Aspectos básicos de linguagem C

Considerações gerais

Muitas linguagens foram desenvolvidas e por muitos anos utilizadas com diferentes objetivos e características, tais como: Fortran, Cobol, Basic, Algols, Pascal e etc. **Mas o que é C?** C é o nome de uma linguagem atualmente utilizada em diferentes áreas e propósitos. Faz parte hoje de uma linguagem considerada avançada, desenvolvida nos laboratórios Bell nos anos 70.

A definição formal da linguagem pode ser encontrada no livro "The C Programming Language" de Brian W. Kernighan e Dennis M. Ritchie (os pais da linguagem). Nos anos 80 iniciou-se um trabalho de criação de um padrão chamado C ANSI (*American National Standardization Institute*).

É uma *linguagem de nível médio*, pois pode-se trabalhar em um nível próximo ao da máquina ou como uma linguagem de alto nível como outras existentes.

Com o C podemos escrever programas concisos, organizados e de fácil entendimento, mas infelizmente a falta de disciplina pode gerar programas mal escritos, difíceis de serem lidos e compreendidos. Não se deve esquecer que C é uma linguagem para programadores, pois impõe poucas restrições ao que pode ser feito. O C é amigável e estruturado para encorajar bons hábitos de programação; cabe ao programador exercitar esses hábitos.

A necessidade de escrever programas, que façam uso de recursos da linguagem de máquina de uma forma mais simples e portátil, fez com que a principal utilização do C fosse a reescrita do sistema operacional UNIX. Sua indicação é principalmente no desenvolvimento de programas, tais como: *compiladores, interpretadores, editores de texto; banco de dados. Computação gráfica, manipulação e processamento de imagens, controle de processos, etc.*

Principais características da linguagem C a serem ponderadas:

- Portabilidade
- Geração de códigos executáveis compactos e rápidos
- Interação com o sistema operacional
- Facilidade de uso (por meio de ambientes como o Code::Blocks)
- Linguagem estruturada
- Confiabilidade
- Simplicidade

1. Elementos básicos

1.1 Identificadores

São utilizados para dar nomes a constantes, variáveis, funções e vários objetos definidos pelo usuário. As regras para formação desses nomes são:

1) Todo identificador deve iniciar por uma letra (a..z ou A..Z) ou um sublinhado
2) Não pode conter símbolos especiais. Após o primeiro caracter pode ser utilizado: letras, sublinhados e/ou dígitos.
3) Utiliza-se identificadores de, no máximo, 32 caracteres por estes serem significativos.
4) Não pode ser palavra reservada e nem nome de funções de bibliotecas.

Obs: letras maiúsculas e minúsculas são tratadas de forma diferente.

1.2 Tipos de dados básicos

Tipo	Número de bytes	Escala
char	1	-128 a 127
int	4[1]	-2147483648 a 2147483647[1]
float	4	aprox. 2^{-149} a aprox.. 2^{128} (+-)
double	8	aprox. 2^{-1074} a aprox.. 2^{1024} (+-)
void	0	sem valor

1.3 Modificadores

Tipo	Número de bytes	Escala
unsigned char	1	0 a 255
unsigned int	4[1]	0 a 4294967295[1]
short int	2	-32768 a 32767
unsigned short int	2	0 a 65535
long int	4	-2147483648 a 2147483647
unsigned long int	4	0 a 4294967295
long long int	8	-2^{63} a $2^{63}-1$
unsigned long long int	8	0 a $2^{64}-1$
long double	10	aprox. 2^{-16445} a aprox.. 2^{16382} (+-)

Observações:

1) O modificador *signed* eventualmente pode ser utilizado, porém o seu uso equivale a utilizar um tipo sem qualquer modificador.
2) A palavra *int* pode ser omitida. Ex: *unsigned long int* ⇔ *unsigned long*

[1] O tamanho do tipo *int* é dependente da plataforma sobre a qual o programa é compilado. Por exemplo, para a compilação na plataforma Windows 8 em modo 32 bits, um dado do tipo *int* ocupa quatro bytes de tamanho

1.4 Declaração de variáveis

A forma geral para declaração de uma variável é:

tipo_da_variável lista_de_variáveis;

onde *tipo_da_variável* é um tipo válido em C (Seções 1.2 e 1.3) e *lista_de_variáveis* pode ser um ou mais nomes de identificadores separados por virgula.
Exemplos:

```
int f, i, k;      /* todas variáveis do tipo int */[2]
float a, A, b;    /* todas variáveis do tipo float */
```

1.5 Constantes

Em C, constantes são valores fixos que não podem ser alterados por um programa.

1.5.1 Constantes em base decimal

1) Constantes numéricas inteiras: podem ser atribuídas a variáveis dos tipos *char* e *int*, modificados ou não, dependendo do valor da constante e da faixa de valores aceita pela variável.

Exemplos: 345 10 0 5000000

2) Constantes numéricas não inteiras: podem ser atribuídas a variáveis dos tipos *float* e *double*, modificados ou não, dependendo do valor da constante e da faixa de valores aceita pela variável.

Exemplos: -56.897 1.2E+5

3) Constantes em forma de caracter: podem ser atribuídas a variáveis do tipo *char*, modificadas ou não. O valor da constante é igual ao valor numérico da tabela ASCII[3] do caracter representado entre ‘ ‘ (comumente chamados de “plicas”)

1.5.2 Constantes em bases hexadecimal e octal[4]

Constantes em base hexadecimal iniciam com 0x, ao passo que constantes em base octal iniciam com um 0.

---

[2] /* e */ delimitam um comentário textual, que não é compilado mas que auxilia o programador na documentação do seu código. Note-se que existe uma versão alternativa para delimitar um comentário até o final da linha, usando //, porém esta versão é padrão C++ e não deve ser utilizada para compilação de código ANSI C (C padrão)

[3] A tabela ASCII é uma tabela padronizada que relaciona um conjunto de caracteres a valores numéricos entre 0 e 255. Por exemplo, o caracter correspondente ao dígito ‘9’ corresponde ao código ASCII 57.

[4] Para maiores informações sobre números octais e hexadecimais, consultar o texto sobre Bases Numéricas

Exemplos: 0xAB (hexadecimal) 016 (octal)

1.5.3 Constantes em forma de sequência de caracteres (strings)

São representadas entre aspas.

Exemplo: "Esta é uma constante em forma de string".

1.5.4 Caracteres de controle da linguagem C

Estes caracteres devem ser representados como caracteres alfanuméricos (entre ' ') ou como conteúdo de uma string

Código	Significado
\a	sinal audível
\b	retrocesso do cursor
\f	alimentação de formulário
\n	nova linha
\r	retorno de carro
\t	tabulação horizontal
\'	aspas
\'	apóstrofo
\0	nulo (zero)
****	barra invertida
\v	tabulação vertical
\a	sinal sonoro
\N	constante octal (onde N é um octal)
\xN	constante hexadecimal (onde N é um hexadecimal)

1.6 Instruções

Uma instrução em linguagem C é uma expressão seguida de um ponto e vírgula. Pode ser uma atribuição, uma chamada de função, um teste de desvio ou um teste de laço.

Exemplo de instrução de atribuição: `x = 12;`

onde o sinal de igual (=) é o *operador de atribuição*. Note-se que o operando do lado esquerdo do operador de atribuição é sempre uma variável, e que o operando do lado direito deve ser de um tipo de dado compatível com o tipo da variável.

1.7 Operadores

1.7.1 Operadores aritméticos

Adição	**+**
Subtração	**-**
Divisão	**/**
Multiplicação	*****

Resto	%

Observações:

1) Todos os operadores são definidos para os tipos inteiros e não inteiros, exceto o operador resto (%) que não é definido para variáveis dos tipos não inteiros.
2) Para qualquer tipo inteiro, a adição de um ao maior número da faixa daquele tipo produz o menor número da faixa. Os erros de estouro nem sempre são detectados, cabendo ao programador tomar cuidado ao dimensionar as variáveis do programa para que eles não ocorram.

Exemplo:

```
unsigned char x;
x = 255;
x = x + 1;  /* x deveria assumir 256, no entanto estoura a faixa e  retorna para o menor valor que é 0 */
```

1.7.2 Operadores relacionais

Menor que	<
Maior que	>
Menor ou igual	< =
Maior ou igual	> =
Igualdade	= =
Desigualdade	! =

Observações:

1) Todas as operações relacionais tem como resultado um inteiro representando um valor lógico (1 = true e 0 = false).
2) Não confundir o operador de atribuição (=) com o operador de igualdade (= =).

1.7.3 Operadores lógicos

e (conjunção)	&&
ou (disjunção)	||
não (negação)	!

Os operadores lógicos podem receber qualquer valor de operando, porém os valores diferentes de zero são sempre interpretados como “true” (verdadeiro) e os iguais a zero são interpretados como “false” (falso). O resultado de uma operação lógica é sempre um valor lógico.

Tabela da verdade

p	Q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1

1	0	0	1	0
1	1	1	1	0

1.7.4 Operadores de atribuição combinados

+= -= *= /=

Exemplos:

```
a += b;      /* a = a + b; */
a -= b;      /* a = a - b; */
a *= b;      /* a = a * b; */
a /= b;      /* a = a / b; */
```

Observações:

1) Todos os operadores de atribuição atribuem o resultado de uma expressão a uma variável

Se o tipo do lado esquerdo não for o mesmo do lado direito, o tipo do lado direito será convertido para o tipo do lado esquerdo. Isto pode causar a perda de precisão em alguns tipos de dados e deve ser levado a sério pelo programador.

1.7.5 Operadores pós-fixados e pré-fixados

Operador	Significado
++variavel	**incrementa a variável antes de usar o seu valor**
Variável++	**incrementa a variável depois de usar o seu valor**
--variavel	**decrementa a variável antes de usar o seu valor**
variavel--	**decrementa a variável depois de usar o seu valor**

Exemplos:

```
int a, b, c;
a = 6;
b = ++a;     /* a recebe 7 e depois b também recebe 7*/
c = a++;     /* c recebe 7 e depois a recebe 8 */
```

1.7.6 Operadores em nível de bit

Deslocamento à esquerda (shift left)	**<<**
Deslocamento à direita (shift right)	**>>**
e (and)	**&**
ou (or)	**|**
ou exclusivo (xor)	**^**
não (not)	**~**

Para uma discussão mais aprofundada sobre o uso de bits e números binários, consultar o texto sobre Bases Numéricas

1.7.7 Operadores de endereço

& - retorna o endereço da variável
* - retorna o conteúdo do endereço armazenado em uma variável do tipo ponteiro

Para saber mais sobre operadores de endereço, consultar o texto sobre Ponteiros.

