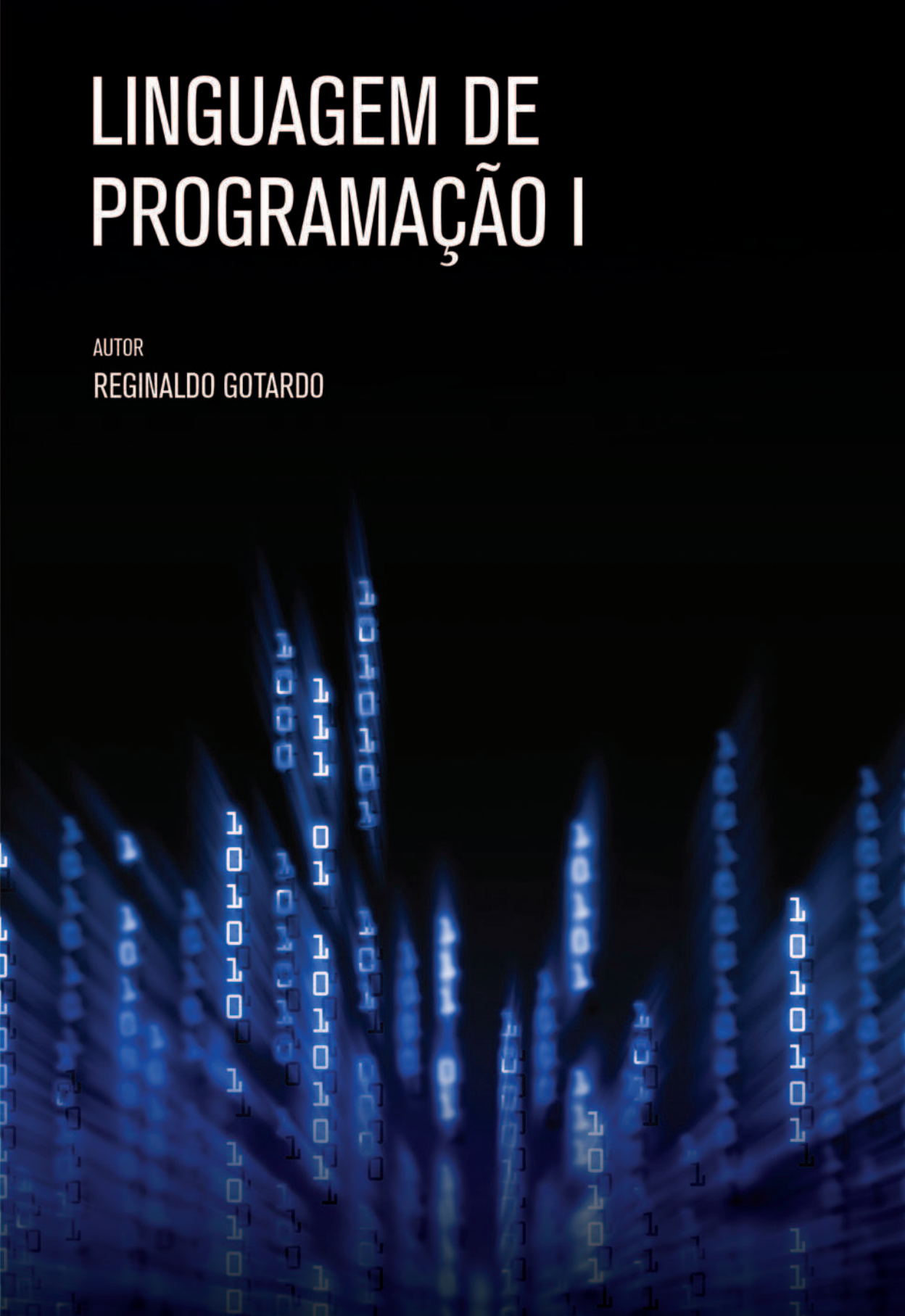
LINGUAGEM DE PROGRAMAÇÃO I

AUTOR
REGINALDO GOTARDO



LINGUAGEM DE PROGRAMAÇÃO

AUTOR
REGINALDO APARECIDO GOTARDO

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2015

Estácio

Conselho editorial REGIANE BURGER; ROBERTO PAES; GLADIS LINHARES.

Autor do original REGINALDO APARECIDO GOTARDO

Projeto editorial ROBERTO PAES

Coordenação de produção GLADIS LINHARES

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística JOICE KAROLINE VASCONCELOS DOS SANTOS, ALINE ZARAMELO SIMÕES E AMANDA DUARTE AGUIAR

Revisão de conteúdo LUIZ GIL SOLON GUIMARÃES

Imagem de capa ALEXMAX | DREAMSTIME.COM

Dados Internacionais de Catalogação na Publicação (CIP)

G683L GOTARDO, REGINALDO

Linguagem de programação I / Reginaldo Gotardo

Rio de Janeiro: SESES, 2015.

200 P. : IL.

ISBN: 978-85-5548-155-0

1. Linguagem. 2. Vetores. 3. Matrizes. I. SESES. II. Estácio.

CDD 005.133

Diretoria de Ensino — Fábrica de Conhecimento

Rua do Bispo, 83, bloco F, Campus João Uchôa

Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

"Por que codificamos, Bruce? Para ensinarmos computadores a resolver problemas mais rapidamente do que nós resolvemos."

A frase acima é uma brincadeira referenciando uma célebre frase do Filme Batman Begins. O pai de Bruce Wayne diz: Por que caímos, Bruce? Para aprendermos a levantar.

Escrever códigos é um grande desafio! Mas é fato também que, cada vez mais pessoas aprendem a programar. Constantemente precisamos de programação. Seja quando configuramos despertadores, micro-ondas, fornos especiais, os antigos vídeo cassetes que gravavam programas quando saímos de casa e não queríamos perde-los.

Agora programamos robôs, nossos carros, nossas casas com sensores para controle de temperatura do ar condicionado, fechadura eletrônica, aquecimento do chuveiro, iluminação!

Nós vamos explorar neste curso não apenas o ato de "Codificar" (ou codar para alguns! Rs), mas os recursos que estão associados a este ato.

Bons estudos!

1

Conceitos Fundamentais sobre Programação

Recentemente um filme chamado "O Jogo da Imitação" (The Imitation Game, 2014) chamou a atenção do público, particularmente a minha, pois mostra um pouco da história do matemático Alan Turing, considerado um dos "pais" da ciência computação. Não é um documentário, é uma ficção e eu não vou contar a história, claro! (rs). Mas, o que posso dizer é que, na minha humilde opinião, mostra o quanto a computação pode ser importante, o quanto a "programação de artefatos computacionais" pode ser relevante para a sociedade. E, claro, mostra Nerds assumindo o papel de heróis! (o máximo, com certeza). Neste capítulo, vou usar um pouco da lógica do Alan Turing para começarmos nossa discussão à respeito de programação de computadores!

OBJETIVOS

Neste capítulo, os objetivos principais são:

- Conceitos fundamentais da programação de computadores
- Algoritmos, Linguagens e Máquinas
- Ambiente de Desenvolvimento
- Linguagem de Estudo
- Introdução à linguagem C
- Entrada e saída na linguagem C

1.1 Conceitos Fundamentais e Máquinas de Turing

O conceito de um artefato (ou dispositivo, mecanismo) universal que realize computação foi descrito por Alan Turing em 1937, a chamada Máquina de Turing. Basicamente, uma Máquina de Turing descreve um mecanismo formal de computação, ou seja, é possível descrever a computação em si através de um conjunto de regras.

Um computador, basicamente, é um processador de dados, ou seja, um transformador de dados iniciais (dados de entrada) em dados finais (dados de saída). Como pode ser visto na figura 1.1, dados entram no nosso computador e dados saem do nosso computador. Neste caso, o modelo de transformação inserido na área de processamento é fixo: ele sempre atuará da mesma forma em dados de entrada.

Figura 1.1 – Computador de Único Propósito.

Este modelo de único propósito atua como uma caixa preta: o que há nele, normalmente, não é visto por quem o usa. Além disto, este modelo é usado, como o próprio nome diz, para resolver um único problema.

Hoje em dia você verá adiante que a noção de caixa preta pode atuar sobre modelos de propósito geral ou específico. Mas, voltaremos nesta parte do assunto.

Vamos a um exemplo de computador de propósito único. Na figura 1.2 você pode observar uma máquina chamada Hello. Nesta máquina você coloca uma nota de dois reais (R$ 2,00) e aperta um botão verde. Após isto cai uma latinha de refrigerante para você. Se você apertar o botão vermelho seu dinheiro é devolvido.

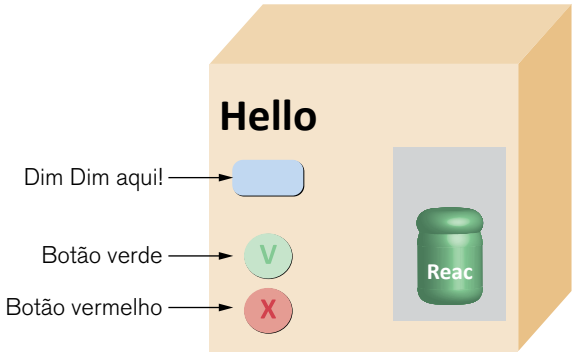


Figura 1.2 – Conheça a Hello, um computador de único propósito.

A Hello parece uma máquina interessante e podemos chama-la de computador, não? Afinal ela recebe dados de entrada (notas de R$ 2,00) e fornece a você dados de saída (uma lata de refrigerante). Mas, a Hello fornece lanches?

Nem precisamos ir muito longe. Você poderia perguntar:

- Ela fornecerá troco se eu colocar uma nota de R$ 5,00 (cinco reais)?

As respostas para estas perguntas serão sempre não. Afinal, Hello é um computador de propósito específico e só funciona da forma que falei. Podemos até melhorar a forma de funcionamento dela, mas ela sempre será uma máquina de refrigerantes.

Daí entra uma questão interessante. Questão que Turing também fez: e se fizermos um modelo programável? Sim, um computador que possa resolver algo que não precisa ser sempre a mesma coisa.

Um programa é um conjunto de instruções a serem realizadas pelo nosso "computador programável". Vamos chamar então (assim como os cientistas chamaram) este computador de: computador de propósito geral. É geral, pois podemos trocar o programa e resolver problemas diferentes com instruções diferentes.

Agora, como você pode observar na figura 1.3, os dados de saída dependem da combinação entre os dados de entrada e o programa inserido.

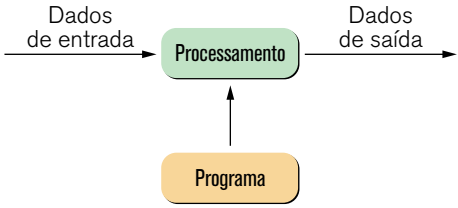


Figura 1.3 – Um computador de propósito geral.

As combinações possíveis são três:

a) Mantém o programa e mantém os dados de entrada – apesar de trivial é igualmente importante, pois garante corretude ao seu computador. Ele sempre responderá de maneira igual para o mesmo programa e para os mesmos dados de entrada. Nossa máquina Hello, por exemplo, sempre deverá um refrigerante se a nota for de R$ 2,00.

b) Mantém o programa, mas muda os dados de entrada – parecido com o modelo de único propósito, pois estaríamos mantendo o propósito (o programa) do computador. As saídas variam apenas em função de dados de entrada diferentes. Parecida com a máquina Hello. Você até poderia melhorar o programa para aceitar notas diferentes, mas o programa sempre devolveria um refrigerante.

c) Muda o programa, mas mantém os dados de entrada – imagine mudar o programa da máquina Hello para tocar música com R$ 2,00 ao invés de fornecer o refrigerante. Imagine mais: você indica o que fazer com a nota de R$ 2,00 (tocar música, voltar um chocolate, ou mesmo um refrigerante).

Vamos agora usar exemplos diferentes. Veja na figura 1.4 que ao trocar o conjunto de instruções (o programa) o computador devolve resultados diferentes para o mesmo conjunto de entrada.

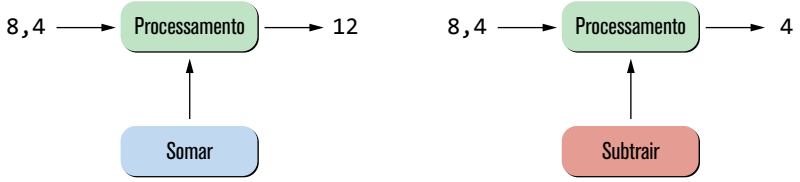


Figura 1.4 – Programas diferentes com o mesmo conjunto de entrada.

Já na figura 1.5, veja que dados de entrada diferentes produzem resultados diferentes para o mesmo programa. A ideia do programa, seu conjunto de instruções, se mantém (que é a soma dos dois números) e varia apenas pela variação nos dados de entrada.

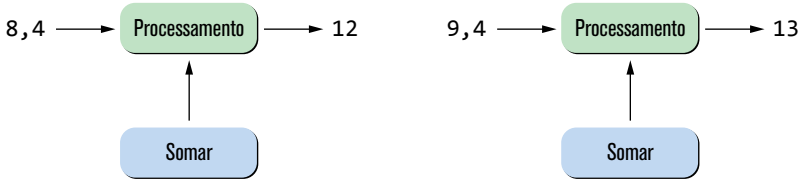


Figura 1.5 – Programas iguais com conjuntos de entrada diferentes.

Nós conhecemos esta proposta atualmente como programação de computadores. Mas, Alan Turing a formulou em 1936. Hoje, há uma área na computação chamada Teoria da Computação que estuda diversos aspectos matemáticos e formais sobre computabilidade. A chamada Máquina de Turing Universal é o conceito de um computador programável e foi a primeira descrição de um computador moderno. A Máquina de Turing sempre vai gerar a resposta desejada para algo que seja computável, desde que você forneça o programa (ou configuração) adequado.

CONEXÃO

O trabalho original do Alan Turing pode ser lido aqui:

http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf (em9/7/2015).

Dentro desse contexto inicial, também recomenda-se o filme "O Jogo da Imitação" (The Imitation Game - 2014), sobre a vida de Alan Turing.

Hoje em dia, como comentamos, ao vermos máquinas (ou computadores) resolvendo alguns problemas, fica difícil imaginar se dentro delas há um computador de único propósito ou propósito geral. Por exemplo, um caixa bancário tem lá dentro um computador convencional ou um computador que só sabe efetuar as operações bancárias? E os videogames? Será que conseguimos rodar um sistema como Windows ou Linux neles?

Nossa concepção de computador para uso diário está muito voltada ao propósito específico. Você verá no curso que é esta lógica a ser trabalhada. Vamos programar computadores.

No entanto, não usaremos uma Máquina de Turing e não criaremos programas para Máquinas de Turing.

Mas, por que não?

1.2 Algoritmos, Linguagens e o Modelo de von Neumann

As primeiras arquiteturas de computadores armazenavam dados em memória, mas os programas eram, geralmente, inseridos por combinações de comutadores ou de um sistema de fios. Isso requeria ligar e desligar a máquina para computações com programas diferentes.

John von Neumann, um matemático húngaro brilhante e mais um fantástico nome na computação (que você não deve deixar de pesquisar e estudar a biografia), propôs um modelo que determina ao programa também ser armazenado na memória. Conceitualmente, um programa é um conjunto de dados (instruções são dados) e ele poderia, assim como os dados de entrada, ficar também armazenado em memória e ser trocado quando necessário.

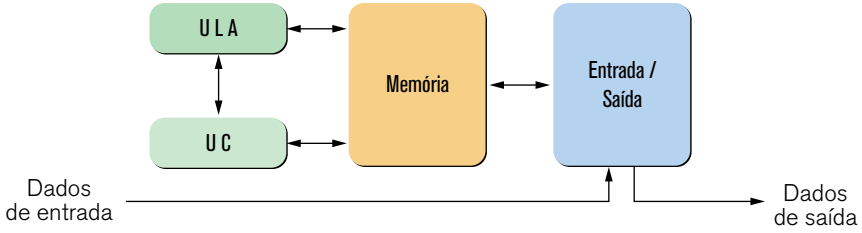


Figura 1.6 – Modelo de von Neumann.

Como pode ser visto na figura 1.6, os computadores construídos com base no modelo de von Neumann possuem quatro componentes principais:

- **Unidade Lógica e Aritmética (ULA):** um computador deve ser capaz de realizar operações básicas sobre dados. É nesta unidade que as operações acontecem.
- **Unidade de Controle (UC):** esta unidade é responsável por "orquestrar" as demais unidades do computador. Ela decodifica as instruções de um programa

num grupo de comandos básicos, blocos de construção, e gerencia como se dá a execução destes comandos com as demais unidades.

• **Memória:** é o local onde os programas e os dados são armazenados durante a execução do programa, ou seja, o processamento dos dados.

• **Unidade de Entrada e Saída:** esta unidade é responsável por receber dados externos ao computador e enviar de volta os resultados obtidos.

Na unidade de controle as instruções de um programa são executadas sequencialmente. Os programas com suas instruções e os dados a serem transformados, como vimos, estão em memória.

Isto é chamado de fluxo de execução!

Claro que podem ocorrer desvios de fluxo com instruções especiais e veremos isto ao longo de nosso curso.

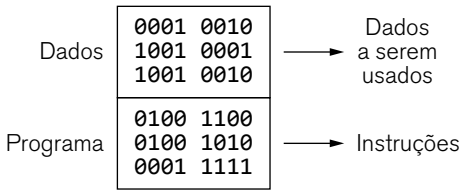


Figura 1.7 – Dados e Programas na Memória.

Tanto os dados quanto os programas em memória possuem o mesmo formato: são padrões binários, sequências de 0s e 1s. Mas, escrever programas usando 0s e 1s é muito complicado.

Para isto usaremos Linguagens de Programação. A base para usarmos linguagens de programação são os Algoritmos!

Se relembrarmos o conceito de algoritmo veremos que trata-se de uma sequência de passos (ou instruções) para resolver um problema. Algo como:

1. Leia um número
2. Leia outro número
3. Multiplique o primeiro pelo segundo
4. Mostre o resultado

Logo, algoritmos são a base para construção de programas. Nós, naturalmente já escrevemos ou executamos algoritmos.

Para escrevermos nossos programas então precisaremos aprender uma linguagem de programação que siga conceitos que entendemos algoritmicamente. Vamos entender então o que há nesse processo de escrita de programas.

1.3 Linguagens de Programação, Compilação e Interpretação

Uma linguagem de programação é um método padronizado que usamos para expressar as instruções de um programa a um computador programável. Ela segue um conjunto de regras sintáticas e semânticas para definir um programa de computador. Regras sintáticas dizem respeito à forma de escrita e regras semânticas ao conteúdo.

Através da especificação de uma linguagem de programação você pode especificar quais dados um computador vai usar; como estes dados serão tratados, armazenados, transmitidos; quais ações devem ser tomadas em determinadas circunstâncias.

Ao usarmos uma linguagem de programação você cria o chamado "Código Fonte". Um código fonte é um conjunto de palavras escritas de acordo com as regras sintáticas e semânticas de uma linguagem.

Neste momento você lê um livro escrito usando a Linguagem Portuguesa-Brasileira (rs). Você está lendo o código fonte do livro e processando as informações. Mas, calma, você não é um computador! Eu também não, afinal, não sei se computadores podem ter senso de humor tão apurado (rs).

O importante para sabermos agora é que o código fonte não é executado pelo computador. O computador não entende o código fonte, mas sim o código que representa seu programa e dados em memória (nos nossos exemplos o código é binário!).

Assim, é preciso traduzir o código fonte para o formato que o computador entenda.

Este formato compreensível pelo computador é chamado de "Código de Máquina".

A este processo de "tradução" é dado o nome de Compilação.

Compiladores são programas especiais que traduzem códigos fontes em códigos alvos (ou código objeto). Trata-se de um código que é executável num computador.

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

int main(int argc,  char  *argv[]  {
    printf(”Hello C World\n”);
    getchar();
    return 0;
}
```

```
0010 0100 1001
1100 0110 0000
1101 0010 0001
```

Figura 1.8A – Traduzindo Códigos.

Como pode ser visto na figura 1.8A, um programa em linguagem C, por exemplo, pode ser traduzido para uma linguagem compreensível para o computador. A compilação de um programa em linguagem C na prática produz um programa em linguagem de montagem, uma linguagem que o processador interpretará.

Esta espécie de “Camada” que atua sobre os “Programas Reais”, em linguagem de montagem, no computador, foi o que permitiu a popularização da escrita de programas de computador. Era muito difícil de aprender tais linguagens e cada arquitetura (reunião entre o sistema operacional do computador e o hardware do computador) funcionava de forma diferente. Sabe o código da figura 1.8? Veja como fica o código de montagem dele:

```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello C World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        movl    $.LC0, %edi
        call    puts
        call    getchar
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE2:
        .size   main, .-main
        .ident  "GCC: (GNU) 4.9.2 20141101 (Red Hat 4.9.2-1)"
        .section        .note.GNU-stack,"",@progbits
```

Figura 1.8B – Traduzindo Códigos.

É mais fácil de entender o código da figura 1.8, não? Mesmo não sabendo qual linguagem há lá é possível ter uma ideia do propósito dele. Veja pela figura 1.9 que o programa objeto, traduzido, funciona como nossos exemplos anteriores para computadores de propósito geral. Ele é inserido na máquina que transformará dados de entrada em dados de saída com base nas instruções do programa traduzido.

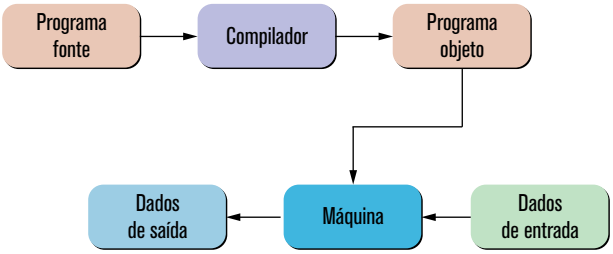


Figura 1.9 – Processo de Compilação.

O uso dos compiladores para a criação de programas insere para nós, desenvolvedores, dois conceitos de tempo:

- Tempo de Compilação
- Tempo de Execução

No tempo de compilação é o compilador que está trabalhando traduzindo seu código fonte num código alvo. Neste caso, sua interação será com ferramentas de edição e serão mostrados erros ou alertas na compilação que possam inviabilizar esta tradução.

O tempo de execução compreende seu código já traduzido e funcionando agora como um programa numa arquitetura computacional para fazê-lo operar. Neste caso seu programa vai operar com entradas, saídas e processamento. Um ponto importante a considerar é que mesmo seu código tendo sido compilado com sucesso não garantirá sucesso em tempo de execução. Podem existir erros de lógica, erros de gerenciamento de memória, operações inválidas não tratadas (como divisão por zero), dentre outros.

Se pensarmos no usuário final dos nossos programas é muito importante que erros de execução sejam evitados. É necessário trabalhar muito o código, testá-lo das mais variadas formas a fim de garantir ou minimizar problemas em

tempo de execução, pois o usuário final, dificilmente terá recursos para lidar com estes problemas.

Resumindo então:

"Programas são a representação de algoritmos que foram escritos em alguma linguagem de programação e compilados para uma linguagem alvo que permite ser executada num dispositivo computacional"

Para termos um algoritmo é preciso:

- Que as instruções sejam um número finito de passos;
- Que cada passo esteja precisamente definido, sem possíveis ambiguidades;
- Existam zero ou mais entradas tomadas de conjuntos bem definidos;
- Existam uma ou mais saídas;
- Exista uma condição de fim sempre atingida para quaisquer entradas e num tempo finito;

Para escrita de nossos programas nós usaremos a Linguagem C. Antes, falaremos um pouco do ambiente de desenvolvimento das nossas soluções.

1.4 Preparação do Ambiente de Desenvolvimento

Para desenvolvimento de nossos programas nós usaremos o Dev-cpp da Bloodshed. Trata-se de um software para edição, depuração e compilação de programas em C e C++. Vamos apelida-lo de DEV.

CONEXÃO

Você pode baixa-lo em (acesso em março de 2015):
http://www.bloodshed.net/dev/devcpp.html

Nós estamos usando a versão 5.0 beta (4.9.9.2) com Mingw/GCC 3.4.2 (que é a versão do compilador). Os meus testes foram feitos num ambiente com Windows 8.1 64bits.

Como você já notou, nossos programas serão escritos usando Linguagem C. Em breve falaremos mais dela e teremos uma apresentação mais formal!

CONEXÃO

Você pode usar também os códigos num compilador online (acesso em março de 2015):
http://www.tutorialspoint.com/compile_cpp_online.php
Este compilador online usa o GCC v4.8.3:
https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/

Para criar um novo projeto no DEV você precisa clicar em arquivo -> novo -> projeto. Escolha um projeto do tipo Console Application (a entrada e saída de nosso projeto será via terminal de acesso – console). Escolha também um "Projeto C". Dê um nome ao seu projeto e pronto! Podemos começar!

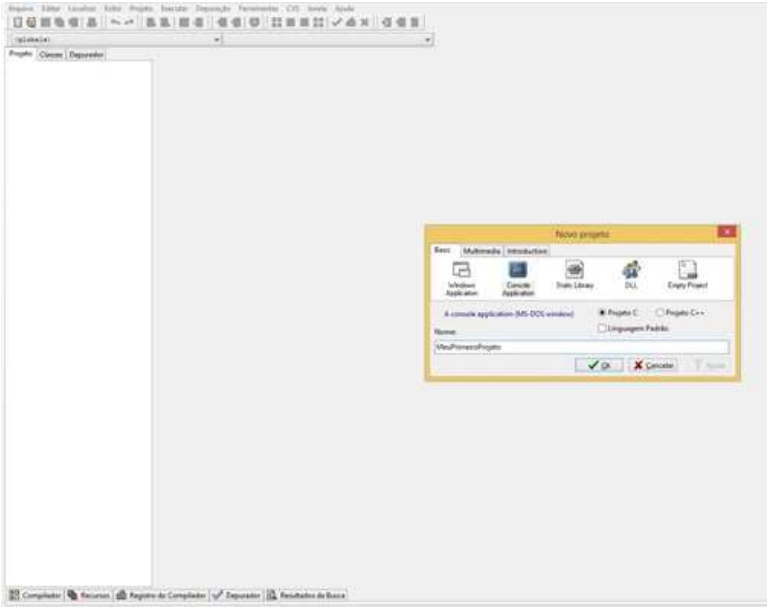


Figura 1.10 – Criando um novo projeto no Dev-cpp.

Assim que você salva o projeto a primeira tela que se abre é um arquivo main.c. Este arquivo, normalmente, já conta com uma estrutura base de um programa em C. Uma estrutura mínima, como pode ser visto na figura 1.11.

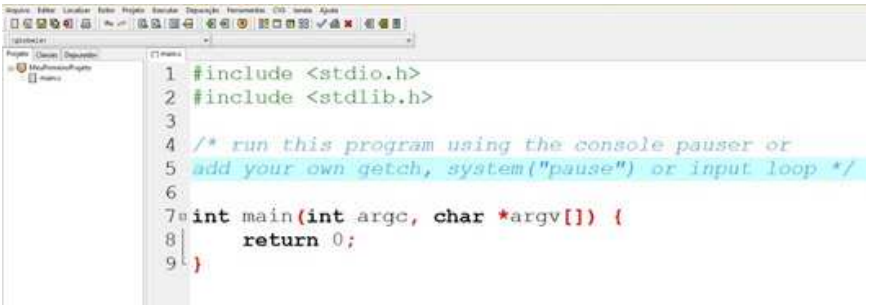


Figura 1.11 – Arquivo .C básico no DEV.

Você pode substituir o conteúdo do código por este:

Código 1.1

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("Hello C World\n");
    return 0;
}
```

Agora podemos realizar nosso primeiro teste clicando no Menu Executar, depois "Compilar & Executar". Se for pedido, salve o arquivo main.c na mesma pasta do seu projeto. Você verá a tela a seguir, na figura 1.12.

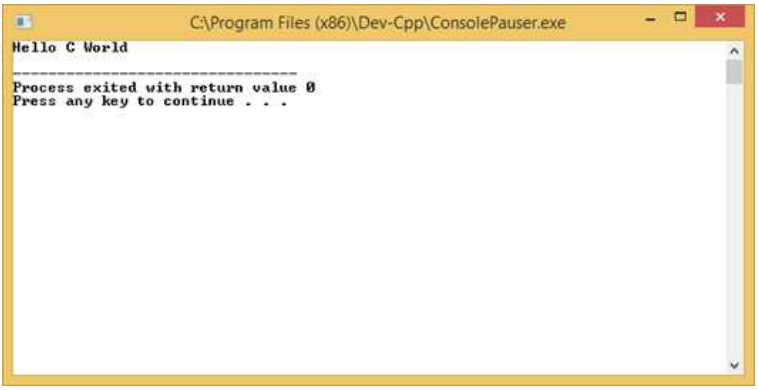


Figura 1.12 – Resultado do nosso primeiro programa.

