LINGUAGEM C

SUMÁRIO

1 A Linguagem C

O C nasceu na década de 70. Seu inventor, Dennis Ritchie, implementou-o pela primeira vez usando um DEC PDP-11 rodando o sistema operacional UNIX. O C é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.
O C é uma linguagem de programação genérica que é utilizada para a criação de programas diversos como processadores de texto, planilhas eletrônicas, sistemas operacionais, programas de comunicação, programas para a automação industrial, gerenciadores de bancos de dados, programas de projeto assistido por computador, programas para a solução de problemas da Engenharia, Física, Química e outras Ciências, etc... É bem provável que o Navegador que você utiliza tenha sido escrito em C ou C++.

Dados históricos:

- 1969 - Desenvolvimento do UNIX (num PDP 7 em linguagem Assembly);
- 1969 - Desenvolvimento da linguagem BCPL, próxima do Assembly;
- 1970 - Desenvolvimento da linguagem B, sucessora da anterior (o B é a 1ª letra de BCPL);
- 1971 - Primeiro desenvolvimento da linguagem C, sucessora da anterior (o C é a 2ª letra de BCPL);
- 1973 - O sistema operativo UNIX é reescrito em linguagem C;
- 1978 - Primeira edição do livro The C Programming Language, Kernighan & Ritchie;
- 1983-1988 - Definição do ANSI C;

A linguagem C pode ser considerada como uma linguagem de médio nível, pois possui instruções que a tornam ora uma linguagem de alto nível e estruturada como o Pascal, se assim se fizer necessário, ora uma linguagem de baixo nível, pois possui instruções tão próximas da máquina, que só o Assembler possui.
De fato com a linguagem C podemos construir programas organizados e concisos (como o Pascal), ocupando pouco espaço de memória com alta velocidade de execução (como o Assembler). Infelizmente, dada toda a flexibilidade da linguagem, também poderemos escrever programas desorganizados e difíceis de serem compreendidos.
Devemos lembrar que a linguagem C foi desenvolvida a partir da necessidade de se escrever programas que utilizassem recursos próprios da linguagem de máquina de uma forma mais simples e portável que o assembler.
A inúmeras razões para a escolha da linguagem C como a predileta para os desenvolvedores “profissionais”. As características abaixo servirão para mostrar o porquê de sua ampla utilização.

Características da Linguagem C:

- Portabilidade entre máquinas e sistemas operacionais.
- Dados compostos em forma estruturada.
- Programas Estruturados.
- Total interação com o Sistema Operacional.
- Código compacto e rápido, quando comparado ao código de outras linguagens de complexidade análoga.

Pergunta:
1. Se você fosse desenvolver um vírus qual linguagem seria utilizada?

2 Sintaxe

A sintaxe são regras detalhadas para cada construção válida na linguagem C.
Estas regras estão relacionadas com os **tipos**, as **declarações**, as **funções** e as **expressões**.

Os **tipos** definem as propriedades dos dados manipulados em um programa.
A linguagem C disponibiliza quatro tipos básicos que podem ser utilizados para a declaração de variáveis:

Tipo	Descrição	Bytes	Intervalo de valores
char	Um único caracter	1	0 a 255
int	Números inteiros (sem casas decimais)	4	-2147483648 a +2147483647
float	Números em ponto flutuantes com precisão simples (7 casas decimais)	4	3,4.10-38 a 3,4.1038
double	Números em ponto flutuante com precisão dupla (15 casas decimais)	8	3,4.10-4932 a 1,1.10-4932

As **declarações** expressam as partes do programa, podendo dar significado a um **identificador**, alocar memória, definir conteúdo inicial, definir funções.
As **funções** especificam as ações que um programa executa quando roda. A determinação e alteração de valores definidas nas **expressões,** que são combinações de variáveis, constantes e operadores.
As **funções** são as entidades operacionais básicas dos programas em C, que por sua vez são a união de uma ou mais funções executando cada qual o seu trabalho.
Há funções básicas que estão definidas na **biblioteca C**. As funções **printf()** e **scanf()** por exemplo, permitem respectivamente escrever na tela e ler os dados a partir do teclado. O programador também pode definir novas funções em seus programas, como rotinas para cálculos, impressão, etc.
Todo programa C inicia sua execução chamando a função **main(),** sendo obrigatória a sua declaração no programa principal.
Comentários no programa são colocados entre /* e */ não sendo considerados na compilação.
Cada instrução encerra com ; (ponto e vírgula) que faz parte do comando.

A tabela abaixo faz uma analogia para se entender melhor as diferenças das estruturas linguagem C com outras sintaxes conhecidas:

Portugol	**Pascal**	**C**
programa	program	main
inicio	begin	{
fim	end	}
var	var	
imprima	write	printf
leia	read	scanf
para	for	for
enquanto	while	while
inteiro	integer	int
real	real	float
caracter	char	char
string	string	char[]
e	and	&&
ou	or	||
negar	not	!
menor	<	<
maior	>	>
maior igual	>=	>=
menor igual	<=	<=
igual	=	==
incremento	inc ou (valor := (valor + 1))	++
decremento	dec ou (valor := (valor - 1))	--
atribuição	:=	= ou ->
caso	case	switch
comentário	{} ou //	/* *\

se condicao **entao**	**if** condicao **then**	**if** (condicao)
para i **de** 1 **ate** 10 **faca**	**for** i:=0 **to** 10 **do**	**for**(i=1;i <= 10;i = i + 1)
enquanto condicao **faca**	**while** (condicao) **do**	**while** (condicao)
	Units (Delphi)	#include

2.1 O C é "Case Sensitive"

É importante ressaltar um ponto de suma importância: o C é "Case Sensitive", isto é, maiúsculas e minúsculas fazem diferença. Se declarar uma variável com o nome soma ela será diferente de Soma, SOMA, SoMa ou sOmA. Da mesma maneira, os comandos do C if e for, por exemplo, só podem ser escritos em minúsculas, pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

2.2 Diretivas de compilação #include

As diretivas dizem ao compilador para incluir na compilação do programa outros arquivos, geralmente estes arquivos contem bibliotecas de funções ou rotinas do próprio C ou do usuário.

Sua sintaxe geral é:

#include "nome_do_arquivo"
ou
#include <nome_do_arquivo>

Arquivo Descrição:

Nome do arquivo	**Descrições das funções**
stdio.h	Funções de entrada e saída (I/O)
string.h	Funções de tratamento de strings
math.h	Funções matemáticas
ctype.h	Funções de teste e tratamento de caracteres
stdlib.h	Funções de uso genérico
conio.h	Funções para controle da tela

2.3 Inicialização

Inicialização corresponde a definir um valor inicial para a variável, o qual será armazenado no momento em que a variável é criada. A inicialização de uma variável pode ser feita com uma constante, expressão ou função.

Exemplo:
int a=10, b=a-50, c=abs(b);
char letra='A';
float vf=25.781;

Na falta de inicialização, variáveis globais são inicializadas automaticamente com zero. As variáveis locais possuem valor indefinido (lixo) quando não inicializadas na declaração, portanto, não podem ter seus valores utilizados em nenhuma operação antes que algum conteúdo seja atribuído, sob pena de que estas operações vão resultar em valores incorretos.
Existe em C a possibilidade de utilização de valores chamado constante ao programa, que são valores que são mantidos fixos pelo compilador.

! Importante destacar que no C um caracter equivale a ‘t’ sempre utilizando aspas simples (‘’). Já uma string equivale a um vetor de caracteres terminado por um caractere nulo ('\0'), ou seja,

nome[20] é uma variável do tipo string que pode ser inicializada por exemplo com “Joana”, veja que a string utiliza aspas duplas (“”). Esse vetor também poderia se inicializado da seguinte forma:
nome[0] = ‘J’; /* Caracter J */
nome[1] = ‘o’; /* Caracter o */
nome[2] = ‘a’; /* Caracter a */
nome[3] = ‘n’; /* Caracter n */
nome[4] = ‘a’; /* Caracter a */

2.4 Palavras Reservadas do C

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser nos seus propósitos originais, isto é, não podemos declarar funções ou variáveis com os mesmos nomes. Como o C é "case sensitive" podemos declarar uma variável **For**, apesar de haver uma palavra reservada **for**, mas isto não é uma coisa recomendável de se fazer, pois pode gerar confusão.
Apresentamos a seguir as palavras reservadas do ANSI C. Veremos o significado destas palavras chave à medida que o curso for progredindo:

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

2.5 Primeiro programa

Vejamos um primeiro programa em C:

```
/*Um Primeiro Programa */
#include <stdio.h>

int main ()
{
        printf ("Ola! Eu estou vivo!\n");
        return(0);
}
```

3 Introdução às Funções

Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O uso de funções permite que o programa fique mais legível, mais bem estruturado. Um programa em C consiste, no fundo, de várias funções colocadas juntas.

Abaixo o tipo mais simples de função:

```
/* Funcao simples: so imprime Ola! */
#include <stdio.h>

int mensagem ()
{
        printf ("Ola! ");
```

```
	system(“pause”);
}

int main ()
{
	mensagem();
	printf ("Eu estou vivo!\n");
	return(0);
}
```

Este programa terá o mesmo resultado que o primeiro programa. O que ele faz é definir uma função **mensagem()** que coloca uma string na tela e retorna 0. Esta função é chamada a partir de **main()** , que, também é uma função. A diferença fundamental entre main e as demais funções do problema é que main é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

3.1 Argumentos

Argumentos são as entradas que a função recebe. É através dos argumentos que passamos **parâmetros** para a função. Já vimos funções com argumentos.
As funções **printf()** e **scanf()** são funções que recebem argumentos. Vamos ver um outro exemplo simples de função com argumentos:

```
/* Calcula o quadrado de x */
#include <stdio.h>

int quadrado (int x)
{
	printf ("O quadrado e %d",(x*x));
	system(“pause”);
}

int main ()
{
	int num;
	printf ("Entre com um numero: ");
	scanf ("%d",&num);
	printf ("\n\n");
	quadrado (num);
	return(0);
}
```

Na definição de **quadrado()** dizemos que a função receberá um argumento inteiro **x**. Quando fazemos a chamada à função, o inteiro **num** é passado como argumento. Há alguns pontos a observar. Em primeiro lugar temos de satisfazer aos requisitos da função quanto ao tipo e à quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo, que o C faz automaticamente, é importante ficar atento. Em segundo lugar, não é importante o nome da variável que se passa como argumento, ou seja, a variável **num**, ao ser passada como argumento para **quadrado ()** é copiada para a variável **x**. Dentro de **quadrado ()** trabalha-se apenas com **x**. Se mudarmos o valor de **x** dentro de **quadrado ()** o valor de **num** na função **main()** permanece inalterado.
Agora veja um exemplo de função de mais de uma variável. Repare que, neste caso, os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um.

