções diferentes para o mesmo problema.

```
1  #include <stdio.h>//define funções de entrada e saída
2  int leNumero();
3  int somaDoisNumeros(int numero1, int numero2);
4  main() { //Função principal chamada main
5    printf("\n Resultado é %d ",somaDoisNumeros(leNumero(),leNumero()));
6  }
7  int somaDoisNumeros(int numero1, int numero2){
8     int resultado = 0;
9     for(int contador=0;contador<numero1;contador++){
10        resultado += numero2;
11    }
12    return resultado;
13 }
14 int leNumero(){
15    int numero;
16    printf("Entre com um valor: ");
17    scanf("\n%d",&numero);
18    return numero;
19 }
```

O segundo exemplo que vemos é o programa que lê dez valores inteiros, imprime na ordem inversa e imprime apenas os números pares (Capítulo 3 Seção 3.3.1). Neste exemplo definimos 3 funções: uma para inicializar o vetor onde passamos a quantidade de valores que deverão ser lidos, outra para imprimir a ordem inversa e, por fim, uma função para imprimir os pares. Nesta solução foi criado um vetor de inteiros global e as funções acessam ele. Uma alternativa a essa implementação é criar um vetor local na função principal e ir passando ele por parâmetro para as demais funções.

```
1  #include <stdio.h>//define funções de entrada e saída
2  int inicializaVetor(int quantidade);
3  void imprimeOrdemInversa();
4  void imprimePares();
5  int vetor[10];
6  main() {
7     int quantidade = 10;
8     inicializaVetor(quantidade);
9     imprimeOrdemInversa();
10    imprimePares();
11 }
12 int inicializaVetor(int quantidade){
13    int numero;
14    for (int i=0;i<quantidade;i++){
15       printf("\nEntre com um número inteiro: ");
16       scanf("%d",&numero);
17       vetor[i]=numero;
18    }
19 }
20 void imprimeOrdemInversa(){
21    printf("\nVetor invertido:");
22    for (int i=9;i>=0;i--){
23       printf("\n%d", vetor[i]);
24    }
25 }
26 void imprimePares(){
27    printf("\n Os números pares são:");
28    for (int i=0;i<10;i++){
29       if(vetor[i]%2==0)
30          printf("\n%d", vetor[i]);
31    }
32 }
```

4.2
RECURSIVIDADE

Recursão é uma técnica de programação que pode ser utilizada sempre que for possível expressar a solução de um determinado problema em termos do próprio problema. Na linguagem de programação C podemos definir funções recursivas que são funções que chamam ela mesma. Ou seja, a função é chamada de recursiva se no seu código tem uma chamada para ela mesma.

A técnica de recursão precisa ser utilizada com cautela pois o problema é como fazer a recursão parar. Caso não for pensado direito a recursão pode ocasionar um *loop* infinito, ou seja, a execução do programa não para mais, podendo gerar algum erro ou até esgotar a memória. Em muitos casos uma solução usando comandos de repetição (também chamada de solução interativa) pode ser mais adequada, pois aloca menos memória e é mais eficiente em termos de performance. No entanto, para alguns problemas a recursão pode ser um recurso que torna o programa mais simples e elegante.

O exemplo clássico de recursão é o cálculo do fatorial de um número. Vejamos a seguir como ficaria um programa em C que calcula o fatorial de um número. Neste exemplo, criamos uma função chamada fatorial onde passamos por parâmetro um número inteiro. A função fatorial é recursiva pois chama ela mesmo (Linha 20) e o critério de parada é quando a variável *numero* for igual a zero.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  #include <stdlib.h>
3  int fatorial(int n);
4  int main() {
5     int numero;
8     printf("Entre com um número: ");
9     scanf("%d",&numero);
10    printf("\nO fatorial de %d é: %d",numero, fatorial(numero));
11 }
12 int fatorial(int numero) {
13    if (numero == 0) //condição de parada da recursão
14       return 1;
15    else
16       if (numero < 0) {
17          printf("Erro: fatorial de número negativo!\n");
18          exit(0);
19       }
20    return numero * fatorial(numero – 1);//chamada recursiva
21 }
```

Vejamos outro exemplo com o uso de uma função recursiva. No programa abaixo temos uma função recursiva que multiplica dois números. Primeiro entramos com os valores dos dois números. Observe que estamos usando neste exemplo um tipo *long int* que possibilita armazenar inteiros longos. A lógica da função *multiplica* é somar o primeiro número *n* vezes, onde *n* é o segundo número. Desta forma, temos a multiplicação de dois números usando uma função recursiva.