1.7.8 Outros operadores

sizeof(*operando*) – fornece o tamanho em bytes do seu operando
Ex:

```
int x;
float y;
char c;
x= sizeof(int);   /* fornece o tamanho do tipo int (2 bytes) */
x= sizeof(y);     /* fornece o tamanho da variável y (4 bytes) */
x= sizeof(c);     /* fornece o tamanho da variável c (1 byte) */
```

1.8 Expressões

Operadores, constantes, variáveis e funções constituem expressões. As principais regras algébricas são consideradas nas expressões. Alguns aspectos das expressões são específicos da linguagem C e são explicados a seguir.

1.8.1 Conversão automática de tipos

Quando constantes, variáveis e funções de tipos diferentes são misturados em uma expressão, elas são todas convertidas para o tipo do *operando maior*. Isto é feito operação a operação, de acordo com as seguintes regras:

1) Todos os operandos dos tipos *char* e *short int* são convertidos para *int*. Todos os operandos do tipo *float* são convertidos para *double*.
2) Para todos os pares de operandos envolvidos em uma operação, se um deles é *long double* o outro operando é convertido para um *long double*. Se um é *double*, o outro é convertido para *double*. Se um é *long*, o outro é convertido para *long*. Se um é *unsigned,* o outro é convertido para *unsigned*.
3) Caso todos os operandos sejam inteiros, o resultado da operação é inteiro. Se pelo menos um dos operandos for não inteiro, o resultado é não inteiro.

Ex:

```
float x, res;
char c;
...
res = x/c;  /* o valor de x/c é convertido para um float, embora
c seja originalmente um char */
```

1.8.2 Conversão explícita de tipos (type casts)

É possível forçar uma expressão a ser de um tipo específico, sem no entanto mudar os tipos das variáveis envolvidas nesta expressão. A esse tipo de operação chama-se conversão explícita de tipo, ou *type cast*.

A forma geral de um *type cast* é:

(tipo) expressão;

onde *tipo* é um dos tipos de dado padrão da linguagem C.

As operações de *type cast* são muito úteis em expressões nas quais alguma operação resulta em perda de precisão devido ao tipo das variáveis ou constantes envolvidas. Por exemplo:

```
float res;
int op1,op2;
op1 = 3;
op2 = 2;
res = op1 / op2;  /* res recebe 1, já que op1 e op2 são ambos
                     números do tipo int e o resultado da sua
                     divisão também é int */
res = (float)op1 / op2; /* res recebe 1.5, já que o type cast
                           forçou o operando op1 a ser um float
                           nesta operação. O resultado da divisão,
                           por consequência, também é float */
```

1.8.3 Espaçamento e parênteses

Podemos colocar espaços em uma expressão para torná-la mais legível. O uso de parênteses redundantes ou adicionais não causará erros ou diminuirá a velocidade de execução da expressão.
Ex:

```
a=b/9.67-56.89*x-34.7;
a = (b / 9.67) - (56.89 * x) - 34.7;      /* equivalente */
```

2. Estrutura de um programa em C

Uma particularidade interessante no programa C é seu aspecto modular e funcional, em que o próprio programa principal é uma função. Esta forma de apresentação da linguagem facilita o desenvolvimento de programas, pois permite o emprego de formas estruturadas e modulares encontradas em outras linguagens.

A estrutura de um programa em C possui os seguintes elementos, sendo que aqueles delimitados por colchetes são opcionais:

[definições de pré-processamento]
[definições de tipo]
[declarações de variáveis globais]
[protótipos de funções]
[funções]

```
main ( )
{
        /* definições de variáveis  */
        /* corpo da função principal, com declarações de suas variáveis,
        seus comandos e funções */
}
```

Definições de pré-processamento são comandos interpretados pelo compilador, em tempo de compilação, que dizem respeito a operações realizadas pelo compilador para geração de código. Geralmente iniciam com uma cerquilha (#) e não são comandos da linguagem C, por isso não serão tratados aqui com maiores detalhes.

Ex:

```
      #include <stdio.h>      /* comando de pré-processador, utilizado
para indicar ao compilador que ele deve ´colar´ as definições do
arquivo stdio.h  neste arquivo antes de compilá-lo */
```

Definições de tipos são definições de estruturas ou tipos de dados especiais, introduzidos pelo usuário para facilitar a manipulação de dados pelo programa. Também não serão tratados aqui em maiores detalhes.

Declarações de variáveis globais são feitas quando é necessário utilizar variáveis globais no programa. O conceito de variável global e as vantagens e desvantagens do seu uso dizem respeito à modularização de um programa em C (consultar o material sobre modularização e funções).

Protótipos de funções e *funções* também dizem respeito a questões de modularização.

main() é a função principal de um programa em C, contendo o código que será inicialmente executado quando o programa em si for executado. **Todo** programa em C deve conter a função *main(),* do contrário será gerado um erro durante o processo de geração do programa (mais especificamente, na etapa de ligação).

3. Funções básicas de E/S

Esta seção descreve algumas das funções básicas de E/S, que serão utilizadas inicialmente para prover o programador de um canal de entrada de dados via teclado e um canal de saída de dados via monitor.

3.1 Função printf( ) (biblioteca stdio.h)

A função *printf( )* é basicamente utilizada para enviar informações ao console de texto, ou seja, imprimir informações. O seu protótipo é o seguinte:

printf(string de dados e formato, var1, var2,..., varN);

onde *string de dados e formato* é formada por dados literais a serem exibidos no monitor (por exemplo, um texto qualquer) mais um conjunto opcional de *especificadores de formato*

(indicados pelo símbolo % e um conjunto de caracteres). Estes especificadores determinarão de que forma o conteúdo dos argumentos *var1* a *varN* será exibido.

var1 a *varN* indicam, por sua vez, os argumentos (variáveis ou constantes) cujos valores serão exibidos no local e no formato determinado pelos especificadores de formato, dentro da string de dados e formato. O número N deve ser igual ao número de especificadores de formato fornecidos.

Especificadores de formato mais utilizados:

%c	caracteres simples (tipo *char*)
%d	inteiro (tipo *int*)
%e	notação científica
%f	ponto flutuante (tipo *float*)
%g	%e ou %f (mais curto)
%o	octal
%s	string
%u	inteiro sem sinal
%x	hexadecimal
%lf	tipo *double*
%u	inteiro não sinalizado (tipo *unsigned int*)
%ld	tipo *long int*

Exemplos:

1)

```
int n = 15;
printf("O valor de n eh %d", n);

/* exibe ´O valor de n eh 15´. Note-se que todo o conteúdo da
string de dados e formato é exibido literalmente, com exceção do
especificador %d, que é substituído pelo valor em formato
inteiro da variável n  */
```

2)

```
char carac = ´A´;
float num = 3.16;
printf("A letra eh %c e o numero eh %f", carac, num);

/* exibe ´A letra eh A e o numero eh 3.16´. Neste caso, o
especificador %c (primeiro da string) é substituído pelo valor
da variável carac e o especificador %f é substituído pelo valor
da variável num. Note-se que os tipos dos especificadores e das
variáveis são compatíveis */
```

3.2 Função scanf() (biblioteca stdio.h)

A função *scanf* é utilizada para receber dados de uma entrada de dados padrão. Consideraremos, para fins de simplificação, que essa entrada padrão é sempre o teclado. O protótipo de *scanf* é o seguinte:

int scanf (string de formato, &var1, &var2, ..., &varN);

onde a *string de formato* contém os especificadores de formato na sequência e relativos a cada um dos dados que se pretende receber. Para uma lista dos especificadores de formato mais utilizados, ver seção 3.1.

var1 a *varN* identificam as variáveis nas quais serão armazenados os valores recebidos por *scanf*, na mesma ordem determinada pelos especificadores de formato. O número N deve ser igual ao número de especificadores de formato fornecidos.

O tipo *int* antes do protótipo da função *scanf* indica que ela retorna como resultado um valor inteiro, representando a quantidade de dados que a função conseguiu converter (obter da entrada padrão e armazenar nas variáveis da lista) com sucesso. Ou seja, o valor de retorno deve, idealmente, ser igual à quantidade de variáveis da lista e, por consequência, à quantidade de especificadores de formato na *string de dados e formato.*

IMPORTANTE: o operador de endereço (&) DEVE obrigatoriamente ser utilizado diante dos identificadores das variáveis, do contrário ocorre um erro. Para maiores detalhes, consultar a teoria sobre ponteiros.

Exemplos:

1)

```
int t;
printf("Digite um inteiro: ");
scanf("%d", &t);  /* aguarda a digitação de um número do tipo
                     int. O número digitado é armazenado na
                     variável t quando o usuário digita ENTER */
```

2)

```
char carac1;
int i;
printf("Digite um caracter e um int, separados por vírgula: ");
scanf("%c, %d", &carac1, &i);

/* neste caso, os especificadores de formato %c e %d estão
separados por vírgula, o que significa que o usuário deve
digitar os valores também separados por vírgula e na ordem correta */
```

3.3 Função getch() (biblioteca conio.h)

A função *getch* é utilizada, basicamente, para esperar o pressionamento de uma tecla pelo usuário. A tecla pressionada pode ser capturada através do valor de retorno da função (para maiores detalhes sobre valor de retorno, consultar a teoria sobre funções).

Pelo fato de interromper a execução até o pressionamento de uma tecla, a função *getch* pode ser utilizada no final de um programa de console para permitir que o usuário visualize o resultado do programa antes que a sua janela se feche.

Exemplo:

```
printf("Estou mostrando uma frase\n");
printf("Digite qualquer tecla para sair do programa");
getch();    /* aguarda aqui até uma tecla ser pressionada */
```

```
/* fim do programa */
```

Observações:

- a função *getche* funciona de forma semelhante, porém exibe na tela o caracter digitado (o nome significa "get char with echo").
- *conio* não é uma biblioteca padrão da linguagem C, portanto não é implementada por todas as ferramentas de desenvolvimento em linguagem C.

3.4 Função clrscr() (biblioteca conio.h)

A função *clrscr* é utilizada para limpar a tela (o nome significa "clear screen").

4. Estruturas de controle

4.1 Comando simples

Uma linha de comando em C sempre termina com um ponto e vírgula (;)

Exemplos:

```
x = 443.7;
a = b + c;
printf("Exemplo");
```

4.2 Bloco de comandos

Utiliza-se chaves ({ }) para delimitar blocos de comando em um programa em C. Estes são mais utilizados no agrupamento de instruções para execução pelas cláusulas das estruturas condicionais e de repetição.

