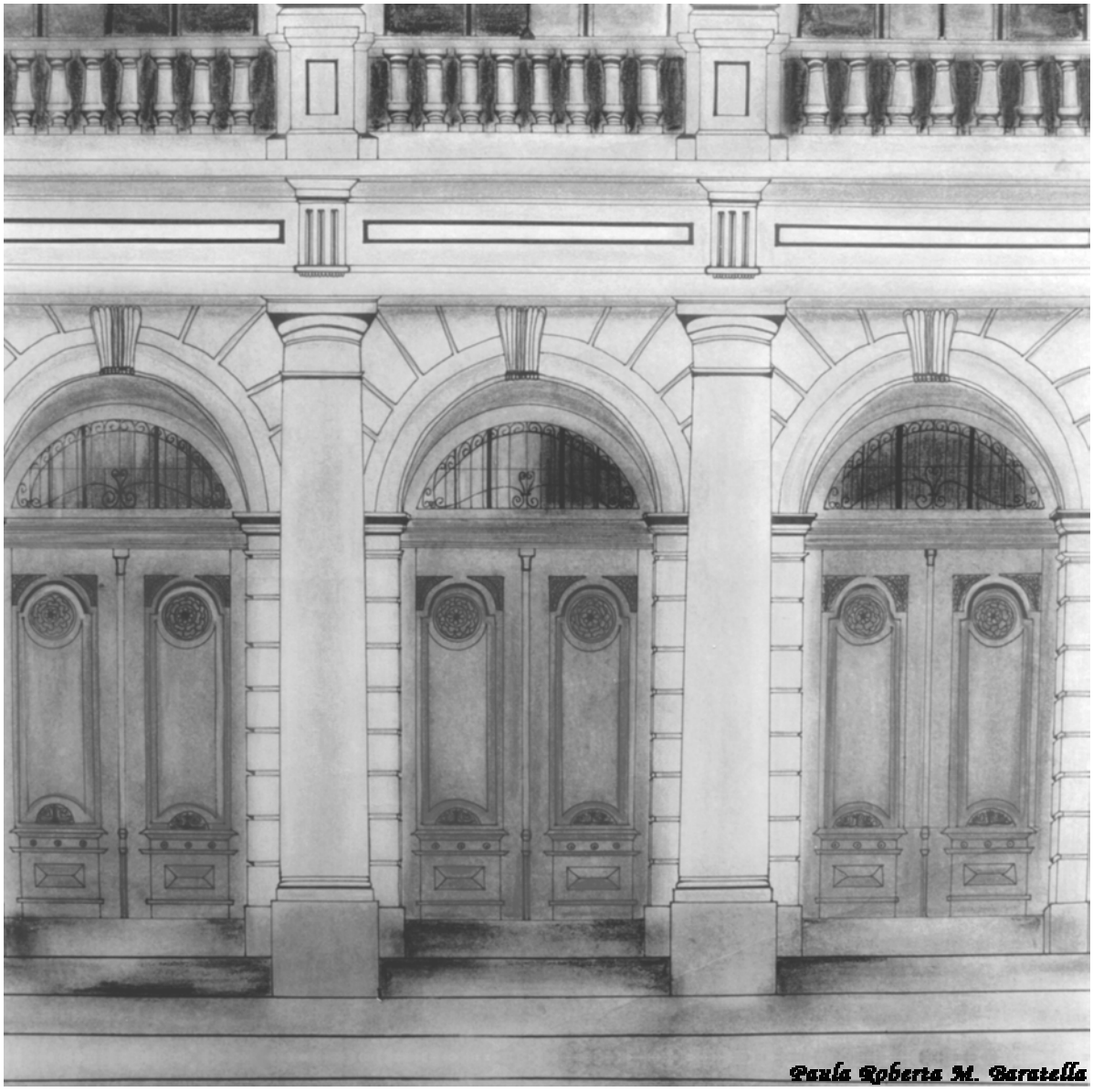

UNICAMP - Colégio Técnico de Campinas
Departamento de Processamento de Dados

Tópicos em Orientação a Objetos

Volume 2
A Linguagem de Programação C

5º Informática Matutino
1º Semestre/2006

Prof. André Luís dos R.G. de Carvalho

Índice

Palavras-Chave

As palavras listadas a seguir são reservadas pela linguagem C para serem usadas como palavras-chave:

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

Operadores e Outros Separadores

Os caracteres e as combinações de caracteres a seguir são reservados para uso como operadores, sinais de pontuação ou são reservados para o preprocessador, e não podem ser utilizados com outra finalidade:

```
!     %     ^     &     *     (     )     -     +     =     {
}     |     ~     [     ]     \     ;     '     :     "     <
>     ?     ,     .     /     ->    ++    --    .*    ->*   <<
>>    <=    >=    ==    !=    &&    ||    *=    /=    %=    +=
-=    <<=   >>=   &=    ^=    |=    #
```

Cada um deles deve ser considerado como um único símbolo.

A função `main`

Um programa deve conter uma função de nome `main` que representa o local de início da execução do programa.

Não existe em C o conceito de procedimento, muito embora possamos definir funções que não retornam valor nenhum, o que dá no mesmo. Isto pode ser feito dizendo que o tipo do retorno da função é `void`, ou seja, nada.

Funções são definidas mencionando o tipo do retorno da função, seguido pelo identificador da função, seguido por um par de parênteses (`()`) contendo, opcionalmente, a lista dos parâmetros formais da função seguidos pelo corpo da função.

Saída Básica

A seguir, veremos uma função básica para a saída de dados. Para empregá-la, faz-se necessário indicar no início do programa o uso da biblioteca na qual elas se encontram, no caso, a biblioteca `stdio`. Isto é feito pela instrução `#include <stdio.h>`. Sua forma geral é:

$$\texttt{printf (Fmt, Exp}_1\texttt{, Exp}_2\texttt{,..., Exp}_n\texttt{);}$$

Esta função escreve na saída padrão o *string* `Fmt` após substituir nele as expressões $\texttt{Exp}_i$. A cada $\texttt{Exp}_i$ a ser substituída, deve haver um sinal introduzido pelo caractere `%` em `Fmt`. Após o caractere `%` deve haver um ou mais caracteres indicadores do tipo da $\texttt{Exp}_i$ a ser substituída, bem como da formatação que desejamos para aquela expressão. Veja a seguir uma lista destes indicadores e seus significados:

- `%c`, indica a escrita de um integral sob a forma de um caractere;
- `%d ou %i`, indica a escrita de um `int` (ou `char`), `signed` ou `unsigned`, sob a forma de um número inteiro na base 10;
- `%o`, indica a escrita de `int` (ou `char`), `signed` ou `unsigned`, sob a forma de um número inteiro na base 8;
- `%x`, indica a escrita de `int` (ou `char`) , `signed` ou `unsigned`, sob a forma de um número inteiro na base 16 (letras minúsculas);
- `%X`, indica a escrita de `int` (ou `char`) , `signed` ou `unsigned`, sob a forma de um número inteiro na base 16 (letras maiúsculas);
- `%u`, indica a escrita de `int` (ou `char`) , `signed` ou `unsigned`, sob a forma de um número inteiro sem sinal na base 10;
- `%hd ou %hi`, indica a escrita de um `short int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 10;
- `%ho`, indica a escrita de um `short int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 8;

- `%hx`, indica a escrita de um `short int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 16 (letras minúsculas);
- `%hX`, indica a escrita de um `short int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 16 (letras maiúsculas);
- `%hu`, indica a escrita de `short int`, `signed` ou `unsigned`, sob a forma de um número inteiro sem sinal na base 10;
- `%ld ou %li`, indica a escrita de um `long int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 10;
- `%lo`, indica a escrita de um `long int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 8;
- `%lx`, indica a escrita de um `long int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 16 (letras minúsculas);
- `%lX`, indica a escrita de um `long int`, `signed` ou `unsigned`, sob a forma de um número inteiro na base 16 (letras maiúsculas);
- `%lu`, indica a escrita de `long int`, `signed` ou `unsigned`, sob a forma de um número inteiro sem sinal na base 10;
- `%f`, indica a escrita de um `float` em notação convencional;
- `%e`, indica a escrita de um `float` em notação científica (‘e’ minúsculo);
- `%E`, indica a escrita de um `float` em notação científica (‘E’ maiúsculo);
- `%lf`, indica a escrita de um `double` em notação convencional;
- `%le`, indica a escrita de um `double` em notação científica (‘e’ minúsculo);
- `%lE`, indica a escrita de um `double` em notação científica (‘E’ maiúsculo);
- `%Lf`, indica a escrita de um `long double` em notação convencional;
- `%Le`, indica a escrita de um `long double` em notação científica (‘e’ minúsculo);

- `%LE`, indica a escrita de um `long double` em notação científica ('E' maiúsculo);
- `%p`, indica a escrita de um ponteiros;
- `%Np`, indica a escrita de um near pointers (em máquinas PC-compatíveis);
- `%Fp`, indica a escrita de um far pointers (em máquinas PC-compatíveis);
- `%s`, indica a escrita de uma cadeia de caracteres;
- `%n`, o argumento associado deve ser um inteiro passado por referência no qual será colocado o número de caracteres escritos até então; e
- `%%`, para o caractere `%`.

Em qualquer dos formatos acima, o caractere `%` pode ser sucedido pelo caractere `*` e, eventualmente, pelo caractere `.` e por mais um caractere `*`. Neste caso, uma expressão para cada `*` deverá constar na lista de expressões a serem escritas e completarão o formato de escrita como se estivessem cada qual no lugar do `*` que lhe corresponde.

Em qualquer dos formatos acima, o caractere `%` pode ser sucedido pelo caractere `#`. Neste caso, se a letra de formato que o suceder for `g`, `f`, `e` ou `E`, teremos a garantia da escrita de um ponto decimal mesmo que não houverem dígitos decimais. No caso do letra que o suceder for `x` (ou `X`) , teremos a escrita de um 0x (ou 0X) no início da expressão a ser escrita.

Considerando que as variáveis Nome e Idade existem e têm, respectivamente, os valores `"Joao"` e `30`, temos que o comando `printf` abaixo:

```
printf ("Nome: %s - Idade: %d\n", Nome, Idade);
```

ocasionaria a seguinte saída:

```
Nome: Joao - Idade: 30
```

Em `Fmt`, entre o caractere `%` e o caractere que indica o tipo da Exp_i a ser substituída, podem aparecer números reais, com ou sem sinal, que impactuarão no alinhamento e na área de escrita.

A quantidade representada pela parte inteira do número real especifica qual o tamanho da área de escrita.

A quantidade representada pela parte fracionária do número real especifica quantos caracteres no máximo deverão ser impressos (para reais, quantos dígitos fracionários deverão ser impressos).

Se positivo, o número real especifica uma escrita alinhada à direita, se negativo, uma escrita alinhada à esquerda.

Considere a escrita do *string* "Alo Mundo" com cada uma das instâncias de Fmt abaixo. Considere ainda que o caractere | delimita a área de escrita.

- `"%7s"` -> `|Alo Mundo|`
- `"%-7s"` -> `|Alo Mundo|`
- `"%20s"` -> `|          Alo Mundo|`
- `"%-20s"` -> `|Alo Mundo          |`
- `"%20.7s"` -> `|             Alo Mun|`
- `"%-20.7s"` -> `|Alo Mun             |`
- `"%.7s"` -> `|Alo Mun|`

Para concluir, vale observar que a presença de um \n em Fmt faz com que ocorra o salto de uma linha, iisto é, que a próxima escrita ocorra no início da próxima linha, e não na seqüência, na mesma linha, como seria habitual.

[Exemplo 01 – bemvindo.c]

```
#include <stdio.h>

void main ()
{
    printf ("\nBem vindos ao estudo da linguagem C!\n");
}
```

Comentários

Os caracteres `/*` iniciam um comentário que terminará com os caracteres `*/`.

Identificadores

Identificadores introduzem nomes no programa. Em C podem ser uma seqüência arbitrariamente longa de letras e dígitos.

Embora algumas implementações imponham restrições ao tamanho máximo de um identificador, C não determina nenhuma restrição a respeito.

O primeiro caractere de um identificador deve necessariamente ser uma letra ou então o caractere sublinhado (_).

É importante ressaltar que, diferentemente de outras linguagens de programação, a linguagem C diferencia letras maiúsculas de letras minúsculas. Em C, identificadores que são lidos da mesma forma, mas que foram escritos de forma diferente no que tange ao emprego de letras maiúsculas e minúsculas, são considerados identificadores diferentes.

Identificadores que se iniciam por sublinhado (_) devem ser evitados, tendo em vista que as implementações de C os reservam para seu próprio uso.

Tipos Básicos

Não são muitos os tipos de que dispomos na linguagem C; na verdade são apenas quatro tipos básicos, a saber: char (caractere), int (inteiro), float (real) e void (vazio).

No entanto, através de qualificadores, podemos criar interessantes variações desses tipos, ampliando assim o conjunto dos tipos básicos.

Observe abaixo os possíveis tipos numéricos existentes em Turbo C++ 3.0, uma particular implementação de C++, que, como de praxe, suporta o desenvolvimento de programas em linguagem C:

1. **Inteiros:**

   Os tipos inteiros permitem a declaração de variáveis capazes de armazenar números inteiros.

Os inteiros simples têm o tamanho natural sugerido pela arquitetura da máquina; os demais tamanhos são fornecidos para atender a necessidades especiais e podem ser tornados equivalentes entre si ou a um inteiro simples.

- char: 8 bits com sinal (de -128 a 127);
- signed char: como char;
- unsigned char: 8 bits sem sinal (de 0 a 255);
- int: 16 bits com sinal (de -32768 a 32767);
- signed int: como int;
- unsigned int: 16 bits sem sinal (de 0 a 65535);
- short int: como int;
- signed short int: como short int;
- unsigned short int: como unsigned int;
- long int: 32 bits com sinal (de -2147483648 a 2147483647);
- signed long int: como long int;
- unsigned long int: 32 bits sem sinal (de 0 a 4294967295).

2. Reais:

Os tipos reais permitem a declaração de variáveis capazes de armazenar números reais. As implementações de C podem decidir livremente as características a conferir a estes tipos.

- float: 32 bits (de $3,4 \times 10^{-38}$ a $3,4 \times 10^{+38}$);
- double: 64 bits (de $1,7 \times 10^{-308}$ a $1,7 \times 10^{+308}$);
- long double: 80 bits (de $3,4 \times 10^{-4932}$ a $1.1 \times 10^{+4932}$).

Observe abaixo os possíveis tipos numéricos existentes em Visual C++ 6.0, uma particular implementação de C++, que, como de praxe, suporta o desenvolvimento de programas em linguagem C:

1. **Inteiros:**

   Os tipos inteiros permitem a declaração de variáveis capazes de armazenar números inteiros. Existem em diversos tamanhos, a saber:

- char: 8 bits com sinal (de -128 a 127);
- signed char: como char;
- unsigned char: 8 bits sem sinal (de 0 a 255);
- short int: 16 bits com sinal (de -32.768 a 32.767);
- signed short int: como short int;
- unsigned short int: 16 bits sem sinal (de 0 a 65.535);
- int: 32 bits com sinal (de – 2.147.483.648 a 2.147.483.647);
- signed int: como int;
- unsigned int: 32 bits sem sinal (de 0 a 4.294.967.295);
- long int: como int;
- signed long int: como long int;
- unsigned long int: como unsigned int.