```
1  #include <stdio.h>//Define funções de entrada e saída
2  long int multiplica(int numero1, int numero2);
3  int main() {
4     int numero1,numero2;
5     printf("Entre com o primeiro número: ");
6     scanf("\n%d", &numero1);
7     printf("Entre com o segundo número: ");
8     scanf("\n%d", &numero2);
9     printf("\n%d multiplicado por %d é: %ld", numero1, numero2,
                                                multiplica(numero1,numero2));
10 }
11 long int multiplica(int numero1, int numero2) {
12    if (numero2 == 0)
13       return 0;
14    else
15       return (numero1 + multiplica(numero1, numero2 - 1));
16 }
```

ATIVIDADES - UNIDADE 4

1) Reescreva os exercícios das atividades 2 e 3 fazendo uso de funções.

2) Escreva um Programa em C que contenha uma função para cada uma das operações aritméticas: soma, subtração, multiplicação e divisão. O programa deverá aceitar apenas dois valores e a operação desejada.

3) Escreva um Programa em C que converta um número decimal para um número binário, ou seja, o programa deverá ter uma função onde recebe um valor em decimal e retorna o valor convertido em binário.

4) Escreva um Programa em C que leia dois conjuntos de valores: Nomes e Matrículas de alunos. Cada Nome é associado a um número inteiro entre os valores 1 e 100 de Matrícula. Imprima a lista de Nomes e Matrículas ordenadas:

a) Por Nome (escreva uma função que receba o conjunto de matrículas e nomes e retorne-os de forma ordenada)

b) Por Número de Matrícula (escreva uma função que receba o conjunto de matrículas e nomes e retorne-os de forma ordenada).

5

REGISTRO

INTRODUÇÃO

Registros são estruturas de dados que permitem armazenar em uma única variável um conjunto de variáveis diferentes. Em linguagens de programação chamamos estas estruturas de registro, no entanto, na linguagem C os registros chamam-se struct. Aqui usaremos registros, estruturas e *struct* como sinônimos.

Programas de computador são estruturados para a resolução de um determinado problema. No capítulo anterior vimos as estruturas de dados de vetores e matrizes que armazenam dados de um mesmo tipo. Entretanto, as vezes precisamos armazenar dados de diferentes tipos, as vezes também chamados de dados complexos. Assim, o estudo de registros em C é importante pois possibilita estruturarmos nossos programas de forma a ficarem mais eficientes e fáceis de serem compreendidos. Imagine que tenhamos que fazer um programa onde envolva manipular um registro de usuário que é composto pelo seu nome, número de identificação, idade, sexo e telefone de contato. Temos, neste caso, pelo menos 3 tipos diferentes de dados: nome e telefone representados por um vetor de char; número de identificação e idade representados por um *int*; e sexo representado apenas por um *char*. Assim, podemos criar uma *struct* que represente o registro de usuários. Um registro em C também pode ser utilizado, por exemplo, para definir um novo tipo de dado que envolva diferentes variáveis.

Neste capítulo veremos como registros são declarados e como podem ser utilizados. Também veremos como podemos manipular vetores de registros. Por fim, as atividades visam reforçar o aprendizado de registros.

5.1 DECLARAÇÃO

Um registro (ou estrutura) pode ser considerado como um template (modelo) usado para definir uma coleção de variáveis por meio de um único nome. Estruturas ajudam os programadores a agruparem elementos de tipos de dados diferentes. No entanto, elas também podem agrupar elementos de um mesmo tipo de dado, mas que tornam o uso dentro de um programa mais claro e simples.

Por exemplo, imaginamos que queiramos armazenar uma data composta por dia, mês e ano em uma estrutura chamada data. A sintaxe para a criação de uma estrutura é:

```
1  struct data //criação da estrutura chamada data
2  {
3     int dia; //define a variável dia
4     int mes; //define a variável mês
5     int ano; //define a variável ano
6  };
```

A Linha 1 define o nome da estrutura. Toda a estrutura deve ficar dentro de um bloco definido por chaves. As Linhas 3, 4 e 5 definem as variáveis que compõem a estrutura data.

Vejamos agora um exemplo onde as variáveis que compõem a estrutura possuem tipos de dados diferentes. Neste exemplo, definimos uma estrutura aluno que contém dados de um aluno como matrícula, nome e sexo. Para a matrícula usamos um tipo inteiro, para o nome usamos um vetor de caracteres e sexo definimos apenas um caractere do tipo *char*.

```
1  struct aluno //criação da estrutura chamada data
2  {
3     int matricula; //define a variável matrícula
4     char nome[30]; //define a variável nome
5     char sexo; //define a variável sexo
6  };
```

Desta forma, um registro em C segue a seguinte estrutura:

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

struct <identificador> <variável>;
```

Onde o identificador define um nome único para o registro e a lista de tipos e membros é uma lista de definição das variáveis que compõem o registro. Podemos dizer que um registro ou uma *struct* é uma variável especial que possui diversas outras variáveis podendo ou não ser de tipos diferentes.

Após a criação do *struct*, precisamos definir a variável que vai utilizá-la. Essa variável é definida por:

```
struct <identificador> <variável>;
```

Onde o identificador é o mesmo identificador usado na criação da struct e a variável é o nome que será utilizado no decorrer do programa.

5.2
ACESSO AOS REGISTROS

Uma vez que criamos a estrutura podemos utilizá-la durante o programa. Vejamos um exemplo simples.

No programa abaixo definimos uma estrutura (Linhas 2 a 7) que define dados de um aluno, como nome, disciplina e nota da primeira e segunda prova. Observe que o nome e a disciplina são vetores de caracteres enquanto as notas das provas são valores decimais.

A linha 8 apresenta a definição de uma variável aluno que possui a estrutura definida na struct dados_aluno. Definimos a seguir duas funções, uma que será usada para escrever os valores lidos do usuário na estrutura e outra que lê e apresenta, para o usuário, os dados contidos na estrutura.

A função main (Linhas 11 a 16) apenas apresenta as mensagens para a escrita e a leitura dos dados, bem como a chamada das respectivas funções.

A função escreveDadosAluno (Linhas 17 a 26) mostra como podemos escrever os dados em uma struct. Nesta função vemos uma forma diferente de escrever em um vetor de caracteres. Como vimos no capítulo de vetores, para escrevermos em um vetor temos que criar um laço (por meio do comando for) que percorre cada índice do vetor e escreve nele. Nesta função temos uma forma mais simples para escrevermos em um vetor de caracteres, por meio da função fgets que recebe, neste caso, três parâmetros: a variável que armazenará o valor; o tamanho, ou seja, a quantidade de caracteres que podem ser lidos; e o parâmetro stdin que define que a leitura será realizada pela entrada padrão que é o teclado. Para lermos os valores das notas usamos a função scanf que temos usado no decorrer deste livro. Observe que a escrita de um valor na estrutura se dá pelo nome da estrutura seguida de um ponto seguido do nome da variável contida na estrutura. No caso do nome do aluno (Linha 19) temos que aluno.nome é a variável que armazenará o nome do aluno.