4.3 Estruturas condicionais

4.3.1 Estrutura if-else

Formato:

```
if ( condição )
{
        bloco de comandos 1
}
else
{
        bloco de comandos 2
}
```

condição é qualquer expressão que possa ser avaliada com o valor verdadeiro (“true”) ou falso (“false”). No caso de expressões que possuam um valor númerico ao invés de um valor lógico, se o valor é diferente de zero a expressão é avaliada com valor lógico “true”, do contrário é avaliada com o valor lógico “false”.

Caso a condição possua um valor lógico “true”, *bloco de comandos 1* é executado. Se o valor lógico da condição for “false”, *bloco de comandos 2* é executado. Para qualquer um dos blocos, se este for formado por um único comando as chaves são opcionais.

A estrutura *if-else* é semelhante a uma estrutura condicional composta, em que um ou outro bloco de comandos é executado; a cláusula *else*, no entanto, é opcional, e se for omitida a estrutura passa a funcionar como uma estrutura condicional simples, em que um bloco de comandos (no caso, o bloco 1) somente é executado se a condição for verdadeira.

Exemplos:

1)

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

if (num < 0)      /*testa se num é menor que zero */
{
      /* bloco de comandos executado se a condição é verdadeira.
            Neste caso, como printf é um único comando as chaves
      poderiam ser omitidas  */

      printf("\nO número é menor que zero");
}
else
{
      /* bloco de comandos executado se a condição é falsa.
      Neste caso, como printf é um único comando as chaves
      poderiam ser omitidas  */

      printf("\nO número é maior que zero");
}
```

2)

```
if ((a == 2) && (b == 5))  /* condição com operação lógica */
      printf("\nCondição satisfeita");  /* bloco de comandos */

getch();    /* esta instrução não faz parte da estrutura
            condicional, logo é sempre executada */
```

3)

```
if (m == 3)
{
      if ((a >=1) && (a <= 31))     /* este if faz parte do bloco
                                    de comandos do if anterior */
      {
            printf("Data OK");
      }
      else        /* este else é do if mais proximo (que faz
                  parte do bloco de comandos)   */
      {
```

```
                printf("Data inválida");
        }
}
```

4.3.2 Estrutura switch

Formato:

```
switch (expressão)
{
        case valor1:
                seq. de comandos 1
                break;
        case valor2:
                seq. de comandos 2
                break;
        . . .
        case valorN:
                seq. de comandos N
                break;
        default:
                seq. padrão
}
```

O comando *switch* avalia *expressão* e compara sucessivamente com uma lista de constantes *valor1* a *valorN* (menos constantes strings). Quando encontra uma correspondência entre o valor da expressão e o valor da constante, salta para a cláusula *case* correspondente e executa a sequência de comandos associada até encontrar um comando *break*, saindo em seguida da estrutura.

A cláusula *default* é executada se nenhuma correspondência for encontrada. Esta cláusula é opcional e, se não estiver presente, nenhuma ação será realizada se todas as correspondências falharem. Ë usada normalmente para direcionar qualquer final livre que possa ficar pendente na declaração *switch*.

OBSERVAÇÕES:

- se o comando *break* for esquecido ao final de uma sequência de comandos, a execução continuará pela próxima declaração *case* até que um *break* ou o final do *switch* seja encontrado, o que normalmente é indesejado.
- nunca duas constantes *case* no mesmo *switch* podem ter valores iguais.
- uma declaração *switch* é mais eficiente do que um encadeamento de *if-else*, além do que pode ser escrito de forma muito mais "elegante".
- *valor1* a *valorN* DEVEM ser valores constantes.

Exemplos:

1)

```
int dia;
printf("Digite um dia da semana, de 1 a 7");
```

```
scanf("%d", &dia);
switch(dia) /* testa o valor da variável dia */
{
      case 1:
            printf("Domingo");
            break;
      case 2:
            printf("Segunda");
            break;
      case 3:
            printf("Terça");
            break;
      case 4:
            printf("Quarta");
            break;
      case 5:
            printf("Quinta");
            break;
      case 6:
            printf("Sexta");
            break;
      case 7:
            printf("Sábado");
            break;
      default:
            printf("Este dia não existe"); /* só entra aqui se o
      usuário não digitar um dia entre 1 e 7  */
            break;
}
```

4.4 Estruturas de repetição

4.4.1 Estrutura while

Formato:

while (condição)
{
    sequência de comandos
}

O comando *while* avalia o valor lógico de *condição;* se o valor lógico for verdadeiro (*true*) a *sequência de comandos* é executada, caso contrário a execução do programa continua após a estrutura *while*. Caso a sequência de comandos seja formada por um único comando, o uso das chaves é opcional.

Após a execução da sequência de comandos, o valor lógico de *condição* é reavaliado e, se continuar sendo verdadeiro (*true*), a sequência de comandos é executada novamente. Este comportamento se repete até que o valor lógico da *condição* seja falso (*false*), quando a execução da estrutura *while* é interrompida e continua na instrução seguinte.

Cada uma das execuções da sequência de comandos é chamada de *iteração* do laço. No caso da estrutura *while* o número de iterações pode variar de 0 até N, sendo N o número da iteração após a qual o teste da condição resulta em um valor lógico falso.

OBSERVAÇÃO: caso a condição seja verdadeira no primeiro teste e a sequência de comandos seja executada, é necessário que esta torne a condição falsa em algum momento; do contrário, a condição sempre será reavaliada como verdadeira e a sequência de comandos será executada em um número infinito de iterações[5].

Exemplo:

```
int x = 0;

/* imprime os valores de x de 0 até 9
o valor 10 não é impresso porque, ao testar a condição para
x igual a 10, o valor lógico é falso e a execução do while
é interrompida  */

while (x < 10)
{
      printf("\nx = %d", x);
      x++;              /* faz a condição tornar-se falsa
                           em algum momento */
}
```

4.4.2 Estrutura do – while

Formato:

do {

sequência de comandos

} while (condição);

A *sequência de comandos* sempre é executada inicialmente em uma estrutura *do-while*. Após a sua execução, o valor lógico da *condição* é avaliado, e se for verdadeiro (*true*) a sequência de comandos é executada novamente. O ciclo se repete até que o valor lógico da condição seja falso (*false*), quando a execução continua na instrução seguinte à estrutura *do-while*. Caso a sequência de comandos seja formada por um único comando, o uso das chaves é opcional.

Diferentemente do que ocorre na estrutura *while,* na estrutura *do-while* o número de iterações varia entre 1 e N, onde N é o número da iteração após a qual o teste da condição resulta em um valor lógico falso.

OBSERVAÇÃO: assim como na estrutura *while,* caso a condição seja verdadeira no primeiro teste é necessário que a sequência de comandos torne a condição falsa em algum momento.

Exemplo:

```
int num;

do {

      printf("Digite um número de 1 a 9: ")
```

[5] Estamos considerando programas executados "linearmente", ou seja, sem a ocorrência de eventos assíncronos (p. ex., interrupções).

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

} while (!((num >=1) && (num <=9)));  /* nesse caso, a obtenção
                                         do valor de num via
                                 scanf pode  tornar a condição falsa */
```

4.4.3 Estrutura for

Formato:

for (inicialização; condição; incremento)
{
    sequência de comandos
}

A *inicialização* é executada uma única vez, no início da execução da estrutura for, e normalmente é uma atribuição utilizada para inicializar alguma variável de controle do laço.

Após a inicialização, o valor lógico da *condição* é testado. Se for verdadeiro (*true*), a *sequência de comandos* é executada, do contrário a execução continua após a estrutura *for*. Ao final da execução da sequência de comandos, o comando correspondente ao *incremento* é executado, e a condição volta a ser testada. O ciclo se repete até que o teste da condição resulte em um valor lógico falso (*false*), quando então a execução prossegue após a estrutura *for.* Caso a sequência de comandos seja formada por um único comando, o uso das chaves é opcional.

A estrutura *for* é equivalente a uma estrutura *while* com o seguinte formato:

inicialização
while (condição)
{
    sequência de comandos
    incremento
}

OBSERVAÇÕES:

- Qualquer uma das cláusulas do cabeçalho (inicialização, condição ou incremento) pode ser omitida; no caso da omissão da inicialização ou do incremento considera-se que estes são comandos nulos (ou seja, não executam nada), já na omissão da condição considera-se que o seu valor lógico é sempre verdadeiro. Os sinais de ponto-e-vírgula que separam cada uma das cláusulas não podem ser omitidos.
- As cláusulas de inicialização e incremento podem se constituir de vários comandos cada uma; nesse caso, os comandos devem ser separados entre si por vírgulas.

Exemplos:

1)

```
/* neste caso, x é usado como variável de controle do laço
(controla a execução entre 1 e 100) e também tem o seu valor
impresso pela função printf  */

for (x = 1; x <= 100; x++)
{
      printf("%d", x);
```

```
}
```

2)

```
/* neste caso a sequência de comandos é nula, e o laço é
utilizado somente para "gastar tempo" contando de 0 a 999 */

for (x = 0; x< 1000; x++);
```

3)

```
/* não há incremento, e o laço é executado até que o valor
digitado pelo usuário seja 10 */

for (x = 0; x != 10;)
      scanf("%d", &x);
```

4)

```
/* duas variáveis são inicializadas, testadas e incrementadas */

for (x = 0, y = 0; x + y < 100; x++, y++)
      printf("%d", x + y);
```

4.5 Comandos de interrupção

4.5.1 Comando break

O comando *break* pode ser utilizado para interromper a execução de um laço a qualquer momento. Somente o laço mais interno é interrompido, e a execução continua no comando seguinte a esse laço.

Exemplo:

```
#include <stdlib.h>  /* requerida para usar rand()  */
#include <stdio.h>

int main()
{

      int sorteio = rand();         /* gera um número aleatório
                                    entre 0 e 32767  */
      int num, x;
      for (x = 0; x<10; x++)
      {
            printf("Tente acertar o número (entre 0 e 32767).");
            printf("Vc tem %d tentativas.", 10 - x);
            scanf("%d", &num);
            if (num == sorteio)     /* se acertou o número */
            {
                  break;      /* interrompe o laço (não são
                              necessárias mais tentativas) */
            }
      }

      /* se x igual a 10, o usuário esgotou suas tentativas
      sem obter sucesso  */
```

```
        if (x < 10)
        {
              printf("Muito bem!");
        }
        else
        {
              printf("Lamentável!");
        }
        return 0;
}
```

4.5.2 Comando continue

O comando *continue* tem funcionamento semelhante ao *break*, com a diferença de que somente a iteração corrente é interrompida; ou seja, a execução do laço continua a partir do início da próxima iteração.