Caso o processo seja rápido e esta tela não apareça para você altere seu código para o seguinte:

Código 1.1 - Alterado

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("Hello C World\n");
     getchar();
    return 0;
}
```

Esta alteração que fizemos na linha 5 do código foi para os casos em que o compilador não coloca um mecanismo de parada (pause) automático para o usuário. O getchar() serve para leitura de caracteres, mas, por hora, para aguardar uma digitação do usuário para seguir.

Quando você precisar adicionar arquivos ao seu projeto (criar uma biblioteca.h) você fará como indicado na figura 1.13.

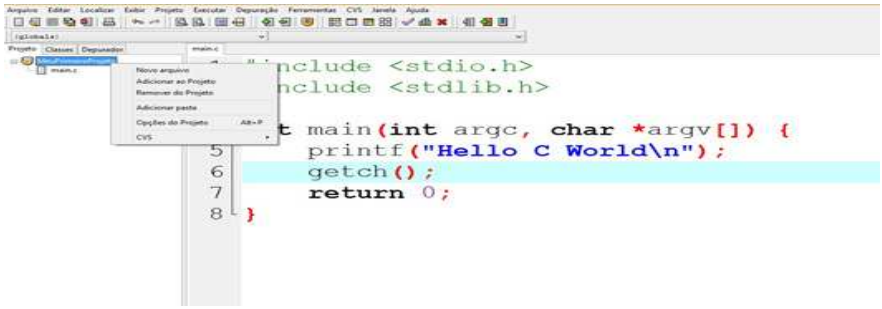


Figura 1.13 – Inserindo novos arquivos ao projeto.

Para isto, você clicará com o botão direito do mouse no projeto e adicionará novos arquivos. Usaremos este recurso mais adiante.

Agora vamos ver como usar o compilador online. Veja na figura 1.14 que ele possui já integrado um console na parte de baixo onde podemos digitar comandos.

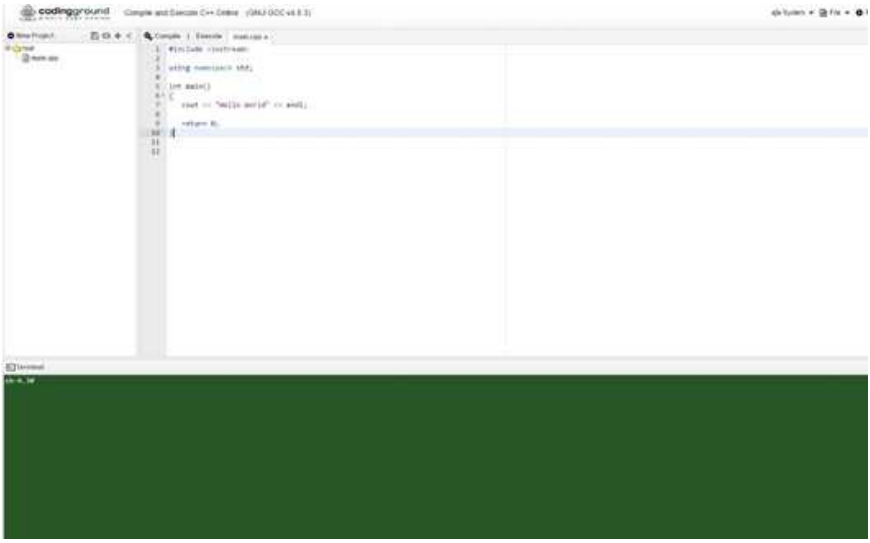


Figura 1.14 – O Compilador online.

Ao entrar no link indicado veja que já há um pequeno projeto também criado. Só que está em C++ (fora do escopo de nossa disciplina).

Minha sugestão é que você feche o arquivo e o exclua, clicando nele com o botão direito. Depois crie novamente outro arquivo, mas, desta vez, o renomeie para main.c (você também pode apenas renomear o arquivo atual para main.c se desejar).

Escreva o código 1 alterado no corpo do seu projeto.

Após isto, clique na região do console (a região abaixo do editor) e digite o seguinte comando:

```
gcc -o main main.c
```

Feito isto, caso não ocorram erros, você digitará o próximo comando (caso ocorram erros a dica é: não copie e cole o código, digite-o novamente, pois estamos usando sistemas diferentes para compilação. Os testes no DEV estão no Windows, os testes online são em Linux!):

```
./main
```

O resultado pode ser conferido na figura.

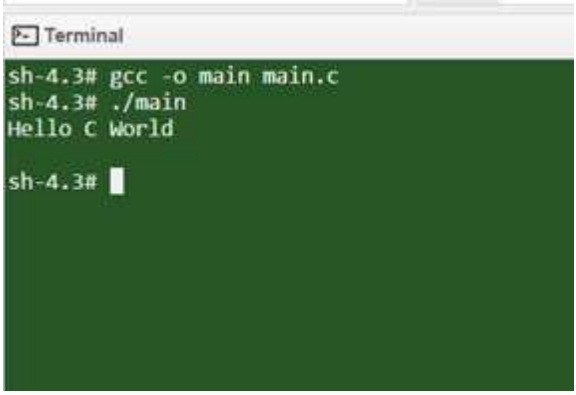


Figura 1.15 – Nosso primeiro programa no compilador online.

Esclarecendo então, nossos testes no DEV são no Windows 8.1, como mencionei. O compilador online simula um ambiente Linux. Logo, os comandos que digitamos são válidos se você usar um Sistema Operacional Linux.

1.5 A Linguagem C

A Linguagem C foi criada por Dennis M. Ritchie e Ken Thompson no laboratório Bell em 1972. Ela foi baseada na linguagem B, de Thompson. A definição da linguagem C está no livro *The C programming language*, escrito por Brian W. Kerninghan e Dennis M. Ritchie.

Rapidamente a linguagem C se tornou uma das linguagens mais populares entre os programadores, pois é poderosa, portátil, flexível e foi desenhada para ser de propósito geral, estruturada, imperativa, procedural e padronizada.

CONEXÃO

Há várias implementações para a Linguagem C e há um padrão ANSI para ela:

- http://www.le.ac.uk/users/rjm1/cotter/page_47.htm
- http://en.wikipedia.org/wiki/ANSI_C

A linguagem C é muito próxima da linguagem algorítmica que aprendemos no início de cursos de computação. Isto ocorre, pois ela é imperativa e estruturada. Vejamos um exemplo de algoritmo:

Código 1.1

```
Algoritmo
Ler um valor para a variável A
Ler um valor para a variável B
Efetuar a adição das variáveis A e B, atribuindo o
resultado a variável X
Apresentar o valor da variável X após a operação de
adição
```

A tradução deste algoritmo para linguagem C fica assim:

Código 1.2

```
#include<stdio.h>
main()
{
    int x;
    int a;
    int b;
    scanf("%d", &a);
    scanf("%d", &b);
    x = a + b;
    printf("%d\n",x);
}
```

Um programa em C possui uma estrutura básica, por isto a linguagem é chamada de estruturada:

Código 1.3

```
[inclusão de bibliotecas]
[declaração de dados globais]
[declaração dos protótipos de funções]
void main (void)
{
 [corpo do programa principal]
}

[implementação das funções do programa]
```

Observe no código 1.4 a seguir como um exemplo de C compreende várias estruturas e um formato padrão. Na linha 1 temos uma chamada ao pré-processador, como cabeçalho, para incluir informações. O uso de "< >" indica que o arquivo a ser incluído está num local padrão.

Código 1.4

```
#include <stdio.h>
int main(void)
{
   printf(“Programar em C eh facil.\n");
   return 0;
}
```

Já na linha 2 há a definição do início de uma função, usando palavras-chave como **int** e **void**. Esta função main é a função principal que retorna um inteiro e não recebe parâmetros (a palavra void significa "vazio").

A linha 3 indica o início de um bloco de comandos. A linha 6 indica o final do bloco de comandos.

A linha 5 serve para informar ao sistema operacional o que ocorreu, através de uma declaração de retorno. Trata-se de um conjunto de boas práticas, apesar de não ser obrigatório o seu uso.

A linha 4 executa a “lógica de negócios” da aplicação, ou seja, faz a tarefa do nosso algoritmo. Trata-se da função printf que serve para mostrar algo na saída padrão. Esta função tem sua definição e implementação em alguma biblioteca (através do cabeçalho). A cadeia a ser exibida termina com uma nova linha (\n) e a maior parte das declarações em C terminam com ponto-e-vírgula (;).

Um ponto importante a informar é que C é sensível à caixa (case sensitive). Assim, escrever main é diferente de escrever Main. Tome bastante cuidado com isto ao longo dos seus programas.

1.6 Variáveis e Tipos Básicos, Armazenamento de Dados na Memória

A linguagem C oferece quatro tipos básicos de dados:

- Inteiro: ex. 4, -8, 12
- Ponto flutuante: ex. 0.12, 0.75
- Precisão dupla: ex. 312E+8, 1E-3
- Caractere: ex. A, b, I, 3

Para usar estes tipos básicos em C é preciso declarar as variáveis antes de usá-las.

Código 1.5

```
#include <stdio.h>
int main(void)
{
   int soma;
   soma = 500 + 15;
   printf(“A soma de 500 e 15 eh %d\n", soma);
}
```

Veja pelo código 1.X que, na linha 4, há a declaração da variável “soma” capaz de armazenar um valor inteiro. Na linha 5 temos uma operação de atribuição na qual soma recebe o resultado de outra operação: a adição entre os valores 500 e 15.

Por fim, o printf é usado para apresentar o resultado na saída padrão com uma formatação de saída para número inteiro. Ele processa o texto inserido, substituindo o %d pelo valor da variável soma. Para que a substituição seja processada corretamente é preciso indicar que soma é do tipo inteiro. Logo, %d é chamado de máscara de formatação de saída.

Temos também os seguintes caracteres de formatação que podem ser usados:

- %c – Caractere
- %d - Decimal com sinal
- %i - Inteiro com sinal
- %f - Números reais
- %o – Octal
- %x – Hexadecimal
- %s – String - conjunto de caracteres
- %s – Cadeia de caracteres
- %e – precisão dupla
- %i – inteiro decimal com sinal
- %E – Notação científica (E maiúsculo)
- %p – ponteiro (endereço)

- %n – ponteiro (inteiro)
- %% - imprime o caractere %

Ao definir nomes para variáveis procure relacionar o nome ao significado de conteúdo dela, por exemplo:

- Media_dos_Valores
- Nota1
- Nome_do_usuario

E evite nomes compactados que sejam de difícil interpretação:

- Mdv, N, Nu (eles só têm significado para o programador e durante o desenvovimento do programa. Passado um tempo, nem mesmo o programador vai se lembrar).

A construção de variáveis na linguagem C deve seguir as seguintes regras:

- Deve começar com uma letra ou "_"
- Pode conter a-z, A-Z, 0-9, _

Por exemplo, não são válidos os seguintes nomes:

- Soma$ (pois contém caractere $)
- Variavel importante (pois contém espaço)
- 3a_variavel (não começa com letra ou _)
- printf (é uma palavra reservada da linguagem).

Todo bom programa em C deve ser bem documentado. Isto é uma boa política de desenvolvimento e manutenção.

Código 1.6

```
/* Este programa calcula a soma de dois
   numeros inteiros                    */
#include <stdio.h>
int main(void)
{
   int soma;  // Variável que guarda a soma
   soma = 500 + 15;  // Soma dos valores
   printf("A soma de 500 e 15 eh %d\n", soma);
   return 0;
}
```

Veja no código 1.6 que os comentários podem ser feitos usando /* e */ para múltiplas linhas e // para comentário de linha única a partir das barras.

No código 1.7 temos agora o uso dos diversos tipos de variáveis.

Código 1.7

```
#include <stdio.h>
int main(void) {
   int    soma;
   float  dinheiro;
   char   letra;
   double pi;
   soma = 10;        // Atribuir um inteiro
   dinheiro = 2.21; // Atribuir um float
   letra = ´A´;       // Atribuir um caracter
   pi = 31415E-4;  // Eis o pi!
   printf("Aqui esta um inteiro (%d) e um ", soma);
   printf("número de ponto flutuante (%f)\n",
dinheiro);
   printf("Eis a letra ´%c´ e o valor de pi: %e\n", letra,
 pi);
   return 0;
}
```

Nas linhas 3 a 7 declaramos 4 tipos de variáveis diferentes. Você pode ver pelos códigos nas linhas 11 a 14 que usamos máscaras diferentes para as variáveis.

A seguir, veja na tabela o conjunto de palavras reservadas da linguagem C:

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

Vejam também na tabela abaixo o resumo dos tipos de dados da linguagem C:

TAMANHO EM BYTES	TIPO	ESCALA
0	void	Ausência de tipo
1	char	-127 a 127
1	unsigned char	0 a 255
1	signed char	-127 a 127
4	int	-2.147.483.648 a 2.147.483.647
4	unsigned int	0 a 4.294.967.295
4	signed int	-2.147.483.648 a 2.147.483.647
2	short int	-32.768 a 32.767

TAMANHO EM BYTES	TIPO	ESCALA
2	unsigned short int	0 a 65.535
2	signed short int	-32.768 a 32.767
4	long int	-2.147.483.648 a 2.147.483.647
4	signed long int	-2.147.483.648 a 2.147.483.647
4	unsigned long int	0 a 4.294.967.295
4	float	Seis digitos de precisão
8	double	Dez digitos de precisão
10	long double	Dez digitos de precisão

A declaração de variáveis é feita da seguinte forma:

Padrão:

```
tipo_de_dado lista_de_variáveis;
```

```
Ex:
o int idade, valor;
```

Declaração com inicialização da variável:

```
tipo_de_dado nome_da_variável  =  valor;
Ex.:
o int idade = 10;
```

Outro conceito importante sobre variáveis é em relação ao seu escopo. Uma variável só existe dentro do bloco no qual ela foi criada. O código abaixo retrata uma variável com escopo local à função main.

Código 1.8

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

Já o código a seguir mostra uma variável que poderá ser acessada em qualquer região do programa, pois foi declarada fora do escopo da função. Esta variável é chamada de variável global.

Código 1.9

```
#include <stdio.h>
int numero;
int main ()
{
  numero = 20;
  putchar (numero);
}
```

Além das variáveis, outro conceito importante sobre o armazenamento de dados em C é o de Constantes. São valores que não serão modificados ao longo do programa, mas devem ser armazenados também na memória.

Em C, os tipos constantes são:

- Caractere (char): ‘d’, ‘&’ - envolvidos por aspas simples;
- Inteiro (int): 20, 17 - números sem componente fracionário;

- Real (float,double): 23.56 - número com componente fracionário;
- Strings (char vet[14]): "Estácio" - caracteres envolvidos por aspas duplas.

Usa-se em C a barra invertida para descrever caracteres da tabela ASCII que não podem ser definidos usando aspas simples, por exemplo:

- \n - nova linha;
- \a - beep sonoro;
- \0 - fim da cadeia de caractere.

1.7 Bibliotecas e Link Edição

Todo compilador em C possui uma biblioteca padrão de funções. Estas bibliotecas facilitam a realização de tarefas como ler uma variável do usuário (usando a entrada padrão) ou imprimir um resultado (usando a saída padrão).

Tais bibliotecas devem ser inseridas num programa em C. Isto ocorre com a inserção das funções usadas no código executável. Este processo é conhecido como link edição.

Como pode ser visto na figura 1.16, na compilação um programa executável é gerado a partir de um código fonte em C. Isto é feito usando-se o arquivo .C (ou os arquivos) e os arquivos .h (bibliotecas do programa, escritas pelo programador). Estes arquivos são compilados, gerando um arquivo .obj, um programa objeto. Na sequência, são adicionadas as bibliotecas padrão da linguagem ao programa .obj, gerando o executável .exe.

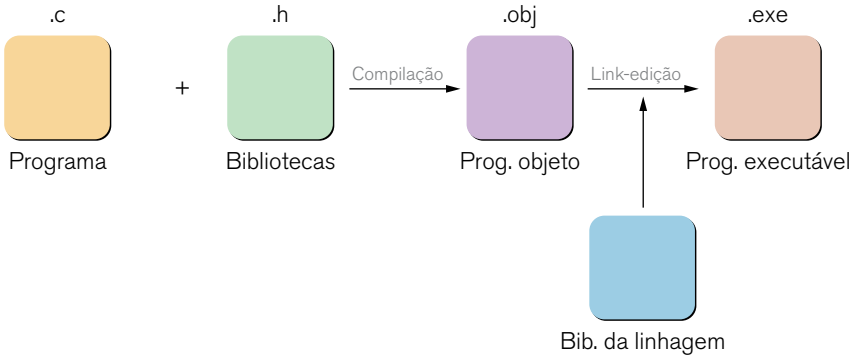


Figura 1.16 – Processo de Link Edição.

1.8 Operadores

Um dos principais operadores na linguagem C é o operador de atribuição "=". É com ele que você "atualiza" o valor de uma variável na memória.

Código 1.10

```
#include <stdio.h>
void main ()
{
  int Inteiro = 15;
  char primeiroCaracter = 'd'
  float mesada = 210.48;
}
```

A seguir vamos listar e agrupar por tipo os principais operadores que usaremos neste material.

1.8.1 Operadores Aritméticos

São usados para as operações de cálculo na linguagem.

OPERADOR	DESCRIÇÃO
*	Multiplicação
/	Divisão
%	Resto
+	Adição
-	Subtração

Por exemplo, o código a seguir usa o operador de resto (%) para calcular qual a sobra da divisão de dois inteiros (10 e 3). O resultado do código será 1.

Código 1.11

```
#include <stdio.h>
int main ()
{
  int var1 = 10;
  int var2 = 3;
  int var3 = var1 % var2;

  printf("\nResto de 10 por 3 eh: %d", var3);
}
```

1.8.2 Operadores Relacionais

São usados para avaliar o resultado de uma expressão lógica. O resultado é 1 se for verdadeira e 0 se for falsa.

OPERADOR	DESCRIÇÃO
<	Menor que
<=	Menor ou igual
>	Maior que
>=	Maior ou igual
==	Igual
!=	Diferente

Por exemplo, veja no código 1.12, na linha 5, que "perguntamos" se 10 é maior que 5, usando o operador >. O resultado será 1.

Código 1.12

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

int main(int argc, char *argv[]) {
    printf("10>5 eh %d", (10>5));
    getchar();
    return 0;
}
```

1.8.3 Operadores de Incremento e Decremento

São dois operadores bastante usados para simplificar expressões:

++ (incremento de 1)
-- (decremento de 1)

Podem ser colocados antes ou depois da variável a ser incrementada ou decrementada.

Se forem inseridos antes modificam o valor antes da expressão ser usada e, se inseridos depois, modificam depois do uso.

Operadores Lógicos

Refere-se a forma como os relacionamentos podem ocorrer entre expressões lógicas. São usados com expressões que retornam verdadeiro ou falso, chamadas booleanas, em testes condicionais.

AÇÃO	OPERADOR	EXPLICAÇÃO
AND	&&	Retorna verdadeiro se ambos os operandos são verdadeiros e falso nos demais casos.
OR	||	Retorna verdadeiro se um ou ambos os operandos são verdadeiros e falso se ambos são falsos.
NOT	!	Usada com apenas um operando. Retorna verdadeiro se ele é falso e vice-versa.

O formato de utilizá-los é o seguinte:

```
(expressão_lógica) && (expressão_lógica)
(expressão_lógica) || (expressão_lógica)
!(expressão_lógica)
```

Por exemplo:

```
(10>5) && (!(10<9)) || (3<=4)
```

Retornará... Verdadeiro (ou seja, 1).

Escreve o programa a seguir, compile e o execute. Depois, confirme o resultado.

Código 1.13

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int varteste = (10>5) && (!(10<9)) || (3<=4);
    printf("resultado da expressao eh ", varteste);
    getchar();
    return 0;
}
```

- Sejam p e q expressões lógicas

1.8.4 Precedência dos Operadores nas Expressões Lógicas

Ao usar mais de um operador, de forma composta, na sua expressão lógica, eles trabalharão com a precedência abaixo:

```
maior
    !
    >  >=  <  <=
    ==  !=
    &&
    ||
menor
```

Você pode mudar esta precedência com o uso de parênteses.

1.8.5 Operador condicional (ternário)

Tem a forma genérica:

```
variável = (expr1) ? (expr2) : (expr3)
```

Substitui declarações condicionais do tipo se-então tipo if-else.
Por exemplo:

```
x = (var==0) ? 2 : 3;
```

O exemplo acima é equivalente a:

```
if( var == 0 )
x = 2;
else
x = 3;
```

Veremos mais detalhes na unidade a seguir sobre operadores condicionais ou de tomada de decisão.

1.8.6 Operadores de endereço

Toda informação armazenada na memória do computador possui um valor e ocupa um determinado espaço. Desta forma, para que se possa ter acesso às informações, a memória é organizada por endereços. São usados basicamente com ponteiros. Também veremos mais sobre ponteiros no capítulo 2.

& (endereço do operando) - retorna o endereço de memória de uma variável, poedndo ser armazenado numa variável do tipo ponteiro.

* (valor no endereço do operando) - retorna o valor armazenado em um determinado endereço de memória.

1.8.7 Operador sizeof

Como já foi visto anteriormente, cada tipo de dado possui um tamanho diferente em bytes, ou seja, podem ocupar mais ou menos espaço na memória.. Desta forma, o operador sizeof retorna o número de bytes ocupados pelo operando, que pode ser uma variável ou um tipo genérico de dado.

Por exemplo:

```
sizeof( float ); //retorna 4, que é o número de bytes de dados tipo float.

char str[] = "valor";

sizeof (str); //retorna 6 (5 caracteres mais o nulo do final da string).
```

1.9 Entrada e Saída

Para que seja possível receber dados do usuário de um programa e para que seja possível devolver resultados a ele nós usaremos comandos de entrada e saída (comandos de E/S) ou input/output (I/O).

O comando printf funciona da seguinte maneira:

```
printf ("série_de_controle", lista_de_argumentos)
```

"série_de_controle" é uma série de caracteres e comandos de formatação de dados que devem ser impressos , ou seja, é uma espécie de máscara de impressão que pode conter dados fixos e, para os variáveis, a forma com que devem ser impressos e “lista_de_argumentos” representa as variáveis e constantes que devem ser trocados pelos formatos especificados na série de controle.

No código 1.14 podemos ver o uso da variável global que será constante (não poderá ser alterada) na linha 2. A lógica do programa envolve calcular a área de um círculo. Veja na linha 7 que usamos o printf apenas com "série de controle". Na linha 9 usamos o printf também com lista de argumentos, no caso o raio. O valor de raio será substituído na impressão no local onde consta %5.2f. Assim, raio terá uma máscara para número de ponto flutuante. O valor 5.2 indica que o número deverá indicar até 5 dígitos sendo 2 depois do ponto decimal (parte fracionária do número).

Código 1.14

```
#include <stdio.h>
const float pi = 3.1415;
int main ()
{
  float area, raio = 3;
  area = pi * raio * raio;
  printf("\nCalculo -> area = pi*raio*raio");

  printf("\nRaio do circulo = %5.2f", raio);
}
```

É possível usar os caracteres de formatação de máscara e algumas opções extras, por exemplo:

- '\n' salta para uma nova linha;
- "%-5.2f" número em ponto flutuante (f) deve ser apresentado com no mínimo 5 dígitos, sendo 2 dígitos para a parte fracionária do número e deve ser justificado à esquerda;
- "%5.2f" número em ponto flutuante (f) deve ser apresentado com no mínimo 5 dígitos, sendo 2 dígitos para a parte fracionária do número e deve ser justificado à direita;
- "%10s" string(s) deve ser apresentado em 10 espaços justificado a direita;
- "%-10s" mesma coisa que o anterior, só que justificado à esquerda;
- "%5.7s" string(s) deve ser apresentado com pelo menos 5 caracteres e não mais que 7 caracteres.

Para a leitura dos dados do usuário é possível usar um conjunto vasto de funções. Uma delas é o scanf que tem a seguinte estrutura:

```
scanf("série_de_controle",lista_de_endereços_dos_argumentos)
```

No scanf os dados são lidos de formata fomatada, ou seja, já com suas respectivas representações (tipo de dados). Os formatos usados na série de controle da função scanf são semelhantes aos formatos usados pela função printf.

Temos o getchar() que espera até que seja digitado um caractere no teclado. Então ele mostra o caractere na tela e sai do processo de entrada de dados.

Temos getch() que efetua a leitura de um caractere do teclado sem ecoar o caractere na tela e o gets() que efetua a leitura de um string de caracteres digitado por meio do teclado até que seja digitado <ENTER>. É importante citar que o caractere da tecla <ENTER> não faz parte do string e no seu lugar é colocado o caractere de fim de cadeia ('\0').

Para escrita na saída padrão há ainda o putchar() que escreve um caractere a partir da posição corrente do cursor na tela. E o puts() que escreve um string na tela seguido por uma nova linha ('\n'). O puts somente opera com argumentos string (cadeias de caracteres).

Código 1.15

```
#include <stdio.h>
int main ()
{
  char nome [10];
  int anonascimento, idadeatual;
  printf ("\nQual e o seu nome: ");
  gets(nome);
  printf("Em que ano voce nasceu: ");
  scanf("%d", &anonascimento);
  idadeatual = 2003 - ano nascimento;
  printf("%s, voce tem em %d anos", nome, idadeatual);
  getchar();
}
```

Veja por exemplo no código 1.15 que declaramos uma variável caractere na linha 4. Por hora, não entraremos no detalhe dos [20], pois veremos isto adiante em vetores. Para leitura do nome usamos, na linha 7, o gets, que retornará o conteúdo lido do usuário. Para leitura do ano de nascimento usamos o scanf.

O & (e comercial) é muito importante, pois indica que o valor a ser lido será colocado num endereço de memória. Em linguagem C é comum indicar o endereço da memória e não apenas a variável em si. Falaremos mais disto e também sobre conceitos de passagem de parâmetro por valor e por referência.

Após o usuário clicar em algo a aplicação finalizará (devido ao uso do getchar()).

Agora, vejamos como fica a memória (ou uma parte dela) enquanto o usuário interage com a aplicação.

Imagine que cada "caixinha" ou "casinha" na memória forneça o espaço de 1 byte, conforme esquematizado na figura 1.17A. Repare que a numeração da direita simboliza o que seriam os endereços de memória, avançando de 1 em 1 byte. Sendo assim, a variável nome precisará de 10 bytes de espaço organizados por índices numerados de 0 a 9 conforme os números da esquerda (cada posição pode receber um caractere diferente compondo a variável nome. O tipo char ocupa 1 byte para cada caractere. As variáveis ano nascimento e idade atual ocuparão 4 bytes cada uma, pois este é o tamanho do tipo inteiro em C (int).

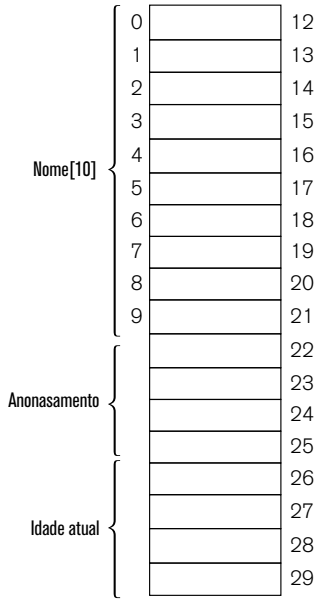


Figura 1.17 A – Estado da memória no programa de exemplo de coleta de dados do usuário.

Após a execução da linha 7 do programa 1.15, se o usuário digitar MARIA então o conteúdo da memória ficará como a figura 1.17B mostra. Perceba que, por mais que o nome tenha no máximo 10 valores de char, MARIA ocupa 6 posições (5 de conteúdo mais o \0 no final que marca o término na string). Vale ressaltar aqui, que na verdade os valores armazenados não são as letras, mas código numérico referente a cada uma delas.

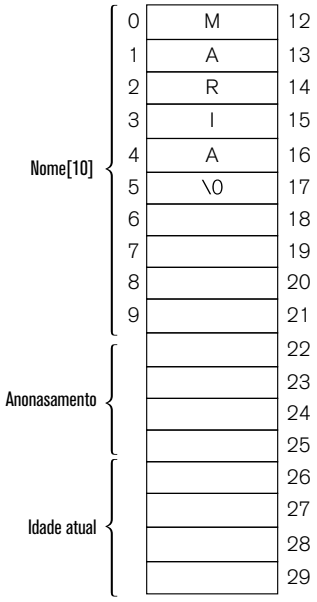


Figura 1.17 B – Estado da memória no programa de exemplo de coleta de dados do usuário.

Após o usuário digitar o valor pedido na linha 9 e o programa calcular a idade atual na linha 10 podemos ver o estado da memória em 1.17C.

Neste exemplo, eu não mostrei o conteúdo "real" da memória, mas simbólico. Ou seja, eu teria que pegar o valor 1990, por exemplo, convertê-la em binário com 32 bits e complemento de 2 (a representação que o computador usa) e distribuir os 32 bits em 4 caixinhas, fazendo a correta separação do número. Vamos trabalhar com esta representação simbólica, pois o valor binária pode mudar de arquitetura para arquitetura e isto fugirá do nosso tema.

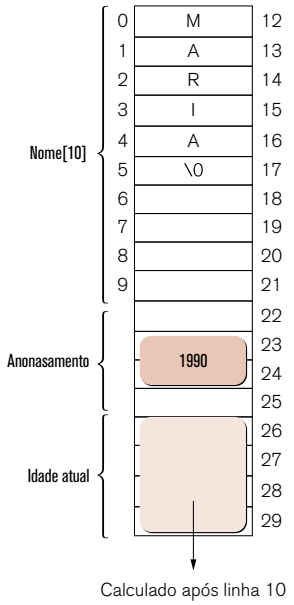


Figura 1.17 C – Estado da memória no programa de exemplo de coleta de dados do usuário.

REFLEXÃO

Neste capítulo nós vimos conceitos fundamentais sobre programação de computadores. Aprendemos a importância de algoritmos, linguagens de programação e vimos também qual será nossa linguagem alvo e nossos ambientes de teste. Conhecemos um pouco da linguagem C, seus operadores e um pouco acerca do processo de compilação. Na próxima unidade avançaremos no estudo da linguagem C e veremos opções de comandos para escrever programas mais complexos.

LEITURA

Para complementar seu aprendizado computação, programação e linguagem C sugiro os seguintes links em inglês:

- http://en.wikiversity.org/wiki/Introduction_to_Computers Acesso em março de 2015.
- http://www.cprogramming.com/ Acesso em março de 2015.

Também sugiro a leitura de livros em português como:

- KERNIGHAN, B. W.; RITCHIE, D. M. C: a linguagem de programação, padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.

E também sugiro os seguintes links em português

- http://pt.wikipedia.org/wiki/C_%28linguagem_de_programa%C3%A7%C3%A3o%29 Acesso em março de 2015.
- https://www.inf.pucrs.br/~pinho/LaproI/IntroC/IntroC.htm Acesso em março de 2015.

REFERÊNCIAS BIBLIOGRÁFICAS

KERNIGHAN, B. W.; RITCHIE, D. M. C: **a linguagem de programação,** padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.

KELLEY, A.; POHL, I. A Book on C: **Programming in** C. 4. ed. Boston: Addison Wesley, 1997. 726p.

ZIVIANI, N. **Projetos de algoritmos:** com implementações em Pascal e C. São Paulo: Pioneira Thomson, 2002. 267p.

ARAÚJO, J. **Dominando a Linguagem C**. 1. ed. Ciência Moderna, 2004, 146p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos:** teoria e prática. Rio de Janeiro: Campus, 2002. 916p.

FORBELLONE, A. L. **Lógica de Programação**. 2. ed. São Paulo: Makron Books, 2000. 195p.

KNUTH, D. E. **The Art of Computer Programming, Fundamental Algorithms.** 3. ed. Boston: Addison Wesley, 1997. 672p. (vol 1)

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. 4. ed. Porto Alegre: Bookman, 2000. 624p.

2

Controle de Fluxo, Tomada de Decisão e Funções

Neste capítulo veremos uma série de comandos que nos permitem programar a resolução dos mais variados tipos de problemas. Tomar decisões, por exemplo, é algo que fazemos rotineiramente. Com programas de computador não é diferente. Veremos estruturas básicas que, juntas, nos permite criar grandes e complexas soluções.

OBJETIVOS

Neste capítulo, os objetivos principais são:

- Controle de Fluxo
- Tomada de decisão
- Laços
- Seleção
- Funções
- Pilha de execução
- Ponteiros de variáveis
- Passagem por valor e por referência

2.1 Controle de Fluxo

Existem vários tipos de comandos na linguagem C que controlam o fluxo de suas instruções. O controle de fluxo serve para que o computador consiga fazer as instruções parcialmente, através de avaliações de condições. Na resolução de problemas é muito comum em que uma parte de código seja executada apenas se uma condição for verdadeira ou falsa. Também é comum que a aplicação passe repetidas vezes em um trecho de código, evitando a cópia do mesmo código em vários trechos da aplicação (o que seria trabalhoso demais). Alguns grupos de comandos também são:

- **Comandos de Seleção:** Inclui comandos como "if" e "switch" também conhecidos pelo termo "comando condicional".
- **Comandos de Iteração:** Conhecido também como comandos de laço, os comandos de iteração são while, for e do-while.

2.2 Tomada de Decisão

Uma decisão é uma resolução que se toma relativamente a algo, geralmente um operador condicional. É o processo de se realizar uma escolha entre diversas alternativas. A tomada de decisões aparece em qualquer contexto do nosso dia a dia, seja a nível profissional, sentimental, familiar e computacional. Seu processo nos permite resolver desafios de uma organização, pessoa ou um sistema computacional.

Comandos de tomada de decisão são importantes em programação, pois nos permite decidir qual ação tomar, quando e o que fazer dentro de uma aplicação. Em linguagens de programação é uma a peça chave.

Quando escrevemos programas, na maioria das vezes, temos a necessidade de decidir o que fazer dependendo de alguma condição encontrada durante a execução da tarefa. Por exemplo:

- Você quer saber se certa variável tem valor numérico maior que outra variável;
- Você quer saber se um conteúdo é formado por letras;
- Você quer saber se um número é ou não divisível por outro;

• Você quer decidir em qual rua virar dependendo da distância a ser percorrida.

Com exceção do último exemplo, são todos casos de blocos básicos de construção de tomada de decisão. Para resolver o problema no último exemplo você, com certeza, usará blocos básicos em sua programação.

São estes **blocos básicos** que aprenderemos aqui!

No uso da tomada de decisão você deverá escrever o código de tal forma que forneça a "habilidade necessária" para seu programa (ou algoritmo) tomar a decisão de qual rumo seguir de acordo com as condições.

Na da linguagem C, os comandos mais comuns utilizados na programação de tomada de decisão são do tipo *if* (se), *while* (enquanto), *switch-case* (escolha, caso), *for* (para), *do-while* (faça enquanto).

2.2.1 O Comando If

Na linguagem C o comando if é uma estrutura de decisão que decide se uma sequência de comandos será ou não executada. É utilizado sempre que é necessário escolher entre dois ou mais caminhos dentro de um programa. Como mostra o código 2.1, a estrutura básica de um *if* é a avaliação de uma condição que, se verdadeira, implicará na execução do conjunto de comandos dentro do bloco.

Código 2.1

```
if (condição)
{
    Sequencia de comandos;
}
```

O significado do comando é muito simples: se a condição for diferente de zero, ela será considerada verdadeira e a sequência de comandos será executada. Se a condição for zero, ela será considerada falsa e a sequência de comandos não será executada, continuando a sequência de processo do programa. Como vimos, C não possui um tipo lógico *boolean* para representar verdadeiro ou falso. Logo, a avaliação das condições lógicas resulta em 0 ou 1.

No código 2.2, temos um exemplo de um programa que consegue ler um número inteiro digitado pelo usuário e informar se o número é maior do que 5:

Código 2.2

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

    printf("Digite um numero: ");
    scanf("%d", &num);

    if(num > 5)
           printf("O Numero eh maior do que 5\n");

    system("pause");
    return 0;
}
```

Agora imagine um programa que pega sua idade e lhe informa se você é maior de idade, e logo depois também te deixa informado que você pode adquirir uma carteira de habilitação. Veja o código 2.3.