A função leDadosAluno (Linhas 27 a 32) mostra como lemos os valores armazenados em uma sctuct. Para a leitura usamos a função printf e da mesma forma que armazenamos os valores nas variáveis da estrutura também lemos elas, ou seja, nome da estrutura seguida de um ponto seguido do nome da variável contida na estrutura.

```c
#include <stdio.h>
struct dados_aluno {
   char nome[50];
   char disciplina[30];
   float nota_prova1;
   float nota_prova2;
};
struct dados_aluno aluno;
void escreveDadosAluno();
void leDadosAluno();
void main() {
   printf("\n---------- Cadastro de aluno ----------\n");
   escreveDadosAluno();
   printf("\n --------- Dados do aluno ---------\n");
   leDadosAluno();
}
void escreveDadosAluno() {
   printf("Nome do aluno .....: ");
   fgets(aluno.nome, 50, stdin);
   printf("Disciplina .....: ");
   fgets(aluno.disciplina, 30, stdin);
   printf("Informe a 1a. nota...: ");
   scanf("%f", &aluno.nota_prova1);
   printf("Informe a 2a. nota...: ");
   scanf("%f", &aluno.nota_prova2);
}
void leDadosAluno() {
   printf("Nome .....: %s", aluno.nome);
   printf("Disciplina .....: %s", aluno.disciplina);
   printf("Nota da Prova 1...: %.2f\n", aluno.nota_prova1);
   printf("Nota da Prova 2...: %.2f\n", aluno.nota_prova2);
}
```

Podemos também ter estruturas dentro de estruturas, ou seja, um struct que define uma variável que também é uma struct. Vejamos o exemplo abaixo, que apresenta os dados de um funcionário.

Neste exemplo temos duas struct, uma para armazenar o endereço e outra para armazenar os dados do funcionário. Na *struct* do funcionário criamos uma variável end que se refere a *struct* endereco. Desta forma quando escrevemos ou lemos temos que referenciar a *struct* correta. Por exemplo, para acessarmos a variável rua temos *funcionario.end.rua*. Neste exemplo, temos duas particularidades. A primeira é a forma que usamos para achar o tamanho da variável *rua*, *endereco* e *nome*. Para não precisarmos colocar explicitamente o valor nos parâmetros, quando lemos ou escrevemos nessa variável, usamos a função *sizeof* passando como valor a variável, sendo que o retorno é o tamanho desta variável. É sempre recomendável que usemos a função sizeof para sabermos o tamanho da variável, assim se precisarmos mudar, por alguma razão, o valor da variável, precisamos mudar apenas na sua

definição. O segundo ponto que observamos é o uso do comando *_fpurge(stdin)*, que serve para limpar o *buffer* antes de ler com a função *fgets*. Como boa prática sempre antes de usarmos a função fgets devemos usar o *_fpurge(stdin)*.

```
1  #include <stdio.h>
2  struct endereco {
3     char rua[30];
4     int numero;
5     char bairro[20];
6     int cep;
7  };
8  struct dados_funcionario {
9     char nome[30];
10    char sexo;
11    int matricula;
12    struct endereco end;
13 };
14 void main() {
15    struct dados_funcionario funcionario;
16    printf("\n --------- Dados do Funcionário ---------\n");
17    printf(" Nome: ");
18    fgets(funcionario.nome, sizeof (funcionario.nome), stdin);
19    printf(" Sexo (M/F): ");
20    scanf("%c", &funcionario.sexo);
21    printf(" Matricula: ");
22    scanf("%d", &funcionario.matricula);
23    printf(" Rua: ");
24    _fpurge(stdin);
25    fgets(funcionario.end.rua, sizeof (funcionario.end.rua), stdin);
26    printf(" Numero: ");
27    scanf("%d", &funcionario.end.numero);
28    printf(" Bairro: ");
29    _fpurge(stdin);
30    fgets(funcionario.end.bairro, sizeof (funcionario.end.bairro), stdin);
31    printf(" CEP: ");
32    scanf("%d", &funcionario.end.cep);
33    printf("\n\n\n -------- Mostra os Dados do Funcionário ---------\n");
34    printf("Nome: %s", funcionario.nome);
35    printf("\n Sexo: %c", funcionario.sexo);
36    printf("\n Matricula: %d", funcionario.matricula);
37    printf("\n Rua: %s", funcionario.end.rua);
38    printf("\n Número: %d", funcionario.end.numero);
39    printf("\n Bairro: %s", funcionario.end.bairro);
40    printf("\n CEP: %d", funcionario.end.cep);
41 }
```

Outra possibilidade com o uso de *struct* é termos um vetor de *struct*. Por exemplo, quando queremos armazenar vários funcionários. Assim, para criarmos um vetor de struct definimos a estrutura da mesma forma e o que muda é na definição da variável da estrutura (Linha 8) para que ela seja um vetor. Para ler e escrever em um vetor usamos um laço (comando *for*).

```
1  #include <stdio.h>
2  struct dados_funcionario {
3     char nome[30];
4     char sexo;
5     int matricula;
6  };
7  void main() {
8     struct dados_funcionario funcionario[5];
9     printf("\n -------- Dados do Funcionário --------\n");
10    for (int i = 0; i < 5; i++) {
11       printf("Nome: ");
12       _fpurge(stdin);
13       fgets(funcionario[i].nome, sizeof (funcionario[i].nome), stdin);
14       printf("Sexo (M/F): ");
15       scanf("%c", &funcionario[i].sexo);
16       printf("Matricula: ");
17       scanf("%d", &funcionario[i].matricula);
18    }
19    printf("\n\n\n -------- Mostra os Dados do Funcionário --------\n");
20    for (int i = 0; i < 5; i++) {
21       printf("Nome: %s", funcionario[i].nome);
22       printf("\n Sexo: %c", funcionario[i].sexo);
23       printf("\n Matricula: %d", funcionario[i].matricula);
24    }
25 }
```

5.3
EXEMPLOS DE *STRUCT*

Vejamos alguns exemplos de resolução de problemas com *struct*.