Exemplo:

```
/* imprime os números pares

for (x = 0; x < 100; x++)
{
      /* se o número não é par, passa para a próxima iteração
      sem imprimir */

      if (x % 2 != 0)
            continue;

      printf("%d, ", x);
}
```

5. Modularização em C

Um programa estruturado pode ser construído somente com sequências, decisões e repetições. Porém, quanto mais complexo o programa, mais difícil se torna gerenciá-lo, dar manutenção, etc.

Modularização - dividir o programa em módulos. Estes podem ser, por exemplo, arquivos-fonte separados para funcionalidades separadas e, em um nível mais baixo, **funções**.

Vantagens de modularizar:

- divisão de tarefas - ou seja, poder delegar a implementação de módulos diferentes para desenvolvedores diferentes.
- facilidade de manutenção - erros que ocorrem em módulos precisam ser corrigidos somente nos módulos - não são multiplicados ao longo do programa.
- facilidade de reutilização - módulos podem ser (re)utilizados várias vezes dentro do contexto do programa.

5.1 Funções

Em C não existe uma distinção entre funções e subrotinas. Ou seja, todas as subrotinas, do ponto de vista de algoritmos, podem ser tratadas como funções que não retornam nenhum valor.

Formato de declaração de funções :

```
Tipo de retorno     identificador_da_função  (tipo1 param1, tipo2 param2,..., tipoN paramN)
{

      /* corpo da função */

      return valor de retorno;

}     /* fim da função */
```

Tipo de retorno especifica o tipo do valor que será retornado para quem chamou a função. Quando o tipo de retorno for **void** isto significa que se trata de uma função que se comporta como uma subrotina; ou seja, a função não necessita retornar nenhum valor, apenas ser chamada.

Exemplos de tipos de retorno nos cabeçalhos das funções:

```
      int func1(...)    /* retorna um valor inteiro */
      void func2(...) /* não retorna nenhum valor. Comporta-se como subrotina */
```

O comando **return** é utilizado para realizar o retorno da função; este pode ser utilizado em qualquer ponto da função que se deseje finalizar a sua execução e retornar o valor (se a função retornar algum valor) para quem a chamou.

Valor de retorno é o valor a ser efetivamente retornado e pode ser tanto uma variável como uma constante; nos casos em que a função não retorna nenhum valor o comando **return** deve ser utilizado sozinho ou pode-se simplesmente omití-lo, o que fará com que a função retorne automaticamente ao seu final.

Exemplos de uso de **return**:

```
      return 0;         /* retorna o valor constante 0 */
      return var;       /* retorna o valor da variável 'var' */
      return;           /* não retorna valor. É usado para funções com retorno do tipo void */
```

Os parâmetros *param1* a *paramN* identificam os parâmetros que se deseja passar para a função. Cada um destes parâmetros passa a ser uma variável local da função de tipo *tipo1* a *tipoN* e é inicializado com o valor que foi passado para si no momento da chamada da função. Funções que não recebem nenhum valor como parâmetro devem ser declaradas com a palavra **void** entre os parênteses.

Exemplos de declarações de parâmetros no cabeçalho das funções:

```
      /* dois parâmetros, um int e um char. O ... se refere a um tipo de retorno qualquer */
      ... Func1(int var, char var2)
```

```
{
}
... Func2 (void)              /* não recebe nenhum parâmetro */
{
}
```

Exemplo de função e programa em C que a chama:

```
int func1 (char carac, int inteiro, float flutuante)  /* declaracão da
função */
{
      /* pode-se declarar outras variáveis aqui dentro, como em um
trecho normal de programa estas variáveis são locais da função */
      int outra;

      /* uso das variáveis recebidas como parâmetro */
      printf("%c", carac);          printf("%f", flutuante);
      scanf("%d",  &outra);

      printf("%d", inteiro + outra);

      return outra;     /* retorna o valor da variável  'outra' */


} /* fim da função */


int main ()  /* programa principal */
{
      char c1;
      float f;
      int resultado;
      int inteiro;
      /*esta variável 'inteiro' existe no escopo da função 'main',
logo não tem nada a ver com a variável 'inteiro' que é criada na
função 'func1' no momento da passagem dos parâmetros */

      /* lê um número inteiro, um caracter e um float */
      scanf("%d, %c, %f", &inteiro, &c1, &f);

      /* chama a função 'func1' com os parâmetros na ordem correta */
      resultado = func1(c1, inteiro, f);
      printf("%d", resultado);      /* imprime resultado da função */
      return 0;
}
```

Observações:

- **main ()** também é uma função, porém especial já que ela representa o ponto de partida para qualquer programa em C;
- A versão completa do cabeçalho de main (`int main(int argc, char* argv[]`) define os seguintes parâmetros:

  - argc: número de argumentos passados para o *loader* do sistema operacional. Ou seja, se o programa for executado em linha de comando, indica quantas "opções de linha de comando" (mais o nome do programa em si) foram fornecidas.
  - argv: *array* de *strings*, cada uma contendo uma das opções de linha de comando fornecidas (argv[0] é o próprio nome do programa). Para maiores detalhes, ver os capítulos sobre *arrays* e *strings.*
  - o valor de retorno corresponde a um código de erro ou sucesso. Normalmente se utiliza o valor 0 para sucesso e -1 para erro.
- O resultado da função 'func1', no exemplo acima, não precisa necessariamente ser atribuído a uma variável (no caso, 'resultado'); se isto não acontecer o valor de retorno da função simplesmente será perdido. Porém, como a função foi feita para retornar um valor inteiro isto deve ser evitado, porque se constitui em uma má estruturação e uso da função;
- Todas as variáveis declaradas dentro do corpo de uma função são *locais* a ela, ou seja, só existem enquanto a função está sendo executada.

Todas as funções devem ser "conhecidas" no local onde forem utilizadas, ou seja, a sua declaração deve vir antes do uso. Caso não se deseje implementar a função antes do local onde ela vai ser utilizada pode-se escrever um *protótipo* da seguinte forma:

```
Tipo de retorno identificador_da_função (tipo1 param1, tipo2 param2,..., tipoN paramN);
```

O protótipo deve ser colocado antes da chamada da função, sinalizando então ao compilador que aquela função existe e vai ser implementada adiante. No nosso exemplo, se quiséssemos escrever a função 'func1' depois da função 'main' deveríamos incluir um protótipo de 'func1' antes dela.

CUIDADO!! O protótipo não é exatamente igual ao cabeçalho da função, ele possui um ponto-e-vírgula a mais no final!

5.2 Variáveis globais

Em C considera-se como *variável global* todas aquelas variáveis declaradas fora do escopo de qualquer função (inclusive da função 'main'). Qualquer variável só é conhecida após a sua declaração, logo costuma-se declarar todas as variáveis globais no início do programa, antes da implementação das funções que a utilizam.

Exemplo de declaração e uso de variáveis globais:

```
int c;
char t;

/* função que retorna um valor inteiro e não recebe parâmetro */
int func1 (void)
{
      /* existe uma variável t que é global, porém esta funciona como uma variável local */
      int t;
      /* c é global, logo pode ser utilizada dentro da função 'func1' */
      if (c!=0)
      {
            c++;
```

```
            t = c*2;
            /* neste caso o valor de t retornado é o da variável local,
      já que definições locais sobrepõem-se a definições globais nos
      escopos onde existem */
            return t;
      }
      else  return 0;

}

int main()
{
      int retorno;
      printf("Entre com um caracter:");
      scanf("%c", &t);
      printf("Entre com um inteiro:");
      scanf("%d", &c);        /*as variáveis t e c podem ser usadas
aqui porque são globais */

      retorno = func1();      /* chama a função func1 e retorna o valor
na variável 'retorno' */

      printf("\nResultado: %d", retorno);


}
```

5.3 Passagem de parâmetros por valor

Na passagem por valor, uma cópia do valor do argumento é armazenado no parâmetro da função chamada. Qualquer alteração deste parâmetro não se reflete no valor original do argumento.

Uma alternativa para a passagem de parâmetro de valor, que é a passagem de parâmetros **por referência utilizando ponteiros**, permitiria que a função alterasse o valor do parâmetro de forma que esta alteração se refletisse no valor original do argumento. Este tipo de passagem de parâmetros será melhor estudado no capítulo sobre 'Ponteiros'.

5.4 Funções com lista variável de parâmetros

Em C é possível declarar funções cuja quantidade de parâmetros não é definida. Cabe então à função, por meio do uso de funções específicas de biblioteca de C, obter cada um dos parâmetros recebidos e convertê-lo para o tipo desejado.

A biblioteca **cstdarg** provê alguns tipos de dados e funções utilizadas para a obtenção dos parâmetros de uma lista:

va_list – tipo de lista de parâmetros variáveis, utilizado para declarar uma estrutura (ver o capítulo sobre "Estruturas de dados") que contém os parâmetros variáveis recebidos.

void va_start(va_list lista, ultimo) – macro utilizada para inicializar a lista de parâmetros do tipo *va_list. Ultimo* é o identificador do último parâmetro à direita que não pertence à lista variável de parâmetros.

tipo va_arg(va_list lista, tipo) – permite, a partir da *lista* do tipo *va_list*, obter o valor de tipo *tipo* do próximo argumento da lista.

void va_end(va_list lista) – finaliza a obtenção dos parâmetros da *lista.*

Para declarar uma função com lista variável de parâmetros:

```
Tipo de retorno identificador_da_função (tipo1 param1, tipo2 param2, ...);
```

Onde a elipse (. . .) denota o início da lista variável de parâmetros.

Um exemplo: função que recebe n valores e retorna a sua média:

```
/* n é a quantidade de valores, que vêm em  seguida na lista de parâmetros */
float media (int n, ...)
{
      float soma = 0;
      int i;
      va_list valores;  /* lista de parâmetros */
      va_start(valores, n);   /* 'n' é o último parâmetro fixo antes da lista de parâmetros variáveis */

      for (i = 0; i < n; i++)
      {
            /* aqui o valor do próximo parâmetro, de tipo float é adicionado a 'soma'*/
            soma += va_arg(valores, float);
      }
      va_end(valores);  /* finaliza a obtenção dos parâmetros */

      return soma/n;

}
```

6. *Arrays (vetores e matrizes)*

6.1 Definição de vetor

Vetor em C é uma variável composta por um conjunto de dados com um mesmo nome (identificador) e individualizadas por um índice.

Observação: basicamente, o que diferencia um vetor de uma matriz em C é que os vetores definem uma única dimensão (portanto um único índice), ao passo que as matrizes definem mais do que uma dimensão. Tecnicamente, ambos são conhecidos pelo nome de *arrays.*

6.2 Declaração de vetor em C

O vetor é declarado da seguinte maneira:

```
tipo  nome [tamanho];
```

Onde *tipo* é o tipo de cada um dos elementos do vetor e *tamanho* é o número de elementos do vetor.