Código 2.3

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

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

    if(idade >= 18)
           printf("Voce eh maior de idade e pode ter
carteira de habilitacao. \n");

    system("pause");
    return 0;
}
```

Nos exemplos acima, percebe-se o uso do comando if, usando condições simples para exemplificar o uso de tomada de decisão. Neste outro programa, no código 2.4, o usuário digita números que o programa avaliará. A avaliação feita informa se o primeiro número é maior do que o segundo número.

Código 2.4

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num1;
    int num2;

    printf("Digite um numero: ");
    scanf("%d", &num1);

    printf("Digite outro numero: ");
    scanf("%d", &num2);

    if(num1 > num2)
    {
           printf("O numero: %d eh maior que o numero:
%d\n " num1, num2);
    }

    system("pause");
    return 0;
}
```

É importante considerarmos que o código anterior ainda não está completo. Por quê? Reflita um pouco. O que ocorre se os números forem iguais? E se o número num2 for maior que num1? Agora veremos como deixar nossas estruturas condicionais mais completas.

2.2.2 O Comando If-Else

O comando if else é uma estrutura de decisão que decide qual dentre duas sequências de comandos será executada. O comando else pode ser entendido como sendo um complemento do comando if, auxiliando o comando na tarefa de escolher dentre os vários caminhos a ser seguido dentro do programa. Podemos pensar no comando else como a negação do comando if, ou seu inverso. Mas, devemos tomar bastante cuidado com comandos if-else compostos, que veremos mais adiante. A sintaxe do comando if-else é:

Código 2.5

```
if(condição)
{
    Sequencia de comandos;
}
else
{
    Sequencia de comandos;
}
```

A semântica do comando if-else é semelhante a do if: se o valor da condição for diferente de zero ou seja, se ela for verdadeira, o sistema executará a sequência de comandos 1 (linha 3 do exemplo); caso a condição seja falsa o sistema irá executar a sequência que existe nos comandos 2 (linha 7).

Como exemplo, um programa pode verificar a paridade de um numero dado pelo usuário. Poderíamos verificar se o resto da divisão do número por dois é igual a Zero. Se a condição for verdadeira, o número informado pelo usuário é par, caso contrário ele seria ímpar.

Código 2.6

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

    printf("Digite um numero: ");
    scanf("%d", &num);

    if(num % 2 == 0)
           printf("O numero: %d, eh par.\n", num);
    else
           printf("O numero: %d, eh impar.\n", num);

    system("pause");
    return 0;
}
```

Veja que a coleta dos dados é feita na linha 8. O usuário digitará o número a ser checado. Depois, na linha 10 é feito cálculo do resto da divisão do número por 2. Sabemos que todo número par tem resto zero quando dividido por 2 e todo número ímpar tem resto 1. É esta avaliação que também é feita na linha 10.

Isto é importante! Na mesma linha 10 são realizadas 2 instruções: o cálculo do resto da divisão do número por 2; a avaliação lógica deste resto, checando se o mesmo é igual à 0.

Como tanto no if quanto no else há apenas um comando, não usamos o delimitador de bloco ({ e }).

Agora, vamos a outro exemplo: pense que você está jogando um jogo eletrônico e um inimigo aproxima-se de você e te acerta um golpe; como seria o tratamento dessa remoção de sua vida pelo valor do golpe do inimigo? Podemos ver este exemplo em um simples código abaixo, utilizando tomada de decisão.

Código 2.7

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int vida;
    int dano;

    printf("Digite a vida do Jogador:");
    scanf("%d", &vida);

    printf("\n Digite o dano que o jogador ira
levar: ");
    scanf("%d", &dano);

    vida = vida - dano;

    if(vida > 0)
           printf("O jogador levou: %d, e ficou com %d
de vida.\n", dano, vida);
    else
           printf("O jogador morreu com o dano: %d",
dano);

    system("pause");
    return 0;
}
```

No código acima podemos notar que existe uma condição para informar a quantidade de vida que ainda resta no jogador (linhas 8 e 9).

A partir da linha 17 avaliamos se a vida do jogador for maior que zero. Se for verdadeira esta afirmação o programa irá informar ao usuário a quantidade do dano que foi dado ao jogador e também a vida que lhe resta, senão ele irá informar que o jogador morreu e a quantidade de dano que fez aquele jogador morrer.

Agora o usuário pode informar um número para o programa, e assim, ele informa para o usuário se o número informado é maior, menor ou igual a zero. Dentro do código será mostrado o uso de várias condições em um mesmo programa. No código podemos ver três condições para verificar se irá ou não executar a sequência de comandos em cada condição.

Código 2.8

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

    printf("Digite um numero: ");
    printf("Este numero pode ser menor, igual ou
maior que zero: ");
    scanf("%d", &num);

    if(num == 0)
           printf("O numero é 0.\n");
    else if(num > 0)
           printf("O numero %d é maior que 0.\n",
num);
    else
           printf("O numero %d é menor que 0.\n",
num);

    system("pause");
    return 0;
}
```

Veja nas linhas 12 a 17 que usamos a construção condicional composta. Um comando if negado equivale a avaliar um comando else parcial, ou seja, não trata apenas do oposto do if, mas parte disto.

Isto ocorre no nosso exemplo, pois a avaliação do número tem três respostas possíveis. Logo, o simples uso do comando if-else não resolveria nosso problema. É preciso "quebrar" uma estrutura lógica binária avaliando primeiro se o número é zero, caso não seja então OU ele é maior do que zero OU ele é menor do que zero.

2.2.3 O Operador Condicional Ternário.

O operador condicional ternário é um operador especial que substitui uma situação particular de comando de decisão. Sua sintaxe é a seguinte:

```
Variável = Expressão logica ? Expressão 1 : Expressão 2;
```

A semântica deste comando funciona da seguinte forma: a Expressão logica é avaliada e se a mesma for diferente de zero o valor da Expressão 1 que está a frente da expressão logica é atribuído a Variável; caso o valor lógico seja igual a zero o valor da Expressão 2 é atribuído a Variável.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num1;
    intnum2;

    printf("Digite um numero: ");
    scanf("&d", &num1);

    printf("Digite outro numero: ");
    scanf("&d", &num2);

    variavel = (num1 > num2) ? num1 : num2 ;

    printf("Numero escolhido é: &d", variavel);

    system("pause");
    return 0;
{
```

2.2.4 Laço ou Loop

Um laço em programação de computadores é uma sequência de ações que se repete por um número específico de vezes ou até que uma condição seja satisfeita. Enquanto a condição for verdadeira, as instruções serão executadas. O laço de repetição também pode ser chamado de loop. Na linguagen C pode-se usar 3 tipos de loops: while, do... while e for.

2.2.5 O Loop Tipo While

O loop while funciona da seguinte forma: ele testa uma condição; caso a condição seja verdadeira, o código dentro da estrutura de repetição é executado e o teste é repetido ao final da execução. Quando a condição for falsa, a execução do código continua logo após a estrutura de repetição, sua sintaxe é definida assim:

Código 2.10

```
while(condição)
{
    Sequencia de comandos;
}
//Instrução logo após estrutura de repetição
```

No exemplo abaixo, imagine um programa que pega duas informações do usuário. O programa vai ler a informação e informar ao usuário um relógio que decrementa a diferença de um valor ao outro.

Código 2.11

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num1;
    int num2;

    printf("Digite um numero: ");
```

```
    scanf("%d", &num1);

    printf("Digite um numero maior que %d : ", num1);
    scanf("%d", &num2);

    while(num1 > num2)
    {
           printf("%d é maior que %d: \n" num1, num2);
           num1--;
    }

    printf("%d é igual que %d: \n" num1, num2);

    system("pause");
    return 0;
}
```

Usando while, você também pode fazer loops infinitos, usando uma condição que é sempre verdadeira, como "1 == 1" ou simplesmente "1".

Código 2.12

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

int main()
{
    int num1;

    while(1)
    {
        printf("\nDigite um numero: ");
        scanf("%d", &num1);
           printf("Seu numero é: %d", num1);
    }

    system("pause");
    return 0;
}
```

2.2.6 O Loop do While

O loop "do ... while" é exatamente igual ao "while" exceto por uma observação: a condição é testada depois do bloco, o que significa que o bloco é executado pelo menos uma vez. A estrutura do ... while executa o bloco, testa a condição e, se esta for verdadeira, volta para o bloco de código. Sua sintaxe é:

Código 2.13

```
do
{
    Sequencia de comandos;
} while(condição);
```

É possível notar que, diferente das outras estruturas de controle, no do... while é necessário colocar um ponto e vírgula após a condição.

Hoje, o que mais temos em programas são menus para escolha do usuário. Iimagine um programa onde o usuário pode informar uma escolha dentre as opções disponíveis. O exemplo abaixo é um simples menu de opções.

Código 2.14

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

int main ()
 {
    int i;

        do {
           printf ("Escolha uma das opções pelo número:\n\n");
           printf ("\t(1) Esquerdo\n");
           printf ("\t(2) Pequeno\n");
           printf ("\t(3) Baixo\n\n");
           scanf("%d", &i);
        } while (i < 1 || i > 3);
```

```
        if(i == 1)
              printf ("Você escolheu Esquerdo, o
oposto é Direito.\n");
        else if(i == 2)
              printf ("Você escolheu Pequeno, o oposto
é Grande.\n");
        else
              printf ("Você escolheu Baixo, o oposto é
alto.\n");

        i = 0;

    system("pause");

    return 0;
 }
```

2.2.7 O Comando For

O loop for permite que alguma inicialização seja feita antes do loop e que um incremento (ou alguma outra ação) seja feita após cada execução sem incluir o código dentro do bloco. Pode se dizer também que é uma estrutura de repetição que repete a execução de uma dada sequência de comandos uma certa quantidade de vezes, que pode ser determinada pelo próprio programa. Sua sintaxe deve ser escrita assim:

Código 2.15

```
for( inicialização; condição; incrementação )
{
    Sequencia de comandos/ instruções;
}
```

Podemos fazer essa sintaxe usando o while para exemplificar suas equivalências.

Código 2.16

```
Inicialização;
while(condição)
{
    Sequência de comandos/ instruções;
    Incremento;
}
```

Imagine que um programa precise dar voltas em um brinquedo. Para isso é preciso configurar a quantidade de voltas que o brinquedo rode até parar. Vamos esquecer a velocidade do brinquedo e fazer um pequeno sistema onde o programa pega a quantidade de voltas que o usuário informa e mostre cada vez que é passado uma volta.

Código 2.17

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

int main()
{
    int voltas;
    int i;

    printf("Digita um numero maior que 2: ");
    scanf("%d", &voltas);

    for (i = 0; i < voltas; i++)
    {
           if (i == 0) printf("Eu vou dar %d voltas.",
voltas);
           printf(" \n Estou na volta %d.", i + 1);
    }

    system("pause");
    return 0;
}
```

Imagine agora que você precise de algum jeito fazer uma soma dentro de cada volta dada pelo brinquedo, seja somar força, velocidade ou qualquer outra coisa que você pretenda utilizar. No código abaixo o programa sempre soma o número escolhido a um total de voltas, e no final mostra o total somado em todas as voltas.

Código 2.18

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

int main()
{
    int voltas;
    int num;
    int i;
    int soma = 0;

    printf("Digita quantidades de voltas: ");
    scanf("%d", &voltas);
    printf("Digite um numero para ser somado a cada
volta: ");
    scanf("%d", &num);

    for (i = 0; i < voltas; i++)
    {
           soma = soma + num;
    }

    printf("A soma  do numero %d pela quantidade de
voltas é: %d", num, soma);

    system("pause");
    return 0;
}
```

2.2.8 Comandos de Seleção Múltipla Switch/Case

Além dos comandos de seleção if e else a linguagem C possui um comando que é de seleção múltipla, chamado de switch. Desenvolvemos programas para executar várias tarefas de forma independente, por exemplo, um programa que gerencia um caixa eletrônico tem como proposta ao se navegar em suas interfaces, opções para que possamos executar uma tarefa especifica como a realização de um saque ou um deposito. É comum este tipo de programa trazer para o usuário várias opções de escolhas, o comando switch tem este objetivo e deve ser escrito no seguinte formato. Perceba que há um comando break ao final do conjunto de instruções (veja a linha 05, por ex.). Este comando é o responsável por assegurar que o case não continue a executar as demais instruções. Experimente não colocar o break e entrar num dos cases na escolha de opção.

Código 2.19

```
switch (expressão)
{
    case valor1:
           Instruções;
           break;
    case valor2:
           Instruções;
           break;
    ...
    default:
           Instruções;
}
```

Outro ponto interessante a citar é o uso do default. O *default* é a opção a ser executada caso o switch não entre em nenhum dos cases. O *default* não é obrigatório.

Podemos notar que o switch não precisa usar chaves em volta dos blocos, a menos que declaremos variáveis neles. Um exemplo simples para o uso de switch é a criação de menu.

Código 2.20

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

int main()
{
    int opcao;
     printf ("[1] Cadastrar cliente\n"
             "[2] Procurar cliente\n"
             "[3] Inserir pedido\n"
             "[0] Sair\n\n"
             "Digite sua escolha: ");
     scanf ("%d", &opcao);

     switch (opcao) {
        case 1:
           printf("Cadastrar Cliente...");
           break;
        case 2:
           printf(" Procurar Cliente...");
           break;
        case 3:
           printf(" Inserir Pedido...");
           break;
        case 0:
           return 0;
        default:
           printf ("Opção inválida!\n");
     }
}
```

Quando você executa o comando switch, o valor da expressão é comparado, na ordem, com cada um dos valores que foram definidos pelo comando case. Se algum desses valores for igual ao valor da variável a sequência de comandos daquele comando case é executado pelo programa, como podemos ver no código acima.

O exemplo anterior do comando switch poderia facilmente ser escrito com a utilização dos comando if..else..if, como podemos ver no código abaixo.

Código 2.21

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

int main()
{
    int opcao;
     printf ("[1] Cadastrar cliente\n"
             "[2] Procurar cliente\n"
             "[3] Inserir pedido\n"
             "[0] Sair\n\n"
             "Digite sua escolha: ");
     scanf ("%d", &opcao);

    if(opcao == 1 )
        printf("Cadastrar Cliente...");
    else if(opcao == 2)
        printf(" Procurar Cliente...");
    else if(opcao == 3)
        printf(" Inserir Pedido...");
    else if(opcao == 0)
        return 0;
    else
        printf ("Opção inválida!\n");

}
```

2.2.9 Funções

A escrita de códigos em linguagem de programação vai se tornando cada vez mais complexa à medida que o tamanho do problema a ser resolvido cresce.

Ao crescer o que ocorre naturalmente é o aumento de número de linhas do programa e a existência de trechos repetíveis, ou seja, trechos do código que fazem a mesma coisa. É como escrever um texto: às vezes dizemos a mesma coisa, de forma diferente, várias vezes. Para isto, programar requer o uso de funções.

Um programa em C pode e deve ser escrito como um conjunto de funções que são executadas a partir da execução de uma função denominada main(). Uma função pode ter diferentes tipos de declarações de variáveis, instruções, ativações de funções próprias do sistema que está rodando a aplicação. O objetivo de se utilizar funções é realizar alguma "sub-tarefa" específica da tarefa que o programa pretende realizar. Podemos dizer que uma função é um trecho de programa que faz tarefas especificas e pode ser chamado de qualquer parte do programa quantas vezes desejarmos. As funções devem realizar uma única tarefa (favorece a reutilização) e devem ser pequenas (caber numa tela).

O fato de executar uma tarefa facilita a sua denominação. A dificuldade de se nomear uma função normalmente está associada ao fato dela não estar bem definida.

Utilizar funções pequenas facilita muito a etapa de testes e manutenção de programas.

O uso de funções permite a reutilização de códigos e auxilia também na manutenção, pois o código fica melhor estruturado.

A ideia principal da função é permitir que o programador possa agrupar várias operações em um só escopo que pode ser chamado através do seu nome. A função deve ser definida da seguinte forma:

Código 2.22

```
[Tipo de retorno da função] [nome da função] (1º parâmetro, 2º parâmetro,...)
{
    //código
}
```

Uma função pode necessitar de dados para realizar uma ação desejada. Esses dados são chamados parâmetros da função. A função também pode retornar um valor é chamado de retorno da função.

Há algumas regras em C a serem seguidas para o uso da função, para o nome da função e dos parâmetros. Para nome dos parâmetros valem as mesmas regras que foram dadas aos nomes de variáveis. A definição da função deve ser codificada antes da função main e o código tem a obrigação de estar dentro de chaves para que funcione.

A compilação da função deve acontecer antes dela ser chamada pela primeira vez. Isso garante que, quando a sua chamada for compilada, o compilador já a conheça, aceitando o código como correto.

O código da função deve estar entre chaves que delimitam seu escopo.

2.2.10 Valor do Retorno

Em C utiliza-se a instrução return para poder retornar algum valor. Para isto é preciso especificar o tipo de retorno, que pode ser char, int, float dentre outros. Um exemplo simples de uma função que retorna um char.

```
char clara()
{
    return `C`;
}
```

Dentro de uma função também podemos retornar um tipo void, que na verdade significa que não há retorno.

2.2.11 Parâmetros

Um parâmetro é um valor que é fornecido à função quando ela é chamada. É comum também chamar os parâmetros de argumentos, embora argumento esteja associado ao valor de um parâmetro.

Os parâmetros de uma função podem ser acessados da mesma maneira que variáveis locais. Eles na verdade funcionam como variáveis locais. Modificar um argumento não modifica o valor original no contexto da chamada de função, pois um argumento numa chamada de função é copiado como uma variável

local desta função.

Portanto, podemos entender que os parâmetros são declarações de variáveis locais da função (somente existem durante a sua execução) e que recebem os valores de entrada que serão processados pela função. O mecnismo de comunicação permite que uma função retorne apenas um valor, fruto do processamento dos dados de entrada.

Da mesma forma que uma variável, uma função pode ser declarada. Isso faz com que não seja obrigatório posicionar o código da função antes das suas chamadas. Caso a função tenha sido declarada, seu código pode aparecer em qualquer posição.

Para declarar a existência de uma função no programa e a presença de parâmetros, usamos uma lista de parâmetros entre parênteses, separados por vírgulas. Cada declaração de parâmetro é feita de maneira semelhante à declaração de variáveis: a forma geral é <tipo nome>. Isto é chamado de prototipação de funções.

A prototipação é uma forma de "avisar" ao compilador sobre a existência da função, seus parâmetros e tipo de retorno.

Alguns exemplos da declaração:

```
int soma (int a, int b);
float subtracao(float a, float b);
```

Em C a declaração de uma função sem parâmetro deve ser desta maneira:

```
void função(void);
```

Agora vamos a um exemplo: vamos criar uma função que recebe dois números inteiros como parâmetros (linha 4) e retorna a soma deles (linha 6). É uma função bem simples e veja que não usamos protótipo. Nós fizemos a declaração e escrita dela já antes da função main.

Nas linhas 9 a 25 nós utilizamos a função através de um exemplo de inserção dos dados pelo usuário e retorno do valor calculado da soma.

Código 2.22

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

int soma(int a, int b)
{
    return a+b;
}

int main()
{
    int a;
    int b;
    int total;

    printf("Digite o numero [a]: ");
    scanf("%d", &a);
    printf("\nDigite o numero [b]: ");
    scanf("%d", &b);

    total = soma(a, b);

    printf("A soma total dos numero é %d.", total);

    return 0;
}
```

Podemos usar funções sem parâmetros, como vimos. Podemos criar funções de mensagens para interagir com o usuário. Pense que quando o usuário entra no programa, recebe boas vindas e, quando ele acaba de usar o programa, recebe uma mensagem de despedida. Fica interativo, não?

Código 2.23

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

int soma(int a, int b)
{
    return a+b;
}
```

```
void inicio()
{
    printf("Bem Vindo ao Programa\n");
}

void final()
{
    printf("\nFechando programa. Obrigado por usar");
}

int main()
{
    int a;
    int b;
    int total;

    inicio();

    printf("Digite o numero [a]: ");
    scanf("%d", &a);
    printf("\nDigite o numero [b]: ");
    scanf("%d", &b);

    total = soma(a, b);

    printf("A soma total dos numero é %d.", total);

    final();

    return 0;
}
```

2.2.12 Ponteiros

O conceito de ponteiros é essencial na linguagem de programação C, mas para entendermos ponteiros é preciso falarmos sobre endereços de memória.

A memória RAM de qualquer computador nada mais é que uma sequência de bytes. Cada byte armazena um de 256 possíveis valores. Cada objeto na memória do computador ocupa certo número de bytes consecutivos. Um char ocupa 1 byte, um int ocupa 4 bytes. Há pequenas variações dependendo da plataforma (hardware + sistema operacional utilizado).

Cada objeto na memória tem um endereço e na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte. Podemos demostrar no exemplo abaixo.

```
int num; //ocupa 4 bytes
float soma; //ocupa 4 bytes
char nome; //ocupa 1 byte
```

Os endereços das variáveis poderiam ser esses:

```
num     98342
soma    98346
nome    98350
```

Todos os endereços são apenas demonstrações, o operador & é o operador que acessa o endereço de uma variável. Nos exemplos acima &num = 98342.

Um ponteiro é uma de variável que armazena endereços. É uma variável especial. Um ponteiro pode ter o valor especial NULL que não é endereço de lugar algum, indica apenas a ausência de um valor para aquele ponteiro. Se um ponteiro p armazena o endereço de uma variável x podemos dizer que p é o endereço de x ou p aponta para x. Se um ponteiro p tem valor diferente de NULL então *p é o valor do objeto apontado por p. Um exemplo simples: se x é uma variável e p é igual a &x então dizer *p (lê-se conteúdo de p) é o mesmo que dizer x.

Há vários tipos de ponteiros: ponteiros para caracteres, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para registros e etc. O computador faz questão de saber qual tipo de ponteiro você está falando. Para declarar um ponteiro p para um inteiro, podemos fazer da seguinte maneira.

```
int *p;
```

Podemos também apontar um ponteiro para outro ponteiro de um inteiro, usamos desta forma.

```
int **p;
```

No código abaixo temos um exemplo de como fazer uma soma de valores entre variáveis utilizando ponteiros.

Código 2.24

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

    int *p;
    int *q;

    printf("Digite o numero [a]: ");
    scanf("%d", &a);
    printf("\nDigite o numero [b]: ");
    scanf("%d", &b);

    p = &a;
    q = &b;
    c = *p + *q;

    printf("A soma total dos numero é %d.", c);

    return 0;
}
```

Logo a seguir, utilizaremos o mesmo código, só que mostrando como ficaria você utilizar um ponteiro do ponteiro para fazer a troca.

Código 2.25

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

    int *p;
    int *q;
    int **r;

    printf("Digite o numero [a]: ");
    scanf("%d", &a);
    printf("\nDigite o numero [b]: ");
    scanf("%d", &b);

    p = &a;
    q = &b;
    r = &q;
    c = **r + *q;

    printf("A soma total dos numero é %d.", c);

    return 0;
}
```

Podemos ver nos exemplos que o ponteiro é uma variável capaz de armazenar um endereço de outra variável e também pode "apontar" para outro ponteiro que "aponta" para uma variável qualquer. Na prática, apontar implica em armazenar valores de memória.

2.2.13 Operações com Ponteiros

Dados dois ponteiros p e q podemos igualá-los fazendo p=q. Repare que estamos fazendo com que p aponte para o mesmo lugar que q, ou seja, é um comando de atribuição normal, só que tanto o conteúdo de p, quanto q, trata-se

de endereço de memória. Logo, p receberá o conteúdo de q que é um endereço de memória e passará a "apontar" para o local que q aponta.

Se quisermos que a variável apontada por p tenha o mesmo conteúdo da variável apontada por q devemos fazer:

```
*p = *q
```

Veja o exemplo abaixo e o resultado de sua execução. No código 2.26 declaramos uma variável inteira chamada a (linha 5) e um ponteiro para inteiro chamado p (linha 6). Em seguida, atribuímos o valor 10 para a variável a (linha 8) e atribuímos à variável p o endereço da variável a (linha 9). Veja que usamos o operador especial de endereço '&' (cuidado para não confundir com o operador lógica & e &&, ok?).

Agora, observe bem a figura 2.1 com o resultado da execução do programa.

Código 2.26

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a;
    int *p;

    a = 10;
    p = &a;

    printf("conteudo de [a] = %d\n", a);
    printf("endereco de [a] = %d\n", &a);
    printf("conteudo de [p] = %d\n", p);
    printf("endereco de [p] = %d\n", &p);
    printf("conteudo apontado por [p] = %d\n", *p);

    return 0;
}
```

Na figura 2.1 pode ser visto o valor da variável a na linha 1. Na linha 2 vemos o endereço da variável a. Agora, observe bem o valor (o conteúdo) da variável p (que é um ponteiro). Veja que seu conteúdo, na linha 3 é exatamente o mesmo do endereço da variável a.

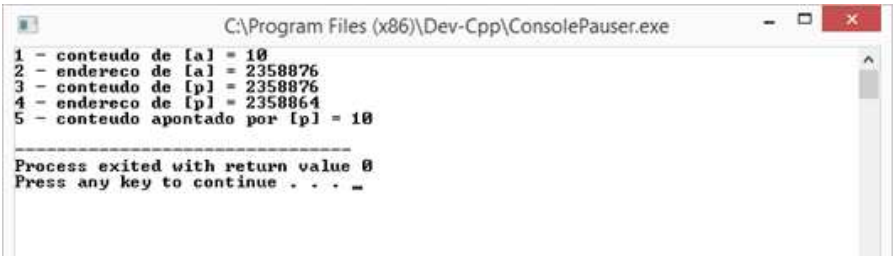


Figura 2.1 – Execução do Código 2.26.

Já o endereço de p assume outro valor. Agora, observe também o conteúdo obtido quando usamos o operador *. Ele retorna o valor apontado por p!

Quando incrementamos um ponteiro aumentamos em xBytes o seu conteúdo. Ou seja, ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Se temos um ponteiro para um inteiro e o incrementamos ele recebe um valor 4 bytes maior. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro char* ele anda 1 byte na memória e se você incrementa um ponteiro double* ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que p é um ponteiro, as operações são escritas como:

```
p++
p--
```

Veja o código 2.27. Agora inserimos uma variável b. O ponteiro p aponta para a variável b agora (linha 11). Veja na figura 2.2 que o conteúdo apontado por p é 20 e não 10 (linha 7 da figura).