Note, também, que os argumentos passados para a função não necessitam ser todos variáveis porque mesmo sendo constantes serão copiados para a variável de entrada da função.

```
/* Multiplica 3 numeros */
#include <stdio.h>

int mult (float a, float b,float c)
{
        printf ("%f",a*b*c);
        system(“pause”);
}

int main ()
{
        float x,y;
        x=23.5;
        y=12.9;
        mult (x,y,3.87);
        return(0);
}
```

3.2 Retornando valores

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui estavam retornando o número 0. Podemos especificar um tipo de retorno indicando-o antes do nome da função. Mas para dizer ao C **o que** vamos retornar precisamos da palavra reservada **return**. Sabendo disto fica fácil fazer uma função para multiplicar dois inteiros e que retorna o resultado da multiplicação.

```
/* Exemplo de return direto sem uso de variável auxiliar */
#include <stdio.h>

int prod (int x,int y)
{
        return (x*y);
}

int main ()
{
        int saida;
        saida=prod (12,7);
        printf ("A saida e: %d\n",saida);
        return(0);
}
```

Veja que, como **prod** retorna o valor de 12 multiplicado por 7, este valor pode ser usado em uma expressão qualquer. No programa fizemos a atribuição deste resultado à variável **saida**, que posteriormente foi impressa usando o **printf**. Uma observação adicional: se não especificarmos o tipo de retorno de uma função, o compilador C automaticamente suporá que este tipo é inteiro.
Com relação à função main, o retorno sempre será inteiro. Normalmente faremos a função main retornar um zero quando ela é executada sem qualquer tipo de erro.

```
/* Mais um exemplo de função, que agora recebe dois floats e também retorna um float */
#include <stdio.h>
```

```
 float prod (float x,float y)
{
        return (x*y);
}

int main ()
{
        float saida;
        saida=prod (45.2,0.0067);
        printf ("A saida e: %f\n",saida);
        return(0);
}
```

3.3 Forma geral

Apresentamos aqui a forma geral de uma função:

tipo_de_retorno *nome_da_função* ***(lista_de_argumentos)***
{
        código_da_função
}

! Importante destacar que no C não existe o conceito de **procedimento,** já que seu compilador reconhece tudo como função, ou seja, se declarado um função sem tipo de retorno, como visto acima o compilador automaticamente irá considerar como *default* retorno inteiro (int). Para você criar uma função que se pareça com um procedimento, ou seja, não retorne nenhum tipo de valor você pode criar uma função com retorno **void** que irá indicar retorno nenhum ou nulo;

3.4 Protótipos de Funções

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função **main()**. Isto é, as funções estão fisicamente antes da função **main()**. Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto as funções foram colocadas antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos.

Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?

A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno *nome_da_função* ***(declaração_de_parâmetros);***

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis.

```
/* Exemplo da função que calcula o quandrado de um número */
#include <stdio.h>
```

```
float Quadrado (float a);

int main ()
{
	float num;
	printf ("Entre com um numero: ");
	scanf ("%f",&num);
	num=Quadrado(num);
	printf ("\n\nO seu quadrado vale: %f\n",num);
	return 0;
}

float Quadrado (float a)
{
	return (a*a);
}
```

Observe que a função Quadrado() está colocada depois de main(), mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.
Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

3.5 Passagem de parâmetros por valor

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

```
/* Exemplo do uso comum de uma função, com chamada por valor */
#include <stdio.h>

float sqr (float num);
int main ()
{
	float num,sq;
	printf ("Entre com um numero: ");
	scanf ("%f",&num);
	sq=sqr(num);
	printf ("\n\nO numero original e: %f\n",num);
	printf ("O seu quadrado vale: %f\n",sq);
}

float sqr (float num)
{
	num=num*num;
	return num;
```

}

4 Instruções de Entrada e Saída

O objetivo de escrevermos programas é em última análise, a obtenção de resultados (Saídas) depois da elaboração de cálculos ou pesquisas (Processamento) através do fornecimento de um conjunto de dados ou informações conhecidas (Entradas).
Para que nosso programa possa receber dados e alocá-los em variáveis, que serão responsáveis por armazenar as informações iniciais, nossa linguagem deverá conter um conjunto de instruções que permitam ao operador interagir com o programa fornecendo os dados quando estes forem necessários.

4.1 scanf()

Uma das mais importantes e poderosas instruções servirá basicamente para promover leitura de dados (tipados) via teclado.
Sua forma geral será:

scanf("string de controle", lista de argumentos);

A string de controle assumir os valores básicos listados a seguir:

%c - leitura de caracter
%d - leitura de números inteiros
%f - leitura de números reais
%s - leitura de caracteres, ou seja, string

A lista de argumentos deve conter exatamente o mesmo número de argumentos quantos forem os códigos de formatação na <string de controle>. Se este não for o caso, diversos problemas poderão ocorrer - incluindo até mesmo a queda do sistema - quando estivermos utilizando programas compilados escritos em C. Felizmente ao utilizarmos o C, apenas uma mensagem de erro será apresentada, para que possamos corrigir o programa sem outros inconvenientes.
Cada variável a ser lida, deverá ser precedida pelo caracter &. Para seqüência de caracteres (%s), o caracter & não deverá ser usado.

A tabela abaixo exibe outras funções que podem ser utilizadas para entrada de dados, ou seja funções de leitura:

Função	**Operação**
getchar ( )	lê um caractere; espera por <enter>
getche ( )	lê um caractere com eco; não espera por <enter>
getch ( )	lê um caractere sem eco; não espera por <enter>
putchar ( )	Imprime um caractere na tela
gets ( )	lê uma string via teclado
puts ( )	Imprime uma string na tela

! Devido à forma como a função scanf trata o *buffer* (área de armazenamento temporária) de entrada, quando ela for utilizada em conjunto com as demais funções de leitura é aconselhável sempre limpar este *buffer* com fflush(stdin);

4.2 printf()

O Comando printf, segue o mesmo padrão de scanf(), porém é destinado a apresentação dos dados, enquanto aquele destina-se a leitura dos dados.

É outro dos mais poderosos recursos da linguagem C, printf() servirá basicamente para a apresentação de dados no monitor.
Sua forma geral será:

printf("string de controle", lista de argumentos);

Necessariamente você precisará ter tantos argumentos quantos forem os comandos de formatação na "string de controle". Se isto não ocorrer, a tela exibirá sujeira ou não exibirá qualquer dado.
Os caracteres a serem utilizados pelo printf() em sua <string de controle>, no momento serão os mesmos de scanf().

Tabela contendo códigos de formatação para printf e scanf:

Código	printf	scanf
%d	imprime um inteiro decimal	lê um inteiro decimal
%f	ponto decimal flutuante	lê um numero com ponto flutuante
%lf	ponto flutuante	lê um número com ponto flutuante
%s	string de caracteres	lê uma string de caracteres
%c	um único caractere	lê um único caractere
%i	decimal	lê um inteiro decimal
%p	imprime um ponteiro	lê um ponteiro
%e	notação cientifica	lê um numero com ponto flutuante
%%	imprime % na tela	

Observemos o quadro de operadores especiais suportados por printf()

Código	**Significado**
\b	Retrocesso (BackSpace)
\f	Salto de Página (Form Feed)
\n	Linha Nova (Line Feed)
\r	Retorno de carro (cr)
\a	Caracter de alerta (beep)

```
/* Exemplo de programa em C*/
#include <stdio.h>

int main(void)
{
        printf("Programa C muito simples\n");
        getchar(); /* Aguarda pressionar Enter */
}
```

clrscr( ) //função predefinida para limpar a tela, deve-se declarar a diretiva #include do arquivo conio.h para utilizá-la.

```
/* Exemplo lê e Mostra Idade */
#include <stdio.h>
#include <conio.h>

main()
{
        clrscr();
        int idade;
        char nome[30];
```

```
        printf(“Digite sua Idade: “);
        scanf(“%d”,&idade);
        printf(“Seu Nome: “);
        scanf(“%s”,nome);    /* Strings não utilizar ‘&’ na leitura */
        printf(“%s sua idade e %d anos. \n”, nome, idade);
}
```

Obs: No compilador DEVC++ substitua o clrscr(); por **system(“cls”);**

4.3 O tamanho e precisão do campo

O tamanho do campo indica quantas posições da saída serão utilizados para exibir o valor correspondente à um especificador de formato. O tamanho do campo é determinado colocando-se um número entre o % e o caracter indicador do formato. Por exemplo, %5d indica que o valor a ser impresso vai ocupa 5 posições na tela. Por padrão, as posições em excesso são preenchidas com brancos e o valor é alinhado à direita do campo.
Se o tamanho do campo for precedido por um símbolo - (menos), então o valor será alinhado à esquerda do campo. Se o tamanho do campo for precedido com o símbolo 0 (zero), então as posições excedentes são preenchidas com zeros. Sempre que o tamanho do campo for insuficiente para exibir o valor solicitado, este tamanho é ignorado. Desta forma um valor nunca é truncado.
Para valores em ponto-flutuante, pode-se definir o número de casas decimais a serem impressas, colocando-se o um ponto (.) e outro número depois do tamanho do campo. Por exemplo, o formato %10.2f, exibe o valor em ponto-flutuante com duas casas decimais, dentro de um campo de tamanho 10. Se esta mesma lógica for utilizada com strings (formato %s), o valor antes do ponto continua indicando a tamanho do campo, porém a segunda parte limitará o número máximo de caracteres da string a serem impressos.