5.3.1 Problema: Tipo de Dado *String*

Um dos tipos de dados, muito comum, que temos em linguagens de programação mais modernas hoje é o *String*, que corresponde a um conjunto de caracteres. Em C não temos esse tipo, mas podemos facilmente criar uma estrutura dessas. Observe que a definição de uma estrutura facilita muito a programação, principalmente quando temos tipos de dados que envolvem dados mais complexos.

```
1  #include <stdio.h>
2  struct tipo_string {
3     char str[100];
4  };
5  struct tipo_string string;
6  void main() {
7     printf("\n Digite uma frase: ");
8        __fpurge(stdin);
9        fgets(string.str, sizeof (string.str), stdin);
10       printf("\nA frase lida é: %s", string.str);
11 }
```

5.3.2 Problema: Cadastro de Alunos

Imagine que queiramos fazer um programa em C para cadastrar alunos. O cadastro do aluno envolve inserir novos alunos, alterar e excluir. Para cada uma das opções podemos fazer uma função.

Primeiro vamos definir a *struct registro_aluno* e a definição das funções, como segue. Defiinimos na Linha 4 uma constante que contém a quantidade de alunos, que neste caso é 10. Após definimos as 3 funções para as operações de inserir, mostrar e excluir aluno. As Linhas 8 a 13 definem a estrutura que um registro de alunos deve ter, ou seja, cada aluno terá matrícula, nome, disciplina e nota.

Observe que na função *main* temos um *for(;;)* sem a definição dos parâmetros. Isto significa que esse comando executará por indeterminadas vezes. A condição de parada não se encontra no comando for mas sim na Linha 32 com o comando *exit(0)* que será executado quando o usuário digitar a opção zero que significa sair.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAX 10
5  void inserir(void);
6  void mostrar(void);
7  void excluir(void);
8  struct registro_aluno {
9     char matricula[15];
10    char nome[30];
11    char disciplina[30];
12    char nota[5];
13 };
14 struct registro_aluno aluno[MAX];
15 int num;
16 int main(void) {
17    for (;;) {
18       printf("\t\t\t\tCADASTRO DE ALUNOS\n");
19       printf("1-Para adicionar alunos\n");
20       printf("2-Para mostrar alunos\n");
21       printf("3-Para excluir um aluno\n");
22       printf("0- Sair\n");
23       _fpurge(stdin);
24       scanf("%d", &num);
25       switch (num) {
26          case 1: inserir();
27             break;
28          case 2: mostrar();
29             break;
30          case 3: excluir();
31             break;
32          case 0: exit(0);
33          default: printf("TENTE NOVAMENTE");
34       }
35    }
36 }
```

As funções estão definidas no trecho de código abaixo. A Função *inserir* (Linhas 1 a 17) apresenta como os alunos são inseridos na *struct*. A função *mostrar* (Linhas 18 a 25) apresenta como a struct é percorrida e os valores contidos nela são mostrados ao usuário. A função excluir (Linhas 26 a 40) apresenta como os valores podem ser *excluídos*. Neste caso a exclusão ocorre da seguinte forma: o usuário entra com o nome do aluno a ser excluído, então verificamos com a função *strcmp* (Linha 32) se o nome corresponde com algum nome contido na struct. Caso encontre os valores serão substituídos por nada (“”).

```
1  void inserir(void) {
2    for (int i = 0; i < MAX; i++) {
3      _fpurge(stdin);
4      printf("\n\n\nEntre com a matricula do aluno: ");
5      fgets(aluno[i].matricula, sizeof (aluno[i].matricula), stdin);
6      _fpurge(stdin);
7      printf("Entre com o nome do aluno: ");
8      fgets(aluno[i].nome, sizeof (aluno[i].nome), stdin);
9      _fpurge(stdin);
10     printf("Entre com a disciplina do aluno:");
11     fgets(aluno[i].disciplina, sizeof (aluno[i].disciplina), stdin);
12     _fpurge(stdin);
13     printf("Entre com a nota do aluno:");
14     fgets(aluno[i].nota, sizeof (aluno[i].nota), stdin);
15     _fpurge(stdin);
16   }
17 }
18 void mostrar(void) {
19   for (int i = 0; i < MAX; i++) {
20     printf("\n Matricula do aluno: %s", aluno[i].matricula);
21     printf("\n Nome do aluno: %s", aluno[i].nome);
22     printf("\n Disciplina do aluno: %s", aluno[i].disciplina);
23     printf("\n Nota do aluno: %s", aluno[i].nota);
24   }
25 }
26 void excluir(void) {
27   char nome[50];
28   printf("Qual o nome do aluno q vc deseja remover?\n");
29   _fpurge(stdin);
30   fgets(nome, sizeof (nome), stdin);
31   for (int i = 0; i < MAX; i++) {
32     if ((strcmp(nome, aluno[i].nome)) == 0)
33       for (; i < MAX - 1; i++) {
34         strcpy(aluno[i].matricula, "");
35         strcpy(aluno[i].nome, "");
36         strcpy(aluno[i].disciplina, "");
37         strcpy(aluno[i].nota, "");
38       }
39   }
40 }
```

ATIVIDADES - UNIDADE 5

1. Escreva um programa em C que crie um cadastro de funcionários. O cadastro deve conter matrícula, nome, sexo, endereço completo (rua, número, bairro, cidade e cep) e salário. Deverá ser possível cadastrar funcionários, visualizar os funcionários cadastrados, excluir um funcionário e alterar os dados de um determinado funcionário.

2. Escreva um programa em C que defina o tipo de dados data contendo dia, mês e ano. O programa deverá ler um número determinado de datas e exibi-las em ordem crescente.

6

ARQUIVOS

INTRODUÇÃO

Até então os dados eram inseridos no programa por meio do teclado. Arquivos podem ser usados para armazenar esses dados vindos do usuário ou dados processados pelos programas. Arquivos não são apenas usados para armazenar dados, mas os próprios programas também são armazenados em arquivos. No sentido de entender como os arquivos funcionam, precisamos aprender como são as operações de entrada e saída de um arquivo, também conhecidas como operações de manipulações de arquivos. Ou seja, como ler dados de um arquivo e como escrever dados em um arquivo.

Um arquivo representa uma sequência de bites no disco rígido (também conhecido como memória secundária) onde um grupo de dados é armazenado. Os arquivos são criados para armazenagem de dados permanentes. Arquivos em C são utilizados tanto para a gravação quanto para a leitura de dados no disco rígido. Em C os arquivos são manipulados pelas funções da biblioteca stdio.h.

Neste capítulo veremos como podemos manipular arquivos. Assim, estudaremos os dois tipos possíveis de acesso a um arquivo: sequencial, ou seja, lendo um registro atrás do outro e aleatório, ou seja, posicionando-se diretamente em um determinado registro. No contexto de arquivos, um registro corresponde a um conjunto de informações ou a uma informação específica em um arquivo. Por meio de exemplos, veremos como podemos utilizar arquivos em C para armazenar dados processados em um programa e como podemos ler dados de um determinado arquivo. Este último capítulo complementa o estudo da linguagem C possibilitando a criação de programas onde os dados permaneçam armazenados mesmo com o término do programa.

6.1
ABRINDO E FECHANDO ARQUIVOS

Existem muitas formas de manipular arquivos. Algumas, por exemplo, só funcionarão com um determinado sistema operacional pois dependem de bibliotecas específicas para acessar o sistema de arquivos do sistema operacional, como é o caso da biblioteca conio.h que funciona no ambiente Windows. Veremos nesta seção formas de manipular arquivos como abrir e fechar arquivos, ler e escrever valores em um arquivo, etc.

Desta Forma, podemos ter bibliotecas específicas que funcionarão para um determinado sistema operacional apenas. No entanto, aqui continuaremos trabalhando com a biblioteca stdio.h, que fornece funções para manipulação de entradas e saídas.

Um conceito importante que precisamos para trabalhar com arquivos é ponteiro. Um ponteiro (ou apontador) é um tipo especial de variável que armazena um endereço de memória. Por exemplo, se temos um ponteiro que armazena o endereço de uma variável chamada valor então temos que o ponteiro aponta para a variável valor ou podemos dizer também que o ponteiro possui o endereço de memória da variável valor.

Para manipularmos arquivos precisamos então definir um ponteiro da seguinte forma:

```
FILE *p;
```

Onde p é o ponteiro que aponta para o arquivo. Sempre definimos ponteiros com um asterisco. Para trabalharmos com arquivos em C precisamos sempre abri-los e fechá-los.
Para abrir um arquivo usamos a função fopen com a seguinte sintaxe:

```
fopen(nome_arquivo, "modo");
```

Onde o *nome_arquivo* corresponde ao local onde o arquivo encontra-se ou o local onde ele deverá ser criado e armazenado. E o *modo* especifica como o arquivo pode ser aberto. Na Tabela abaixo temos alguns modos mais comuns utilizados para abrir arquivos de texto e binário. Um arquivo texto é todo arquivo onde os caracteres estão de acordo com a tabela ASCII e um arquivo binário é todo arquivo que não é arquivo texto, por exemplo, um arquivo de imagem ou som, um programa de computador compilado, etc.

Tabela 11 – Modos para abrir arquivos

Modo	Descrição
r	Abre um arquivo texto para leitura. Condição: o arquivo precisa existir antes de ser aberto
w	Abre um arquivo texto para gravação. Condição: se o arquivo não existir ele é criado ou se já existir o conteúdo anterior será apagado
a	Abre um arquivo texto para gravação. Condição: se o arquivo existir os dados serão adicionados no fim do arquivo ou se não existir um novo arquivo será criado
rb	Abre um arquivo binário para leitura. Condição: mesma do modo "r", só que o arquivo é binário
wb	Abre um arquivo binário para gravação. Condição: mesma do modo "w"
ab	Abre um arquivo binário para gravação de dados no fim do arquivo. Condição: mesma do modo "a"
r+	Abre um arquivo texto para leitura e gravação. Condição: o arquivo deve existir e pode ser modificado
w+	Abre um arquivo texto para leitura e gravação. Condição: se o arquivo existir, o conteúdo anterior será apagado ou se não existir, um novo arquivo será criado
a+	Abre um arquivo texto para gravação e leitura. Condição: se o arquivo existir, os dados serão adicionados no fim do arquivo, ou se não existir um novo arquivo será criado
r+b	Abre um arquivo binário para leitura e escrita. Condição: mesma do modo "r+"
w+b	Abre um arquivo binário para leitura e escrita. Condição: mesma do modo "w+"
a+b	Abre um arquivo binário para gravação de dados no fim do arquivo. Condição: mesma do modo "a+"

Fonte: (SCHILDT, H., 1997, p. 219).

Quando manipula-se arquivos, via programas em C, sempre precisamos fechar o arquivo no final do programa. Para fecharmos um arquivo podemos usar a função *fclose, que possui a seguinte sintaxe*:

```
int fclose(FILE *p);
```

Onde passamos o ponteiro para o arquivo criado. Observe que a função retorna um inteiro. Desta forma, se a função retornar zero significa que o arquivo foi fechado com sucesso, caso contrário aconteceu algum erro no fechamento do arquivo.

No exemplo abaixo, temos um programa que abre e fecha um arquivo. Primeiro definimos dois ponteiros, um para a localização do arquivo no disco rígido (Linha 3) e outro para o ponteiro do arquivo (Linha 4). Com o comando *fopen* abrimos o arquivo no *modo wb* (Linha 5). Se o ponteiro do arquivo p retornar *null* significa que ocorreu algum erro na abertura do arquivo, caso contrário o arquivo foi aberto com sucesso. Por fim o arquivo é fechado (Linha 10).