Para acessar um elemento do vetor a sintaxe é:

```
nome [índice];
```

IMPORTANTE! O índice do primeiro elemento de um vetor é SEMPRE ZERO! Assim, *indice* pode variar entre 0 e o valor de *tamanho* – 1.

Por exemplo, para a declaração de um vetor chamado *teste* cujo tipo dos dados é *char* e que tenha 4 posições declara-se:

```
char teste [4];
```

O índice do último elemento indexável do vetor é 3, pois em C a primeira posição utilizada é a posição 0. Neste caso as posições disponíveis no vetor são as seguintes:

teste[0]
teste[1]
teste[2]
teste[3]

6.3 Passagem de vetor como parâmetro para função

Existem três maneiras possíveis:

```
tipo_retorno nome (tipo v[tam], ...);
tipo_retorno nome (tipo v[], ...);
tipo_retorno nome (tipo * v, ...);
```

Em todos os casos a função recebe uma referência (endereço). Note que na última maneira é utilizado um ponteiro, que será explicado mais adiante.

Por ser passada uma **referência**, as alterações feitas nos elementos do vetor dentro da função **serão refletidas nos valores originais do vetor** (já que se utilizará sua posição real na memória).

Por exemplo:

```
/* as alterações no vetor v, efetuadas dentro da função, serão
refletidas nos valores originais do vetor que é passado como argumento
na chamada da função (ver main abaixo).  */
void troca (int v[])
{
      int aux;
      aux = v[0];
      v[0] = v[1];
      v[1] = aux;
}

int main()
{
```

```
      int nums[2];
      nums[0] = 3;
      nums[1] = 5;
      troca (nums); /* O argumento é o nome do vetor */
      /* imprime '5, 3', já que os valores do vetor nums foram
trocados dentro da função 'troca' */
      printf("%d, %d", nums[0], nums[1]);
      return 0;
}
```

6.4 Declaração de matriz

A declaração de matrizes se dá da seguinte maneira:

```
tipo nome[dim1][dim2];
```

Onde *dim1* e *dim2* são as duas dimensões da matriz (no caso de uma matriz bi-dimensional). Para se acessar um elemento da matriz a sintaxe é:

```
nome[ind1][ind2];
```

Onde *ind1* e *ind2* seguem as mesmas regras dos índices de vetores unidimensionais (ou seja, podem assumir valores entre 0 e a dimensão – 1), sendo *ind1* o índice da linha e *ind2* o índice da coluna.

A declaração em C (considerando, p. ex., o tipo *char*) e a correspondente representação gráfica de uma matriz M 3x2 se dão da seguinte maneira:

```
char M[3][2];
```

M[0][0]	**M[0][1]**
M[1][0]	**M[1][1]**
M[2][0]	**M[2][1]**

Na memória ela pode ser vista da seguinte forma (obs: os valores da esquerda representam endereços arbitrários de memória, considerando uma matriz de elementos *char* de um byte):

0100	M[0][0]
0101	M[0][1]
0102	M[1][0]
0103	M[1][1]
0104	M[2][0]
0105	M[2][1]

6.5 Passagem de matriz como parâmetro para função

As possibilidades são as seguintes:

```
tipo retorno nome(tipo m[dim1][dim2],...)
  tipo retorno nome(tipo m[][dim2],...)
    tipo retorno nome(tipo *m,...)
```

No primeiro caso, *dim2* deve ser fornecido para que o compilador possa calcular o deslocamento em bytes em relação ao endereço do primeiro elemento para uma determinada posição. Ver seções "Relação ponteiro-vetor" e "Alocação dinâmica de Memória", do capítulo sobre "Ponteiros", para maiores informações.

No segundo caso, m só pode ser utilizado por meio da aritmética de ponteiros (explicada adiante).

Assim como na passagem de *array* unidimensional como parâmetro para função, a passagem de *arrays* multidimensionais também é feita por referência. Ou seja, quaisquer alterações efetuadas dentro da função serão refletidas na matriz original passada como argumento.

Exemplo:

```
void inverte_linha(int m[][2])
{
      int aux1, aux2;
      aux1 = m[0][0];
      aux2 = m[0][1];
      m[0][0] = m[1][0];
      m[0][1] = m[1][1];
      m[1][0] = aux1;
      m[1][1] = aux2;
}

int main()
{
      int m[2][2];
      .
      .
      .
      //passagem por referência -> função pode alterar o
      //conteúdo original da matriz.
      inverte_linha(m);
      .
      .
      .
}
```

6.6 Inicialização de vetores e matrizes

Para vetores: valores entre chaves, separados por vírgulas. Por exemplo:

```
      int primos [7] = {2, 3, 5, 7, 11, 13, 17};
```

Caso o número de valores de inicialização seja menor que o tamanho do vetor, as posições restantes serão preenchidas com zeros. Por exemplo:

```
      int teste[5] = {1, 2, 3}; /* teste[3] e teste[4] recebem 0 */
```

Para matrizes cada linha é preenchida entre chaves, com valores separados por vírgulas. Todas as linhas ficam entre chaves.
Ex:

```
      int m[5][3] = {{1, 2, 3,}, {3, 2, 1}, {3, 3, 2}, {1, 2, 1} {3,
2, 0}};
```

Caso algum elemento não seja explicitado, ele será preenchido com zero.
Ex:

```
int m2[3][4] = {{3, 2, 5}, {4, 6}, {1, 2, 3, 4}};
```

3	2	5	0
4	6	0	0
1	2	3	4

7 Ponteiros

Ponteiro em C é uma variável que, ao invés de armazenar um dado de um determinado tipo, armazena o endereço de um dado de um determinado tipo:

Ponteiros são usados freqüentemente para:

- Acesso a E/S mapeada em memória
- Uso de alocação dinâmica de memória.
- Alternativa para passagem de parâmetros por referência (em C++)

7.1 Declaração de ponteiros em C

Os ponteiros são declarados da seguinte maneira:

```
tipo *nome;
```

Onde *nome* é o identificador do ponteiro e *tipo* é o tipo de dado para o qual ele pode apontar.
Ex:

```
int *d;
short int *ptr;
float *ptr2;
```

Observação importante: o tipo de dado para o qual o ponteiro aponta não tem relação com o tamanho do ponteiro em si (quantidade de espaço que ele ocupa em memória). Em geral o tamanho do ponteiro está relacionado ao tamanho do espaço de memória endereçável (e, portanto, ao valor máximo de um endereço), sendo modernamente 32 ou 64 bits no caso dos computadores pessoais.

7.2 Operadores de ponteiro

Operador &: Operador de referenciação. Retorna o endereço de uma variável. Pode ser utilizado para inicializar um ponteiro.

Operador *: Operador de derreferenciação. Retorna o conteúdo do endereço apontado por um ponteiro.
Ex:

```
int x,a;
int *ptr;
```

```
x = 30;
ptr = &x;   /*  ptr <- endereço de x   */
.
.
.
a = *ptr;  /* a recebe o conteúdo do endereço apontado*/
```

Um modo didático para o entendimento de ponteiros é "ler" o significado de * e & como "conteúdo do endereço apontado por" e "endereço de", respectivamente. Por exemplo no seguinte código:

```
int *ptr;
int x;
x = 10;
*ptr = 3; /* O CONTEÚDO DO ENDEREÇO APONTADO POR ptr recebe 3 */
ptr = &x; /* ptr recebe o ENDEREÇO DE x */
```

7.3 Problemas no uso de ponteiros

Ponteiros sempre devem apontar para endereços correspondentes a variáveis que tenham sido declaradas ou a regiões de memória nas quais não existam dados ou código de outros programas. Por exemplo, o seguinte código armazena o conteúdo da variável x em um endereço qualquer, que pode ser um endereço inválido.

```
int *ptr;
int x = 3;
*ptr = x; /* ERRO! Para onde o ptr aponta???  */
```

Ponteiros devem apontar para dados do mesmo tipo de sua declaração, do contrário podem ocorrer interpretações erradas na operação de derreferenciação. Por exemplo, o seguinte código não armazena o valor 56 na variável f, já que o ponteiro para *float* tentará ler o tamanho de um dado *float* a partir do endereço de memória da variável x e não o tamanho de um *int*, que é o tipo declarado da variável x.

```
int x = 56;
float *ptr;
float f;
ptr = &x; /* Ponteiro para float aponta para int */
.
.
.
f = *ptr; /* ERRO! Valor de F não é 56 */
```

7.4 Aritmética de ponteiros

Valores numéricos inteiros podem ser adicionados ou subtraídos de um ponteiro. O resultado é um endereço que segue as regras da aritmética de ponteiro, ou seja:

Para um ponteiro declarado da seguinte maneira:

```
tipo* ptr;
```

e inicializado com um endereço *end1*:

```
ptr = end1;
```

a operação *ptr + N,* onde *N* é um número inteiro, resulta um endereço que é igual a *end1* mais *N* **vezes o tamanho do tipo de dado apontado** (ou seja, o tamanho em bytes de *tipo*). Por exemplo, considerando que a variável x foi alocada no endereço 120:

```
int x, y;
int * ptr;
ptr = &x;          /* ptr recebe o endereço 120 */
y = *(ptr + 4);    /* y recebe o conteúdo do endereço 120 + 4*(tamanho
do int) == endereço 136 (caso o int tenha 4 bytes) */
```

Outro exemplo:

```
float *ptr;
ptr = (float*)100;    /* ponteiro é 'forçado' para o end. 100 */
.
.
.
*(ptr + 3) = 15; /* Número 15 é armazenado no endereço 100 + 3x4
= 112 */
```

7.5 Relação ponteiro-vetor

Quando um vetor é declarado, o identificador deste vetor marca o endereço de início da área de memória alocada por ele. Assim, o nome do vetor pode ser usado como referência de endereço com os mesmos operadores utilizandos para ponteiros. Portanto:

```
int vetor[10], b;        /* vetor alocado no end. 100 (p. ex.) */
.
.
.
b = vetor[3];            /* posição 3 == end. 112 de memória (para
int de 4 bytes) */
```

Equivale a:

```
int vetor[10], b;        /* vetor alocado no end. 100 (p. ex.) */

/* Na memória isso será guardado na posição 112 -> 100 + 3 x
Tamanho do tipo de dado apontado (int = 4 bytes) */
b = *(vetor + 3);
```

7.6 Uso de ponteiro para passagem de parâmetros

Em muitos casos é interessante que uma função forneça mais do que um valor de saída como seu resultado. Porém a sintaxe de linguagem C permite somente um valor de retorno direto (através do comando **return**).

