Um Guia para Programação em *C*

"vade meCum"

Terceira Edição, 1998

Julio M. Stern e Routo Terada

$$
\begin{array}{cccc}
\downarrow & \mathrm{C} & \mathrm{P} & \mathrm{L} \\
\mathrm{B} & \mathrm{C} & \mathrm{P} & \mathrm{L} \\
\mathrm{B} & \downarrow & \downarrow & \downarrow \\
\downarrow & \mathrm{C} & \downarrow & \downarrow \\
\downarrow & \mathrm{C} & + & +
\end{array}
$$

Depto. de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Este texto é destinado primordialmente aos alunos de Introdução à Computação que o Depto. de Ciência da Computação oferece em um semestre letivo aos alunos do primeiro ano em várias faculdades da USP. Recomendamos que seja complementado por listas de exercícios e exercícios-programa disponíveis aos professores e alunos neste mesmo Depto.

Agradecemos ao Instituto de Matemática e Estatística da Universidade de São Paulo e aos alunos e colegas do Depto. de Ciência da Computação pelas sugestões e incentivos na elaboração deste texto.

Quaisquer sugestões para o aprimoramento deste trabalho são bem-vindas.

Um Guia para Programação em *C* – “vade meCum”

Julio M. Stern (jstern@ime.usp.br) é Físico e Mestre em Matemática Aplicada pela USP, e Ph.D. em Pesquisa Operacional e Engenharia Industrial pela Universidade de Cornell (N. York, EUA)

Routo Terada (rt@ime.usp.br) é Engenheiro Eletricista-Eletrônico e Mestre em Matemática Aplicada pela USP, e Ph.D. em Ciência da Computação pela Universidade de Wisconsin-Madison (Wisconsin, EUA).

Um guia para programação em C – “vade meCum”

Julio M. Stern e Routo Terada

Depto. de Ciência da Computação – IME – USP

Contents

1 Por que *C* ?

A linguagem *C* é derivada de outras anteriores: *CPL* (1963), *BCPL* (1967), e *B* (1970). Existem bons motivos para escolher *C* como linguagem de programação para um curso intermediário, ou mesmo um curso básico, em Ciência da Computação. Os melhores atributos de *C* são, ao nosso ver, dois: portabilidade e versatilidade. Por portabilidade entendemos disponibilidade e compatibilidade. Disponibilidade significa que *C* está disponível em virtualmente qualquer computador, do menor dos micros aos mais modernos supercomputadores com arquiteturas as mais exóticas. Quanto à disponibilidade, só FORTRAN rivaliza com *C*. Compatibilidade significa que todas as implementações da linguagem diferem muito pouco entre si. *C* foi originalmente desenvolvida tendo em vista programação de sistemas, com o objetivo de facilmente portar estes sistemas para diferentes arquiteturas e máquinas. A aplicação em si era muito intolerante com incompatibilidades, e o principal sistema desenvolvido em *C*, o sistema operacional UNIX, tornou-se rapidamente popular, juntamente com sua linguagem base, a *C*.

Por versatilidade entendemos que *C* pode não ser a primeira escolha para uma aplicação específica, mas será quase sempre uma boa alternativa. Assim, por exemplo, *C* não é tão didática e "limpa" como Pascal, uma linguagem que pela clareza e elegância com que enfatiza certos conceitos é uma excelente escolha para um primeiro curso. Todavia, evitando abusar de construções arcanas e convolutas, é fácil ressaltar em *C* todos os bons hábitos de programação estruturada. Código executável gerado via linguagem *C* ainda perde uns vinte por cento de eficiência em relação ao correspondente código em FORTRAN para certas aplicações em análise numérica; não é tão conveniente como SNOBOL, LISP ou AWK para algumas aplicações de manipulação simbólica; não oferece diretamente incorporadas à linguagem todas as facilidades para manipulação de arquivos, dados e processos do COBOL ou PERL; nem a orientação a objetos da *C++*; mas é uma boa alternativa (muitas vezes a única disponível) para muitas destas aplicações. A versatilidade da linguagem *C* se reafirma em projetos em que temos que lidar com vários aspectos-alvo de linguagens específicas simultaneamente, como cálculo numérico em estruturas de dados complexas, manipulação e interpretação simbólica para aplicações em sistemas, etc. Por fim, conta como motivação adicional para aprender *C* o fato de várias linguagens modernas, das anteriormente citadas AWK, *C++* e PERL, terem uma sintaxe baseada e assemelhada à da *C*.

Este livreto é um *vade mecum* - literalmente: venha comigo - um guia para ajudá-lo a escrever seus primeiros programas em *C*. De maneira nenhuma esta é uma obra de referência sobre *C*: Alguns aspectos mais periféricos e intrincados da linguagem, cujo emprego não é necessário ou mesmo recomendável num primeiro curso, não são mencionados. Outros aspectos foram simplificados, não tendo sido apresentados com a máxima generalidade. Finalmente não mencionamos formas arcaicas da linguagem, que de qualquer forma não deveriam ser mais empregadas após o estabelecimento do padrão ANSI. Procuramos contudo dar uma visão global e coerente da linguagem, de modo que quaisquer detalhes suplementares possam ser facilmente assimiláveis com a ajuda de uma obra de referência, como [Kernighan88].

A ordem com que os vários tópicos da linguagem são apresentados segue a ordem com que estes se tornam necessários didaticamente, começando com a descrição de algoritmos simples, e terminando com a construção de estruturas de dados complexas. Exemplos completos podem ser vistos na Seção 19.

Table 1: Lista das 28 palavras-chave reservadas em C

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

2 Características do Compilador C. Depuração.

Uma nota sobre estilo: Compiladores C sempre distinguem letras maiúsculas de minúsculas, e é usual escrever quase todo o texto dos programas em minúsculas. Maiúsculas são usadas geralmente apenas para nomes de constantes, macros, etc.

Toda a linguagem C é constituida de algumas poucas palavras-chave: 28. Elas tem significado especial (que veremos nas seções seguintes), e seu uso é de caráter reservado, e portanto não devem ser usadas pelo programador para outros significados. Veja a Tabela 1.

Para criarmos um programa em C, inicialmente precisamos de um editor de texto (como o Emacs ou o editor do Turbo-C ou do QuickC) para digitar e armazenar em disco magnético o texto em linguagem C, constituindo o *arquivo-fonte* do programa. A seguir, esse programa deve ser traduzido para a *linguagem de máquina* do computador sendo usado, pelo chamado *compilador C*; este gera *automaticamente* uma seqüência de processos de transformação do arquivo-fonte em *arquivo-executável*. No micro-computador, este arquivo termina em ".exe". Ilustramos na Tabela 2 estas etapas de transformação do arquivo-fonte em arquivo-executável. Mais detalhes sobre pré-processamento podem ser encontrados na Seção 8.

Após a criação do arquivo-executável, este deve ser executado para sabermos se o seu comportamento é da forma que imaginamos. Se não o for, tentamos encontrar e corrigir o(s) erro(s) (de lógica) no arquivo-fonte (via o editor de texto) e compilá-lo e executá-lo novamente. Este processo é chamado de *depuração (ou "debugging") do programa*, e costuma ser demorado se o programador for inexperiente ou descuidado. Recomendamos que o programador organize e "limpe" bem o seu programa ainda no papel, ANTES de iniciar este processo de depuração, para evitar demora e desgaste desnecessários. A experiência mostra que se o programador dedicar 70 porcento do tempo total gasto em um programa ANTES do início da depuração, muitos dos erros de lógica podem ser sanados ANTES do processo de depuração.

Para auxiliá-lo em depuração, veja a Seção 18 na página 30. Ademais, recomendamos "ferramentas" de auxílio a depuração como o sdb, CodeView, ou TurboDebugger.

3 Tipos Básicos e Operações Básicas

São tipos básicos em C : `char`, um caractere; `int`, um inteiro; `flutuante (float)`, um número em ponto flutuante; e `double`, um flutuante em precisão dupla. Outros tipos

Table 2: Etapas de transformação pelo compilador *C*

Editor de texto
↓
Arquivo-fonte em *C*
↓
Pré-processador *C*
↓
Arquivo-fonte expandido em *C*
↓
Compilador *C* próprio
↓
Arquivo em linguagem simbólica
↓
Montador ("Assembler")
↓
Arquivo-objeto
(deste programa e de arquivos de biblioteca)
↓
Linker
(vincula este programa à biblioteca)
↓
Arquivo-executável

básicos são especificados aplicando qualificadores como `short`, `long`, `signed` e `unsigned`. Assim `long double` pode especificar um flutuante em precisão quádrupla, e `long int` uma representação de inteiros usando mais posições de memória que `int`. Os detalhes de como cada tipo básico é representado depende da implementação da linguagem, i.e. da arquitetura do computador e do compilador.

Os valores máximo e mínimo de cada tipo são identificados usualmente no arquivos de sistema chamados limits.h ou float.h, que podem ser usados via `#include <nome-do-arquivo>`, como explicado na Seção 8. Listamos na na Tabela 3 alguns destes identificadores e seus valores usuais.

Algumas idiossincrasias: Constantes inteiras decimais não devem ser escritas começando por zero, caso contrário serão interpretadas como constantes octais: `011`$= 1*8^1+1*8^0 = 9$. Constantes em ponto flutuante têm que conter o ponto, ou estar em notação científica, assim, se `x` é do tipo `flutuante`, `x=10.0;` ou `x=1E+1;` têm o resultado esperado, mas `x=10;` não. `1E+1` significa `1` multiplicado por `10` elevado à potência `+1`.

A Tabela 4 nos dá a ordem de avaliação em expressões envolvendo todos os operadores da *C* que descreveremos. Assim a expressão `(3+5/2/3)` é calculada como `(3+(5/2/3))`, pois a precedência do operador `+` é menor que a do operador `/`, e esta última expressão é equivalente a `(3+((5/2)/3))`, pois o operador `/` se associa da esquerda para a direita. Colocar parênteses que facilitem a leitura de expressões intrincadas não prejudica ninguém, e mostra bom estilo e educação. Um alerta: A precedência e associatividade dos operadores implicam numa ordem de avaliação dos operadores de uma expressão, mas a ordem

Table 3: Tabela de Identificadores

Identificador	Valor	Significado
`SHRT_MAX`	$+32767$	valor máximo de `short`
`SHRT_MIN`	-32767	valor mínimo de `short`
`INT_MAX`	$+32767$	valor máximo de `int`
`INT_MIN`	-32767	valor mínimo de `int`
`LONG_MAX`	$+2147483647$	valor máximo de `long`
`LONG_K`	-2147483647	valor mínimo de `long`
`UINT_MAX`	65535	valor máximo de `unsigned int`
`ULONG_MAX`	4294967295	valor máximo de `unsigned long`
`USHRT_MAX`	65535	valor máximo de `unsigned short`
`FLT_MAX`	$1E+37$	valor máximo de `float`
`FLT_MIN`	$1E-37$	valor mínimo de `float`

Table 4: Ordem de avaliação de operadores

Precedência	Associatividade
`() [] -> .`	$\longrightarrow$
`! ++ -- - (tipo) * & sizeof`	$\longleftarrow$
`* / %`	$\longrightarrow$
`+ -`	$\longrightarrow$
`< <= > >=`	$\longrightarrow$
`== !=`	$\longrightarrow$
`&&`	$\longrightarrow$
`||`	$\longrightarrow$
`?:`	$\longleftarrow$
`= += -= *= /= %=`	$\longleftarrow$

de avaliação dos operandos de um operador é, salvo algumas expressões, dependente do compilador. Assim, o exemplo `x=1; y=(++x)+(x);`, pode atribuir a `y` o valor 3 ou 4, dependendo da ordem em que os operandos da adição forem avaliados.