Código 2.27

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a;
    int b;
    int *p;

    a = 10;
    b = 20;
    p = &b;

    printf("1 - conteudo de [a] = %d\n", a);
    printf("2 - endereco de [a] = %d\n", &a);
    printf("3 - conteudo de [b] = %d\n", b);
    printf("4 - endereco de [b] = %d\n", &b);
    printf("5 - conteudo de [p] = %d\n", p);
    printf("6 - endereco de [p] = %d\n", &p);
    printf("7 - conteudo apontado por [p] = %d\n",
*p);
    printf("**************************************
****\n");
    p++; //incrementamos o ponteiro p
    printf("8 - endereco de [p++] = %d\n", &p);
    printf("9 - conteudo de [p++] = %d\n", p);
    printf("10 - conteudo apontado por [p++] =
%d\n", *p);

    return 0;
}
```

A linha 23 do código 2.27 é chave no nosso exemplo. Nós incrementamos o ponteiro p. Veja na figura 2.2, linha 5, que o conteúdo de p terminava em 72. Agora, veja na linha 9 que ele termina em 76, ou seja, recebeu um incremento de 4 bytes. O que ocorreu com ele é que, por acaso, ele agora aponta para a posição de memória da variável b (veja linha 4 da figura).

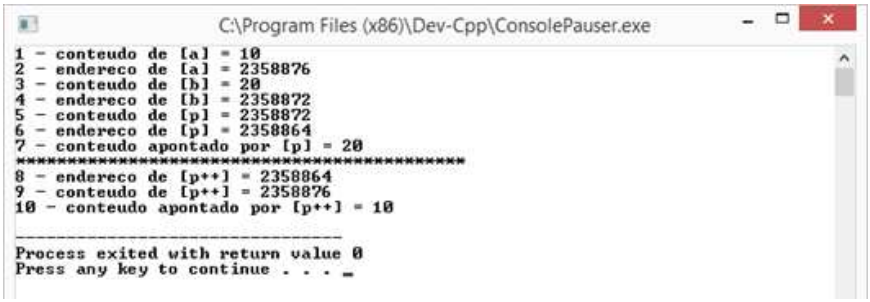


Figura 2.2 – Execução do Código 2.27.

Por que usamos o "por acaso"? Você deve tomar muito cuidado com o uso de ponteiros, pois perceba na figura que os valores de memória foram decrescendo ao invés de crescer. Isto se deve pelo funcionamento da alocação de memória pelo sistema operacional. Não se pode apenas usar uma operação p++ e esperar que tudo ocorra bem. Deve-se tomar muito cuidado e saber exatamente qual alteração você quer fazer no seu ponteiro.

Mas, e se quiséssemos incrementar o conteúdo apontado por um ponteiro?

Estamos falando de operações com ponteiros e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro p, faz-se:

```
(*p)++;
```

Ou seja, usamos o operador que acessa o conteúdo de p (*) e depois fazemos a operação de incremento. O uso de parênteses é obrigatório, pois ++ tem precedência sobre o *.

2.2.14 Ponteiros como Parâmetros de Funções e Pilha de Execução de Programa

Quando se usa uma função para fazer a troca de duas variáveis, é necessário passar variáveis de ponteiros nos parâmetros. No exemplo abaixo tem-se uma função de troca de variáveis que, por falta dos ponteiros é impossível acessar seu endereço de memória e alterá-las dentro da função.

```
void troca(int p, int q)
{
    int aux;
    aux = p;
    p = q;
    q = aux;
}
```

Isto ocorre, pois quando há a chamada de função há um empilhamento da execução do programa, ou seja, o código atual para sua execução até que o código da função termine e devolva um retorno (é por isto que funções possuem retorno, mesmo que vazio). Após o retorno o programa retoma a execução no ponto onde parou.

A pilha de execução de um programa também é usada para a comunicação entre funções. Funções são independentes entre si e possuem seu próprio escopo, o que define seu conjunto de variáveis. Assim, as variáveis nos parâmetros e as variáveis declaradas no bloco da função são locais a esta função e não serão visíveis a outras funções e nem ao programa principal.

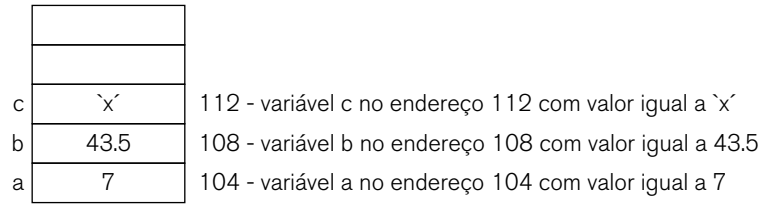


Figura 2.3 – Pilha de Execução.

Vejamos o código 2.28. Nós criamos uma função que calcula o fatorial de um número, a função fat. Ele recebe um número inteiro n e devolve outro número inteiro. O fatorial de um número é na forma fat(n) = n x n-1 x n-2 x ... x 1.

A chamada para a função fat ocorre na linha 8. Veja nas linhas 13 a 20 que ela se trata de um loop multiplicador de n por n-1 até que n não seja 0 (consulte o conceito da função fatorial se precisar, ok?).

Código 2.28

```
#include <stdio.h>
#include <stdlib.h>
int fat(int);

int main (void) {
  int n = 5;
  int r;
  r = fat ( n );
  printf("Fatorial de %d = %d \n", n, r);
  return 0;
}

int fat (int n) {
  int f = 1;
  while (n != 0) {
    f *= n;
    n--;
  }
  return f;
}
```

Agora, veja que o valor n da função fat é um parâmetro. A alteração do valor de n em fat não implica alteração do valor de n em main. É possível ter variáveis com mesmo nome em escopos diferentes.

Na figura 2.4 é possível acompanharmos o que ocorre na pilha de execução com as variáveis de main e de fat, durante a execução do programa. Primeiro a pilha de execução aloca espaço para as variáveis locais de main (r e n). A variável n é inicializada com 5. A variável r não é inicializada. Quando a função fat é chamada na linha 8 do código 2.28 há o empilhamento da execução de fat e agora as variáveis de fat também aparecem na pilha. Veja na linha 8 que a variável r de main receberá o retorno da função fat. Na pilha vemos as variáveis n de fat (que recebe o valor passado como cópia de n do main) e a variável f de fat, declarada localmente e inicializada com 1. Após execução de fat f contém o valor do fatorial de 5, que é 120. No final da execução fica na pilha o valor 120 (return f) atribuído à r (linha 8 do programa).

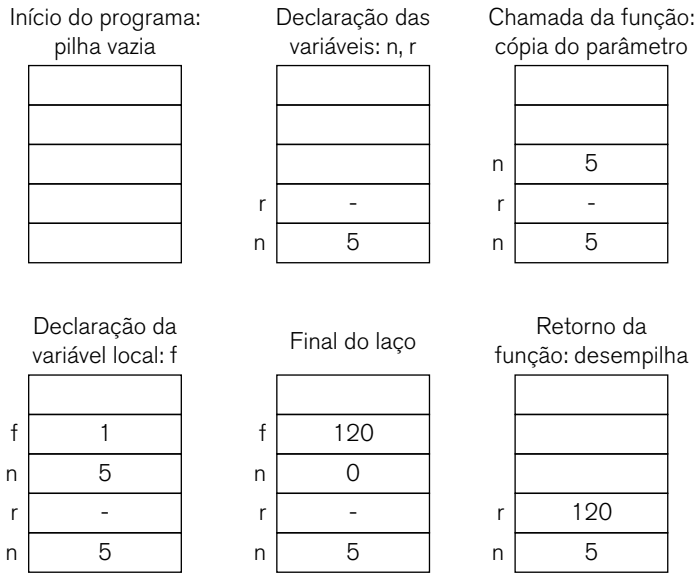


Figura 2.4 – Pilha de Execução durante a execução da função fat.

Para fazermos a função de troca funcionar precisamos reescreve-la da seguinte forma:

```
void troca(int *p, int *q)
{
    int aux;
    aux = *p;
    *p = *q;
    *q = aux;
}
```

Podemos perceber que passamos o endereço de memória das variáveis, desse modo podemos altera-las direto em seu endereço.

No código 2.29 você pode ver o uso da função troca. Escreva este código e teste-o. Tente também usar a função de troca sem receber ponteiros, como vimos no começo desta seção 2.14.

Código 2.29

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

void troca(int *p, int *q)
{
    int aux;
    aux = *p;
    *p = *q;
    *q = aux;
}

int main()
{
    int a;
    int b;

    printf("Digite o numero [a]: ");
    scanf("%d", &a);
    printf("\nDigite o numero [b]: ");
    scanf("%d", &b);

    printf("\nvariavel [a] igual %d.", a);
    printf("\nvariavel [b] igual %d.", b);

    troca(&a, &b);

    printf("\nTrocando os valores usando endereço de
memoria...");
    printf("\nvariavel [a] igual %d.", a);
    printf("\nvariavel [b] igual %d.", b);

    return 0;
}
```

Agora veja no código 2.30 que é o mesmo exemplo de 2.29, mas mostramos os valores de endereço de memória das variáveis. Observe que cada variável mantém seus valores de endereço, mas seus conteúdos foram alterados.

Código 2.30

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

void troca(int *p, int *q)
{
    int aux;
    aux = *p;
    *p = *q;
    *q = aux;
}

int main()
{
    int a;
    int b;

    printf("Digite o numero [a]: ");
    scanf("%d", &a);
    printf("\nDigite o numero [b]: ");
    scanf("%d", &b);

    printf("\nvariavel [a] igual %d.", a);
    printf("\nEndereço de memoria [a] igual %d.", &a);
    printf("\nvariavel [b] igual %d.", b);
    printf("\nEndereço de memoria [b] igual %d.", &b);

    troca(&a, &b);

    printf("\nTrocando os valores usando endereço de
memoria...");
    printf("\nvariavel [a] igual %d.", a);
    printf("\nEndereço de memoria [a] igual %d.", &a);
    printf("\nvariavel [b] igual %d.", b);
    printf("\nEndereço de memoria [b] igual %d.", &b);
    printf("\n");

    return 0;
}
```

As técnicas de programação definem que, numa chamada de função, os parâmetros podem ser passados por valor ou por referência.

A passagem por valor apenas informa um valor, não permitindo, assim, que a função altere o valor fornecido na variável de origem.

Já uma passagem por referência, permite que a função altere o valor armazenado na variável da função que realizou a chamada. Portanto, quando se faz uso de uma passagem de parâmetro por referência, estamos permitindo que a função determine o valor de uma variável da função que realizou a chamada. Este recurso permite, na verdade, que a função determine quantos valores forem necessários, embora ela sóssa retornar um valor por definição.

A linguagem C só permite a passagem de parâmetros por valor. No entanto, se utilizarmos ponteiros, podemos simular a passagem por referência.

Se os valores forem de tipos comuns (int, float etc.), só recebemos o valor sem sabermos onde eles estão originalmente armazenados, não sendo possível alterar seu local de origem.

Já se o o valor passado for um endereço de memória, podemos recebe-lo em uma variável do tipo ponteiro e alterar seu valor na origem.

Quando passamos parâmetros para funções que são ponteiros usamos o nome de passagem de parâmetro por referência. Quando usamos variáveis que não são ponteiros chamamos de passagem de parâmetros por valor.

- Passagem de parâmetro por valor:

1. Copia o valor do argumento no parâmetro formal.
2. Alterações feitas nos parâmetros da função não têm efeitos nas variáveis usadas para chamá-la.

- Passagem de parâmetro por referência

1. O endereço de um argumento é copiado no parâmetro.
2. Alterações feitas no parâmetro afetam a variável usada para chamar a função.

2.2.15 Reflexões: Aplicações com Passagem de Valores Numéricos e Endereços de Memória

Nesse capítulo vimos os mais importantes controles de fluxo (comandos if... else, switch) para auxiliar na seleção e tomada de decisões dentro do programa. Também aprendemos sobre os laços que podem ser chamados de loops e são usados para executar um conjunto ou bloco de instruções. Vimos sobre funções e a importância delas para melhoria da organização e manutenção do código.

Agora vamos colocar em prática tudo que nós vimos: imagine um programa que permita ao usuário ter acesso a um menu com as opções de inserir números, realizar operações de soma, subtração, multiplicação, fazer a troca dos valores das variáveis através de ponteiros e multiplicar um número por ele mesmo.

Analise o código, escreva-o, compile-o e teste-o, ok?

Código 2.31

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

void saudacaoAplicacao(void);
void saidaAplicacao(void);
void menu(void);
void pularLinha(int);
void troca(int*, int*);

int calculadora(int, int*, int*);
int multiplicaPorEleMesmo( int );

void main(void)
{
    int num1 = 200;
    int num2 = 100;

    int run = 1;
    int tipoCena = 0;
```

```
    saudacaoAplicacao();
    system("pause");

    while(run != 0)
    {
            if(tipoCena == 0)
            {
                    menu();
                    scanf("%d", &tipoCena);
            }
            else if(tipoCena == 1)
            {
                    system("cls");
                    printf("Insira o primeiro Numero
inteiro: ");
                    scanf("%d", &num1);
                    printf("Insira o segundo Numero
inteiro: ");
                    scanf("%d", &num2);
                    printf("Inseriu os numeros com
sucesso...\n");
                    system("pause");
                    tipoCena = 0;
            }
            else if(tipoCena == 2)
            {
                    system("cls");
                    printf("A soma dos numeros %d e %d e
igual a: %d\n", num1, num2, calculadora(1, &num1, &num2));
                    system("pause");
                    tipoCena = 0;
            }
            else if(tipoCena == 3)
            {
                    system("cls");
                    printf("A subtração dos numeros %d e %d e
```

```
 igual a: %d\n", num1, num2, calculadora(2,
&num1, &num2));
                    system("pause");
                    tipoCena = 0;
            }
            else if(tipoCena == 4)
            {
                    system("cls");
                    printf("A Multiplicação dos numeros
%d e %d e igual a: %d\n", num1, num2, calculadora(3,
&num1, &num2));
                    system("pause");
                    tipoCena = 0;
            }
            else if(tipoCena == 5)
            {
                    system("cls");
                    printf("\nO Primeiro numero e igual
%d", num1);
                    printf("\n... e corresponde a este
endereço de memoria: %d", &num1);
                    pularLinha(1);
                    printf("\nO Segundo numero e igual
 %d", num2);
                    printf("\n... e corresponde a este
endereço de memoria: %d", &num2);
                    pularLinha(2);

                troca(&num1, &num2);

                printf("Trocando os valores de uma
variavel para outra ...\n");
                pularLinha(2);

                    printf("\nO Primeiro numero e igual
%d", num1);
```

```
                printf("\n... e corresponde a este
endereço de memoria: %d", &num1);
                pularLinha(1);
                printf("\nO Segundo numero e igual
%d", num2);
                printf("\n... e corresponde a este
endereço de memoria: %d", &num2);

                pularLinha(1);
                troca(&num1, &num2);

                system("pause");
                tipoCena = 0;
        }
        else if(tipoCena == 6)
        {
                system("cls");
                printf("O primeiro numero %d
multiplicado por ele mesmo é: %d \n", num1,
multiplicaPorEleMesmo(num1));
                printf("O segundo numero %d
multiplicado por ele mesmo é: %d \n", num2,
multiplicaPorEleMesmo(num2));
                system("pause");
                tipoCena = 0;
        }
        else if(tipoCena == 9)
        {
                run = 0;
        }
    }

    saidaAplicacao();
    system("pause");
}
```

```
int calculadora(int selecao, int *salario1, int
*salario2)
{
    switch(selecao)
    {
           case 1:
                  return ((*salario1) + (*salario2));
                  break;
           case 2:
                  return ((*salario1) - (*salario2));
                  break;
           case 3:
                  return ((*salario1) * (*salario2));
                  break;
    }
}

void saudacaoAplicacao()
{
    printf("Bem-vindo ao programa em C");
    pularLinha(1);
}

void saidaAplicacao()
{
    printf("Saindo do programa em C ....");
    pularLinha(1);
}

void menu()
{
    system("cls");
    printf("[1] Inserir Numeros.");
    pularLinha(1);
    printf("[2] Somar.");
    pularLinha(1);
```

```
    printf("[3] Subtrair.");
    pularLinha(1);
    printf("[4] Multiplicar.");
    pularLinha(1);
    printf("[5] Trocar Numeros usando Parametros de
 ponteiro.");
    pularLinha(1);
    printf("[6] Multiplicar os numeros por eles
mesmos.");
    pularLinha(1);
    printf("[9] Sair.");
    pularLinha(1);
}

void pularLinha(int amount)
{
    int i;
    for(i = 0; i < amount; i++)
           printf("\n");
}

void troca(int *p, int *q)
{
    int aux;
    aux = *p;
    *p = *q;
    *q = aux;
}

int multiplicaPorEleMesmo(int num)
{
    return num*num;
}
```

LEITURA

Para complementar seu aprendizado computação, programação e linguagem C sugiro os seguintes links em inglês:

- http://www.cprogramming.com/tutorial/c/lesson3.html Acesso em março de 2015.
- http://www.cprogramming.com/tutorial/c/lesson4.html Acesso em março de 2015.
- http://www.cprogramming.com/tutorial/c/lesson5.html Acesso em março de 2015.
- http://www.cprogramming.com/tutorial/c/lesson6.html Acesso em março de 2015.

Também sugiro a leitura de livros em português como:

- KERNIGHAN, B. W.; RITCHIE, D. M. C: a linguagem de programação, padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.

E também sugiro os seguintes links em português

- http://www2.ic.uff.br/~hcgl/tutorial.html Acesso em março de 2015.

REFERÊNCIAS BIBLIOGRÁFICAS

KERNIGHAN, B. W.; RITCHIE, **D. M. C: a linguagem de programação, padrão** ANSI. Rio de Janeiro: Campus, 1995. 289p.

KELLEY, A.; POHL, I. A Book on C: **Programming in C.** 4. ed. Boston: Addison Wesley, 1997. 726p.

ZIVIANI, N. **Projetos de algoritmos:** com implementações em Pascal e C. São Paulo: Pioneira Thomson, 2002. 267p.

ARAÚJO, J. **Dominando a Linguagem C**. 1. ed. Ciência Moderna, 2004, 146p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. 916p.

FORBELLONE, A. L. **Lógica de Programação.** 2. ed. São Paulo: Makron Books, 2000. 195p.

KNUTH, D. E. **The Art of Computer Programming, Fundamental Algorithms.** 3. ed. Boston: Addison Wesley, 1997. 672p. (vol 1)

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. 4. ed. Porto Alegre: Bookman, 2000. 624p.

3

Vetores e Cadeias de Caracteres

Neste capítulo estudaremos o conceito de vetores e cadeias de caracteres. Vetores são estruturas simples, mas extremamente úteis em programação de computadores. Como veremos, poderemos realizar leituras de variáveis do mesmo tipo (chamadas variáveis homogêneas) usando laços e simplificando nossas tarefas.

Vetores também são a base para outro tipo em Linguagem C: cadeias de caracteres (ou strings). Trata-se de um tipo útil para armazenamento de palavras e textos, pois, como vimos, o tipo char armazena um único caractere.

OBJETIVOS

Neste capítulo, os objetivos principais são aprender sobre:

- Vetores
- Declaração
- Uso e manipulação
- Vetores e funções
- Cadeias de caracteres
- Caracteres, tabela ASCII e caracteres de controle
- Cadeia de caracteres (strings)
- Leitura de caracteres e de cadeia de caracteres
- Funções pré-definidas de manipulação de strings (comprimento, cópia e concatenação)
- Parâmetros da função main()

3.1 Vetores e Cadeias de Caracteres

Vetor, também chamado array ou arranjo, é uma maneira de armazenar vários dados numa mesma variável através do uso de índices numéricos. Os vetores são chamados também de tipos homogêneos, pois devem sempre conter dados do mesmo tipo de variável.

A declaração dos vetores é de maneira semelhante na declaração de variáveis normais. A diferença é que depois do nome da variável deve ser informada a quantidade de elementos do vetor.

Eles são usados para armazenar grandes quantidades de informação em um tipo de variável e para facilitar a manipulação destes dados. Imagine que você precise criar um sistema para uma escola onde você vai armazenar "n" notas, para "n" Alunos, imagine quantas variáveis você teria que declarar! É quase impossível prever a quantidade exata para esses registros. Mas, em linguagens de programação temos o auxílio dos vetores. Sua sintaxe é:

```
tipo nome[numero de elementos];
```

Vamos agora declarar vetor de números inteiros chamado V de 5 posições:

```
int V[5];
```

É importante saber que a quantidade de elementos de um vetor não pode ser alterada depois que o mesmo for declarado. Para criar vetores de tamanho dinâmico, podemos usar ponteiros. Da mesma maneira que podemos inicializar uma variável junto com sua declaração, podemos usar as chaves ({}) para inicializar um vetor.

```
int vetor[5] = {1,3,5,25,58};
```

Para fazer referência a um valor de um elemento contido em um vetor, usamos a notação vetor[índice], que serve tanto para obter quanto para definir o valor de um elemento específico, dada sua posição. Vale ressalter que o primeiro índice de um vetor na linguagem C é sempre zero, ou seja, a nossa variável vetor declarada com 5 posições terá o valor 1 na posição 0, 3 na posição 1, 5 na

posição 2, 25 na posição 3 e 58 na posição 4. Abaixo outra pequena anotação do uso de vetor.

Código 3.1

```
vetor[0] = 3;
 int x = vetor[2];
 int y = vetor[5]; //ERRO EM TEMPO DE EXECUÇÃO!!!
```

Repare em que a última linha contém um erro: ela referencia um elemento do vetor que não existe. Mas, o compilador não se recusará a compilar esse código; dará apenas um aviso. Se essa linha for executada, a variável y receberá um valor que está fora dos limites de um vetor. Em C, o tratamento deste limite fica sob responsabilidade do programador. Mas, dependendo da instrução programada podem ocorrer erros em tempo de execução. Se você tenta acessar uma posição de memória não permitida isto ocasionará um erro e travamento do programa.

Outro aspecto importante a ser citado é que vetores contém seu primeiro elemento na posição 0. Esta posição é chamada de posição base. A partir desta base é calculado um deslocamento de até n-1 posições num vetor declarado com n elementos.

Vejamos o que ocorre na execução do código 3.2:

Código 3.2

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

int main()
{
    int V[5] = {1,3,5,25,58};
    int a = V[0];
    int b = V[2];
    int c = V[5]; //ERRO ?

    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    return 0;
}
```

Declaramos um vetor com 5 elementos na linha 6. Inicializamos as variáveis a, b e c nas linhas 7 a 9. Agora, o que ocorrerá nas linhas 11 a 13? Execute o código e veja o valor resultante da linha 13. É um valor fora dos limites do vetor. Lembre-se, V[5] é a sexta posição do vetor, pois ele começa a contar a partir de 0!

Em C podemos omitir o número de elementos na inicialização de um vetor. O tamanho do vetor será o número de valores inicializados. Por exemplo, as duas notações abaixo são equivalentes:

```
int x[5] = {1, 2, 3, 4, 5};
int x[] = {1, 2, 3, 4, 5};
```

3.2 Inserindo valores em vetores

Para exemplificar, vamos fazer um sistema onde o usuário vai poder inserir 10 números e, logo depois, estes números serão impressos na tela.

Código 3.3

```
#include <stdio.h>

#define MAX 10

int main()
{
   int vetor[MAX];
   int i;

   printf("Insira dez numeros:\n");
   for (i = 0; i < MAX; i++)
   {
      scanf("%d", &vetor[i]);
   }

   printf("Os numeros inseridos no vetor foram:\n");
   for (i = 0; i < MAX; i++)
   {
       printf(" : %d : ", vetor[i]);
   }
   return 0;
}
```

Perceba nas linhas 11 a 14 e 17 a 20, do código 3.3, que usamos laços do tipo for variando o índice i do vetor de acordo com a variação do próprio laço. Esta é a forma mais comum de inserirmos diversas informações num vetor, pois agrupamos as instruções no laço e código fica mais enxuto!

Neste outro exemplo 3.4 pense num programa onde o usuário entra com certa quantidade de números e o programa vai mostrar ao usuário qual foi o maior número digitado.

Código 3.4

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

int main(void)
{
   int vetor[5];
   int x, i;

   printf ("digite 5 numeros\n");

   for (i = 0; i < 5; i++)
/*laço para ler cada elemento do vetor*/
   {
      scanf("%d", &vetor[i] );
   }
   i = 1;
   x = vetor[0];

   while (i < 5)
/* laço de comparação dos elementos */
   {
       if (vetor[i] > x)
       {
          x = vetor[i];
       }
       i++;
```

```
    }

    printf("\n O maior numero digitado foi %d\n", x);

  return 0;
}
```

Podemos ver que o primeiro for é usado para inserir os números pelo usuário. Em seguida atribuímos à variável x o primeiro número do vetor. No while começamos a comparar cada valor dentro do vetor com o x para verificar qual é o maior número dentre eles. Caso o número do vetor seja maior que o número x, passaremos a informação do vetor para a variável x. Ao final do while, o programa informará qual número dentro do vetor é o maior.

Vamos agora fazer um programa, que escolhe 10 números aleatórios, e o usuário pode inserir um número para poder verificar se existe aquele número dentro do vetor, usaremos dois cabeçalhos para poder usar alguns métodos que serão comentados dentro do código.

Código 3.5

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // para usar rand ou srand
#include <time.h> //necessário p/ função time()
#define MAX 10

int main()
{
   int vetor[MAX], num;
   int i;

   /* srand(time(NULL)) serve para inicaiar o gerador
    * de números aleatórios com o valor da função
    * time(NULL). Este por sua vez, é calculado
    * como sendo o total de segundos passados desde
    * 1 de janeiro de 1970 até a data atual.
    * Desta forma, a cada execução o valor da "semente"
```

```
     * será diferente.
     */
    srand(time(NULL));

    printf("Gerando 10 numeros aleatorios. \n");

    for (i = 0; i < MAX; i++)
    {
        vetor[i] = rand() % 100;
// %100 indica que os numeros serão entre 0 a 99, resto da divisão
por 100
    }
    for (i = 0; i < MAX; i++)
    {
       printf("%d :: ", vetor[i]);
    }
    printf("\nDigite um numero verificarmos se ele
exite na lista: ");
    scanf("%d", &num);

    for (i = 0; i < MAX; i++)
    {
        if(num == vetor[i]){
            printf("%d -> este numero existe no vetor
e esta na posição %d.", num, i);
            break;
        }
        else if(i == MAX - 1)
        {
            printf("Este numero não existe no vetor:
 %d", num);
        }
    }
    return 0;
}
```

3.3 Vetores e Funções

Na passagem de vetores como parâmetros de funções estamos sempre passando um ponteiro para o primeiro elemento do vetor, ou seja ele contém o endereço de seu 1º elemento. Isto implica dizer que a passagem de vetores como parâmetros de funções é sempre por referência. Por exemplo:

```
float f (float V1[MAX]);
```

A função f seria chamada da seguinte maneira:

Código 3.6

```
#define MAX 100

int main() {

        double a, A[MAX];
        //algum código aqui
        a = f(A);
        //algum código aqui
}
```

Podemos notar no código 3.6 que declaramos as variáveis a (double) e A (um vetor de variáveis double) na linha 5. O vetor é passado apenas pelo nome, pois ele contém o endereço do 1º elemento.

Sabemos agora que o nome do vetor utilizado na sua declaração pode ser utilizado também como um ponteiro que aponta para o endereço da primeira posição do vetor, ou seja, o nome do vetor é também um ponteiro que contém o endereço de seu primeiro elemento.

Não esqueça então que: na linguagem C o vetor é sempre passado para uma função por referência, ou seja, apenas o nome (ponteiro) do vetor é passado na chamada de uma função e será acessado internamente na sua posição original. Não é uma cópia de seus valores que são passados e alterações serão refletidas no vetor original.

Vamos ver outro exemplo, definindo uma função que recebe dois vetores de tamanho MAX e retorna o seu produto escalar. A declaração desta função seria:

```
double prodEscalar (double V1[MAX], double V2[MAX], int N);
```

A função recebe os vetores V1 e V2, e um inteiro N. Veja que cada vetor possui MAX elementos cada. O produto escalar é calculado somando-se todas as multiplicações de cada um dos elementos de um vetor pelo outro na mesma posição.

Código 3.7

```
double prodEscalar (double V1[MAX], double V2[MAX], int N) {
  int i;
  double res = 0;
  for (i=0; i<N; i++)
     res = res + V1[i] * V2[i];
  return res;
}
```

Acreditamos que esse seja o protótipo mais "claro", pois ele define que V1 e V2 são vetores de tamanho máximo. Porém, há outro protótipo possível:

```
double prodEscalar (double *V1, double *V2, int N);
```

Note que o tamanho do vetor é facultativo na definição da função, e como o nome do vetor é um ponteiro, podemos até mesmo definir esses parâmetros como ponteiros no protótipo da função.

No código 3.7, a seguir, podemos ver a aplicação da nossa função.

Código 3.8

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

#define MAX 100 // tamanho maximo do vetor definido
por uma constante
#define EPS 0.001 //define um parametro de comparacao

double prodEscalar (double V1[MAX], double V2[MAX],
 int N);
```

```
int main (void) {
  int n, i;
  double vetA[MAX], vetB[MAX];
  double prod;

  /* lendo o tamanho dos vetores */
  printf("Tamanho dos vetores: ");
  scanf("%d", &n);
  printf("Valores do vetor A\n");

  for (i=0; i<n; i++)
   scanf("%f",&vetA[i]);

  printf("Valores do vetor B\n");

  for (i=0; i<n; i++)
   scanf("%f",&vetB[i]);

  prod = prodEscalar(vetA, vetB, n);

  printf("\n%d <- prod\n", prod);
  //aqui usamos o EPS como variavel de precisao
  if (prod < EPS && prod > -EPS)
    printf("Vetores Ortogonais\n");
  else
    printf("Vetores NAO Ortogonais\n");
  return 0;
}

double prodEscalar (double V1[MAX], double V2[MAX],
int N) {
  int i;
  double res = 0;
  for (i=0; i<N; i++)
     res = res + V1[i] * V2[i];
  return res;
}
```

Vamos agora fazer uma função para retornar um vetor com a soma de dois outros vetores. A função agora deverá receber 2 vetores de entrada e devolver 1 vetor como resultado. Sendo assim, temos o seguinte protótipo:

```
void somaVetorial ( int A[MAX], int B[MAX], int Res[MAX], int N);
```

Onde os vetores A e B são entradas e Res é o vetor de saída. O valor N é usado para indicar o limite até onde somaremos o vetor. Uma possível implementação para esta função seria:

Código 3.9

```
void somaVetorial ( int A[MAX], int B[MAX], int Res[MAX], int N) {
  int i;
  for(i=0; i<N; i++)
    Res[i] = A[i] + B[i];
}
```

Agora, vejamos como fica a implementação de um programa que use nossa função somaVetorial().