O uso de ponteiros cria uma alternativa para que uma função forneça mais do que um valor de saída, baseado no fato de que o conceito de endereço em um programa é independente de escopo. Ou seja, se uma função chamadora fornecer para a função chamada os endereços de suas variáveis, a função chamada poderá recebê-los em ponteiros e preencher valores nestes endereços, que por sua vez estarão disponíveis para acesso direto pela função chamadora.

Exemplo:

```
/* função recebe dois endereços de 'int' como parâmetros  */
void troca (int *a, int *b)
{
      int aux;
      aux = *a;         /* conteúdo do endereço recebido como
      parâmetro */
      *a = *b;
      *b = aux;
}

int main()
{
      int n1 = 8, n2 = 5;
      /* endereços de n1 e n2 são passados para a função troca()
*/
      troca (&n1, &n2);
      printf("%d, %d", n1, n2);
      return 0;
}
```

7.7 Alocação dinâmica de memória

Alocação dinâmica de memória consiste em reservar espaço para o armazenamento de dados sob demanda, liberando este espaço quando não for mais necessário.

A alocação dinâmica apresenta duas vantagens principais:

1) A memória pode ser utilizada somente quando for necessária e durante o tempo que for necessária.
2) A memória pode ser alocada na medida certa, ou seja, sem sobras. Isso é possível porque a alocação dinâmica de memória permite o uso de um valor variável como parâmetro para a quantidade de memória a ser alocada.

Para fazer alocação dinâmica são utilizadas funções da biblioteca **alloc.h**, das quais as principais serão apresentadas aqui.

A função *malloc* é usada para tentar alocar um espaço contíguo de *n* bytes de memória. Caso consiga ela retorna o endereço de início da área de memória, caso contrário retorna zero. O protótipo da função é:

```
void* malloc (int n);
```

No exemplo abaixo, *v* é um ponteiro para inteiro. Se este ponteiro apontar para uma área de memória reservada com determinado tamanho, ele pode ser utilizado da mesma forma que um vetor, com o operador de colchetes para indexação. Isto decorre do fato de que:

```
int vetor[10];
x = vetor[i]            equivale a      x = *(vetor+i)
```

Portanto, se temos o ponteiro

```
int *v
x = *(v + i)
```
equivale a `x = v[i],`

desde que v aponte para uma área de memória de tamanho suficiente para conter pelo menos *i+1* elementos. No exemplo, para que isso aconteça, o tamanho em bytes da área reservada (via malloc) deve ser igual ao número de elementos desejados (*n*) vezes o tamanho de cada elemento em bytes.

Exemplo:

```
int *v;
int n;
printf ("Quantos elementos no vetor?");
scanf ("%d", &n);
v = (int*) malloc(n * sizeof(int)); /* Alocar n vezes o tamanho
de um 'int' */
if (v == 0)
{
      printf ("Erro");
}
else
{
      /* aqui poderia vir o código para manipulação do vetor
      .
      .
      .
      */
      free (v);
}
```

A função *free()* é chamada ao final da utilização do espaço de memória dinamicamente alocado para liberar este espaço, permitindo que seja utilizado por outras operações de alocação dinâmica. O protótipo de *free* é o seguinte:

```
void free(void* ptr)
```

onde *ptr* contém o endereço inicial da área de memória a ser desalocada.

A alocação de espaço para uso como *array* multidimensional ("matriz") pode ser efetuada de duas formas distintas:

1) Efetuando a alocação de um espaço de $M_1.M_2. ...M_n$ posições de tamanho *tam*, onde M_1 a M_n são as dimensões do *array* e *tam* é o tamanho em bytes de cada elemento. Neste caso, o acesso a uma determinada posição do *array* não poderá ser feito por meio do operador de colchetes, devendo ser utilizado um cálculo de aritmética de ponteiros levando em consideração os tamanhos de cada uma das dimensões M_1 a M_n.

Exemplo:

```
int *m;
int m1, m2;
/* neste exemplo, m é matriz bidimensional de int */
printf ("Dimensões M1 e M2 da matriz?");
scanf ("%d,%d", &m1, &m2);
```

```
m = (int*) malloc(m1 * m2 * sizeof(int)); /* Alocar m1 * m2
vezes o tamanho de um 'int' */
if (m == 0)
{
      printf ("Erro");
}
else
{
      /* ... */
      /* por exemplo, acesso ao elemento m[3][5] */

      printf("%d", *(m + 3*m2 + 5));   /* m2 é a quantidade de
      "colunas" em cada "linha" */

      free (m);
}
```

2) Efetuando a alocação de um vetor de M_1 posições de tamanho tam_p cada, onde M_1 é o número de "linhas" de uma matriz bidimensional e tam_p é o tamanho de um **ponteiro** para o tipo de cada elemento. Após isso, para cada elemento *i* do vetor alocado, deve-se alocar um novo vetor de M_2 posições de tamanho *tam* cada, onde M_2 é o número de "colunas" da matriz bidimensional e *tam* é o tamanho de cada elemento a ser armazenado. A vantagem desta abordagem é que cada elemento da matriz resultante pode ser acessado fazendo-se uso do operador de colchete, conforme em uma matriz bidimensional comum alocada estaticamente.

Exemplo:

```
int **m;
int m1, m2; /* ponteiro para ponteiro (pois será um "vetor de
ponteiros") */
/* neste exemplo, m é matriz bidimensional de int */
printf ("Dimensões M1 e M2 da matriz?");
scanf ("%d,%d", &m1, &m2);
m = (int**) malloc(m1 * sizeof(int*)); /* Alocar m1 vezes o
tamanho de um 'int*' */
if (m == 0)
{
      printf ("Erro");
}
else
{
      for (int i = 0; i < m1; i++)
      {
            /* cada elemento m[i] corresponde a um endereço de
            int, que é preenchido com o endereço de alocação de
            um vetor de tamanho m2 */
            m[i] = (int*)malloc(m2 * sizeof(int));
      }
      /* ... */
      /* por exemplo, acesso ao elemento m[3][5] */

      printf("%d", m[3][5]);

      /* ... */


      for (int i = 0; i < m1; i++)
      {
            /* cada elemento m[i] deve ser individualmente
```

```
                desalocado */
                free(m[i]);
        }
        free (m);
}
```

8 Strings

8.1. Definição de string

Strings são seqüências de caracteres diversos. São conhecidos por "literais" na teoria de algoritmos estruturados, sendo representados entre aspas. Alguns exemplos de strings:

"Fulano da Silva",
"? Interrogação? ",
"1,234",
"0".

Em C, strings são representadas por meio de vetores de caracteres, terminados com o caractere de fim de string cujo valor na tabela ASCII é zero (0 ou \0).

8.2 Declaração de string

Um vetor em C que pretenda armazenar uma string **n** caracteres deve ser alocado com **n+1** posições do tipo *char* para conter o terminador de string. A inicialização de uma string pode ser efetuada com o uso de uma sequência de caracteres entre aspas.

Exemplos de declarações de string:

```
char frase [50]; /* este vetor pode abrigar uma string de 49 caracteres.
                    Inicialmente ele está "vazio" -> não possui uma
                    string válida. */
char frase[50] = "";    /* este vetor contém uma string
                    que não tem nenhum caracter (tamanho 0). */
char frase[] = "Primeira frase"; /*Inicialização sem a dimensão.
Compilador assume que o tamanho do vetor é o tamanho da frase mais
1.*/
char frase[16] = "Primeira frase";
char frase[6] = {'T', 'e', 's', 't', 'e', 0);    /* inicializado como
um vetor de caracteres comum, 'forçando' o caracter terminador */
```

No caso do terceiro e do quarto exemplos, a representação do vetor da string *frase* é:

'P'	'r'	'i'	'm'	'e'	'i'	'r'	'a'	' '	'f'	'r'	'a'	's'	'e'	0

Onde cada quadrado representa um byte de memória (tamanho de um *char*).

8.3 Operações sobre string

String não é um tipo primitivo da linguagem C, por isso as seguintes operações **NÃO** são válidas:

```
char str1[10];
char str2[] = "Palavra 2";
str1 = str2; /* ERRO! Não copia str2 em str1 */

if (str1 == str2) /* ERRO! Não compara str1 com str2 */
{
      .
      .
      .
}
```

Para operar sobre strings são utilizadas funções da biblioteca **string.h**. Esta biblioteca possui algumas dezenas de funções com diversas variações e por questões de simplificação apenas algumas das principais serão explicadas neste material. Para maiores detalhes sobre as demais funções, consultar documentação sobre a biblioteca (geralmente disponível nos arquivos de ajuda dos ambientes de desenvolvimento).

8.3.1 strlen

Protótipo:

```
int strlen (char *string)
```

Descrição: Retorna o número de caracteres de uma string (exceto o caractere de fim de string).
Exemplo:

```
char nome[] = "Fulano";
printf ("O nome possui %d letras", strlen (nome));
```

8.3.2 strcpy

Protótipo:

```
char *strcpy (char *string1, char *string2)
```

Descrição: Copia o conteúdo de *string2* em *string1* e retorna o endereço de *string1*.
Exemplo:

```
char str1[10];
char str2[] = "Palavra";
strcpy (str1, str2); /* Agora str1 também contém "Palavra" */
```

Observações:

- *strcpy()* não faz qualquer verificação sobre o tamanho máximo do vetor de destino, portanto é responsabilidade do programador garantir que a string copiada caiba no espaço declarado. Uma forma alternativa de reforçar que o espaço do vetor de destino não seja excedido é utilizar a função *strncpy(dest, orig, tam),* sendo o parâmetro *tam* um inteiro indicando o número máximo de caracteres copiados (sem contar o terminador).

8.3.3 strcmp

Protótipo:

```
int strcmp (char *string1, char *string2)
```

Descrição: Compara os conteúdos de *string1* e *string2* caracter a caracter e retorna

- 0 se string1 = string2
- <0 se string1 < string2 (ou seja, se string1 é anterior a string2 na ordem alfabética, levando-se em consideração que o conjunto de letras maiúsculas é anterior ao conjunto de letras minúsculas)
- >0 se string1 > string2

Exemplo:

```
char nome1[] = "Fulano"
char nome2[] = "Beltrano";
if (strcmp (nome1, nome2) == 0)
{
      printf ("Nomes são iguais");
}
else
{
      printf ("Nomes são diferentes);
}
```

Observações:

- *strcmpi()* funciona de forma semelhante porém desconsidera diferenças entre maiúsculas e minúsculas. Por exemplo, a string "ABC" seria considerada igual à string "abc" por esta função.