Em caso de dúvida *sempre* inclua parênteses explicitando a ordem desejada de avaliação.

Os operadores de comparação – `==` *igual*, `!=` *diferente*, `<` *menor*, `<=` *menor-ou-igual*, etc. – são avaliados com os valores `1` (verdadeiro) ou `0` (falso), conforme a igualdade ou desigualdade de seus operandos. Observe a precedência dos operadores lógicos – `!` *não*, `&&` *e*, `||` *ou* – e a dos operadores de comparação, que tornam desnecessário sobrecarregar de parênteses muitas expressões.

Em expressões condicionais [1] qualquer valor não nulo é considerado verdadeiro, equiv-

[1] como em `if`(expressão condicional) ou em `while`(expressão condicional)

> Em expressões condicionais qualquer valor não nulo é equivalente a verdadeiro.

Assim, `!v` vale `1` se e só se `v` vale `0`. E `v&&w` vale `1` se e só se `v` e `w` não valem `0`. E `v||w` vale `0` se e só se `v` e `w` valem `0`.

Alguns exemplos de expressões condicionais: `(a<=10.2) && chovendo || (i>j+1)`, `!acabou || j==(M+51)`, `acabou && j!=(M+51)`.

A expressão `a%b`, `a` *módulo* `b`, vale o resto da divisão de `a` por `b`. Assim, `(13%5==3)` vale `1`. A expressão condicional `(condicao?a:b)` tem valor `a` se a `condicao` é verdadeira, e valor `b` caso contrário, assim `(a>b?a:b)` vale $max(a, b)$.

Atribuições do tipo `y+=x;` ou `y*=x;` são apenas abreviações para as atribuições `y=y+x;` e `y=y*x;`. Os operadores de *incremento* e *decremento*, `++` e `--` , aumentam ou diminuem de uma unidade o valor de uma variável inteira. Uma variável pós-fixada pelo operador de incremento, `k++` , é primeiro avaliada e depois incrementada; uma variável pré-fixada é primeiro incrementada e depois avaliada. Assim, são equivalentes os comandos abaixo que estão na mesma linha.

```
y=++k;                  { k+=1; y=k; }
y=k++;                  { y=k; k+=1; }
```

O ponto-e-vírgula (`;`) ; é o *terminador* de comandos simples (e não um *separador* como em outras linguagens). O caractere nova-linha (`\n`), o espaço-em-branco e a marca-de-tabulação (`\t`), todos estes chamados *brancos*, são apenas separadores, de modo que:

```
                                  x =
  x=y+z;          e                 y
                                   +z  ;
```

significam exatamente a mesma coisa.

Declaração de variáveis

As variáveis a serem usadas num programa precisam ser *declaradas*, em princípio para que o compilador lhes reserve o espaço necessário. Assim, as sentenças

```
int i, j, k;
float x[10], y, z;
```

declaram 3 variáveis de *tipo* inteiro, `i`, `j` e `k`, duas variáveis de *tipo* flutuante, `y` e `z`, e um *array* ou vetor com 10 posições de *tipo* flutuante. Um nome de variável pode ser qualquer seqüência de letras e dígitos, começando por uma letra. Letras são as maiúsculas, as minúsculas e a sublinha "_". Arrays em C sempre são indexados a partir de 0, de modo que com a declaração acima podemos nos referir a `x[0]`, `x[1]`, até `x[9]`, mas não a `x[10]`. Mais detalhes sobre arrays na Seção 9.

4 Funções Básicas de Entrada e Saída de Dados

As funções explicadas nesta Seção exigem `#include <stdio>` como usado na Seção 19, no início do arquivo-fonte.

Os dados ou informações a serem processados por um programa devem ser fornecidos pelo usuário do programa: são os chamados *dados de entrada*. Diz-se então que o programa "lê" os dados de entrada. Em *C*, a funo básica para a entrada de tais dados é `scanf`, como em: `scanf("%d %f", &i, &y);` Note que tanto `i` como `y` devem ser precedidos pelo caractere especial `&`. Esta funo, ao ser executada, vai atribuir um valor inteiro à variável `i`, e um valor flutuante à variável `y`. Estes dados o usuário deve digitar no *dispositivo padrão de entrada*, em geral o teclado, em uma mesma linha, como em `1.55 1901`, seguido por toque da tecla ENTER. O branco separando os dois números é necessário. Alternativamente, no lugar de um branco, pode-se ter uma vírgula, como em `1.55,1901`, ou outro símbolo separador (exceto obviamente o ponto decimal: por quê?).

Os dados a serem exibidos ao usuário são os chamados *dados de saída*. Diz-se então que o programa "imprime" ou "grava" os dados. Em *C*, a funo básica para a saída de tais dados é `printf`, como em:

```
printf("y vale %f e i vale %d \n", y, i);
```

que, ao ser executado, provoca a saída do tipo `y vale 1.55` e `i vale 1901` no *dispositivo padrão de saída*, em geral a tela do terminal. `\n` que ocorre na funo, entre aspas, é para provocar um *pulo* para uma nova linha. No lugar de `y` ou de `i` pode-se colocar uma expressão aritmética, como ilustrado a seguir.

Um programa em *C* é precedido por `main(){` e terminado por `}`. Por exemplo, o programa abaixo "lê" dois números flutuantes, e exibe a sua soma e o seu produto, precedidos de explicações do tipo " ... vale ...".

```
main(){
  float x, y;
  scanf("%f %f", &x, &y);
  printf("A soma vale %f, e o produto vale %f",
           x+y, x*y);
  }
```

Vamos supor que este programa após compilado se chama `lua`. Para uma ilustração de entrada e saída de dados para o programa `lua`, veja a Figura 1.

Arquivos e Redirecionamento

Os dados de entrada podem ser lidos de um arquivo armazenado em disco magnético, sem alterar o programa escrito para ler do teclado. Por exemplo, se o arquivo-executável do programa acima se chama `lua`, e se existe um arquivo chamado `lua.ent` contendo uma linha com dois valores flutuantes, o comando `lua<lua.ent` provoca a execução de `lua` com leitura dos dados em `lua.ent`. Analogamente, os dados de saída podem ser "redirecionados", isto é, gravados em disco magnético, ao invés de serem exibidos na tela. Por exemplo, `lua <lua.ent >lua.sai` provoca a gravação dos dados de saída no arquivo chamado `lua.sai`, além da leitura dos dados em `lua.ent`.

O redirecionamento de dados por `<` ou `>` não é uma característica da linguagem *C*, mas sim do sistema operacional UNIX e MS-DOS. Em outros sistemas operacionais ele pode existir com uma outra forma ou sintaxe, ou não existir.

Figure 1: Ilustração de entrada e saída de dados

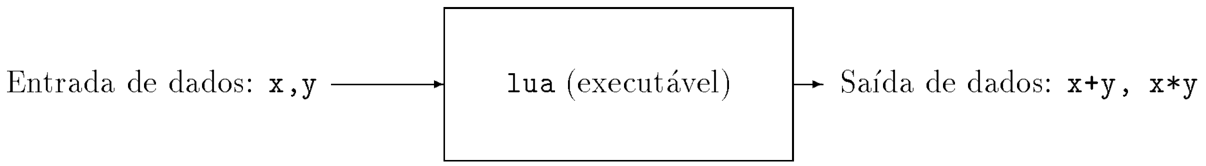


Figure 2: `for`, `while`, e `do-while`

```
main(){                   *   main(){          *   main(){
  int i, j;               *     int i, j;      *     int i, j;
  i=0;                    *     i=0; j=1;      *     i=0; j=0;
  for(j=1; j<11; j=j+1)   *     while(j<11){   *     do{
    i=i+j;                *       i=i+j;       *        j= j+1;
  printf("%d",i);         *       j=j+1;       *        i= i+j;
}                         *     }              *     }while(j<10);
                          *     printf{"%d",i};*     printf("%d",i);
                          *   }                *   }
                          *                    *
```

5 Controle de Fluxo de Execução: Laço

Um programa em C é o corpo da função principal, `main(){/*programa*/}`. A execução do programa começa após a primeira chave, `main(){`, e flui até a correspondente chave de fechamento. O fluxo de execução do programa pode ser controlado com comandos `for`, `while` e `do-while`, exemplificados abaixo. (O fluxo do programa é também alterado ao invocarmos uma função, como explicado na seção 12.)