2. **Reais:**

   Os tipos reais permitem a declaração de variáveis capazes de armazenar números reais. Existem em três tamanhos, a saber:

- float: 32 bits (de $3,4 \times 10^{-38}$ a $3,4 \times 10^{+38}$);
- double: 64 bits (de $1,7 \times 10^{-308}$ a $1,7 \times 10^{+308}$);
- long double: como double.

Observe abaixo outros tipos típicos da linguagem C, independentemente de implementação:

3. **Vazio:**

O tipo void especifica um conjunto vazio de valores. Ele é usado como tipo de retorno para funções que não retornam nenhum valor. Não faz sentido declarar variáveis do tipo void.

4. **Booleanos:**

   Vale ressaltar que na linguagem C não existe um tipo específico para armazenar valores lógicos. Para tanto, qualquer tipo integral pode ser empregado. Faz-se a seguinte convenção: o valor zero simboliza a falsidade, e qualquer valor diferente de zero, simboliza a verdade.

Constantes

Constantes são instâncias de tipos das linguagens. Temos duas variedades de constantes, a saber, constantes literais e constantes simbólicas.

Constantes Literais

Constantes literais são aquelas que especificam literalmente uma instância de um tipo da linguagem. A seguir apresentaremos as constantes literais existentes na linguagem C++ (e também na linguagem C).

Constantes Literais do Tipo Inteiro

Uma constantes inteira constituída de uma seqüência de dígitos é considerada decimal, a menos que inicie por um dígito 0, caso em que será considerada uma constante octal. É importante notar que os caracteres 8 e 9 não são dígitos octais válidos, e por isso não podem ser empregados em tais constantes.

Uma seqüência de dígitos precedida por 0x ou 0X é considerada uma constante hexadecimal. Além dos dígitos, também as letras de a (ou A) até f (ou F) são consideradas válidas para compor tais constantes.

O tipo de uma constante inteira depende de sua forma, valor e sufixo. Se ela for decimal e não tiver nenhum sufixo, ela será do primeiro desses tipos no qual seu valor puder ser representado: int, long int, unsigned long int.

Se ela for octal ou hexadecimal e não tiver nenhum sufixo, ela será do primeiro desses tipos no qual seu valor puder ser representado: int, unsigned int, long int, unsigned long int.

Independentemente da base, se ela for seguida pelo caractere u ou U, ela será do primeiro desses tipos no qual seu valor puder ser representado: unsigned int, unsigned long int.

Também independentemente da base, se ela for seguida pelo caractere l ou L, ela será do primeiro desses tipos no qual seu valor puder ser representado: long int, unsigned long int.

Por fim, ainda independentemente da base, se ela for seguida por ul, lu, uL, Lu, Ul, lU, UL ou LU, ela será unsigned long int.

Constantes Literais do Tipo Real

Uma constante real consiste numa parte inteira com sinal opcional, um ponto decimal e uma parte fracionária, um e (ou E), um expoente inteiro com sinal opcional, e um sufixo de tipo.

Tanto a parte inteira como a parte fracionária podem ser omitidas (mas não ambas). Tanto o ponto decimal seguida da parte fracionária como o e (ou E) e o expoente podem ser omitidos (mas não ambos). O sufixo de tipo também pode ser omitido.

O tipo de uma constante real é double float, a menos que explicitamente seja especificado um outro tipo. Os sufixos f (ou F) e l (ou L) especificam, respectivamente float e long double.

Constantes Literais do Tipo Caractere

Uma constantes caractere é constituída por um único caracteres delimitado por apostrofes (`'c'`). Constantes caractere são do tipo `char`, que é um tipo integral. O valor de uma constante caractere é o valor numérico do caractere no conjunto de caracteres da máquina.

Certos caracteres não visíveis, o apóstrofe (`'`), as aspas (`"`), o ponto de interrogação (`?`) e a barra invertida (`\`) podem ser representados de acordo com a seguinte tabela de seqüências de caracteres:

‘\b’	retrocesso (backspace)
‘\t’	tabulação (tab)
‘\v’	tabulação vertical (vtab)
‘\n’	nova linha (new line)
‘\f’	avanço de formulário (form feed)
‘\r’	retorno do carro (carriage return)
‘\a’	alerta (bell)

'\0'	nulo (null)
'\''	apóstrofe (single quote)
'\"'	aspas (double quote)
'\\'	barra invertida (backslash)
'\OOO'	caractére ASCII (OOO em octal)
'\xHH'	caractére ASCII (HH em hexadecimal)

Uma seqüência como estas, apesar de constituídas por mais de um caractere, representam um único caractere.

Vale ressaltar que o emprego de uma barra invertida (\) seguida por um outro caractere que não um dos especificados acima implicará em um comportamento indefinido em C++ (e também em C).

Constantes Literais do Tipo *String*

Uma constante *string* é constituída por uma seqüência de caracteres (podendo incluir caracteres visíveis e invisíveis; neste último caso, os caracteres serão representados segundo a tabela discutida acima) delimitada por aspas (").

Constantes string são vetores de char que contém toda a seqüência dos caracteres constituintes do *string* seguida pelo caractere \0 que funciona como marcador de final de *string*.

Constantes Simbólicas

Constantes simbólicas são aquelas que associam um nome a uma constante literal, de forma que as referenciamos no programa pelo nome, e não pela menção de seu valor literal.

Constantes simbólicas são aquelas que associam um nome a uma constante literal, de forma que as referenciamos no programa pelo nome, e não pela menção de seu valor literal.

Em C++ (e também em C), isto é feito através do uso da instrução `#define`. Sendo Tipo um tipo, Const o identificador de uma constante e Lit uma constante literal, temos que a forma geral de uma declaração de constante simbólica é como segue:

#define Const Lit

Entretanto, a linguagem C++ possui uma outra forma de declarar constantes simbólicas, cuja principal finalidade é possibilitar a declaração de objetos constantes. É importante ressaltar que essas duas formas de declarar constantes tem significados e usos completamente diferentes. Esta forma será apresentada em um momento mais apropriado.

Variáveis

Variáveis são definidas quando mencionamos o nome de um tipo, e em seguida uma série de identificadores separados por vírgulas (`,`) e tendo no final um ponto-e-vírgula (`;`). Cada um dos identificadores será uma variável do tipo que encabeçou a definição.

Sendo Tipo um tipo e Var_i nomes de identificadores, temos que a forma geral de uma declaração de variáveis é como segue:

$$\text{Tipo } Var_1, Var_2, \ldots, Var_n;$$

Variáveis podem ser iniciadas no ato de sua definição, i.e., podem ser definidas e receber um valor inicial. Isto pode ser feito acrescentando um sinal de igual (=) e o valor inicial desejado imediatamente após o identificador da variável que desejamos iniciar.

Sendo Tipo um tipo, Var_i nomes de identificadores e $Expr_i$ expressões que resultam em valores do tipo Tipo, temos que a forma geral de uma declaração de variáveis iniciadas é como segue:

$$\text{Tipo } Var_1 = Expr_1, Var_2 = Expr_2, \ldots, Var_n = Expr_n;$$

Entrada Básica

A seguir, veremos uma função básica para a entrada de dados. Para empregá-la, faz-se necessário indicar no início do programa o uso da biblioteca na qual elas se encontram, no caso, a biblioteca `stdio`. Isto é feito pela instrução `#include <stdio.h>`. Sua forma geral é:

$$\texttt{scanf (Fmt, } Ptr_1\texttt{, } Ptr_2\texttt{, ..., } Ptr_n\texttt{);}$$

Esta função lê da entrada padrão as variáveis cujos endereços são indicados pelos Ptr_i. Tais endereços podem ser obtidos através do operador prefixo unário simbolizado pelo caractere `&` (este operador, quando aplicado a uma variável, resulta no endereço de memória que lhe é próprio).

Para cada variável a ser lida, deve haver na string Fmt uma especificação de tipo/formato. Como na função `printf`, esses especificadores são constituídos pelo caractere `%` seguido por um ou mais caracteres que indicam o tipo e o formato do dado a ser lido.

Outros caracteres, que não os especificadores de tipo/formato, presentes na string Fmt deverão estar presentes LITERALMENTE na entrada.

Não mencionaremos aqui os especificadores de tipo/formato da função `scanf` em virtude desses serem os mesmos da função `printf` apresentados e discutidos anteriormente.

Em qualquer um dos especificadores de tipo/formatos da função `scanf`, o caractere `%` pode ser sucedido pelo caractere `*`, caso em que um valor será lido e não será atribuído a nenhuma variável (não deve constar nenhuma variável associada a esse especificado de tipo/formato).

[Exemplo 02 – bemvindo.c]

```
#include <stdio.h>

void main ()
{
    char  Nome [31];
    float Nota;

    printf ("\nQual o seu nome? ");
    scanf  ("%s", Nome);

    printf ("\nBem vindo(a) ao estudo de C, %s!\n\n", Nome);

    printf ("Com que nota voce pretende passar? ");
    scanf  ("%f", &Nota);

    printf ("\n%s, desejo sucesso a voce;\n"      , Nome);
    printf ("que voce passe com %4.1f ou mais!\n", Nota);
}
```

O Qualificador `Register`

Somente variáveis inteiras podem ser qualificadas com este qualificador e, quando o fazemos, devemos entender que estamos instruindo o compilador para que, sempre que possível, procure manter estas variáveis permanentemente em registradores de máquina e não na memória.

Como não são muito numerosos os registradores de máquina, devemos qualificar com `register` apenas aquelas variáveis que tiverem um papel chave no desempenho de um programa.

O qualificador `register` se torna inócuo no caso da impossibilidade de manter as variáveis assim qualificadas em registradores de máquina.

Escopo

A área de código onde um identificador é visível é chamada de escopo do identificador. O escopo de um nome é determinado pela localização de sua declaração. Em C existem dois tipos de escopo, a saber: escopo local e escopo global.

Escopo Local

Um nome tem escopo local se for declarado dentro de um bloco. Ele pode ser usado naquele bloco e em todos os outros blocos contidos dentro daquele bloco. Quando o bloco terminar, todos os nomes declarados dentro dele não mais estarão disponíveis.

Quando um bloco estiver dentro de outro, as variáveis declaradas dentro do bloco maior estarão disponíveis para o menor. Se ela for redefinida no bloco menor, a nova declaração só estará disponível para este último. A declaração original somente será restaurada quando a execução deixar o bloco menor e retornar para o bloco maior.

Somente variáveis podem ter escolo local. Nomes de argumentos formais de uma função são entendidos como variáveis definidas no bloco de mais alto nível da função, i.e., daquele que define o corpo da função.

Escopo Global

Um nome tem o escopo global se a sua declaração ocorrer fora de todas os blocos. Ele será visível desde o ponto onde foi declarado até o final do arquivo. Diz-se que nomes declarados com este escopo são nomes globais.

Expressões

Uma expressão é uma seqüência de operadores e operandos que especifica um computação. Uma expressão pode resultar em um valor e pode causar efeitos colaterais.

A ordem de avaliação de subexpressões é determinada pela precedência e pelo agrupamento dos operadores. As regras matemáticas usuais para associatividade e comutatividade de operadores podem ser aplicadas somente nos casos em que os operadores sejam realmente associativos e comutativos.

A ordem de avaliação dos operandos de operadores individuais é indefinida. Em particular, se um valor é modificado duas vezes numa expressão, o resultado da expressão é indefinido, exceto onde uma ordem seja garantida pelos operadores envolvidos.

O tratamento do estouro e a verificação da divisão na avaliação de expressões é dependente da implementação. A maior parte das implementações de C existentes ignora o estouro de inteiros. O tratamento da divisão por zero e de todas as exceções em ponto flutuante varia entre as diversas máquinas e é usualmente ajustável por uma função de biblioteca.

Operadores Aritméticos Convencionais

Os operadores aritméticos da linguagem C são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre números e resultam números. São eles:

1. + (soma);
2. − (subtração);
3. * (multiplicação);
4. / (divisão);

5. % (resto da divisão inteira); e
6. − (menos unário).

Não existe em C um operador que realize a divisão inteira, ou, em outras palavras, em C uma divisão sempre resulta um número real. Para resolver o problema, podemos empregar um conversão de tipo para forçar o resultado da divisão a ser um inteiro. Neste caso, o real resultante terá truncada a sua parte fracionária, se transformando em um inteiro.

O operador % (resto da divisão inteira) opera sobre dois valores integrais. Seu resultado também será um valor integral. Ele resulta o resto da divisão inteira de seu primeiro operando por seu segundo operando.

O operador − (menos unário) opera sobre um único operando, e resulta o número que se obtém trocando o sinal de seu operando.

Operadores de Incremento e Decremento

Os operadores de incremento e decremento da linguagem C não são usualmente encontrados em outras linguagens de programação. Todos eles são operadores unários e operam sobre variáveis inteiras e resultam números inteiros. São eles:

1. ++ (prefixo);
2. −− (prefixo);
3. ++ (posfixo);
4. −− (posfixo).

O operador ++ (prefixo) incrementa seu operando e produz como resultado seu valor (já incrementado).

O operador -- (prefixo) decrementa seu operando e produz como resultado seu valor (já decrementado).

O operador ++ (posfixo) incrementa seu operando e produz como resultado seu valor original (antes do incremento).

O operador -- (posfixo) decrementa seu operando e produz como resultado seu valor original (antes do decremento).

Observe, abaixo, uma ilustração do uso destes operadores:

```
...
int a=7, b=13, c;
printf ("%d %d\n", a, b);       // 7 13

a--; b++;
printf ("%d %d\n", a, b);       // 6 14

--a; ++b;
printf ("%d %d\n", a, b);       // 5 15

c = a++ + ++b;
printf ("%d %d %d\n", a, b, c); // 6 16 21
...
```

Operadores Relacionais

Os operadores relacionais da linguagem C são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre números e resultam valores lógicos. São eles:

1. == (igualdade);
2. != (desigualdade);
3. < (inferioridade);
4. <= (inferioridade ou igualdade);
5. > (superioridade); e
6. >= (superioridade ou igualdade).

Operadores Lógicos

Os operadores lógicos da linguagem C são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre valores lógicos e resultam valores lógicos. São eles:

1. && (conjunção ou *and*);

2. || (disjunção ou *or*); e
3. ! (negação ou *not*).

Operadores de Bit

Os operadores de bit da linguagem C são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre valores integrais e resultam valores integrais. São eles:

1. & (*and bit* a *bit*);
2. | (*or bit* a *bit*);
3. ^ (*xor bit* a *bit*);
4. ~ (*not bit* a *bit*);
5. << (*shift left bit* a *bit*); e
6. >> (*shift right bit* a *bit*).

Observe, abaixo, uma ilustração do uso destes operadores:

```
...
short int a,
          b=0x7C3A, // 0111 1100 0011 1010
          c=ox1B3F; // 0001 1011 0011 1111

a = b & c;          // 0001 1000 0011 1010
a = b | c;          // 0111 1111 0011 1111
a = b ^ c;          // 0110 0111 0000 0101
a = ~b;             // 1000 0011 1100 0101
a = b << 3;         // 1110 0001 1101 0000
a = c >> 5;         // 0000 0000 1101 1001
...
```

Operador de Atribuição com Operação Embutida

Os operadores aritméticos e de *bit* podem ser combinados com o operador de atribuição para formar um operador de atribuição com operação embutida.