Código 3.10

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

#define MAX 10

void somaVetorial ( int A[MAX], int B[MAX], int
 Res[MAX], int N);

int main (void) {
  int n, i;
  int vetA[MAX], vetB[MAX], vetRes[MAX];

  /* lendo o tamanho dos vetores */
  printf("Tamanho dos vetores: ");
  scanf("%d", &n);
```

```
  printf("Valores do vetor A\n");
  for (i=0; i<n; i++)
   scanf("%d",&vetA[i]);

  printf("Valores do vetor B\n");
  for (i=0; i<n; i++)
   scanf("%d",&vetB[i]);

  somaVetorial(vetA, vetB, vetRes, n);
  printf("Resultado: ");
  for (i=0; i<n; i++)
   printf("%d ",vetRes[i]);

  printf("\n");
  return 0;
}

void somaVetorial ( int A[MAX], int B[MAX], int
Res[MAX], int N) {
  int i;
  for(i=0; i<N; i++)
    Res[i] = A[i] + B[i];
}
```

3.4 Vetores E Cadeias De Caracteres

Na linguagem C há dois tipos de declaração para caracteres: sem sinal (= unsigned characters) e com sinal (= signed characters). Um caractere sem sinal é um número natural entre 0 e 255; um caractere com sinal é um número inteiro entre -128 e 127.

```
unsigned char u;
```

Para criar uma variável c do segundo tipo:

```
char c;
```

Cada caractere ocupa um byte na memória do computador e os padrões de bits dos dois tipos são os mesmos: de 00000000 a 11111111.

3.5 Tabela ASCII

ASCII é um acrônimo para *"American Standard Code for Information Interchange"* (Código Padrão Norte Americano para Intercâmbio de Informações). Trata-se de um código binário que codifica sinais gráficos e sinais de controle. Sinais gráficos são letras, sinais de pontuação e sinais matemáticos. Sinais de controle são caracteres não imprimíveis, em geral, para ajustes dos sinais gráficos (exemplo: início de cabeçalho).

CONEXÃO

Você pode ler mais sobre a tabela ASCII aqui:

- http://pt.wikipedia.org/wiki/ASCII

Veja abaixo um recorta da tabela ASCII na figura 3.1:

BIN	OCT	DEC	HEX	SINAL	BIN	OCT	DEC	HEX	SINAL	BIN	OCT	DEX	HEX	SI-NAL
0010 0000	040	32	20	(espaco)	0100 0000	100	64	40	@	0110 0000	140	96	60	`
0010 0001	041	33	21	!	0100 0001	101	65	41	A	0110 0001	141	97	61	a
0010 0010	042	34	22	"	0100 0010	102	66	42	B	0110 0010	142	98	62	b
0010 0011	043	35	23	#	0100 0011	103	67	42	C	0110 0011	143	99	63	c
0010 0100	044	36	24	$	0100 0100	104	68	44	D	0110 0100	144	100	64	d
0010 0101	045	37	25	%	0100 0101	105	69	45	E	0110 0101	145	101	65	e
0010 0110	046	38	26	&	0100 0110	106	70	46	F	0110 0110	146	102	66	f
0010 0111	047	39	27	'	0100 0111	107	71	47	G	0110 0111	147	103	67	g
0010 1000	050	40	28	(	0100 1000	110	72	48	H	0110 1000	150	104	68	h
0010 1001	051	41	29	)	0100 1001	111	73	49	I	0110 1001	151	105	69	i
0010 1010	052	42	2A	*	0100 1010	112	74	4A	J	0110 1010	152	106	6A	j

Figura 3.1 – Recorte da tabela ASCII. Fonte: http://pt.wikipedia.org/wiki/ASCII.

Você também pode executar um código em C que imprima sua própria tabela ASCII. Veja o código 3.11. Neste caso, são só os caracteres imprimíveis, por isto i começa em 32, na linha 07. Se você usar outras máscaras de formatação poderá ver os valores em hexa ou octal.

Código 3.11

```
#include <stdio.h>

int main(void)
{
    int i;

    for (i = 32; i <= 126; i++)
    {
        printf("%c [%d]\n", i, i);
    }

    return 0;
}
```

3.6 Vetores e Cadeias de Caracteres

A linguagem C não possui um tipo para strings ou cadeia de caracteres. Por isto, nós usamos um vetor de *char* para representar uma string. Trata-se de um conjunto finito de caracteres com marcador de final '\0'. Os caracteres contidos na string serão caracteres da tabela ASCII.

Para definir e inicializar uma cadeia de caracteres podemos fazer da seguinte forma:

```
char saudacao[20] = "Ola mundo!";
```

Acima, temos um vetor com até 20 caracteres. Como precisaremos definir o final da string com '\0' teremos 19 caracteres possíveis de conteúdo válido.

Cadeias de caracteres são vetores, como vimos, e podemos acessar uma determinada posição para alterar seu valor.

```
saudacao[9] = "?";
```

Neste outro exemplo definimos uma cadeia de caracteres através de reserva de espaço em memória, usando a função *malloc()*. Veremos mais sobre a função de reserva de memória, mas, por hora, o que precisamos é saber que a função reserva memória de forma dinâmica (pode mudar ao longo do programa) e apontaremos para esta memória reservada através de um ponteiro.

Código 3.12

```
char *s;
s = malloc( 10 * sizeof (char));
s[0] = 'A';
s[1] = 'B';
s[2] = 'C';
s[3] = '\0';
s[4] = 'D';
```

No código 3.11 nós criamos um ponteiro para char, na linha 1. Em seguinda reservamos 10 espaços consecutivos de char na memória, linha 2. Depois, veja que podemos acessar estes espaços como se fosse um vetor!

Após a execução desse código, o vetor s conterá a string "ABC". O caractere nulo marca o fim dessa string, logo a a porção s[4..9] do vetor será ignorada.

O tamanho (length) de uma string é o seu número de caracteres, sem contar o caractere nulo final. O endereço de uma string é o endereço do seu primeiro caractere, da mesma forma que o endereço de um vetor é o endereço de seu primeiro elemento.

Para mostrar e ler uma cadeia de caracteres usando os comandos printf e scanf usamos este formato:

```
printf("%s", saudacao);

scanf("%s", saudacao);
```

Usamos a sequência especial %s dentro dos comandos, para poder mostrar e fazer a leitura de uma cadeia de caracteres.

Código 3.13

```
#include <stdio.h>

int main()
{
   char saudacao[100];

   printf("Escreva um nome:");
   scanf("%s", saudacao);
   printf("\nOla %s, como vai?", saudacao);

   return 0;
}
```

Se quiséssemos ler ou exibir apenas 1 caractere usaríamos:

```
printf(“%c”, &texto);
scanf(“%c”, texto);
```

Neste pequeno código em 3.14 podemos ver um exemplo de um programa que faz a leitura somente de um caractere.

Código 3.14

```
#include <stdio.h>

int main()
{
   char texto;
   printf("Digite uma letra: ");
   scanf("%c", &texto);

   printf("\nEsta letra eh: %c\n", texto);

   return 0;
}
```

Experimento digitar mais de uma letra na leitura e veja o que acontece. Na figura abaixo nós mostramos!

Figura 3.2 – Leitura de caracteres em Linguagem C.

Agora vamos a um exercício desafiador! Imagine que precisamos criar um programa que faz a leitura de uma palavra e que informe a quantidade de vogais que está palavra tem. Abaixo, no código 3.15, vamos uma função que recebe uma string como parâmetro e retorna a quantidade de vogais que existe nessa string.

Código 3.15

```
int quantasVogais( char *s) {
  int numVogais, i;
  char *vogais;
  vogais = "aeiouAEIOU";
  numVogais = 0;

  for (i = 0; s[i] != '\0'; ++i) {
     char ch = s[i];
     int j;
     for (j = 0; vogais[j] != '\0'; ++j) {
        if (vogais[j] == ch) {
           numVogais += 1;
           break;
        }
     }
  }
  return numVogais;
}
```

Agora, no código 3.16, criamos um exemplo para testar nossa função. Insira o código da sua função depois do código main.

Código 3.16

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

int quantasVogais(char *s);

int main()
{
   char entrada[100];
   printf("Palavra para teste: ");
   scanf("%s", entrada);

   printf("\nNumero de vogais: %d\n",
quantasVogais(entrada));

   return 0;
}
```

3.7 Funções Pré-Definidas de Manipulação de Strings (Comprimento, Cópia e Concatenação)

Em Linguagem C, a biblioteca padrão que possui funções para manipular strings é a *string*.h. Para fazer uso das funções você deverá usar o include apropriado.

A seguir, vamos apresentar exemplos de várias funções para manipular strings. Um detalhe importante é que, como vimos, string é um vetor de char, logo você mesmo poderia implementar cada uma destas funções!!

Usaremos a função *gets()* para capturar strings ao invés de *scanf()*. Faremos isto, pois a última não captura cadeias complexas, com espaços, por exemplo ou tabulações.

O funcionamento de *gets()* é muito simples:

```
char buffer[50];
gets(buffer);
```

Você declara uma variável que será a string alvo, no caso um vetor de char. E passa o nome da variável para a função gets().

Para verificar o tamanho de uma string (tamanho será a quantidade de valores úteis/válidos na string, ou seja, tudo antes do '\0') usamos a função strlen(). O nome da função é a abreviação de *string length* e devolve o comprimento da string passada como parâmetro.

Vamos a um exemplo! Veja no código 3.17 um programa que lê um email do usuário e informa o tamanho deste email em caracteres.

Código 3.17

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

int main(void)
{
   char email[100];
   int tamanho;

   printf("Informe um email: ");
   gets(email);

   tam = strlen(email);

   printf("\nSeu email %s tem %d caracteres", email,
tam);

   return 0;
}
```

É possível tratar a entrada do usuário de forma que o tamanho do email digitado seja válido.

Código 3.18

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

#define MAX 20

int main(void)
{
   char email[100];
   int tamanho = 0;

   do
   {
       printf("Informe um email: ");
       gets(email);

       int aux = strlen(email);
   } while (tamanho <= 0 || tamanho >= MAX);
   return 0;
}
```

Para operações de cópia ou atribuições com strings usamos a função strcpy(). Isto, pois a string é um vetor de char. A função strcpy (o nome é uma abreviatura de string copy) recebe duas strings e copia a segunda (inclusive o caractere nulo final) para o espaço ocupado pela primeira. O conteúdo original da primeira string é perdido. Antes de chamar a função, você deve se certificar de que o espaço alocado para a primeira string é suficiente para guardar a cópia da segunda.

Imagine um programa que leia o nome e o sobrenome de um usuário e depois inverta estes valores nas variáveis usando strcpy.

Código 3.19

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

int main(void){
   char nome[30], sobre[30], aux[30];
   printf("Qual o seu nome? ");
   gets(nome);
   printf("Qual o seu sobrenome? ");
   gets(sobrenome);

   //troca nome e sobrenome
   strcpy(aux,nome);
   strcpy(nome,sobrenome);
   strcpy(sobrenome,aux);
   printf("\nSeu nome eh %s", nome);
   printf("\nSeu sobrenome eh %s", sobrenome);
   return 0;
}
```

Veja na linha 13 que usamos a variável aux, declarada na linha 6, para receber temporariamente o valor de nome. Se não fizéssemos isto perderíamos este valor, pois seria sobrescrito pelo sobrenome na linha 14.

Para comparar duas strings em C usamos a função strcmp (abreviatura de *string compare*) que faz uma comparação lexicográfica. Ela devolve um número negativo se a primeira string for lexicograficamente menor que a segunda, devolve 0 se as duas strings são iguais e devolve um número positivo se a primeira string for lexicograficamente maior que a segunda.

Código 3.20

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

int main(void){
   char palavra1[30], palavra2[30];
   int comp;
```

```
    printf("Qual a primeira palavra? ");
    gets(palavra1);
    printf("Qual a segunda palavra? ");
    gets(palavra2);

    //compara as palavras
    comp = strcmp(palavra1, palavra2);

    if(comp == 0)
        printf(“\nAs palavras são iguais”);

    if(comp > 0)
        printf(“\nPalavra 1 eh maior que Palavra 2”);

    if(comp < 0)
        printf(“\nPalavra 1 eh menor que Palavra 2”);

    return 0;
}
```

Quando precisamos unir duas strings usamos a função strcat(). A função strcat() é usada para concatenar a string fonte, na string destino. Por exemplo um programa que receba o nome e o sobrenome do usuário em campos separados e unifica os valores num mesmo campo. Vejamos o código 3.21.

Código 3.21

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

int main(void) {
   char nome[30], sobrenome[30];

   // receber as variaveis
   printf("Informe o nome: ");
```

```
    gets(nome);
    printf("Informe o sobrenome: ");
    gets(sobrenome);

    // Juntar os nomes e exibir
    strcat(nome, sobrenome);

    printf("\nNome Completo: %s", nome);
    getch();
    return 0;
}
```

Pelo código 3.21, na linha 14, vemos que a variável nome receberá a variável sobrenome. Agora, atente-se ao resultado da execução na figura 3.3. Veja que o nome ficou justaposto ao sobrenome. É preciso adicionar um espaço entre os dois, certo?

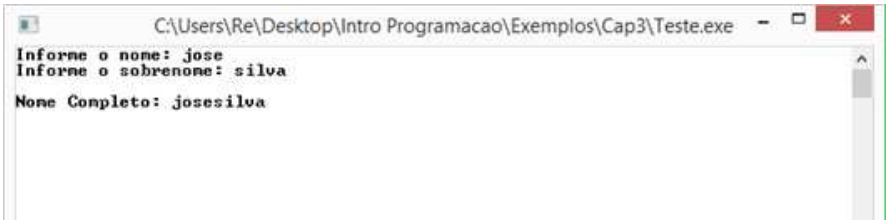


Figura 3.3 – Concatenando strings em Linguagem C.

No código 3.22 fiz um pequeno ajusteque você pode conferir na linha 14. Faça também no seu código, compile-o e execute-o novamente.

Código 3.22

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

int main(void) {
    char nome[30], sobrenome[30];
```

```
    // receber as variaveis
    printf("Informe o nome: ");
    gets(nome);
    printf("Informe o sobrenome: ");
    gets(sobrenome);

    // Juntar os nomes e exibir
    strcat(nome, " ");
    strcat(nome, sobrenome);

    printf("\nNome Completo: %s", nome);
    getch();
    return 0;
}
```

Agora, veja na figura 3.4 que nosso exemplo deu certo! Ou seja, agora nome e sobrenome não estão justapostos, mas sim com um espaço entre eles.

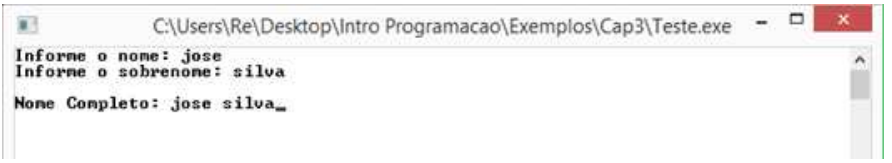


Figura 3.4 – Concatenando strings em Linguagem C, adicionando um espaço no exemplo!

Nós havíamos comentado sobre a possibilidade de escrevermos nossas próprias funções para tratamento de strings em Linguagem C. Agora, veremos um pouco mais sobre isto. Vamos implementar as seguintes funções: concatenacao(), tamanho(), impressao(), copia().

Para todos os exemplos vou apresentar as funções. Você deve inserí-las no seu programa colocando os devidos protótipos e chamando-as na sua função main(), ok?

Vamos começar com uma função que recebe uma string e a imprime. Veja no código 3.23 que usamos um for (linha 4) para percorrer o vetor de caracteres s.

Código 3.23

```
void impressao (char* s)
{
    int i;
    for (i=0; s[i] != '\0'; i++)
           printf(“%c”, s[i]);
    printf(“\n”);
}
```

Agora vejamos uma função que retorna o tamanho de uma string, percorrendo-a até encontrar o elemento ‘\0’e contando os caracteres válidos antes disto.

Código 3.24

```
int tamanho (char* s)
{
    int i;
    int n = 0;
    /* contador */
    for (i=0; s[i] != '\0'; i++)
           n++;
    return n;
}
```

A função do código 3.25 copia os elementos de uma cadeia de origem (orig) para uma cadeia de destino (dest). A cadeia de destino deverá ter espaço suficiente para receber a origem.

Código 3.25

```
void copia (char* dest, char* orig)
{
    int i;
    for (i=0; orig[i] != '\0'; i++)
           dest[i] = orig[i];
    /* limitação da cadeia copiada */
    dest[i] = '\0';
}
```

A função “concatenacao” copia os elementos de uma cadeia de origem (orig) para o final da cadeia de destino (dest) colocando-os de forma justaposta.

Código 3.26

```
void concatenacao (char* dest, char* orig)
{
    int i = 0;
    int j;

    i = 0;
    while (dest[i] != '\0')
           i++;

    for (j=0; orig[j] != '\0'; j++)
    {
           dest[i] = orig[j];
           i++;
    }

    dest[i] = '\0';
}
```

Veja na função do código 3.26 que usamos dois índices para concatenar as cadeias: i e j. O índice i é usado na cadeia de destino, começando em zero. O índice j é usado na cadeia de origem. Nas linhas 7 a 8 o while é usado para encontrar o final da cadeia de destino (afinal, a concatenação será a partir dali).

Nas linhas 10 a 14 o for é usado para copiar cada caractere da cadeia de origem na cadeia de destino. Depois, colocamos o marcado de final de cadeia na linha 16 para fechar a cadeia de destino.

3.8 O Parâmetros da Função Main()

É possível passar informações para um programa quando ele é executado. Essas informações são passadas por meio de argumentos de linha de comando que devem estar declarados na função main.

Tratam-se de dois argumentos internos especiais: *argc e argv*.

• **Parâmetro argc:** Contém o número de argumentos da linha de comando. O nome do programa é qualificado como um primeiro argumento

• **Parâmetro argv:** É um ponteiro para uma matriz de ponteiros para caracteres. Cada elemento nessa matriz aponta para um argumento da linha de comando. Os argumentos da linha de comando são string.

Quaisquer números passados na linha de comando terão que ser convertidos pelo programa no formato interno apropriado. Os argumentos são separados por um espaço ou um caractere de tabulação.

Por exemplo, vamos criar um programa hello que nós dá uma saudação e nos fala sobre nossa idade se ela for passada.

Código 3.10

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

// [] indica que a matriz eh
// de tamanho indeterminado

void main (int argc, char *argv[]) {
  if (argc < 2) {
    printf("O nome deve ser informado. \n");
    exit(EXIT_FAILURE);
  }
  else if (argc == 2) {
    printf("Ola %s, como vai?\n", argv[1]);
    exit(EXIT_SUCCESS);
  }
  else if (argc == 3) {
    printf("Ola %s, como vai?\n", argv[1]);
    printf("Hmm, voce tem %s anos!\n", argv[2]);
    exit(EXIT_SUCCESS);
  }
  else {
    printf("Voce digitou mais de 2 argumentos\n");
    printf("So sei trabalhar com 2!\n");
    exit(EXIT_FAILURE);
  }
}
```

Só que, desta vez, vamos usar o compilador online (você também pode usar o Dev, desde que vá na pasta bin e coloca lá seu arquivo .exe depois de compilar seu programa. Ou, configure o caminho da pasta bin do seu Dev na variável de ambiente path). Veja o link do compilador online no capítulo 1.

Após digitar seu programa, compile-o com o comando abaixo:

```
gcc -o hello hello.c
```

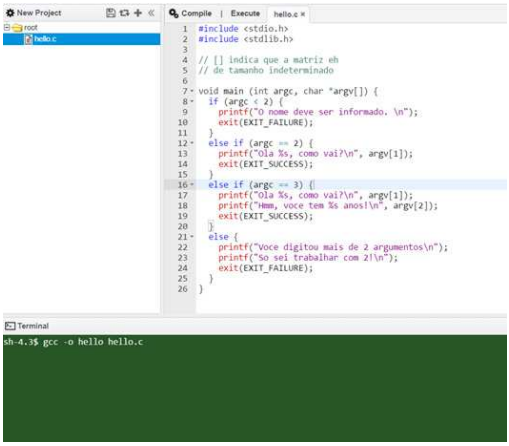


Figura 3.5 – Escrevendo e compilando nosso exemplo no compilador online.

Com o programa compilado agora vamos testar as possibilidades. Primeiro digite o comando abaixo e dê enter:

```
./hello
```

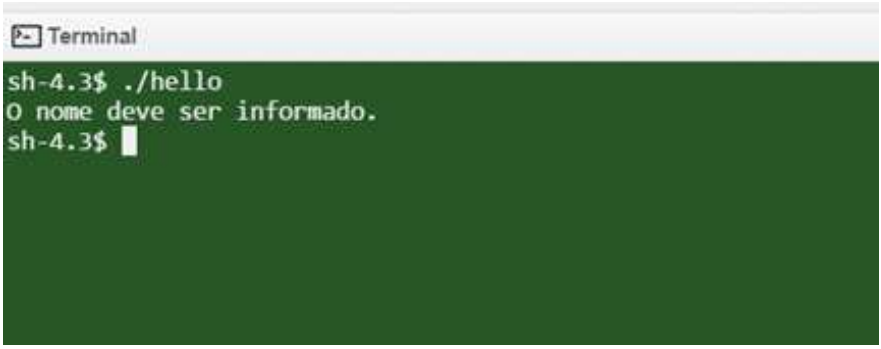


Figura 3.6 – O primeiro teste sem informar parâmetros.

Veja que ele trata o nome do programa como primeiro parâmetro. Faltou passar o segundo parâmetro que é o nome da pessoa! Agora tente assim:

```
./hello reginaldo
```

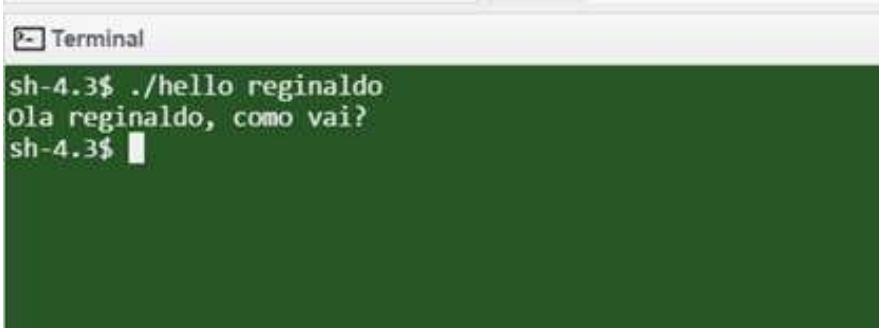


Figura 3.7 – O segundo teste informando apenas 1 parâmetro.

Ele nos dá uma saudação! Tente agora:

```
./hello Reginaldo 18
```

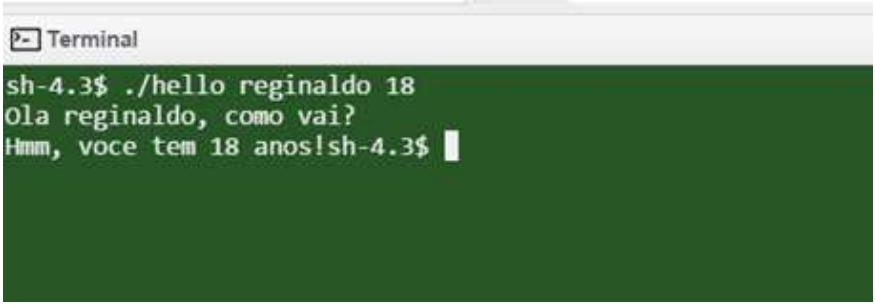


Figura 3.8 – O terceiro teste com dois parâmetros.

Ele nos dá uma saudação e também fala sobre nossa idade! Por fim, tente:

```
./hello Reginaldo 18 de idade
```

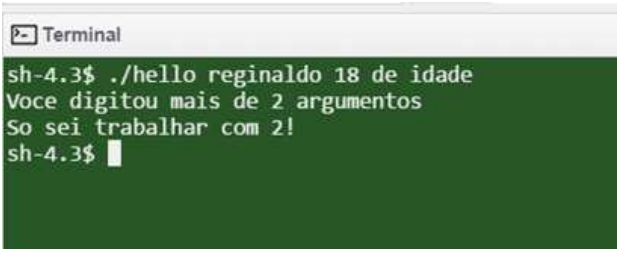


Figura 3.9 – O quarto teste com mais de dois parâmetros.

Ele diz que não sabe trabalhar com mais de 2 argumentos.

Isto dá uma boa base para escrever um programa interativo para interpretar comandos, não?

REFLEXÃO

Nesse capítulo vimos as mais variadas formas de utilização de vetores, do uso de cadeias de caracteres, a passagem de vetores como parâmetros de funções e conhecemos a tabela ASCII.

Também vimos como receber argumentos em linha de comando, algo que pode ser muito legal para escrever programas que conversem com comandos!!

Convido você a pesquisar sobre o que é um shell, como são criados e tentar melhorar nosso programa hello. Topa o desafio?

LEITURA

Para complementar seu aprendizado computação, programação e linguagem C sugiro os seguintes links em inglês:

- http://www.cprogramming.com/tutorial/c/lesson8.html Acesso em março de 2015. Um detalhe importante aqui é que ainda não falamos de matrizes. Vetores são estruturas unidimensionais. Veremos sobre matrizes no capítulo seguinte.
- http://www.cprogramming.com/tutorial/c/lesson9.html Acesso em março de 2015.

Também sugiro a leitura de livros em português como:

- KERNIGHAN, B. W.; RITCHIE, D. M. C: a linguagem de programação, padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.

E também sugiro os seguintes links em português

- http://www2.ic.uff.br/~hcgl/tutorial.html Acesso em março de 2015.

REFERÊNCIAS BIBLIOGRÁFICAS

KERNIGHAN, B. W.; RITCHIE, D. M. C: **a linguagem de programação**, padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.

KELLEY, A.; POHL, I. A Book on C: **Programming in C**. 4. ed. Boston: Addison Wesley, 1997. 726p.

ZIVIANI, N. **Projetos de algoritmos:** com implementações em Pascal e C. São Paulo: Pioneira Thomson, 2002. 267p.

ARAÚJO, J. **Dominando a Linguagem C**. 1. ed. Ciência Moderna, 2004, 146p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos**: teoria e prática. Rio de Janeiro: Campus, 2002. 916p.

FORBELLONE, A. L. **Lógica de Programação**. 2. ed. São Paulo: Makron Books, 2000. 195p.

KNUTH, D. E. **The Art of Computer Programming**, Fundamental Algorithms. 3. ed. Boston: Addison Wesley, 1997. 672p. (vol 1)

SEBESTA, R. W. **Conceitos de Linguagens de Programação.** 4. ed. Porto Alegre: Bookman, 2000. 624p.

4

Tipos Estruturados e Matrizes

Estruturas são usadas para agrupamento de dados de diferentes tipos. Imagine, por exemplo, a necessidade de um tipo particular de dado para armazenar informações sobre alunos, como nome, número de matrícula, disciplinas que está cursando com as respectivas notas etc. Sim, você poderia usar variáveis de tipo primitivo ou até mesmo vetores. Mas as variáveis são limitadas ao próprio tipo e os vetores só podem possuir um único tipo de dado para todas as variáveis.

Você já conhece os tipos de dados mais simples e usados em linguagem C: char, int, double e void. Agora, vamos usar estes e outros tipos para criar tipos complexos, definidos conforme nossa necessidade de aplicação.

Também veremos a criação de matrizes, estruturas homogêneas com mais de uma dimensão (veremos o que o termo dimensão significa) e tipos enumerados, pré-definidos.

OBJETIVOS

Neste capítulo, os objetivos principais são aprender sobre:

- Estruturas Heterogêneas (Registros).
- Criação e Manipulação de Estruturas.
- Estruturas e Funções.
- Ponteiros e Estruturas.
- Vetores de Estruturas.
- Aninhamento de Estruturas.
- Tipos Enumerados.
- Matrizes.
- Matrizes e Funções.

4.1 Estruturas Heterogêneas

No estudo sobre vetores nós vimos que eles são usados para armazenamento de dados do mesmo tipo, também chamados de estruturas homogêneas. Ao usar um índice do vetor estamos acessando uma variável (um espaço de memória) do mesmo tipo declarado deste vetor.

Na linguagem C nós podemos definir novos tipos de dados, através do agrupamento de outros tipos. Isto é chamado de estrutura heterogênea ou simplesmente estrutura. É comum alguns autores usarem também o nome de registros.

Para este tipo de definição usaremos um operador especial chamado *struct.* Este operador serve para agrupar outros tipos de dados numa definição de novo tipo.

Na definição de estruturas podemos usar tipos primitivos, ou seja tipos já previstos pela linguagem de programação, ou tipos compostos, aqueles criados pelas definições do programador.

Os elementos internos a uma estrutura são chamados de membros e serão acessados por operador próprio. Assim, a estrutura é vista como um tipo composto e poderemos ter variáveis neste formato.

A forma padrão de declaração de uma estrutura é:

Código 4.1

```
struct nome_estrutura
{
//declaração dos membros
} definição de variáveis (opcional);
```

O exemplo abaixo de código mostra a declaração de uma estrutura que contém um elemento do tipo inteiro e outro elemento do tipo char.

Código 4.2

```
     struct exemplo
{
           int num;
           char letra;
           };
```

É importante separar o conceito de declaração de estrutura da declaração de uma variável. A declaração de estrutura envolve a definição de tipo. A declaração de variável envolve a reserva de espaço em memória para uma dada variável.

Assim, nos códigos 4.1 e 4.2 ainda não há reserva de memória, apenas a definição de tipo. Veja o exemplo do código 4.3.

Código 4.3