Os três programas na Figura 2 calculam $i = \sum_{j=1}^{10} j$. No primeiro programa o comando `for` significa que `j`, a *variável* do `for`, é *inicializada* com o valor 1 (`j=1;`). O *comando* ou *laço* do `for` (i.e., `i=i+j;`) será executado com argumento `j` enquanto a *condição* do `for`, `j<11;`, for verdadeira. Após a execução do laço, será realizada a *operação* do *for*, que atribuirá à variável `j` um novo valor, no caso `j+1`. O *laço* se repete até que a condição do `for` se verifique falsa, quando o laço é interrompido (o comando do `for` não é executado), e o programa prossegue executando o comando seguinte ao `for`, que é `printf(....`

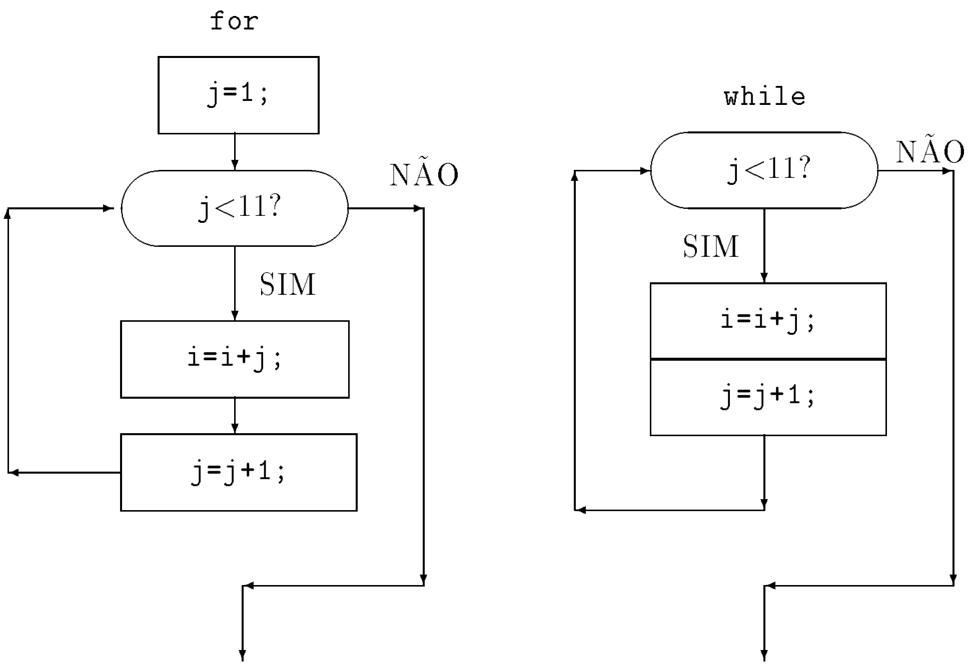
Esta é a forma em C de se dizer o equivalente a:

---

para `j=1`, faça `j=i+j;` incrementando `j` de `1` em `1` até `j` valer `10`.

Veja a Figura 3 para entender melhor como o comando `for` é. O `for` e os outros controles de fluxo em C formam um laço que repete um único comando. Se quisermos

Figure 3: Ilustração de `for` e `while`



repetir vários *comandos simples* devemos agrupá-los com um par de chaves num *comando composto*, como:

```
for(inicializacao; condicao; operacao){
  comando1; comando2; comando3;
}
comando-seguinte;
```

O segundo programa na Figura 2 usa o comando de controle de fluxo `while` (que significa "enquanto"), que repete um comando enquanto a condição de controle se verifica verdadeira. Na forma `while(condicao){comando;}` a `condicao` é verificada antes de executar o `comando`.

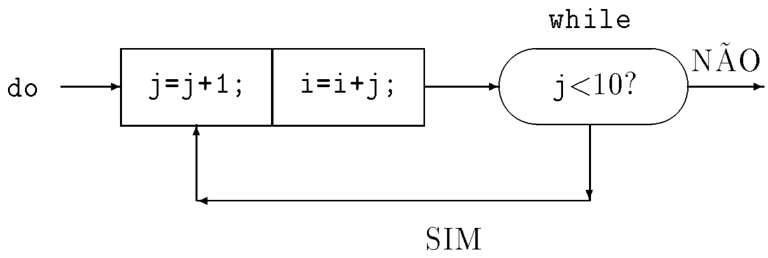
O `while` permite em *C* expressar o equivalente a "enquanto chover, fique em casa, jogue baralho e durma.":

```
while(chover){
  fique em casa;
  jogue baralho;
  durma;
  }
```

Veja a Figura 3 para entender melhor como o comando `while` funciona.

O terceiro programa na Figura 2 usa o comando de controle de fluxo `do-while` (que significa "faça-enquanto"), Na forma `do{comando;}while(condicao);` a `condicao` é ver-

Figure 4: Ilustração de `do-while`



ificada depois de executar o `comando`; assim o `comando` é executado ao menos uma vez, independentemente da `condicao`.

O `do-while` é a forma em C de se dizer o equivalente a "faça: fique em casa, jogue baralho e durma, enquanto chover.":

```
do{
  fique em casa;
  jogue baralho;
  durma;
  } while(chover)
```

Para entender melhor como o comando `do-while` funciona veja a Figura 4.

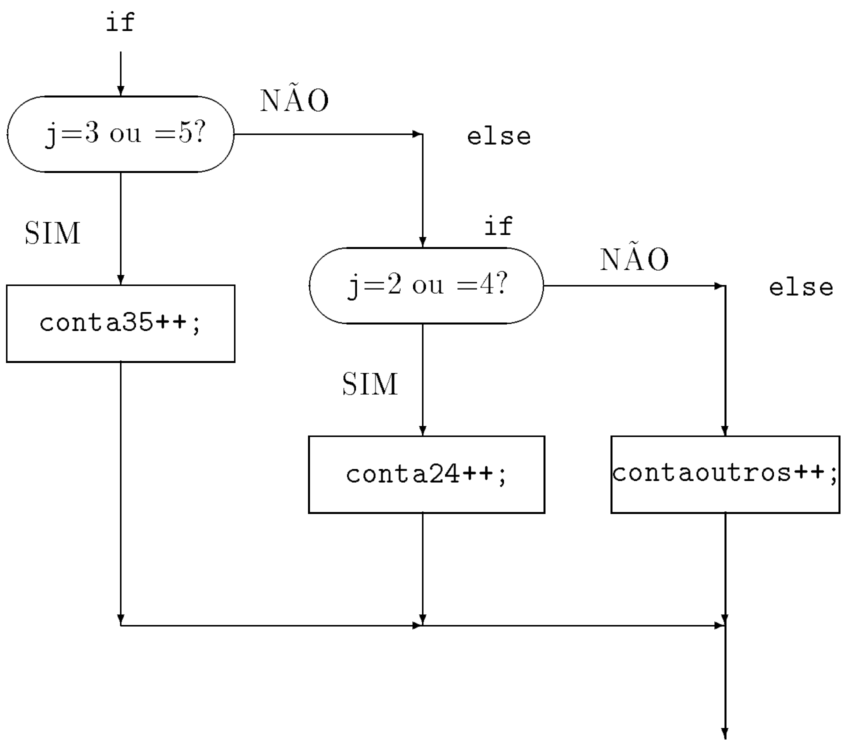
Exercício: modificar os programas dados na Figura 2 para calcular $\sum_{j=1}^{10} j^2$.

6 Comandos Condicionais `if` e `switch`

O fluxo de execução do programa pode ser controlado também com comandos como `if` e `switch`, exemplificados abaixo. O primeiro programa ilustra o uso dos comando `if` e resolve o seguinte: dos números de 1 a 100, conta quantos terminam em 3 ou 5, quantos terminam em 2 ou 4, e quantos são os outros.

```
main(){
  int i,j, conta35, conta24, contaoutros;
  for(i=1; i<101; i++){
    j=i-10*(i/10);
    if(j==3||j==5)
      conta35++;
    else
      if(j==2||j==4)
           conta24++;
      else contaoutros++;
    }
  printf("conta35 = %d, conta24 = %d, outros = %d",
          conta35, conta24, contaoutros);
  }
```

Figure 5: Ilustração de `if`



Para uma ilustração do comando `if` veja a Figura 5.

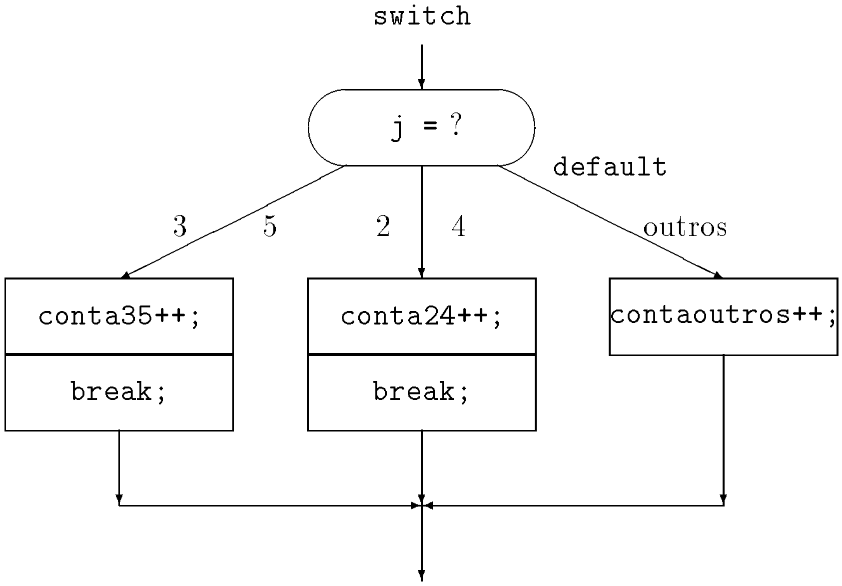
O comando `if(condicao)` suprime a execução do comando seguinte caso a sua condição seja falsa. O comando *if-else*, `if(condicao) com1; else com2;` executa o comando `com1` se a condição do `if` for verdadeira, e executa o comando `com2` caso contrário. Esta é a forma em *C* de se dizer o equivalente a "*se* não chover, então vá, *senão* fique." (`if` (!chover) vá; `else` fique;).

O programa a seguir resolve o *mesmo* problema anterior (resolvido com o comando `if`), agora com o uso do comando `switch`.

```
main(){
  int i,j, conta35, conta24, contaoutros;
  for(i=1; i<101; i++){
    j=i-10*(i/10);
    switch(j){
      case 3: case 5: conta35++;
                      break;
      case 2: case 4: conta24++;
                      break;
      default: contaoutros++;
    }
  printf("conta35 = %d, conta24 = %d, outros = %d",
          conta35, conta24, contaoutros);
  }
```

O comando `switch` na sua forma geral é:

Figure 6: Ilustração de `switch`




```
switch(expressao){
  case expressao-const1: comandos-1; break;
  case expressao-const2: comandos-2; break;
  ...
  default: comandos;
  }
```

Para uma ilustração do comando `switch` veja a Figura 6.

Este comando altera o fluxo de execução da seguinte forma: a `expressao` do `switch` é avaliada, e se o seu valor for igual a uma das expressões constantes de um dos `case`, os comandos à frente deste `case` são executados; caso contrário, os comandos à frente do `default` são executados. No exemplo acima, note que à frente do `case 3:` existe um comando "vazio", e por isso, no caso de `j` valer 3, o comando `conta35++` é executado. Analogamente para `case 2:`. Esta é a forma equivalente em *C* de se dizer "*se* a *chave* (i.e., `switch`) j for do *caso* (i.e., `case`) 3 ou 5, faça conta35++, ou *se* for do *caso* 2 ou 4, faça conta24++".

Como ilustrado também no exemplo, o comando `break` dentro do `switch` é para terminar a execução de todo o comando `switch`.

Não esqueça do `break;`. Se *não* houver `break;` a execução prossegue no primeiro comando do `case` seguinte!