5 Operadores

5.1 Operadores aritméticos

São utilizados para efetuar as operações aritméticas com os seus operandos. Estes operandos podem ser utilizados com qualquer tipo de dados, exceto o resto da divisão, o qual não pode ter operandos em ponto flutuante.

Operador	**Descrição**
=	Atribuição
+	Soma
-	Subtração
*	Multiplicação
/	Divisão (se os dois valores forem int, o resultado não terá casas decimais)
%	Resto da divisão inteira
++	Incremento pré ou pós-fixado
--	Decremento pré ou pós-fixado

```
/* Exemplo utilizando diversos operadores aritméticos */
#include <stdio.h>

main(void)
{
        int a,b,c;
        a=b=c=2;
        b= ++a;
        c= b++;
        printf("a:%d\nb:%d\nc:%d\n", a, b, --c);
```

```
	getchar();
}


/* Exemplo: Dado um número, calcule seu quadrado. */
#include <sdtio.h>
#include <conio.h>

main()
{
	clrscr();
	int numero;
	printf(“Digite um Numero: “);
	scanf(“%d”,&numero);
	printf(“O %d elevado ao quadrado resulta em %d. \n”,numero,numero*numero);
	system("pause");
}
```

Atenção especial deve ser dada à operação de divisão. Numa operação onde tanto o dividendo como o divisor forem valores inteiros o resultado perderá as casas decimais, independente do tipo da variável ao qual estará sendo atribuído.

```
/* Exemplo destacando os cuidados com as operações e variáveis relacionadas à divisão */
#include <stdio.h>
main(void)
{
	int dividendo=10, divisor=3;
	float quociente=0.0;
	quociente = dividendo / divisor;
	printf("%d/%d = %.2f\n", dividendo, divisor, quociente);
	getchar();
}
```

5.2 Operadores combinados

Sempre que em um programa C aparece uma expressão onde o resultado da operação está sendo atribuída para o seu primeiro operando (da esquerda), conforme o formato x = x op y; esta expressão pode ser reduzida para o formato x op= y;

Expressão Normal	Expressão Simplificada
a = a + b;	a+=b;
a = a - b;	a-=b;
a = a * b;	a*=b;
a = a / b;	a/=b;
a = a % b;	a%=b;

5.3 Operadores relacionais

Os operadores relacionais são utilizados em expressões condicionais para a comparação do valor de duas expressões.

Operador	Descrição
>	Maior que
>=	Maior ou igual à
<	Menor que

<=	Menor ou igual à
==	Igual à
!=	Diferente de

5.4 Operadores lógicos

Os operadores lógicos são utilizados para conectar expressões lógicas sendo geralmente utilizados em expressões condicionais.

Operador	**Descrição**
&&	AND lógico
||	OR lógico
!	NOT lógico

Usando os operadores lógicos podemos realizar uma grande gama de testes. A tabela-verdade destes operadores é dada a seguir:

P	**Q**	**P AND Q**	**P OR Q**
F	F	F	F
F	V	F	V
V	F	F	V
V	V	V	V

5.5 Operador ternário

O nome deste operador deve-se ao fato que ele possui 3 (três) operandos. O primeiro é uma expressão condicional que será avaliada (testada). Se esta condição for verdadeira, o segundo operando é utilizado (o valor que está após o ponto de interrogação). Se a condição for falsa, será utilizado o terceiro operando (o último valor, após o dois-pontos)

Sintaxe:

(condição)?valor1:valor2

```
/* Exemplo demonstrando a utilização do operador ternário */
#include <stdio.h>

int main(void)
{
        int n1, n2, maior;
        printf("Digite dois valores:\n");
        scanf("%d\n%d", &n1, &n2);
        maior = (n1>n2)?n1:n2;
        printf("O maior e' %d\n", maior);
        fflush(stdin);
        getchar();
}
```

5.6 Operador sizeof

Este operador retorna o tamanho em bytes ocupado na memória pela expressão ou pelo tipo indicado. O tamanho de uma variável nunca depende do seu conteúdo armazenado, mas apenas do tipo com o qual ela foi declarada. O tamanho de um arranjo é igual a soma do tamanho de seus elementos.

Sintaxe:

sizeof(expressão) ou sizeof(tipo)

Exemplo:

```
int r, x=100, vet[3];
r = sizeof(x); /* r recebe 4 (int -> 4 bytes) */
r = sizeof(double); /* r recebe 8 (double -> 8 bytes) */
r = sizeof(vet); /* r recebe 12 (3*4 -> 12) */
```

5.7 Operador de moldagem ou cast

Colocando-se o nome de um tipo de dados entre parênteses à esquerda de uma expressão, força-se aquela expressão a assumir o tipo indicado, isto é, converte-se o valor naquele ponto do programa. Quando utilizado com variável, o tipo dela não é modificado, apenas o seu valor é temporariamente convertido.

Sintaxe:

(tipo) expressão

```
/* Exemplo utilizando moldagem (cast) */
#include <stdio.h>

main(void)
{
	int dividendo=10, divisor=3;
	float quociente=0.0;
	quociente = (float)dividendo / divisor;
	printf("%d/%d = %.2f\n", dividendo, divisor, quociente);
	getchar();
}
```

5.8 Expressões

Expressões são combinações de variáveis, constantes e operadores. Quando montamos expressões temos que levar em consideração a ordem com que os operadores são executados, conforme a tabela de precedência vista abaixo:

Maior precedência

```
() [] ->
! ~ ++ -- . -(unário) (cast) *(unário) &(unário) sizeof
* / %
+ -
<<  >>
<<=  >>=
== !=
&
^
|
&&
||
?
```

= += -= *= /=

,
Menor precedência

Exemplos de expressões:

Anos=Dias/365.25;
i = i+3;
c= a*b + d/e;
c= a*(b+d)/e;

6 Estruturas de controle de fluxo

As estruturas de controle de fluxo são fundamentais para qualquer linguagem de programação. Sem elas só haveria uma maneira do programa ser executado: de cima para baixo comando por comando. Não haveria condições, repetições ou saltos. A linguagem C possui diversos comandos de controle de fluxo. É possível resolver todos os problemas sem utilizar todas elas, mas devemos nos lembrar que a elegância e facilidade de entendimento de um programa dependem do uso correto das estruturas no local certo.

6.1 O Comando if

Já introduzimos o comando if. Sua forma geral é:

if (condição)
    declaração;

A expressão, na condição, será avaliada. Se ela for zero, a declaração não será executada. Se a condição for diferente de zero a declaração será executada.

```
/* Exemplo do uso do comando if */
#include <stdio.h>

int main ()
{
        int num;
        printf ("Digite um numero: ");
        scanf ("%d",&num);
        if (num>10)
                printf ("\n\nO numero e maior que 10");
        if (num==10)
        {
                printf ("\n\nVoce acertou!\n");
                printf ("O numero e igual a 10.");
        }
        if (num<10)
                printf ("\n\nO numero e menor que 10");
        return(0);
}
```

6.1.1 O comando else

Podemos pensar no comando else como sendo um complemento do comando **if**:

if (condição)

```
        declaração_1;
else
        declaração_2;
```

A expressão da condição será avaliada. Se ela for diferente de zero a declaração 1 será executada. Se for zero a declaração 2 será executada. É importante nunca esquecer que, quando usamos a estrutura **if-else**, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas.

```
/* Exemplo anterior utilizando else */
#include <stdio.h>

int main ()
{
        int num;
        printf ("Digite um numero: ");
        scanf ("%d",&num);
        if (num==10)
        {
                printf ("\n\nVoce acertou!\n");
                printf ("O numero e igual a 10.\n");
        }
        else
        {
                printf ("\n\nVoce errou!\n");
                printf ("O numero e diferente de 10.\n");
        }
        getch( );
}
```

6.1.2 O if-else-if

A estrutura if-else-if é apenas uma extensão da estrutura **if-else**. Sua forma geral pode ser escrita como sendo:

```
if (condição_1)
        declaração_1;
else if (condição_2)
        declaração_2;
else if (condição_3)
        declaração_3;
else if (condição_n)
        declaração_n;
else
        declaração_default;
```

A estrutura acima funciona da seguinte maneira: o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero. Neste caso ele executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à primeira condição que der diferente de zero. A última declaração (default) é a que será executada no caso de todas as condições darem zero e é opcional.

```
/* Um exemplo da estrutura acima */
```

```
#include <stdio.h>
int main ()
{
	int num;
	printf ("Digite um numero: ");
	scanf ("%d",&num);
	if (num>10)
		printf ("\n\nO numero e maior que 10");
	else if (num==10)
	{
		printf ("\n\nVoce acertou!\n");
		printf ("O numero e igual a 10.");
	}
	else if (num<10)
		 printf ("\n\nO numero e menor que 10");
	getch( );
}
```

6.1.3 A expressão condicional

Quando o compilador avalia uma condição, ele quer um valor de retorno para poder tomar a decisão. Mas esta expressão não necessita ser uma expressão no sentido convencional. Uma variável sozinha pode ser uma "expressão" e esta retorna o seu próprio valor. Isto quer dizer que teremos as seguintes expressões:

```
int num;
if (num!=0) ....
if (num==0) ....

for (i = 0; string[i] != '\0'; i++)
```

equivalem a

```
	int num;
	if (num) ....
	if (!num) ....

for (i = 0; string[i]; i++)
```

Isto quer dizer que podemos simplificar algumas expressões simples.