```
1  #include <stdio.h>
2  void main() {
3     char *path = "/meusprogramas/teste/arquivos/teste1";
4     FILE *fp; //declaração do ponteiro do arquivo
5     fp = fopen(path, "wb"); //o arquivo se chama teste1 e está localizado no diretório
   /meusprogramas/teste/arquivos/
6     if (!fp)
7        printf("Erro ao abrir arquivo!");
8     else
9        printf("Arquivo aberto com sucesso!");
10    fclose(fp);
11 }
```

6.2
LENDO ARQUIVOS

Uma vez que abrimos um arquivo, podemos ler (caso exista algum dado nele) ou escrever no arquivo. Para lermos um dado do arquivo podemos usar as funções *fgetc* ou *fscanf*.

A Sintaxe da função *fgetc* é:

```
int fgetc(FILE *p)
```

Onde o parâmetro é o ponteiro do arquivo e a função retorna um inteiro que representa o caractere lido, ou caso aponte para o *fim* do arquivo retorna EOF (*End of File*), que vale -1. Como caracteres são representados por inteiros, que vão de 0 até 255, um caractere no arquivo nunca terá o valor -1, ou seja, os caracteres sempre retornarão valores inteiros entre 0 e 255. Caso retorne -1 sabemos que estamos no fim do arquivo.

A função fgetc lê o primeiro caractere de um arquivo e automaticamente já se posiciona no próximo. E assim, é possível ler caractere por caractere, até encontrar a constante EOF que significa o fim do arquivo. Abaixo um exemplo do uso da função *fgetc*.

Na Linha 6 abrimos o arquivo conforme já foi visto anteriormente. Caso o ponteiro do arquivo retorne *NULL* significa que ocorreu algum erro na sua abertura. Desta forma, usamos a função perror que no caso de algum erro vai imprimir na tela a string "Erro:*ofpen*" e a mensagem de erro é retornada, para que o usuário saiba porque não foi possível abrir o arquivo. A função exit (Linha 9) aborta o programa e a constante *EXIT_FAILURE* é usada para mostrar que o programa terminou por algum erro. As Linhas de 11 a 13 são onde os caracteres do arquivo serão lidos, um a um. Assim, o programa vai lendo, por meio da função fgetc, os caracteres do arquivo até chegar no final do arquivo identificado pela constante *EOF*.

Eventualmente podemos ter algum erro na leitura com o *getc*. As Linhas 14 de 15 apresentam uma verificação da constante *EOF*, ou seja, se chegamos no final do arquivo (*caractere*==*EOF*) e o *EOF* possui valor 0 e temos algum erro (ferror(p) != 0)) então imprimimos o erro para o usuário. Por fim, se tudo deu certo e não temos nenhum erro fechamos o arquivo e abortamos o programa com a função exit e a constante *EXIT_SUCCESS*.

```c
#include <stdio.h>
#include <stdlib.h>
void main() {
   int caractere;
   FILE *p;
   p = fopen("/meusprogramas/teste/arquivos/teste1", "r");
   if (p == NULL) {
      perror("Erro: fopen");
      exit(EXIT_FAILURE);
   }
   while ((caractere = fgetc(p)) != EOF) {
      printf("Caractere lido: %c\n", caractere);
   }
   if ((caractere == EOF) && (feof(p) == 0) && (ferror(p) != 0)){
      perror("Erro: fgetc");
   }
   fclose(p);
   return EXIT_SUCCESS;
}
```

A Sintaxe da função *fscanf* é:

```
int fscanf(FILE *arquivo, char *string_formatada)
```

Onde os parâmetros são: o ponteiro para o arquivo e a *string* formatada que segue a mesma formatação da função *scanf*, ou seja, o formato e o endereço da variável a ser armazenada.

A função *fscanf* é utilizada quando temos que ler entradas de um arquivo em um determinado formato. Por exemplo, se temos um arquivo com 3 valores inteiros que representam a idade, altura e peso de indivíduos. No exemplo abaixo temos um programa em C que lê esses valores de um arquivo chamado *indivíduos.txt* e imprime na tela os valores lidos.

```c
#include <stdio.h>
void main() {
   FILE *arquivo;
   int idade, peso, altura;
   char nomeArquivo[] = "individuos.txt";
   arquivo = fopen(nomeArquivo, "r");
   if (arquivo == NULL)
      printf("Erro, nao foi possivel abrir o arquivo\n");
   else
      while ((fscanf(arquivo, "%d %d %d\n", &idade, &altura, &peso)) != EOF)
         printf("Idade %c Altura %c Peso %c\n", idade, altura, peso);
   fclose(arquivo);
   return 0;
}
```

6.3
ESCREVENDO EM ARQUIVOS

Agora veremos como podemos escrever em arquivos. Isso é útil, por exemplo, quando queremos armazenar o resultado de um processamento de forma permanente. Assim, podemos usar arquivos para a armazenagem. Veremos duas funções para escrita em arquivos: *fputc* e *fprintf*.

A Sintaxe da função *fputc* é:

```
void fputc(int caractere, FILE *p);
```

Onde os parâmetros são: o caractere a ser escrito no arquivo e o ponteiro do arquivo. A função. O programa abaixo lê uma frase de até 20 caracteres e escreve (Linha 14), caractere por caractere, no arquivo.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void main() {
4     FILE *arquivo;
5     char string[20];
6     arquivo = fopen("escrita.txt","w");
7     if (!arquivo) {
8        printf("Erro ao abrir arquivo");
9        exit(0);
10    }
11    printf("Entre com uma frase de até 20 caracteres: ");
12    scanf("%s", &string);
13    for (int i = 0; string[i]; i++){
14       fputc(string[i], arquivo); //escreve caractere a caractere no arquivo
15    }
16    fclose(arquivo);
17    return 0;
18 }
```

Outra forma de escrevermos em um arquivo, mas agora escrevermos algo formatado. Suponhamos que queiramos gerar um arquivo com 3 números inteiros que representam a idade, a altura e o peso de indivíduos. Assim, podemos utilizar a função *fprintf* para escrevermos dados formatados em um arquivo.