Figure 7: Ilustração de `continue` e `break`

```
main(){                        *   main(){            * main(){
  int i, j;                    *     int i, j;        *   int i,j;
  i=0;                         *     i=0; j=1;        *   i=0;
  for(j=1; j<11; j=j+1){       *     while(j<11){     *   for(j=1;j<11;j=j+2)
    if(j%2==0) continue;       *       if(j==6) break;*     i=i+j;
    else i=i+j;                *       else {         *   printf("%d",i);
    }                          *         i=i+j;       * }
  printf("%d",i);              *         j=j+1;
}                              *         }
                               *     }
                               *     printf{"%d",i);
                               *   }
```

7 Comandos `continue` e `break`

Se dentro de um laço ocorrer um comando `continue`, a execução do resto do laço é suprimida, e volta para o *início* do laço. Por outro lado, se ocorrer um comando `break`, a execução de todo o laço (mais interno) é interrompida. Nos exemplos a seguir ilustramos o uso destes comandos.

O programa à esquerda na Figura 7 calcula $i = \sum_{j=1}^{10} j$, mas só para `j` ímpar, pois quando `j` é par, o comando `continue` é executado. O programa à direita na Figura 7 faz o mesmo cálculo, só que com `j=j+2;` no `for`. No programa ao meio, o laço do `while` é executado completamente apenas para `j` igual a $1, 2, ..., 5$ pois o comando `break` é executado para o valor de `j` igual a 6.

8 Pré-Processamento

O pré-processador do compilador C pode ser pensado como um processador de texto que realiza certas operações de substituição, inclusão e eliminação de trechos de texto do programa, antes que este seja compilado. Linhas com comandos de pré-processamento começam com o símbolo `#`.

Uma *macro*, `#define MACROX definicao`, simplesmente substitui no texto do programa, antes de sua compilação , cada ocorrência do nome da macro, `MACROX`, pela sua `definicao`. Um nome de macro segue as mesmas regras de formação que um nome de variável, e o que segue após o primeiro branco até o fim da linha é a definição da macro. Macros podem tomar *argumentos*. Por exemplo, o primeiro programa abaixo, será, antes de ser compilado, substituído pelo segundo programa.

```
#define N 10
#define QUAD(x) ( x ) * ( x )
#define ABS(x) ((x>=0)?(x):-(x))
#define MAX(a,b) ((a>b)?(a):(b))
```

```
main(){
   ...
  while(x<=N){
     ...
    x=QUAD(MAX(x+1,y) ;
     ...
   }
}

*************************************************************

main(){
   ....
  while(x<=10){
     ...
    x=( ((x+1)>(y)?(x+1):(y)) ) * ( ((x+1)>(y)?(x+1):(y)) ) ;
     ...
   }
}
```

O comando de pré-processsamento `#include "nome-do-arquivo"` inclui, nas linhas subseqüentes, o conteúdo do arquivo `nome-do-arquivo`. Ao invés das aspas, podemos indicar o nome de alguns arquivos padrão pré-definidos no sistema entre o `<` e o `>`, como em `#include <nome-do-arquivo>`, indicando que o arquivo está num diretório padrão no sistema.

Comentários
Comentários em C têm seu começo e seu fim marcados pelos dígrafos $/*$ e $*/$. Em C comentários NÃO podem ser aninhados.

Veja exemplos de comandos `#` e comentários na Seção 19.

9 Ponteiros, Arrays e Matrizes

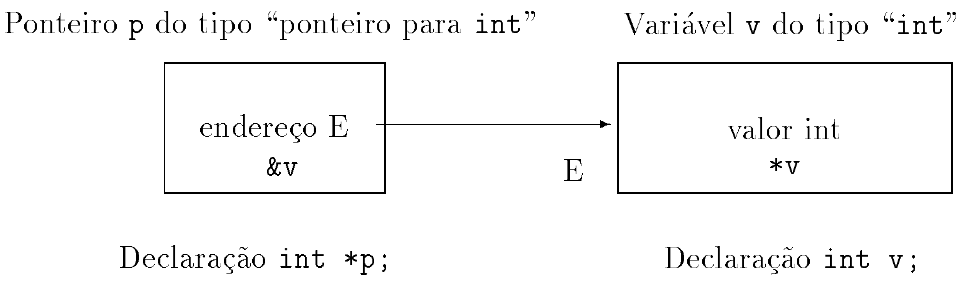
Em C a declaração `tipo-válido nome-do-objeto` significa que o compilador aloca (reserva) um endereço (posição de memória), ou uma série de endereços consecutivos, para guardar o objeto de nome `nome-do-objeto`. Para saber explicitamente qual é o primeiro destes endereços, usamos o *operador de referência*, `&` , como em `p = &nome-do-objeto;`, que atribui ao *ponteiro* `p` o primeiro destes endereços. Ponteiros são também chamados *apontadores*, e dizemos que `p` *aponta* o objeto de nome `nome-do-objeto`.

O *operador de dereferência*, `*` , fornece o objeto para o qual um ponteiro aponta. Assim se `p=&x; y = *p;` atribui a `y` o valor do objeto *apontado* por `p`, ou equivalentemente, copia `x` em `y` como na atribuição `y = x;`.

Para uma ilustração de como um ponteiro é internamente, veja a Figura 8.

Se `x` e `y` são variáveis de um *tipo* que o compilador guarda em k posições consecutivas de memória, então `y=*p` deve copiar o conteúdo das k posições alocadas para guardar `x`, nas k posições alocadas para guardar `y`. Desta última consideração fica patente que o compilador

Figure 8: Ilustração de ponteiro após execução de `p= &v;`



precisa saber para objetos de que tipo um apontador aponta; e isto fica estabelecido ao declararmos os ponteiros.

Para os exemplos dos próximos parágrafos considere as declarações:

```
int i, j, *pi, ai[10]; float x, y, *pf, *qf, af[10];
```

onde declaramos `i` uma variável inteira, `pi` um ponteiro para inteiros, `ai` um array de inteiros de 10 elementos, `x` um variável de ponto flutuante, e `pf` um apontador para variáveis em ponto flutuante, e `af` um array de 10 variáveis de ponto flutuante.

A sintaxe de declarações é fácil de entender: Cada declaração simplesmente lista objetos de um mesmo tipo. Assim, se listamos `*pf` como um flutuante, então declaramos `pf` como um apontador para flutuante. Veremos exemplos mais complexos adiante.

É um erro fazer atribuições entre tipos incompatíveis, como:

```
pf=&i; pi=&x; pi=pf; i=x; x=*pi; j=*pf; /*erros*/
```

Logo após uma declaração como `float *pf;`, o valor de `pf` é indefinido, i.e., `pf` não está inicializado, da mesma forma que outra variável de qualquer outro tipo como `x` logo após `float x;`.

Nunca esqueça de inicializar qualquer ponteiro. Só a sua declaração não implica em inicialização!

Veja erros causados por esquecimento desta natureza na Seção 18.

Array

A declaração de um *array* de N elementos aloca um bloco de memória (posições consecutivas) para guardar N objetos do tipo declarado. O operador `sizeof` nos dá o tamanho de um objeto ou tipo, i.e. quantas posições de memória são gastas para guardar um objeto, ou para guardar objetos de um determinado tipo. Assim, se flutuantes forem guardados em quatro posições de memória, então `sizeof(float)==4`, `sizeof(x)==4` e `sizeof(af)==40`.

O endereço da primeira posição de memória do array `af[]` pode ser obtido por `pf=&af[0];`, e este é precisamente o significado do nome de um array, i.e. este endereço pode também ser obtido como `pf=af;`. Analogamente, o endereço da primeira posição de memória alocada para guardar o i-ésimo elemento deste array, `af[i]`, é dada por `pf=&af[i];`.

Figure 9: Ilustração de array e ponteiros

Endereços:	`af` ou `&af[0]`	`af+1` ou `&af[1]`		`af+i` ou `&af[i]`
	`af[0] ou *pf`	`af[1] ou *(pf+1)`	···	`af[i] ou *(pf+i)`
Ends. após `pf=af;`	`pf`	`pf+1`		`pf+i`

Em C estão definidas algumas operações aritméticas com ponteiros. Toda esta aritmética de ponteiros só tem sentido se os ponteiros envolvidos apontam para objetos dentro de um bloco de memória que foi alocado para guardar objetos de um mesmo tipo, como por exemplo quando fazemos aritmética com apontadores de flutuantes dentro de um bloco alocado para um array de flutuantes.

Para uma ilustração de array e ponteiros veja a Figura 9.

Operações aritméticas com ponteiros:

- Adição : Seja `p` um ponteiro para um objeto de tamanho `sizeof(*p)`, então `q=p+i` aponta para o `i`-ésimo objeto (deste mesmo tipo) consecutivo na memória após `*p`. Portanto, se `*p` começa na posição de memória de endereço E, `*q` começa na posição $E + i * \texttt{sizeof}(*\texttt{p})$.
- Subtração : Se a soma `q=p+i;` está bem definida, a subtração `q-p` vale `i`.
- Comparação : O resultado dos operadores de comparação entre ponteiros é definido a partir do operador de subtração . Substituindo `>` por qualquer outra operação de comparação : `(q>p)` se e só se `(q-p>0)`.

Uma vez atribuído ao ponteiro `pf` o valor `pf=&af[i];` a atribuição `qf=pf+j` é equivalente a `qf=&af[i+j]`. São também equivalentes as expressões : `af[j]` e `*(af+j)`. Na verdade, esta é a definição do que significa em C o colchete!

Matriz

Em C é possível representar uma matriz como um array de arrays. Consideremos as declarações: `float x, *pf, mat[2][4], *apf[4], (*paf)[4];`

De acordo com a declaração de `mat[][]`, `float (mat[2])[4]`, cada elemento de `mat[]` é um array de 4 flutuantes, logo `mat[][]` é um array de 2 elementos, onde cada elemento é um array de 4 elementos, onde cada elemento é um flutuante. `mat[1][2]` é o terceiro elemento do segundo array, ou mais explicitamente `mat[1][2] == *(mat[1]+2) ==*(*(mat+1)+2)`. Vale notar que em C matrizes são guardadas "por linha", ao contrário de FORTRAN que guarda matrizes "por coluna". Ou seja, a ordem dos elementos é `mat[0][0] mat[0][1] mat[0][2] mat[0][3] mat[1][0] mat[1][1]`, etc..

A atribuição `pf= mat[1];` está correta, pois o primeiro elemento do array `mat[1]` é um flutuante, exatamente o tipo que `pf` deve apontar. Todavia a atribuição `pf=mat` não tem nexo, pois o primeiro elemento de `mat` é um array de 4 flutuantes.