Sendo Var uma variável, Opr um operador aritmético ou de bit e Expr uma expressão, temos que:

Var Opr= Expr;

é equivalente a

Var = Var Opr (Expr);

Observe, abaixo, uma ilustração do uso destes operadores:

```
...
float a, b, c, d, e;

a = a + (3 * b);          // a += 3*b;
b = b / (a - sin (a));    // b /= a-sin(a);
c = c << d;               // c <<= d;
d = d & (a + b);          // d &= a+b;
e = e >> (d + 3);         // e >>= d+3;
...
```

Expressões Condicionais

Expressões condicionais representam uma forma compacta de escrever comandos a escolha de um dentre uma série possivelmente grande de valores.

Sendo Cond uma condição booleana e $Expr_i$ expressões, temos que:

Condição? $Expr_1$: $Expr_2$

resulta $Expr_1$ no caso de Cond ser satisfeita, e $Expr_2$, caso contrário.

Observe, abaixo, uma ilustração do uso deste operador:

```
...
int a, b;
...
printf ("%d\n", a>b?a:b);
...
```

equivale a

```
...
int a, b;
...
int maior;

if (a>b)
    maior = a;
else
    maior = b;

printf ("%d\n", maior);
...
```

Conversões de Tipo

Expressões podem ser forçadas a resultar um certo tipo se precedidas pelo tipo desejado entre parênteses.

Sendo Tipo um tipo e Expr uma expressão que resulta um valor que não é do tipo Tipo, temos que:

(Tipo) Expr

resulta a expressão Expressão convertida para o tipo Tipo.

Observe, abaixo, uma ilustração do uso deste operador:

```
...
float f  = 3.14;
int i = (int)f;

printf ("%d\n", i); // 3
...
```

O Comando de Atribuição

Comandos são a unidade básica de processamento de uma linguagem de programação imperativa.

O comando de atribuição tem a seguinte forma básica: em primeiro lugar vem o identificador da variável receptora da atribuição, em seguida vem o operador de atribuição (=), em seguida vem a expressão a ser atribuída, em seguida vem um ponto-e-vírgula (`;`).

Sendo Var o identificador de uma variável e Expr uma expressão, temos abaixo a forma geral do comando de atribuição:

Var = Expr;

[Exemplo 03 – media.c]

```
#include <stdio.h>

void main ()
{
    char  Nome [31];
    float NtPrv1, NtPrv2, NtPrvs;

    printf ("\nQual o seu nome? ");
```

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

    printf ("\nBem, %s, nao sei quantas provas seu professor deu",
            Nome);
    printf ("\ne nem seus pesos...;");
    printf ("\nacho 2 provas com pesos iguais uma boa alternativa... ");
    printf ("vamos considerar essa possibilidade...");

    printf ("\nQuanto voce tirou na 1a Prova? ");
    scanf  ("%f", &NtPrv1);

    printf ("Quanto voce tirou na 2a Prova? ");
    scanf  ("%f", &NtPrv2);

    NtPrvs = (NtPrv1 + NtPrv2) / 2.0;

    printf ("\n%s, sua nota de prova foi %4.1f!\n", Nome, Media);
}
```

Blocos de Comando

Já conhecemos o conceito de bloco de comandos, que nada mais é do que uma série de comandos delimitada por um par de chaves (`{}`).

Blocos de comando não são terminados por ponto-e-vírgula (`;`).

Blocos de comando são muito úteis para proporcionar a execução de uma série de subcomandos de comandos que aceitam apenas um subcomando.

O Comando `if`

O comando `if` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `if`, em seguida vem, entre parênteses, a condição a ser avaliada, em seguida vem o comando a ser executado no caso da referida condição se provar verdadeira.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando if (sem else):

```
if (Cond)
        Cmd;
```

O comando `if` pode também executar um comando, no caso de sua condição se provar falsa. Para tanto, tudo o que temos que fazer é continuar o comando `if` que já conhecemos, lhe

acrescentando a palavra chave `else`, e em seguida o comando a ser executado, no caso da referida condição se provar falsa.

Sendo Cond uma condição booleana e Cmd_i dois comandos, temos abaixo forma geral do comando if com else:

```
if (Cond)
        Cmd1;
else
        Cmd2;
```

Observe que, tanto no caso da condição de um comando `if` se provar verdadeira, quanto no caso dela se provar falsa, o comando `if` somente pode executar um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a execução de mais de um comando.

Por isso, nos lugares onde se espera a especificação de um comando, podemos especificar um bloco de comandos.

O Comando `switch`

O comando `switch` tem a seguinte forma: em primeiro lugar vem a palavra-chave `switch`, em seguida vem, entre parênteses, uma expressão integral a ser avaliada, em seguida vem, entre chaves (`{}`), uma série de casos.

Um caso tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `case`, em seguida vem uma constante integral especificando o caso, em seguida vem o caractere dois pontos (`:`), em seguida vem uma série de comandos a serem executados no caso.

Sendo Expr uma expressão, $Const_I$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
case Const1:    Cmd1a;
                Cmd1b;
                …
case Const2:    Cmd2a;
                Cmd2b;
                …
```

```
        case Const3:    Cmd3a;
                        Cmd3b;
                        …
        …
}
```

Se, em alguma circunstância, desejarmos executar uma mesma série de comandos em mais de um caso, tudo o que temos a fazer é especificar em seqüência os casos em questão, deixando para indicar somente no último deles a referida seqüência de comandos.

Sendo Expr uma expressão, $Const_{I,J}$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
        case Const1,1:
        case Const1,2:
        …               Cmd1a;
                        Cmd1b;
                        …
        case Const2,1:
        case Const2,2:
        …               Cmd2a;
                        Cmd2b;
                        …
        case Const3,1:
        case Const3,2:
        …               Cmd3a;
                        Cmd3b;
                        …
        …
}
```

Se, em alguma circunstância, desejarmos executar uma série de comandos qualquer que seja o caso, tudo o que temos a fazer é especificar um caso da forma: em primeiro lugar vem a palavra-chave `default`, em seguida vem o caractere dois pontos (`:`), em seguida vem uma série de comandos a serem executados qualquer que seja o caso.

Sendo Expr uma expressão, $Const_{I,J}$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
```

```
{
        case Const1,1:
        case Const1,2:
        …               Cmd1a;
                        Cmd1b;
                        …
        case Const2,1:
        case Const2,2:
        …               Cmd2a;
                        Cmd2b;
                        …
        …
        default:
                        CmdDa;
                        CmdDb;
                        …
}
```

É importante ficar claro que o comando switch não se encerra após a execução da seqüência de comandos associada a um caso; em vez disso, todas as seqüências de comandos, associadas aos casos subseqüentes, serão também executadas.

Se isso não for o desejado, basta terminar cada seqüência (exceto a última), com um comando break.

Sendo Expr uma expressão, $Const_{I,J}$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch com breaks:

```
switch (Expr)
{
        case Const1,1:
        case Const1,2:
        …               Cmd1a;
                        Cmd1b;
                        …
                        break;
        case Const2,1:
        case Const2,2:
        …               Cmd2a;
                        Cmd2b;
                        …
                        break;
        …
        default:
```

Cmd_{Da};
Cmd_{Db};
...
}

Observe, abaixo, uma ilustração do uso deste comando:

```
...
int a, b, c;
char Operacao;
...
switch (Operacao)
{
    case '+': a = b + c;
              break;

    case '-': a = b - c;
              break;

    case '*': a = b * c;
              break;

    case '/': a = b / c;
}

cout << a << flush;
...
```

O Comando `while`

O comando `while` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `while`, em seguida vem, entre parênteses, a condição de iteração, em seguida vem o comando a ser iterado. A iteração se processa enquanto a condição de iteração se provar verdadeira.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando while:

```
while (Cond)
    Cmd;
```

Observe que, conforme especificado acima, o comando `while` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, em lugar de especificar o comando a ser iterado, podemos especificar um bloco de comandos. Assim conseguiremos o efeito desejado.

O Comando `do-while`

O comando `do-while` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `do`, em seguida vem o comando a ser iterado, em seguida vem a palavra-chave `while`, em seguida vem, entre parênteses, a condição de iteração. A iteração se processa enquanto a condição de iteração se provar verdadeira.

A diferença que este comando tem com relação ao comando `while` é que este comando sempre executa o comando a ser iterado pelo menos uma vez, já que somente testa a condição de iteração após tê-lo executado. Já o comando `while` testa a condição de iteração antes de executar o comando a ser iterado, e por isso pode parar antes de executá-lo pela primeira vez.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando `do-while`:

```
do Cmd;
while (Cond);
```

Observe que, conforme especificado acima, o comando `do-while` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, em lugar de especificar o comando a ser iterado, podemos especificar um bloco de comandos. Assim, conseguiremos o efeito desejado.

O Comando `for`

O comando `for` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `for`, em seguida vem, entre parênteses e separadas por pontos-e-vírgulas (`;`), a sessão de iniciação, a condição de iteração, e a sessão de reiniciação. Em seguida vem o comando a ser iterado. A iteração se processa enquanto a condição de iteração se provar verdadeira.

No caso de haver mais de um comando na sessão de iniciação ou na sessão de reiniciação, estes devem ser separados por virgulas (`,`).

Sendo $Cmd_{I,i}$, $Cmd_{R,i}$ e Cmd comandos, e Cond uma condição booleana, temos abaixo a forma geral do comando `for` (os números indicam a ordem de execução das partes do comando `for`):

for ($Cmd_{I,1}$, $Cmd_{I,2,...}$; Cond; $Cmd_{R,1}$, $Cmd_{R,2,...}$) Cmd;

1	**2**		**3**
	5	**4**	**6**
	8	**7**	**9**
	11	**10**	**12**
	...		
	n	**n-1**	

Observe que, conforme especificado acima, o comando `for` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, no lugar onde se espera a especificação de um comando, podemos especificar um bloco de comandos. Assim, conseguiremos o efeito desejado.

`Observe, abaix`o, uma ilustração do uso deste comando, que escreve na tela 10 vezes a frase "C e demais!":

```
...
int i;
...
for (i=1; i<=10; i++)
    printf ("C e demais!\n");
...
```

equivale a

```
...
int i=1;

for (; i<=10; i++)
    printf ("C e demais!\n");
...
```

equivale a

```
...
int i=1;

for (; i<=10;)
{
    printf ("C e demais!\n");
```

```
    i++;
}
...
```

equivale a

```
...
int i=1;

for (;;)
{
    printf ("C e demais!\n");
    i++;
    if (i>10) break;
}
...
```

Observe, abaixo, outra ilustração do uso deste comando, que inverte um vetor v de 100 elementos:

```
...
int i, j;
...
for (i=0, j=99; i<j; i++, j--)
{
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
...
```

O Comando `continue`

O comando continue força o reinicio de uma nova iteração nos comandos while, do-while e for. Assim, o comando

```
continue;
```

provoca a saída imediata do comando while, do-while, for ou switch, dentro do qual se encontra encaixado mais diretamente.

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela os números de 1 a 7, inclusive, seguidos pelos números de 17 a 20, inclusive:

```
...
int i = 0;

while (i<20)
{
    i++;
    if (i>7 && i<17) continue;
```

```
    printf ("%d\n", i);
}
...
```

O Comando `break`

O comando break força a saída imediata dos comandos while, do-while, for e switch (veja abaixo os comandos while, do-while e for). Assim, o comando

break;

provoca a saída imediata do comando while, do-while, for ou switch, dentro do qual se encontra encaixado mais diretamente.

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela os números de 1 a 7, inclusive:

```
...
int i = 0;

while (i<20)
{
    i++;
    if (i>7 && i<17) break;
    printf ("%d\n", i);
}
...
```

O Comando `goto`

O comando `goto`, também conhecido como desvio incondicional, força o desvio da execução para um ponto determinado do programa.

O ponto em questão deve ser marcado com um rótulo, ou, em outras palavras, o comando para onde se deseja desviar incondicionalmente a execução deve ser precedido por um identificador (que será o rótulo ao qual nos referimos) seguido pelo caractere dois pontos (`:`).

Sendo R um rótulo e Cmd_i comandos, temos abaixo a forma geral do comando goto:

```
        ...
R:      Cmd1;
        Cmd2;
        ...
        goto R;
        ...
```

[Exemplo 04 – media.c]

```
#include <stdio.h>

void main ()
{
    char  Nome [31];
    int   Prv, QtsPrvs;
    float NtPrv, NtPrvs;

    printf ("\nQual o seu nome? ");
    scanf  ("%s", Nome);

    printf ("\nQuantas provas foram dadas? ");
    scanf ("%d", &QtsPrvs);

    NtPrvs = 0.0;

    for (Prv=1; Prv<=QtsPrvs; Prv++)
    {
        printf ("Quanto voce tirou na %da Prova? ";
        scanf  ("%f", &NtPrv);
        NtPrvs += NtPrv;
    }

    NtPrvs /= QtsPrvs;
    printf ("\n%s, sua nota de prova foi %f!\n", Nome, NtProvs);
}
```

Funções

Todo processamento em C acontece dentro de funções. Funções são definidas mencionando o tipo do retorno da função, seguido pelo identificador da função, seguido por um par de parênteses (`()`) contendo, opcionalmente, a lista dos parâmetros formais da função, seguido por um bloco de comandos representando o corpo da função.

Uma lista de parâmetros formais nada mais é do que uma série de definições de parâmetros formais separadas por vírgulas (`,`). A definição de um parâmetro formal é feita mencionando o nome do tipo do parâmetro e em seguida o identificador do parâmetro formal.

O tipo do retorno de uma função pode ser omitido, caso em que assumir-se-á que a mesma retorna um valor do tipo `int`.

Não existe em C o conceito de procedimento, muito embora possamos definir funções que não retornam valor nenhum, o que dá no mesmo. Isto pode ser feito dizendo que o tipo do retorno da função é `void`.

O comando `return` é empregado para fazer o retorno do resultado da função a seu chamante.

[Exemplo 05 – fatorial.c]

```
#include <stdio.h>

long int Fatorial (long int N)
{
    if (N < 0)
      return -1;
    else
    {
      int F;

      for (F=1; N>1; F*=N--);

      return F;
    }
}

void main ()
{
    long int Numero, Resultado;

    printf ("\n*** Programa para calcular fatorial\n\n");
    printf ("Entre com um numero natural: ");

    scanf ("%ld", &Numero);

    Resultado = Fatorial (Numero);

    if (Resultado < 0)
      printf ("\n%ld nao e um numero natural!\n", Numero);
    else
      printf ("\nO fatorial de %ld vale %ld\n", Numero, Resultado);
}
```

Parâmetros por Referência

Para passarmos um parâmetro por referência em C, é preciso explicitamente usar ponteiros. Um parâmetro por referência, na realidade, é um ponteiro para a variável que desejamos passar. Desta forma, quando desejamos tomar ou alterar o valor da referida variável, o fazemos sempre através do ponteiro.

A declaração de uma variável que é um ponteiro para um dado tipo é idêntica à declaração de uma variável do tipo dado, exceto pelo fato de que antepomos ao identificador da variável o caractere asterisco (`*`).

Como veremos com mais detalhes quando estivermos considerando o tipo ponteiro, existe um operador prefixo unário simbolizado pelo caractere `*` que opera sobre ponteiros. Trata-se do operador de derreferenciação. Este operador, quando aplicado a um ponteiro P, resulta no conteúdo da memória apontada por P.

Existe um operador prefixo unário simbolizado pelo caractere `&`. Este operador, quando aplicado a uma variável, resulta no endereço de memória que lhe próprio. Este operador é muito útil na passagem de parâmetros por referência.