6.1.4 ifs aninhados

O **if** aninhado é simplesmente um **if** dentro da declaração de um outro **if** externo. O único cuidado que devemos ter é o de saber exatamente a qual **if** um determinado **else** está ligado.

```
/* Exemplo utilizando ifs aninhados */

#include <stdio.h>

int main ()
{
	int num;
	printf ("Digite um numero: ");
	scanf ("%d",&num);
```

```
        if (num==10)
        {
                printf ("\n\nVoce acertou!\n");
                printf ("O numero e igual a 10.\n");
        }
        else
        {
                if (num>10)
                {
                        printf ("O numero e maior que 10.");
                 }
                else
                {
                        printf ("O numero e menor que 10.");
                }
        }
        return(0);
}
```

6.1.5 O Operador ?

Uma expressão como:

```
if (a>0)
        b=-150;
else
        b=150;
```

pode ser simplificada usando-se o operador ? da seguinte maneira:

```
b=a>0?-150:150;
```

De uma maneira geral expressões do tipo:

if (condição)
        expressão_1;
else
        expressão_2;

podem ser substituídas por:

        condição?expressão_1:expressão_2;

O operador ? é limitado (não atende a uma gama muito grande de casos) mas pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular.

```
/* Exemplo do comando ? */
#include <stdio.h>

int main()
{
        int index = 0, contador;
        char letras[5] = "Joao";
        for (contador=0; contador < 1000; contador++)
        {
                printf("\n%c",letras[index]);
                (index==3) ? index=0: ++index;
```

```
        }
        getch();
}
```

O nome Joao é escrito na tela verticalmente até a variável contador determinar o término do programa. Enquanto isto a variável index assume os valores 0, 1, 2, 3, , 0, 1, ... progressivamente.

6.2 O Comando switch

O comando **if-else** e o comando switch são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o **if**, mas o comando switch tem aplicações valiosas. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando switch é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos. Sua forma geral é:

```
switch (variável)
{
        case constante_1:
                declaração_1;
                break;
        case constante_2:
                declaração_2;
                break;
        case constante_n:
                declaração_n;
                break;
        default
                declaração_default;
}
```

Podemos fazer uma analogia entre o switch e a estrutura **if-else-if** já visto acima. A diferença fundamental é que a estrutura switch não aceita expressões. Aceita apenas constantes. O switch testa a variável e executa a declaração cujo case corresponda ao valor atual da variável. A declaração default é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.
O comando **break**, faz com que o switch seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando switch. Se após a execução da declaração não houver um **break**, o programa continuará executando. Isto pode ser útil em algumas situações, mas eu recomendo cuidado.

```
/* Exemplo do comando switch */
#include <stdio.h>

int main ()
{
        int num;
        printf ("Digite um numero: ");
        scanf ("%d",&num);
        switch (num)
        {
                case 9:
                        printf ("\n\nO numero e igual a 9.\n");
                        break;
                case 10:
```

```
                printf ("\n\nO numero e igual a 10.\n");
                break;
        case 11:
                printf ("\n\nO numero e igual a 11.\n");
                break;
        default:
                printf ("\n\nO numero nao e nem 9 nem 10 nem 11.\n");
    }
    system("pause");
}
```

6.3 O Comando for

for é a primeira de uma série de três estruturas para se trabalhar com loops de repetição. As outras são **while** e **do**. As três compõem a segunda família de comandos de controle de fluxo. Podemos pensar nesta família como sendo a das estruturas de repetição controlada.
Como já foi dito, o loop for é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

for (inicialização;condição;incremento) declaração;

O melhor modo de se entender o loop for é ver como ele funciona "por dentro". O loop for é equivalente a se fazer o seguinte:

```
inicialização;
if (condição)
{
    declaração;
    incremento;
    "Volte para o comando if"
}
```

Podemos ver, então, que o for executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Um ponto importante é que podemos omitir qualquer um dos elementos do for, isto é, se não quisermos uma inicialização poderemos omiti-la. Abaixo vemos um programa que coloca os primeiros 100 números inteiros na tela:

```
/* Exemplo que coloca os primeiros 100 números inteiros na tela */
#include <stdio.h>

int main ()
{
    int count;
    for (count=1; count<=100; count++) printf ("%d ",count);
    getch();
}
```

Note que, no exemplo acima, há uma diferença em relação ao exemplo anterior. O incremento da variável count é feito usando o operador de incremento que nós agora já conhecemos. Esta é a forma usual de se fazer o incremento (ou decremento) em um loop for.
O for na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Qualquer uma destas partes do for pode ser uma expressão qualquer do C, desde que

ela seja válida. Isto nos permite fazer o que quisermos com o comando. As três formas do for abaixo são válidas:

for (count = 1; count < 100 ; count++) { ... }
for (count = 1; count < NUMERO_DE_ELEMENTOS ; count++) { ... }
for (count = 1; count < BusqueNumeroDeElementos() ; count+=2) { ... }

Preste atenção ao último exemplo: o incremento está sendo feito de dois em dois. Além disto, no teste está sendo utilizada uma função (BusqueNumeroDeElementos()) que retorna um valor que está sendo comparado com count.

6.3.1 O loop infinito

O loop infinito tem a forma:

for (inicialização; ;incremento) declaração;

Este **loop** chama-se **loop** infinito porque será executado para sempre (não existindo a condição, ela será sempre considerada verdadeira), a não ser que ele seja interrompido. Para interromper um **loop** como este usamos o comando **break**. O comando **break** vai quebrar o **loop** infinito e o programa continuará sua execução normalmente.

/* Como exemplo vamos ver um programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuario aperte uma tecla sinalizadora de final (um FLAG). O nosso FLAG será a letra 'X'. Repare que tivemos que usar dois scanf() dentro do for. Um busca o caractere que foi digitado e o outro busca o outro caracter digitado na seqüência, que é o caractere correspondente ao <ENTER> */

```
#include <stdio.h>

int main ()
{
        int Count;
        char ch;
        printf(" Digite uma letra - <X para sair> ");
        for (Count=1;;Count++)
        {
                scanf("%c", &ch);
                if (ch == 'X')
                        break;
                printf("\nLetra: %c \n",ch);
                scanf("%c", &ch);
        }
        return(0);
}
```

6.3.2 O loop sem conteúdo

Loop sem conteúdo é aquele no qual se omite a declaração. Sua forma geral é (atenção ao ponto e vírgula!):

for (inicialização;condição;incremento);

Uma das aplicações desta estrutura é gerar tempos de espera.

```
/* Exemplo de geração de tempo de espera */
#include <stdio.h>

int main ()
{
        long int i;
        printf("\a");  /* Imprime o caracter de alerta (um beep) */
        for (i=0; i<10000000; i++);  /* Espera 10.000.000 de iteracoes */
        printf("\a"); /* Imprime outro caracter de alerta */
        return(0);
}
```

6.4 O Comando while

O comando while tem a seguinte forma geral:

while (condição) declaração;

Assim como fizemos para o comando for, vamos tentar mostrar como o while funciona fazendo uma analogia. Então o while seria equivalente a:

```
 if (condição)
{
declaração;
"Volte para o comando if"
}
```

Podemos ver que a estrutura while testa uma condição. Se esta for verdadeira a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do **for**, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo. Vamos ver um exemplo do uso do while.

```
/* Exemplo executa enquanto i for menor que 100. */
#include <stdio.h>

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

```
/* Exemplo espera o usuário digitar a tecla 'q' e só depois finaliza */
#include <stdio.h>

int main ()
{
```

```
        char Ch;
        Ch='\0';
        while (Ch!='q')
        {
                scanf("%c", &Ch);
        }
        return(0);
}
```

6.5 O Comando do-while

A terceira estrutura de repetição que veremos é o do-while de forma geral:

do
{
    declaração;
} while (condição);

Mesmo que a declaração seja apenas um comando é uma boa prática deixar as chaves. O ponto-e-vírgula final é obrigatório. Vamos, como anteriormente, ver o funcionamento da estrutura do-while "por dentro":

declaração;
if (condição)
    "Volta para a declaração"

Vemos pela análise do bloco acima que a estrutura do-while executa a declaração, testa a condição e, se esta for verdadeira, volta para a declaração. A grande novidade no comando do-while é que ele, ao contrário do **for** e do **while**, garante que a declaração será executada pelo menos uma vez.

```
/* Um dos usos da extrutura do-while é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme exemplo */
#include <stdio.h>

int main ()
{
        int  i;
        do
        {
                printf ("\n\nEscolha a fruta pelo numero:\n\n");
                printf ("\t(1)...Mamao\n");
                printf ("\t(2)...Abacaxi\n");
                printf ("\t(3)...Laranja\n\n");
                scanf("%d", &i);
        } while ((i<1)||(i>3));

        switch (i)
        {
                case 1:
                        printf ("\t\tVoce escolheu Mamao.\n");
                        break;
                case 2:
                        printf ("\t\tVoce escolheu Abacaxi.\n");
```

```
                break;
        case 3:
                printf ("\t\tVoce escolheu Laranja.\n");
                break;
    }
    system("pause");
}
```

6.6 O Comando break

Nós já vimos dois usos para o comando break: interrompendo os comandos **switch** e **for**. Na verdade, estes são os dois usos do comando break: ele pode quebrar a execução de um comando (como no caso do switch) ou interromper a execução de qualquer loop (como no caso do for, do while ou do do while). O break faz com que a execução do programa continue na primeira linha seguinte ao loop ou bloco que está sendo interrompido.
Observe que um break causará uma saída somente do laço mais interno. Por exemplo:

```
for(t=0; t<100; ++t)
{
    count=1;
    for(;;)
    {
        printf("%d", count);
        count++;
        if(count==10)
            break;
    }
}
```

O código acima imprimirá os números de 1 a 10 cem vezes na tela. Toda vez que o break é encontrado, o controle é devolvido para o laço for externo.
Outra observação é o fato que um break usado dentro de uma declaração switch afetará somente os dados relacionados com o switch e não qualquer outro laço em que o switch estiver.

6.7 O Comando continue

O comando continue pode ser visto como sendo o oposto do **break**. Ele só funciona dentro de um loop. Quando o comando continue é encontrado, o loop pula para a próxima iteração, sem o abandono do loop, ao contrário do que acontecia no comando break.