De acordo com a declaração de paf, (*paf) é um array de 4 flutuantes, logo paf deve apontar um array de 4 flutuantes. Portanto o ponteiro paf foi declarado corretamente para a atribuição paf=mat;. Já de acordo com a declaração de apf, um elemento de (apf[]) aponta um flutuante, logo apf[] é um array de 4 ponteiros para flutuantes. Consulte a tabela de precedência e associatividade de operadores em caso de dúvida.

Uma maneira alternativa, e freqüentemente mais legível, de fazer declarações de tipos derivados dos fundamentais é definir tipos derivados com a palavra reservada typedef. A sintaxe da definição de tipo é similar a sintaxe de declarações, com o nome do tipo sendo definido no lugar do nome de uma variável (do mesmo tipo) sendo declarada. No exemplo abaixo o tipo POSICAO é definido como array de 3 flutuantes, e o tipo PPOS é definido como apontador de posições; as variáveis x e y são declaradas do tipo POSICAO, e as variáveis px e py são declaradas do tipo PPOS.

```
typedef float POSICAO[3];
typedef POSICAO *PPOS;
POSICAO x, y;  PPOS px, py;
```

Enumeração

Em C é possível representar índices de uma matriz por nomes do tipo *enumeração*:

```
enum dia {seg = 2, ter, qua, qui, sex, sab, dom};
```

e então ter fica sendo sinônimo de 3. Ao declararmos:

```
enum dia hoje; float gasto[7];
```

pode-se atribuir à variável hoje = qua; ou então compararmos

```
if(hoje == sab) gasto[dom] = gasto[hoje];.
```

Este tipo de declaração torna o programa mais fácil de ser compreendido, e portanto é recomendável.

10 Inicializações e Strings

Variáveis podem ser *inicializadas* ao serem declaradas. A declaração int i=9948; equivale a declarar i como uma variável inteira, e logo em seguida atribuir-lhe o valor 9948. Também arrays podem ser inicializados ao serem declarados. Assim a primeira das declarações abaixo declara e inicializa o array a. Se omitirmos neste tipo de declaração o número de elementos do array, o compilador presuporá que o número de elementos do array é o número de elementos na lista de inicialização. Assim a segunda declaração é equivalente a primeira. A quinta declaração inicializa um array de arrays. Dando o número de elementos destes arrays podemos omitir as chaves internas, como na sexta declaração que é equivalente à anterior.

O código binário de um caractere é obtido colocando este caractere entre aspas simples. Assim 'J' é o código da letra J-maiúsculo. Existem vários sistemas de codificação, como ASCII, EBCDIC ou ABICOMP, e qual o particular sistema adotado depende do compilador. O código é um valor inteiro. Assim um caractere pode ser usado em expressões aritméticas como 'J'+1, 'b'-'a', etc..

Considerando as declarações char c; int i;, a função atoi(c) vale o código do caractere c, e a função itoa(i) é o caractere correspondente ao código i.

Vários caracteres especiais são representados com auxílio do caractere contra-barra, \, como nova-linha \n, retrocesso \b, contrabarra \\, aspa-simples \', aspa-dupla \", alarme

\a, etc. *Strings* são arrays de caracteres cujo término é marcado pelo caractere nulo, \0. Uma forma alternativa de inicializar strings é trocar a lista de códigos de caracteres pela seqüência dos caracteres entre aspas duplas, sem incluir explicitamente o terminador `\0`, como na quarta declaração abaixo, que é equivalente à terceira.

```
int a[5] = {1, 2, 3, 4, 5 };
int a[] = {1, 2, 3, 4, 5, };
char nome[] = {'D', 'i', 'a', '-', 'D', '\0' };
char nome[] = "Dia-D";
int ident2[][] = {{1,0},{0,1}};
int ident2[2][2] = {1,0,0,1};
```

A leitura de um string pode ser efetuada através da especificação `%s` como em `scanf("%s", nome);`. Note que neste caso `nome` *não* é precedido por `&`.

A impressão ou gravação de um string pode ser feita também por `%s`, como em `printf("%s", nome);`.

A seguir um programa para ilustrar uso de strings. O programa lê um `nome` de até 80 letras e conta quantas letras `a` nele ocorrem. Note que o caractere `\0` é gerado pelo toque da tecla ENTER, podendo ocorrer antes de completar 80 letras.

```
main(){
  char nome[80];
  int i, nletras_a;
  printf("Digitar um nome e teclar ENTER (max de 80 letras)");
  scanf("%s", nome);
  i=0; nletras_a=0;
  while(nome[i]!='\0'){
    if(nome[i] == 'a') nletras_a++;
    i++;
    }
  printf("%d letras `a' ocorrem no nome", nletras_a);
  }
```

11 Regras de Escopo. Bloco

```
main(){                  /* aqui comeca o 1-o bloco */
  int i=1, j=2, k=3;     /* variaveis internas do 1-o bloco */
  {                      /* aqui comeca o 2-o bloco */
    int i, j=5, m;       /* variaveis internas do 2-o bloco */
    i=5; j++; m=i+j;
    printf("%d %d %d %d",i,j,k,m);  /* 5 6 3 11 */
  }                      /* aqui termina o 2-o bloco */
  printf("%d %d %d %d",i,j,k,m); /* 1 2 3 erro */
}                        /* aqui termina o 1-o bloco */
```

C é uma linguagem estruturada em blocos. Isto significa que podemos definir variáveis no início de qualquer bloco, i.e. logo após a chave que abre um comando composto. Variáveis definidas dentro de um bloco são ditas *internas* a este bloco. Variáveis internas são *locais*, e, via de regra, só existem dentro do bloco onde foram definidas. Em caso de *conflito* , i.e. se o nome de uma variável local coincide com o nome de uma variável definida fora do bloco, este nome se refere, dentro do bloco em questão, à variável local.

Assim, no exemplo acima, a primeira e a segunda série de comandos `printf()` produzem resultados diferentes. O `erro` significa que fora do bloco mais interno não existe nenhuma variável `m`. O *escopo* de uma variável é a região do programa onde a ela podemos nos referir. Assim o escopo da variável `m` é o 2-o bloco, o escopo da variável `k` é todo o 1-o bloco, o escopo das variáveis `i` e `j` definidas dentro do 2-o bloco é o 2-o bloco, e o escopo das variáveis `i` e `j` definidas no 1-o bloco é a região do 1-o bloco fora do 2-o bloco.

12 Funções. Parâmetros e Argumentos

```
float potencia( float x, int n);  /* declaracao da funcao */

main(){
 int i; float y, z;
 i=3; y=2.0;
 /* nesta linha tem-se uma invocacao da funcao potencia */
 z = potencia(y,i);
 printf("%f", z);
}  /* programa principal */

float potencia( float x, int n ){
 int j; float w;
 if(n<0)
   return 0.0;
 else
   w = 1.0;
   for(i=1; i<=n; i++)
     w = w*x;
   return w;
} /* definicao da funcao */
```

Em todas as modernas linguagens de programação há maneiras de evitar repetir um longo trecho de programa que gostaríamos de reutilizar muitas vezes; estas maneiras (subrotinas, procedimentos, etc.) se denominam em *C funções*.

No exemplo acima definimos a função `potencia`, que eleva um flutuante `x` a uma potência inteira `n`. O *corpo* da função é o comando composto que segue a sua lista de parâmetros:

`nome-da-funcao(lista-de-parametros){corpo-da-funcao}`.

O tipo de uma função é, por definição, o tipo do valor que ela *retorna*, ou seja, do valor da função. Este tipo tem de ser declarado antes do nome da função. Funções que não retornam valor algum são declaradas do tipo `void`.

Após o nome da função segue-se, entre parênteses, a lista dos *parâmetros* da função. Parâmetros são variáveis internas ao corpo da função. O tipo de cada parâmetro também deve ser declarado, para que possamos detectar incompatibilidades caso invoquemos a função equivocadamente. Mesmo que a lista de parâmetros seja vazia, não podemos dispensar, ao invocar a função, um par de parênteses para indicar a lista vazia, como em `x=f();`. Ao declarar ou definir uma função sem parâmetros devemos declarar explicitamente que esta não toma parâmetros, como em `float random(void);`. Ao invocarmos a função, com dados *argumentos*, o valor destes argumentos é copiado nos respectivos parâmetros da função. Como o escopo dos parâmetros é o interior do corpo da função, o que quer que aconteça no corpo da função não pode afetar o argumento com que invocamos a função. Este mecanismo chama-se *passagem por valor*. Se, dentro do corpo da função, uma referência a um parâmetro fosse entendida como uma referência ao argumento com que invocamos a função, isto seria uma *passagem por referência*. Em C é fácil simular passagens por referência passando para a função apontadores para variáveis, ao invés das próprias variáveis.

Exemplo:

```
 void trocanao( float x, float y);
 void trocasim( float *px, float *py);
 main(){
   float x=1.0, y=2.0;
   printf("%f %f",x,y);      /* 1.0 2.0 */
   trocanao(x,y);            /* 2.0 1.0 */
   printf("%f %f",x,y);      /* 1.0 2.0 */
   trocasim(&x, &y);         /* 2.0 1.0 */
   printf("%f %f",x,y);      /* 2.0 1.0 */
 }

void trocanao(float x, float y){
  float aux;
  aux=x; x=y; y=aux;
  printf("%f %f",x,y);
}

void trocasim(float *px, float *py)
  float aux;
  aux=*px; *px=*py; *py=aux;
  printf("%f %f",*px,*py);
}
```

Como no nosso exemplo o ponto de invocação da função precede a sua definição é conveniente declarar a função antes deste ponto. A forma desta declaração é um *protótipo* da função, que é essencialmente uma cópia do cabeçalho (header) da função. As regras que definem o escopo de uma função, i.e. onde podemos a ela nos referir, são as mesmas regras que definem o escopo de variáveis, como descritas nas Seções 11 e 15.

O comando `return` especifica o valor a ser retornado pela função, e também retorna o fluxo de execução do programa ao ponto de invocação da função. Caso não tenha sido encontrado um comando `return`, o fluxo de execução também retorna ao ponto de invocação ao chegarmos ao fim do corpo da função, i.e. à chave que fecha o corpo da função.

Array

Quando um argumento de uma função é um array este é passado por referência, i.e. não copiamos o (array) argumento num (array) parâmetro, mas simplesmente passamos o nome do array, que aponta o primeiro elemento do argumento. Como a função não tem que alocar memória para uma copia do array, não é necessário explicitar o número de elementos do array na declaração de parâmetros, basta explicitar o tipo de apontador a ser passado. Seguem alguns exemplos de protótipos equivalentes de `faf()`, uma função que toma por argumento um array de flutuantes, e `fmat()`, uma função que toma por argumento uma matriz com um número arbitrário de linhas e 4 colunas, i.e. uma função para a qual passamos um apontador de arrays de 4 flutuantes. Em seguida temos exemplos de invocações equivalentes destas funções.

```
void faf( float af[] );
void faf( float *pf );
void fmat( float mat[2][4] );
void fmat( float mat[][4] );
void fmat( float (*paf)[4] );

   faf( af );
   faf( &af[0] );
   fmat( mat );
   fmat( &mat[0] );
```

13 Funções de Entrada e Saída

Muitas tarefas que em outras linguagens são executadas por comandos da própria linguagem, são em *C* executadas invocando funções de uma *biblioteca padrão*. Descrevemos a seguir algumas funções para ler e escrever arquivos, cujos protótipos estão no arquivo `<stdio.h>`.