Isto porque, quando desejamos passar uma variável como um parâmetro efetivo por referência, o que passamos não é de fato a variável, e sim o seu endereço, já que a função receptora espera receber um ponteiro.

Considerando que `A` e `B` são duas variáveis do tipo `int`, veja no exemplo abaixo como as passaríamos para a função `Troca` definida no exemplo acima:

```
Troca (&A, &B);
```

[Exemplo 06 – troca.c]

```
#include <stdio.h>

void Troca (int *X, int *Y)
{
    int T;

     T = *X;
    *X = *Y;
    *Y =  T;
}

void main ()
{
    int A =  7,
        B = 13;

    printf ("\nA = %2d - B = %2d\n", A, B);

    Troca (&A, &B);

    printf ("A = %2d - B = %2d\n", A, B);

}
```

Recursão

Sabemos que o conceito de recursão se encontra entre os mais poderosos recursos de programação de que se dispõe e a linguagem C, sendo uma linguagem completa, não poderia deixar de suportá-lo.

Trata-se da possibilidade de escrever funções que ativam outras instâncias de execução de si próprias na implementação de suas funcionalidades

Como pressupomos um bom conhecimento de programação em alguma linguagem estruturada completa (como Pascal, por exemplo), entendemos que o estudo minucioso desta técnica de programação foge ao escopo deste texto e, por isso, limitar-nos-emos a mencionar sua existência a assumir que o leitor saiba como empregá-la.

[Exemplo 07 – fatorial.c]

```
#include <stdio.h>

long int Fatorial (long int N)
{
    if (N < 0)
        return -1;
    else
        if (N <= 1)
            return 1;
        else
            return N * Fatorial (N-1);
}

void main ()
{
    long int Numero, Resultado;

    printf ("\n*** Programa para calcular fatorial\n\n");
    printf ("Entre com um numero natural: ");

    scanf ("%ld", &Numero);

    Resultado = Fatorial (Numero);

    if (Resultado < 0)
      printf ("\nEsse numero nao e um numero natural!\n");
    else
      printf ("\nO fatorial de %ld vale %ld\n", Numero, Resultado);
}
```

Classes de Armazenamento

Existem duas classes de armazenamento, a saber: a automática e a estática. Objetos automáticos são locais a cada invocação de um bloco. Objetos estáticos existem e retêm seus valores até o final da execução de todo o programa.

Variáveis automáticas são iniciadas cada vez que a execução atinge o bloco no qual elas estão encaixadas, e serão destruídas cada vez que a execução deixa o referido bloco.

Todos as variáveis globais têm classe de armazenamento estática, ao passo que objetos locais podem ter classe de armazenamento estática somente através do uso explícito do especificador de classe de armazenamento `static`.

Outros Tipos

Além dos tipos básicos, a linguagem C nos disponibiliza ainda alguns outros tipos, a saber: enumerações, vetores, estruturas, uniões e ponteiros.

typedefs

Trata-se de uma forma que C disponibiliza para dar um nome a um tipo. A declaração de um tipo tem a seguinte forma: em primeiro lugar vem a palavra-chave `typedef`, em seguida, terminada por um ponto-e-vírgula, vem a declaração do tipo para o qual desejamos dar um nome.

Essa declaração tem a mesma forma de uma declaração de variável, exceto pelo fato de começar pela palavra-chave typedef.

```
...
typedef int Inteiro;

Inteiro I;
...
```

Enumerações

Uma enumeração é um tipo integral distinto com constantes nomeadas. Seu nome se torna uma palavra reservada dentro do seu escopo.

A declaração de uma enumeração tem a seguinte forma: em primeiro lugar vem a palavra-chave `enum`, em seguida vem o identificador da enumeração, em seguida, entre chaves (`{}`), vêm, separados por vírgulas, os itens da enumeração.

Um item de uma enumeração consiste de um identificador, seguido, opcionalmente por uma especificação de valor.

Uma especificação de valor consiste de um sinal de atribuição (=) seguido por um valor integral.

```
...
typedef
     enum
     {
          false,
          true
     }
     boolean;
...
```

```
...
typedef
     enum
     {
          azul,
          vermelho,
          preto,
          verde
     }
CorDeCaneta;
...
```

```
...
typedef
     enum
     {
          dom,
          seg,
          ter,
          qua,
          qui,
          sex,
          Sab
     }
     DiaDaSemana;
...
```

Vetores

Um vetor é uma coleção de informações de mesma natureza. Vetores podem ter um número arbitrariamente grande de subscritos. Subscritos identificam de forma única um elemento dentro da coleção.

A declaração de um vetor tem a seguinte forma: primeiramente vem o tipo dos elementos do vetor, em seguida vem o identificador do vetor, em seguida vem uma série de especificações de dimensão.

Uma especificação de dimensão consiste de um número natural entre colchetes (`[]`). Este número denota o tamanho da referida dimensão.

```
...
int V [100];
...
```

Para acessar um elemento de um vetor, basta mencionar o seu nome e, entre colchetes o subscrito desejado. Os subscritos em C são sempre números naturais, sendo zero o primeiro deles.

```
...
V [i] = 0;
...
```

Iniciação de Vetores

Vetores somente podem ser iniciados se forem estáticos. Vetores são iniciados da mesma forma que qualquer variável. O valor inicial de um vetor consiste dos valores dos seus elementos separados por vírgulas (`,`) e delimitados por chaves (`{}`).

```
...
int V [] = {0,2,4,6,8};
...
```

Vetores e Ponteiros

Em C, todo vetor pode ser visto como um ponteiro (e vice-versa) para o início de uma região de memória cujo tamanho $T = D_1 \times D_2 \times ... \times D_n \times T_T$ *bytes*, onde os D_i são as dimensões do vetor e T_T é o tamanho em *bytes* do tipo dos elementos do vetor.

Considerando uma implementação usual da linguagem C para um computador IBM PC Compatível, temos que:

1. O vetor `V` do exemplo acima seria um ponteiro para uma área de memória cujo tamanho T seria dado por: T = 7 x 2 = 14 *bytes*. Isto porque o vetor tem 7 posições, sendo que cada posição é um `int` (e é sabido que `int`s têm tamanho 2 *bytes*).
2. A matriz `M` do exemplo acima seria um ponteiro para uma área de memória cujo tamanho T seria dado por: T = 15 x 4 = 60 *bytes*. Isto porque a matriz tem 15 posições (5 x 3), sendo que cada posição é um `float` (e é sabido que `float`s têm tamanho 4 *bytes*).

Como sabemos, vetores são ponteiros que apontam para o início de uma região de memória destinada a armazenar os elementos do vetor. No início desta região encontramos o primeiro elemento do vetor, e em posições subsequentes, os elementos subsequentes.

Como veremos com mais detalhes quando estivermos considerando o tipo ponteiro, existe um operador prefixo unário simbolizado pelo caractere `*` que opera sobre ponteiros. Trata-se do operador de derreferenciação. Este operador, quando aplicado a um ponteiro P, resulta no conteúdo da memória apontada por P.

Ainda considerando o vetor `V` que exemplificamos acima, temos que, se `V` aponta para a primeira posição do vetor, então `*V` é o valor da primeira posição do vetor. Em outras palavras, `*V` é a mesma coisa que `V [0]`.

Vetores podem ser envolvidos com qualquer tipo integral em operações de soma e subtração. O resultado da operação será um vetor. O curioso é que quando somamos uma unidade em um vetor, obtemos um vetor cujo início é o segundo elemento do vetor original. Raciocínio análogo pode ser feito com a operação de subtração.

Fazendo um raciocínio análogo ao que fizemos para concluir que `*V` é a mesma coisa que `V [0]`, podemos concluir que `*(V + 1)` é a mesma coisa que `V [1]`, `*(V + 2)` é a mesma coisa que `V [2]`, e assim por diante.

Vale lembrar que este raciocínio não vale apenas para vetores unidimensionais, sendo válido também para vetores com qualquer número de dimensões.

Posto que vetores são ponteiros e vice-versa, temos que vetores podem também ser declarados como ponteiros diretamente.

A declaração de uma variável que é um ponteiro para um dado tipo é idêntica à declaração de uma variável do tipo dado, exceto pelo fato de que antepomos ao identificador da variável o caractere asterisco (`*`).

Observe que este vetor no fundo é apenas um ponteiro; não há nenhuma memória alocada armazenar os elementos do vetor. Esta é uma forma bastante interessante de declarar de vetores. É bastante útil para possibilitar a existência de vetores dinâmicos.

É importante ressaltar que jamais devemos acessar os elementos do vetor `V` antes de termos alocado memória para eles.

Para alocar memória, fazemos uso de uma função chamada `malloc` cuja declaração se encontra também em `<stdlib.h>`. Esta função recebe como argumento a quantidade de

bytes que se deseja alocar e resulta em um ponteiro sem tipo para o bloco de memória alocado. Caso não tenha sido possível alocar a memória solicitada, `malloc` retorna o ponteiro `NULL`.

Como `malloc` resulta um ponteiro sem tipo, antes de proceder ao armazenamento do ponteiro para a memória alocada, faz-se necessário proceder a uma conversão de tipo.

Por razões de portabilidade, sempre que desejamos saber o tamanho de um tipo, empregamos para efetuar o cálculo a função `sizeof`.

Vetores alocados dinamicamente somente são desalocados dinamicamente. Para desalocar memória, fazemos uso de uma função chamada `free` cuja declaração se encontra também em `<stdlib.h>`. Esta função recebe como argumento o ponteiro para o nó a ser desalocado e não produz nenhum valor como resultado.

[Exemplo 08 – ordvetor.c]

```
#include <stdio.h>

typedef
    enum
    {
      false,
      true
    }
    boolean;

void Ordene (int* Vetor, int Tamanho)
{
    boolean Trocou;
    int     Indice;

    do
    {
      Trocou = false;

      for (Indice = 0; Indice < Tamanho - 1; Indice++)
          if (Vetor [Indice] > Vetor [Indice + 1])
          {
            int Temp = Vetor [Indice];
            Vetor [Indice] = Vetor [Indice + 1];
            Vetor [Indice + 1] = Temp;
            Trocou = true;
          }
    }
    while (Trocou);
}

void main ()
{
```

```
    static int Vetor [10] = {9, 6, 4, 3, 0, 7, 2, 8, 5, 1};
    int Indice;

    printf ("\nVetor Original: {");

    for (Indice = 0; Indice <= 9; Indice++)
      printf ("%d, ", Vetor [Indice]);

    printf ("\b\b}\n");

    Ordene (Vetor, 10);

    printf ("Vetor Ordenado: {");

    for (Indice = 0; Indice <= 9; Indice++)
      printf ("%d, ", Vetor [Indice]);

    printf ("\b\b}\n");
}
```

[Exemplo 09 – ordvetor.c]

```
#include <stdio.h>

typedef
    enum
    {
      false,
      true
    }
    boolean;

void Ordene (int* Vetor, int Tamanho)
{
    boolean Trocou;
    int     Indice;

    do
    {
      Trocou = false;

      for (Indice = 0; Indice < Tamanho - 1; Indice++)
          if (*(Vetor + Indice) > *(Vetor + Indice + 1))
          {
            int Temp = *(Vetor + Indice);
            *(Vetor + Indice) = *(Vetor + Indice + 1);
            *(Vetor +Indice + 1) = Temp;
            Trocou = true;
          }
    }
    while (Trocou);
}

void main ()
{
    static int Vetor [10] = {9, 6, 4, 3, 0, 7, 2, 8, 5, 1};
    int Indice;
```

```
    printf ("\nVetor Original: {");

    for (Indice = 0; Indice <= 9; Indice++)
      printf ("%d, ", Vetor [Indice]);

    printf ("\b\b}\n");

    Ordene (Vetor, 10);

    printf ("Vetor Ordenado: {");

    for (Indice = 0; Indice <= 9; Indice++)
      printf ("%d, ", Vetor [Indice]);

    printf ("\b\b}\n");
}
```

[Exemplo 10 – ordvetor.c]

```
#include <stdio.h>

typedef
    enum
    {
      false,
      true
    }
    boolean;

void Ordene (int* Vetor, int Tamanho)
{
    boolean Trocou;
    int*    Atual;
    int*    Proximo;

    do
    {
      Trocou = false;

      for (Atual = Vetor, Proximo = Vetor + 1;
           Atual < Vetor + Tamanho - 1;
           Atual++, Proximo++)
        if (*Atual > *Proximo)
        {
          int Temp = *Atual;
          *Atual = *Proximo;
          *Proximo = Temp;
          Trocou = true;
        }
    }
    while (Trocou);
}

void main ()
{
    static int Vetor [10] = {9, 6, 4, 3, 0, 7, 2, 8, 5, 1};
    int Indice;
```

```
    printf ("\nVetor Original: {");

    for (Indice = 0; Indice <= 9; Indice++)
      printf ("%d, ", Vetor [Indice]);

    printf ("\b\b}\n");

    Ordene (Vetor, 10);

    printf ("Vetor Ordenado: {");

    for (Indice = 0; Indice <= 9; Indice++)
      printf ("%d, ", Vetor [Indice]);

    printf ("\b\b}\n");
}
```

Strings

Já sabemos que em C *strings* são vetores de caracteres terminados pelo caractere `'\0'` que serve ao propósito de marcar o final do *string*.

Sendo vetores, e por conseguinte ponteiros, atribuições e comparações entre *strings* não tem em C o significado que seria de se esperar. Por isso, para operar sobre *strings* deveremos empregar funções de biblioteca que a linguagem disponibiliza para tal finalidade ou manipulá-lo como um vetor de caracteres.

A Biblioteca <ctype.h>

Trata-se de uma biblioteca padrão da linguagem C. Nela encontraremos um conjunto de funções para manipular caracteres. Comentaremos a seguir algumas delas.

1. **isalnum (C)**

   Resulta em um integral diferente de zero se o caractere C for uma letra ou um dígito, e zero, caso contrário.

2. **isalpha (C)**

   Resulta em um integral diferente de zero se o caractere C for uma letra, e zero, caso contrário.

3. **iscntrl (C)**

   Resulta em um integral diferente de zero se o caractere C estiver entre 0x00 e 0x1F ou for igual a 0x7F, e zero, caso contrário.

4. isdigit (C)

Resulta em um integral diferente de zero se o caractere C for um dígito, e zero, caso contrário.

5. isgraph (C) ou isprint (C)

Resulta em um integral diferente de zero se o caractere C for imprimível (ou for branco), e zero, caso contrário.

6. islower (C)

Resulta em um integral diferente de zero se o caractere C for uma letra minúscula, e zero, caso contrário.

7. ispunct (C)

Resulta em um integral diferente de zero se o caractere C for um caractere de pontuação, e zero, caso contrário.

8. isspace (C)

Resulta em um integral diferente de zero se o caractere C for um espaço, tabulação horizontal, tabulação vertical, alimentação de formulário, retorno de carro ou um caractere de nova linha, e zero, caso contrário.

9. isupper (C)

Resulta em um integral diferente de zero se o caractere C for uma letra maiúscula, e zero, caso contrário.

10. isxdigit (C)

Resulta em um integral diferente de zero se o caractere C for um dígito hexadecimal válido, e zero, caso contrário.

11. tolower (C)

Resulta no equivalente minúsculo do caractere C se o mesmo for uma letra maiúscula, caso contrário o caractere C e retornado sem alterações.

12. toupper (C)

Resulta no equivalente maiúsculo do caractere C se o mesmo for uma letra minúscula, caso contrário o caractere C e retornado sem alterações.