8.3.4. gets

Protótipo:

```
void gets (char *string1)
```

Descrição: Recebe uma string via teclado e armazena em *string1*. Os caracteres são armazenados até que o enter seja pressionado.
Exemplo:

```
char nome[10];
gets (nome);
```

Observações:

- a função gets() permite que o usuário forneça mais caracteres do que os que podem ser armazenados no vetor, o que pode causar um erro. Para evitar este problema, pode-se utilizar a função *fgets*:

```
char nome[10];
fgets(nome, 10, stdin);   /* 'stdin' é um arquivo aberto por padrão, relacionado aos dados digitados via teclado  */
```

No exemplo mostrado, *fgets* receberia 9 caracteres (ou até que o usuário teclasse enter) e armazenaria os dados digitados na string *nome,* adicionando o caracter terminador de string. É importante observar que, caso o usuário digitasse ENTER antes de 9 caracteres, o caracter de nova linha ('\n') também seria armazenado no vetor.

- gets() termina quando o usuário digita um enter e armazena a string inteira digitada, diferentemente de scanf("%s"), que também termina no enter porém armazena a string somente até o primeiro espaço. Para contornar este problema pode-se utilizar opções de recebimento de dados do scanf:

```
scanf ("%s", str); /* Armazena uma string até o primeiro espaço
inserido */
scanf ("%[^\n]s", str) /* Recebe uma string até que seja enviado
o caractere ASCII \n, que corresponde a enter */
```

8.4 Entrada controlada de dados

É possível fazer uma entrada de dados controlada (ou seja, os caracteres são checados assim que são recebidos) recebendo os mesmos um a um. No exemplo a seguir implementaremos uma entrada de senha que mostra os caracteres * na tela ao invés das letras correspondentes utilizando a função *getch* (que não ecoa o caracter digitado para o monitor), da biblioteca *conio.h*. Note que só serão aceitos letras e não números e símbolos.

```
int i = 0; char str[9];
printf ("Digite uma senha de oito letras");
while (i < 8)
{
      str[i] = getch();
      if (((str[i] >= 'a') && (str[i] <= 'z')) || ((str[i] >=
      'A') && (str[i] <= 'Z')))
      {
            printf ("*");
            i++;
      }
}
str[i] = 0;       //insere o terminador de string
```

Vale ressaltar que a função *getch()* é bloqueante, ou seja, não retorna enquanto não houver caracter disponível no buffer de entrada. Caso o comportamento de bloqueio seja indesejado (p. ex. em uma aplicação como um jogo, que necessita que diversas tarefas como movimentação de personagens, etc., sejam efetuadas independente de o jogador teclar algo ou não), pode-se utilizar a função *kbhit()* para testar se existe tecla disponível antes de chamar a função *getch()*.

```
if (kbhit())      //retorna 0 se não tem tecla, != 0 se tem
{
      carac = getch();
      //efetua o processamento da tecla
      //…
}
else
{
      //não tem tecla pressionada, continua com outras ações
}
```

9 Registros

9.1 Definição de registro

Um registro em linguagem C é um recurso programático que permite o agrupamento de dados que tenham alguma relação entre si. O uso de registros visa facilitar a organização dos dados e simplificação do código, por consequência.

Por exemplo, em um programa que pretende gerenciar os dados acadêmicos dos alunos matriculados em uma universidade, pode ser interessante agrupar estes dados em registros para facilitar as operações de criação, edição, deleção, consulta e também o seu armazenamento em algum tipo de base de dados (arquivo, etc.)

9.2 Declaração de registro

A declaração de um registro para utilização em um programa em C depende de 2 etapas:

1) Definição de um **tipo** específico para o registro
2) Declaração de uma **variável composta** do tipo previamente definido, a qual ocupará memória e conterá os dados que foram agrupados quando da definição do seu tipo.

A forma padrão da linguagem C para definição de um tipo específico para um registro é a seguinte:

```
typedef struct
{
      Tipo1 campo1;
      Tipo2 campo2;
      TipoN campoN;

} NomeDoTipo;
```

Onde *campo1* a *campoN* são os campos, que representam os nomes internos dos dados sendo agrupados.
Tipo1 a *TipoN*, por sua vez, são os tipos de cada um dos campos. Note-se que cada campo em si pode ser de qualquer tipo, inclusive de um outro tipo definido anteriormente de registro.
NomeDoTipo é o nome que será utilizado posteriormente para declarar variáveis compostas do tipo sendo definido.

Exemplo:

```
typedef struct
{
      char nome[50];
      int codigo_matricula;
      double coeficiente;
      int código_curso;
} Aluno;
```

Uma vez que o tipo do registro esteja definido, a declaração de uma variável composta daquele tipo pode ser feita da seguinte forma:

```
NomeDoTipo NomeDaVariavel;
```

onde *NomeDoTipo* é o nome utilizado na definição do tipo e *NomeDaVariavel* é o identificador da variável em si.
Como *NomeDoTipo* passa a ser identificado como um tipo comum pelo compilador C, pode-se utilizá-lo também para declarações de arrays e ponteiros.

Exemplos:

```
Aluno a1;               /* variável composta do tipo Aluno */
Aluno turma[44];        /* array de alunos */
Aluno* prox_aluno;      /* ponteiro para aluno */
```

9.3 Utilização de registro

Uma vez que uma variável composta foi criada a partir de um tipo de registro, os seus campos podem ser acessados utilizando-se o operador de ponto (.) da seguinte forma:

```
NomeDaVariavel.campo
```

Exemplo:

```
Aluno a1;               /* variável composta do tipo Aluno */

a1.coeficiente = 1.0;
/* atribuindo valor inicial ao coeficiente do aluno */

printf("\nDigite o código de matricula do aluno: ");
scanf("%d", &a1.codigo_matricula);
/* obtendo código de matricula do aluno a1 via scanf */

printf("O código do curso do aluno eh %d", a1.codigo_curso);
/* mostrando curso do aluno */
```

Caso se tente acessar um campo a partir de um **ponteiro** para o registro, utiliza-se o operador de flecha (->) da seguinte forma:

```
NomeDoPonteiro->campo
```

Exemplo:

```
void imprime_dados_aluno(struct Aluno* ptr)
{

      printf("\nNome: %s", ptr->nome);
      printf("\nCodigo do curso: %d", ptr->código_curso);
      printf("\nCoeficiente: %lf", ptr->coeficiente);
      printf("\nCodigo de matricula: %d", ptr->código_matricula);
}
```

9.4 Registros como parâmetros para funções

Em linguagem C, os registros são passados como parâmetros para funções **por valor.** Ou seja, o conteúdo completo do registro é copiado para dentro da função, sendo que alterações efetuadas na cópia não afetam o conteúdo original.
Como um registro pode ser composto por muitos bytes, geralmente opta-se por passar o registro por referência utilizando ponteiro, acessando-se então os seus campos dentro da função por meio do operador de flecha. Nesse caso, como a passagem é por referência,

somente o valor do endereço armazenado no ponteiro é de fato copiado para dentro da função, evitando cópias desnecessárias dos campos do registro.

10 Manipulação de arquivos

10.1 Conceito de arquivo

Um arquivo é uma abstração lógica relacionada a um conjunto de dados. Tipicamente esta abstração é utilizada pelos sistemas operacionais para se referir a dados que estão armazenados em algum dispositivo de memória não-volátil (p. ex. HD, drive de rede, etc.), embora possam também utilizar esta abstração para outros fins como, p. ex., permitir a obtenção de dados de uma fonte, como o teclado, utilizando-se do mesmo conjunto de APIs.

Um arquivo geralmente possui as seguintes propriedades:

- *Nome do arquivo*, que permite identificá-lo em um sistema de arquivos.
- *Dados do arquivo,* que correspondem ao conjunto de dados que estão efetivamente nele armazenados. Geralmente estes dados se apresentam para um programa em C na forma de uma sequência de bytes (semelhante a um array unidimensional).
- *Tamanho do arquivo,* que indica a quantidade de dados armazenados.
- *Atributos de arquivo,* que fornecem indicações sobre de que forma um arquivo se apresenta no sistema de arquivos e como ele pode ser acessado. P. ex., os atributos podem definir se o arquivo é somente para leitura, se ele corresponde a um diretório, se ele é oculto, entre outras.

A figura a seguir mostra como o arquivo “Hello world.txt”, na pasta “C:\teste” de um sistema de arquivos, poderia ser abstraído em função de suas propriedades.

Nome: C:\teste\Hello world.txt
Atributos: A (arquivo)
Tamanho: 11 bytes
Conteúdo:

‘H’	‘e’	‘l’	‘l’	‘o’	‘ ‘	‘W’	‘o’	‘r’	‘l’	‘d’

Início do arquivo — Fim do arquivo

O formato dos *dados do arquivo* depende do que estes dados significam. Geralmente, um determinado programa ou conjunto de programas que gera estes dados segue uma especificação exata de formato, e esta especificação também tem que ser seguida pelo programa ou conjunto de programas que utilizarão estes dados posteriormente (podem ser os mesmos programas que geraram os dados).

Independente do significado dos dados, existem duas grandes categorias em que um arquivo pode ser enquadrado no que diz respeito ao formato dos dados:

Arquivos de texto – correspondem aos arquivos cujos dados estão somente na forma textual, ou seja, podem ser exibidos por um editor de texto que saiba interpretar os códigos de caracteres (p. ex. Bloco de notas). Exemplos de arquivos de texto incluem os arquivos de código-fonte de linguagens de programação (.c, .cpp, .java) e arquivos .txt.

Arquivos binários – correspondem aos arquivos cujos dados (bytes) podem assumir qualquer valor numérico entre 0 e 255. O significado do número armazenado em cada byte depende do formato do arquivo, implementado pelos programas que o geram e que o utilizam.
Exemplos: arquivos .exe (código executável no Windows), arquivos .doc (documentos do Word), arquivos .mp3 (áudio codificado segundo o padrão MPEG-2 Camada 3), etc.

10.2 Operações de arquivo

As operações a seguir são realizáveis por um programa em um arquivo de um sistema de arquivos típico:

- **Abertura do arquivo**, permitindo que este fique disponível para utilização pelo programa. É a primeira operação a ser realizada.
- **Leitura dos dados do arquivo,** geralmente atualizando automaticamente o marcador de acesso.
- **Escrita/atualização de dados do arquivo,** geralmente atualizando automaticamente o marcador de acesso.
- **Alteração/reposicionamento do marcador de acesso.** Este corresponde a uma posição no arquivo (iniciando em 0), da qual se obtém o próximo dado a ser lido ou na qual se escreve o próximo dado. O marcador de acesso é atualizado (avançado) automaticamente pelas operações de escrita e leitura, porém pode ser reajustado quando necessário.
- **Fechamento do arquivo,** forçando a sua atualização e sinalizando que este programa está encerrando as operações sobre o arquivo.