Abrir e fechar arquivo. Descritor

Todo arquivo em meio magnético deve ser aberto, antes de ser lido ou gravado, e depois ser fechado. Todas as informações que o compilador necessita sobre um arquivo estarão no descritor do arquivo, numa variável do tipo `FILE`. O descritor é criado e inicializado ao *abrirmos* um arquivo, e este descritor continua a disposição das funções de entrada e saída até que o arquivo seja novamente *fechado*. Toda referência a um arquivo será feita através de um apontador para seu descritor. A forma do descritor, i.e. o tipo `FILE`, varia conforme o compilador, o sistema operacional, o computador, o dispositivo físico de entrada e saída, etc..

Corrente – "Stream"

Um apontador de arquivo, ou mais exatamente um apontador para o descritor de um arquivo, é denominado a *corrente* (stream) associada a este arquivo. Usaremos correntes como uma abstração da noção de arquivo, ou da noção de dispositivo genérico de entrada e saída. O conteúdo de uma *corrente de texto* é uma seqüência de caracteres, incluindo possivelmente caracteres de formatação. O conteúdo de uma *corrente binária* é a transcrição exata de uma seqüência de posições de memória. Toda corrente aberta tem um

Table 5: Modos de Abertura de Arquivos Texto

`"r"`	abre um arquivo para leitura ao seu início
`"w"`	cria um novo arquivo para gravação ao seu início
`"a"`	abre um arquivo para gravação ao seu final

indicador de posição, que indica a próxima posição da corrente a ser lida ou gravada. O tipo do indicador de posição também depende da implementação, mas geralmente é `long int`.

Vejamos a seguir as formas genéricas, e exemplos de uso, de algumas funções da biblioteca padrão de entrada e saída:

```
corrente = fopen("nome-do-arquivo", "modo");
fclose(corrente);
fprintf(corrente, "cadeia-de-controle", lista-de-valores);
fscanf(corrente, "cadeia-de-controle", lista-de-enderecos);
feof(corrente);

FILE *lixoentra, *lixosai; long int posicao; int i; float x;
lixoentra = fopen("dados.dat", "r");
lixosai = fopen("resultds.dat", "w");
fscanf(lixoentra, " %d %f", &i, &x);
fprintf(lixosai, "Em %d\n prevemos %f %% de inflacao\n", i, x);
do{ fscan(lixoentra,"%f ", &x); papalixo(x);
}while(!feof(lixoentra););
```

A função `fopen(...)` abre um arquivo, associa a ele uma corrente, e retorna a corrente a ele associada. Os argumentos de `fopen(...)` são duas cadeias: A primeira contém o nome do arquivo; A segunda contém o modo de abertura, que estabelece o tipo de operações de entrada e saída a serem realizadas. Alguns modos de abertura de arquivos texto são listados na Tabela 5.

A função `fclose(...)`, fecha a corrente que toma por argumento. Toda corrente aberta num programa, tem de ser fechada antes do término do programa, sob pena de perda do conteúdo da corrente. A função `feof(...)` retorna o valor lógico `TRUE` (i.e., verdadeiro) se o indicador de posição, da corrente que toma por argumento, indica o fim da corrente, e o valor `FALSE` (i.e., falso) caso contrário.

A função `fprintf(...)` escreve em correntes de texto. Seu primeiro argumento é a corrente, o segundo argumento é uma cadeia de controle, e os argumentos seguintes são valores a serem tomados e interpretados segundo estabelecido na cadeia de controle. A cadeia de controle contém caracteres a serem copiados na corrente, e especificadores de formato. Especificadores de formato são formados pelo caractere `%` seguido de um código de conversão. Cada especificador de formato na cadeia de controle indica que o próximo argumento na lista de valores será gravado na corrente, após tradução para uma forma de texto apropriada. Entre o caractere `%` e o código de conversão, podemos incluir argumentos opcionais que modificam o código de conversão ou a forma de gravação na corrente. Alguns exemplos de especificadores de formato podem ser vistos na Tabela 6.

Table 6: Especificadores de Formato

`%d`	Inteiro decimal
`%ld`	Inteiro longo decimal
`%f`	Flutuante em precisão simples ou dupla
`%E`	Flutuante em notação cientifica
`%12.6f`	Flutuante num campo de no minimo 12 caracteres com 6 digitos apos o ponto decimal.
`%%`	O caractere %

A função `fscanf()` lê correntes de texto. Caracteres na cadeia de controle forçam a função a ler e descartar caracteres correspondentes na corrente. Um espaço em branco na cadeia de controle força a função a ler e descartar brancos até o próximo caractere não branco na corrente. Lembre-se, brancos são os caracteres espaço-em-branco, marca-de-tabulação e nova-linha. Cada especificador de formato na cadeia de controle indica que o próximo argumento na lista de endereços receberá um valor lido como texto na corrente.

A função `feof()` verifica se o indicador de posição indica o fim do arquivo (i.e., end-of-file). Muitas outras funções de entrada e saída estão a disposição na biblioteca padrão. Entre as que lhe poderão ser úteis no futuro destacamos `getc()`, `putc()`, `fread()`, `fwrite()`, `fseek()` e `ftell()`. Consulte o manual do seu compilador.

14 Conversões e Alocação Dinâmica

Muitas vezes queremos converter um objeto de um tipo para outro tipo. Isto é feito através do operador de conversão (cast), como em:

`int i=1; float x=1.0; i=(int)x; x=(float)j;.`

Regras específicas determinam como estas conversões são feitas. Por exemplo, um objeto do tipo `float` é convertido num objeto do tipo `int` por *truncamento*. Quando aplicamos alguns operadores, como os operadores aritméticos, de atribuição e de comparação , algumas conversões são feitas automaticamente caso seja necessário. Estas conversões automáticas sempre "promovem", numa hierarquia de tipos aritméticos, o operando de tipo mais baixo na hierarquia para o tipo do operando de tipo mais alto. Esta hierarquia de tipos estabelece que, usando `>` para indicar mais alto que,

`long double > double > float > long int > int.`

Assim, os dois programas abaixo atribuem o mesmo valor à variável `d`.

```
double d;               *       double d; float f, g, tempf;
float f, g;             *       long int l, templ; int i;
long int l;             *
int i;                  *       templ = l * (long int)i;
                        *       tempf = f * g;
d = f*g + l*i;          *       tempf = tempf + (float)templ;
                        *       d = (double)tempf;
```

Ponteiros podem ser convertidos de um tipo para outro pelo operador de conversão. A sintaxe do especificador de tipo para o operador de conversão segue a mesma sintaxe de declarações, exceto que omitimos o nome de um objeto a declarar. Assim `pf = (float *) p` converte um apontador `p` para um apontador para flutuantes, e `paf = (float (*)[4]) p` converte um apontador `p` para um apontador de arrays de 4 flutuantes.

Um ponteiro genérico aponta uma posição de memória, ou aponta objetos que ocupam uma única posição de memória. O tipo de um ponteiro genérico é `(void *)`. Existe também um endereço inválido padrão, este endereço é `NULL`. Podemos, por exemplo inicializar um ponteiro `p` com o valor `NULL`, e mais adiante no programa verificar, com a condição `(p!=NULL)`, se já foi atribuído ao `p` um endereço válido.

A biblioteca padrão tem funções que alocam e desalocam memória dinamicamente: `void *malloc(int numero);` aloca um bloco de `numero` posições de memória consecutivas, e retorna um ponteiro para a primeira destas posições. `void *calloc(int numero, int tamanho);` aloca `numero*tamanho` posições, i.e. memória para guardar `numero` objetos que ocupem `tamanho` posições cada. Ao contrario de `malloc()`, `calloc()` zera as posições alocadas. `void free(void *p);` libera um bloco previamente alocado por `p=malloc(numero)` ou `p=calloc(numero,tamanho)`.

15 Classes de Armazenamento

Todas as variáveis que vimos até agora foram declaradas dentro de algum bloco, i.e. foram variáveis *internas*. A classe de armazenamento padrão de variáveis internas é `auto` (automática). Variáveis internas são locais, i.e. seu escopo se limita ao interior do bloco onde foram declaradas. Variáveis *internas automáticas* deixam de existir quando o fluxo de execução do programa sai deste bloco, i.e. a memória alocada no início do bloco para guardar estas variáveis é liberada ao sairmos do bloco. Se por acaso o fluxo reentrar no bloco, as variáveis serão novamente criadas. Este mecanismo tem a vantagem de economizar memória.

Podemos também declarar variáveis internas da classe `static`. Variáveis *internas estáticas* têm o mesmo escopo que outras variáveis internas, i.e. só podemos nos referir a elas dentro do bloco onde foram declaradas, mas a memória reservada pelo compilador para guardá-las não é liberada ao sairmos do bloco. Se por acaso reentrarmos no bloco, i.e. voltarmos ao escopo da variável, o último valor da variável estará preservado. Exemplo:

```
float acumule(float x, int reset);

main(){
 float v[]={1.0, 2.0, 3.0}, x;  int i;
 acumule(0.0, 1);       /* reset total=0.0 */
 for(i=0; i<4; i++)
   acumule(v[i], 0);    /* ignoramos o valor retornado */
 x=acumule(0.0, 0);
 printf("%f",x);  /* no exemplo, 6.0 */
}

float acumule(float x, int reset){
  static float total;
  if(reset==1)
    total=x;
```

```
  else
    total = total+x;
  return total;
}
```

Podemos ainda declarar variáveis fora de qualquer bloco. Estas são ditas variáveis *externas*. Variáveis externas são, via de regra, *globais* i.e. podemos nos referir a elas de qualquer lugar, desde que já tenham sido declaradas e que não conflitem com um nome interno. Variáveis externas devem ser definidas uma única vez. Para acessar uma variável externa que foi definida em outro arquivo é necessário preceder a declaração da variável com a palavra reservada `extern`. Se quisermos definir variáveis externas que não sejam acessíveis a nenhum outro arquivo, devemos preceder sua definição com a palavra reservada `static`. Não haverá conflito entre o nome de uma variável *externa estática* e o nome de outras variáveis externas. Exemplo:

```
***************************************************************
*       arquivo f.c            *     arquivo globichos.h      *
***************************************************************
*                              *                              *
*  int i, j;                   *  extern int i, j, k;         *
*  int f(int l, int m){        *  extern int f(int l, int m); *
*   /* definicao de f */ }     *                              *
*                              *                              *
***************************************************************
*       arquivo g.c            *     arquivo principal.c      *
***************************************************************
*                              *                              *
*  static int i;               *  #include "globichos.h"      *
*  static float f(float arg){} *                              *
*  /* define i e f protegidas  *  main(){                     *
*    para uso privado em g */  *  int l, m, n;  float x, y;   *
*                              *                              *
*  float g(float arg){}        *   i++;  x=g(y);              *
*   /* definicao de g */       *   k= f(i,l);       ...   }   *
*                              *                              *
***************************************************************
```

16 Estruturas

```
struct pixel{                /*Definicao da estrutura pixel*/
  float horizontal; float vertical;      /*Lista de membros*/
  int cor; int saturacao; int intensidade;
} ;

struct pixel foco1, foco2, elipse[100],
             *ppix;
     /* Declaracao de variaveis do tipo pixel */
```

Estruturas permitem referências a um conjunto de variáveis de tipos *distintos*, de forma *agregada*. No exemplo acima, primeiro definimos o modelo da estrutura `pixel`. Depois declaramos quatro objetos do tipo `struct pixel`, as variáveis `foco1` e `foco2`; `elipse`, um array de 100 pixels; e `ppix`, um apontador de pixels. Note que a especificação de tipo é feita pela palavra reservada `struct` seguida da *etiqueta* (tag) da estrutura.