```
/* Exemplo usando do continue */
#include <stdio.h>

int main()
{
    int opcao;
    while (opcao != 5)
    {
        printf("\n\n Escolha uma opcao entre 1 e 5: ");
        scanf("%d", &opcao);
        if ((opcao > 5)||(opcao <1))
            continue;  /* Opcao invalida: volta ao inicio do loop */
        switch (opcao)
        {
```

```
                        case 1:
                                printf("\n --> Primeira opcao..");
                                break;
                        case 2:
                                printf("\n --> Segunda opcao..");
                                break;
                        case 3:
                                printf("\n --> Terceira opcao..");
                                break;
                        case 4:
                                printf("\n --> Quarta opcao..");
                                break;
                        case 5:
                                printf("\n --> Abandonando..");
                                break;
                }
        }
        system("pause");
}
```

O programa acima ilustra uma aplicação simples para o continue. Ele recebe uma opção do usuario. Se esta opção for inválida, o continue faz com que o fluxo seja desviado de volta ao início do loop. Caso a opção escolhida seja válida o programa segue normalmente.

7 Vetores e Matrizes

7.1 Vetores

Vetores nada mais são que matrizes unidimensionais. Vetores são uma estrutura de dados muito utilizada. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Para se declarar um vetor podemos utilizar a seguinte forma geral:

tipo_da_variável nome_da_variável [tamanho];

Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos: float exemplo [20];
O C irá reservar 4x20=80 bytes. Estes bytes são reservados de maneira contígua. Na linguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

exemplo[0]
exemplo[1]
.
.
.
exemplo[19]
Mas ninguém o impede de escrever:

exemplo[30]
exemplo[103]

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que você deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter variáveis sobreescritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
/*Exemplo do uso de vetor na linguagem C */
#include <stdio.h>

int main ()
{
        int num[100];  /* Declara um vetor de inteiros de 100 posicoes */
        int count=0;
        int totalnums;
        do
        {
                printf ("\nEntre com um numero (-999 p/ terminar): ");
                scanf ("%d",&num[count]);
                count++;
        } while (num[count-1]!=-999);
        totalnums=count-1;
        printf ("\n\n\n\t Os numeros que voce digitou foram:\n\n");
        for (count=0;count<totalnums;count++)
                printf (" %d",num[count]);
        return(0);
}
```

No exemplo acima, o inteiro count é inicializado em 0. O programa pede pela entrada de números até que o usuário entre com o Flag -999. Os números são armazenados no vetor num. A cada número armazenado, o contador do vetor é incrementado para na próxima iteração escrever na próxima posição do vetor. Quando o usuário digita o flag, o programa abandona o primeiro loop e armazena o total de números gravados. Por fim, todos os números são impressos. É bom lembrar aqui que nenhuma restrição é feita quanto a quantidade de números digitados. Se o usuário digitar mais de 100 números, o programa tentará ler normalmente, mas o programa os escreverá em uma parte não alocada de memória, pois o espaço alocado foi para somente 100 inteiros. Isto pode resultar nos mais variados erros no instante da execução do programa.

7.2 Matrizes

7.2.1 Matrizes bidimensionais

Já vimos como declarar matrizes unidimensionais (vetores). Vamos tratar agora de matrizes bidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

tipo_da_variável nome_da_variável [altura][largura];

É muito importante ressaltar que, nesta estrutura, o índice da esquerda indexa as linhas e o da direita indexa as colunas. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia

mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador.

```
/*Exemplo do uso de uma matriz*/
#include <stdio.h>

int main ()
{
        int mtrx [20][10];
        int i,j,count;
        count=1;
        for (i=0;i<20;i++)
                for (j=0;j<10;j++)
                {
                        mtrx[i][j]=count;
                        count++;
                        printf("\nLinha:%d Col:%d INT:%d",i,j,mtrx[i][j]);
                }
        getch( );
}
```

No exemplo acima, a matriz mtrx é preenchida, seqüencialmente por linhas, com os números de 1 a 200. Você deve entender o funcionamento do programa acima antes de prosseguir.

7.2.2 Inicialização

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como inicialização é:

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

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
char str [10] = "Joao";
char str_vect [3][10] = { "Joao", "Maria", "Jose" };
```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde matrx está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o ; no final da inicialização.

7.2.2.1 Inicialização sem especificação de tamanho

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho a priori. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```
char mess [] = "Linguagem C: flexibilidade e poder.";
int matrx [][2] = { 1,2,2,4,3,6,4,8,5,10 };
```

No primeiro exemplo, a string mess terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

8 Ponteiros

O C é altamente dependente dos ponteiros. Para ser um bom programador em C é fundamental que se tenha um bom domínio deles. Por isto, é recomendado uma atenção especial. Ponteiros são tão importantes na linguagem C que você já os viu e nem percebeu, pois mesmo para se fazer uma introdução básica à linguagem C precisa-se deles.

! O uso indevido de ponteiros podem acarretar em problemas difíceis de serem identificados;
Forma geral:

*tipo *nomevar;*

Ponteiro é uma variável que contém o endereço de outra variável. Os ponteiros são utilizados para alocação dinâmica, podendo substituir matrizes com mais eficiência. Também fornecem a maneira pelas quais funções podem modificar os argumentos chamados.

8.1 Declarando Ponteiros

Se uma variável irá conter um ponteiro, então ela deve ser declarada como tal:

```
int x,*px;
px=&x; /*a variável px aponta para x */
```

Se quisermos utilizar o conteúdo da variável para qual o ponteiro aponta:

```
y=*px;
```

O que é a mesma coisa que:

```
y=x;
```

8.2 Manipulação de Ponteiros

Desde que os pointers são variáveis, eles podem ser manipulados como as variáveis podem. Se py é um outro ponteiro para um inteiro então podemos fazer a declaração:
py=px;

```
/*Exemplo de manipulação de ponteiros*/
#include <stdio.h>

void main()
{
        int x,*px,*py;
        x=9;
        px=&x;
        py=px;
        printf("x= %d\n",x);
        printf("&x= %d\n",&x);
        printf("px= %d\n",px);
        printf("*px= %d\n",*px);
        printf("*px= %d\n",*px);
}
```

8.3 Expressões com Ponteiros

Os ponteiros podem aparecer em expressões, se px aponta para um inteiro x, então *px pode ser utilizado em qualquer lugar que x seria. O operador * tem maior precedência que as operações aritméticas, assim a expressão abaixo pega o conteúdo do endereço que px aponta e soma 1 ao seu conteúdo.
y=*px+1;

No próximo caso somente o ponteiro será incrementado e o conteúdo da próxima posição da memória será atribuído a y:
y=*(px+1);

Os incrementos e decrementos dos endereços podem ser realizados com os operadores ++ e --, que possuem procedência sobre o * e operações matemáticas e são avaliados da direita para a esquerda:
px++; / sob uma posição na memória */
(px--); / mesma coisa de *px-- */

No exemplo abaixo os parênteses são necessários, pois sem eles px seria incrementado em vez do conteúdo que é apontado, porque os operadores * e ++ são avaliados da direita para esquerda.
(*px)++ /* equivale a x=x+1; ou *px+=1 */

```
/*Exemplo da utilização de expressões com ponteiros*/
#include <stdio.h>

main()
{
	int x,*px;
	x=1;
	px=&x;
	printf("x= %d\n",x);
	printf("px= %u\n",px);
	printf("*px+1= %d\n",*px+1);
	printf("px= %u\n",px);
	printf("*px= %d\n",*px);
	printf("*px+=1= %d\n",*px+=1);
	printf("px= %u\n",px);
	printf("(*px)++= %d\n",(*px)++);
	printf("px= %u\n",px);
	printf("*(px++)= %d\n",*(px++));
	printf("px= %u\n",px);
	printf("*px++-= %d\n",*px++);
	printf("px= %u\n",px);

	system(“pause”);
}
```

8.4 Ponteiros para ponteiros

Um ponteiro para um ponteiro é uma forma de indicação múltipla. Num ponteiro normal, o valor do ponteiro é o valor do endereço da variável que contém o valor desejado. Nesse caso o primeiro ponteiro contém o endereço do segundo, que aponta para a variável que contém o valor desejado.

```
float **balanço; /*Balanço é um ponteiro para um ponteiro float*/

/*Exemplo do uso de ponteiro para ponteiros*/
#include <stdio.h>

main()
{
	int x,*p,**q;
	x=10;
	p=&x;
	q=&p;
	printf("%d \n",**q);

	system(“pause”);
}
```

8.5 Problemas com ponteiros

O erro chamado de ponteiro perdido é um dos mais difíceis de se encontrar, pois a cada vez que a operação com o ponteiro é utilizada, poderá estar sendo lido ou gravado em posições desconhecidas da memória. Isso pode acarretar em sobreposições sobre áreas de dados ou mesmo área do programa na memória.

```
int,*p;
x=10;
*p=x;
```

Estamos atribuindo o valor 10 a uma localização desconhecida de memória. A conseqüência desta atribuição é imprevisível.

8.6 Ponteiros e Matrizes

Em C existe um grande relacionamento entre ponteiros e matrizes, sendo que eles podem ser tratados da mesma maneira. As versões com ponteiros geralmente são mais rápidas.