A Biblioteca <string.h>

Trata-se de uma biblioteca padrão da linguagem C. Nela encontraremos um conjunto de funções para manipular *strings*. Comentaremos a seguir algumas delas.

1. strcpy (D, O)

Copia o *string* `O` dado no *string* `D` também dado.

2. strcat (D, O)

Concatena ao *string* `D` dado o *string* `O` também dado.

3. strcmp (S1, S2)

Compara os *strings* `S1` e `S2` dados. Resulta em um integral negativo, se `S1 < S2`; igual a zero, se `S1 = S2`; e positivo se `S1 > S2`.

4. strncpy (D, O, I)

Copia não mais do que I caracteres do *string* `O` dado no *string* `D` também dado.

5. strncat (D, O, I)

Concatena ao *string* `D` dado não mais do que `I` caracteres do *string* `O` também dado.

6. strncmp (S1, S2, I)

Compara não mais do que `I` caracteres dos *strings* `S1` e `S2` dados. Resulta em um integral negativo, se `S1 < S2`; igual a zero, se `S1 = S2`; e positivo se `S1 > S2`.

7. strlen (S)

Calcula o tamanho do *string* `S` dado, retornando-o como resultado.

8. strchr (S, C)

Procura no *string* `S` pela primeira ocorrência do caractere `C`. Caso encontre, devolve um ponteiro para este caractere em `S`. Devolve `NULL` caso contrário.

9. strrchr (S, C)

Procura no *string* `S` pela última ocorrência do caractere `C`. Caso encontre, devolve um ponteiro para este caractere em `S`. Devolve `NULL` caso contrário.

10. strpbrk (S1, S2)

Retorna um ponteiro para o primeiro caractere da string `S1` que coincide com algum dos caracteres da string `S2`. Caso nenhuma coincidência seja detectada, retorna o ponteiro `NULL`.

11. strspn (S1, S2)

Retorna o índice do primeiro caractere da string `S1` que não coincide com nenhum dos caracteres da string `S2`.

12. strstr (S1, S2)

Retorna um ponteiro para o primeiro a primeira ocorrência do string `S2` no string `S1`. Caso nenhuma ocorrência seja detectada, retorna o ponteiro `NULL`.

13. strtok (S1, S2)

Retorna um string contendo a próxima palavra no string `S1`. Os caracteres que funcionam como delimitadores de palavra são os que constituem o string `S2`.

Para procurar a primeira palavra de `S1`, é preciso fornecer como parâmetro `S1` o string onde procurar; na busca pelas próximas palavras, deve-se fornecer como parâmetro `S1` o ponteiro `NULL`.

É importante ressaltar que esta função modifica o string `S1`. Caso esta função seja chamada quando não houverem mais palavras no string `S1`, ela retorna o ponteiro `NULL`.

14. strerror (I)

Devolve um *string* que expressa por extenso a mensagem de erro associada ao código de erro `I` obtido em `errnum`. Ano se deve modificar o *string* obtido.

15. memchr (S, C, I)

Procura no *string* `S` pela primeira ocorrência do caractere `C` nas primeiras `I` posições de `S`. Caso encontre, devolve um ponteiro para este caractere em `S`. Devolve `NULL` caso contrário.

16.memcmp (S1, S2, I)

Compara os `I` primeiros caracteres do *string* `S1` com os do *string* `S2`. Resulta em um integral negativo, se `S1 < S2`; igual a zero, se `S1 = S2`; e positivo se `S1 > S2`.

17.memcpy (D, O, I)

Copia os `I` primeiros caracteres do *string* `O` para o string `D` (para `O` e `D` diferentes).

18.memmove (D, O, I)

Copia os `I` primeiros caracteres do *string* `O` para o string `D` (`O` e `D` podem ser o mesmo *string*).

19.memset (S, C, I)

Copia o caractere `C` nos `I` primeiros caracteres do *string* `S`. Devolve `S`.

[Exemplo 11 – palindrm.c]

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

typedef
    enum
    {
      false,
      true
    }
    boolean;

boolean ePalindrome (char* Palavra)
{
    char* Inicio;
    char* Fim;

    for (Inicio = Palavra, Fim = Palavra + strlen (Palavra - 1);
       Inicio < Fim;
       Inicio++, Fim--)
      if (*Inicio != *Fim)
          return false;

    return true;
}
```

```
void main ()
{
    char* Palavra1 = "aluno";
    char* Palavra2 = "arara";
    int* Atual;

    if (ePalindrome (Palavra1))
      printf ("\nA palavra %s e palindrome.", Palavra1);
    else
      printf ("\nA palavra %s nao e palindrome", Palavra1);

    if (ePalindrome (Palavra2))
      printf ("\nA palavra %s e palindrome.\n", Palavra2);
    else
      printf ("\nA palavra %s nao e palindrome\n", Palavra2);
}
```

Estruturas

Uma estrutura é uma coleção de informações de natureza potencialmente diversa. Estruturas podem ter um número arbitrariamente grande de campos. Chamamos de campo cada um dos identificadores que usamos para referenciar de forma única um elemento dentro da coleção.

A declaração de uma estrutura tem a seguinte forma: em primeiro lugar vem a palavra-chave `struct`, em seguida vem o identificador da estrutura, em seguida, entre chaves (`{}`), vêm, terminados por pontos-e-vírgulas (`;`), a declaração de cada um dos campos da estrutura.

A declaração dos campos da estrutura é feita da mesma forma como fazemos declarações de variáveis.

O acesso aos campos de uma estrutura se faz apondo ao nome de uma variável do tipo o caractere ponto (`.`), seguido pelo nome do campo desejado.

```
...
typedef
     struct
     {
          char RA   [10];
          char Nome [31];
     }
Aluno;
...
```

Campos de *Bits*

Se definirmos um registro, e dentro dele um campo integral, e apusermos ao nome do campo o caractere dois pontos (`:`), e a ele um número natural positivo, teremos declarado um campo cujo comprimento em *bits* é dado pelo numero referido número natural positivo.

```
...
typedef
     struct
     {
          unsigned int b7:1;
          unsigned int b6:1;
          unsigned int b5:1;
          unsigned int b4:1;
          unsigned int b3:1;
          unsigned int b2:1;
          unsigned int b1:1;
          unsigned int b0:1;
     }
     Byte;
...
```

Uniões

Uniões são conceitualmente muito parecidas com estruturas. Também são coleções de informações de natureza potencialmente diversa. Também podem um número arbitrariamente grande de campos. Neste contexto, campo também é o nome que damos para os identificadores que usamos para referenciar de forma única um elemento dentro da coleção.

A declaração de uma união também é muito parecida com a declaração de uma estrutura. Basta escrever `union` onde escrevíamos `struct`.

Também declaramos os campos de uma união da mesma forma que declaramos os campos de uma estrutura.

O acesso aos campos de uma união também se faz apondo ao nome de uma variável do tipo o caractere ponto (`.`), seguido pelo nome do campo desejado.

Mas existe uma drástica diferença entre estruturas e uniões. Os campos de uma estrutura se sucedem na memória destinada à estrutura, ou seja, na posição imediatamente seguinte àquela onde termina o primeiro campo, começa o segundo campo, na posição imediatamente seguinte àquela onde termina o segundo campo, começa o terceiro campo, e assim por diante.

Desta forma, o tamanho de uma estrutura é dado pela somatória dos tamanhos de cada um de seus campos. Nas uniões não. Todos os campos começam na mesma posição de memória e se sobrepõem. O tamanho de uma união é o tamanho de seu maior campo.

```
...
typedef
enum
{
     PessoaFisica,
     PessoaJuridica
}
Pessoa;

typedef
     struct
     {
          int  Cod;
          char Nome     [31];
          char Endereco [51];
          char Telefone [11];
          Pessoa Tipo;

          union
          {
           char CPF  [12];
           char CNPJ [15];
          }
          Id;
     }
     Cliente;
...
```

Ponteiros

Como sabemos, ponteiros em C tem vários papeis. Já vimos ponteiros na passagem de parâmetros por referência. Já vimos ponteiros como implementação de vetores. Agora veremos ponteiros no seu papel mais convencional, ou seja, na implementação de estruturas de dados encadeadas.

A declaração de uma variável que é um ponteiro para um dado tipo é idêntica à declaração de uma variável do tipo dado, exceto pelo fato de que antepomos ao identificador da variável o caractere asterisco (*).

Para implementar os nós das estruturas encadeadas, nos apoiaremos sobremaneira no conhecimento que já adquirimos de estruturas, já que os nós de uma estrutura encadeada nada

mais são do que registros que contém, dentre outras informações, ponteiros para o mesmo tipo de nó.

O final de uma estrutura encadeada é marcado pelo ponteiro `NULL`, cuja declaração pode ser encontrada em `<stdlib.h>`.

Não podemos nos esquecer de que estruturas encadeadas são compostas por nós dinâmicos, i.e., que precisam ser alocados e desalocados.

Para alocar memória, fazemos uso de uma função chamada `malloc` cuja declaração se encontra também em `<stdlib.h>`. Esta função recebe como argumento a quantidade de bytes que se deseja alocar e resulta em um ponteiro sem tipo para o bloco de memória alocado. Caso não tenha sido possível alocar a memória solicitada, `malloc` retorna o ponteiro `NULL`.

Como `malloc` resulta um ponteiro sem tipo, antes de proceder ao armazenamento do ponteiro para a memória alocada, faz-se necessário proceder a uma conversão de tipo.

Por razões de portabilidade, sempre que desejamos saber o tamanho de um tipo, empregamos para efetuar o cálculo a função `sizeof`.

Para desalocar memória, fazemos uso de uma função chamada `free` cuja declaração se encontra também em `<stdlib.h>`. Esta função recebe como argumento o ponteiro para o nó a ser desalocado e não produz nenhum valor como resultado.

Para acessar os campos de um nó cujo ponteiro dispomos, empregamos o operador de derreferenciação (`*`).

O operador de derreferenciação é um operador prefixo unário simbolizado pelo caractere `*` que opera sobre ponteiros. Este operador, quando aplicado a um ponteiro P, resulta no conteúdo da memória apontada por P.

Assim, se `Inicio` é um ponteiro para um nó, `*Inicio` é o próprio nó. Então, para acessar um determinado campo, basta apor o caractere ponto (`.`) seguido pelo identificador do campo.

Vale ressaltar a existência de uma notação alternativa mais compacta para acessar os campos de um nó cujo ponteiro dispomos. Basta apor ao identificador do ponteiro a seqüência -> seguida pelo identificador do campo.

[Exemplo 12 – lista.c]

```
#include <stdio.h>

struct sNo
{
    int         Info;
    struct sNo* Prox;
};

struct sNo* Insira (struct sNo* Inicio, int Numero)
{
    if (Inicio == NULL)
    {
      struct sNo* Novo = (struct sNo*) malloc (sizeof (struct sNo));

      Novo -> Info = Numero;
      Novo -> Prox = NULL;

      return Novo;
    }
    else
      if (Numero < Inicio -> Info)
      {
          struct sNo* Novo = (struct sNo*) malloc (sizeof (struct sNo));

          Novo -> Info = Numero;
          Novo -> Prox = Inicio;

          return Novo;
      }
      else
      {
          Inicio -> Prox = Insira (Inicio -> Prox, Numero);
          return Inicio;
      }
}

struct sNo* Remova (struct sNo* Inicio, int Numero)
{
    if (Inicio == NULL)
      return Inicio;
    else
      if (Inicio -> Info == Numero)
      {
          struct sNo* Removido = Inicio;
          Inicio = Inicio -> Prox;
          free (Removido);
          return Remova (Inicio, Numero);
      }
      else
      {
          Inicio -> Prox = Remova (Inicio -> Prox, Numero);
```

```
          return Inicio;
      }
}

void Imprima (struct sNo* Inicio)
{
    if (Inicio == NULL)
      printf ("{}\n");
    else
    {
      printf ("{");

      for (; Inicio != NULL; Inicio = Inicio -> Prox)
          printf ("%d, ", Inicio -> Info);

      printf ("\b\b}\n");
    }
}

void main ()
{
    struct sNo* Inicio = NULL;
    Imprima (Inicio);

    Inicio = Insira (Inicio, 3);
    Imprima (Inicio);

    Inicio = Insira (Inicio, 1);
    Imprima (Inicio);

    Inicio = Insira (Inicio, 2);
    Imprima (Inicio);

    Inicio = Remova (Inicio, 2);
    Imprima (Inicio);
}
```

Ponteiros para Função

Em C é possível definir variáveis do tipo ponteiro para função. Se derreferenciarmos variáveis deste tipo, obteríamos uma função, e esta, poderia sem problemas ser chamada.

[Exemplo 13 – lista.c]

```
#include <stdio.h>

struct sNo
{
    int         Info;
    struct sNo* Prox;
}* Inicio;

struct sNo* Insira (struct sNo* Inicio, int Numero)
{
    if (Inicio == NULL)
    {
      struct sNo* Novo = (struct sNo*) malloc (sizeof (struct sNo));
```

```
      Novo -> Info = Numero;
      Novo -> Prox = NULL;

      return Novo;
    }
    else
      if (Numero < Inicio -> Info)
      {
          struct sNo* Novo = (struct sNo*) malloc (sizeof (struct sNo));

          Novo -> Info = Numero;
          Novo -> Prox = Inicio;

          return Novo;
      }
      else
      {
          Inicio -> Prox = Insira (Inicio -> Prox, Numero);
          return Inicio;
      }
}

struct sNo* Remova (struct sNo* Inicio, int Numero)
{
    if (Inicio == NULL)
      return Inicio;
    else
      if (Inicio -> Info == Numero)
      {
          struct sNo* Removido = Inicio;
          Inicio = Inicio -> Prox;
          free (Removido);
          return Remova (Inicio, Numero);
      }
      else
      {
          Inicio -> Prox = Remova (Inicio -> Prox, Numero);
          return Inicio;
      }
}

void Imprima (struct sNo* Inicio)
{
    if (Inicio == NULL)
      printf ("{}\n");
    else
    {
      printf ("{");

      for (; Inicio != NULL; Inicio = Inicio -> Prox)
          printf ("%d, ", Inicio -> Info);

      printf ("\b\b}\n");
    }
}

void Insersao ()
{
    int Numero;
```

```
    printf ("Entre com o inteiro a inserir: ");
    scanf  ("%d", &Numero);

    Inicio = Insira (Inicio, Numero);
}

void Remocao ()
{
    int Numero;

    printf ("Entre com o inteiro a remover: ");
    scanf  ("%d", &Numero);