```
#include<stdio.h>

struct exemplo
{
    int num;
    char letra;
};

int main(void) {
    struct exemplo var1;

    return 0;
}
```

Veja no código 4.3 que a declaração é feita antes do código da função main(). Na função main a variável var1 é do tipo da estrutura definida, como visto na linha 10.

Em linguagem C normalmente as declarações de estruturas são globais. Elas são implementadas próximas ao topo do arquivo com o código fonte do programa, assim elas são visíveis por todas as funções.

Na declaração do código 4.3, feita entre as linhas 3 e 7, é criado um novo tipo de estrutura com o nome de exemplo. Esta estrutura contém um número inteiro chamado num e um caractere chamado letra.

Quando declaramos variáveis de tipos primitivos é preciso informar o nome do tipo e o nome da variável. Isso também acontece com estruturas, mas há a necessidade do uso da palavra reservada *struct* na frente do nome da estrutura (observe a linha 10 do código 4.3):

```
struct exemplo var1;
```

É possível declarar variáveis do tipo definido colocando-as na frente da declaração, como visto no exemplo abaixo do código 4.4 .

Código 4.4

```
struct exemplo
{
            int num;
            char letra;
      } var1;

// ou ...
    struct exemplo
{
            int num;
            char letra;
      } var1, var2, var3 ,..., varN;
```

Quando declaramos a estrutura antes da função main ela funciona como uma declaração global, ou seja, seu escopo será global e poderá ser usada em qualquer parte do programa.

Mas, é possível realizar também declarações locais de estruturas. Vejamos o código 4.5.

Código 4.5

```
#include<stdio.h>

int main(void) {
    struct exemplolocal
    {
        int num;
        char letra;
    } var2;

    struct exemplolocal var1;

     return 0;
}
```

Nas linhas 4 a 8 declaramos uma estrutura chamada exemplolocal que contém um número inteiror e um caractere apenas. Na linha 10 declaramos a variável var1 que é do tipo *struct exemplolocal*. A variável var2 também é do mesmo tipo e foi declarada logo após a definição da estrutura.

Porém, neste caso, o escopo da estrutura declarada está restrito à função *main()*. Se você tentar usar uma variável deste tipo em outra função receberá um erro de compilação.

Digite o código 4.6 e depois tente compila-lo.

Código 4.5

```
#include<stdio.h>

void funcao(void);

int main(void) {
    struct exemplolocal
    {
        int num;
        char letra;
    } var2;

    struct exemplolocal var1;

     return 0;
}

void funcao(void) {

    struct exemplolocal varF;

}
```

Você verá um erro de compilação, como mostra a figura 4.1.

Figura 4.1 – Estrutura com declaração local.

Para que a *struct exemplolocal* possa ser usada na sua funcao() é necessário declará-la antes da função *main()*.

4.2 Acesso aos Membros de uma Estrutura

Para atribuir ou obter os valores dos membros de uma estrutura devemos acessar suas variáveis internas. Isto é feito através dos nomes dados à elas na definição da estrutura.

Quando declaramos uma estrutura, ela é definida e associada com uma alocação de memória suficiente para alocar os bytes de seus elementos. No exemplo 4.3, uma estrutura representa uma variável inteira e uma variável do tipo char.

O acesso aos valores de uma estrutura para leitura ou escrita é feito através do operador ponto (.):

```
exe.num //acessa variável num na estrutura nome exe

exe.nome //acessa variável nome na estrutura exe
```

Os membros da estrutura exe como o num ou nome são variáveis a serem usadas como variáveis comuns e os mesmos operadores que já vimos. Por exemplo, podemos usar a função *scanf()* para leitura de uma dessas variáveis.

O código 4.6 mostra um exemplo de utilização da estrutura exemplo.

Código 4.6

```
#include <stdio.h>
struct exemplo
{
    int n;
    char c;
} exe;

int main()
{
    exe.n = 3;
    exe.c = 'M';

    printf("Estrutura:  %d, %c", exe.n, exe.c);

    exe.n++;
    exe.c = 'N';

    printf("\nAtualizado:  %d, %c", exe.n, exe.c);

    return 0;
}
```

Veja que definimos a estrutura nas linhas 2 a 6 e já declaramos a variável "exe" do tipo desta estrutura definida.

Na linha 10 atribuímos valor ao membro "n" da estrutura que criamos. Na linha 11 atribuímos valor ao membro "c" que é do tipo *char*.

Veja pela figura 4.2 que as linhas 15 e 16 surtem o efeito desejado.

Figura 4.2 – Manipulação de estruturas.

Um ponto importante a destacar é que o operador ponto (.) tem precedência sobre o operador de incremento unário (++). Do contrário teríamos um erro de compilação.

Caso tente acessar um nome de membro que não foi declarado dentro daquela estrutura, causara um erro de compilação também. Substitua a linha 15 por esta:

```
exe.x = 10;
```

Agora veja o resultado na figura 4.3.

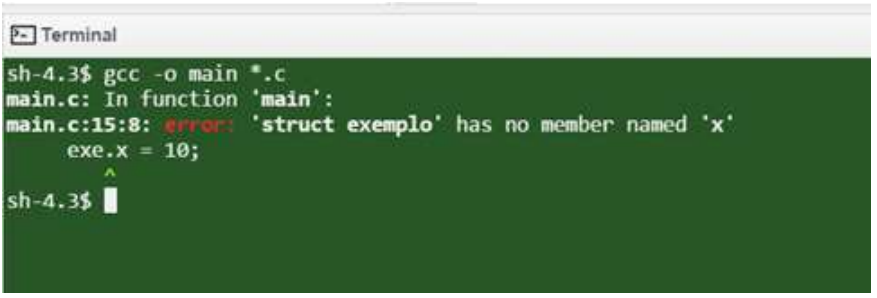


Figura 4.3 – Acesso a membro não declarado numa estrutura.

É possível também realizar a atribuição direta entre estruturas, ou seja, copiar uma estrutura para outra estrutura, através do operador "=". Para isto, basta que as estruturas sejam do mesmo tipo.

Código 4.7

```
#include <stdio.h>
struct data
{
    int dia;
    char mes;
    int ano;
};
```

```
int main()
{
    struct data d1, d2;

    d1.dia = 3;
    d1.mes = 'D';
    d1.ano = 2015;

    /* Passando valor de e1 para e2*/
    d2 = d1;

    printf("Data 1:  %d / %c / %d", d1.dia, d1.mes, d1.ano);
    printf("\nData 2:  %d / %c / %d", d2.dia, d2.mes, d2.ano);

    return 0;
}
```

No código 4.7 criamos uma estrutura chamada struct data que contém variáveis para representar dia, mês e ano. Ao executar o código 4.7 vemos o resultado da figura 4.4.

Figura 4.4 – Estrutura data.

A cópia entre as duas estruturas *data* ocorreu na linha 18.

Mas, perceba que o mês é um caractere único o que dificultará a compreensão nos meses com o mesmo caractere. Assim, vamos reformular o código 4.7 para o código 4.8.

Código 4.8

```
#include <stdio.h>
#include <string.h>
struct data
{
    int dia;
    char mes[12];
    int ano;
};

int main()
{
    struct data d1, d2;

    d1.dia = 3;
    strcpy(d1.mes, "dezembro");
    //d1.mes = "dezembro"; //erro de compilacao
    d1.ano = 2015;

    /* Passando valor de e1 para e2*/
    d2 = d1;

    printf("Data 1:  %d / %s / %d", d1.dia, d1.mes, d1.ano);
    printf("\nData 2:  %d / %s / %d", d2.dia, d2.mes, d2.ano);
    printf("\n");
    return 0;
}
```

Na linha 14 não foi usado o operador de atribuição, pois trata-se de uma string. A operação de cópia de valores de strings, como já foi visto, não usa o operador de igualdade (=), mas sim a função *strcpy()*.

Nas linhas 22 a 25 nós também trocamos o %c do printf por %s para que função opere adequadamente.

Lembre-se que a atribuição "=" é ilegal com *arrays.* Tentar fazer isto com dois arrays causa um erro de compilação. Use o exemplo de código abaixo, forçando erro ao tentar atribuir um array ao outro.

```
    int a[5], b[5];

                /* Não irá compilar, um array não pode ser atribuido ao
outro usando “=” */
                a = b;
```

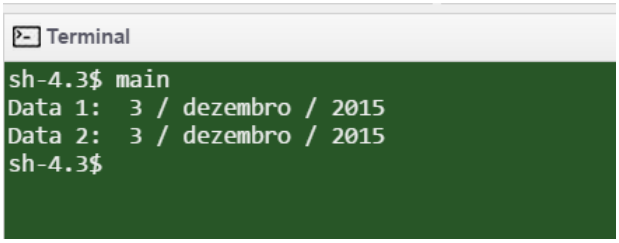


Figura 4.5 – Estruturas, cópias e vetores de caracteres (strings).

Se não forem atribuídos valores iniciais para as variáveis de uma estrutura é comum que seus membros possuam valores indefinidos. Como em outras variáveis, os campos das estruturas podem ser inicializados na própria declaração. Esta inicialização é parecida com a inicialização de arrays. O exemplo abaixo ilustra a inicialização de estruturas:

Código 4.9

```
#include <stdio.h>
#include <string.h>
struct data
{
    int dia;
    char mes[12];
    int ano;
};
```

```
int main()
{
    struct data d1 = {10, "agosto", 2016};
    struct data d2 = {11, "setembro", 2015};

    printf("Data 1:  %d / %s / %d", d1.dia, d1.mes, d1.ano);
    printf("\nData 2:  %d / %s / %d", d2.dia, d2.mes, d2.ano);

    return 0;
}
```

Podemos ver que uma lista de valores separados por vírgula fica entre chaves ({ e }) para definir os valores ao declarar uma variável. Devemos saber que os valores de inicialização devem estar na mesma ordem dos membros na declaração da estrutura. Na figura 4.6 podemos ver o resultado da execução do código 4.9.

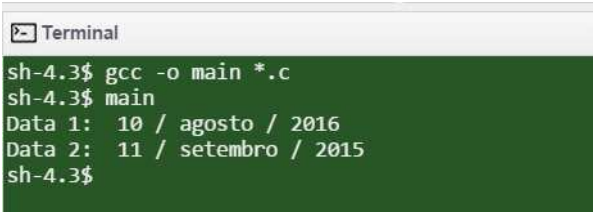


Figura 4.6 – Incializando estruturas.

4.3 Estruturas como Argumentos de Função e Valores de Retorno

Assim como qualquer outro tipo primitivo na linguagem C (*int, floa*t), valores de estruturas podem ser passados como argumentos para funções e podem ser retornados de funções.

No código 4.9 criamos uma função que fará a impressão de uma estrutura endereço criada para armazenar 4 variáveis vetores de char (*strings*). Também

criamos uma função fará a coleta dos dados do usuário e devolverá uma estrutura endereço pronta.

Como mostra a figura 4.7, ao executarmos o código 4.9 poderemos fazer a leitura e impressão dos valores através das funções criadas para manipulação de estruturas.

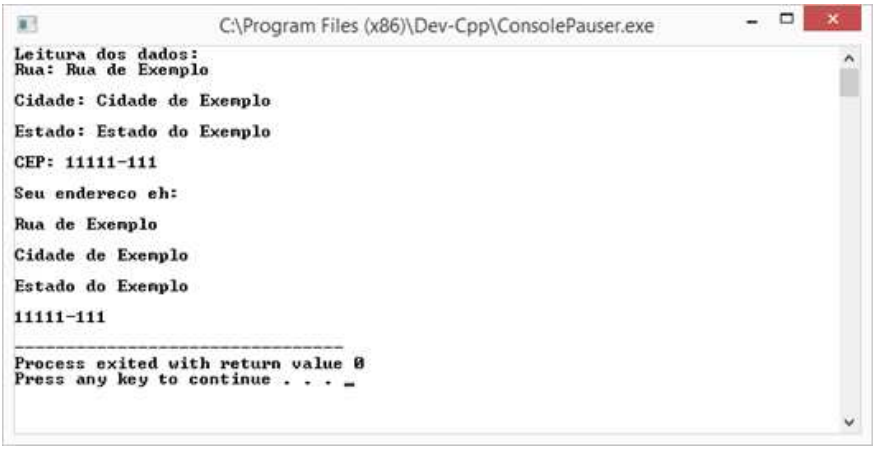


Figura 4.7 – Funções e Estruturas.

Código 4.9

```
#include <stdio.h>
#define TAMANHO 50

struct endereco
{
    char rua[TAMANHO];
    char cidade[TAMANHO];
    char estado[TAMANHO];
    char cep[TAMANHO];
};

struct endereco leitura(void);
void impressao(struct endereco);

int main()
{
```

```
    struct endereco end1;

    printf("Leitura dos dados:");
    end1 = leitura();

    printf("\nSeu endereco eh:\n");
    impressao(end1);

    return 0;
}

struct endereco leitura(void)
{
    struct endereco _endereco;

    printf("\nRua: ");
    gets(_endereco.rua);
    printf("\nCidade: ");
    gets(_endereco.cidade);
    printf("\nEstado: ");
    gets(_endereco.estado);
    printf("\nCEP: ");
    gets(_endereco.cep);

    return _endereco;
}

void impressao(struct endereco _endereco)
{
    printf("\n%s\n", _endereco.rua);
    printf("\n%s\n", _endereco.cidade);
    printf("\n%s\n", _endereco.estado);
    printf("\n%s\n", _endereco.cep);
}
```

No código 4.9, a estrutura "endereco" tem 4 arrays com tamanho máximo 50. Na função é declarada uma variável do tipo endereço para podermos pegar a entrada de dados do usuário (linha 30). Usamos a função gets() para leitura das informações.

Após o preenchimento dos dados retornamos a variável criada com os valores escolhidos pelo usuário (linha 41). Esta variável será copiada na variável da linha 20 na função *main()*. Logo após o retorno da variável, nós a passamos para a função *impressao()*, onde o valor de cada membro da estrutura será exibido na tela.

Estruturas podem ser passadas por valor ou por referência. No exemplo que vimos, a função *impressão()* recebe uma variável estrutura e a função *leitura()* retorna uma variável estrutura pelo retorno da pilha de execução.

No entanto, seria possível passar a estrutura para a função *leitura()* por referência e não necessitar do parâmetro de retorno. Antes de fazermos esta alteração veremos como lidar com estruturas e ponteiros.

4.4 Ponteiro para Estruturas

Como sabemos, podemos declarar ponteiros para tipos de diferentes de variáveis. É possível, também, declarar um ponteiro para uma variável do tipo *struct.* Uma variável do tipo ponteiro de estrutura armazena um endereço, assim, podemos acessar os campos dessa estrutura através de seu ponteiro, como é feito abaixo:

Código 4.10

```
struct exemplo
{
    int n;
    char c;
};

struct exemplo *p;

(*p).n = 12.0;
```

No código 4.10 o acesso à variável “n”, interna à estrutura exemplo, é feito com o operador “*”, assim como vimos anteriormente. A declaração de ponteiros para estrutura, como visto na linha 7, é feita também de maneira parecida ao que já vimos, adicionando-se o “*” na frente da variável.

Um ponto muito importante, que vou grifar para você é o seguinte:

Para acessar um elemento de uma estrutura por um ponteiro este ponteiro deve antes ter recebido a devida referência à estrutura. Não basta apenas declarar um ponteiro do tipo estrutura, pois ele não conterá uma região de memória para armazenar uma estrutura, mas sim para armazenar o endereço de memória de uma estrutura.

Vejamos o código 4.11.

Código 4.11

```
#include <stdio.h>

struct exemplo
{
    int n;
    char c;
};

int main()
{
    struct exemplo e1;
    struct exemplo *pE1;

    pE1 = &e1;

    (*pE1).n = 3;
    (*pE1).c = 'M';

    printf("Dados:  %d, %c\n", (*pE1).n , (*pE1).c );
    printf("Dados:  %d, %c\n", e1.n , e1.c );

    return 0;
}
```

Na linha 14 o ponteiro pE1 recebe o endereço da variável estrutura e1. Assim, é possível acessar seus elementos nas linhas 16 e 17.

O resultado na figura 4.8 mostra que o programa executou como esperávamos.

Figura 4.8 – Ponteiros para Estruturas.

O que aconteceria se a linha 14 não existisse no código 4.11? Um erro de compilação ou um erro de execução? Comente a linha 14 e faça a compilação (e talvez execução) do seu programa.

Nas linhas 16 e 17 o acesso à memória foi feito usando o parênteses também. Isto ocorre, pois o operador ponto tem precedência sobre o operador asterisco (*). Assim, se você remover os parênteses e tentar compilar ocorrerá o seguinte:

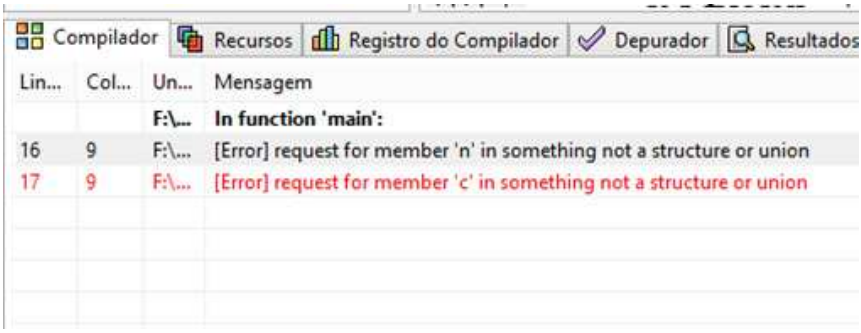


Figura 4.9 – Precedência do Operador . sobre *.

Na figura 4.9 é possível perceber o erro de compilação quando não usamos o parênteses no acesso aos membros da estrutura. Na tentativa de resolver primeiro o operador ponto o compilador tentar acessar o membro n, do ponteiro, que ainda não existe, pois primeiro você deve obter a estrutura apontada pelo ponteiro, ou seja, seu valor (usando o *).

A linguagem C também oferece um operador de acesso especial para um membro de uma estrutura quando usamos ponteiros. Assim podemos dispensar o uso dos parênteses e simplificar um pouco a codificação.

O operador especial é composto por um traço seguido de um sinal de maior (->), formando uma seta. Agora que sabemos como acessar de modo mais fácil um ponteiro para uma estrutura vamos reescrever as atribuições do código 4.11.

Código 4.12

```
#include <stdio.h>

struct exemplo
{
    int n;
    char c;
};

int main()
{
    struct exemplo e1;
    struct exemplo *pE1;

    pE1 = &e1;

    pE1->n = 3;
    pE1->c = 'M';

    printf("Dados:  %d, %c\n", pE1->n , pE1->c );
    printf("Dados:  %d, %c\n", e1.n , e1.c );

    return 0;
}
```

Veja as diferenças nas linhas 16, 17, 19 e 20. Ficou mais simples, não?

Vou usar a figura 4.10 para uma anotação importante:

- O operador seta (->) é para ponteiros de estruturas e não para estruturas. Para estruturas continua valendo o operador ponto (.) que vimos, ok?

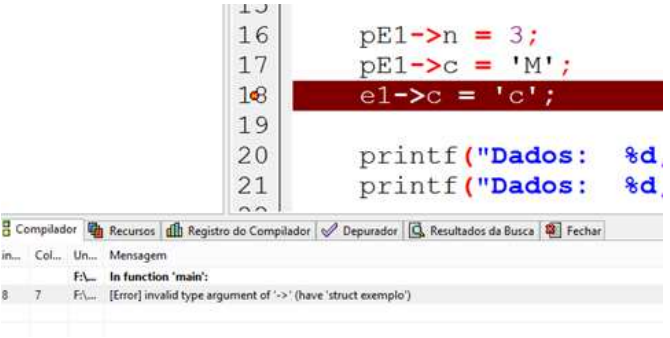


Figura 4.10 – Operador . versus operador ->.

4.5 Arrays (Vetores) de Estruturas

Como já estudamos, existem arrays de variáveis comuns como inteiros ou caracteres. Também podemos criar arrays de estruturas da mesma forma. Para estudar o uso de arrays com estruturas veremos o código 4.13 a seguir. Para melhorar a compreensão desta vez dividiremos o código em 3 partes.

Iniciando pela parte A, temos a declaração da estrutura que armazenará os dados de um aluno. Veja que ela conterá o código (linha 8), o nome do aluno (linha 9) e as notas deste aluno (linha 10). Perceba também que as notas de um aluno são representadas por um vetor. Assim, temos mais flexibilidade se quisermos mudar a quantidade de notas, apenas alterando o valor de MAX_NOTAS.

Código 4.13 (A)

```
#include <stdio.h>
#define MAX_CHAR 20
#define MAX_NOTAS 2
#define MAX_VETOR 2

struct aluno
{
    int codigo;
    char nome[MAX_CHAR];
    float notas[MAX_NOTAS];
};
```

```
void lerAluno(struct aluno [] , int);
void imprimirAluno(struct aluno);
```

Nas linhas 13 e 14 temos os protótipos das funções para leitura de um aluno específica e para impressão de um aluno. Veja que a leitura de um aluno recebe um vetor do tipo *struct aluno* como parâmetro, além de um número inteiro (o índice do vetor a ser lido).

Na sequência, no código 4.13B temos a implementação das funções prototipadas na parte A do código. A leitura de um aluno usa a variável *index* para alterar os dados deste aluno na posição indicada no vetor. Importante citar que vetores são sempre passados por referência, como vimos, e, por isto, conseguimos alterar este vetor de alunos dentor da função *lerAluno()*.

Nota que usamos uma função especial chamada *fflus()*. A função recebeu como parâmetro *stdin* que a entrada padrão (teclado) do sistema. O uso desta função é necessário devido a natureza do funcionamento de *scanf()* e *gets()* que, quando usados juntos, provocam resultados indesejados.

CONEXÃO

Para mais detalhes sobre isto consulte:

- http://www.cprogressivo.net/2012/12/Buffer--o-que-e-como-limpar-e-as-funcoes-fflush-e-fpurge.html Acesso em março de 2015.

Código 4.13 (B)

```
void lerAluno(struct aluno _alunos[], int index )
{
    int i;
    printf("\nCodigo: ");
    scanf("%d", &_alunos[index].codigo);
    fflush(stdin);
    printf("\nNome: ");
    gets(_alunos[index].nome);
    for (i = 0; i < MAX_NOTAS; i++)
    {
```

```
        printf("\nNota[%d]: ", i);
        scanf("%f", &_alunos[index].notas[i]);
        fflush(stdin);
    }
}

void imprimirAluno(struct aluno _aluno)
{
    int i;
    printf("\nCodigo: ");
    printf("%d", _aluno.codigo);
    printf("\nNome: ");
    printf("%s", _aluno.nome);
    for (i = 0; i < MAX_NOTAS; i++)
    {
        printf("\nNota[%d]: ", i);
        printf("%f", _aluno.notas[i]);
    }
}
```

Veja na função de leitura, linhas 5, 8 e 12, que a leitura usa a posição passada pelo índice *(index*). O acesso ao vetor é feito da mesma forma como em vetores com tipos primitivos. Só devem ser tomados os devidos cuidados no acesso às variáveis internas da estrutura. Por exemplo, veja que na linha 5, temos:

```
scanf("%d", &_alunos[index].codigo);
```

Assim, será acesso o endereço (&) da variável código que está dentro da estrutura aluno que está na posição apontada por *index* no vetor de alunos (_alunos).

Como pode ser visto na figura 4.11, a estrutura aluno é o elemento homogêneo do vetor. Então, é preciso cuidado ao acessar suas variáveis internas.

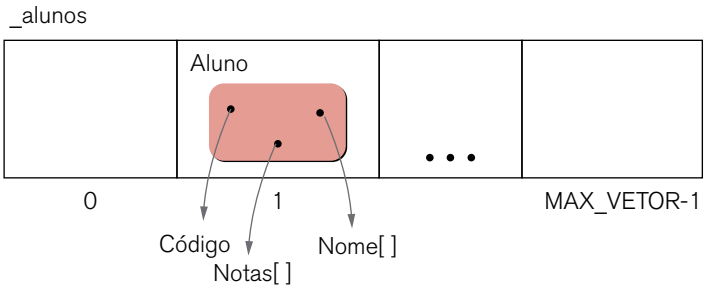


Figura 4.11 – Vetor de Estruturas na Memória.

Por fim, veja no código 4.13B que, nas linhas 9 a 14, tornamos a entrada de notas dinâmicas, ou seja, dependem de MAX_NOTAS. Se mudarmos a constante alteramos facilmente o funcionamento do programa.

Já a função *imprimirAlunos()* é bem simples. Recebe apenas uma variável por valor do tipo struct aluno e imprime os dados na tela.

Código 4.13C

```
int main()
{
    struct aluno lista[MAX_VETOR];
    int i;
    printf("<<<<< LEITURA >>>>>\n");
    for (i = 0; i < MAX_VETOR; i++)
    {
        printf("\nAluno[%d] - dados:\n", i);
        lerAluno(lista, i);
        imprimirAluno(lista[i]);
    }

    printf("\n<<<<< IMPRESSAO >>>>>\n");
    for (i = 0; i < MAX_VETOR; i++)
    {
        printf("\nAluno[%d] - dados:\n", i);
        imprimirAluno(lista[i]);
    }

    return 0;
}
```

O código 4.13C usa as funções que declaramos. Se quiser testar com mais valores de alunos basta mudar o valor de MAX_VETOR.

Minha sugestão é que você escreva todos estes códigos e os teste num programa novinho!

4.6 Estruturas Aninhadas

Assim como é possível termos vetores dentro de estruturas e vetores de estruturas, também podemos ter estruturas dentro de estruturas.

Estruturas internas a outras estruturas são chamadas de aninhadas. Para demostrar as estruturas aninhadas, vejamos o código 4.14. Neste código, a estrutura funcionário contém duas variáveis do tipo da estrutura endereço.

Código 4.14

```
#define TAMANHO 50

struct endereco
{
    char rua[TAMANHO];
    char cidade[TAMANHO];
    char estado[TAMANHO];
    char cep[TAMANHO];
};

struct funcionario
{
    char id[10];
    int idade;
    struct endereco casa;
    struct endereco empresa;
};

struct funcionario pessoa;
```

O acesso às estruturas aninhadas segue as mesmas regras vistas. Por exemplo, o acesso das variáveis de um funcionário será feito da seguinte forma:

```
pessoa.id
pessoa.casa.rua
pessoa.empresa.rua
```

O operador ponto é que nos permitirá navegar internamente nestas estruturas aninhadas.

4.7 Estrutura de Tipos Enumerados

Estruturas de tipos enumerados, também chamadas de ENUM, representam um conjunto de valores inteiros de identificadores associados a conteúdos constantes:

```
enum<nome> {<constante1>, <constante2>, … , <constanteN>,}
```

Um exemplo do uso de enum é o armazenamento dos meses do ano:

```
enum Meses {Janeiro = 1, Fevereiro, Marco, Abril, Maio, Junho,Julho,
Agosto, Setembro, Outubro, Novembro, Dezembro} meses;
```

Os valores de índices das constantes de um enum são crescentes em relação ao primeiro valor atribuído. Se você não adicionar nada, por padrão o primeiro valor será zero (0) e os demais também seguirão em ordem crescente.

No código 4.15 nós criamos um enum MESES contendo os meses do ano e iniciamos janeiro com 1. A variável meses é do tipo MESES!

Agora, veja que, a partir da linha 13, nós usamos a variável meses com suas constantes para identificar o valor digitado.

Código 4.15

```
#include <stdio.h>
//definicao da enum
enum Meses {Janeiro=1, Fevereiro, Marco, Abril, Maio,
Junho,
     Julho, Agosto, Setembro, Outubro, Novembro,
Dezembro}meses;

int main(void)
{
  printf("Digite o numero do mes: ");
  scanf("%d",&meses);

  switch(meses)
  {
      case Janeiro:
           printf("%d - Janeiro",meses);
      break;

      case Fevereiro:
           printf("%d - Fevereiro",meses);
      break;

      case Marco:
           printf("%d - Marco",meses);
      break;

      case Abril:
           printf("%d - Abril",meses);
      break;

      case Maio:
           printf("%d - Maio",meses);
      break;
```

```
        case Junho:
             printf("%d - Junho",meses);
        break;

        case Julho:
             printf("%d - Julho",meses);
        break;

        case Agosto:
             printf("%d - Agosto",meses);
        break;

        case Setembro:
             printf("%d - Setembro",meses);
        break;

        case Outubro:
             printf("%d - Outubro",meses);
        break;

        case Novembro:
             printf("%d - Novembro",meses);
        break;

        case Dezembro:
             printf("%d - Dezembro",meses);
        break;

        default:
         printf("<Valor INVALIDO!!!>\n");
         printf("Valor deve ser entre 1 e 12\n\n");
    }
  return 0;
}
```

4.8 Matrizes

Nós já estudamos o conceito de vetores e tipos homogêneos. Um vetor de 10 variáveis inteiras, por exemplo, é declarado da seguinte forma:

```
int v[10];
```

Também podemos criar uma constante para o tamanho do vetor:

```
#define MAX 100
int v[MAX];
```

Vetores são matrizes unidimensionais. Quando o elemento de um vetor (ou matriz unidimensional) for um vetor, diremos que trata-se de uma matriz bidimensional. Só precisamos declará-lo da seguinte forma:

```
tipo nome[LINHAS][COLUNAS];
```

Por exemplo:

```
int mat[3][4];
```

No exemplo acima, conceitualmente nós teríamos uma matriz com 3 linhas (0 a 2) e 4 colunas (0 a 3).

Dizemos conceitualmente, pois a memória RAM é um vetor de posições com palavras de memória, ou seja, ela é unidimensional. Assim, qualquer matriz, de qualquer dimensão, será armazenada linearmente na memória RAM.

Para nós desenvolvedores basta pensar conceitualmente e acessar as posições da seguinte forma:

mat[0][0]	mat[0][1]	mat[0][2]	mat[0][4]
mat[1][0]	mat[1][1]	mat[1][2]	mat[1][4]
mat[2][0]	mat[2][1]	mat[2][2]	mat[2][4]

Tanto a atribuição quanto a leitura de valores de matrizes seguem a mesma ideia do que vimos em vetores:

```
float Media[3][3]; //declaração de matriz 3x3
Media[0][1] = 5; //atribuição simples
scanf(“%f”, &Media[0][0]); //leitura de um elemento
printf(“%f”, Media[1][2]); //impressão de elemento
```

Uma forma simples, prática e intuitiva para ler ou imprimir todos os elementos de uma matriz é usar dois laços aninhados. Veja o código 4.16.

Código 4.16