8.7 Manipulando Matrizes Através de Ponteiros

Considerando a declaração da matriz **int a[10];**
Sendo **pa** um ponteiro para inteiro então:

```
pa=&a[0]; /*passa o endereço inicial do vetor a para o ponteiro pa */
pa=a; /* é a mesma coisa de pa=&a[0]; */
x=*pa; /*(passa o conteúdo de a[0] para x */
```

Se **pa** aponta para um elemento particular de um vetor **a**, então por definição **pa+1** aponta para o próximo elemento, e em geral **pa-i** aponta para **i** elementos antes de **pa** e **pa+i** para **i** elementos depois.
Se **pa** aponta para **a[0]** então:

```
*(pa+1) aponta para a[1]
pa+i é o endereço de a[i] e *(pa+i) é o conteúdo.
```

É possível fazer cópia de caracteres utilizando matrizes e ponteiros:

```
/*Exemplo utilizando matriz*/
#include <stdio.h>

main()
{
        int i=0;
        char t[10];
        static char s[]="abobora";
        while(t[i]=s[i])i++;
        printf("%s\n",t);

        system(“pause”);
}
```

```
/*Exemplo utilizando ponteiro*/
#include <stdio.h>

main()
{
        char *ps,*pt,t[10],s[10];
        strcpy(s,"abobora");
        ps=s;
        pt=&t[0];
        while(*ps)*pt++=*ps++;
        printf("%s \n",t);

        system(“pause”);
}
```

8.8 String de Ponteiros

Sendo um ponteiro para caracter **char *texto**;:, podemos atribuir uma constante string para texto, que não é uma cópia de caracteres, somente ponteiros são envolvidos. Neste caso a string é armazenada como parte da função em que aparecem, ou seja, como constante.
Char *texto="composto"; /* funciona como **static char texto[]=“composto”;** */

```
/*Exemplo string de ponteiros*/
#include <stdio.h>

main()
{
        char *al="conjunto";
        char re[]=”simples”;
        puts(al);
        puts(&re[0]); /* ou puts(re); */
        for(;*al;al++) putchar(*al);
        putchar('\n');

        system("pause");
}
```

8.9 Matrizes de Ponteiros

A declaração de matrizes de ponteiros é semelhante a qualquer outro tipo de matrizes:
int *x[10];

Para atribuir o endereço de uma variável inteira chamada **var** ao terceiro elemento da matriz de ponteiros:
x[2]=&var;

Verificando o conteúdo de **var**:
*x[2]

As matrizes de ponteiros são tradicionalmente utilizadas para mensagens de erro, que são constantes:

```
char *erro[]={"arquivo não encontrado\n","erro de leitura\n"};
printf("%s",erro[0]);
```

```
printf("%s",erro[1]);


/*Exemplo de matrizes de ponteiros*/
#include <stdio.h>

main()
{
	char *erro[2];
	erro[0]="arquivo nao encontrado\n";
	erro[1]="erro da leitura\n";
	for(;*erro[0];)
	printf("%c",*erro[0]++);

	system("pause");
}
```

9 Funções avançado

9.1 Passagem de parâmetros por valor e passagem por referência

No exemplo visto no capítulo 3 o parâmetro formal **num** da função **sqr()** sofre alterações dentro da função, mas a variável **num** da função **main()** permanece inalterada: é uma chamada por valor.
Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).
O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. Há, entretanto, no C, um recurso de programação que podemos usar para simular uma chamada por referência.
Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo **ponteiros**. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um **&** na frente das variáveis que estivermos passando para a função.

```
/* Exemplo do uso de uma função, com passagem de parâmetros por referência */
#include <stdio.h>

void Swap (int *a,int *b);
int main (void)
{
        int num1,num2;
        num1=100;
        num2=200;
        Swap (&num1,&num2);
        printf ("\n\nEles agora valem %d  %d\n",num1,num2);

        system(“pause”);
}

void Swap (int *a,int *b)
{
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
```

Não é muito difícil. O que está acontecendo é que passamos para a função Swap o endereço das variáveis **num1** e **num2**. Estes endereços são copiados nos ponteiros a e b. Através do operador * estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Mas, quem é este conteúdo? Nada mais que os valores armazenados em **num1** e **num2**, que, portanto, estão sendo modificados!
Espere um momento... Será que nós já não vimos esta estória de chamar uma função com as variáveis precedidas de **&**? Já! É assim que nós chamamos a função **scanf()**. Mas porquê? Vamos pensar um pouco. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e portanto precisa alterar seus valores. Por isto passamos para a função o endereço da variável a ser modificada!

9.2 Arquivo-cabeçalhos

Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em **.h**. A extensão **.h** vem de **header** (cabeçalho em inglês). Já vimos exemplos como **stdio.h**, **conio.h**, **string.h**. Estes arquivos, na verdade, não possuem os códigos completos das funções. Eles só contêm *protótipos* de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem". Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.
Se você criar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-cabeçalhos e incluí-los também.
Suponha que a função 'int EPar(int a)', seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado por exemplo de 'funcao.h' teremos a seguinte declaração:
int EPar(int a);
O código da função será escrito num arquivo a parte. Vamos chamá-lo de 'funcao.h'. Neste arquivo teremos a definição da função:

```
int EPar (int a)
{
	if (a%2)		/* Verifica se a e divisivel por dois */
		return 0;
	else
		return 1;
}
```

Por fim, no arquivo do programa principal teremos o programa principal. Vamos chamar este arquivo aqui de 'princip.c'.

```
#include <stdio.h>
#include "funcao.h"
void main ()
{
	int num;
	printf ("Entre com numero: ");
	scanf ("%d",&num);
	if (EPar(num))
	printf ("\n\nO numero e par.\n");
	else
	printf ("\n\nO numero e impar.\n");
}
```

10 Tipos de Dados Definidos Pelo Usuário

10.1 Estruturas

Uma estrutura agrupa várias variáveis numa só. Funciona como uma ficha pessoal que tenha nome, telefone e endereço. A ficha seria uma estrutura. A estrutura, então, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

10.1.1 Criando

Para se criar uma estrutura usa-se o comando **struct**. Sua forma geral é:

```
struct nome_do_tipo_da_estrutura
{
	tipo_1 nome_1;
	tipo_2 nome_2;
	...
	tipo_n nome_n;
} variáveis_estrutura;
```

O nome_do_tipo_da_estrutura é o nome para a estrutura. As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo nome_do_tipo_da_estrutura. Um primeiro exemplo:

```
struct est
{
	int i;
	float f;
} a, b;
```

Neste caso, est é uma estrutura com dois campos, i e f. Foram também declaradas duas variáveis, a e b que são do tipo da estrutura, isto é, a possui os campos i e f, o mesmo acontecendo com b.
Vamos criar uma estrutura de endereço:

```
struct tipo_endereco
{
        char rua [50];
        int numero;
        char bairro [20];
        char cidade [30];
        char sigla_estado [3];
        long int CEP;
};
```

Vamos agora criar uma estrutura chamada ficha_pessoal com os dados pessoais de uma pessoa:

```
struct ficha_pessoal
{
        char nome [50];
        long int telefone;
        struct tipo_endereco endereco;
};
```

Vemos, pelos exemplos acima, que uma estrutura pode fazer parte de outra (a struct tipo_endereco é usada pela struct ficha_pessoal).

10.1.2 Usando

Vamos agora utilizar as estruturas declaradas na seção anterior para escrever um programa que preencha uma ficha.

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

struct tipo_endereco
{
        char rua [50];
        int numero;
        char bairro [20];
        char cidade [30];
        char sigla_estado [3];
        long int CEP;
};

struct ficha_pessoal
{
        char nome [50];
        long int telefone;
        struct tipo_endereco endereco;
};

main (void)
{
        struct ficha_pessoal ficha;
        strcpy (ficha.nome,"Luiz Osvaldo Silva");
```

```
        ficha.telefone=4921234;
        strcpy (ficha.endereco.rua,"Rua das Flores");
        ficha.endereco.numero=10;
        strcpy (ficha.endereco.bairro,"Cidade Velha");
        strcpy (ficha.endereco.cidade,"Belo Horizonte");
        strcpy (ficha.endereco.sigla_estado,"MG");
        ficha.endereco.CEP=31340230;
        return 0;
}
```

O programa declara uma variável ficha do tipo **ficha_pessoal** e preenche os seus dados. O exemplo mostra como podemos acessar um elemento de uma estrutura: basta usar o ponto (**.**). Assim, para acessar o campo telefone de ficha, escrevemos:

ficha.telefone = 4921234;

Como a struct ficha pessoal possui um campo, endereco, que também é uma struct, podemos fazer acesso aos campos desta struct interna da seguinte maneira:

ficha.endereco.numero = 10;
ficha.endereco.CEP=31340230;

Desta forma, estamos acessando, primeiramente, o campo endereco da struct ficha e, dentro deste campo, estamos acessando o campo numero e o campo CEP.

10.1.3 Atribuindo

Podemos atribuir duas estruturas que sejam do *mesmo* tipo. O C irá, neste caso, copiar uma estrutura, campo por campo, na outra. Veja o programa abaixo:

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

struct est1
{
        int i;
        float f;
};

void main()
{
        struct est1 primeira, segunda; /* Declara primeira e segunda como structs do
        tipo est1 */
        primeira.i = 10;
        primeira.f = 3.1415;
        segunda = primeira; /* A segunda struct e' agora igual a primeira */
        printf("Os valores armazenasdos na segunda struct sao :  %d  e  %f ", segunda.i , segunda.f);

        system("pause");
}
```

São declaradas duas estruturas do tipo **est1**, uma chamada **primeira** e outra chamada **segunda**. Atribuem-se valores aos dois campos da struct primeira. Os valores de **primeira** são copiados em **segunda** apenas com a expressão de atribuição:

segunda = primeira;

Todos os campos de primeira serão copiados na segunda. Note que **isto é diferente do que acontecia em vetores**, onde, para fazer a cópia dos elementos de um vetor em outro, tínhamos que copiar elemento por elemento do vetor. Nas structs é muito mais fácil!

10.2 Enumerações

Numa enumeração podemos dizer ao compilador quais os valores que uma determinada variável pode assumir. Sua forma geral é:

enum nome_do_tipo_da_enumeração {lista_de_valores} lista_de_variáveis;

Vamos considerar o seguinte exemplo:

enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado,domingo};

O programador diz ao compilador que qualquer variável do tipo dias_da_semana só pode ter os valores enumerados. Isto quer dizer que poderíamos fazer o seguinte programa:

```
#include <stdio.h>

enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};

main (void)
{
        enum dias_da_semana d1,d2;
        d1=segunda;
        d2=sexta;
        if (d1==d2)
        {
                printf ("O dia e o mesmo.");
        }
        else
        {
                printf ("São dias diferentes.");
        }
        return 0;
}
```

11 Utilizando arquivos

11.1 Entradas e Saídas em Dispositivos – Arquivos

A biblioteca <stdio.h> provê um vasto conjunto de funções e macros para entrada e saída (E/S), além daquelas já estudadas e até aqui utilizadas para ler e escrever nos dispositivos padrões (normalmente vídeo e teclado). Estas novas funções são úteis para E/S em arquivos, normalmente armazenados em meio magnético, embora, por definição, no mundo Unix, qualquer dispositivo conectado ao sistema pode ser considerado um arquivo (inclusive teclado, modem, impressora, etc.).
Todo arquivo precisa ser aberto para que o seu conteúdo esteja disponível ao programa. A ação de abrir o arquivo envolve reservar áreas de memória para armazenamento temporário de dados necessários à transferência e a solicitação do acesso ao sistema operacional. Após a abertura, se esta

teve sucesso, o programa pode utilizar as funções adequadas para ler ou escrever dados no arquivo aberto. Eventualmente a abertura de um arquivo pode falhar, como nos casos em que o arquivo a ser lido não existe, o usuário não tem permissão de acesso ao arquivo ou diretório, entre outros. Finalmente, após os dados terem sido processados pelo programa e quando este não necessitar mais acessar o conteúdo do arquivo, este deve ser fechado.
Todo o programa C que necessitar abrir arquivos deverá declarar, para cada arquivo aberto, uma variável do tipo FILE *. Esta variável será associada com o nome do arquivo no momento da abertura e todo o acesso posterior, através das funções adequadas, fará uso desta variável. A seguir, serão apresentadas funções utilizadas para manipulação de arquivos.