    Inicio = Remova (Inicio, Numero);
}

void Impressao ()
{
    Imprima (Inicio);
}

void Termino ()
{
    exit (0);
}

void main ()
{
    static void (*Funcao [4]) () =
    {&Insersao, &Remocao, &Impressao, &Termino};
    int Opcao;

    for (;;)
    {
      printf ("\nOpcao (0=Insercao/1=Remocao/2=Impressao/3=Termino): ");
      scanf ("%d", &Opcao);
      (*Funcao [Opcao]) ();
    }
}
```

Declarações e Definições

As declarações introduzem nomes em um programa. Toda declaração é uma definição a menos que:

1. Declare uma função, externa ou não, sem incluir o corpo da função;
2. Declare uma variável externa sem iniciá-la;
3. Declare um tipo com `typedef`.

Uma variável, função ou enumerador pode ter somente uma definição, mas pode ter mais de uma declaração, naturalmente, desde que elas todas combinem entre si.

Organização de Programa

Uma dúvida que boa parte dos programadores iniciantes têm é como dividir um programa em módulos.

Embora não seja obrigatório particionar os programas em módulos, recomenda-se fortemente que isso seja feito, especialmente programas de um porte maior. Isso fará com que o programa se torne mais manutenível, além de minimizar a perda de tempo com compilações, já que módulos são unidades de compilação separada.

Todo programa tem pelo menos um módulo, o módulo principal, que e onde encontramos a função `main`, podendo, eventualmente, ter também outros módulos.

Todo módulo, exceto o módulo principal, consiste de dois arquivos: (1) o arquivo principal do módulo; e (2) o cabeçalho do módulo. O módulo principal não tem cabeçalho, tem somente o arquivo principal do módulo.

O arquivo principal do módulo deve ter um nome com extensão `.c`, ao passo que o cabeçalho do módulo deve estar gravado em um arquivo com extensão `.h`.

Deve haver uma correspondência biunívoca entre módulos e *headers*. Isso porque o objetivo de um *header* é especificar a interface do seu módulo. Tudo o que for importante para o usuário de um módulo conseguir utilizar aquele módulo, deve estar no *header* do módulo.

Headers devem conter apenas declarações e não definições. Compõem os headers as declarações de constante, tipos, variáveis e funções. Mas note, apenas as declarações que constituírem a interface do módulo devem estar no *header* do módulo; declarações relacionadas com a implementação do módulo devem estar no próprio módulo, e portanto, ocultadas do usuário.

Para um programa poder ser executado, primeiramente cada um de seus módulos devem ser compilados a fim de gerar um arquivo objeto relocável. Os arquivos principais dos módulos são compilados diretamente, já os cabeçalhos são compilados indiretamente, sempre que incluídos no arquivo principal de um módulo.

Conceitualmente, a compilação de um módulo se dá em duas fases, a saber: o preprocessamento e a compilação efetiva.

A primeira fase realiza basicamente a inclusão de arquivos, a substituição de macros e compilações condicionais. O preprocessamento é controlado por diretivas que se iniciam pelo caractere # (não necessariamente na primeira coluna do arquivo).

O resultado do preprocessamento é uma seqüência de símbolos. Tal seqüência de símbolos é o que de fato vai ser compilado na fase seguinte, gerando um arquivo-objeto relocável.

Ato contínuo, todos os arquivos-objeto relocáveis, resultantes da compilação de cada um dos módulos do programa devem ser ligados para gerar o que chamamos de arquivo-executável.

Em ambientes integrados de desenvolvimento, a compilação dos módulos e a ligação dos arquivos objeto relocáveis se da de forma automática e relativamente transparente.

Todo módulo inclui o seu header e os headers de todos os módulos que lhe prestam serviços. O único módulo que não tem um header é o módulo principal, já que este módulo não presta serviços a nenhum outro módulo; são os outros módulos que, diretamente ou indiretamente, lhe prestam serviços.

Nomes Globais Estáticos

Quando nomes globais são definidos em um módulo, se estes não forem qualificados explicitamente como estáticos (através do qualificador `static`), outros módulos do mesmo programa poderão acessá-los se os declararem como nomes externos.

Variáveis são declaradas como externas se forem qualificadas com `extern`. Funções serão externas se as declararmos seus protótipos, i.e., as declararmos sem corpos.

Quando nomes globais são definidos em um módulo com o qualificador `static`, eles passam a ter seu escopo irremediavelmente restrito a seu módulo, i.e., outros módulos do mesmo programa não poderão em hipótese nenhuma acessá-los.

Instruções para o Preprocessador

#include <header.h>

Já conhecemos a instrução `#include <header.h>`. Ela instrui o preprocessador para substituir a instrução pelo conteúdo do arquivo nela especificado. O fato do nome do arquivo vim entre *angle brackets* (<>) indica que se trata de um arquivo *header* de um módulo do próprio ambiente de desenvolvimento.

Vale ressaltar que é permitido a inclusão aninhada de *headers*, i.e., é lícito incluir um *header*, sendo que este, por sua vez, inclua outros, e assim por diante.

#include "header.h"

A instrução `#include "header.h"` instrui o preprocessador para substituir a instrução pelo conteúdo do arquivo nela especificado. O fato do nome do arquivo vim entre aspas (") indica que se trata de um arquivo *header* de um módulo do usuário.

Vale ressaltar que é permitido a inclusão aninhada de *headers*, i.e., é lícito incluir um *header*, sendo que este, por sua vez, inclua outros, e assim por diante.

#define Nome Literal

Já conhecemos a instrução `#define Nome Literal`. Ela instrui o preprocessador sobre a definição de um nome para um determinado literal. A partir da definição, o nome será conhecido, e, sempre que mencionado no programa no decorrer da compilação, o preprocessador fará a substituição do nome pelo literal.

Trata-se de uma forma de implementar constantes simbólicas, ou seja, constantes que, além de um valor, também têm um nome. Isto faz com que a programação se torne mais clara e mais limpa, já que, com isso, deixaríamos de mencionar no programa literais sem nenhum significado, passando a mencionar nomes.

É interessante notar que o nome que esta instrução define, pode eventualmente ser parametrizado, implementando assim uma "função" que é macro substituida.

O operador # pode ser usado na composição do Literal e faz com que o argumento que o sucede seja considerado uma string.

O operador ## pode ser usado na composição do Literal e tem a função de concatenar duas palavras.

Vale ressaltar que é necessária atenção redobrada com macros deste tipo, pois a substituição ocorre exatamente da forma como foi definida. Isso pode levar a efeitos não imaginados.

#define Nome

A instrução #Define Nome instrui o preprocessador para tornar definido o símbolo identificado por Nome. O propósito de definições como estas, sem a associação de valor ao símbolo definido é subsidiar as instruções de compilação condicional sobre as quais comentaremos logo mais.

#undef Nome

A instrução #undef Nome instrui o preprocessador para tornar indefinido o símbolo identificado por Nome. O propósito desta instrução é subsidiar as instruções de compilação condicional sobre as quais comentaremos logo mais.

#ifdef Nome ... #else ... #endif

Estas instruções, em conjunto com a instrução #define Nome acima, promovem compilação condicional. O trecho de programa compreendido na primeira área marcada com reticências somente será compilado no caso do símbolo Nome estar definido, ao passo que o trecho de programa compreendido na segunda área marcada com reticências somente será compilado no caso complementar.

#ifndef Nome ... #else ... #endif

Estas instruções, em conjunto com a instrução #define Nome acima, promovem compilação condicional. O trecho de programa compreendido na primeira área marcada com reticências somente será compilado no caso do símbolo Nome não estar definido, ao passo que o trecho de programa compreendido na segunda área marcada com reticências somente será compilado no caso complementar.

#if ExpCte$_1$ … #elif ExpCte$_2$ … else … #endif

Estas instruções promovem compilação condicional. O trecho de programa compreendido na primeira área marcada com reticências somente será compilado no caso da ExpCte$_1$ ser satisfeita, o trecho de programa compreendido na segunda área marcada com reticências somente será compilado no caso da ExpCte$_2$ ser satisfeita, e assim por diante, até que o trecho de programa compreendido na última área marcada com reticências somente será compilado no caso de nenhuma das ExpCte$_i$ serem satisfeitas.

#line Nro NomArq

Estas instruções fazem com que o compilador pense estar compilando a linha de número `Nro` e do arquivo de nome `NomArq`. `NomArq` pode ser omitido.

#pragma

Várias instruções podem ser dadas ao compilador dependendo da implementação deste. Deve-se consultar o manual do compilador a fim de conhecer as possíveis diretivas deste tipo.

[Exemplo 14 – lista.h]

```
#ifndef LISTA
#define LISTA

struct sNo
{
    int       Info;
    struct sNo* Prox;
};

extern struct sNo* Insira (struct sNo* Inicio, int Numero);
extern struct sNo* Remova (struct sNo* Inicio, int Numero);
extern void Imprima (struct sNo* Inicio);

#endif
```

[Exemplo 14 – lista.c]

```
#include <stdlib.h>
#include "lista.h"

struct sNo* Insira (struct sNo* Inicio, int Numero)
{
    if (Inicio == NULL)
    {
```

```
      struct sNo* Novo = (struct sNo*) malloc (sizeof (struct sNo));

      Novo -> Info = Numero;
      Novo -> Prox = NULL;

      return Novo;
    }
    else
      if (Numero < Inicio -> Info)
      {
          struct sNo* Novo = (struct sNo*) malloc (sizeof (struct sNo));

          Novo -> Info = Numero;
          Novo -> Prox = Inicio;

          return Novo;
      }
      else
      {
          Inicio -> Prox = Insira (Inicio -> Prox, Numero);
          return Inicio;
      }
}

struct sNo* Remova (struct sNo* Inicio, int Numero)
{
    if (Inicio == NULL)
      return Inicio;
    else
      if (Inicio -> Info == Numero)
      {
          struct sNo* Removido = Inicio;
          Inicio = Inicio -> Prox;
          free (Removido);
          return Remova (Inicio, Numero);
      }
      else
      {
          Inicio -> Prox = Remova (Inicio -> Prox, Numero);
          return Inicio;
      }
}

void Imprima (struct sNo* Inicio)
{
    if (Inicio == NULL)
      printf ("{}\n");
    else
    {
      printf ("{");

      for (; Inicio != NULL; Inicio = Inicio -> Prox)
          printf ("%d, ", Inicio -> Info);

      printf ("\b\b}\n");
    }
}
```

[Exemplo 14 – menu.h]

```
#ifndef MENU
#define MENU

extern void (*Funcao [4]) ();

extern void Insersao ();
extern void Remocao ();
extern void Impressao ();
extern void Termino ();

#endif
```

[Exemplo 14 – menu.c]

```
#include <stdio.h>
#include "menu.h"
#include "lista.h"

void (*Funcao [4]) () = {&Insersao, &Remocao, &Impressao, &Termino};

static struct sNo* Inicio;

void Insersao ()
{
    int Numero;

    printf ("Entre com o inteiro a inserir: ");
    scanf  ("%d", &Numero);

    Inicio = Insira (Inicio, Numero);
}

void Remocao ()
{
    int Numero;

    printf ("Entre com o inteiro a remover: ");
    scanf  ("%d", &Numero);

    Inicio = Remova (Inicio, Numero);
}

void Impressao ()
{
    Imprima (Inicio);
}

void Termino ()
{
    exit (0);
}
```

[Exemplo 14 – controle.c]

```
#include <stdio.h>
#include "menu.h"

void main ()
{
    int Opcao;

    for (;;)
    {
      printf ("\nOpcao (0=Insercao/1=Remocao/2=Impressao/3=Termino): ");
      scanf ("%d", &Opcao);
      (*Funcao [Opcao]) ();
    }
}
```

Mais sobre Entrada e Saída

Para possibilitar a entrada e a saída de dados, contamos na linguagem C com uma coleção de funções agrupadas na biblioteca padrão `<stdio.h>`. Nela, além das funções printf e scanf, que já conhecemos, encontraremos algumas declarações de tipo, bem como de diversas outras funções para fazer entrada e saída.

O tipo mais importante nela encontrado é o tipo `FILE` que é usado para manipular arquivos. Outro tipo importante é o tipo `size_t` que representa o conceito de tamanho em diversas operações relativas a arquivos. E por fim temos também o tipo `fpos_t` que define todas as estruturas necessárias à especificação precisa de uma posição dentro de um arquivo.

Parte I: Acesso aos Dispositivos Padrão de E/S

1. getchar()

Lê um caractere do dispositivo padrão de entrada, retornando-o como resultado. Retorna `EOF` caso leia o final de arquivo.

2. putchar ()

Escreve um caractere no dispositivo padrão de saída.

3. gets (Str)

Lê uma seqüência de caracteres do dispositivo padrão de entrada e a coloca no string Str. Retorna NULL caso de falha.

4. puts (Str)

Escreve o string Str no dispositivo padrão de saída.

Parte II: Acesso a Arquivos (texto e binário)

1. FILE*

Tipo que nos possibilita o acesso a arquivos. Para declarar um arquivo faça:

```
FILE *Fil;
```

2. fopen (Str, Mdo)

Abre o arquivo cujo nome é dado por `Str` para o modo de acesso dado por `Mdo`. `Mdo` pode ser:

- `"r"`, se desejamos abrir um arquivo texto para leitura (o arquivo deve existir previamente);
- `"w"`, se desejamos criar um arquivo texto para escrita (mesmo que o arquivo já existisse seria recriado);
- `"a"`, se desejamos abrir um arquivo texto para acréscimo (se o arquivo existir, o mesmo será aberto para a escrita de dados no seu final, caso contrário, o arquivo será criado com a mesma finalidade);
- `"rb"`, se desejamos abrir um arquivo binário para leitura (o arquivo deve existir previamente);
- `"wb"`, se desejamos criar um arquivo binário para escrita (mesmo que o arquivo já existisse seria recriado);
- `"ab"`, se desejamos abrir um arquivo binário para acréscimo (se o arquivo existir, o mesmo será aberto para a escrita de dados no seu final, caso contrário, o arquivo será criado com a mesma finalidade);
- `"r+"`, se desejamos abrir um arquivo texto para leitura e escrita (o arquivo deve existir previamente);
- `"w+"`, se desejamos criar um arquivo texto para leitura e escrita (mesmo que o arquivo já existisse seria recriado);

- "a+", se desejamos abrir um arquivo texto para leitura e escrita (se o arquivo existir, o mesmo será aberto para a escrita de dados no seu final, caso contrário, o arquivo será criado com a mesma finalidade);
- "rb+", se desejamos abrir um arquivo binário para leitura e escrita (o arquivo deve existir previamente);
- "wb+", se desejamos criar um arquivo binário para leitura e escrita (mesmo que o arquivo já existisse seria recriado);
- "ab+", se desejamos abrir um arquivo binário para leitura e escrita (se o arquivo existir, o mesmo será aberto para a escrita de dados no seu final, caso contrário, o arquivo será criado com a mesma finalidade);

A função devolve um FILE* que deve ser armazenado em uma variável do tipo. A função retorna NULL se não for possível abrir o arquivo.

3. tmpfile ()

Abre um arquivo temporário cujo nome foi gerado de forma a assegurar que nenhum outro arquivo de mesmo nome exista.

A função devolve um FILE* que deve ser armazenado em uma variável do tipo. A função retorna NULL se não for possível abrir o arquivo.

O arquivo criado é automaticamente removido quando o arquivo for fechado ou quando o programa terminar.

4. tmpname (Str)

Cria um nome de arquivo de forma a assegurar que nenhum outro arquivo de mesmo nome exista.

A função devolve NULL em caso de insucesso.

5. freopen (Str, Mdo, Fil)

Fecha o arquivo Fil anteriormente aberto e abre o arquivo cujo nome é dado por Str para o modo de acesso dado por Mdo (os mesmos que valem para fopen).

A função devolve um `FILE*` que deve ser armazenado em uma variável do tipo. A função retorna `NULL` se não for possível abrir o arquivo.

6. rewind (Fil)

Reinicia o arquivo `Fil` de modo a deixá-lo como se tivesse acabado de ser aberto.

7. fclose (Fil)

Fecha o arquivo `Fil`.

8. remove (Str)

Apaga o arquivo cujo nome é especificado através do parâmetro Str. Retorna zero se a operação foi bem sucedida e um valor diferente de zero caso contrário.

9. rename (Str1, Str2)

Troca o nome do arquivo cujo nome é especificado através do parâmetro Str1 para o nome especificado através do parâmetro Str2. Retorna zero se a operação foi bem sucedida e um valor diferente de zero caso contrário.

10. feof (Fil)

A função `feof` verifica se o final do arquivo `Fil` foi atingido, devolvendo um valor diferente de zero em caso positivo e um valor igual a zero caso contrário.

11. ferror (Fil)

A função `ferror` verifica se ocorreu algum erro no último acesso ao arquivo `Fil`, devolvendo um valor diferente de zero em caso positivo e um valor igual a zero caso contrário.

12. clearerr (Fil)

A função `clearerr` desliga qualquer indicação de erro com respeito ao arquivo `Fil`.

13. fflush (Fil)

A função `fflush` esvazia o buffer associado ao arquivo `Fil`.

14. fgetc (Fil) ou getc (fil)

Lê do arquivo `Fil` um caractere, retornando-o como resultado. Retorna `EOF` caso leia o final de arquivo.

15. ungetc (C, Fil)

Retorna para o buffer de leitura do arquivo `Fil` o caractere `C` anteriormente lido daquele arquivo.

16. fputc (C, Fil) ou putc (C, Fil)

Escreve o caractere `C` no arquivo `Fil`.

17. fgets (Str, Tam, Fil)

Lê do arquivo `Fil` um string `Str` de no máximo tamanho `Tam`. Retorna `NULL` no caso de insucesso.

18. fputs (Str, Fil)

Escreve o string `Str` no arquivo `Fil`. Retorna um inteiro negativo em caso de falha.

19. fprintf (Fil, Fmt, Exp_1, Exp_2,..., Exp_n)

Análogo ao `printf`. Em vez de escrever na saída padrão, escreve no arquivo `Fil`.

20. fscanf (Fil, Fmt, Ptr_1, Ptr_2,..., Ptr_n)

Análogo ao `scanf`. Em vez de ler da entrada padrão, le do arquivo `Fil`.

21. fread (Buf, Siz, Qtd, Fil)

Lê do arquivo `Fil` uma quantidade `Qtd` de blocos de `Siz` *bytes* e os armazena no endereço dado por Buf. Retorna o número de *bytes* efetivamente lidos, que pode ser igual ou inferior à quantidade solicitada (caso em que o final do arquivo deve ter sido atingido antes da quantidade solicitada ter podido ser lida).

22. fwrite (Buf, Siz, Qtd, Fil)

Escreve no arquivo `Fil` uma quantidade Qtd de blocos de Siz *bytes*. Os *bytes* em questão devem estar armazenados no endereço dado por Buf. Retorna o número de *bytes* efetivamente escritos, que pode ser igual ou inferior à quantidade solicitada.

23. fseek (Fil, Ofs, Org)

Posiciona o *file pointer* no deslocamento `Ofs` a partir da origem `Org`. Se `Ofs` for positivo, o deslocamento será para frente, caso contrário, para trás. Org pode ser: (1) SEEK_SET, para tomar como origem o início do arquivo; (2) SEEK_CUR, para tomar como origem a posição corrente do *file pointer*; e (3) SEEK_END, para tomar como origem o final do arquivo.

24. ftell (Fil)

Retorna o valor corrente do *file pointer*.

25. fgetpos (Fil, Pos)

Armazena em Pos o valor atual do *file pointer*. O parâmetro Pos deve ser do tipo fpos_t e deve ser passado por referência. O valor armazenado em Pos serve apenas para ser passado como argumento a uma futura chamada de fsetpos.

26. fsetpos (Fil, Pos)

Posiciona o *file pointer* na posição indicada por Pos. O parâmetro Pos deve ser do tipo fpos_t e deve ser passado por referência. O valor armazenado em Pos deve ter sido obtido anteriormente através de uma chamada da função fgetpos.

Parte III: Outras Funções

1. sprintf (Str, Fmt, Exp_1, Exp_2,..., Exp_n)

Análogo ao `printf`. Em vez de escrever na saída padrão, "escreve" no *string* `Str`.

2. sscanf (str, Fmt, Ptr_1, Ptr_2,..., Ptr_n)

Análogo ao `scanf`. Em vez de ler da entrada padrão, "lê" do *string* `Str`.

[Exemplo 15 – texto.h]

```
#ifndef TEXTO
#define TEXTO

#include <stdio.H>

typedef
    enum
    {
      false,
      true
    }
    boolean;

extern boolean feoln (FILE* ArqEntrada);
extern void ComeBrancos (FILE* ArqEntrada);
extern void GravaNaoBrancos (FILE* ArqEntrada, FILE* ArqSaida);

#endif
```

[Exemplo 15 – texto.c]

```
#include "texto.h"

boolean feoln (FILE* ArqEntrada)
{
    char Caractere;

    Caractere = getc (ArqEntrada);
    ungetc (Caractere, ArqEntrada);

    return Caractere == '\n';
}

void ComeBrancos (FILE* ArqEntrada)
{
    char Caractere;

    do
    {
      Caractere = getc (ArqEntrada);
    }
    while (Caractere == ' ');

    ungetc (Caractere, ArqEntrada);
}

void GravaNaoBrancos (FILE* ArqEntrada, FILE* ArqSaida)
{
    char Caractere;

    Caractere = getc (ArqEntrada);

    while (Caractere != ' ' && Caractere != '\n' && Caractere != EOF)
    {
      putc (Caractere, ArqSaida);
      Caractere = getc (ArqEntrada);
    }
```

```
    ungetc (Caractere, ArqEntrada);
}
```

[Exemplo 15 – sbrancos.c]

```
#include <stdio.h>
#include "texto.h"

void main ()
{
     char NomArqEntrada [255], NomArqSaida [255];

     FILE* ArqEntrada;
     FILE* ArqSaida;

     printf ("\n*** Eliminacao de Brancos Superfluos\n\n");

     printf ("Entre com o nome do Arquivo de Entrada: ");
     scanf ("%s", NomArqEntrada);

     if ((ArqEntrada = fopen (NomArqEntrada, "r")) == NULL)
       printf ("Arquivo de Entrada nao existe!\n");
     else
     {
       printf ("Entre com o nome do Arquivo de Saida..: ");
       scanf ("%s", NomArqSaida);

       if ((ArqSaida = fopen (NomArqSaida, "w")) == NULL)
           printf ("Impossivel criar o Arquivo de Saida!\n");
       else
       {
           for (;;)
           {
             ComeBrancos (ArqEntrada);

             for (;;)
             {
                 GravaNaoBrancos (ArqEntrada, ArqSaida);
                 ComeBrancos (ArqEntrada);

                 if (feoln (ArqEntrada) || feof (ArqEntrada))
                   break;
                 else
                   putc (' ', ArqSaida);
             }

             if (feof (ArqEntrada))
                 break;
             else
             {
                 getc (ArqEntrada);
                 putc ('\n', ArqSaida);
             }
           }

           close (ArqEntrada);
           close (ArqSaida);
       }
```

```
    }
}
```

De volta à função `main`

Como sabemos, todo programa deve conter uma função de nome `main` que representa o local de início da execução do programa. Esta função não pode ser chamada explicitamente no programa e não pode ser declarada como `static`. Seu endereço não é possível de ser obtido.

A função `main` pode eventualmente receber dois argumentos e retornar um inteiro. Os argumentos que pode receber são, respectivamente: (1) um valor inteiro representando a quantidade de parâmetros passados ao programa pelo ambiente no qual o programa executa; e (2) se o primeiro parâmetro for diferente de zero, então este segundo argumento, que é um vetor de *strings*, conterá: (a) na posição zero, um *string* contendo o nome usado para invocar o programa; e (b) nas posições subsequentes, *strings* contendo parâmetros passados ao programa pelo ambiente no qual ele executa. O inteiro que pode ser retornado pela função `main` representa o valor resultante do programa e é retornado ao ambiente no qual o programa executa.

A iniciação de variáveis estáticas não locais de um módulo ocorre antes da primeira utilização de qualquer função ou variável definida no módulo. Nenhuma ordem é imposta na iniciação de variáveis de diferentes módulos.

A função `atexit` nos permite registrar uma função para ser executada por ocasião do término do programa.

A função `void exit (int)` pode ser chamada em qualquer função do programa e provoca (1) o fechamento de todos os arquivos abertos; (2) a escrita efetiva de todas as pendências em buffers de saída; (3) a execução da função registrada com a função `atexit;` e (4) o encerramento do programa (retornando ao processo chamante o inteiro que lhe vem como argumento). A função `exit` é declarada em `<stdlib.h>`.

O comando `return` na função `main` tem o efeito de chamar a função `exit` passando o valor de retorno como argumento.

Variáveis estáticas não locais são destruídas sempre que a função `exit` é chamada.

A função `abort ()` declarada na `<stdlib.h>` pode ser chamada para encerrar o programa sem a execução da função indicada pela função `atexit`.

[Exemplo 16 – copia.c]

```
#include <stdio.h>

int main (int argc, char* argv [])
{
    if (argc != 3)
    {
        printf ("Quantidade invalidade de argumentos!\n");
    }
    else
    {
        FILE* ArqEntrada;

        if ((ArqEntrada = fopen (argv [1], "r")) == NULL)
            printf ("Arquivo de Entrada nao existe!\n");
        else
        {
            FILE* ArqSaida;

            if ((ArqSaida = fopen (argv [2], "w")) == NULL)
                printf ("Nao e possivel criar o arquivo de Saida!\n");
            else
            {
                int  QtdBytesLidos;
                char Buffer [1024];

                while ((QtdBytesLidos =
                        fread (ArqEntrada, Buffer, 1024)) != 0)
                    fwrite (ArqSaida, Buffer, QtdBytesLidos);

                fclose (ArqSaida);
            }

            fclose (ArqEntrada);
        }
    }

    return 0;
}
```

Número de Argumentos Variável

É possível escrever funções que recebam um número não determinado de argumentos. Para tanto faz-se preciso incluir o arquivo `<stdarg.h>`.

Funções que recebem uma quantidade variável de argumentos devem necessariamente receber ao menos um argumento passado formalmente em seu cabeçalho. Os parâmetros variáveis serão necessariamente os últimos a serem realmente passados para a função.

Para recuperar um parâmetro real não especificado formalmente deve-se conhecer seu tipo. Deve-se ainda de alguma forma conhecer a quantidade destes parâmetros.

[Exemplo 17 – soma.h]

```
#ifndef SOMA
#define SOMA

extern int Soma (int Qtos, ...);

#endif
```

[Exemplo 17 – soma.c]

```
#include <stdarg.h>
#include "soma.h"

int Soma (int Qtos, ...)
{
    int Total = 0;
    va_list LstArgs;

    va_start (LstArgs, Qtos);

    for (; Qtos > 0; Qtos--)
        Total += va_arg (LstArgs, int);

    va_end (LstArgs);

    return Total;
}
```

[Exemplo 17 – controle.c]

```
#include <stdio.h>
#include "soma.h"

void main ()
{
    printf ("\n0 + 1 + 3 + 5 + 7 + 8 + 9 = %d\n",
            Soma (7, 1, 3, 5, 7, 8, 9));

    printf ("2 + 4 + 6                   = %d\n",
            Soma (3, 2, 4, 6));
}
```

Anexo I: Funções Matemáticas (#include <math.h>)

1. **acos (N)**

   Retorna a medida em radianos do arco cujo coseno é `N`. `N` deve estar entre –1 e 1.

2. **asin (N)**

   Retorna a medida em radianos do arco cujo seno é `N`. `N` deve estar entre –1 e 1.

3. **atan (N)**

   Retorna a medida em radianos do arco cuja tangente é `N`. `N` deve estar entre –1 e 1.

4. **atan2 (X, Y)**

   Retorna a medida em radianos do arco cuja tangente é `Y/X`. Esta função usa os sinal dos argumentos para calcular o quadrante do ângulo devolvido.

5. **ceil (N)**

   Retorna o arredondamento para cima do número real `N`.

6. **cos (A)**

   Retorna o coseno do ângulo `A` expresso em radianos.

7. **cosh (A)**

   Retorna o coseno hiperbólico do ângulo `A` expresso em radianos.

8. **exp (N)**

   Retorna o número de Neper (e) elevado à potência `N`.

9. **fabs (N)**

   Retorna o valor absoluto do número real `N`.

10. **floor (N)**

Retorna o arredondamento para baixo do número real `N`.

11.fmod (A, B)

Retorna o resto da divisão inteira dos reais `A` e `B`.

12.frexp (N, E)

Decompõe o número real `N` em uma mantissa real e um expoente inteiro de tal forma que a mantissa é retornada pela função e o expoente é retornado no parâmetro `E` passado à função por referência.

13.ldexp (M, E)

A função retorna o número real expresso pala mantissa `M` e pelo expoente `E`.

14.log (N)

Retorna o logaritmo natural do real `N`.

15.log10 (N)

Retorna o logaritmo de base 10 do real `N`.

16.modf (N, I)

Decompõe o número real `N` em suas partes inteira a fracionária. A parte fracionária é retornada pela função e a parte inteira é retornada no parâmetro `I` passado à função por referência.

17.pow (B, E)

Retorna a base `B` elevada ao expoente `E`.

18.sin (A)

Retorna o seno do ângulo `A` expresso em radianos.

19.sinh (A)

Retorna o seno hiperbólico do ângulo `A` expresso em radianos.

20.sqrt (N)

Retorna a raiz quadrada do número real `N`.

`21.tan (A)`

Retorna a tangente do ângulo `A` expresso em radianos.

`22.tanh (A)`

Retorna a tangente hiperbólica do ângulo `A` expresso em radianos.

Anexo II: Funções de Tempo (#include <time.h>)

Nesta biblioteca temos definidos três tipos, a saber: `clock_t`, `time_t` e `tm`. Os dois primeiros tipos permitem representar a data e o horário do sistema em um número inteiro longo, enquanto o último tipo define uma estrutura de dados que da acesso de forma decomposta a esses mesmos elementos. A estrutura `tm` e definida como segue:

```
struct tm
{
      int tm_sec;    /* segundos                                  - de 0 a  59 */
      int tm_min;    /* minutos                                   - de 0 a  59 */
      int tm_hour;   /* horas                                     - de 0 a  23 */
      int tm_mday;   /* dia do mês                                - de 1 a  31 */
      int tm_mon;    /* mes (primeiro = janeiro)                  - de 0 a  11 */
      int tm_year;   /* ano a partir de 1900                      - de 0 a  99 */
      int tm_wday;   /* dia da semana (primeiro = Domingo)        - de 0 a   6 */
      int tm_yday;   /* dia do ano                                - de 0 a 365 */
      int tm_isdst; /* horário de verao (>0: Sim, 0: Não, <0: Ñ/Disponível) */
};
```

Além disso, também encontramos definida nessa biblioteca a macro CLK_TCK que expressa o número de ciclos de máquina por segundo.

Veja abaixo algumas das principais funções definidas nesta biblioteca.

1. asctime (T)

Retorna um *string* que expressa a conversão de `T` do tipo `struct TM` para o formato "SSS MMM DD HH:MM:SS AAAA", onde SSS são as iniciais do dia da semana, MMM são as iniciais do mês, DD é o dia, HH é a hora, MM são os minutos, SS são os segundos, e AAAA é o ano.

2. clock (T)

Retorna o valor aproximado do tempo de execução do programa que a chama expresso em ciclos de máquina.. Para transformar esse valor em segundos, basta dividi-lo por CLK_TCK. Devolve –1 quando essa informação de tempo não está disponível.

3. ctime (T)

Retorna um *string* que expressa a conversão de `T` do tipo `time_t*` para o formato "SSS MMM DD HH:MM:SS AAAA", onde SSS são as iniciais do dia da semana, MMM são as iniciais do mês, DD é o dia, HH é a hora, MM são os minutos, SS são os segundos, e AAAA é o ano.

4. difftime (T1, T2)

Retorna a diferença expressa em segundos entre `T1` e `T2` (`T2 - T1`), ambos do tipo `time_t`.

5. gmtime (T)

Retorna, sob a forma de uma `struct TM*` (a estrutura deve ser copiada para uma variável própria), o GMT equivalente a `T` do tipo `time_t`.

6. localtime (T)

Retorna, sob a forma de uma `struct TM*` (a estrutura deve ser copiada para uma variável própria), o horário local equivalente a `T` do tipo `time_t`.

7. mktime (T)

Retorna sob a forma de uma de um `time_t`, o horário equivalente a `T` do tipo `struct TM*`.

8. strftime (S, M, F, T)

Esta função opera de forma semelhante a função `sprintf`, i.e., reconhece um conjunto de especificações de formato em um string de formatação, e coloca sua saída formatada em um *string* de saída. As informações que é capaz de formatar incluem, entre outras, a data e horário do sistema.

Para tanto, emprega as especificações de formato encontrados no *string* `F` e as informações de tempo presentes em `T` do tipo `struct TM*`. No máximo `M` caracteres são colocados no *string* de saída `S`.

As especificações de formato reconhecidas são as seguintes:

- `%a`, nome do dia da semana abreviado;

- `%A`, nome do dia da semana por extenso;
- `%b`, nome do mês abreviado;
- `%B`, nome do mês por extenso;
- `%c`, *string* de data e hora padrão;
- `%d`, dia do mês (de 01 a 31);
- `%H`, hora (de 00 a 23);
- `%I`, hora (de 01 a 12);
- `%j`, dia do ano (de 1 a 366);
- `%m`, mês (de 01 a 12);
- `%M`, minutos (de 01 a 59);
- `%p`, equivalente local de AM e PM;
- `%S`, segundos (01 a 59);
- `%U`, semana do ano (de 00 a 52, sendo domingo o primeiro dia);
- `%w`, dia da semana (de 0 a 6, sendo 0 o domingo);
- `%W`, semana do ano (de 00 a 52, sendo segunda-feira o primeiro dia);
- `%x`, *string* de data padrão;
- `%X`, *string* de horário padrão;
- `%y`, ano sem século (de 00 a 99);
- `%Y`, ano com século (de 0000 a 9999);
- `%Z`, nome do *time zone* local; e
- `%`, para o caractere `%`.

`9. time (T)`

Obtém o horário atual do sistema, retornando-o e também colocando-o em `T` (parâmetro do tipo `time_t` passado por referência). Retorna –1 se o horário atual não pode ser obtido. Aceita `NULL` em `T`, caso em que atua como se a função não recebesse parâmetro algum.

Anexo III: Funções de Alocação Dinâmica (#include <stdlib.h>)

1. calloc (N, T)

Aloca memória suficiente para armazenar `N` objetos de tamanho `T`, devolvendo um ponteiro sem tipo (`void*`) para o primeiro byte da região alocada. Retorna `NULL` se não conseguir realizar a alocação pretendida.

2. free (P)

Libera a memória apontada pelo ponteiro P. Essa memória deve ter sido previamente alocada com `malloc`, `calloc` ou `realloc`.

3. malloc (B)

Aloca `B` bytes de memória, devolvendo um ponteiro sem tipo (`void*`) para o primeiro byte da região alocada. Retorna `NULL` se não conseguir realizar a alocação pretendida.

4. realloc (P, T)

Modifica o tamanho da memória previamente alocada apontada pelo ponteiro P para T. O tamanho T pode ser maior ou menor que o original. Devolve um ponteiro para o bloco de memória redimensionado, que pode permanecer no endereço antigo ou pode ter sido movido na memória.

No caso desta função não conseguir simplesmente modificar o tamanho do bloco de memória original, esta aloca em outra região de memória um bloco do tamanha pretendido e move o conteúdo do bloco antigo para o novo bloco antes de liberá-lo, de forma que o conteúdo prévio fica preservado.

Se P for NULL, esta função simplesmente aloca um bloco de memória e devolve um ponteiro para a memória alocada. Se T é zero, a função simplesmente libera a memória apontada por P. Retorna `NULL` se não conseguir realizar a alocação pretendida e o bloco original é preservado.

Anexo IV: Funções Diversas

Parte I: #include <stdio.h>

Na biblioteca <stdio.h> temos definidos dois tipos, a saber: `div_t` e `ldiv_t` que são os tipos dos valores devolvidos por `div` e `ldiv`. Ainda nesta biblioteca temos definido o tipo `size_t` que é o tipo devolvido por `sizeof`.

Além desses tipos, a biblioteca define ainda as constantes `NULL` (que representa um ponteiro nulo), `RAND_MAX` (que representa o maior número aleatório que pode ser devolvido por rand), `EXIT_FAILURE` (que representa o valor devolvido para o processo que chamou o processo em execução, no caso deste último ter terminado com insucesso) e `EXIT_SUCCESS` (que representa o valor devolvido para o processo que chamou o processo em execução, no caso deste último ter terminado com sucesso).

1. abort ()

Provoca a interrupção imediata da execução do processo em execução, devolvendo ao processo que o chamou `EXIT_FAILURE`. Não fecha um único arquivo antes de executar.

2. abs (N)

Retorna o valor absoluto do inteiro fornecido como argumento.

3. atexit (F)

Registra a função apontada por F como sendo a função a ser executada quando do término normal do programa. A função apontada por F deve ser um função quer não receba e nem retorne nenhum valor.

Devolve o inteiro zero no caso do registro ter sido feito com sucesso, e devolve um inteiro diferente de zero, caso contrário.

Pelo menos 32 funções te terminação podem ser estabelecidas, e estas executarão na ordem inversa do seu registro.

4. atof (S)

Converte o *string* S, que contém um número real válido expresso em ASCII, em um `float`.

5. atoi (S)

Converte o *string* S, que contém um número inteiro válido expresso em ASCII, em um int.

6. bsearch (C, V, N, T, F)

Realiza uma busca binária no vetor ordenado `V` (que tem `N` elementos e cada elemento tem `T` bytes) que coincide com a chave de busca `C`. A função apontada por `F` deve comparar um elemento de `V` com a chave e deve ter o seguinte formato:

```
int Comparacao (void *A, void *B)
```

e deverá devolver um número menor que zero, no caso de `A` ser menor que `B`, um número igual a zero no caso de `A` ser igual a `B`, e um número maior que zero, caso contrário.

7. div (N, D)

Os inteiros N e `D` representam, respectivamente, o numerador e o denominador de uma certa fração dada. Devolve uma estrutura do tipo `div_t` definida em `<stdlib.h>` pelo menos com os atributos `quot` (que expressa a divisão de N por D) e `rem` (que expressa o resto da divisão inteira de `N` por `D`.

8. exit (C)

Provoca a interrupção imediata do processo em execução devolvendo o código de erro C ao processo que o chamou.

9. getenv (S)

Devolve um *string* que contém o valor da variável de ambiente cujo nome está expresso no *string* `S`.

10. labs (N)

Retorna o valor absoluto do inteiro longo fornecido como argumento.

11. ldiv (N, D)

Os inteiros longos N e D representam, respectivamente, o numerador e o denominador de uma certa fração dada. Devolve uma estrutura do tipo `div_t` definida em `<stdlib.h>` pelo menos com os atributos `quot` (que expressa a divisão de N por D) e `rem` (que expressa o resto da divisão inteira de `N` por `D`.

12.qsort (V, N, T, F)

Ordena pelo método *quick sort* um vetor `V` (que tem `N` elementos e cada elemento tem `T` bytes). A função apontada por `F` deve comparar dois elementos de `V` e seu formato deve precisa ser o seguinte:

```
int Comparacao (void *A, void *B)
```

A função apontada por `F` deverá devolver um número menor que zero, no caso de `A` ser menor que `B`, um número igual a zero no caso de `A` ser igual a `B`, e um número maior que zero, caso contrário.

13.rand ()

Gera uma seqüência de números aleatórios. Cada vez que é chamada, devolve um número inteiro entre zero e RAND_MAX

14.srand (N)

Indica o inteiro N para ser a semente do algoritmo de geração dos números aleatórios devolvidos por rand.

15.strtod (S, R)

Converte o *string* `S`, que contém um número real válido expresso em ASCII, em um `double`. No final da operação o ponteiro para caractere `R` passado por referência apontará para o primeiro caractere de `S` após o `double` que não entra na composição de um `double`.

16.strtol (S, R, B)

Converte o *string* `S`, que contém um número inteiro válido expresso em ASCII, em um `long int`. No final da operação o ponteiro para caractere `R` passado por referência apontará para o primeiro caractere de `S` após o `long int` que não entra na composição

de um `long int`. O inteiro `B` indica a base em que se encontra expresso o número inteiro em questão.

17.strtoul (S, R, B)

Converte o *string* `S`, que contém um número inteiro válido expresso em ASCII, em um `unsigned long int`. No final da operação o ponteiro para caractere `R` passado por referência apontará para o primeiro caractere de `S` após o `unsigned long int` que não entra na composição de um `unsigned long int`. O inteiro `B` indica a base em que se encontra expresso o número inteiro em questão.

18.system (S)

Passa o *string* `S` para o processador de comandos do sistema operacional.

Parte II: #include <assert.h>

1. assert (E)

Escreve mensagens que informam a falha da asserção E dada, no caso da expressão E resultar false, e não tem nenhum efeito, caso contrário.

Parte III: #include <setjmp.h>

1. longjmp (B, S)

Esta função representa um meio de fazer desvios incondicionais entre funções.

Faz com que a execução do programa prossiga a partir do ponto onde, pela última vez, a função `setjmp` foi chamada e atribuída a `B` (que e do tipo `envbuf`).

S representa o valor retornado por `setjmp` e possibilita a determinação da origem do desvio.

2. setjmp (B)

Esta função representa um meio de fazer desvios incondicionais entre funções. Salva em B (que é do tipo `envbuf`) as informações necessárias para `longjmp`). Retorna o valor que a chamada de `longjmp` receber como segundo argumento.

Parte IV: #include <signal.h>

1. raise (S)

Esta função envia o sinal inteiro S ao processo executor do processo em execução. Devolve zero em caso de sucesso, e um número diferente de zero caso contrário.

2. signal (S, F)

Esta função define que a função cujo endereço vem em F deve ser executada quando o sinal inteiro S for recebido.

F pode ser o endereço de uma função sem retorno e sem argumentos ou então as constantes SIG_DFL (indica que se deseja usar o tratador de sinal padrão) ou SIG_IGN (indica que se deseja ignorar o sinal).

Anexo V: Exercícios Propostos

1. Escreva uma macro que troca o conteúdo de duas variáveis inteiras. Dica: blocos podem ajudar.
2. Escreva uma função que converte graus Farenheit em graus Célcius. Saiba que C = 5/9.(F-32).
3. Escreva uma função que converte uma cadeia de caracteres que contém somente dígitos hexadecimais em um número inteiro.
4. Escreva uma função que converte um número inteiro em uma cadeia de caracteres que contém somente dígitos binários.
5. Escreva uma função com 3 argumentos similar à função *itoa*. Nela, o terceiro argumento indicará o comprimento da cadeia de caracteres resultante (deve-se completar com brancos a esquerda se necessário).
6. Escreva uma função que converte uma cadeia de caracteres em um *float*. A função deve lidar também com números em notação científica.
7. Escreva uma função que converte eventuais caracteres invisíveis que compõem uma cadeia de caracteres em seqüências de \, e.g., <TAB> por \t.
8. Escreva uma função que elimine os brancos supérfluos de uma cadeia de caracteres, i.e., que retire os brancos iniciais e finais, e que substitua cadeias com dois ou mais brancos por apenas um branco.
9. Escreva uma função que manipule cadeias de caracteres e e expanda subcadeias da forma "<$Char_0$>-<$Char_n$>" em subcadeias da forma "<$Char_0$><$Char_1$> ... <$Char_n$>". Assim, a cadeia de caracteres "qweA-Epio0-4mbn" seria expandida na cadeia "qweABCDEpio01234mbn".
10. Escreva uma função que converta letras minúsculas, que eventualmente componham uma cadeia de caracteres, em letras maiúsculas.
11. Rescreva a função acima, usando expressões condicionais (□?□:□).

12. Escreva uma função que inverta uma cadeia de caracteres.

13. Escreva uma função recursiva que inverta uma cadeia de caracteres.

14. Escreva uma função para verificar se uma cadeia de caracteres é palíndrome.

15. Escreva uma função que troque os caracteres <TAB> de uma cadeia de caracteres por 8 caracteres brancos.

16. Escreva uma função que troque subcadeias de 8 brancos de uma cadeia de caracteres por um caractere <TAB>.

17. Escreva uma função para localizar a posição mais a direita de uma dada subcadeia em uma cadeia de caracteres.

18. Escreva uma função para deletar todas as ocorrências de um certo caractere de uma cadeia de caracteres.

19. Escreva uma função para deletar de uma cadeia de caracteres C todas as ocorrências dos caracteres que compõem a cadeia de caracteres P.

20. Escreva uma função que inspeciona uma cadeia de caracteres C e retorna a primeira posição em que pode ser encontrado um caractere dos que compõem a cadeia de caracteres P.

21. Escreva uma função que retorne para o buffer de leitura uma cadeia de caracteres.

22. Escreva uma função que realiza a busca binária de uma cadeia de caracteres em um vetor de cadeias de caracteres.

23. Escreva uma função que roda um dado número de vezes os bits de um número inteiro.

24. Escreva uma função que elimina de uma lista de números todas as ocorrências de um dado elemento.

25. Escreva uma função para inverter uma lista.

26. Escreva uma função para concatenar duas listas.

27. Escreva uma função para incluir ordenadamente um elemento em uma lista de números.

28. Escreva uma função para excluir um elemento de uma lista de números.

29. Escreva uma função para ordenar uma lista numérica.

30. Escreva uma função para fazer a intercalação de duas listas numéricas ordenadas.

31. Escreva um predicado para comparar duas listas.

32. Escreva uma função que conta as linhas de um texto.

33. Escreva uma função que conta as palavras de um texto.

34. Escreva um programa que copia um arquivo em outro.

35. Escreva um programa que copia um arquivo na saída padrão substituindo linhas idênticas em seqüência por apenas uma linha.

36. Escreva um programa que copia para a saída padrão as n últimas linhas da entrada padrão.

37. Escreva um programa que compara dois arquivos e imprime na saída padrão a primeira linha onde difiram.

38. Escreva um programa que busca um padrão em um texto.

39. Escreva um programa que calcula a incidência percentual dos caracteres em um texto.

40. Escreva um programa que não mutila palavras e que quebra as linhas de um texto em linhas de no máximo 80 caracteres.

41. Escreva um programa que remove todos os comentários de um programa C. Não esquecer coisas do tipo " ... /* ...".

42. Escreva um programa que processa um arquivo e substitui todas as ocorrências de caracteres não imprimíveis em números hexadecimais delimitados por barras (/hexa/).