A sintaxe da função *fprintf* é:

```
int fprintf(FILE *arquivo, char *string_formatada);
```

Onde os parâmetros são: o ponteiro para o arquivo e a *string* formatada que segue a mesma formatação da função *printf*, ou seja, o formato e a variável a ser armazenada. O programa a baixo lê a idade, altura e peso de um único indivíduo e armazena no arquivo *indivíduos.txt*.

```
1  #include <stdio.h>
2  void main() {
3     FILE *arquivo;
4     int idade, peso, altura;
5     char nomeArquivo[] = "individuos.txt";
6     arquivo = fopen(nomeArquivo, "r");
7     if (arquivo == NULL)
8        printf("Erro, nao foi possivel abrir o arquivo\n");
9     else {
10       printf("Idade: ");
11       scanf("%d", &idade);
12       fprintf(arquivo, "%d\n", idade);
13       printf("Altura: ");
14       scanf("%d", &altura);
15       fprintf(arquivo, "%d\n", altura);
16       printf("Peso: ");
17       scanf("%d", &peso);
18       fprintf(arquivo, "%d\n", peso);
19    }
20    fclose(arquivo);
21    return 0;
22 }
```

6.4 OUTRAS FUNÇÕES

Além das funções vistas aqui temos algumas outras que podem ser úteis. Por exemplo, para a leitura de arquivos vimos a fgets que lê caractere por caractere e a fscanf que lê uma *string* formatada. Além dessa, temos também a função *fgets* que lê geralmente uma linha inteira e a função *fread* que lê dados binários de um arquivo.

Para a escrita de arquivos vimos as funções *fputc* que escreve um único caractere em um arquivo e *fprinf* escreve um string formatada em um arquivo. Além dessas, temos a função *fputs* que escreve uma string sem formatação no arquivo e fwrite que escreve dados binários em um arquivo.

Com isso, temos uma série de funções para leitura e escrita de arquivos em C. Também vimos algumas funções auxiliares durante este capítulo, como por exemplo a função *perror*. Existem duas funções para tratamento de erro em arquivos: a *ferror* e a *perror*.

Quando um arquivo é acessado, a função *ferror* pode retornar zero, caso nenhum erro tenha ocorrido, ou um valor diferente de zero, caso algum erro ocorreu. Geralmente é possível acessar arquivos sem problemas, no entanto, em situações específicas como: disco rígido cheio ou arquivo protegido pode ser que o programa não consiga acessar. Desta forma a utilização dessa função é útil para sabermos se o acesso foi realizado com sucesso ou não. Pode ser utilizada em conjunto com a função *ferror* para sabermos em que parte do programa ocorreu o erro.

Também vimos que a constante *EOF* indica o fim de um arquivo. As vezes, podemos querer saber se um arquivo chegou ao fim sem ler caractere por caractere. Assim, usamos a função feof que retorna zero se o arquivo chegou no final, ou não caso não tenha chegado ao final.

Uma função muito útil é a *fseek*, que posiciona a leitura ou gravação em um determinado ponto do arquivo. Ou seja, a função *fseek* move a posição corrente de leitura ou escrita no arquivo de um valor especificado, a partir de um ponto especificado.

A sintaxe da função *fseek* é:

```
int fseek (FILE *p, long numbytes, int origem);
```

Onde *p* é o ponteiro do arquivo, *numbyte* é o número de *bytes* de movimentação que serão contados (ou seja, quantos bytes de deslocamento serão dados na posição atual) e *origem* pode assumir três possíveis constantes:

- **SEEK_SET:** posiciona no início do arquivo e possui valor 0.
- **SEEK_CUR:** posiciona no ponto corrente no arquivo e possui o valor 1.
- **SEEK_END:** posiciona no fim do arquivo e possui o valor 2.

O programa C abaixo abre um arquivo chamado *funcaofseek.txt* (Linha 4), escreve no arquivo (Linha 5). Então é usado o comando fseek para posicionar no início do arquivo (*SEEK_SET*) e começar a escrever a 10 bytes (Linha 6). Desta forma, o texto a ser escrito na Linha 7 irá ser escrito no arquivo no começo dele a 10 *bytes* do início da linha e logo após vem o texto que foi escrito pela Linha 5.

```
1  #include <stdio.h>
2  void main() {
3     FILE *arquivo;
4    arquivo = fopen("funcaofseek.txt","w+");
5    fputs("texto no inicio do arquivo", arquivo);
6    fseek( arquivo, 10, SEEK_SET);
7    fputs("Linguagem de Programação C", arquivo);
8    fclose(arquivo);
9    return(0);
14 }
```

Outra função útil na manipulação de arquivos é a função *rewind* que basicamente volta para o começo do arquivo. A sintaxe da função *rewind* é:

```
void rewind (FILE *p);
```

Onde p é o ponteiro para o arquivo. A função pode ser útil, por exemplo, quando escrevemos alguma coisa no arquivo e depois queiramos ler novamente o arquivo desde o início. O programa abaixo mostra um exemplo da função *rewind*. Entre as Linhas 6 e 8 abrimos o arquivo, escrevemos uma frase e então fechamos. Na linha 9 abrimos novamente o arquivo e lemos, caractere por caractere o conteúdo do arquivo. Neste exemplo, queremos ler novamente o arquivo então usamos a função *rewind* (Linha 16) para posicionar o ponteiro do arquivo no início do arquivo e lemos novamente.

```c
#include <stdio.h>
void main() {
   FILE *arquivo;
   char string[] = "Exemplo de uso da função rewind";
   char caractere;
   arquivo = fopen("funcaorewind.txt", "w");
   fprintf(arquivo, "%s", string);
   fclose(arquivo);
   arquivo = fopen("funcaorewind.txt", "r");
   while (1) {
      caractere = fgetc(arquivo);
      if (feof(arquivo))
         break;
      printf("%c", caractere);
   }
   rewind(arquivo);
   printf("\n");
   while (1) {
      caractere = fgetc(arquivo);
      if (feof(arquivo))
         break;
      printf("%c", caractere);
   }
   fclose(arquivo);
   return (0);
}
```

ATIVIDADES -UNIDADE 6

1) Faça um programa que leia para 10 pessoas o seu nome, idade, peso e altura. Armazene essas informações em um arquivo. Leia esse arquivo e calcule o IMC de cada uma delas, grave o resultado em outro arquivo que deve ter apenas o nome o valor do IMC. Após as pessoas por ordem alfabética com o resultado do seu IMC.