```
for ( i=0; i<2; i++ ){
    for ( j=0; j<2; j++ ){
printf ("%d", matriz[ i ][ j ]);
}
}
```

Neste trecho de código a variável i representa a linha e j representa a coluna da matriz. Usando o laço for podemos percorrer cada posição dentro da matriz e receber os valores usando a função *scanf()*, por exemplo.

Para mostrar os elementos de uma matriz podemos usar o mesmo algoritmo porém com o comando *printf()*.

Vamos fazer um exemplo de um programa em C que preenche uma matriz com 2 linhas X 2 colunas e depois mostra os dados recebidos.

O exemplo no código 4.17 lê os valores para uma matriz 2x2 e depois imprime-os na tela.

Código 4.17

```
#include<stdio.h>
#define L 2
#define C 2

int main (void )
{
    int matriz[L][C],i, j;

    printf ("\nEntre com os dados\n\n");
```

```
    for ( i=0; i<L; i++ ){
            for ( j=0; j<C; j++ )
            {
                    printf ("\nMatriz[%d][%d] = ", i, j);
                    scanf ("%d", &matriz[i][j]);
            }
    }
    for ( i=0; i<L; i++ ){
            printf("\n");
            for ( j=0; j<C; j++ )
            {
                    printf ("\t%d", matriz[i][j]);
            }
    }

    getch();
    return(0);
}
```

4.9 Matrizes e Funções

Matrizes são muito comuns em matemática, particularmente em álgebra. O traço de uma matriz quadrada, em Álgebra, é a soma dos elementos de sua diagonal principal. Assim, vamos criar uma função chamada *traco()* que recebe uma matriz quadrada e devolve o valor calculado da soma dos elementos de sua diagonal principal.

CONEXÃO

Mais informações sobre traço de uma matriz (acesso em março de 2015):
https://pt.wikipedia.org/wiki/Tra%C3%A7o_(%C3%A1lgebra_linear)

Para isto, antes, precisamos entender como uma matriz é passada para uma função.

Para podermos operar valores recebidos em funções como matrizes do tipo

```
pessoas[2][1]
```

precisamos passar para uma função o tamanho da segunda dimensão da matriz. Assim, nossa função terá o protótipo:

```
int traco(int mat[][LIMITE], int n);
```

O valor LIMITE será uma constante definida na diretiva *#define* para delimitar o tamanho das matrizes quadradas.

Nossa função *traco()* definida no código 4.18A receberá a matriz quadrada *mat[][]* e o tamanho n que indica os limites de *mat*. A implementação da função, basicamente, percorre todo seu espaço de memória somando os elementos que tiverem o mesmo índice de linha e coluna (linha 19), ou seja, elementos da diagonal principal.

Código 4.18A

```
#include<stdio.h>
#define LIMITE 10

void imprime(int mat[][LIMITE], int n);
int traco(int mat[][LIMITE], int n);

int main(void) {

    ...

    return 0;
}

int traco(int mat[][LIMITE], int n) {
    int soma = 0, i, j;
    for(i = 0; i < n; i++){
          for(j = 0; j < n; j++){
                if(i == j){

                      soma = soma + mat[i][j];
                }
          }
    }
    return soma;
}
```

Depois, basta retornar o valor somado na variável soma (linha 23).

Código 4.18B

```
void imprime(int matriz[][LIMITE], int n) {
    int i, j;
    for ( i=0; i<n; i++ ){
           printf("\n");
           for ( j=0; j<n; j++ )
           {
                  printf ("\t%d", matriz[i][j]);
           }
    }
}
```

Para facilitar a visualização dos resultados nós criamos também a função *imprime()* (código 4.18B) que recebe a matriz e faz a impressão tabulada da mesma, simulando sua disposição lógica dos dados.

Por fim, no código 4.18C apresentamos o conteúdo da função *main()* para testar as funções criadas!

Código 4.18C

```
...
int main(void) {

    int n;
    int i, j;
    int soma = 0;
    int mat[LIMITE][LIMITE];
    do {
        printf("N: ");
       scanf("%d", &n);
    }
    while((n <= 0) || (n > LIMITE));

    printf("\nValores da Matriz\n");
    for(i = 0; i < n; i++){
          for(j = 0; j < n; j++){
                printf("\nMat[%d][%d]: ", i, j);
```

```
                scanf("%d", &mat[i][j]);
            }
        }
        imprime(mat, n);
        printf("\nTraco da Matriz = %d", traco(mat, n));

        return 0;
    }
```

Por fim, para deixarmos o código ainda mais elegante e enxuto sugiro a criação de uma função para leitura dos dados da matriz e alteração do código existente na função main().

Código 4.18D

```
...
void leitura(int mat[][LIMITE], int n);
...
int main(void) {

...
//O for das linhas 15 a 20 no código 4.18C
//foi troca apenas pela linha abaixo

    leitura(mat, n);
    imprime(mat, n);
    printf("\nTraco da Matriz = %d", traco(mat, n));

    return 0;
}

int leitura(int mat[][LIMITE], int n) {
    int i, j;
    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            printf("\nMat[%d][%d]: ", i, j);
            scanf("%d", &mat[i][j]);
        }
    }
}
```

Vamos a outro exemplo de aplicação de matrizes. Criaremos uma função que calcula e devolve a Matriz Transposta Mt de uma Matriz M. Matriz transposta é obtida pela troca de linhas e colunas de uma dada matriz.

Por exemplo, imagine a matriz abaixo:

```
[a1, a2, a3]
[b1, b2, b3]
```

Sua matriz transposta seria:

```
[a1, b1]
[a2, b2]
[a3, b3]
```

Agora vamos pegar do exemplo acima de construir um código onde o usuário informa, o número de linhas e coluna, e logo depois este programa vai informar a matriz original e sua matriz transposta. Como mostra a figura 4.12, o usuário fará a entrada dos dados.

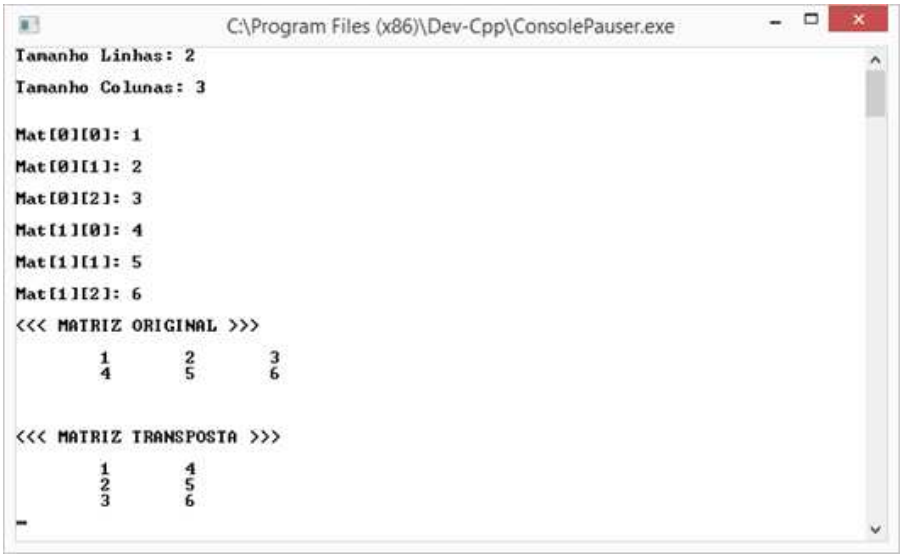


Figura 4.12 – Transposta de uma matriz.

No código 4.19A declaramos os protótipos (linhas 4 a 7) das funções que serão importantes para nós e temos também o conteúdo do código *main()*, responsável pela lógica principal do programa. Atente-se ao protótipo da função *transposta()*. Veja que esta função recebe duas matrizes. Nós devolveremos a variável Mtransp (linha 4) como sendo a matriz transposta da variável matriz.

Código 4.19A

```
#include<stdio.h>
#define MAX 10

void transposta(int matriz[][MAX], int Mtransp[][MAX], int l, int
c);
void imprime(int matriz[][MAX], int l, int c);
int leitura(int mat[][MAX], int l, int c);

int main(void) {
    int l = 0, c = 0;
    int i, j;
    int mat[MAX][MAX], matTrans[MAX][MAX];

    do {
        printf("Tamanho Linhas: ");
        scanf("%d", &l);
    } while((l <= 0) || (l > MAX ));
    printf("\n");

    do {
        printf("Tamanho Colunas: ");
        scanf("%d", &c);
    } while((c <= 0) || (c > MAX ));
    printf("\n");

    leitura(mat, l, c);
    printf("\n<<< MATRIZ ORIGINAL >>>\n");
    imprime(mat, l, c);
    //obtendo M_transposta
```

```
    transposta(mat, matTrans, l, c);
    printf("\n<<< MATRIZ TRANSPOSTA >>>\n");
    imprime(matTrans, c, l);

    return 0;
}
```

No código 4.19B temos a função que faz a transposição da matriz recebida na função. São passadas também as variáveis l e c para informar os limites das matrizes. Veja, na linha 8, que a transposição é feita, basicamente, percorrendo-se a matriz inicial e atribuindo na matriz transposta os valores da matriz inicial de forma invertida em relação às linhas e colunas.

```
Mtransp[j][i] = matriz[i][j];
```

Na instrução, os valores que representam linha (i) e coluna (j) da matriz inicial (matriz) são colocados como linha (j) e coluna (i) da matriz transposta (Mtransp).

Código 4.19B

```
...
void transposta(int matriz[][MAX], int Mtransp[][MAX], int l, int
c) {
    int i, j;
    for(i = 0; i < l; i++){
        printf("\n");
        for(j = 0; j < c; j++){
            Mtransp[j][i] = matriz[i][j];
        }
    }
}
```

Observe no código 4.19C que fizemos alguns ajustes nas nossas funções imprime e leitura, pois agora as matrizes não são necessariamente quadradas.

Código 4.19C

```
void imprime(int matriz[][MAX], int l, int c) {
    int i, j;
    for ( i=0; i<l; i++ ){
           printf("\n");
           for ( j=0; j<c; j++ )
           {
                  printf ("\t%d", matriz[i][j]);
           }
    }
    printf("\n");
}

int leitura(int mat[][MAX], int l, int c) {
    int i, j;
    for(i = 0; i < l; i++){
          for(j = 0; j < c; j++){
                printf("\nMat[%d][%d]: ", i, j);
                scanf("%d", &mat[i][j]);
          }
    }
}
```

Assim, é preciso passar para as funções os limites de linha e coluna nas variáveis *l* e *c.*

REFLEXÃO

Nesse capítulo nós vimos o conceito de estruturas heterogêneas que nos permite criar tipos complexos de dados nas nossas aplicações. Aprendemos como cria-las, passa-la para funções como parâmetros, acessar seus dados por valor e por referência.

Também vimos como criar vetores dessas estruturas.

Aprenderemos no capítulo 5 como persistir dados, uma importante forma de tornar mais robustas e interessantes nossas aplicações.

Os conceitos de matrizes nos permitiram utilizar dados em mais de uma dimensão (mesmo que logicamente e não fisicamente).

Agora deixo para você a missão de testar todos os códigos e divertir-se, ok?
Bom trabalho!

LEITURA

Para complementar seu aprendizado computação, programação e linguagem C sugiro os seguintes links em inglês:

- http://www.cprogramming.com/ Acesso em março de 2015.
- Multiplicação e Adição de Matrizes (Acesso em março de 2015):
- http://www.programmingsimplified.com/c-program-multiply-matrices
- http://www.programmingsimplified.com/c-program-add-matrices

Também sugiro a leitura de livros em português como:

- MIZRAHI V. V.: Treinamento em Linguagem C. São Paulo: Pearson Prentice Hall, 2008.

E também sugiro os seguintes links em português Matrizes e vetores em C (Acesso em março de 2015):

- http://www.cprogressivo.net/2013/03/Vetores-multidimensionais-Matrizes-em-C--vetor-de-vetores.html

REFERÊNCIAS BIBLIOGRÁFICAS

KERNIGHAN, B. W.; RITCHIE, D. M. C: **a linguagem de programação,** padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.

KELLEY, A.; POHL, I. A **Book on C: Programming in C.** 4. ed. Boston: Addison Wesley, 1997. 726p.

ZIVIANI, N. Projetos de algoritmos: com implementações em Pascal e C. São Paulo: Pioneira Thomson, 2002. 267p.

ARAÚJO, J. **Dominando a Linguagem C.** 1. ed. Ciência Moderna, 2004, 146p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos:** teoria e prática. Rio de Janeiro: Campus, 2002. 916p.

FORBELLONE, A. L. **Lógica de Programação**. 2. ed. São Paulo: Makron Books, 2000. 195p.

KNUTH, D. E. T**he Art of Computer Programming,** Fundamental Algorithms. 3. ed. Boston: Addison Wesley, 1997. 672p. (vol 1)

SEBESTA, R. W. **Conceitos de Linguagens de Programação.** 4. ed. Porto Alegre: Bookman, 2000. 624p.

5

Manipulação de Arquivos, Biblioteca de Funções e Definições de Tipos

Neste capítulo nós veremos como manter os dados de nossos programas após encerrá-los, após desligar o computador ou numa queda de energia. Afinal, não dá para contarmos apenas com os programas executando em memória principal.

Também veremos como podemos criar tipos com uma instrução especial. Com isto você ganhará um novo recurso para aprimorar seu aprendizado do capítulo 4.

OBJETIVOS

Neste capítulo, os objetivos principais são:

- Persistência de dados.
- Manipulação de arquivos.
- Modo texto e modo binário.
- O uso de typedef.
- Bibliotecas de funções.
- Aplicações diversas.

5.1 Persistindo Dados com Arquivos

Variáveis, vetores, matrizes e estruturas são todos regiões da memória RAM. Quando um computador ou dispositivo é desligado ou os programas são finalizados por algum motivo, perdemos as informações, pois a memória RAM é volátil.

Uma forma de não perdemos as informações é usar os chamados arquivos. Arquivos, em linguagem C, são mais amplos do que a definição de dados em disco, ou memória secundária. A palavra arquivo é usada para indicar *streams* ou fluxos de bytes, assim, podemos nos referir a arquivos em disco, teclado, vídeo, impressora e portas de comunicação como sendo arquivos.

Em linguagem C é possível trabalhar com o acesso a arquivos em alto-nível ou em baixo-nível. As funções para manipulação de arquivos em alto-nível são *bufferizadas*, ou seja, mantém uma região de memória com os dados antes de enviá-los para o destino. Já as funções de baixo nível não são bufferizadas. Nosso escopo compreende apenas o acesso de alto nível, mas, se tiver interesse em conhecer o acesso de baixo nível consulte as leituras recomendadas e bibliografias.

Arquivos são de grande importância e uma das suas principais vantagens é a facilidade para utilizar os dados armazenados em qualquer momento, em vários programas diferentes. Importante saber que quando um computador é desligado, não perdemos os arquivos salvo na memória secundaria, o disco.

Outra vantagem dos arquivos é a maior capacidade para manipulação de dados do que vetores e matrizes oferecem.

Quando falamos em utilização e manipulação de arquivos estamos falando de criação e abertura do arquivo, gravação e leitura de dados no arquivo e fechamento do arquivo.

O acesso a um arquivo começa pela verificação de sua existência no disco.

Um arquivo pode ser acessado de forma sequencial ou aleatória:

Acesso Sequencial:

- É feito de forma contínua.
- Dados inseridos um após o outro a partir do primeiro registro.
- Registro a registro, até localizar a primeira posição vazia após o último registro.

• Para ler um registro primeiro será necessário ler todos os registros que o antecedem.
• Processo lento.

Acesso Aleatório:

• Ocorre por meio da transferência de dados diretamente para qualquer posição do arquivo.
• Não é preciso ler as informações anteriores.
• 3 formas diferentes com relação ao posicionamento do ponteiro dentro do arquivo.
• Início do arquivo.
• Fim do arquivo.
• Posição atual do ponteiro no arquivo.

Em linguagem C, os dados podem ser transferidos na sua representação binária interna ou num formato de texto legível ao usuário.

A linguagem C oferece várias funções específicas de leitura e de escrita de dados num arquivo. Se o arquivo for do tipo texto, você deve utilizar funções específicas para arquivos texto. Se ele for binário (do tipo que armazena registros ou estruturas), você deve utilizar as funções de leitura e/ou escrita em arquivos binários.

Na linguagem C as operações de entrada e saída, incluindo as relacionadas a arquivos, encontram-se na biblioteca stdio.h. Essa biblioteca é importante porque também define macros, dentre elas NULL e EOF, que representam um ponteiro nulo e o fim de arquivo, respectivamente. Além disso, nela está definido o tipo FILE que vamos usar para armazenar um arquivo em memória RAM.

Se você examinar stdio.h encontrará algo semelhante ao conteúdo da figura 5.1.

```
136 #ifndef _FILE_DEFINED
137 #define _FILE_DEFINED
138 typedef struct _iobuf
139 {
140     char*   _ptr;
141     int _cnt;
142     char*   _base;
143     int _flag;
144     int _file;
145     int _charbuf;
146     int _bufsiz;
147     char*   _tmpfname;
148 } FILE;
149 #endif  /* Not _FILE_DEFINED */
```

Figura 5.1 – Trecho de stdio.h.

Você pode verificar que o tipo FILE é uma estrutura contendo informações descritivas do arquivo em questão.

Em Linguagem C nós trabalharemos com arquivos bufferizados através de *Streams*. Este formato nos fornecerá um bom nível de abstração, torna o acesso a arquivos independente do dispositivo real. As *streams* podem ser textuais ou binárias. Como cada dispositivo (disco, pen drives, cds, etc) é manipulado de forma diferente no sistema operacional, nós faremos o acesso aos arquivos pelas *streams:*

Stream Texto:

- É uma sequência de caracteres no padrão ANSI C.
- Pode ser organizada em linhas terminadas por um caractere de nova linha.
- O caractere de nova linha é opcional na última linha e é determinado pela implementação do compilador.
- Certas traduções podem ocorrer conforme exigido pelo sistema hospedeiro.
- Uma nova linha pode ser convertida em um par "retorno de carro/alimentação de linha".

Stream Binária

- É uma sequência de bytes.
- Há uma correspondência de um para um com aqueles encontrados no dispositivo externo.
- Não ocorre nenhuma tradução de caracteres.
- O número de bytes escritos/lidos é o mesmo que o encontrado no dispositivo externo.

5.2 Manipulação de Arquivos

Antes de começarmos a criar, abrir, manipular, salvar em arquivos, vamos ver algumas funções encontradas dentro da biblioteca na linguagem C. Abaixo, uma lista das principais funções da biblioteca de manipulação de arquivos.

- *fopen()* - Abre um arquivo.
- *fclose()* - Fecha o arquivo garantindo a transferência de *buffer*.
- *fflush()* - Descarrega o *buffer*.
- fscanf() - Leitura de entrada formatada.
- *fprintf()* - Escrita de saída formatada.
- *fgets()* - Obtém uma *string* do arquivo.
- *fgetc()* - Obtém um caractere no arquivo.
- *fputs()* - Insere uma string no arquivo.
- *fputc()* - Insere um caractere no arquivo.
- *fread()* - Lê um bloco de dados do arquivo.
- *fwrite()* - Escreve um bloco de dados no arquivo.
- *fseek()* - Reposiciona o ponteiro.
- *rewind()* - Reposiciona o ponteiro para o inicio do arquivo.
- *ftell()* - Retorna a posição do ponteiro.

Para facilitar o uso das diversas funções vamos agrupá-las da seguinte forma:

- Funções para leitura e escrita de um caractere por vez: ***fputc() e fgetc()***
- Funções para leitura e escrita linha a linha: ***fputs() e fgets()***

- Funções para leitura e escrita formatada: ***fprintf() e fscanf()***
- Funções para leitura e escrita de blocos de bytes: ***fwrite() e fread()***

Na linguagem C os arquivos não podem ser manipulados diretamente, é necessário que exista uma variável onde possamos referenciá-lo. Esta variável deve ser do tipo FILE*. Sempre é necessário criar uma variável deste tipo para poder utilizar o arquivo. Todas as funções de manipulação de arquivo necessitam de uma variável deste tipo para poder manipular o arquivo.

Vamos ver como é fácil declarar uma variável do tipo FILE.

```
FILE* arquivo; // cria uma variável que manipula arquivos.
```

Após declarar a variável que vai referenciar o arquivo, é possível usar a função *fopen()* para abrí-lo. A sintaxe para utilização do *fopen()*:

```
nome_variavel = fopen("nome_arquivo.extensão". "modo+tipo");
```

A "nome_variavel" é do tipo FILE que vai representar o arquivo dentro do programa, nome_arquivo é o arquivo que existe ou vai existir em memória secundaria, já o "modo" é como abrir, criar ou sobrescrever o arquivo, e o "tipo" é definido como texto ou binário. Nessa operação, é associada uma stream com um arquivo específico. Quando o arquivo é aberto, informações podem ser trocadas entre a stream e o programa.

Abaixo uma lista de tipos de modos de abertura de um arquivo:

- "r" – opção para abrir um arquivo em modo texto para leitura. O arquivo deve existir antes de ser aberto.
- "w" – opção para abrir um arquivo em modo texto para gravação. Neste caso, se o arquivo não existir, será criado, se já existir, o conteúdo anterior será destruído. Muito cuidado com esta opção!
- "a" – opção para abrir um arquivo em modo texto para gravação na qual os dados serão adicionados no fim do arquivo, se ele já existir, ou será criado um novo arquivo, no caso do arquivo ainda não existir.
- "rb" – opção para abrir um arquivo em modo binário para leitura, parecido com o modo "r", só que trata de um arquivo em binário.
- "wb" – opção para criar um arquivo em modo binário para escrita, como no modo "w" anterior, só que com arquivo binário.

• “ab” – opção para acrescentar dados binários no fim do arquivo, como no modo “a” anterior, só que o arquivo é binário.

• “r+” – opção para abrir um arquivo em modo texto para leitura e gravação. Se o arquivo existir seu conteúdo anterior será destruído, se não, um novo arquivo será criado.

• “a+” – opção para abrir um arquivo em modo texto para gravação e leitura. Se o arquivo existir os dados serão adicionados no final, senão um novo arquivo será criado.

• “r+b” – opção para abrir um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que com arquivo binário.

• “w+b” – opção para criar um arquivo em modo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário.

• “a+b” – opção para acrescentar dados ou criar um arquivo em modo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário.

Após a manipulação do arquivo é preciso usar uma função para fechá-lo:

```
fclose(nome_variavel);
```

Ao abrir um arquivo é preciso ter cuidado, pois a função de abertura (*fopen*) pode não retornar com o comportamento desejado. Um arquivo pode não ser aberto para gravação por falta de espaço em memória ou pode não ser aberto para leitura por ainda não existir.

Código 5.1

```
#include <stdio.h>
int main()
{
    FILE *arquivo;
    int idade = 23;

    arquivo = fopen("Arquivo.txt", "w+t");

    if (arquivo==NULL)
       printf ("Erro na abertura do arquivo.");
    else
    {
```

```
        printf("Arquivo aberto com sucesso.");
        fprintf(arquivo, "%d", idade);
        fprintf(arquivo, "\nJose tem %d anos.",
 idade);
    }

    fclose(arquivo);
    return 0;
}
```

Observe na linha 7 do código 5.1 que utilizamos o *fopen()*, passando como argumento de parâmetro o nome do arquivo e sua extensão, junto com o modo e tipo que foi criado. Como segundo parâmetro observamos o "w+t", isso significa que foi dada a instrução para criar um arquivo texto. Logo após executar o programa observa no diretório onde está salva a aplicação que um arquivo texto com o nome de Arquivo foi criado.

Caso a instrução *fopen()* não retorne o ponteiro para estrutura do arquivo criado então haverá NULL, linha 09, e isto nos permitirá tratar adequadamente a aplicação.

A função *fprintf()*, nas linhas 14 e 15, nos permite escrever dentro de um arquivo. A função pode ser definida assim:

```
fprintf("nome_do_arquivo", "texto", [variáveis]);
```

Note a grande semelhança no uso do *printf().* São quase idênticas. Sua única diferença é que você tem que informar no primeiro parâmetro o nome da variável que armazena o arquivo FILE, onde os dados vão ser gravados em um arquivo.

Podemos perceber, abrindo o documento "Arquivo.txt" que se encontra no diretório onde está salva a sua aplicação, que o conteúdo passado a *fprintf()* está salvo agora em um arquivo do tipo texto e este dado pode ser recuperado mesmo depois que a aplicação ou o sistema for desligado.

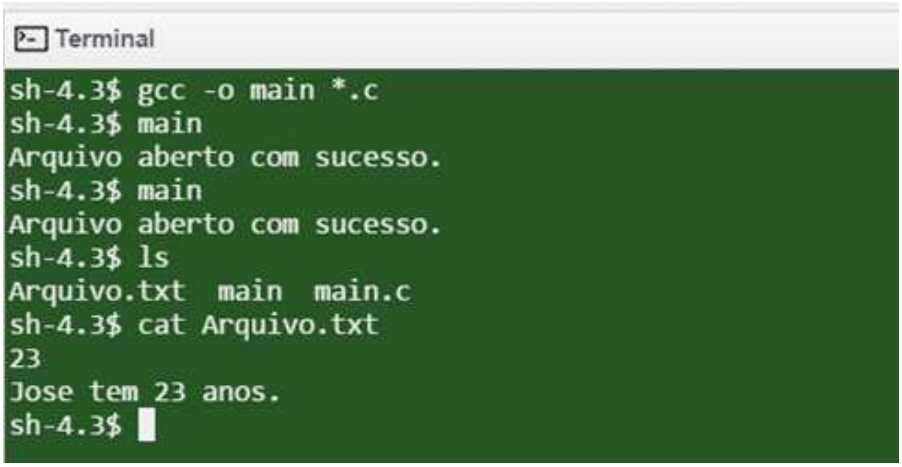


Figura 5.2 – Os dados de um arquivo gravado.

Veja na figura 5.2 que é possível checar os dados gravados no "Arquivo.txt". Certo, agora conseguimos gravar os dados num arquivo do tipo texto! Após gravar os dados, como vamos poder recuperá-los?

5.3 Arquivos Modo Texto

Para recuperar os dados de um arquivo tipo texto, usaremos uma função chamada fgetc(), que pega um caractere por vez dentro do arquivo.

Como não vamos mais precisar criar um arquivo novo, mas utilizaremos um arquivo já criado nos exemplos anteriores, o parâmetro de abertura do arquivo será "r+t", informando que vamos abrir um arquivo texto somente para leitura e que já existe no diretório do programa.

Código 5.2

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *arquivo;
    char ch;
    arquivo = fopen("Arquivo.txt", "r+t");
```

```
    if (arquivo==NULL) {
       printf ("Erro na abertura do arquivo.\n");
       exit(EXIT_FAILURE);
    }
    else
    {
        ch = fgetc(arquivo);
        while(ch != EOF)
        {
            printf("%c", ch);
            ch = fgetc(arquivo);
        }
    }
    fclose(arquivo);
    return 0;
}
```

Veja na figura 5.3 que a execução do nosso programa mostra exatamente o conteúdo gravado no código 5.1.

Sabemos que a função *fgetc()* faz a leitura de um caractere por vez dentro do arquivo (linha 13). Ao fazer a leitura, existe um ponteiro da própria função informando em qual posição atual do arquivo. Para facilitar a movimentação, este ponteiro só aponta ao começo do arquivo se usarmos outra função, a qual vamos explicar mais adiante. Para que possamos escrever toda informação que está no arquivo, usamos o *fgetc()* dentro de um loop (linha 15). Ele ficará buscando cada caractere até o caractere lido dentro do arquivo ser um valor especial que indica o final do arquivo, ou seja, EOF (linha 15).

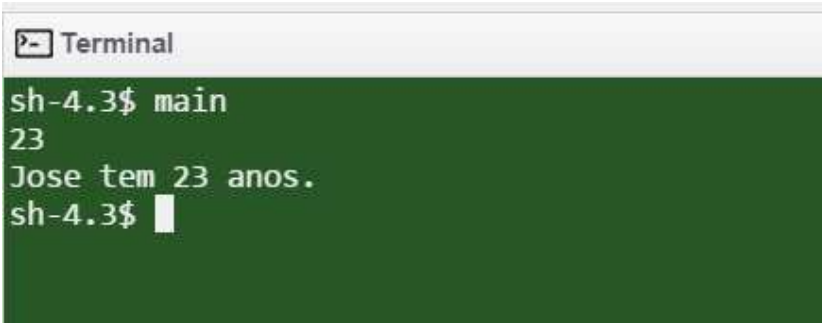


Figura 5.3 – Lendo e Imprimindo os dados de um arquivo gravado.

Como estudo, experimente excluir o arquivo criado (Arquivo.txt) e tente executar seu programa do código 5.2 novamente. Veja o resultado esperado:

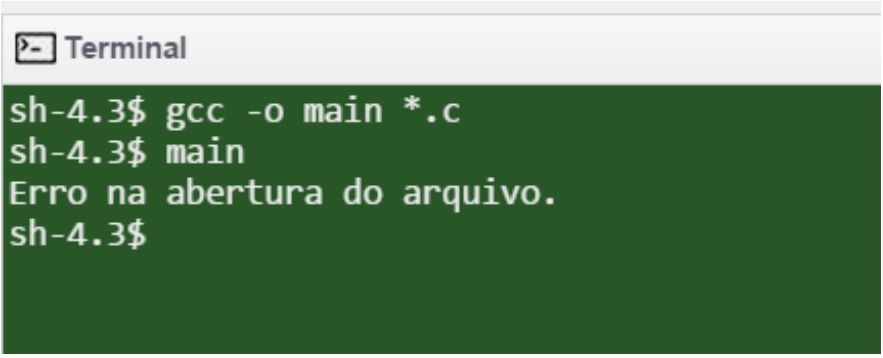


Figura 5.4 – Erro na abertura de um arquivo.

Isto ocorrerá, pois você está tentando abrir seu arquivo apenas para leitura e ele não existe!

Agora vamos criar um programa que terá uma estrutura simples de um cliente. Vamos registrar 3 clientes, depois salvaremos a informação em disco.