Para acessar os membros de uma variável de tipo estrutural, usamos o operador *membro de*, `.` (ponto). Assim `foco1.cor` é o membro `cor` da variável `foco1`. Note que `foco1` é uma variável do tipo `struct pixel`, enquanto `foco1.cor` é uma variável inteira. Desta forma são válidas as atribuições e referências:

```
int i, j=15;
foco1.cor=9948; foco2.cor=foco1.cor; i=foco2.cor;
elipse[j].cor=8831;  ppix=&elipse[0];
(*ppix).cor=8831; ppix->cor=8831;
elipse[0].cor=8831;  ppix=elipse;
(*(ppix+j)).cor=8831;  (ppix+j)->cor=8831;
```

As expressões usando o operador `->` são apenas sinônimos das expressões que as precedem, usando os operadores de dereferência e membro-de, i.e. `a->b` é só uma abreviação de `(*a).b`.

Os exemplos seguintes ilustram várias formas alternativas para declaração e definição de estruturas:

```
struct pixel{ /* mesma lista */ } foco1, foco2, elipse[100], ppix;

typedef struct pixel{ /* mesma lista */ } Pixel;

typedef struct { /* mesma lista */ } Pixel;

Pixel foco1, foco2, elipse[100], ppix;
```

- Podemos definir uma estrutura e declarar variáveis deste mesmo tipo estrutural num único comando, listando as variáveis declaradas entre o corpo da definição da estrutura e o ponto-e-vírgula que termina o comando.

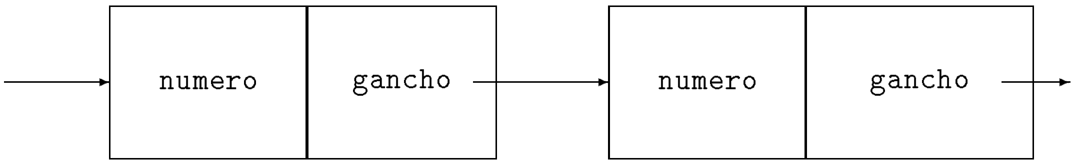
- Se a lista de variáveis declaradas na definição de uma estrutura não é vazia, a etiqueta é opcional.

- Podemos usar `typedef` para criar um especificador de tipo que prescinda da palavra `struct`.

Não haverá conflito entre etiquetas ou nomes de membros com nomes de variáveis ordinárias, sendo a distinção feita pelo contexto. Exemplo:

```
struct confusao{ int caos };
struct confusao confusao;
int i, j, caos=1;
confusao.caos=caos;
i=++confusao.caos; j=--caos;
```

Figure 10: Ilustração de objetos do tipo `elo`



17 Estruturas Encadeadas

Muitas vezes é útil encadear objetos que se referem mutuamente. Segue um exemplo de definição e uso de uma lista encadeada ou ligada para representar conjuntos. O programa "lê" um conjunto de números de um arquivo, e a seguir verifica se um dado número pertence ao conjunto.

A figura abaixo ilustra o tipo de estrutura de dado chamado *elo* definido no programa a seguir.

Para uma ilustração da estrutura de dados com objetos do tipo `elo`, veja a Figura 10.

```
#include <stdio.h>
#define PNOVOELO  (Elo *) malloc(sizeof(Elo))

typedef struct elo{
  int numero;
  struct elo *gancho; /* ponteiro para proximo elemento */
    }Elo;
Elo *leconjunto(FILE *pif); /* funcao */
int pertinencia( int numero, Elo *conjunto); /* outra funcao */

void main(){
  FILE *pif, *pof;
  Elo *conjunto1;  /* ponteiro para conjunto representado por */
                   /* uma lista ligada  */
  int pertence, numero={5};
  pif=fopen("input","r");      /* arquivo de entrada chamado input */
  conjunto1= leconjunto(pif);
  pertence= pertinencia(numero, conjunto1);
  close(pif);
  if(pertence==1)
    printf("%d pertence ao conjunto\n", numero);
  else
  printf("%d nao pertence ao conjunto\n", numero);
}/* main */

Elo *leconjunto(FILE *pif ){  /* le conjunto de numeros do arquivo */
  Elo *comeco, *pproximo, *ppultimo; int numero;
```

```
  int cont=0;
  comeco=pproximo=PNOVOELO; /* ponteiro para proximo elemento */
  fscanf(pif, " ");
  while(!feof(pif)){
    fscanf(pif, " %d ", &numero);
    pproximo->numero=numero;
    ppultimo=pproximo;
    pproximo=pproximo->gancho=PNOVOELO; /* ponteiro para proximo
                                                     elemento */
    cont++;
  }
  if(cont>0)
    ppultimo->gancho=NULL;
  else
    comeco=NULL;
  free(pproximo);
  printf("conjunto de %d elementos\n",cont);
  return comeco; /* ponteiro para inicio de lista ligada */
}/* leconjunto */


int pertinencia( int numero, Elo *conjunto){
  if(conjunto == NULL)
    return 0;
  while( conjunto->gancho != NULL && conjunto->numero != numero)
    conjunto=conjunto->gancho;
  if( conjunto->numero == numero)
    return 1;
  else
    return 0;
}/* pertinencia */
```

Na definição de `Elo` listamos, como um de seus membros, um apontador de objetos do tipo `Elo`. Este tipo de referência recursiva na definição de estruturas não apresenta maiores problemas, pois o tamanho de um ponteiro independe do que ele aponta, sendo apenas um endereço de memória. Já ter um objeto do tipo `Elo`, como membro de `Elo`, não seria possível (por quê?).

18 Erros Mais Comuns em C

Listamos abaixo os erros mais comuns que os programadores iniciantes em C costumam cometer. Correspondendo a cada erro, recomendamos uma forma de evitá-lo.

1. Ponteiro deve ser inicializado antes de ser usado. No exemplo abaixo, o ponteiro `pi` possui um valor "lixo", digamos, `x`. E é no endereço `x` que o valor `51` é armazenado,

destruindo o valor anterior que pode ser um reservado para uma outra variável ou mesmo um código executável, dependendo do valor de x.

```
main(){
  int *pi;
  *pi = 51;
  printf(" %d", *pi);
}
```

2. Confundindo o operador = de atribuição com == de igualdade. O operador = como em a = b; significa "atribuir o valor de b para a", enquanto == como em if(a==b) significa "se a for igual a b". O trecho de programa:

   ```
   if(a=b) printf("Igual");
     else printf("Desigual");
   ```

   provoca a mensagem "Igual" só após a atribuição do valor de b a a, se b não for nulo.

3. Strings podem ser declarados como array do tipo char ou como ponteiro para char, mas os dois casos tem uma grande diferença que ilustraremos com o exemplo abaixo. A declaração de meunome reserva 20 posições de memória, mas a declaração de pnome não.

   ```
   main(){
     char meunome[20];
     char *pnome;
     scanf("%s", pnome);   /* erro 1 */
     meunome= "Fernandinho";          /* erro 2 */
     printf("%s %s", pnome, meunome);
     }
   ```

   A funo scanf é aceita pelo compilador, mas nenhuma posição de memória foi reservada a partir do endereço apontado por pnome. Provavelmente, as letras digitadas serão armazenadas em posições alocadas para outras variáveis. Entretanto, o comando scanf("%s", meunome), está correto.

   O segundo erro é no comando seguinte, pois o string "Fernandinho" possui um endereço mas ele não pode ser atribuído a meunome, mas o endereço associado a meunome não pode ser alterado. Por outro lado, o comando pnome = "Fernandinho" é correto.

   A forma abaixo também está correta:

   ```
   main(){
     char meunome[20];
     char *pnome;
     pnome = (char *) malloc(20); /* reserva de 20 posicoes */
     scanf("%s", pnome);
     strcpy(meunome, "Fernandinho");   /* funcao especial para copiar */
     printf("%s %s", pnome, meunome);
     }
   ```

4. Arrays como `int melis[50]` tem o primeiro elemento indexado por `0` e não por `1`, e o último elemento por `49`. No exemplo abaixo o primeiro elemento não é alterado, além de o elemento `melis[50]` não ser aceitável:

   ```
   for(i=1; i<=50; i++)   /* erro: i=0;i<50 seria correto */
     melis[i] = i*i;
   ```

   Recomendamos que o valor so índice *sempre* seja verificado como em `if(i>=0 && i<50) melis[i]=i*i;`.

5. Se `&` na funo `scanf` for esquecido, como em:

   ```
   main(){
     int m,n,prod;
     scanf("%d %d", m, n);   /* &m, &n seria correto */
     prod = m*n;
     printf("Produto de m e n: %d", prod);
   }
   ```

   os valores de `m` e `n` seriam usados como endereços em cujas posições são armazenados os valores digitados, e os valores das variáveis `m` e `n` *não* são alterados.