11.1.1 Função fopen

Abre um arquivo, tornando-o disponível a ser acessado pelo programa.

*FILE *fopen(const char* NomeArquivo, const char* modo);*

Parâmetros:
• nome: String contendo o caminho e nome do arquivo a ser aberto.
• modo: Modo de abertura do arquivo. Indica qual tipo de acesso ao arquivo está sendo solicitado para o sistema operacional. Deve ser uma string contendo uma combinação válida dos caracteres apresentados na tabela abaixo.

Opções para abertura de arquivos

r	Abrir arquivo existente para leitura
w	Abrir (criar se necessário) para gravação
a	Abrir (criar se necessário) arquivo para acréscimo
r +	Abrir arquivo para leitura e escrita
w +	Criar e abrir arquivo para leitura e escrita
a +	Abrir arquivo para leitura acréscimo
rb	Abrir arquivo binário para leitura
wb	Abrir arquivo binário para escrita
ab	Abrir arquivo binário para acréscimo
rt	Abrir arquivo texto para leitura
wt	Criar arquivo texto para escrita
at	Abrir arquivo texto para acréscimo
r+b	Abrir arquivo binário para leitura e escrita
w+b	Criar arquivo binário para escrita
a+b	Abrir arquivo binário para acréscimo

Valor de retorno:
Se o arquivo foi aberto, retorna um endereço que deve ser atribuído para uma variável de tipo FILE * para uso posterior com as outras funções. Retorna NULL em caso de erro.

```
/*Exemplo de abertura de arquivo*/
#include <stdio.h>

FILE *fp;

main(void)
{
        fp=fopen("c:\\cadastro.dat","a+"); /*Abre arquivo em append*/
        if ( fp == NULL)
        {
```

```
            printf("Erro na Abertura do Arquivo\n");
            exit(-1);
        }
        return(0);
}
```

11.1.2 Função fclose

Fecha um arquivo aberto, tornando-o indisponível para o programa.

*int fclose(FILE *arquivo)*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo a ser fechado.

Valor de retorno:
A função retorna 0 (zero) se o arquivo pode ser fechado ou EOF se algum erro ocorreu.
Alguns arquivos, associados à dispositivos muito utilizados, são definidos e abertos automaticamente pelo C, podendo ser utilizados por qualquer programa sem a necessidade de declaração de variável e chamada à função fopen:

Arquivo	**Descrição**
Stdin	Dispositivo de entrada padrão: normalmente o teclado, mas podem ser redirecionado.
Stdout	Dispositivo de saída padrão: normalmente o vídeo, mas podem ser redirecionado.
Stderr	Dispositivo de saída de erros: o vídeo.

11.2 Funções de acesso seqüencial

O acesso sequencial (ou serial) em arquivos caracteriza-se pelo fato de que cada operação de leitura ou escrita acessa a posição imediatamente seguinte à operação anterior. Cada arquivo possui associado um cursor que indica qual a posição onde será feita o próximo acesso. Quando um arquivo é aberto este cursor está na sua posição inicial (exceto se ele foi aberto com o modo de abertura "a"). Após cada operação de escrita ou leitura, este cursor avança automaticamente para a próxima posição no arquivo. O arquivo possui uma marca de final de arquivo (EOF) que pode ser utilizada para testar se o final foi atingido.

11.2.1 Leitura e escrita de caracteres

As seguintes funções podem ser utilizadas para ler/escrever um caracter por vez de/para um arquivo aberto. Apesar dela efetuarem operações com caracteres, os valres retornados por estas funções, se forem armazenados em variáveis, estas devem ser declaradas com o tipo int.

11.2.1.1 Função fgetc

Lê o próximo caracter de um arquivo aberto.

*int fgetc(FILE *arquivo)*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo que será lido.
Valor de retorno:
Retorna o caracter lido, em caso de leitura bem sucedida, ou EOF, se o final do arquivo foi alcançado.

11.2.1.2 Função fputc

Escreve um caracter em um arquivo aberto.

*int fputc(int c, FILE *arquivo)*

Parâmetros:
• c: O caracter a ser gravado no arquivo.
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo que será escrito.
Valor de retorno:
Retorna o próprio caracter , se ele foi escrito com sucesso , ou EOF, em caso de erro.

```
/* Exemplo do uso das funções de abertura fechamento e leitura de caracteres*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
	FILE *fp;
	char ch;

	fp=fopen("c:\\arquivo.dat","w"); /*Abre o arquivo */
	if ( fp == NULL)
	{
		printf("Erro na Abertura do Arquivo\n");
		exit(-1);
	}
	do
	{
		ch = getchar();
		 putc(ch, fp);
	 } while(ch!='S');
	fclose(fp);
}
```

11.2.2 Leitura e escrita de strings

11.2.2.1 Função fgets

Lê a próxima linha de texto em um arquivo, armazenando-a em uma string.

*char *fgets(char *str, int n, FILE *arquivo)*

Parâmetros:
• str: Variável string que irá receber o conteúdo da linha lida (inclusive o '\n').
• n: Limita o número máximo de caracteres a serem lidos em uma linha em n-1 caracteres. Normalmente este valor é igual ao tamanho da string, ou seja, sizeof(str). Se a linha armazenada no arquivo for maior que este tamanho, é lida apenas a sua parte inicial. O restante será lido na próxima chamada à função fgets.
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo que será lido.
Valor de retorno:
Retorna o endereço do primeiro caracter da string lida, em caso de sucesso, ou NULL, se o final do

arquivo foi alcançado.
Devido à problemas de segurança da função gets, é mais adequado utilizar fgets, lendo do arquivo stdin, no caso de leitura de strings na entrada-padrão: fgets(str, sizeof(str), stdin);

11.2.2.2 Função fputs

Escreve uma string em um arquivo (exceto o '\0').

*int fputs(char *str,FILE *arquivo)*

Parâmetros:
• str: A string que possui o conteúdo a ser escrito.
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo que será escrito.
Valor de retorno:
Retorna EOF, em caso de erro, ou algum valor positivo, se a escrita foi bem sucedida.

11.3 Funções de acesso aleatório

Este método somente pode ser utilizado quando os registros que compõem um arquivo possuem um tamanho fixo e determinado. Cada registro possui associado um número inteiro que indica a sua posição relativa ao início do arquivo. Cada vez que se lê/escreve n bytes no arquivo este cursor é incrementado em n. A diferença, em comparação ao acesso sequencial, é que este cursor pode ser reposicionado pelo programador, através de funções apropriadas, a fim de localizar o dado a ser lido. Normalmente (mas não obrigatóriamente) estas funções são utilizadas para a leitura e escrita de variáveis de estruturas (struct) em um arquivo.

11.3.1 Função fread

Lê um conjunto de bytes de um arquivo e armazena-os na posição de memória indicada no primeiro parâmetro.

*int fread(void *endereco, int tamanho, int num, FILE *arquivo)*

Parâmetros:
• endereco: Endereço da área de memória onde serão armazenados os dados lidos do arquivo.
• tamanho: Tamanho, em bytes, da variável a ser lida.
• num: Número de variáveis a serem lidas em um único acesso. Geralmente 1 (um).
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo que será lido.

Valor de retorno:
A função retorna o número de variáveis realmente lidas e avança o cursor de arquivo tamanho*num bytes.

11.3.2 Função fwrite

Escreve um conjunto de bytes em um arquivo. Se o cursor estiver apontando para uma área já existente do arquivo, então os novos dados irão sobrescrever os anteriores. Se estiver apontando para o final do arquivo, então o tamanho do arquivo será aumentado e os novos dados serão anexados.

*int fwrite(void *buffer, int tamanho, int num, FILE *arquivo)*

Parâmetros:
• endereco: Endereço da área de memória onde estão os dados a serem escrito no arquivo.

• tamanho: Tamanho, em bytes, da variável a ser escrita.
• num: Número de variávies a serem gravadas na mesma operação. Geralmente 1.
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo que será escrito.
Valor de retorno:
A função retorna o número de variáveis realmente gravados e avança o cursor tamanho*num bytes.

11.3.3 Função fseek

Altera a posição do cursor de um arquivo, indicando onde será feito o próximo acesso ao arquivo.

*int fseek(FILE *arquivo, int deslocamento, int onde)*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo cujo cursor será reposicionado.
• deslocamento: Quantidade de bytes que o cursor será ser movimentado. Este valor depende do parâmetro a seguir:
• onde: Indica uma da posições possíveis, relativas a qual a movimentação será feita. Deve-se utilizar uma das seguintes macros:

Valor	Int	Descrição
SEEK_SET	0	Posiciona a partir do início do arquivo
SEEK_CUR	1	Relativo à posição atual
SEEK_END	2	Retrocede do final do arquivo

Valor de retorno:
Retorna 0 se OK, ou EOF, em caso de erro.