A biblioteca *stdio* da linguagem C oferece um conjunto de funções que implementam as operações de arquivo citadas, conforme descrito a seguir.

10.2.1 Abertura de arquivo – função fopen

Protótipo:

```
FILE * fopen (const char * nome, const char * modo);
```

Tenta abrir o arquivo indicado por *nome* no modo de acesso indicado por *modo.* Caso seja bem sucedido, retorna um ponteiro para registro do tipo FILE, que será utilizado posteriormente para indicar este arquivo quando as outras funções foram chamadas. Se não conseguir abrir o arquivo, retorna 0.

A string de *modo* pode conter um dos seguintes valores:

"r" – somente leitura. O arquivo deve existir, do contrário ocorre um erro.
"w" – somente escrita. Cria um novo arquivo se ele não existir, trunca (tamanho 0) o arquivo se ele já existir.
"a" – permite adicionar dados ao arquivo (ou seja, marcador de acesso fica sempre no fim do arquivo)
"r+" – escrita e leitura. O arquivo deve existir, do contrário ocorre um erro.
"w+" – escrita e leitura. Cria um novo arquivo se ele não existir, trunca (tamanho 0) o arquivo se ele já existir.
"rb", "wb", "r+b", etc. – idem, porém para operações em arquivos no formato binário.

Exemplo:

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

int main()
{
    FILE* arq;
    /* tenta abrir o arquivo teste.txt no modo de leitura de texto */
    arq = fopen("d:\\temp\\teste.txt", "r");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {
        /* agora pode efetuar operações sobre o arquivo utilizando
        o ponteiro arq */
    }

    return 0;
}
```

10.2.2 Fechamento de arquivo – função fclose

Protótipo:

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

Fecha o arquivo indicado pelo registro apontado por *arq.* Retorna 0 se bem sucedido, diferente de 0 caso contrário.

IMPORTANTE: a operação de fechamento de arquivo é fundamental porque indica ao sistema operacional que se deseja encerrar o acesso. Em sistemas de arquivos que implementam estratégias tais como escrita atrasada (*delayed write*), o fechamento do arquivo indica ao sistema que os dados que ainda não foram escritos (estão "bufferizados") devem ser escritos imediatamente, mantendo o conteúdo do arquivo atualizado.

Exemplo:

```
/* supondo que já temos um arquivo previamente aberto e cujo
identificador é arq */

fclose(arq);
```

10.2.3 Leitura e escrita de arquivo – funções fscanf, fread, fprintf e fwrite

Ao se efetuar leitura ou escrita de arquivo, é importante verificar se os dados a serem lidos ou escritos estão no formato de texto ou no formato binário**.** Dependendo deste formato pode-se selecionar funções mais adequadas para as operações de leitura ou escrita correspondentes.

Arquivos de texto

Para escrita e leitura de arquivos de texto, pode-se utilizar as funções *fprintf* e *fscanf*. Protótipos:

```
int fprintf ( FILE * arq, const char * format, ... );
int fscanf ( FILE * arq, const char * format, ... );
```

Estas funções operam como as suas similares *printf* e *scanf*, com a diferença de que *fprintf* envia os caracteres para o arquivo indicado por *arq* ao invés de enviar para o console de texto, e *fscanf* obtém os caracteres que compõem a entrada a partir do arquivo indicado por *arq* ao invés de obter do teclado.

Exemplos:

Programa1 – escrita em arquivo de texto

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

int main()
{
    FILE* arq;
    int codigo;
    char nome[50];

    /* tenta abrir o arquivo teste.txt no modo de escrita de texto */
    arq = fopen("d:\\temp\\teste.txt", "w");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {
        printf("Nome do aluno (max. 49 caracs): ");
        scanf("%[^\n]s", nome);
        printf("Código do aluno: ");
        scanf("%d", &codigo);

        /* escrevendo os dados fornecidos pelo usuário no arquivo,
        linha por linha */
        fprintf(arq, "%d\n", codigo);
        fprintf(arq, "%s", nome);

        fclose(arq);
    }
    return 0;
}
```

Programa2 – leitura de arquivo de texto

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

int main()
{
    FILE* arq;
    int codigo;
    char nome[50];
```

```
    /* tenta abrir o arquivo teste.txt no modo de leitura de texto */
    arq = fopen("d:\\temp\\teste.txt", "r");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {
        /* lendo os dados do arquivo, conforme o formato estabelecido
        quando este arquivo foi escrito */

        fscanf(arq, "%d\n", &codigo);
        fscanf(arq, "%s", nome);

        printf("\nCodigo do aluno: %d", codigo);
        printf("\nNome do aluno: %s", nome);

        fclose(arq);
    }
    return 0;
}
```

Arquivos binários

Para escrita e leitura de arquivos binários, pode-se utilizar as funções *fwrite* e *fread*. Protótipos:

```
size_t fwrite(const void *ptr, size_t tam, size_t count, FILE * arq );
size_t fread(const void *ptr, size_t tam, size_t count, FILE * arq );
```

Onde:

ptr – ponteiro para uma região de memória (pode ser um array) que vai receber os dados lidos do arquivo (no caso de *fread*) ou de onde serão obtidos os dados para serem escritos no arquivo (no caso de *fwrite*).
tam – tamanho em bytes de cada elemento a ser escrito ou lido do arquivo.
count – quantidade de elementos de tamanho *tam* a serem escritos ou lidos do arquivo.
arq – indicador do arquivo obtido em *fopen.*

Tanto *fwrite* quanto *fread* retornam um número inteiro indicando quantos elementos puderam ser escritos ou lidos (no máximo *count*). Este valor é útil, particularmente para *fread*, para permitir que se verifique se a quantidade de dados que se pretendia ler foi de fato lida.

OBS: o tipo *size_t* indicado nos protótipos corresponde ao tipo *int.*

Exemplos:

Programa1 – escrita em arquivo binário

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

int main()
```

```
{
    FILE* arq;
    int codigo;
    float coef;

    /* tenta abrir o arquivo teste.txt no modo de escrita binária */
    arq = fopen("d:\\temp\\teste.dat", "wb");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {
        printf("Código do aluno: ");
        scanf("%d", &codigo);
        printf("Coeficiente do aluno: ");
        scanf("%f", &coef);

        /* escrevendo os dados fornecidos pelo usuário no arquivo,
        na forma binária.
        sizeof -> operador que permite obter o tamanho de um dado ou
        tipo em bytes */

        fwrite(&codigo, sizeof(int), 1, arq);
        fwrite(&coef, sizeof(float), 1, arq);

        fclose(arq);
    }
    return 0;
}
```

Programa2 – leitura de arquivo binário

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

int main()
{
    FILE* arq;
    int codigo;
    float coef;

    /* tenta abrir o arquivo teste.dat no modo de leitura binária */
    arq = fopen("d:\\temp\\teste.dat", "rb");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {
        /* lendo os dados do arquivo, conforme o formato estabelecido
        quando este arquivo foi escrito */

        fread(&codigo, sizeof(int), 1, arq);
        fread(&coef, sizeof(float), 1, arq);


        printf("\nCodigo do aluno: %d", codigo);
        printf("\nCoef do aluno: %f", coef);

        fclose(arq);
```

```
    }
    return 0;
}
```

10.2.4 Alteração do marcador de acesso – função fseek

Protótipo:

```
int fseek ( FILE * arq, long int offset, int origem );
```

Onde *offset* define a posição em bytes, a partir da *origem*, na qual o marcador de acesso do arquivo indicado por *arq* deve ser reposicionado. O número inteiro *origem* pode assumir um dos seguintes valores simbólicos:

SEEK_SET – a partir do início do arquivo (posição 0).
SEEK_CUR – a partir da posição atual do marcador de acesso.
SEEK_END – a partir do fim do arquivo (última posição).

O valor de retorno é 0 se a função foi bem sucedida, diferente de 0 caso contrário.

Exemplos:

Programa1 – escrita em arquivo binário de um conjunto de registros

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

/* Esta diretiva (pragma) instrui o compilador a alinhar todos os
dados de registros em múltiplos de 2 bytes.
Caso não seja declarada, o alinhamento padrão é de 4 bytes e,
portanto, cada registro do tipo Aluno terá 60 bytes ao invés de 58. */

#pragma pack(2)

typedef struct
{
      int codigo;
      float coef;
      char nome[50];
} Aluno;

int main()
{
    Aluno aluno;
    FILE* arq;

      /* tenta abrir o arquivo teste.dat no modo de escrita binária */
    arq = fopen("d:\\temp\\teste.dat", "wb");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {

        do
```

```
        {
            printf("Código do prox. aluno (ou 0 para sair): ");
            scanf("%d", &aluno.codigo);
            if (aluno.codigo == 0)
            {
                break;
            }
            printf("Coeficiente do aluno: ");
            scanf("%f", &aluno.coef);
            printf("Nome do aluno: ");

            /* Esta instrução tem por objetivo esvaziar o buffer
            de entrada do teclado. Isso impede que o scanf com %s
            receba erroneamente o caracter de pulo de linha gerado
            na entrada anterior e considere a string vazia. */
            fflush(stdin);
            scanf("%[^\n]s", aluno.nome);

             /* escrevendo os dados fornecidos pelo usuário no
            arquivo, na forma binária.*/
            fwrite(&aluno, sizeof(Aluno), 1, arq);

        } while (1 == 1);

        fclose(arq);
    }
    return 0;
}
```

Programa2 – leitura de um registro específico do arquivo binário

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

#pragma pack(2)
typedef struct
{
      int codigo;
      float coef;
      char nome[50];
} Aluno;

int main()
{
    Aluno aluno;
    FILE* arq;
    int pos;

      /* tenta abrir o arquivo teste.dat no modo de leitura binária */
    arq = fopen("d:\\temp\\teste.dat", "rb");
    if (arq == 0)
    {
        printf("Erro na abertura do arquivo");
    }
    else
    {
        printf("Digite a posição dos dados do aluno que deseja
consultar (0 para primeira): ");
        scanf("%d", &pos);

        /* reposiciona o marcador de acesso para a posição desejada
```

```
        (a partir do início, SEEK_SET).
        A próxima leitura obterá diretamente o registro desejado. */
        fseek (arq, pos*sizeof(Aluno), SEEK_SET);

        /* lendo os dados do arquivo */
        fread(&aluno, sizeof(Aluno), 1, arq);
        fclose(arq);

        printf("\nCodigo do aluno: %d", aluno.codigo);
        printf("\nNome do aluno: %s", aluno.nome);
        printf("\nCoeficiente do aluno: %f", aluno.coef);
    }
    return 0;
}
```