19 Exemplos de Programa em C

```
/*
 *  BUSCASEQ.C--Exemplo de Busca Sequencial
 *  Problema: dada uma sequencia de N numeros reais (do tipo float)
 *  e um numero real X, verificar se X ocorre na sequencia
 */
#include <stdio.h>
#define Nmax 101           /* Numero maximo de elementos em R[] */
void main()
{
   int N,   /* numero de elementos em R[] */
        i;
   float R[Nmax],  /* vetor de N elementos */
         X;        /* elemento a ser procurado em R[] */
   /*
    *  Leitura dos parametros
    */
   printf("\nDigitar o numero de elementos de R[] -> ");
   scanf("%d", &N);
   printf("\nN = %d\n", N);
   if(N>Nmax-1){
     printf("\nNumero maximo de elementos foi excedido\n");
     exit(0);
     } /*end if */
```

```
   printf("Digitar os elementos de R[] -> ");
   for(i=0; i<N; i=i+1)
     scanf("%f", &R[i]);
   printf("\n");
   for(i=0; i<N; i=i+1)
     printf("R[%d] = %f", i, R[i]);
   printf("\nDigitar o elemento X a ser procurado em R[] -> ");
   scanf("%f", &X);
   printf("\nX = %f", X);
   /*
    * Busca Sequencial de X em R[], em tempo proporcional a N
    */
   R[N] = X;   /* valor X como ''sentinela'' apos ultimo
                          elemento valido de R[] */
   i=0;
   while(R[i] != X)
     i= i+1;
   /*
    *  Dar resposta final
    */
   if( i != N )
     printf("\n--- X = %f ocorre em R[]\n", X);
   else
       printf("\n --- X = %f nao ocorre em R[]\n", X);
} /* end main */


/*
 *  BUSCABIN.C--Exemplo de Busca Binaria
 *  Problema: dada uma sequencia ordenada de N numeros reais
 *  (tipo float)
 *  em ordem crescente, e um numero real X, verificar se X ocorre na
 *  sequencia.
 */
#include <stdio.h>
#define Nmax 100           /* Numero maximo de elementos em R[] */
void main()
{
   int N,   /* numero de elementos em R[] */
        i,
        Esq, /* indice do elemento mais aa esquerda
                no intervalo de busca em R[] */
        Dir; /* idem aa direita */
   float R[Nmax],  /* vetor de N elementos ja ordenados */
         X;        /* elemento a ser procurado em R[] */

   /*
    *  Leitura dos parametros
```

```
 */
printf("\nDigitar o numero de elementos de R[] -> ");
scanf("%d", &N);
printf("\nN = %d\n", N);
if(N>Nmax){
  printf("\nNumero maximo de elementos foi excedido\n");
  exit(0);
  } /*end if */
printf("Digitar os elementos de R[] em ORDEM CRESCENTE -> ");
for(i=0; i<N; i=i+1)
  scanf("%f", &R[i]);
printf("\n");

for(i=0; i<N; i=i+1){
  if(i!=0 && R[i-1] > R[i])
    printf("\nOs elementos de R[] nao estao em ordem crescente\n");
  printf("R[%d] = %f", i, R[i]);
  } /* end for i */
printf("\nDigitar o elemento X a ser procurado em R[] -> ");
scanf("%f", &X);
printf("\nX = %f", X);
/*
 * Busca Binaria de X em R[], em tempo proporcional
 * a log N na base 2
 */
Esq= 0; Dir= N-1;
i=(Esq+Dir)/2;     /* indice do elem. do "meio"de R[] */
while(Esq <= Dir && R[i] != X){
  if(R[i]<X) Esq = i+1;
  else Dir = i-1;
  i=(Esq+Dir)/2;    /* novo indice do elem. do "meio"de R[] */
  } /* end while */

/*
 *  Dar resposta final
 */
if(R[i] == X)
  printf("\n--- X = %f ocorre em R[]\n", X);
else
    printf("\n --- X = %f nao ocorre em R[]\n", X);
} /* end main */
```

```
/*
 *  ORDEDIR.C--Exemplo de Ordenacao por Selecao Direta
 *  Problema: dada uma sequencia de N numeros reais (tipo float)
 *  ordena'-la em ordem crescente.
 */
#include <stdio.h>
#define Nmax 100            /* Numero maximo de elementos em R[] */
void main()
```

```
{
   int N,   /* numero de elementos em R[] */
        i, j,
        IndMin; /* indice do minimo temporario */
   float R[Nmax],  /* vetor de N elementos */
         temp;     /* variavel temporaria */

   /*
    *  Leitura dos parametros
    */
   printf("\nDigitar o numero de elementos de R[] -> ");
   scanf("%d", &N);
   printf("\nN = %d\n", N);
   if(N>Nmax){
     printf("\nNumero maximo de elementos foi excedido\n");
     exit(0);
     } /*end if */
   printf("Digitar os elementos de R[] a serem ordenados -> ");
   for(i=0; i<N; i=i+1)
     scanf("%f", &R[i]);
   printf("\n");
   for(i=0; i<N; i=i+1){
     printf("R[%d] = %f", i, R[i]);
     } /* end for i */

   /*
    * Ordenacao dos elementos em R[], em tempo proporcional a N*N
    */
   for(i=0; i<N; i=i+1){
     IndMin = i;         /* indice do Minimo temporario */
     for(j=i+1; j<N; j=j+1){
       if(R[IndMin] > R[j])
         IndMin = j;
       } /* end for j */
     temp = R[IndMin];
     R[IndMin] = R[i];
     R[i] = temp;
     } /* end for i */
   /*
    *  Dar resposta final
    */
   printf("\n Elementos de R[], em ordem crescente:\n");
   for(i=0; i<N; i=i+1){
     printf(" R[%d]=%f ", i, R[i]);
     } /* end for i */
} /* end main */

/*
 *  ORDEINS.C--Exemplo de Ordenacao por Insercao Direta
 *  Problema: dada uma sequencia de N numeros reais (tipo float)
```

```
 *  ordena'-la em ordem crescente.
 */
#include <stdio.h>
#define Nmax 100           /* Numero maximo de elementos em R[] */
void main()
{
   int N,   /* numero de elementos em R[] */
        i, j,
        IndMin; /* indice do minimo temporario */
   float R[Nmax],  /* vetor de N elementos */
         temp;     /* variavel temporaria */

  /*
   *  Leitura dos parametros
   */
  printf("\nDigitar o numero de elementos de R[] -> ");
  scanf("%d", &N);
  printf("\nN = %d\n", N);
  if(N>Nmax){
    printf("\nNumero maximo de elementos foi excedido\n");
    exit(0);
    } /*end if */
  printf("Digitar os elementos de R[] a serem ordenados -> ");
  for(i=0; i<N; i=i+1)
    scanf("%f", &R[i]);
  printf("\n");
  for(i=0; i<N; i=i+1){
    printf("R[%d] = %f", i, R[i]);
    } /* end for i */

   /*
    * Ordenacao dos elementos em R[], em tempo
    * proporcional a N*N
    */
   for(i=1; i<N; i=i+1){
     temp = R[i];        /* elemento a ser inserido
                            no local adequado */
     j = i-1;
     while(j>=0 && R[j]>temp){
       R[j+1] = R[j];         /* desloca elemento maior
                                 para direita */
       j = j-1;
       } /* end while j */
     R[j+1] = temp;           /* elemento temp no
                                 local adequado */
     } /* end for i */

  /*
   *  Dar resposta final
   */
  printf("\n Elementos de R[], em ordem crescente:\n");
```

```
  for(i=0; i<N; i=i+1){
    printf(" R[%d]=%f ", i, R[i]);
    } /* end for i */
} /* end main */
```

```
/*
 *  MATSIM.C--Verifica se matriz quadrada e' simetrica
 *  Problema: dada uma matriz de N por N  numeros reais (tipo float)
 *  verificar se ela e' simetrica.
 */
#include <stdio.h>
#define Nmax 100            /* Numero maximo de linhas em R[][] */
void main()
{
   int N,   /* numero de linhas em R[][] */
        i, j,
        esimetrico;
   float R[Nmax][Nmax];  /* matriz de N por N elementos */

   /*
    *  Leitura dos parametros
    */
   printf("\nDigitar o numero de linhas da matriz quadrada R[][] -> ");
   scanf("%d", &N);
   printf("\nN = %d\n", N);
   if(N>Nmax){
     printf("\nNumero maximo de linhas foi excedido\n");
     exit(0);
     } /*end if */
   printf("Digitar os elementos de R[] linha por linha -> ");
   for(i=0; i<N; i=i+1)
     for(j=0; j<N; j=j+1)
       scanf("%f", &R[i][j]);
   printf("\n");
   for(i=0; i<N; i=i+1){
     for(j=0; j<N; j=j+1)
       printf("R[%d][%d] = %f", i, j, R[i][j]);
     } /* end for i */

   /*
    * Verificacao da simetria da matriz R[][]
    */
   esimetrico = 1;          /* supondo inicialmente ser simetrico */
   i = 1;
   while(esimetrico && i<N){
     j = 0;
     while(esimetrico && j<=(i-1)){
       if(R[i][j] != R[j][i]) esimetrico = 0; /* achou assimetria */
       j = j+1;
       } /* end while j */
```

```
    i = i+1;
    } /* end while i */
  /*
   *  Dar resposta final
   */
  if(esimetrico) printf("\n Matriz R[][] e' simetrica.\n");
    else printf("\nMatriz R[][] nao e' simetrica\n");
} /* end main */
```

Produto de Matrizes: Dadas as matrizes R com $N \times M$ elementos e S com $M \times L$ elementos, obter a matriz produto T com $N \times L$ elementos, definido por:

$$\forall i : 0 \leq i \leq N-1, \forall j : 0 \leq j \leq L-1, T[i][j] = \sum_{k=0}^{M-1} R[i][k] \times S[k][j].$$

O corpo do algoritmo para calcular T é como segue:

```
for(i=0; i<N; i=i+1)
  for(j=0; j<L, j=j+1){
    temp = 0;            /* variavel temporaria */
    for(k=0; k<M; k=k+1)
      temp = temp + R[i][k]*S[k][j];
    T[i][j] = temp;
  } /* for j */
```

Bibliografia

[Baker89] L.Baker, 1989. *C Tools for Scientists and Engineers.* McGraw-Hill, New York.

[Baker91] L.Baker, 1991. *More C Tools.* McGraw-Hill, New York.

[Gonnet84] G.H.Gonnet, 1984. *Handbook of Algorithms and Data Structures.* Addison-Wesley, London.

[Kernighan88] B.W.Kernighan and D.M.Ritchie, 1988. *The C Programming Language, second edition: ANSI C.* Prentice Hall, Englewwod Cliffs.

[Kernighan] B.W.Kernighan e D.M.Ritchie, *Linguagem de Programao C.* Editora Campus.

[Koenig89] A.Koenig, 1989. *C Traps and Pitfalls.* Addison-Wesley, New York.

[Wyk89] C.J.van Wyk, 1989. *Data Structures and C Programs.* Addison Wesley, New York.

Index