11.3.4 Função rewind

Posiciona o cursor no início do arquivo. É identico à fseek(arquivo,0L,SEEK_SET);

*void rewind(FILE *arquivo)*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo cujo cursor será reposicionado.
Valor de retorno:
Esta função não retorna valor.

```
/* Escreve alguns dados nao-caracteres em um arquivo em disco e le de volta*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        FILE *fp;
        double d = 12.23;
        int i = 101;
        long l = 123023L;

        fp=fopen("c:\\arquivo.dat","wb+"); /*Abre o arquivo e permite inserção de valores
binários*/
        if ( fp == NULL)
```

```
        {
                printf("Erro na Abertura do Arquivo\n");
                exit(-1);
        }
        fwrite(&d, sizeof(double), 1, fp);
        fwrite(&i, sizeof(int), 1, fp);
        fwrite(&l, sizeof(long), 1, fp);

        rewind(fp);

        fread(&d, sizeof(double), 1, fp);
        fread(&i, sizeof(int), 1, fp);
        fread(&l, sizeof(long), 1, fp);

        printf("%f %d %ld", d, i, l);
        fclose(fp);
}
```

11.3.5 Função ftell

Obtêm a posição atual do cursor de arquivo, isto é, em qual posição será feita a próxima operação de
escrita ou leitura.

*int ftell(FILE *arquivo)*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo cuja posição do cursor será obtida.
Valor de retorno:
Retorna a posição atual do cursor, na forma de um número inteiro positivo, ou EOF no caso de erro.

11.3.6 Função feof

Testa se o final do arquivo foi atingido.

*int feof(FILE *arquivo);*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo cujo final será testado.
Valor de retorno:
Retorna um valor diferente de zero (verdadeiro) se o programa tentou ultrapassar o final do arquivo, ou zero (falso) caso contrário.

11.4 Removendo arquivos

A função **remove()** apaga o arquivo especificado. Seu protótipo é:

*int remove(const char *arquivo);*

Parâmetros:
• arquivo: A variável que recebeu o valor de retorno da função fopen, correspondente ao arquivo.
Valor de retorno:
Ela retornará 0, caso seja bem-sucedida, e um valor diferente de zero, caso contrário.

```
/*Exemplo das funções já vistas dando destaque para o fseek e uso de estruturas, utilizando os primeiros passos para a criação de um sistema de arquivos*/
#include <stdio.h>
#include <stdlib.h>

struct Exemplo{
        int  x;
        int  y;
        char  a;
} Reg;

int main()
{
        FILE *fp;
        int erro=0;
        fp=fopen("c:\\teste.dat","a+");
        if ( fp==0 )
        {
                printf("Erro na Abertura do Arquivo\n");
                exit(-1);
        }
        /* Grava Primeiro Registro no Arquivo */
        Reg.x=1;
        Reg.y=5;
        Reg.a='C';
        fwrite(&Reg,sizeof (struct Exemplo),1,fp);

        /* Grava Segundo Registro no Arquivo */
        Reg.x=3;
        Reg.y=6;
        Reg.a='B';
        fwrite(&Reg,sizeof (struct Exemplo),1,fp);

        erro=fseek(fp,-sizeof(struct Exemplo),2);

        fread(&Reg,sizeof (struct Exemplo),1,fp);

        printf("\nX=%d\nY=%d\nA=%c\n",Reg.x,Reg.y,Reg.a);

        getch();
}
```

12 Funções diversas

Como visto nos capitulos iniciais a linguagem C oferece diversas bibliotecas de funções já implementadas, ou seja, prontas para serem usadas. As bibliotecas em C funcionam quase da mesma forma que as units do Delphi, ou seja, para utilizar as funções deve-se declarar elas no cabeçalho do programa.

12.1 stdlib.h

Função exit: *void exit(int codigo_retorno);*
Esta função provoca a conclusão imediata e anormal do programa.

Retorno igual a 0 ira informar ao sistema operacional um término de programa normal. Qualquer outro resultado irá indicar término com erro.

Função abs: *int abs(int num);*
Esta função converte os números digitados pelo usuário em seus valores absolutos.

Função div: *div_t div(int numerador,int denominador);*
A função div() retorna o resto da divisão. Esta função pode ser utilizada para determinar se um numero é par ou não. Utilize no numerador o número que se deseja verificar e no denominador utiliza 2.

```
/*Exemplo utilizando div*/
#include <stdio.h>
#include <stdlib.h>

main()
{
	system("cls");
	int a;
	div_t num;
	printf("Entre com o numerador: ");
	scanf("%d",&a);
	num = div(a,2);
	if (num.rem == 0)
		printf("Numero par");
	else
		printf("Numero impar");
	getch();
}
```

Função atof: *double atof (const char *str);*
Esta função converte a string apontada por str em um valor double e retorna o resultado. A string deve conter um número em ponto flutuante válido.

Função atoi: *int atoi (const char *str);*
Esta função converte a string apontada por str em um valor int e retorna o resultado. A string deve conter um número inteiro válido.

Função atol: *int atol (const char *str);*
Esta função converte a string apontada por str em um valor long int e retorna o resultado. A string deve conter um inteiro longo válido.

Função qsort: *void qsort (void *buf, size_t num, size_t size, int(*compare)(const void *, const void *));*
Esta função ordena a matriz apontada por *buf*, usando quicksort.
A função apontada por *compare* é usada para comparar um elemento da matriz com a chave. A forma compare deve ser:

*int compare (const void *arg1, const void arge));*

A função pode ter o nome que vc quiser. No entanto ela deve devolver os seguintes valores:
Se arg1 é menor que arg2, devolve menor que zero.
Se arg1 é igual a arg2, devolve zero.
Se arg1 é maior que arg2 maior que zero.

```
/*Exemplo utilizando qsort*/
```

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

int num[10] =
{
        1, 3, 6, 5, 8, 7, 9, 6, 2, 0
};

int comp(const void *, const void *);

int main(void)
{
        int i;

        printf("Matriz original: ");
        for(i=0; i<10; i++) printf("%d ", num[i]);

        qsort(num, 10, sizeof(int), comp);

        printf("\nMatriz ordenada: ");
        for(i=0; i<10; i++) printf("%d ", num[i]);
}


/* compara os inteiro */
comp(const void *i, const void *j)
{
        return *(int *)i - *(int *)j;
}
```

Função rand: *int rand(void);*
Esta função gera uma seqüência de números randômicos.

```
/*Exemplo utilizando rand*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
        int i;

        for(i=0; i<10; i++)
        printf("%d ", rand());
}
```

Função srand: *int srand(unsigned seed);*
Esta função estabelece um ponto de partida para seqüência randômica gerada por **rand()**.
Esta função é geralmente usada para permitir que os programas usem seqüências diferentes a cada chamada de rand.

Função system: *int system(const char *str);*
Esta função passa a string apontada pó str como um comando para o processador de comandos do sistema operacional.

```
/*Exemplo utilizando system*/
#include <stdlib.h>

int main(void)
{
  system("dir");
  getch();
}
```

12.2 ctype.h

Função isalnum: *int isalnum (int ch);*
Esta função devolve um valor diferente de zero se o argumento for uma letra ou um dígito.

Função isalpha: *int isalpha (int ch);*
Esta função devolve um valor diferente de zero se ch for uma letra do alfabeto, caso contrário, devolverá zero.

Função ispunct: *int ispunct (int ch);*
Esta função devolve um valor diferente de zero se ch é um caracter de pontuação, caso contrário, devolverá zero.

Função isspace: *int isspace (int ch);*
Esta função devolve um valor diferente de zero se ch for um espaço, caso contrário, devolverá zero.

Função islower: *int islower (int ch);*
Esta função devolve um valor diferente de zero se ch é uma letra minúscula, caso contrário, devolve zero.

Função isupper: *int isupper (int ch);*
Esta função devolve um valor diferente de zero se ch é uma letra maiúscula, caso contrário, devolve zero.

Função tolower: *int tolower (int ch);*
Esta função devolve o equivalente minúsculo de ch se ch é uma letra caso contrário, ch é devolvido sem alteração.

Função toupper: *int toupper (int ch);*
Esta função devolve o equivalente maiúsculo de ch se ch é uma letra caso contrário, ch é devolvido sem alteração.

Função isupper: *int isupper (int ch);*
Esta função devolve um valor diferente de zero se ch é uma letra maiúscula, caso contrário, devolve zero.

12.3 string.h

Função strcat: *char *strcat(char *str1, const char *str2);*
Esta função concatena uma cópia de str2 em str1 com um nulo.

```
/*Este exemplo concatena as strings s1 e s2. Suponha que seja digitado alo e aqui o resultado será
aquialo*/
#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);

  strcat(s2, s1);
  printf(s2);

  system("pause");
}
```

Função strcmp: *int *strcmp(char *str1, const char *str2);*
Esta função compara duas strings e devolve um inteiro baseado no resultado.
Menor que zero: str1 é menor que str2.
Zero: str1 é igual a str2.
Maior que zero: str1 é maior que str2.

Função strcpy: *int *strcpy(char *str1, const char *str2);*
Esta função copia o conteúdo de str2 em str1. str2 deve ser um ponteiro para uma string terminada com um nulo.

12.4 math.h

Função cos: *double cos(double arg);*
Esta função devolve o co-seno de arg. O valor de arg deve estar em radianos.

Função sin: *double sin(double arg);*
Esta função devolve o seno de arg. O valor de arg deve estar em radianos.

Função tan: *double tant(double num);*
Esta função devolve a tangente de arg.

Função pow: *double pow(double base, doublé exp);*
Esta função devolve base elevada a potência exp.

Função sqrt: *double sqrt(double num);*
Esta função devolve a raiz quadrada de num.

Referências

HTTP.**Apostilas diversas encontradas, portal onde você encontra tudo sobre C e C++.** Disponível em www.portalc.nip.net/. Acesso em 10 de Junho de 2003.
HTTP.**Apostila do Curso de Linguagem C da UFMG**. Disponível em www.ead.eee.ufmg.br/cursos/C/. Acesso em 25 de Junho de 2004.
HTTP.**Introdução a linguagem C,** Gacli – Centro de Computação – UNICAMP. Disponível em www.portalc.nip.net/. Acesso em 25 de Junho de 2004.
HTTP.Apostila Linguagem C. Disponível em http://vitoria.upf.br/~brusso/progc. Acesso em 10 de Junho de 2003.