Antes de tudo devemos criar a estrutura do cliente, o código abaixo mostra como vai ser a estrutura que vamos utilizar em nosso programa:

Código 5.3

```
#define MAX 3
struct Cliente
{
    char nome[NOME];
    int codigo;
    int idade;
};
```

Após declarar uma estrutura "Cliente" simples, que contém o nome, código e idade, nosso programa deve conter uma função onde vamos poder inserir uma estrutura Cliente no arquivo texto. A implementação para esta função está no código 5.4:

Código 5.4

```
void gravarCliente(struct Cliente *c) {
    FILE *arquivo;

    arquivo = fopen("Cliente.txt", "a");

    if(arquivo == NULL)
        arquivo = fopen("Cliente.txt", "r+t");

    if(arquivo == NULL) {
        printf ("Erro na abertura do arquivo
Cliente.");
        exit(EXIT_FAILURE);
    }
    else
    {
        printf("Nome do Cliente:");
        scanf("%s", &c->nome);
        printf("Codigo do Cliente:");
        scanf("%d", &c->codigo);
        printf("Idade do Cliente:");
        scanf("%d", &c->idade);

        fprintf(arquivo, "%s %d %d ", c->nome, c-
>codigo, c->idade);

        printf("\nArquivos salvos com sucesso\n");
        fclose(arquivo);
    }
}
```

Está função é importante, pois tem o objetivo de salvar um cliente dentro de um arquivo texto. A estrutura cliente é passada como parâmetro da função (na verdade, um ponteiro para ela), na linha 1.

Na linha 4 tentamos abrir o arquivo para gravação adicionando os dados no final (parâmetro a). Se o arquivo ainda não existir usamos a linha 6 para criar um novo arquivo.

Dentro da função foi criada uma variável FILE quer será nossa referência para o arquivo. Depois de ter criado a variável vamos tentar ler ou criar um arquivo do tipo Cliente, se a criação ou leitura do arquivo for executada com sucesso, vamos começar a pegar informação digitada pelo usuário e a estrutura que for preenchida completa vamos salvar dentro do arquivo "Cliente" usando a função *fprintf()*, como podemos ver dentro da implementação da função *gravarCliente()*.

Após implementar a função para gravar uma estrutura "Cliente" dentro de um arquivo texto, vamos fazer uma função com o objetivo de imprimir toda informação que está salva dentro do arquivo texto, recuperando, assim, sua informação e estrutura. O código para implementar está função ficará assim:

Código 5.5

```
void lerClientes(struct Cliente *c) {
    FILE *arquivo;
    char ch;

    arquivo = fopen("Cliente.txt", "r");

    if(arquivo == NULL) {
        printf("Erro na abertura do arquivo
Cliente.\n");
        printf("Nao existe nenhum arquivo Cliente
ainda\n");
    }
    else
    {
        ch = fscanf(arquivo,"%s %d %d ", &c->nome,
 &c->codigo, &c->idade);
        int index = 0;
        while(ch != EOF)
        {
            index++;
```

```
            printf("A ordem no arquivo desse cliente
eh: %d", index);
            printf("\nO Nome do Cliente eh: %s", c-
>nome);
            printf("\nO Codigo do Cliente eh: %d", c-
>codigo);
            printf("\nA Idade do Cliente eh: %d", c-
>idade);
            printf("\n\n");
            ch = fscanf(arquivo,"%s %d %d ", &c-
>nome, &c->codigo, &c->idade);
        }
        fclose(arquivo);
    }
}
```

Como o próprio nome já diz, está função tem como objetivo ler um arquivo texto com o nome "Cliente". Como é uma função somente de leitura e não iremos escrever nada dentro do arquivo, vamos abrir o arquivo utilizando o modo "r", o qual indica que este arquivo pode somente ser lido. Depois que conseguimos abrir o arquivo o próximo passo será fazer a leitura das informações, e para este processo utilizaremos a função *fscanf()*, no qual tem objetivo de pegar as informações do arquivo de acordo com as informações passadas por parâmetro.

Vamos entender como funciona o *fscanf()*, para podermos entender melhor o código:

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

Como podemos ver, teremos que passar por parâmetro o ponteiro do arquivo o qual vai ser lido e um char, que seria a *string* formatada, mas como essa string funciona? É simples! Quando é necessário ler um arquivo é preciso de uma *string* formatada para que a informação possa ser lida corretamente. Suponha que exista um arquivo com os dados abaixo:

```
a b c
d f g
h i j
```

Como usar o fscanf para ler estas informações? Para poder ler um arquivo é preciso primeiramente saber qual o formato do arquivo. Neste exemplo é um formato simples, são 3 caracteres separados por um espaço e no ultimo uma quebra de linha, para passar este tipo de informação para o *fscanf()*, seria desta maneira "%c %c %c\n".

Para que possamos pegar a primeira linha do arquivo exemplo ficaria desta maneira:

```
char a, b, c;
// abrir arquivo …..
// usar função fscanf()
fscanf(nome_arquivo, "%c %c %c\n", &a, &b, &c);
```

A função *fscanf* no exemplo acima recuperou as informações de acordo com o formato passado, e colocou a informação dentro das variáveis em suas respectivas ordens. Dentro da nossa função *lerClientes(), o fscanf()* também tem sua string formatada, com o objetivo de buscar uma *string* e dois números inteiros. Continuando o funcionamento da função, podemos perceber que é buscada informação dentro do *while*, informando no console o conteúdo, até o ponteiro dentro do arquivo informar que já é o fim do da leitura (EOF) e depois fechamos o arquivo.

Essas sãos as funções principais para que nosso programa possa inserir e ler clientes dentro de arquivo textos. Código 5.6 contém o conteúdo da função main. Você pode agora organizar todo o código e as funções de forma a executar o programa completo.

Código 5.6

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define NOME 100

struct Cliente
{
    char nome[NOME];
    int codigo;
    int idade;
};
```

```
void gravarCliente(struct Cliente *c);
void lerClientes(struct Cliente *c);

int main() {

    struct Cliente cliente;
    int index_menu;
    do
    {
        printf("Qual opção voce deseja executar\n");
        printf("[0] - Sair\n");
        printf("[1] - Salvar Clientes no Arquivo
Texto\n");
        printf("[2] - Imprimir Clientes do Arquivo
Texto \n");
        scanf("%d", &index_menu);
        switch(index_menu)
        {
            case 0:
            {
                index_menu = 0;
                break;
            }
            case 1:
                {
                    gravarCliente(&cliente);
                    break;
                }
            case 2:
                {
                    lerClientes(&cliente);
                    break;
                }
            default:
                {
                    printf("Escolha Errada");
                    break;
                }

        }
    } while(index_menu!=0);
}
```

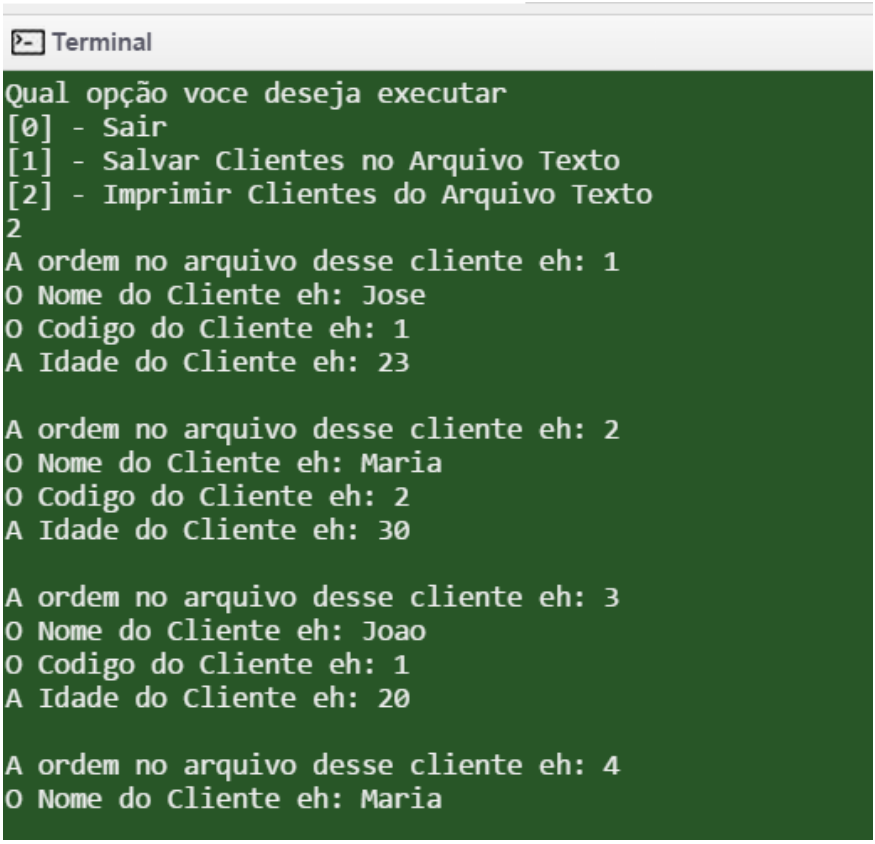


Figura 5.5 – Execução do código 5.6.

Ao executar o código 5.6 você poderá cadastrar clientes e imprimi-los, como mostra a figura 5.5. Como eu executei o código mais de uma vez e os dados ficam salvos, veja que já tenho 4 clientes cadastrados.

Faça também o seguinte. Abra o arquivo "Cliente.txt" com um editor de textos e veja o conteúdo dele. Na figura 5.6 eu apresento o conteúdo do meu arquivo usando a função *cat do Linux.*

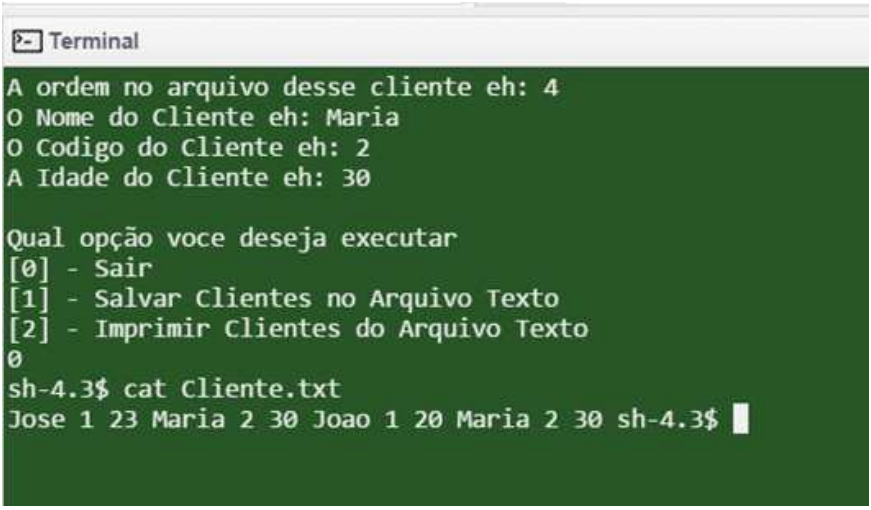


Figura 5.6 – Conteúdo do Arquivo Clientes.txt.

5.4 Arquivos Binários

Além de manipular e escrever em arquivos textos, também podemos escrever em arquivos binários, usando as mesmas funções. A linguagem C oferece outras funções que usaremos no próximo exemplo para demostrar como salvar estruturas complexas em arquivos binários. Essas funções são *fwrite()* e *fread().*

A função *fwrite()* é muito usada para gravar estruturas complexas em um arquivo binário, tendo quatro entradas de parâmetros para ser usada. Sua chamada pode ser assim:

```
fwrite(const void *elemento, size_t tamanho_elemento, size_t n_de_
elementos, FILE *arq_binario)
```

Onde:

- **elemento:** Ponteiro do elemento que vai ser escrito no arquivo.
- **tamanho_elemento:** tamanho em *byte* de cada elemento que vai ser escrito.

- **n_de_elementos:** número de elementos que serão escritos.
- **arq_binario:** ponteiro para o arquivo de gravação.

A função *fread()* é usada para recuperar os dados de arquivos binários. Sua sintaxe pode ser escrita assim:

```
fread(const void *elemento, size_t tamanho_elemento, size_t n_de_
elementos, FILE *arq_binario)
```

Os parâmetros são os mesmo usados no *fwrite():*

- **elemento :** Ponteiro para um bloco do elemento onde vai ser escrito a informação buscada no arquivo.
- **tamanho_elemento:** tamanho em *bytes* de cada elemento que vai ser escrito.
- **n_de_elementos:** número de elementos que serão escritos.
- **arq_binario:** ponteiro para o arquivo de leitura.

Para demonstrar o uso das funções em um programa, vamos utilizar o mesmo programa feito para a leitura de arquivo em texto, mas mudando as funções para gravar as estruturas de modo simples e salvando em arquivo binário.

Como sabemos a função *gravarCliente()*, tem o objetivo de abrir um arquivo, neste exemplo vamos mudamos o tipo do arquivo de texto para binário, podemos observar no parâmetro de chamada na função fopen() que informamos o tipo "b".

Veja no código 5.7A, linha 4 que tentamos abrir o arquivo para escrita complementar em binário (ab). O restante do código é muito parecido com nossa função para o modo texto. Agora, usamos a função *fwrite()* (linha 24) para escrita dos dados no arquivo binário.

Código 5.7A

```
void gravarCliente(void) {
    FILE *arquivo;
    struct Cliente c;
    arquivo = fopen("ClienteBinario.txt", "ab");
```

```
    if(arquivo == NULL)
        arquivo = fopen("ClienteBinario.txt", "w+b");

    if(arquivo == NULL) {
        printf ("Erro na abertura do arquivo
Cliente.");
        exit(EXIT_FAILURE);
    }
    else
    {
        printf("Nome do Cliente:");
        gets (c.nome);
        printf("Codigo do Cliente:");
        scanf("%d", &c.codigo);
        fflush(stdin);
        printf("Idade do Cliente:");
        scanf("%d", &c.idade);
        fflush(stdin);
        fwrite(&c, sizeof(struct Cliente), 1,
 arquivo);

        printf("\n Arquivos salvos com sucesso");
        fclose(arquivo);

    }
}
```

A diferença nesta função é que estamos salvando em um arquivo binário e usando uma função para salvar arquivos complexos como uma estrutura completa sem a necessidade de passa todos os campos, somente sua variável do tipo criado.

A função *fwrite()* simplifica a escrita, pois basta passar a estrutura Cliente e o seu tamanho (através da função *sizeof()* obtemos o tamanho de Cliente).

Por mais que tenhamos criado o arquivo com o ClienteBinario.txt, ou seja,

extensão textual, o seu conteúdo é binário (o .txt é apenas um indicar, o que vale é o conteúdo do arquivo), como você pode ver na figura 5.7.

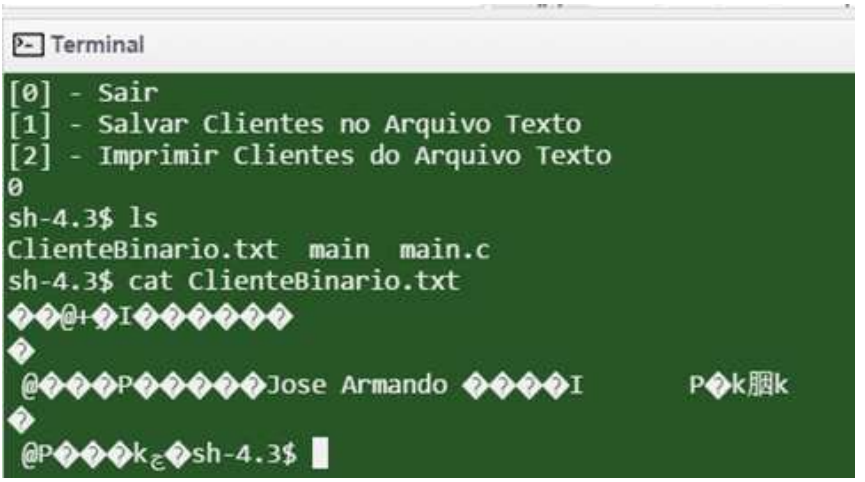


Figura 5.7 – Conteúdo do Arquivo ClienteBinario.txt.

A próxima função a ser ajustada é *lerClientes()*, responsável por ler todo conteúdo dentro do arquivo texto ou binário. Agora faremos a leitura de um arquivo binário, mas devemos saber que a estrutura foi gravada com a função *fwrite()*, desse modo vamos fazer a leitura do arquivo utilizando *fread()* que auxilia na busca de informação dentro do arquivo passando o tamanho de *byte* da estrutura salva como visto no código anterior.

Código 5.7B

```
void lerClientes(void) {
    FILE *arquivo;
    struct Cliente c;
    int index;
    arquivo = fopen("ClienteBinario.txt", "rb");

    if(arquivo == NULL) {
        printf("Erro na abertura do arquivo
Cliente.\n");
        printf("Nao existe nenhum arquivo Cliente
 ainda\n");
```

```
    }
    else
    {
        index = 0;
        while(fread(&c, sizeof(struct Cliente)
 ,1,arquivo))
        {
            index++;
            printf("A ordem no arquivo desse cliente
eh: %d", index);
            printf("\nO Nome do Cliente eh: %s",
 c.nome);
            printf("\nO Codigo do Cliente eh: %d",
 c.codigo);
            printf("\nA Idade do Cliente eh: %d",
 c.idade);
            printf("\n\n");
        }
        fclose(arquivo);
    }
}
```

Veja na linha 17 o uso de *fread()* parecido com *fwrite()*, também simplificando o acesso aos dados gravados.

Para testar a aplicação completa, o código 5.7C traz o conteúdo da função *main():*

Código 5.7C

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 3
#define NOME 100

struct Cliente
{
    char nome[NOME];
```

```
    int codigo;
    int idade;
};

void gravarCliente(void);
void lerClientes(void);

int main() {

    int index_menu;
    do
    {
        printf("Qual opção voce deseja executar\n");
        printf("[0] - Sair\n");
        printf("[1] - Salvar Clientes no Arquivo
Texto\n");
        printf("[2] - Imprimir Clientes do Arquivo
Texto \n");
        scanf("%d", &index_menu);
        fflush(stdin);
        switch(index_menu)
        {
            case 0:
            {
                index_menu = 0;
                break;
            }
            case 1:
                {
                    gravarCliente();
                    break;
                }
            case 2:
                {
                    lerClientes();
                    break;
```

```
                }
            default:
                {
                    printf("Escolha Errada");
                    break;
                }

        }
    } while(index_menu!=0);
}
```

5.5 Outras Funções Importantes para Manipulação de Arquivos

Rewind: Reposiciona o indicador de posição de arquivo no início do arquivo.

```
/* Protótipo */
void rewind(FILE *fp);
```

Ferror: Determina se uma operação com arquivo produziu um erro, retornando verdadeiro se ocorreu um erro durante a última operação no arquivo, caso contrário retorna falso. É importante chama-la imediatamente após cada operação com arquivo, pois toda operação modifica a condição de erro

```
/* Protótipo */
int ferror(FILE *fp);
```

Remove: Apaga o arquivo e retorna zero se for bem sucedida ou um valor diferente de zero caso contrário.

```
/* Protótipo */
int remove(const char *filename);
```

Fflush: Esvazia o conteúdo de uma stream de saída. Retorna 0 para indicar

sucesso ou EOF caso contrário. Escreve o conteúdo do buffer para o arquivo. Se for chamada com um valor nulo todos os arquivos abertos para saída serão descarregados.

```
/* Protótipo */
int fflush(FILE *fp);
/* Em Linux use a função abaixo */
__fpurge(stdin);
```

Fseek: Modifica o indicador de posição de arquivo. Devolve 0 se bem sucedida ou um valor diferente de zero se ocorrer um erro.

```
/* Protótipo */
int fseek(FILE *fp, long numbytes, int origin);
```

Onde:

- fp é o ponteiro de arquivo devolvido na chamada a *fopen().*
- num_*bytes* é o número de *bytes* a partir de *"origin"*, que se tornará a nova posição.
- *origin* o ponto de origem do deslocamento.

ORIGIN	NOME DA MACRO
Início do arquivo	SEEK_SET
Posição atual	SEEK_CUR
Final do arquivo	SEEK_END

5.6 Definindo Tipos com Typedef

Nós aprendemos no capítulo 4 sobre a criação e utilização de estruturas heterogêneas. No entanto, toda utilização de estrutura necessitava da palavra reservada struct.

Ao usar *typedef* podemos evitar o uso de *struct* e, ao mesmo tempo, definir um novo tipo abreviado de uma estrutura. Funciona com variáveis simples, vetores, etc, também!

Sua forma geral é:

```
typedef <tipo> <novo_nome>;
```

Por exemplo:

```
typedef unsigned int uint;
 typedef int* pint;
 typedef float Vetor[4];
 uint i,j;
 Vetor v;
 v[0] = 3;
```

Veja o código 5.8 no qual usamos typedef com estruturas mais complexas.

Código 5.8

```
...
struct end {
      char rua[40];
      int num;
      char bairro[30];
      char cidade[20];
};
typedef struct end endereco;

endereco e1;
strcpy(e1.rua, "Tiradentes");
strcpy(e1.bairro, "Jardim dos Inconfidentes");
```

```
strcpy(e1.cidade, "Brasilia");
e1.num = 100;

printf("Rua: %s, %d\n", e1.rua, e1.num);
printf("Bairro: %s\n", e1.bairro);
printf("Cidade: %s\n", e1.cidade);
```

A partir da linha 8 a estrutura end passa a poder ser referenciada como apenas endereco.

É possível ainda usar *typedef* de outras duas formas com estruturas:

Inserir na própria declaração:

```
typedef struct end {
      char rua[40];
      int num;
      char bairro[30];
      char cidade[20];
} Endereco;
```

Inserir na própria declaração e não etiquetar (colocar nome) na estrutura, apenas no tipo:

```
typedef struct {
      char rua[40];
      int num;
      char bairro[30];
      char cidade[20];
} Endereco;
```

5.7 Criando Bibliotecas de Funções

Quando temos várias funções num programa o mesmo passa a ficar complexo, com muitas linhas de código e isto, com o tempo, vai dificultando alterações e

manutenções. Por isto, podemos colocá-las em um arquivo separado, organizando-os por quais funções serão armazenadas em quais arquivos. É interessante que todas as funções que estão conceitualmente relacionadas fiquem no mesmo arquivo.

Ao fazer isto estamos criando uma biblioteca de funções!

Tecnicamente uma biblioteca é diferente de um arquivo de funções compilado separadamente, pois quando as rotinas em uma biblioteca são linkeditadas com o restante do programa apenas as funções que o programa realmente usa são carregadas e linkeditadas. Em um arquivo compilado separadamente, todas as funções são carregadas e linkeditadas com o programa.

Os compiladores C vem com biblioteca C padrão de funções para realizar as tarefas mais comuns (como vimos ao longo dos capítulos). O padrão ANSI C especifica um conjunto mínimo que estará contido no compilador.

Quando fazemos referência a uma função da biblioteca padrão usamos a seguinte diretiva:

```
#include <arquivo_cabeçalho>
```

Para criarmos nossa biblioteca vamos criar os arquivos *bib.h*, *bib.c* e o arquivo main. Basta adicionar estes arquivos no seu projeto no dev-c. Se estiver usando o compilar online, basta criar os arquivos separadamente. Depois mostrarei o comando de compilação.

O código 5.9 mostra o arquivo *bib.h* e seu conteúdo. Veja que ele contém apenas as definições das funções, seus protótipos.

Código 5.9 – bib.h

```
#ifndef BIB_H_INCLUDED
#define BIB_H_INCLUDED
#define PI 3.14159
float retangulo(float base, float altura);
float triangulo(float base, float altura);
float quadrado(float base);
float circulo(float raio);
#endif // BIB_H_INCLUDED
```

Por segurança, nós usamos as diretivas ifndef e endif para que o conteúdo ali seja tratado como uma macro que só será compilada e inserida se ainda não foi feito por nenhuma outra biblioteca.

Já o arquivo bib.c, no código 5.10 contém as implementações das funções da biblioteca. Veja que ele precisa "incluir" a bib.h em suas diretivas. Outro detalhe é o uso de aspas (") ao invés de "<" e ">". Isto se dá, pois a biblioteca definida está em nossa pasta de trabalho. Os sinais de menor e maior são usados para as bibliotecas na pasta padrão de seu compilador.

Código 5.10 – bib.c

```
#include "bib.h"
float retangulo(float base, float altura) {
    return base * altura;
}
float triangulo(float base, float altura) {
    return base * altura/2;
}
float quadrado(float base) {
    return area_retangulo(base, base);
}
float circulo(float raio) {
    return PI * raio * raio;
}
```

Por sim, o código 5.11 apresenta nossos testes, mais uma vez incluindo bib.h entre aspas.

Código 5.11 - Main

```
#include <stdio.h>
#include <stdlib.h>
#include "bib.h"
int main()
```

```
{
    printf("Area ret: %.2f \n", retangulo(12, 3.3f));
    printf("Area tri: %.2f \n", triangulo(10, 4.3f));
    printf("Area quadrado: %.2f \n", quadrado(14));
    printf("Area circulo: %.2f \n", circulo(8.4f));
    return 0;
}
```

Veja que, pela figura 5.8, os 3 arquivos fazem parte do seu projeto agora.

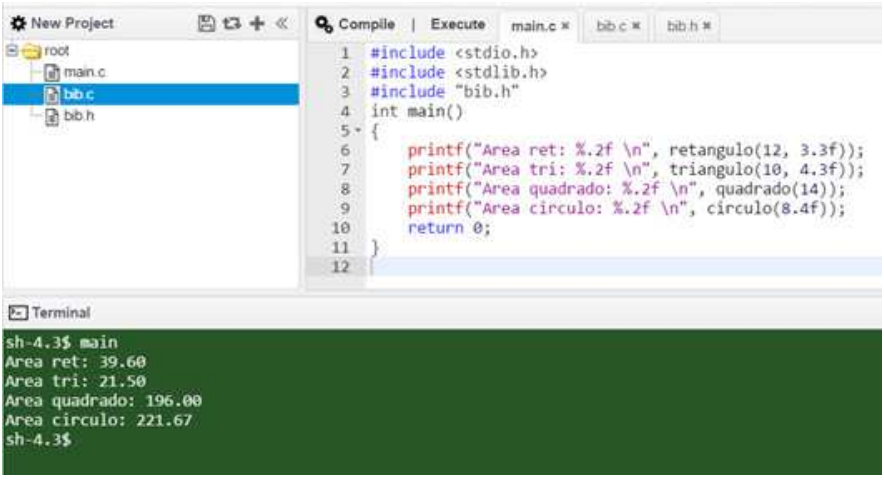


Figura 5.8 – Criando arquivos com bibliotecas de funções.

Se estiver usando a compilação em linha de comando (seja no simulador que passei, seja no Linux ou Windows) será preciso fazer assim para compilar e linkeditar suas bibliotecas:

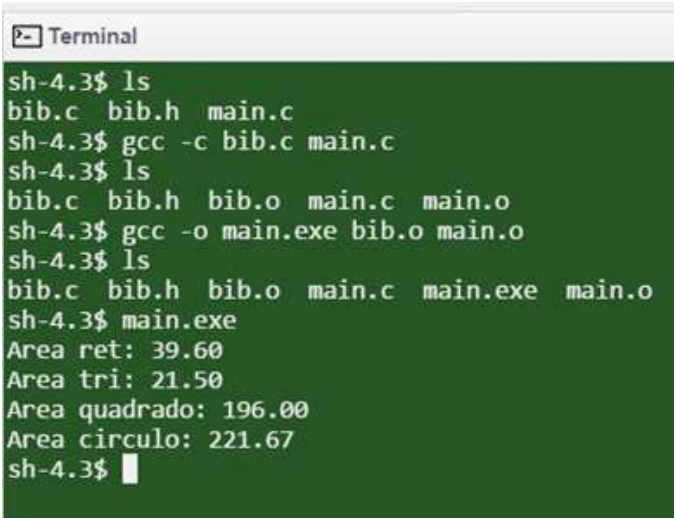


Figura 5.9 – Compilando bibliotecas em linha de comando.

O comando *ls* nos mostra o conteúdo do diretório: apenas nossos 3 arquivos.

Depois, o comando:

```
gcc -c bib.c main.c
```

Compila a biblioteca e nosso arquivo main. Veja que um novo ls mostra os arquivos bib.o e main.o.

Depois disto, o comando:

```
gcc -o main.exe bib.o main.o
```

Cria o arquivo main.exe (que não precisa desta extensão no Linux) adicionando nossa biblioteca criada e compilada no programa main.

Se você estiver usando a IDE Dev-cpp basta compilar e executar o seu programa.

REFLEXÃO

Neste capítulo expandimos nossos recursos para programação em linguagem C. Vimos como persistir dados, manipulando arquivos em modo texto ou binário, aprendemos um pou-

co sobre a criação de bibliotecas de funções e sobre diretivas para criação delas.
Minha sugestão agora é que você use todo o conhecimento aprendizado e aplique nos exemplos anteriores do livro. Tente criar aplicações persistentes em exemplos anteriores, salvando o estado das suas execuções.
Bom trabalho!

LEITURA

Para complementar seu aprendizado computação, programação e linguagem C sugiro os seguintes links em inglês:

- Mais sobre diretivas de pré processamento:
- http://www.cprogramming.com/reference/preprocessor/ o Acesso em março de 2015.
- http://www.cprogramming.com/ Acesso em março de 2015.

Também sugiro a leitura de livros em português como:

- MIZRAHI V. V.: Treinamento em Linguagem C. São Paulo: Pearson Prentice Hall, 2008.

E também sugiro os seguintes links em português

- Uso de funções, link-edição, etc:
- http://www.klebermota.eti.br/2013/03/11/usando-o-gcc-e-o-make-para-compilar-lincar-e-criar-aplicacoes-cc/ Acesso em março de 2015.

REFERÊNCIAS BIBLIOGRÁFICAS

KERNIGHAN, B. W.; RITCHIE, D. M. C: **a linguagem de programação,** padrão ANSI. Rio de Janeiro: Campus, 1995. 289p.
KELLEY, A.; POHL, I. A **Book on C: Programming in C**. 4. ed. Boston: Addison Wesley, 1997. 726p.
ZIVIANI, N. **Projetos de algoritmos:** com implementações em Pascal e C. São Paulo: Pioneira Thomson, 2002. 267p.
ARAÚJO, J. **Dominando a Linguagem C.** 1. ed. Ciência Moderna, 2004, 146p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos:** teoria e prática. Rio de Janeiro: Campus, 2002. 916p.

FORBELLONE, A. L. **Lógica de Programação.** 2. ed. São Paulo: Makron Books, 2000. 195p.

KNUTH, D. E. **The Art of Computer Programming, Fundamental Algorithms.** 3. ed. Boston: Addison Wesley, 1997. 672p. (vol 1)

SEBESTA, R. W. **Conceitos de Linguagens de Programação.** 4. ed. Porto Alegre: Bookman, 2000. 624p.

ISBN 978-85-5548-155-0