2) Faça um programa que armazene notas de alunos em um arquivo. O programa deverá ler uma quantidade determinada pelo usuário de alunos e para cada aluno o seu nome e duas notas. O programa deverá ler o arquivo e imprimir na tela o nome do aluno e a sua nota final que deverá ser a média aritmética de suas notas.

CONSIDERAÇÕES FINAIS

Neste livro vimos um pouco da linguagem de programação C. O livro serve como um guia para o aprendizado da linguagem C. No entanto, cabe lembrar que a lógica para a realização dos exercícios é a mesma utilizada em algoritmos. Outra observação importante quando aprendemos a programar é que precisamos praticar, fazer e refazer os exercícios várias vezes, tentando cada vez utilizar recursos diferentes que a linguagem de programação provê.
No Capítulo 1 vimos um pouco sobre linguagens de programação em geral e um dos ambientes de programação C. Já no Capítulo 2 os conceitos básicos da linguagem foram vistos como os comandos condicionais e de repetição. No Capítulo 3 vimos a representação de vetores e matrizes, estruturas de dados muito utilizadas em programas de computador. No Capítulo 4 vimos funções e como podemos reestruturar programas com o uso de funções. No Capítulo 5 vimos que os registros em C são representados por uma estrutura de dados chamada struct. No Capítulo 6 aprendemos como trabalhar com arquivos.

A linguagem de programação C é uma linguagem abrangente, completa e ainda muito utilizada, tanto na academia quanto na indústria de software. Este livro apresentou um panorama sobre a linguagem e os principais conceitos utilizados pela linguagem, no entanto, existem uma infinidade de funções e conceitos avançados sobre a linguagem C que vimos aqui e que podemos ser uteis para determinados tipos de problemas. Lembre-se que a linguagem de programação C é uma linguagem de propósito geral, assim sempre é aconselhável a utilização de guias de referências sobre a linguagem.

REFERÊNCIAS

ARAUJO, J. **Dominando a Linguagem C**. Rio de Janeiro: Ciência Moderna, 2004.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos da Programação de Computadores: algoritmos, Pascal e C/C++**. São Paulo: Prentice-Hall, 2002.

CELES FILHO, W. **Introdução a Estruturas de Dados com Técnicas de Programação em C**. Rio de Janeiro: Elsevier, 2004.

KERNIGHAN, B. W. C: A linguagem de programação padrão ANSI. Rio de Janeiro: Elsevier, 1989.

LOPES, A.; GARCIA, G. **Introdução à Programação**. Rio de Janeiro: Campus, 2002.

LOUDEN, KENNETH C. **Compiladores**. São Paulo: Pearson Addison Wesley, 2008.

SCHILDT, H**. C Completo e Total**. São Paulo: Makron Books, 1997.

SILVA, O. Q. **Estruturas de Dados e Algoritmos usando C**: fundamentos e aplicações. Rio de Janeiro: Ciência Moderna, 2007.

TENENBAUM, A. M. **Estruturas de Dados usando C**. São Paulo: Pearson Makron Books, 2009.

ZIVIANI, N. **Projeto de algoritmos com implementação em Pascal e C**. São Paulo: Pioneira Thomson Learning, 2002.

APRESENTAÇÃO DOS PROFESSORES RESPONSÁVEIS PELA ORGANIZAÇÃO DO MATERIAL DIDÁTICO

Os autores deste livro são docentes do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) /Campus Frederico Westphalen.

O professor **Cristiano Bertolini** possui graduação em Ciência da Computação pela UPF (Universidade de Passo Fundo), mestrado em Ciência da Computação pela PUCRS (Pontifícia Universidade Católica do Rio Grande do Sul), doutorado em Ciência da Computação pela UFPE (Universidade Federal de Pernambuco) e pós-doutorado pela United Nations University (Macau) ná área de Métodos Formais e Informática Aplicada à Saúde. Suas áreas de interesse envolvem, principalmente, Métodos Formais, Teste de Software e Engenharia de Software.

O professor **Fábio José Parreira** possui graduação em Ciência da Computação pelo UNITRI (Centro Universitário do Triângulo), Especialista em Produção de Material Didático para EaD pela UFAM (Universidade Federal do Amazonas), Mestrado em Processamento Digital de Imagens pela UFU (Universidade Federal de Uberlândia) e Doutorado em Inteligência Artificial e Informática de Sinais Biomédicos pela UFU (Universidade Federal de Uberlândia). Atualmente é Professor Associado do Departamento de Tecnologia da Informação no Campus de Frederico Westphalen - RS da UFSM (Universidade Federal de Santa Maria). Suas áreas de interesse envolvem, principalmente: Educação a Distância, Inteligência Artificial e Jogos Educacionais Digitais.

O professor **Guilherme Bernardino da Cunha** possui graduação em Ciência da Computação, mestrado em Ciências (2003) com Ênfase em Inteligência Artificial e Processamento Digital de Imagens e doutorado em Ciências, com ênfase em Engenharia Biomédica, pela Universidade Federal de Uberlândia. Atualmente é professor Adjunto da Universidade Federal de Santa Maria - Campus Frederico Westphalen. Tem experiência na área de Ciência da Computação, atuando principalmente nos seguintes temas: Engenharia de Software, Inteligência Artificial, Análise de Séries Temporais, Epidemiologia, Banco de Dados, Redes Neurais Artificiais, Algoritmos Genéticos e Bioinformática.

O professor **Ricardo Tombesi** Macedo possui graduação em Ciência da Computação pela Universidade de Cruz Alta (Unicruz), mestrado em Engenharia da Produção pela Universidade Federal de Santa Maria (UFSM) e doutorado em Ciência da Computação pela Universidade Federal do Paraná (UFPR). Participou do Programa de Doutorado Sanduíche no Exterior (PDSE) na Universidade de La Rochelle - França. Seus interesses de pesquisa consistem em redes sem fio, redes veiculares, gerenciamento de identidades e redes definidas por software.